CSC411: Deep Neural Networks for Handwritten Digit and Face Recognition

Yoshiki Shoji **& Zi Mo Su**

March 5, 2017

Corollary

Before we start with executing the code, we must obtain all the 60,000 images from the sets of interest, which are of training and test. We observe that there are 10 different images, each containing their own respective amounts of data. When we instantiate in concatenating all the data set images, we will also create the one-hot encoding output vectors. Hence, the first for loop will create the Data set. The second loop will create the proper outputs, with each "subsection" divided by the length of each training set image containing its proper output of interest. The code is given as follows:

```
def get_data(M):

    #we have a total 10 images
    #we first obtain the data images and concatenate vertically
    #stacking them the end result will have a height of 60,000

    Data = M["train0"]
    for i in range(1,10):
        Data = np.vstack((Data,M["train" + str(i)]))

    #repeat(x[newaxis,:], 3, 0)

    for i in range(0,10):
        if i == 0:
            y = np.zeros((1,10))
            y[0][i] = 1
            y = repeat(y[0][newaxis,:],len(M["train"+str(i)]),0)
            y_one_hot = y
        else:
            y = np.zeros((1,10))
            y[0][i] = 1
            y = repeat(y[0][newaxis,:],len(M["train"+str(i)]),0)
            y_one_hot = np.vstack((y_one_hot,y))

    return Data,y_one_hot
```

Part 1

The digits data set.

Observing Figure 1, we see that 10 different styles are shown for each digit. The data is sourced from the mnistall.mat file. Some images have heavier strokes, while others have lighter ones. In some cases, images are more stretched out than others. The code to generate this is given as follows using matplotlib and subplot:

```
plt.figure()
for i in range(0,10):
    for j in range(1,11):
        plt.subplot(10,10,j+(i*10))
        imshow(M["train" + str(i)][150+j].reshape((28,28)), cmap=cm.gray)

plt.show()
```

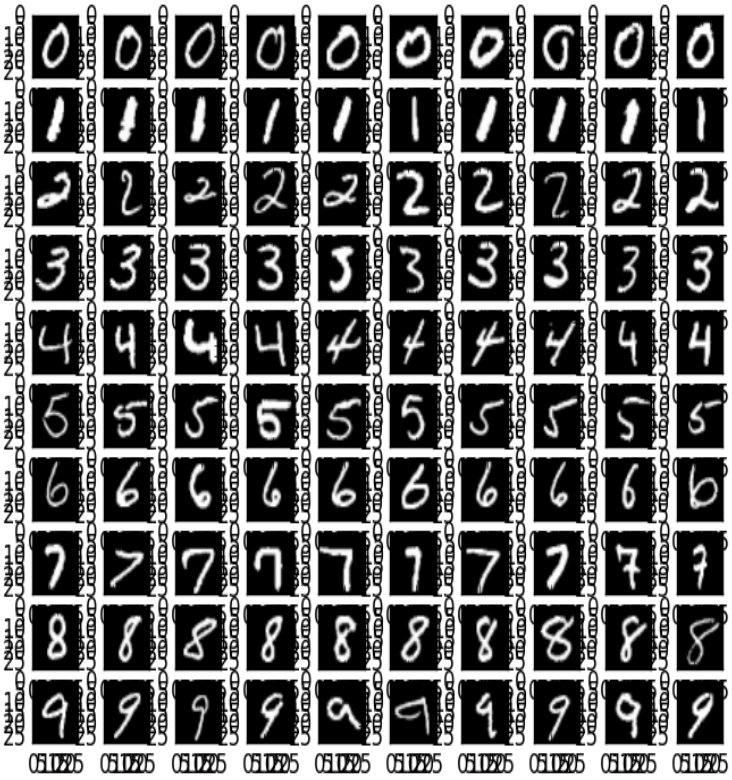


Figure 1: Ten different handwriting styles of each digit.

Part 2

Basic neural network.
We desire to calculate the output units due to the 28x28 flattened image vector, already done for us both in the training and test set. Instead of requiring a for loop to iterate, we use numpy.matmul such that each pixel element will be "dotted" by the columns of the weight matrix. The dimensions of the input will be of (Mx784), where "M" is the number of images. The weight matrix will be of size (784xN), where "N" correspeonds to the number of output units, where for our case will be of 10. However to compute softmax, we must first obtain the proper output. This function will be defined as sum_w_bias. The code is as follows:

```
    def sum_w_bias(x,initial_w,bias):
        y = np.matmul(x,initial_w) + bias
        return y

    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 outputs
        for a single case, and M is the number of cases'''
        return exp(y)/tile(sum(exp(y),0), (len(y),1))
```

Part 3

Gradient computation for basic neural network.
As we have computed the probabilities of each individual output in Part 2, we first take the log of the computed probability output matrix from the prior softmax function. We then utilize the output matrix as discussed in the Corollary. Furthermore, we also give the function of the negative-log loss function:

$$negative\ log\ loss = \sum_{i=1}^{9} y_i log(p_i)$$

From the function above, we then use the given function template and insert the desired parameters. In our case, the desired output will be denoted as get_data(M)[0] and the probability from the softmax helper function. The code is given as follows:

```
#y_ = get_data(M)[1]
#y = softmax(sum_w_bias(get_data(M)[0],initial_w,bias))
def NLL(y, y_):
    return -sum(y_*log(y))
```

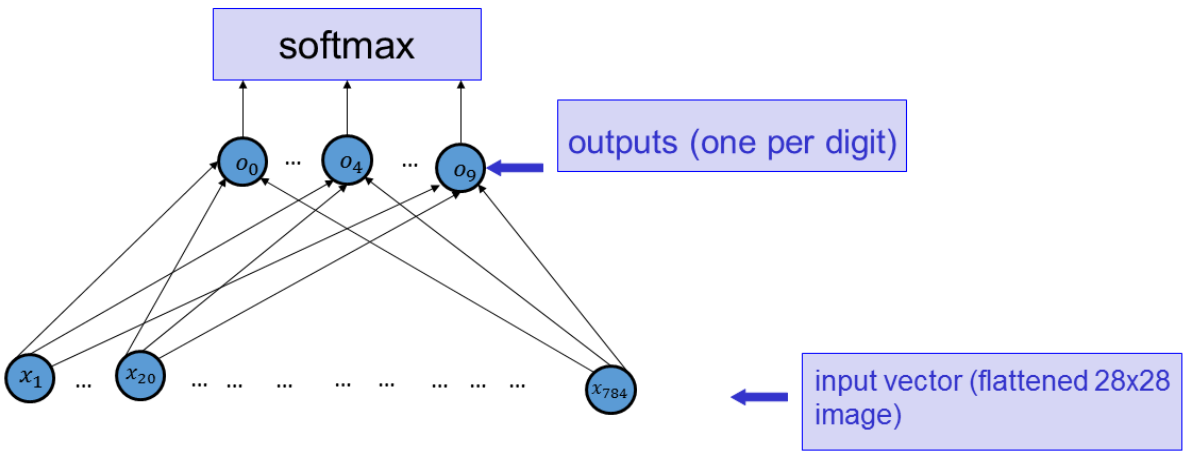


Figure 2: softmax network

Part 3a.

We consider how the output, and hence, the cost reacts due to the change in weight, w_{ij}. In mathematical form, this is given as: $\frac{\partial C}{\partial w_{ij}}$, where "i" indicates the unit and "j" indicates which unit the weight is connected to in the next layer. Applying the chain rule by observing figure 2, we obtain the following form:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial O_j}\frac{\partial O_j}{\partial w_{ij}} \ (eq1)$$

Observing (eq1), we know what the cost function is, and we also know the output. Hence, (eq1) becomes:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial \sum_{i=1}^{9} O_i log(p_i)}{\partial O_j}\frac{\partial (\sum_{k=1}^{9} w_{ik}x_i + b_{ij})}{\partial w_{ij}}$$

Observing the first partial derivative term from the chain rule, we again use chain rule and obtain $p_i - y_i$. Observing the second partial derivative term, the only surviving term is when $(i, k) = (i, j)$. From this we obtain the following final form in indical notation:

$$\frac{\partial C}{\partial w_{ik}} = x_i(p_k - y_k)$$

where i = 1,..,784 and k = 1,...,10, hence at the very end, giving us a dimension of 10x784, which must be true as each 784 individual inputs are fully connected and so we must have 10x of these as there are 10 individual outputs.

Hence, vectorizing the equation above as now we have 60,000 images for training, realizing this into code, we obtain:

```
def gradient(P, Y, X):
    # P dimension: 10x60000
    # Y dimension: 10x60000
    # X dimension: 10x784
    grad = np.dot((P - Y).T, X)
    return grad
```

As the code suggests, we use the output computed from the corollary and the softmax, along with the data we have obtained. As the gradient will be used hereafter for training due to backpropagation, we also recognize that the returned dimension from the gradient function is of 10x784, which complements the dimensions of the desired weights.

Part 3b.

We will use the finite differences of the form which takes the differences between the mid-point values. Hence mathematically it is obtained as:

$$\frac{\partial C}{\partial w} \simeq \lim_{h->0} \frac{C(P(w+h/2)) - C(P(w-h/2))}{h}$$

We use this form as it is much stronger than the backward or foward finite difference methods when dealing with unbehaved functions. Hence realizing this into code we obtain:

```
def finite_diff(W,x,y,h):
    #Param W: initial weights of 10x784. When computing
    #the gradient we fix all the values except one of them.
    #Cost is dependant on softmax which is
    #dependant on the output where we will change the weights.
    grad = np.zeros((10,784))
    for i in range(0,10):
        for j in range(0,785):
            H = np.zeros((10,784))
            H[i,j] = h
            P_plushalf = softmax(sum_w_bias(x,W+H/2,bias))
            P_minhalf = softmax(sum_w_bias(x,W-H/2,bias))
            dcdw = (NLL(P_plushalf,y) - NLL(P_minhalf,y))/h
            grad[i,j] = dcdw
    return grad
```

Observing the code, we first initialize an empty numpy array of size (10x784) which will store all the gradient values. Another numpy array of the same size, called "H" is the matrix which dynamically stores the "h" values corresponding to the index notation due to the nested for loop. Once $C(P(w+h))$ and $C(P(w-h))$ are computed, we then take the difference of these values relative to "h" which will give us the proper gradient value.

