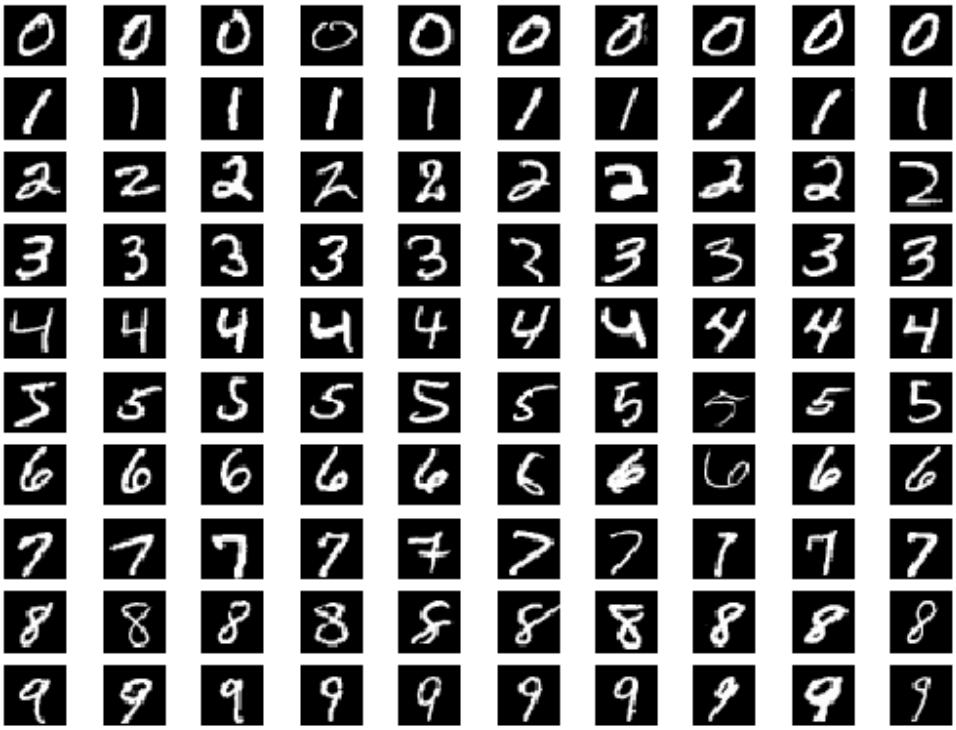
CSC411 Project 2 Report

Katie Datsenko

March 11, 2017

Part 1

The images of a single digit class vary due to different handwriting style. Some digits are slanted at different angles, others have different thickness. Some clearly have different styles, for instance the digit 2 can be written with a loop or straight line segment at its base. The digits can vary in size and its segments can be in different proportion to each other, for instance the 8 digit varies in size, it can appear compressed horizontally, and its two loops may be larger or smaller relative to eachother. Some curves also have a discontinuity, for instance the 0 digit.