To confirm our algorithm, we shall try different values of step size, h. The conjecture must be that if "h" gets arbitrarily small, the difference in the theoretical value compared to that of finite difference must start approaching a value of zero. We reach this conclusion as the data is shown below:

h value	Number of iterations	Accumulated value of: abs(finite difference - vectorized gradient)
1	30	315.36
1e-1	30	233.23
1e-3	30	0.114
1e-8	30	0.0039

Table 1: Theoretical values vs. finite difference values

Observing Table 1, we see that our conjecture is true. The "h" value indicates the step size. The number of iterations indicate the coordinate value. In this case, we observe the first input, x_1 and evaluate the first 30 individual gradient values with respect to the weights. We further note that in obtaining the difference data, we add a layer of code under the gradient_descent function which will be given below:

```
    grad_diff = []
    p = softmax(x,y,W,bias)
    vec_grad = gradient(p,x,y)
    for i in range(0,30):
        grad_diff.append(abs(grad[0,i] - vec_grad[0,i]))

    grad_diff_accum = sum(grad_diff)
    return grad_diff_accum
```

Part 4

Training and testing the basic neural network.
To optimize the network, we apply gradient descent for which we will use the vectorized version. Furthermore, as we iterate through a while loop, we place a restriction as a breaker to not enter the realm of infinite looping, and also checking for cases when the euclidean distance from the previous and new weight are greater than some arbitrary amount we set. Realizing this into code, we obtain:

```
def grad_descent(Y, X, W, alpha):
    '''
    Executes gradient descent on the data set; computes the W matrix
    for which the cost function is minimized
    :param Y: classification matrix where each row is the one-hot encoding
     vector for an image (nx10 numpy array)
    :param X: flattened image matrix where each row is a flattened
     image (nx785 numpy array)
    :param W: weight matrix where the ith row corresponds to the weights
     for output i (10x785 numpy array)
    :param alpha: step size for gradient descent
    :return: W matrix for which cost is minimized
    '''
    EPS = 1e-6
    prev_W = W - 10*EPS
    max_iter = 3000
    iter = 0
    # for LaTeX
    train_out = ""
    test_out = ""

    X_test, Y_test = get_data('test')

    while np.linalg.norm(W-prev_W) > EPS and iter < max_iter:
        prev_W = W.copy()
        P = softmax(X, W)
        W -= alpha*gradient(P, Y, X)
        if iter % 100 == 0:
            print 'Iteration:', iter
            print 'Cost', cost(P, Y)
            train_perf = get_performance(Y, X, W)
            test_perf = get_performance(Y_test, X_test, W)
            print 'Train Performance', train_perf
            print 'Test Performance', test_perf
            train_out += str((iter, train_perf))
            test_out += str((iter, test_perf))
            print
        iter += 1

    c = cost(P, Y)
    return W, train_out, test_out
```

While performing gradient descent, the performance was tested at intervals of 50 iterations up to 2000 total iterations. The learning curve is shown in Figure 3. The test set consisted of all test digits from the data file. The training set consisted of all training digits from the data file. The code for extracting the weights from the weight matrix and saving them as images is shown below:

```
print 'running part 4...'
W, train_out, test_out = grad_descent(Y, X, W, 1e-5)

print "Train plot for LaTeX:", train_out
print "Test plot for LaTeX:", test_out

for i in range(10):
    w = W[i, 1:]
    w = w.reshape([28, 28])
    imsave("digits/digit"+str(i)+".jpg", w)

print 'done part 4'
```

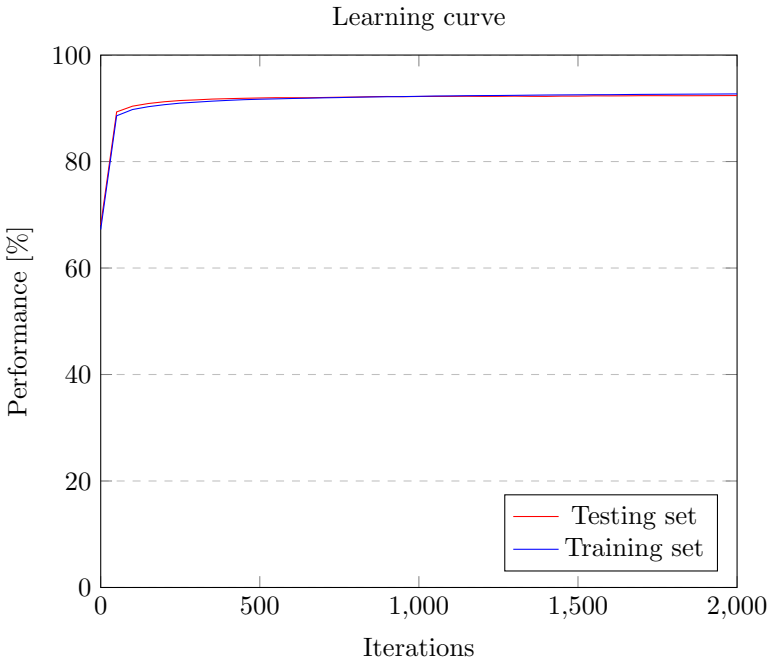
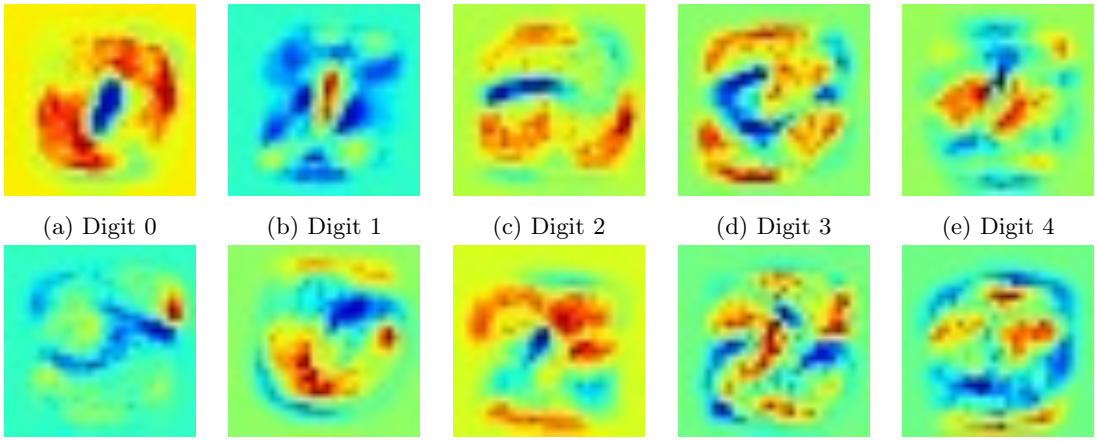


Figure 3

The weights connecting to each output are visualized in Figure 4. These weights are extracted from the end of gradient descent over 2000 iterations.



(a) Digit 0 (b) Digit 1 (c) Digit 2 (d) Digit 3 (e) Digit 4

(f) Digit 5 (g) Digit 6 (h) Digit 7 (i) Digit 8 (j) Digit 9

Figure 4: Visualizations of the weights connected to the output layer.

Part 5

Comparing multinomial logistic regression to linear regression. To show that with some data sets used for training, the performance of the test set of multinomial logistic regression is substantially better to multinomial linear regression, we will consider classification of 2 objects. To generate the data points for training, we consider the two scenarios: high and low variance utilizing the code given to us. The idea will then be to use these randomly generated data sets, and show with low variance the logistic and linear classification perform roughly the same, whereas with high variance, logistic does better consistently. We give the steps as follows:

step 1 : We generate our data set. With high variance, we must have data points scattered much more than the ones with low variance. We give two graphs below to demonstrate this and the code below:

```
def gen_data(theta, N, sigma):

    # Actual data
    x_raw = 100*(random.random((N))-.5)
    x = vstack((    ones_like(x_raw),
                    x_raw,
                    ))

    x2 = vstack((    ones_like(x_raw),
                    x_raw,
                    ))

    y1 = dot(theta, x) + scipy.stats.norm.rvs(loc = 100, scale= sigma,size=N)
    y2 = dot(theta, x2) + scipy.stats.norm.rvs(loc = -100,scale= sigma,size=N)

    plot(x[1,:], y1, "bo", label = "Training set (y = 1)")
    plot(x2[1,:], y2, "ro", label = "Training set (y = 0)")
    # Actual generating process
    #
    plot_line(theta, -100, 100, "b", "Actual generating process")


    legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
          fancybox=True, shadow=True, ncol=6)
    xlim([-100, 100])
    ylim([-200, 500])

    #Creating correct outputs. If point is above line, assign y = 1, else
    #assign y =0.

    return plt.show()
```

Observing the code above, we see two thing of importance: A decision boundary is made by the "actual" line generated, and the training points are created by generated points above and below this line using an error term due to Gaussian distribution. Anything above is classified as y=1 and y=0 for anything underneath. Observing the figure below, we see that with high variance of 100 (see fig4), some training points are within a region where they are not supposed to be. However, we treat these points as noise to mimic more real world scenarios.

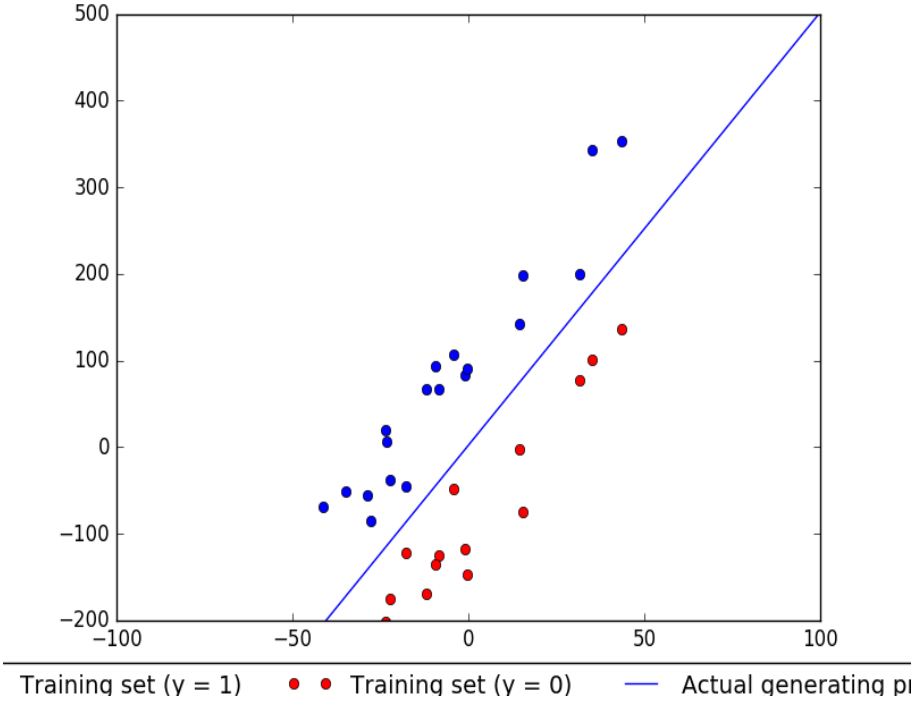


Figure 5: Training set data: Sigma = 30

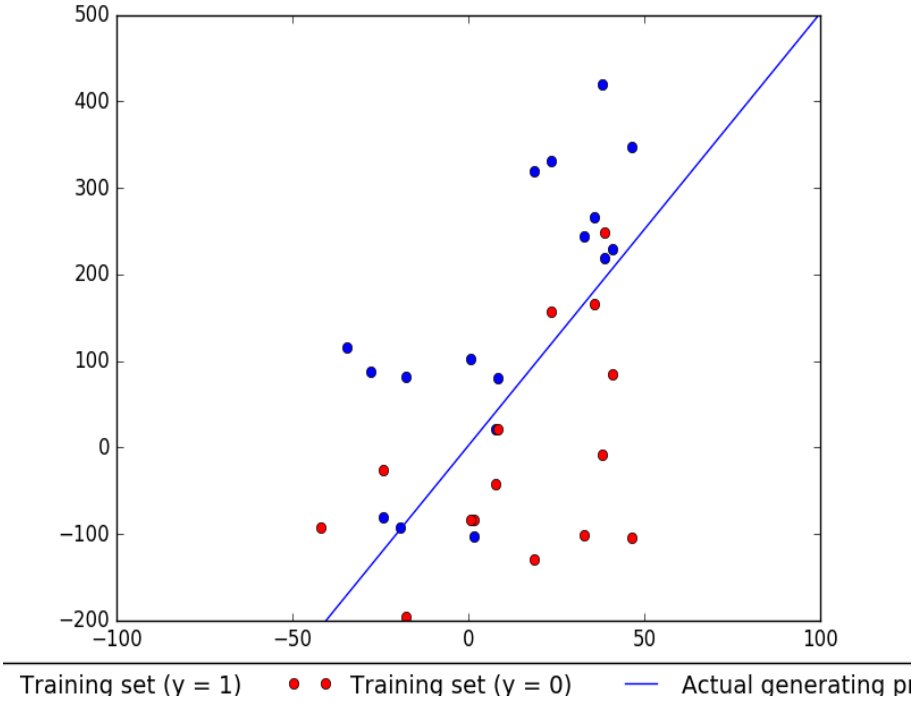


Figure 6: Training set data: Sigma = 100

Step 2 : To compare the performance of linear and logistic, we use the functions created in Part 4 and in Assignment 1 (Refer to Appendix A). As we have generated the data points, now we must construct the decision boundary from both. In order to do so, we first note the linear regression model must use the quadratic cost function due to definition. Hence, we will use the gradient descent function which is given in the Appendix as it was created in Assignment 1. As with logistic regression, this was created in Part 4 of this assignment. Furthermore, we note that the training set used in creating the actual decision boundary will be used as the test set. We make an argument as to why this is viable as follows: The training set was created by using the actual decision boundary line by using a normal distribution to get points above and below this line. These were then used in the gradient descent functions for linear and logistic regression to create the parameters required to generate their own respective decision boundary. These two parameters however, intuitively must then not be the same as they use different cost function as long as there is variance involved (more generally, with the variance increasing we expect them to start diverging from each other, but with logistic still having a good decision boundary hence giving higher performance at the end). Hence, by using the training set once again for testing, we expect for some points that were once above the actual line to be under the decision boundary's made with logistic or linear regression (although this might not be the case all the time, we still expect these events to be higher). Furthermore, at the very end there will be a graph that contains a noticeable "outlier", and show that even amongst cases like these, logistic regression will do fairly well.

Step 3 : In calculating the performance, as we use the training set again, we use the data points generated in Step 1. All that is left is to use the two decision boundary's created in step 2 for linear and logistic. If the points lie above the line, but if it happens to be underneath the actual line, then we have do not have the right classification and vice versa. We give the code below and a table of performance below:

```
def get_performance_logp5(x,y,theta):
    a = 0
    b = 0
    for i in range(0,len(x)):
        if i < (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T)) ) > 0:
                a += 1
        elif i > (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T))) < 0:
                b += 1
    performance = (float(a+b))/(len(x))
    return performance

def get_performance_linp5(x,y,theta):
    a = 0
    b = 0
    for i in range(0,len(x)):
        if i < (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T)) ) > 0:
                a += 1
        elif i > (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T))) < 0:
                b += 1

    performance = (float(a+b))/(len(x))
    return performance
```

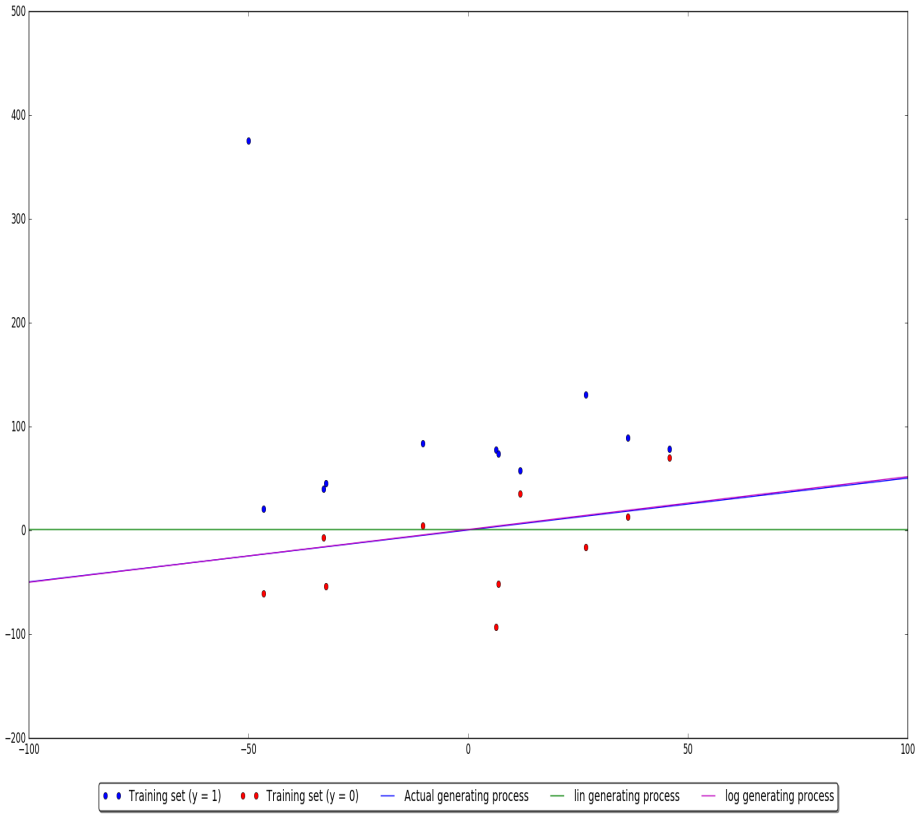


Figure 7: decision boundary created by lin and log with one crude value acting as an "outlier"

Variance value	Performance of Linear Regression (%)	Performance of Logistic Regression (%)	Difference in Performance (%)
10	78.04	82.92	4.88
30	74.19	87.09	12.9
50	54.84	80.64	25.80

Table 2: Performance of the decision boundaries of linear and logistic regression with different variances.

Observing Figure 5, we see that with linear model, if there is a value with a extremely high variance from the true decision boundary, compared with the decision boundary generated using logistic, the performance of logistic will do much better. In fact in this case, the logistic almost overlaps with the true decision boundary. Observing Table 2 below, we see that our conjecture in the performance is correct. If the variance is not too low, the performance for linear and logistic do fairly well with 4.88% difference. However, if we keep on increasing the variance, we see that the difference in the performance start increasing. However, logistic regression shows to prevail, as it gives consistent performance whereas with linear, we see that if the variance starts increasing too much, the decision boundary gets much worse and hence, the performance starts dropping.

Part 6

Obersvations using mathematics

Suppose we are working with a network that has N fully connected layers with K neurons each.
Proposition: Backpropagation compared to computing the gradient with respect to each weight individually is generally faster.

Proof:

We first generalize the network by assuming a set of activation functions. Call this set $\mathbf{F} = \{\text{Relu,Sigmoid,Tanh}\}$. In this proof, the activation function is not of importance, as we will see in the proof it is implied by the hidden layer symbol, h. Now, let us observe differentiating the Cost with respect to each individual weight:

$$\frac{\partial C}{\partial W^{(i,j,m)}}$$

where i = 1,..,N and j,m = 1,..,K and furthermore let the hidden units be denoted as $h_{i,m}$
Using chain rule, we formulate:

$$\frac{\partial C}{\partial W^{(i,j,m)}} = \frac{\partial C}{\partial h_{i,m}} \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}}$$

Once again, we apply the chain rule for the first term and obtain:

$$\frac{\partial C}{\partial W^{(i,j,m)}} = \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}} \sum_{m=1}^{K} \frac{\partial C}{\partial h_{i+1,m}} \frac{\partial h_{i+1,m}}{\partial h_{i,m}}$$

Lastly, let us apply the chain rule one last time. We obtain the following form:

$$\frac{\partial C}{\partial W^{(i,j,m)}} = \frac{\partial h_{i,m}}{\partial W^{(i,j,m)}} \left(\sum_{m=1}^{K} \left(\sum_{m=1}^{K} \frac{\partial C}{\partial h_{i+2,m}} \frac{\partial h_{i+2,m}}{\partial h_{i+1,m}} \right) \frac{\partial h_{i+1,m}}{\partial h_{i,m}} \right)$$

It then becomes evident that we keep using the chain rule observing how the cost changes due to other hidden layers until we reach the final output layer. Let us assume that each individual partial term takes a constant time. First, we observe a good upper bound if we want to determine how the cost changes due to a weight connected from the N-1 layer to an output (this would be the layer beneath the outputs). We expect for this to have O(2) as we only evaluate 2 partial terms. So then we try to observe how our cost changes due to a change in a weight in the N-2 layer, and thus, ask how the running time would change. Observing the equation above, we see two partial terms that must be computed first. The summation sign indicates with have "N" partial terms to evaluate, regardless of whether some of the terms drop out. Hence, for the N-1 layer, we expect then to have O(K+2), or more generally O(K). Hence if we keep observing changes in the cost due to weights in deeper layers, we expect a sequence of the form: O(1) $\rightarrow$ O(K) $\rightarrow ... \rightarrow O(K^N)$. The last term is developed by observation the nature of the nested summation executions. As all of our sum spans until N, the most inner sum must first have N terms. As each of these term is a variable now to undergo a summation, we then have K*K terms. As we have "N" layers, we then conclude with O(K^N).

As with matrix multiplication, we first consider a matrix with entries that contain all the cost with respect to individual weights:

$$\begin{bmatrix} \frac{\partial C}{\partial W^{(111)}} & \frac{\partial C}{\partial W^{(112)}} & \frac{\partial C}{\partial W^{(113)}} & \frac{\partial C}{\partial W^{(114)}} & \cdots & \frac{\partial C}{\partial W^{(11K)}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial W^{(1K1)}} & \frac{\partial C}{\partial W^{(1K2)}} & \frac{\partial C}{\partial W^{(1K3)}} & \frac{\partial C}{\partial W^{(1K4)}} & \cdots & \frac{\partial C}{\partial W^{(1KK)}} \end{bmatrix}$$

Observing the matrix above, we see that it is of dimension KxK. In order to establish the matrix multiplication form, we utilize what was formulated above in evaluting individual weights one by one. Hence, let us consider the matrices in a more compact form:

$$\frac{\partial C}{\partial \boldsymbol{W^T}} = \frac{\partial \boldsymbol{H_{1^{st}layer}}}{\partial \boldsymbol{W^T}} \frac{\partial \boldsymbol{H_{2^{nd}layer}}}{\partial \boldsymbol{H_{1^{st}layer}}} \cdots \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{O^T}}$$

As a corollary, let us further note that if we have two matrices, where we let $\boldsymbol{A}$ be of size nxm and let $\boldsymbol{B}$ be of size mxl, we do $nxmxl$ in total steps, and hence will obtain O(nxmxl) running time. Now, coming back to our fully connected layer, we consider first a simple example. Suppose our fully connected network has three layers, each with 2 neurons, with the first layer being the input, second layer being the hidden units, and the last layer as the output. By observing the compact matrix equation, we see that we will have:

$$\begin{bmatrix} \mathrm{x}_1 \sum_1 & \mathrm{x}_1 \sum_2 \\ \mathrm{x}_2 \sum_1 & \mathrm{x}_2 \sum_2 \end{bmatrix}$$

which is equivalent to:

$$\begin{bmatrix} \mathrm{x}_1 \\ \mathrm{x}_2 \end{bmatrix} \begin{bmatrix} \sum_1 & \sum_2 \end{bmatrix}$$

where the two sigma signs denote the sum terms due to chain rule. Hence, it becomes evident that if we keep increasing the layers, and that we have "K" neurons for each layer, we determine that the terms like the above keep on increasing, due to the amount of summations that keep on increasing from the chain rule until we hit the last term where we observe the direct change in cost due to the final output. Hence, we establish that we have $O(K^3)$ and conclude that whenever we have more than 3 layers, matrix multiplication does indeed perform much faster.

Part 7

Facial classification of actors/actresses.
To classify the actors/actresses, a system was created with four adjustable parameters. Firstly, all the images were downloaded and preprocessed. Each image URL was extracted from the source file, as well as the actor/actress' name, the cropping coordinates and the SHA256 hash code. The image was downloaded, assigned a name, tested against the hash code to confirm its correctness, converted to grayscale, cropped at the extracted coordinates, and re-sized to size 32×32.

The images were then randomly partitioned into training, test and validation sets, where the test and validation sets consisted of 30 images of each actor/actress and the training set consisted of the rest of the images.
The four variable parameters for the system are characteristics of the neural network. They are the activation function of the first layer, the activation function of the second layer (output layer), the number of hidden neurons in the first layer and the value of λ for regularization. The values used were:

i. activation function 1: $g(t) = t, tanh(t), ReLu(t)$

ii. activation function 2: $g(t) = t, tanh(t), ReLu(t)$

iii. number of hidden neurons: 30, 70, 100, 300

iv. λ: 0.00000, 0.00005, 0.00010, 0.00015

We generate a total of $3 \times 3 \times 4 \times 4 = 144$ neural networks and then trained them using minibatches consisting of 30 randomly picked images of each actor/actress from the training set at each iteration for 2000 total iterations. The weights for both layers are initially set to random values with standard deviation of 0.01 in the generation process. An alternative method for selecting the initial weight is to save the weights from a previous run of the network and initialize using this.

Each of the networks were tested using the validation set and the peak performance was recorded. The network that had the greatest accuracy for the validation set was then used as the final network.

The final network was once again executed with minibatches of size 30 images per actor/actress. This time the network was tested on the test set, validation set and entire training set. The learning curve of the network is graphed in Figure 8.