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

def compute_softmax(x, W, b):
    '''Return the softmax matrix of any given 28x28 input image (flattened).
    W is a 784x10 matrix of weights
    x is an 784xN  matrix where N is the number of training cases
    b is a 10x1 vector
    y is a 10xN, where N is the number of training cases.
    (10x784)(784xN) + (10x1) = 10xN
    '''
    # calculate outputs
    o = dot(W.T, x)+b
    # calculate the softmax matrix
    return softmax(o)
```

Part 3

Part 3(a)

First (1) we compute all the relevant partial derivatives for a single training sample, and later (2) generalize each expression across all training samples. Finally, we produce (3) the fully vectorized result for $\frac{\partial C}{\partial w_{ij}}$ from the indexed sum expression.

1. Computation of all partial derivatives in view of a single training sample:
   $p_i = \frac{e^{o_i}}{\sum_l e^{o_l}}$
   We calculate $\frac{\partial p_j}{\partial o_i}$ between single o_i and a single p_j neuron. There are two cases:
   (a) Say $i = j$:

$$\frac{\partial p_j}{\partial o_i} = \frac{\partial p_i}{\partial o_i} = \frac{e^{o_i}}{(\sum_l e^{o_l})^2} - \frac{(e^{o_i})^2}{(\sum_l e^{o_l})^2} = \frac{e^{o_i}}{\sum_l e^{o_l}} \cdot (\frac{1}{\sum_l e^{o_l}} - \frac{e^{o_i}}{\sum_l e^{o_l}}) = p_i(1 - p_i)$$

   (b) Say $i \neq j$:

$$\frac{\partial p_j}{\partial o_i} = -\frac{e^{o_j} \cdot e^{o_i}}{(\sum_l e^{o_l})^2} = -p_j p_i$$

   Next, we calculate $\frac{\partial C}{\partial p_j}$:

$$C = -\sum_j y_j \log p_j$$

$$\frac{\partial C}{\partial p_j} = -\frac{y_j}{p_j}$$

   So that we can calculate $\frac{\partial C}{\partial o_i}$:

$$\frac{\partial C}{\partial o_i} = \sum_j \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i}$$

As two different cases exist for $i = j$ and $i \neq j$, we separate the $i = j$ case from the rest of the sum:

$$= (\sum_{\forall j, j \neq i} \frac{\partial C}{\partial p_j} \frac{\partial p_j}{\partial o_i}) + \frac{\partial C}{\partial p_i} \frac{\partial p_i}{\partial o_i}$$

$$= (\sum_{\forall j, j \neq i} -\frac{y_j}{p_j} \cdot -p_j p_i) - \frac{y_i}{p_i} \cdot p_i(1 - p_i)$$

$$= (\sum_{\forall j, j \neq i} y_j p_i) + y_i p_i - y_i = \sum_j y_j p_i - y_i = p_i \sum_j y_j - y_i$$

Since y encoded using one hot encoding, $\sum_j y_j = 1$ for any $y^{(i)}$:

$$\frac{\partial C}{\partial o_i} = p_i - y_i$$

Next, we calculate $\frac{\partial o_i}{\partial w_{ij}}$ (still only for a single training sample). Let m be the dimension of a single training example (i.e. 784 in the example). Let n be the number of categories that we can classify a training example as (i.e. the dimension of the output vector p, 10 in the example). W is the matrix of weights of dimension $n \times m$. A single entry is denoted as w_{ij}.