After executing the code, it was determined the best performing parameters were:

i. activation function 1: $g(t) = tanh(t)$

ii. activation function 2: $g(t) = t$

iii. number of hidden neurons: 30

iv. λ: 0.00005

with performance of 88.33% on the validation set.

After running this network and testing on all sets, the performance was 87.22% for the test set and 100.0% for the training set.

Please refer to Appendix B for this part's code.

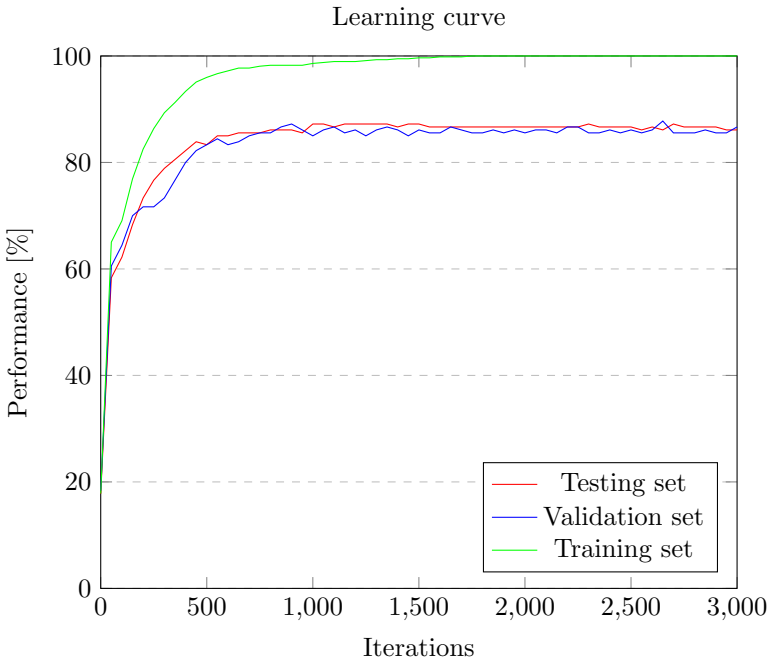

Learning curve
Performance [%]
Iterations
0
20
40
60
80
100
0
500
1,000
1,500
2,000
2,500
3,000
Testing set
Validation set
Training set

Figure 8

Part 8

Regularization for facial classification.
L2 regularization is implemented in the code. This type of regularization, of the form $\lambda \cdot w^2$, which is added to the cost, results in weights that are of lower value on average. It reduces the impact of specific neurons and spreads it across multiple neurons. The benefit of this is that all of the inputs will be likely used as opposed to a few.

In particular, the use of L2 regularization for images of faces is useful because some general properties of faces; e.g., nose, eyes, mouth, shadows, etc. are consistent across input images regardless of the actor/actress. Without regularization, these features would not contribute much to classification. By including regularization, weights are forced to be minimized and thus neurons that were previously inactive would be activated and contribute to classifying. Thus, any details specific to a certain actor/actress would be engaged in the classification process.

The benefit of regularization is its prevention of overfitting. For small training sets, it is common to overfit. By employing regularization, this overfitting will be omitted. This can be seen in the test results shown below. A training set of size 30 (5 images per actor/actress) was used. The performance on the test set of the regularized network is higher on average than that of the non-regularized. The regularization used was $\lambda = 1.5$. After testing with this parameter and multiple others ranging from 0.0001 to 2.0000, it was found that the results would be consistent for this value.

Please refer to Appendix B for this part's code.

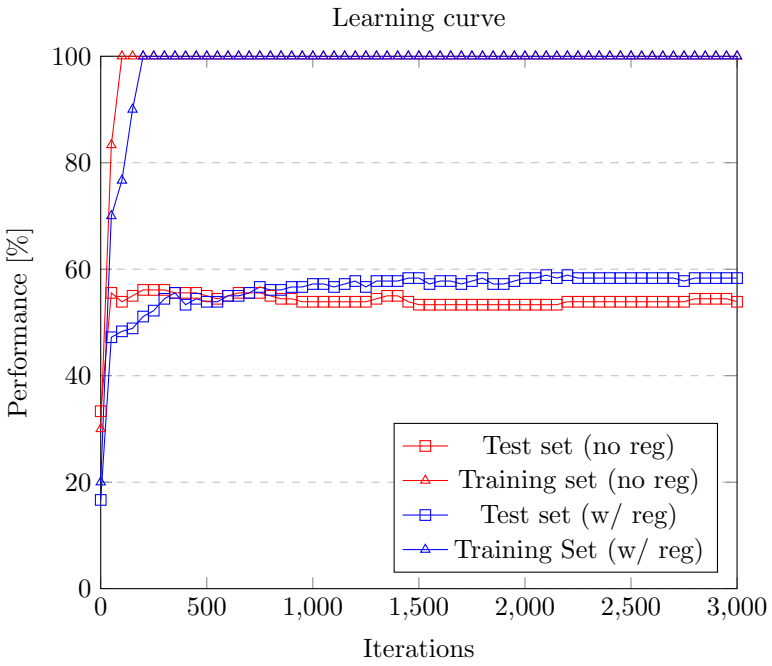
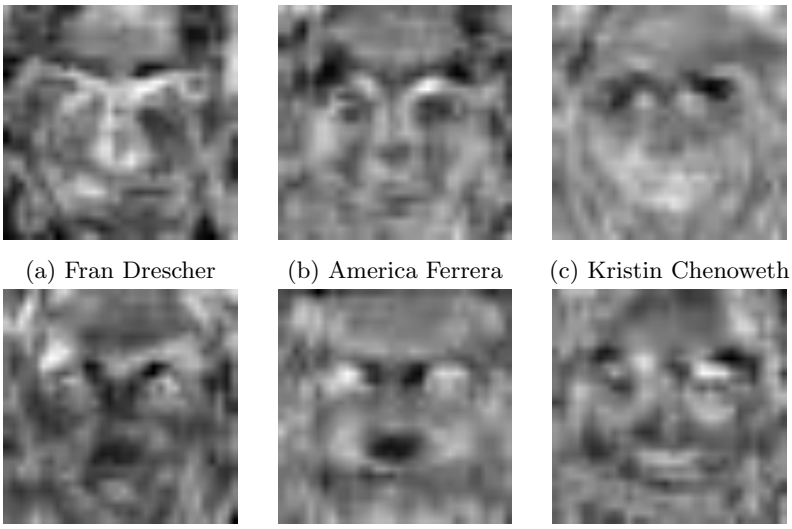


Figure 9

Part 9

Visualizing neural network weights.
Neurons that are activated by a given input will have weights that resemble the input. By feeding in a set of images, one image of each actor/actress, the maximum of all the neurons in the hidden layer was extracted. The weights connected to the maximum hidden neuron for the given actor/actress were then visualized, as can be seen in Figure 10. Before extracting the images, it was ensured that the image would actually classify correctly. If it was an incorrect classification, the weights would not resemble the actor/actress.

Please refer to Appendix B for this part's code.



(a) Fran Drescher (b) America Ferrera (c) Kristin Chenoweth

(d) Alec Baldwin (e) Bill Hader (f) Steve Carell

Figure 10: Visualizations of the weights connected to the max activated hidden neuron for a given input image of an actor/actress.

Part 10

Facial classification of actors/actresses using AlexNet.
By implementing AlexNet for facial classification, the neural network is able to classify actors/actresses with much greater accuracy. The implemented network can be broken down to two parts. The first part is AlexNet (up to the fourth convolution layer). The second part is a hidden layer, output and softmax as was implemented in Part 7. Holistically, this can be thought of as a neural network with multiple convolution and pooling layers (AlexNet), a fully connected layer and finally a softmax output.

The connection between the two components of the neural network is completed as follows. The output from conv4 of AlexNet when feeding in n 227×227 images is of size $n \times 13 \times 13 \times 384$. This 4D-array was reshaped into a 2D-array of size $n \times 64896$, where each image is flattened into a row. This data was then fed as the input of the single-hidden layer neural network. A network with 30 hidden units, activation function $g(t) = tanh(t)$ followed by $g(t) = t$ and $\lambda = 0.00005$ as determined in Part 7. Testing this network over 500 iterations in terms of performance for test, validation and training sets yielded results as shown in Figure 11. The performance of this network was 98.33% for the test set. This performance compared to the network in Part 7 (87.22%) represents a 12.74% performance boost. The performance of the training set was 100.0%, which is the same as Part 7.

Please refer to Appendix C for this part's code.

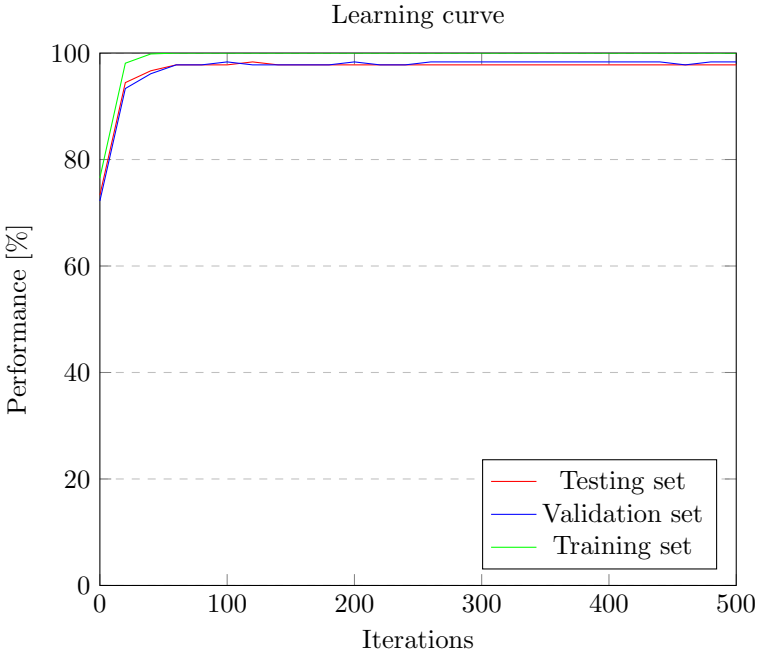


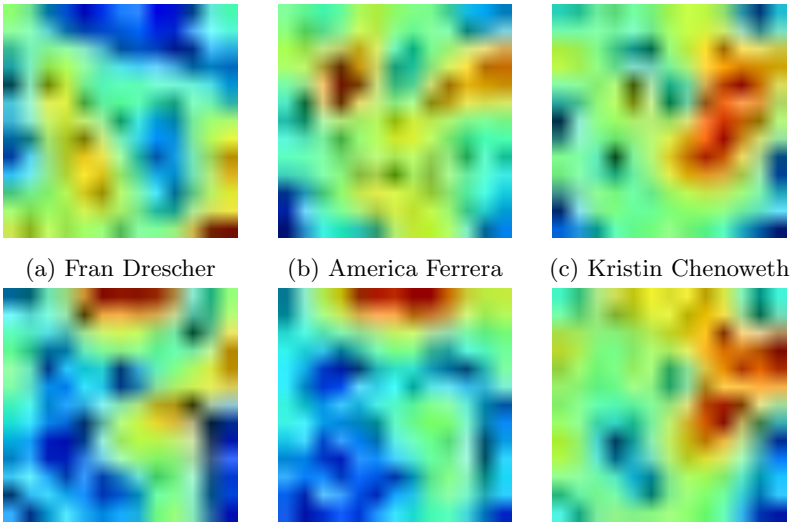
Figure 11

Part 11

Visualizing feature weights.
The weights of the input to the single hidden layer, fully connected network were visualized; that is, the weights that connect the flattened $13 \times 13 \times 384$ array (which is obtained from the output of conv4) to the hidden layer consisting of 30 neurons. The selection process was identical to Part 9, where the maximum neuron was taken when the network was fed an image of a specific actor/actress. The weights connecting to the max neuron would contain features that are inherent to the specific actor/actress that has been provided.

Since the input to the network is now of size $13 \times 13 \times 384$ instead of 32×32 (as was the case in Part 9), we will have $13 \times 13 \times 384$ weights. To visualize this, we chose to convolve the array with a $1 \times 1 \times 384$ filter of all 1s and a bias of all zeros. This way we have an equal weighting of all 384 features.

Figure 12 shows the resultant images which are of size 13×13. The actor/actress that the feature was taken from is labeled.



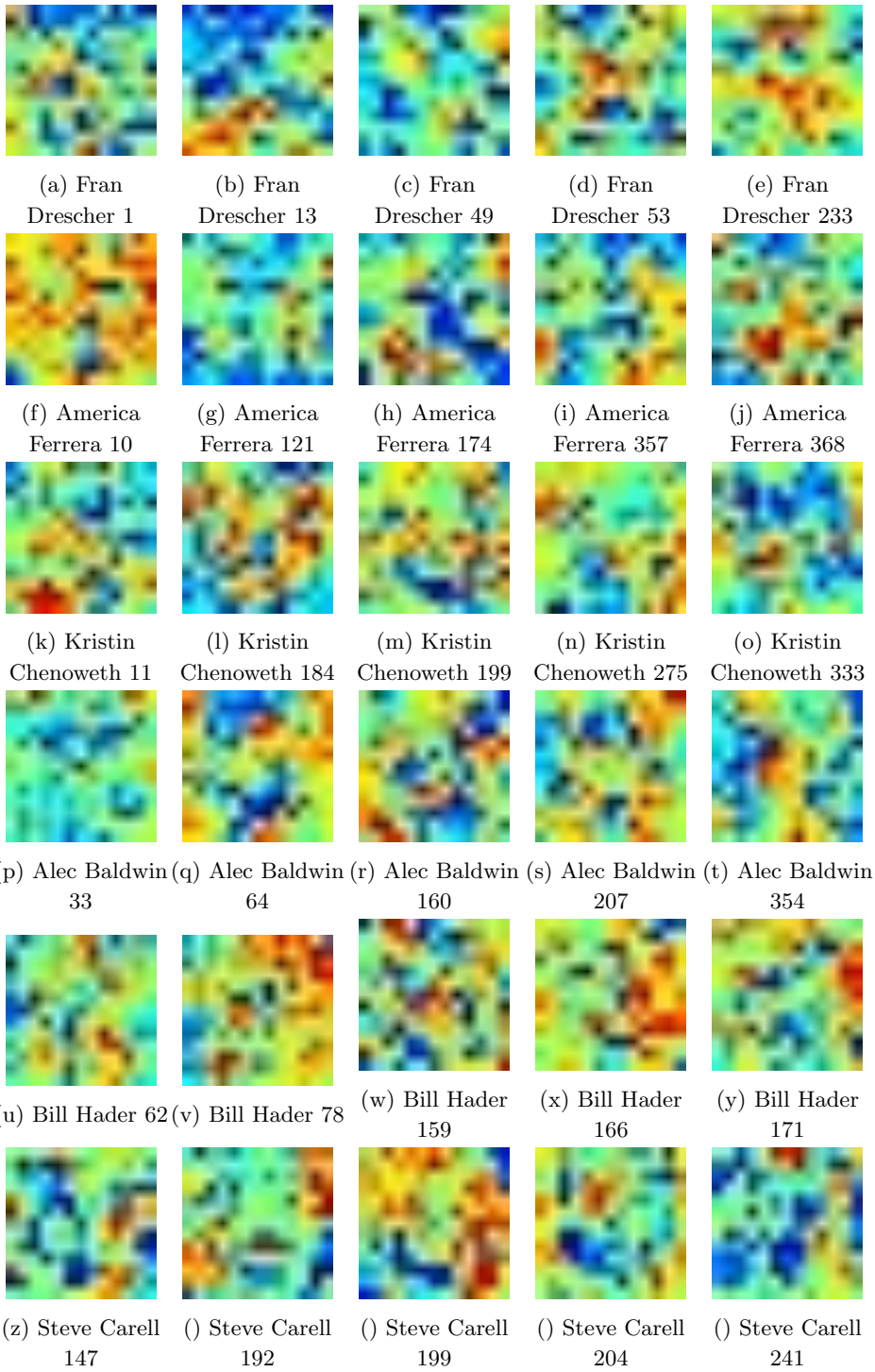
(a) Fran Drescher (b) America Ferrera (c) Kristin Chenoweth

(d) Alec Baldwin (e) Bill Hader (f) Steve Carell

Figure 12: Visualizations of the weights connected to the max activated hidden neuron for a given input image of an actor/actress.

The next method of visualization was simply randomly selecting a set of five layers from the 384 available and visualizing them individually. This is shown in Figure 13; each image is labeled with the actor/actress and the layer that is visualized.

Please refer to Appendix C for this part's code.



(a) Fran Drescher 1 (b) Fran Drescher 13 (c) Fran Drescher 49 (d) Fran Drescher 53 (e) Fran Drescher 233

(f) America Ferrera 10 (g) America Ferrera 121 (h) America Ferrera 174 (i) America Ferrera 357 (j) America Ferrera 368

(k) Kristin Chenoweth 11 (l) Kristin Chenoweth 184 (m) Kristin Chenoweth 199 (n) Kristin Chenoweth 275 (o) Kristin Chenoweth 333

(p) Alec Baldwin 33 (q) Alec Baldwin 64 (r) Alec Baldwin 160 (s) Alec Baldwin 207 (t) Alec Baldwin 354

(u) Bill Hader 62 (v) Bill Hader 78 (w) Bill Hader 159 (x) Bill Hader 166 (y) Bill Hader 171

(z) Steve Carell 147 () Steve Carell 192 () Steve Carell 199 () Steve Carell 204 () Steve Carell 241

Figure 13: Visualizations of the weights connected to the max activated hidden neuron for a given input image of an actor/actress. The number in the label is the layer number that was selected.

Appendix A

digits.py (Python 2.7)

```
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import time
from scipy.misc import imread
from scipy.misc import imresize
import scipy.stats
import matplotlib.image as mpimg
from scipy.ndimage import filters
import urllib
from numpy import random
from matplotlib.font_manager import FontProperties

import cPickle
import os
import timeit
from scipy.io import loadmat

#Load the MNIST digit data
M = loadmat("mnist_all.mat")

#Display the 150-th "5" digit from the training set

# plt.figure()
# for i in range(0,10):
#     for j in range(1,11):
#         plt.subplot(10,10,j+(i*10))
#         imshow(M["train" + str(i)][150+j].reshape((28,28)), cmap=cm.gray)
#
# plt.show()

def get_data(M):
    #we have a total 10 images
    #we first obtain the data images and concatenate vertically stacking them
    #The end result will have a height of 60,000
    Data = M["train0"]
    for i in range(1,10):
        Data = np.vstack((Data,M["train" + str(i)]))

    for i in range(0,10):
        if i == 0:
            y = np.zeros((1,10))
            y[0][i] = 1
            y = repeat(y[0][newaxis,:],len(M["train"+str(i)]),0)
            y_one_hot = y
        else:
            y = np.zeros((1,10))
            y[0][i] = 1
            y = repeat(y[0][newaxis,:],len(M["train"+str(i)]),0)
```

```
            y_one_hot = np.vstack((y_one_hot,y))

    return Data.T,y_one_hot.T

#input x to be of the image from the training set.
#input already divided into its flattened image vector (len of 784)
initial_w = np.ones((10,784))/10000
bias = 1
# def sum_w_bias(x,initial_w,bias):
#     y = np.matmul(initial_w,x) + bias
#     return y
#
# def softmax(x,y,initial_w,bias):
#     '''Return the output of the softmax function for the matrix of output y. y
#     is an NxM matrix where N is the number of outputs for a single case, and M
#     is the number of cases'''
#     y = sum_w_bias(x,initial_w,bias)
#     return exp(y)/(tile(sum(exp(y),0), (len(y),1)))

def softmax(X, W):
    '''
    Takes the matrices X and W and returns the softmax probabilities matrix P
    :param X: flattened image matrix where each row is a flattened image with a bias (
        nx785 numpy array)
    :param W: weight matrix where the ith row corresponds to the weights for output i
        (10x785 numpy array)
    :return: the softmax probabilities matrix (nx10 numpy array)
    '''
    O = np.dot(X, W.T)
    P = np.exp(O)/(np.array([np.sum(np.exp(O), 1)])).T
    return P


def tanh_layer(y, W, b):
    '''Return the output of a tanh layer for the input matrix y. y
    is an NxM matrix where N is the number of inputs for a single case, and M
    is the number of cases'''
    return tanh(dot(W.T, y)+b)

def forward(x, W0, b0, W1, b1):
    L0 = tanh_layer(x, W0, b0)
    L1 = dot(W1.T, L0) + b1
    output = softmax(L1)
    return L0, L1, output

# y_ = get_data(M)[1]
# y_out = sum_w_bias(get_data(M)[0],initial_w,bias
# y = softmax(y_out)
def NLL(y, y_):
    return -sum(y_*log(y))

def gradient(P,X,Y):
    grad = np.dot(X.T,P-Y)
```

```
    return grad

def finite_diff(W,x,y,h):
    #Param W: initial weights of 10x784. When computing the gradient we fix all the
    #values except one of them.
    #Cost is dependant on softmax which is dependant on the output where we will
        change
    #the weights.
    grad = np.zeros((10,784))
    for i in range(0,10):
        for j in range(0,785):
            H = np.zeros((10,784))
            H[i,j] = h
            P_plushalf = softmax(x,y,W+H/2,bias)
            P_minhalf = softmax(x,y,W-H/2,bias)
            dcdw = (NLL(P_plushalf,y) - NLL(P_minhalf,y))/h
            grad[i,j] = dcdw
            #print(grad[i,j])
            if i == 0 and j == 30:
                break
        break

    grad_diff = []
    p = softmax(x,y,W,bias)
    vec_grad = gradient(p,x,y)
    for i in range(0,30):
        grad_diff.append(abs(grad[0,i] - vec_grad[0,i]))

    grad_diff_accum = sum(grad_diff)
    return grad_diff_accum

def get_performance_log(Y, X, W):
    '''
    Gets the performance of the data Y, X given the W matrix
    :param Y: classification matrix where each row is the one-hot encoding vector for
        an image (nx10 numpy array)
    :param X: flattened image matrix where each row is a flattened image (nx785 numpy
        array)
    :param W: weight matrix where the ith row corresponds to the weights for output i
        (10x785 numpy array)
    :return: percentage of correct classifications
    '''
    P = softmax(X, W)
    indices_test = np.argmax(P, 1)
    indices_actual = np.argmax(Y, 1)
    performance = (indices_test.shape[0] - np.count_nonzero(indices_test -
        indices_actual))/(.01*indices_test.shape[0])
    return performance

def grad_descent(Y, X, W, alpha):
    '''
    Executes gradient descent on the data set; computes the W matrix for which the
        cost function is minimized
```

```
    :param Y: classification matrix where each row is the one-hot encoding vector for
        an image (nx10 numpy array)
    :param X: flattened image matrix where each row is a flattened image (nx785 numpy
        array)
    :param W: weight matrix where the ith row corresponds to the weights for output i
        (10x785 numpy array)
    :param alpha: step size for gradient descent
    :return: W matrix for which cost is minimized
    '''
    EPS = 1e-6
    prev_W = W - 10*EPS
    max_iter = 10000
    iter = 0

    #X_test, Y_test = get_data('test')

    while np.linalg.norm(W-prev_W) > EPS and iter < max_iter:
        prev_W = W.copy()
        P = softmax(X,W)
        W -= alpha*gradient(P, Y, X)
        # if iter % 100 == 0:
        #     print 'Iteration:', iter
        #     print 'Cost', cost(P, Y)
        #     print 'Train Performance', get_performance(Y, X, W)
        #     print 'Test Performance', get_performance(Y_test, X_test, W)
        #     print
        iter += 1
    #c = cost(P, Y)

    #print "Minimum found at", W, "with cost function value of", c, "on iteration",
        iter
    return W

#Part 5:
def df(x, y, theta):
    return -sum((y.T-dot(theta, x.T))*x.T, 1)

#Using The Derivative of the Cost Function to Evaluate Grad. Descent To evaluate Theta
    .
#J(theta0,...,thetaN) - alpha*grad(J(theta0,...,thetaN))
def grad_descent_lin(df, x, y, init_t, alpha):
    #Evaulating Gradient Descent
    EPS = 1e-10 #EPS = 10**(-5)
    prev_t = init_t-10*EPS
    t = init_t.copy()
    max_iter = 10000
    iter  = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        t -= alpha*df(x, y, t)
        #print "Iter", iter
        #print "x = (%.2f, %.2f, %.2f), f(x) = %.2f" % (t[0], t[1], t[2], f(x, y, t))
        #print "Gradient: ", df(x, y, t), "\n"
```

```
        iter += 1
    return t

def get_performance_logp5(x,y,theta):
    a = 0
    b = 0
    for i in range(0,len(x)):
        if i < (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T)) ) > 0:
                a += 1

        elif i > (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T))) < 0:
                b += 1

    performance = (float(a+b))/(len(x))
    return performance

def get_performance_linp5(x,y,theta):
    a = 0
    b = 0
    for i in range(0,len(x)):
        if i < (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T)) ) > 0:
                a += 1

        elif i > (len(x)/2):
            if (y[i][0] - float(dot(theta,x[i].T))) < 0:
                b += 1

    performance = (float(a+b))/(len(x))
    #print ("linear regression performance:", performance)
    return performance

def plot_line(theta, x_min, x_max, color, label):
    x_grid_raw = arange(x_min, x_max, 0.01)
    x_grid = vstack((    ones_like(x_grid_raw),
                         x_grid_raw,
                    ))
    y_grid = dot(theta, x_grid)
    plot(x_grid[1,:], y_grid, color, label=label)

theta=np.array([0.5,0.5])

#gen_data(theta,50,20,1e-5,1e-8,np.ones((1,2)))
def gen_data(theta, N, sigma,alpha1,alpha2,init_t):

    # Actual data
    x_raw = 100*(random.random((N))-.5)
    x1 = vstack((    ones_like(x_raw),
                    x_raw,
                    ))
```

```
    x2 = vstack((    ones_like(x_raw),
                    x_raw,
                    ))

    y1 = dot(theta, x1) + scipy.stats.norm.rvs(loc = 2.5*sigma,scale=sigma,size=N)
    y2 = dot(theta, x2) - scipy.stats.norm.rvs(loc = .5*sigma,scale= sigma,size=N)
    #outlier
    x_out = -50
    x_out = vstack((    ones_like(x_out),
                    x_out,
                    ))
    y_out = dot(theta,x_out) + 400

    plot(x1[1,:], y1, "bo", label = "Training set (y = 1)")
    plot(x2[1,:], y2, "ro", label = "Training set (y = 0)")
    plot(x_out[1,:],y_out,"bo")

    #Apply gradient descent and calculate performance
    #linear regression
    y_temp1 = np.ones((N+1,1))
    y_temp2 = np.zeros((N,1))
    y = np.array((2*N,1))
    y = np.vstack((y_temp1,y_temp2))
    x = np.vstack((x1.T,x_out.T))
    x = np.vstack((x,x2.T))
    t_lin = grad_descent_lin(df, x, y, init_t, alpha1)
    t_log = grad_descent(y,x,init_t,alpha2)

    plot_line(theta, -100, 100, "b", "Actual generating process")
    plot_line(t_lin[0], -100, 100, "g", "lin generating process")
    plot_line(t_log[0], -100, 100, "m", "log generating process")
    legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),

          fancybox=True, shadow=True, ncol=6)
    xlim([-100, 100])
    ylim([-200, 500])

    y1 = np.array([y1])
    y2 = np.array([y2])
    y3 = np.array([y_out])
    y_actual = np.vstack((y1.T,y3))
    y_actual = np.vstack((y_actual,y2.T))
    p_lin = get_performance_linp5(x,y_actual,t_lin[0])
    p_log = get_performance_logp5(x,y_actual,t_log)

    return plt.show(),t_lin[0],t_log[0],p_lin,p_log
```