$$o_i = \sum_j w_{ij} x_j + b_i$$

$$\frac{\partial o_i}{\partial w_{ij}} = x_j$$

2. Now we generalize to the entire training sample space:
   Let N be the number of training cases. Let s denote the current sample in the training set. Let m be the dimension of a single training example (i.e. 784 in the example). Let j denote the index of the input dimension (1 to 784). Let n be the number of categories that we can classify a training example as (i.e. the dimension of the output vector p, 10 in the example). Let i denote the index of the output (1 to 10).
   X is a matrix of training examples of dimension $m \times N$ (each column contains a training sample). A single entry is denoted as x_{is}.
   W is the matrix of weights of dimension $m \times n$. A single entry is denoted as w_{ij}.
   P is a matrix containing the softmax output of the network for every training example (each column corresponds to a training example), of dimension $\times N$. A single entry is denoted as p_{is}.
   Y is a matrix containing the target outputs for each training example using one-hot encoding, of dimension $n \times N$. A single entry is denoted as y_{is}.

$$\frac{\partial C}{\partial w_{ij}} = \sum_s \frac{\partial C}{\partial o_{is}} \cdot \frac{\partial o_{is}}{\partial w_{ij}}$$

Before we computed $\frac{\partial o_i}{\partial w_{ij}}$ and $\frac{\partial C}{\partial o_i}$ in terms of a single training example. We can generalize this to the training space using w to denote each training sample.

$$\frac{\partial o_{is}}{\partial w_{ij}} = x_{js}$$

$$\frac{\partial C}{\partial o_{is}} = p_{is} - y_{is}$$

$$\frac{\partial C}{\partial w_{ij}} = \sum_s \frac{\partial C}{\partial o_{is}} \cdot \frac{\partial o_{is}}{\partial w_{ij}} = \sum_s (p_{is} - y_{is}) \cdot x_{js}$$

3. Fully vectorized result for $\frac{\partial C}{\partial W}$ using X, Y, and P.

$$\frac{\partial C}{\partial W} = X(P - Y)^T$$

Part 3(b)

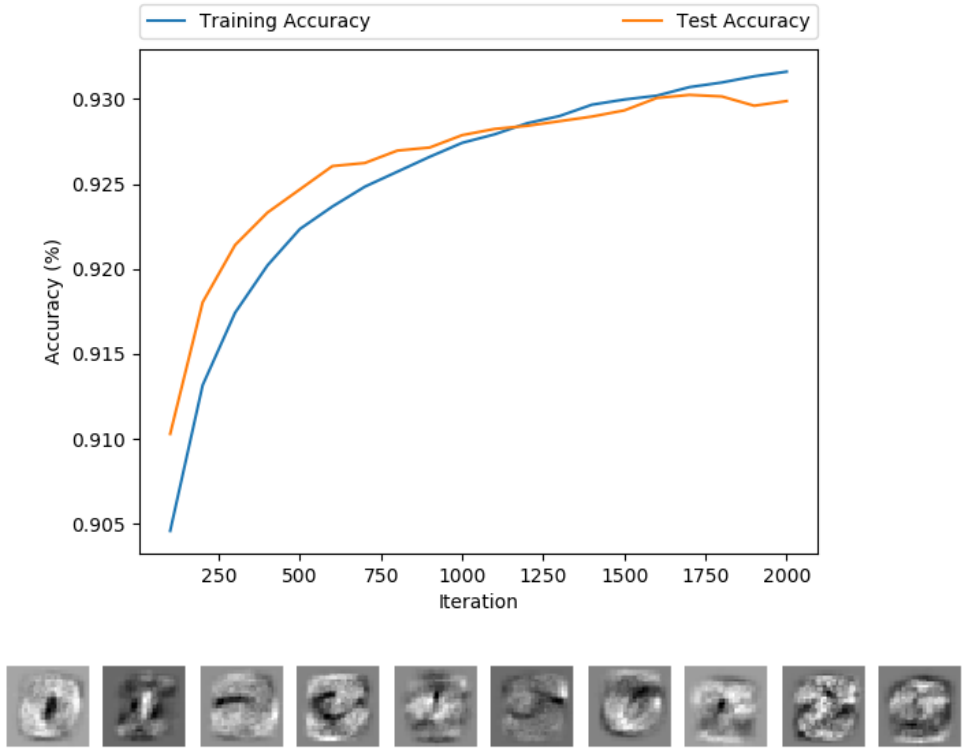
```
def gradient_Wb(x, W, b, t):
    '''Return the gradient of the cost function with respect to W and b.
    W - 784x10 matrix of weights
    x - 784xN  matrix where N is the number of training cases
    b - 10x1 vector
    y - 10xN, where N is the number of training cases.
    t - 10xN, representing the target outputs using one-hot codes (each col)
    @Returns:
    dCdW is a 784x10 matrix
    dCdb is a 10x1 vector
    '''
    # gradient of cost function is p-t, where t is a 10xM one-hot encoding
    # vector, and p is the softmax of y