Appendix B

faces.py (Python 2.7)

```
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import time
from scipy.misc import imread
from scipy.misc import imresize
import matplotlib.image as mpimg
from scipy.ndimage import filters
import urllib
from numpy import random
import cPickle
import os
from scipy.io import loadmat
import tensorflow as tf
import hashlib
from scipy.misc import imsave

# generate random seed based on current time
t = int(time.time())
print "t=", t
random.seed(t)

def timeout(func, args=(), kwargs={}, timeout_duration=1, default=None):
    '''
    From: http://code.activestate.com/recipes/473878-timeout-function-using-threading/
    '''
    import threading
    class InterruptableThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            try:
                self.result = func(*args, **kwargs)
            except:
                self.result = default

    it = InterruptableThread()
    it.start()
    it.join(timeout_duration)
    if it.isAlive():
        return False
    else:
        return it.result

def crop(img, coords):
    '''
    Get cropped RGB image.
```

```
    :param img: image as numpy array (n x m x 3).
    :param coords: coordinates of crop location of the form [x1, y1, x2, y2], where (
        x1, y1) is the top-left pixel and
                    (x2, y2) is the bottom right pixel.
    :return: cropped RGB image.
    '''
    img_array = imread(img, mode="RGB")
    return img_array[int(coords[1]):int(coords[3]), int(coords[0]):int(coords[2])]

def rgb2gray(rgb):
    '''
    Convert RGB image to grayscale image.
    :param rgb: image as numpy array (n x m x 3).
    :return: grayscale image.
    '''
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray/255.

def create_set(act):
    '''
    Create set data set containing all valid images of actors/actresses in act.
    :param act: list of actors/actresses' names.
    :return: data set of the form:
                {
                actor1: [[img1] ... [imgn]],
                actor2: [[img1] ... [imgn]],
                   :
                actor6: [[img1] ... [imgn]]
                }
    '''
    testfile = urllib.URLopener()
    set = {}

    # loop through each actor/actress in act
    for a in act:
        a_array = zeros((0, 1024))
        name = a.split()[1].lower()
        i = 0

        # loop through each line in the raw data (every image of every
        # actor/actress is in 'faces_subset.txt'
        for line in open("faces_subset.txt"):
            if a in line:
                # filename is of the form: '[name][###].[ext]'
                filename = name+str(i).zfill(3)+"."+line.split()[4].split(".")[-1]
                # A version without timeout (uncomment in case you need to
                # unsupress exceptions, which timeout() does)
                # testfile.retrieve(line.split()[4], "uncropped/"+filename)
                # timeout is used to stop downloading images which take too long to
                    download
                timeout(testfile.retrieve, (line.split()[4], "uncropped/"+filename),
                    {}, 30)
```

```
                # remove images that are unreadable
                try:
                    imread("uncropped/"+filename)
                except IOError:
                    if os.path.isfile("uncropped/"+filename):
                        os.remove("uncropped/"+filename)
                    print "IOError"
                    continue

                if not os.path.isfile("uncropped/"+filename):
                    continue

                # remove images that have mismatched sha256 codes
                h = hashlib.sha256()
                h.update(open("uncropped/"+filename).read())
                print h.hexdigest(), line.split()[6]
                if str(h.hexdigest()) != line.split()[6]:
                    print "SHA256 does not match"
                    continue

                print filename

                # crop image
                coords = line.split("\t")[4].split(",")
                img = crop("uncropped/"+filename, coords)
                # imsave("cropped/crop_"+filename, img)

                # convert image to grayscale
                img = rgb2gray(img)
                # imsave("gray/gray_"+filename, img)

                # resize image to 32 x 32
                try:
                    img = imresize(img, (32,32))
                except ValueError:
                    os.remove("uncropped/"+filename)
                    print "ValueError"
                    continue

                # reshape image to 1 x 1024
                img = img.reshape((1, 1024))

                # store image in set
                a_array = vstack((a_array, img))

                i += 1

        set[name] = a_array

    return set

def partition(A):
```

```
    '''
    Partition data set into training, validation and testing sets. The training and
        validation sets will have 30 images
    of each actor/actress, randomly chosen. The training set will contain the
        remaining images.
    :param A: data set generated from create_set()
    :return: x and y matrices for the training, validation, and testing sets.
        train_set is a set of same format as
             A but containing only the partitioned training data.
    '''
    train_x = zeros((0, 32*32))
    train_y_ = zeros((0, 6))
    test_x = zeros((0, 32*32))
    test_y_ = zeros((0, 6))
    val_x = zeros((0, 32*32))
    val_y_ = zeros((0, 6))
    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]
    train_set = {}

    for k in range(6):
        size = len(A[names[k]])

        random_perm = random.permutation(size)

        idx_test = array(random_perm[:30])
        idx_val = array(random_perm[30:60])
        idx_train = array(random_perm[60:])

        train_set[names[k]] = (array(A[names[k]])[idx_train])

        test_x = vstack((test_x, ((array(A[names[k]])[idx_test])/255.)))
        val_x = vstack((val_x, ((array(A[names[k]])[idx_val])/255.)))
        train_x = vstack((train_x, ((array(A[names[k]])[idx_train])/255.)))

        one_hot = zeros(6)
        one_hot[k] = 1

        test_y_ = vstack((test_y_,   tile(one_hot, (30, 1))))
        val_y_ = vstack((val_y_,   tile(one_hot, (30, 1))))
        train_y_ = vstack((train_y_,   tile(one_hot, (size - 60, 1))))

    return train_x, train_y_, test_x, test_y_, val_x, val_y_, train_set

def get_batch(train_set, N):
    '''
    Gets a batch for minibatch training from the partitioned training set of size N.
    :param train_set: partitioned trianing set data.
    :param N: size of minibatch.
    :return: minibatch.
    '''
    n = N/6
    batch_x = zeros((0, 32*32))
    batch_y_ = zeros((0, 6))
```

```
    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]

    for k in range(6):
        train_size = len(train_set[names[k]])
        idx = array(random.permutation(train_size)[:n])
        batch_x = vstack((batch_x, ((array(train_set[names[k]])[idx])/255.)))
        one_hot = zeros(6)
        one_hot[k] = 1
        batch_y_ = vstack((batch_y_, tile(one_hot, (n, 1))))
    return batch_x, batch_y_

def create_nn(act1, act2, nhid = 300, sdev = 0.01, lam = 0.0):
    '''
    Creates the neural network given the input parameters.
    :param act1: activation function for first layer
    :param act2: activation function for second layer (output layer)
    :param nhid: number of hidden neurons
    :param sdev: standard deviation of weight initialization
    :param lam: lambda value for regularization
    :return: neural network (required variables/parameters for outer scope
        computations)
    '''
    # create placeholder for input (1024 neurons)
    x = tf.placeholder(tf.float32, [None, 1024])

    # randomly initialize the weights and biases for each layer with a normal
        distribution (standard deviation of 0.01)
    # layer connecting input to hidden
    W0 = tf.Variable(tf.random_normal([1024, nhid], stddev=sdev))
    b0 = tf.Variable(tf.random_normal([nhid], stddev=sdev))

    # layer connecting hidden to output
    W1 = tf.Variable(tf.random_normal([nhid, 6], stddev=sdev))
    b1 = tf.Variable(tf.random_normal([6], stddev=sdev))

    # initialize layers with activation functions
    if act1 == "t":
        layer1 = tf.matmul(x, W0)+b0
    elif act1 == "tanh":
        layer1 = tf.nn.tanh(tf.matmul(x, W0)+b0)
    elif act1 == "relu":
        layer1 = tf.nn.relu(tf.matmul(x, W0)+b0)

    if act2 == "t":
        layer2 = tf.matmul(layer1, W1)+b1
    elif act2 == "tanh":
        layer2 = tf.nn.tanh(tf.matmul(layer1, W1)+b1)
    elif act2 == "relu":
        layer2 = tf.nn.relu(tf.matmul(layer1, W1)+b1)

    # softmax output layer
    y = tf.nn.softmax(layer2)
```

```
255
    # create placeholder for classification input (6 neurons)
    y_ = tf.placeholder(tf.float32, [None, 6])

    # define cost and training step
260 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)

265 # init = tf.initialize_all_variables()
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)

270 correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1

275 def get_best_nn(act1s, act2s, nhids, lams):
    '''
    Loops through different neural networks and tests on the validation set. Returns
        the best performing network
    parameters.
    :param act1s: list of activation functions for first layer to test
280 :param act2s: list of activation functions for second layer to test
    :param nhids: list of number of hidden neurons to test
    :param lams: list of lambda values to test
    :return: best combination of parameters
    '''
285 max_acc = 0
    for act1 in act1s:
        for act2 in act2s:
            for nhid in nhids:
                for lam in lams:
290                 max_val_acc = 0
                    sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1 =
                         create_nn(act1, act2, nhid=nhid, sdev=0.01, lam=lam)
                    print "Testing with:"
                    print "\tFirst layer activation function:", act1
                    print "\tSecond layer (output) activation function:", act2
295                 print "\tNumber of hidden neurons:", nhid
                    print "\tLambda", lam
                    for i in range(2000):
                        if minibatch_enabled:
                            batch_x, batch_y_ = get_batch(train_set, N)
300                         sess.run(train_step, feed_dict={x: batch_x, y_: batch_y_})
                        else:
                            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

                        val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
305                     if val_acc > max_val_acc:
```

```
                            max_val_acc = val_acc

                    if max_val_acc > max_acc:
                        max_acc = max_val_acc
                        params = [act1, act2, nhid, lam]
                    print "Best validation performance was:", max_val_acc
                    print

    print "The chosen parameters are:"
    print "\tFirst layer activation function:", act1
    print "\tSecond layer (output) activation function:", act2
    print "\tNumber of hidden neurons:", nhid
    print "\tLambda", lam
    print

    return params

if __name__ == "__main__":
    act = ["Fran Drescher", "America Ferrera", "Kristin Chenoweth", "Alec Baldwin", "
        Bill Hader", "Steve Carell"]
    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]

    if not os.path.exists("act_set.npy"):
        A = create_set(act)
        np.save("act_set.npy", A)
    else:
        A = np.load("act_set.npy").item()

    train_x, train_y_, test_x, test_y_, val_x, val_y_, train_set = partition(A)


    # PART 7: choosing best network parameters and training/testing on best
    N = 180
    minibatch_enabled = True
    val_test = False  # set True for Part 7

    if val_test:
        # test on validation set and check which set of parameters are the best
        act1s = ["relu", "tanh", "t"]
        act2s = ["relu", "tanh", "t"]
        nhids = [30, 70, 100, 300]
        lams = [0.00000, 0.00005, 0.0001, 0.00015]

        params = get_best_nn(act1s, act2s, nhids, lams)

    else:
        params = ["tanh", "t", 30, 0.00005]

    # now use the best results from the validation set to test on the rest of the sets
    act1, act2, nhid, lam = params[0], params[1], params[2], params[3]
    sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1 = create_nn(act1,
         act2, nhid=nhid, sdev=0.01, lam=lam)
```

```
    test_plot = ""
    val_plot = ""
    train_plot = ""

    for i in range(0):
        if minibatch_enabled:
            batch_x, batch_y_ = get_batch(train_set, N)
            sess.run(train_step, feed_dict={x: batch_x, y_: batch_y_})
        else:
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 50 == 0:
            print "i=", i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print "Test:", test_acc
            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print "Validation:", val_acc
            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print "Train:", train_acc
            print "Penalty:", sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print "Output for LaTeX plotting:"
    print "Test", test_plot
    print "Validation", val_plot
    print "Train", train_plot


    # PART 8: use a small training set to test performance (5 images of each actor)
        with and without regularization
    train_x, train_y_ = get_batch(train_set, 30)
    minibatch_enabled = False

    # no regularization
    params = ["tanh", "t", 30, 0.00000]

    act1, act2, nhid, lam = params[0], params[1], params[2], params[3]
    sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1 = create_nn(act1,
         act2, nhid=nhid, sdev=0.01, lam=lam)

    test_plot = ""
    val_plot = ""
    train_plot = ""

    print "NO REGULARIZATION"

    for i in range(3001):
        if minibatch_enabled:
```

```
            batch_x, batch_y_ = get_batch(train_set, N)
            sess.run(train_step, feed_dict={x: batch_x, y_: batch_y_})
        else:
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 50 == 0:
            print "i=", i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print "Test:", test_acc
            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print "Validation:", val_acc
            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print "Train:", train_acc
            print "Penalty:", sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print "Output for LaTeX plotting:"
    print "Test", test_plot
    print "Validation", val_plot
    print "Train", train_plot

    # with regularization
    params = ["tanh", "t", 30, 1.5]

    act1, act2, nhid, lam = params[0], params[1], params[2], params[3]
    sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1 = create_nn(act1,
         act2, nhid=nhid, sdev=0.01, lam=lam)

    test_plot = ""
    val_plot = ""
    train_plot = ""

    print "REGULARIZATION WITH LAMBDA =", lam

    for i in range(3001):
        if minibatch_enabled:
            batch_x, batch_y_ = get_batch(train_set, N)
            sess.run(train_step, feed_dict={x: batch_x, y_: batch_y_})
        else:
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 50 == 0:
            print "i=", i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print "Test:", test_acc
            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print "Validation:", val_acc
```

```
            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print "Train:", train_acc
            print "Penalty:", sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print "Output for LaTeX plotting:"
    print "Test", test_plot
    print "Validation", val_plot
    print "Train", train_plot


    # PART 9: visualizing weights
    neurons, neuron_ids = [], []

    # get 1 image of each actor/actress and input to neural network
    batch_x, batch_y_ = get_batch(train_set, 6)
    for i in range(6):
        single_x = np.array([batch_x[i, :]])
        single_y_ = np.array([batch_y_[i, :]])

        acc = sess.run(accuracy, feed_dict={x: single_x, y_: single_y_})

        # image is correctly classified
        if acc == 1:
            # get the output from the hidden layer
            hidden_layer = sess.run(layer1, feed_dict={x: single_x, y_: single_y_})
            # add the index of the max hidden layer neuron
            neurons += [argmax(hidden_layer)]
            # stores which actor/actress does this image belong to
            neuron_ids += [argmax(single_y_)]

    W = sess.run(W0)
    for i in range(len(neurons)):
        w = W[:, neurons[i]]
        a = act[neuron_ids[i]]
        save_name = names[neuron_ids[i]]
        w = reshape(w, [32, 32])
        imsave('weights/w'+str(save_name)+'.jpg', w)
```

Appendix C

deepfaces.py (Python 2.7)

```
from pylab import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import time
from scipy.misc import imread
from scipy.misc import imresize
import matplotlib.image as mpimg
from scipy.ndimage import filters
import urllib
from numpy import random
import cPickle
import os
from scipy.io import loadmat
import tensorflow as tf
import hashlib
from scipy.misc import imsave
from numpy import *
import os
from pylab import *
import time
from scipy.misc import imread
from scipy.misc import imresize
from scipy.ndimage import filters
import urllib
from numpy import random
from caffe_classes import class_names

# generate random seed based on current time
t = int(time.time())
print "t=", t
random.seed(t)

def timeout(func, args=(), kwargs={}, timeout_duration=1, default=None):
    '''
    From: http://code.activestate.com/recipes/473878-timeout-function-using-threading/
    '''
    import threading
    class InterruptableThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            try:
                self.result = func(*args, **kwargs)
            except:
                self.result = default

    it = InterruptableThread()
```

```
    it.start()
    it.join(timeout_duration)
    if it.isAlive():
        return False
    else:
        return it.result

def crop(img, coords):
    '''
    Get cropped RGB image.
    :param img: image as numpy array (n x m x 3).
    :param coords: coordinates of crop location of the form [x1, y1, x2, y2], where (
        x1, y1) is the top-left pixel and
                    (x2, y2) is the bottom right pixel.
    :return: cropped RGB image.
    '''
    img_array = imread(img, mode="RGB")
    return img_array[int(coords[1]):int(coords[3]), int(coords[0]):int(coords[2])]

def create_set(act):
    '''
    Create set data set containing all valid images of actors/actresses in act.
    :param act: list of actors/actresses' names.
    :return: data set of the form:
                {
                actor1: [[img1] ... [imgn]],
                actor2: [[img1] ... [imgn]],
                   :
                actor6: [[img1] ... [imgn]]
                }
    '''
    testfile = urllib.URLopener()
    set = {}

    # loop through each actor/actress in act
    for a in act:
        a_array = []
        name = a.split()[1].lower()
        i = 0

        # loop through each line in the raw data (every image of every
        # actor/actress is in 'faces_subset.txt'
        for line in open("faces_subset.txt"):
            if a in line:
                # filename is of the form: '[name][###].[ext]'
                filename = name+str(i).zfill(3)+"."+line.split()[4].split(".")[-1]
                # A version without timeout (uncomment in case you need to
                # unsupress exceptions, which timeout() does)
                # testfile.retrieve(line.split()[4], "uncropped/"+filename)
                # timeout is used to stop downloading images which take too long to
                    download
                timeout(testfile.retrieve, (line.split()[4], "part10/"+filename), {},
                    30)
```

```
                # remove images that are unreadable
                try:
                    imread("part10/"+filename).astype(float32)
                except IOError:
                    if os.path.isfile("part10/"+filename):
                        os.remove("part10/"+filename)
                    print "IOError"
                    continue

                if not os.path.isfile("part10/"+filename):
                    continue

                # remove images that have mismatched sha256 codes
                h = hashlib.sha256()
                h.update(open("part10/"+filename).read())
                # print h.hexdigest(), line.split()[6]
                if str(h.hexdigest()) != line.split()[6]:
                    print "SHA256 does not match"
                    continue

                print filename

                # crop image
                coords = line.split("\t")[4].split(",")
                img = crop("part10/"+filename, coords)
                # imsave("cropped/crop_"+filename, img)

                # resize image to 227 x 227
                try:
                    img = imresize(img, (227, 227))
                except ValueError:
                    os.remove("part10/"+filename)
                    print "ValueError"
                    continue

                # swap red and blue channels
                img[:, :, 0], img[:, :, 2] = img[:, :, 2], img[:, :, 0]

                # store image in set
                a_array += [img]

                i += 1

        set[name] = a_array

    return set

def partition(conv4):
    '''
    Partition data set into training, validation and testing sets. The training and
        validation sets will have 30 images
    of each actor/actress, randomly chosen. The training set will contain the
```

```
        remaining images.
    :param conv4: output from conv4 layer
    :return: x and y matrices for the training, validation, and testing sets.
        train_set is a set of same format as
             conv4 but containing only the partitioned training data.
    '''
    train_x = zeros((0, 13*13*384))
    train_y_ = zeros((0, 6))
    test_x = zeros((0, 13*13*384))
    test_y_ = zeros((0, 6))
    val_x = zeros((0, 13*13*384))
    val_y_ = zeros((0, 6))
    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]
    train_set = {}

    for k in range(6):
        size = len(conv4[names[k]])

        random_perm = random.permutation(size)

        idx_test = array(random_perm[:30])
        idx_val = array(random_perm[30:60])
        idx_train = array(random_perm[60:])

        train_set[names[k]] = (array(conv4[names[k]])[idx_train])

        test_x = vstack((test_x, ((array(conv4[names[k]])[idx_test])/255.)))
        val_x = vstack((val_x, ((array(conv4[names[k]])[idx_val])/255.)))
        train_x = vstack((train_x, ((array(conv4[names[k]])[idx_train])/255.)))

        one_hot = zeros(6)
        one_hot[k] = 1

        test_y_ = vstack((test_y_, tile(one_hot, (30, 1))))
        val_y_ = vstack((val_y_, tile(one_hot, (30, 1))))
        train_y_ = vstack((train_y_, tile(one_hot, (size - 60, 1))))

    return train_x, train_y_, test_x, test_y_, val_x, val_y_, train_set

def get_batch(train_set, N):
    '''
    gets a batch for minibatch training from the partitioned training set of size N.
    :param train_set: partitioned trianing set data.
    :param N: size of minibatch.
    :return: minibatch.
    '''
    n = N/6
    batch_x = zeros((0, 13*13*384))
    batch_y_ = zeros((0, 6))

    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]

    for k in range(6):
```

```
        train_size = len(train_set[names[k]])
        idx = array(random.permutation(train_size)[:n])
        batch_x = vstack((batch_x, ((array(train_set[names[k]])[idx])/255.)))
        one_hot = zeros(6)
        one_hot[k] = 1
        batch_y_ = vstack((batch_y_, tile(one_hot, (n, 1))))
    return batch_x, batch_y_

def create_nn(act1, act2, nhid = 300, sdev = 0.01, lam = 0.0):
    '''
    Creates the neural network given the input parameters.
    :param act1: activation function for first layer
    :param act2: activation function for second layer (output layer)
    :param nhid: number of hidden neurons
    :param sdev: standard deviation of weight initialization
    :param lam: lambda value for regularization
    :return: neural network (required variables/parameters for outer scope
        computations)
    '''
    # create placeholder for input (13*13*384 neurons)
    x = tf.placeholder(tf.float32, [None, 13*13*384])

    # randomly initialize the weights and biases for each layer with a normal
        distribution (standard deviation of 0.01)
    # layer connecting input to hidden
    W0 = tf.Variable(tf.random_normal([13*13*384, nhid], stddev=sdev))
    b0 = tf.Variable(tf.random_normal([nhid], stddev=sdev))

    # layer connecting hidden to output
    W1 = tf.Variable(tf.random_normal([nhid, 6], stddev=sdev))
    b1 = tf.Variable(tf.random_normal([6], stddev=sdev))

    # initialize layers with activation functions
    if act1 == "t":
        layer1 = tf.matmul(x, W0)+b0
    elif act1 == "tanh":
        layer1 = tf.nn.tanh(tf.matmul(x, W0)+b0)
    elif act1 == "relu":
        layer1 = tf.nn.relu(tf.matmul(x, W0)+b0)

    if act2 == "t":
        layer2 = tf.matmul(layer1, W1)+b1
    elif act2 == "tanh":
        layer2 = tf.nn.tanh(tf.matmul(layer1, W1)+b1)
    elif act2 == "relu":
        layer2 = tf.nn.relu(tf.matmul(layer1, W1)+b1)

    # softmax output layer
    y = tf.nn.softmax(layer2)

    # create placeholder for classification input (6 neurons)