    p = compute_softmax(x, W, b)
    dCdy = p - t

    dCdW = dot(x, dCdy.T)
    dCdb = dot(dCdy, ones((dCdy.shape[1], 1)))
    return dCdW, dCdb

def cost(x, W, b, t):
    '''
    t - target
    y - output
    '''
    y = compute_softmax(x, W, b)
    return -sum(t*log(y))

def finite_diff_Wij(x, W, b, i, j, t):
    '''
   Returns dW using finite difference approximate of the gradient of the cost
   with respect to W, at coordinate i
    '''
    h_w = zeros(W.shape)
    h_val = 0.00001
    h_w[i][j] = h_val
    finite_diff_w = (cost(x, W + h_w, b, t) - cost(x, W - h_w, b, t))/ (2*h_val)
    return finite_diff_w

def finite_diff_Bj(x, W, b, j, t):
    '''
   Returns dW using finite difference approximate of the gradient of the cost
   with respect to W, at coordinate i
    '''
    h_b = zeros(b.shape)
    h_val = 0.00001
    h_b[j] = h_val
    finite_diff_b = (cost(x, W, b + h_b, t) - cost(x, W, b - h_b, t))/ (2*h_val)
```

```
    return finite_diff_b

def compare_gradient(num_components=10):
    '''Print out the difference in gradient computation with function of part 3
    and a finite difference approximation function for the same set of data'''
    # Initiate the data for comparison
    np.random.seed(0)
    W = np.random.rand(784, 10)
    W /= W.size
    b = zeros((10, 1))
    learning_rate = 0.001

    x = M["test0"][10].T/255.0
    x = x.reshape((784, 1)) #a single training case
    t = zeros((10, 1))
    t[0] = 1

    dW, db = gradient_Wb(x, W, b, t)

    #get rand sample of components of W...
    i_sample = np.random.randint(0, W.shape[0], num_components)
    j_sample = np.random.randint(0, W.shape[1], num_components)

    for c in range(num_components):
        i = i_sample[c]
        j = j_sample[c]
        fd_dW = finite_diff_Wij(x, W, b, i, j, t)
        fd_db = finite_diff_Bj(x, W, b, j, t)

        print("dC/dWij predicted by finite-diff for (i="+repr(i)+", j="+repr(j)+"): "+repr(fd_dW))
        print("dC/dWij computed precisely by gradient func for (i="+repr(i)+", j="+repr(j)+"): "+r
        print("dC/dbj predicted by finite-diff for j="+repr(j)+": "+repr(fd_db))
        print("dC/dbj computed precisely by gradient func for j="+repr(j)+": "+repr(db[j]))
        print("----------------------------------------------------------------------")
```

```
================== RUNNING PART 3(b) ==================
Approximating the gradient at several coordinates using finite differences.
dC/dWij predicted by finite-diff for (i=702, j=6): 0.0
dC/dWij computed precisely by gradient func for (i=702, j=6): 0.0
dC/dbj predicted by finite-diff for j=6: 0.10000155825640177
dC/dbj computed precisely by gradient func for j=6: array([ 0.10000156])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=262, j=1): 0.098396574910175602
dC/dWij computed precisely by gradient func for (i=262, j=1): 0.098396574917197735
dC/dbj predicted by finite-diff for j=1: 0.099964647826133998
dC/dbj computed precisely by gradient func for j=1: array([ 0.09996465])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=682, j=5): 0.0
dC/dWij computed precisely by gradient func for (i=682, j=5): 0.0
dC/dbj predicted by finite-diff for j=5: 0.099999962177577104
dC/dbj computed precisely by gradient func for j=5: array([ 0.09999996])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=689, j=1): 0.0
dC/dWij computed precisely by gradient func for (i=689, j=1): 0.0
dC/dbj predicted by finite-diff for j=1: 0.099964647826133998
dC/dbj computed precisely by gradient func for j=1: array([ 0.09996465])
```