    y_ = tf.placeholder(tf.float32, [None, 6])
```

```
    # define cost and training step
    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)

    # init = tf.initialize_all_variables()
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)

    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    return sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1

def conv(input, kernel, biases, k_h, k_w, c_o, s_h, s_w,  padding="VALID", group=1):
    '''
    From https://github.com/ethereon/caffe-tensorflow
    '''
    c_i = input.get_shape()[-1]
    assert c_i%group==0
    assert c_o%group==0
    convolve = lambda i, k: tf.nn.conv2d(i, k, [1, s_h, s_w, 1], padding=padding)


    if group==1:
        conv = convolve(input, kernel)
    else:
        input_groups = tf.split(input, group, 3)
        kernel_groups = tf.split(kernel, group, 3)
        output_groups = [convolve(i, k) for i,k in zip(input_groups, kernel_groups)]
        conv = tf.concat(output_groups, 3)
    return  tf.reshape(tf.nn.bias_add(conv, biases), [-1]+conv.get_shape().as_list()
        [1:])

if __name__ == "__main__":
    act = ["Fran Drescher", "America Ferrera", "Kristin Chenoweth", "Alec Baldwin", "
        Bill Hader", "Steve Carell"]
    names = ["drescher", "ferrera", "chenoweth", "baldwin", "hader", "carell"]

    if not os.path.exists("part10_set.npy"):
        input = create_set(act)
        np.save("part10_set.npy", input)
    else:
        input = np.load("part10_set.npy").item()

    train_x = zeros((1, 227, 227, 3)).astype(float32)
    xdim = train_x.shape[1:]

    net_data = load("bvlc_alexnet.npy").item()

    x = tf.placeholder(tf.float32, (None,) + xdim)
```

```
    # conv1
    # conv(11, 11, 96, 4, 4, padding='VALID', name='conv1')
    k_h = 11; k_w = 11; c_o = 96; s_h = 4; s_w = 4
    conv1W = tf.Variable(net_data["conv1"][0])
    conv1b = tf.Variable(net_data["conv1"][1])
    conv1_in = conv(x, conv1W, conv1b, k_h, k_w, c_o, s_h, s_w, padding="SAME", group
        =1)
    conv1 = tf.nn.relu(conv1_in)

    # lrn1
    # lrn(2, 2e-05, 0.75, name='norm1')
    radius = 2; alpha = 2e-05; beta = 0.75; bias = 1.0
    lrn1 = tf.nn.local_response_normalization(conv1,
                                                      depth_radius=radius,
                                                      alpha=alpha,
                                                      beta=beta,
                                                      bias=bias)

    # maxpool1
    # max_pool(3, 3, 2, 2, padding='VALID', name='pool1')
    k_h = 3; k_w = 3; s_h = 2; s_w = 2; padding = 'VALID'
    maxpool1 = tf.nn.max_pool(lrn1, ksize=[1, k_h, k_w, 1], strides=[1, s_h, s_w, 1],
        padding=padding)

    # conv2
    # conv(5, 5, 256, 1, 1, group=2, name='conv2')
    k_h = 5; k_w = 5; c_o = 256; s_h = 1; s_w = 1; group = 2
    conv2W = tf.Variable(net_data["conv2"][0])
    conv2b = tf.Variable(net_data["conv2"][1])
    conv2_in = conv(maxpool1, conv2W, conv2b, k_h, k_w, c_o, s_h, s_w, padding="SAME",
         group=group)
    conv2 = tf.nn.relu(conv2_in)

    # lrn2
    # lrn(2, 2e-05, 0.75, name='norm2')
    radius = 2; alpha = 2e-05; beta = 0.75; bias = 1.0
    lrn2 = tf.nn.local_response_normalization(conv2,
                                                      depth_radius=radius,
                                                      alpha=alpha,
                                                      beta=beta,
                                                      bias=bias)

    # maxpool2
    # max_pool(3, 3, 2, 2, padding='VALID', name='pool2')
    k_h = 3; k_w = 3; s_h = 2; s_w = 2; padding = 'VALID'
    maxpool2 = tf.nn.max_pool(lrn2, ksize=[1, k_h, k_w, 1], strides=[1, s_h, s_w, 1],
        padding=padding)

    # conv3
    # conv(3, 3, 384, 1, 1, name='conv3')
    k_h = 3; k_w = 3; c_o = 384; s_h = 1; s_w = 1; group = 1
    conv3W = tf.Variable(net_data["conv3"][0])
```

```
    conv3b = tf.Variable(net_data["conv3"][1])
    conv3_in = conv(maxpool2, conv3W, conv3b, k_h, k_w, c_o, s_h, s_w, padding="SAME",
         group=group)
    conv3 = tf.nn.relu(conv3_in)

    # conv4
    # conv(3, 3, 384, 1, 1, group=2, name='conv4')
    k_h = 3; k_w = 3; c_o = 384; s_h = 1; s_w = 1; group = 2
    conv4W = tf.Variable(net_data["conv4"][0])
    conv4b = tf.Variable(net_data["conv4"][1])
    conv4_in = conv(conv3, conv4W, conv4b, k_h, k_w, c_o, s_h, s_w, padding="SAME",
        group=group)
    conv4 = tf.nn.relu(conv4_in)

    init = tf.initialize_all_variables()
    sess = tf.Session()
    sess.run(init)

    if not os.path.exists("conv4_set.npy"):
        conv4_set = {}
        for i in range(6):
            conv4_out = sess.run(conv4, feed_dict={x: input[names[i]]})
            a_array = zeros((0, 13*13*384))
            for j in range(conv4_out.shape[0]):
                flattened = conv4_out[j, :, :, :].reshape([1, 13*13*384])
                a_array = vstack((a_array, flattened))
            conv4_set[names[i]] = a_array

        np.save("conv4_set.npy", conv4_set)
    else:
        conv4_set = np.load("conv4_set.npy").item()

    train_x, train_y_, test_x, test_y_, val_x, val_y_, train_set = partition(conv4_set
        )

    # PART 10: deep neural network with convolution layer 4 output as input of single-
        hidden layer network
    N = 180
    minibatch_enabled = True

    act1, act2, nhid, lam = "tanh", "t", 30, 0.00005
    print "The chosen parameters are:"
    print "\tFirst layer activation function:", act1
    print "\tSecond layer (output) activation function:", act2
    print "\tNumber of hidden neurons:", nhid
    print "\tLambda", lam
    print
    sess, x, y_, train_step, decay_penalty, accuracy, W0, W1, layer1 = create_nn(act1,
         act2, nhid=nhid, sdev=0.01, lam=lam)

    test_plot = ""
    val_plot = ""
    train_plot = ""
```

```
    for i in range(501):
        if minibatch_enabled:
            batch_x, batch_y_ = get_batch(train_set, N)
            sess.run(train_step, feed_dict={x: batch_x, y_: batch_y_})
        else:
            sess.run(train_step, feed_dict={x: train_x, y_: train_y_})

        if i % 20 == 0:
            print "i=", i

            test_acc = sess.run(accuracy, feed_dict={x: test_x, y_: test_y_})
            print "Test:", test_acc
            val_acc = sess.run(accuracy, feed_dict={x: val_x, y_: val_y_})
            print "Validation:", val_acc
            train_acc = sess.run(accuracy, feed_dict={x: train_x, y_: train_y_})
            print "Train:", train_acc
            print "Penalty:", sess.run(decay_penalty)

            test_plot += str((i, test_acc*100))
            val_plot += str((i, val_acc*100))
            train_plot += str((i, train_acc*100))

    print
    print "Output for LaTeX plotting:"
    print "Test", test_plot
    print "Validation", val_plot
    print "Train", train_plot

    # PART 11: visualizing weights
    neurons, neuron_ids = [], []

    # get 1 image of each actor/actress and input to neural network
    batch_x, batch_y_ = get_batch(train_set, 6)
    for i in range(6):
        single_x = np.array([batch_x[i, :]])
        single_y_ = np.array([batch_y_[i, :]])

        acc = sess.run(accuracy, feed_dict={x: single_x, y_: single_y_})

        # image is correctly classified
        if acc == 1:
            # get the output from the hidden layer
            hidden_layer = sess.run(layer1, feed_dict={x: single_x, y_: single_y_})
            # add the index of the max hidden layer neuron
            neurons += [argmax(hidden_layer)]
            # stores which actor/actress does this image belong to
            neuron_ids += [argmax(single_y_)]

    W = sess.run(W0)
    for i in range(len(neurons)):
        w = W[:, neurons[i]]
        a = act[neuron_ids[i]]
```

```
        save_name = names[neuron_ids[i]]
        w = reshape(w, [1, 13, 13, 384])

        rand = random.permutation(384)[:5]

        for j in rand:
            wj = w[:, :, :, j]
            wj = wj.reshape([13, 13])

            imsave('part11/p11w'+str(save_name)+'feat'+str(j)+'.jpg', wj)

        x = tf.placeholder(tf.float32, (1, 13, 13, 384))
        k_h = 1; k_w = 1; c_o = 1; s_h = 1; s_w = 1
        convW = tf.Variable(ones((1, 1, 384, 1)).astype(float32))
        convb = tf.Variable(zeros((1,)).astype(float32))
        flat = conv(x, convW, convb, k_h, k_w, c_o, s_h, s_w, padding="SAME", group=1)
        init = tf.initialize_all_variables()
        sess = tf.Session()
        sess.run(init)

        flat_out = sess.run(flat, feed_dict={x: w})
        flat_out = flat_out[0, :, :, :]
        flat_out = flat_out.reshape((13, 13))

        imsave('part11/w'+str(save_name)+'.jpg', flat_out)
```