```
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=108, j=7): 0.0
dC/dWij computed precisely by gradient func for (i=108, j=7): 0.0
dC/dbj predicted by finite-diff for j=7: 0.09996351264529578
dC/dbj computed precisely by gradient func for j=7: array([ 0.09996351])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=696, j=5): 0.0
dC/dWij computed precisely by gradient func for (i=696, j=5): 0.0
dC/dbj predicted by finite-diff for j=5: 0.099999962177577104
dC/dbj computed precisely by gradient func for j=5: array([ 0.09999996])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=701, j=6): 0.0
dC/dWij computed precisely by gradient func for (i=701, j=6): 0.0
dC/dbj predicted by finite-diff for j=6: 0.10000155825640177
dC/dbj computed precisely by gradient func for j=6: array([ 0.10000156])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=187, j=7): 0.0027440964167624311
dC/dWij computed precisely by gradient func for (i=187, j=7): 0.0027440964251131592
dC/dbj predicted by finite-diff for j=7: 0.09996351264529578
dC/dbj computed precisely by gradient func for j=7: array([ 0.09996351])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=633, j=5): 0.064313701164664394
dC/dWij computed precisely by gradient func for (i=633, j=5): 0.064313701166109682
dC/dbj predicted by finite-diff for j=5: 0.099999962177577104
dC/dbj computed precisely by gradient func for j=5: array([ 0.09999996])
----------------------------------------------------------------------
dC/dWij predicted by finite-diff for (i=598, j=0): -0.89293969753878588
dC/dWij computed precisely by gradient func for (i=598, j=0): -0.89293969753329161
dC/dbj predicted by finite-diff for j=0: -0.89999850936273151
dC/dbj computed precisely by gradient func for j=0: array([-0.89999851])
----------------------------------------------------------------------
```

Figure 1: Part 4 Learning Curves

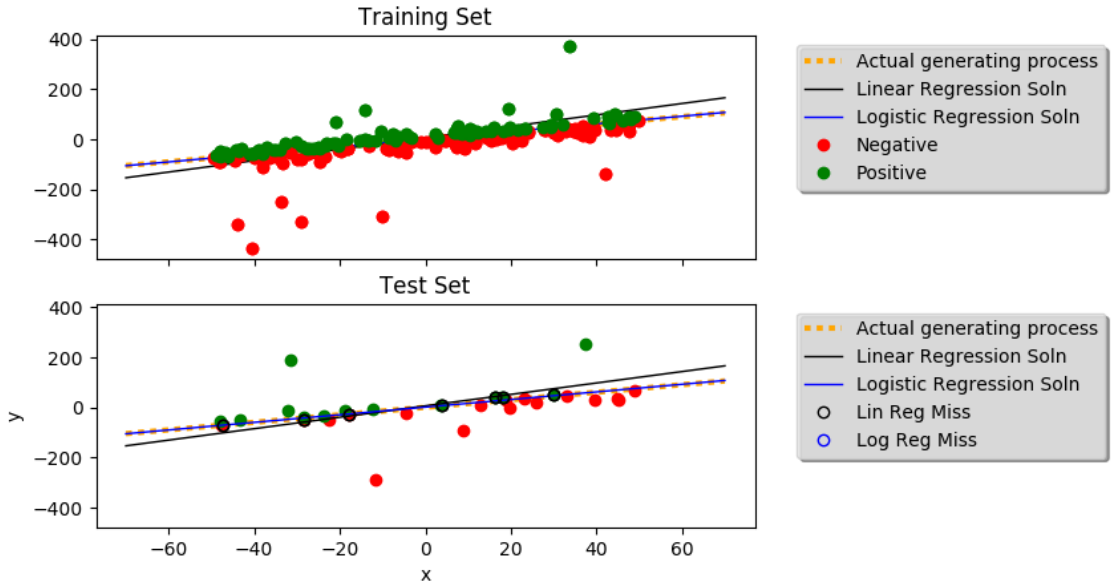


Part 4

The gradient descent algorithm (similar to Project 1) was implemented. See learning curves and weights connected to each of the output units above. The final parameters were:

- alpha = 0.00001
- max_iter = 2000
- EPS = 1e-10

To optimize the results of the algorithm, it was necessary to tune the learning rate (alpha parameter). I found that if the alpha parameter is too large, the algorithm overshoots the minimum and the values of the gradient, the cost and theta begin to overflow. If the alpha parameter is too small, the algorithm may not converge to the minimum fast enough because there are not enough iterations or the EPS is too big comparatively to the steps taken via the alpha. For sufficiently small alpha, the value of the cost function will appear to get smaller with each iteration. But if alpha is too small, the algorithm is slow to converge. Trying a bigger alpha may get closer to the minimum faster, as along as it doesn't overshoot the minimum. The EPS value is also important. If we intend for the alpha to decrease in value, the EPS value should also decrease in value to compensate for the smaller steps in change of gradient. As the point gets closer to the minimum, the gradient magnitude becomes smaller (geometrically, the area around the minimum flattens out), thus a smaller EPS value will allow the algorithm to converge as close as possible to the minimum at the cost of greater runtime. I found that setting the limit on the total number of iterations to 2000 is fine, and it also serves as a way of preventing overfitting on the training data to maximize performance on the validation set.



Part 5

```
================== RUNNING PART 5 ==================

------------------------------------------------------------
Results for Part 5 experiment (2 labels for the target 0/1):
Experiment of training set size 200 and test set size 30.
Linear Regression correctly classified 76.6666666667% of test set
Logistic Regression correctly classified 100.0% of test set
```

The disadvantage of using linear regression is that very incorrect predictions produce a large error, and correct predictions with high confidence will result in a large loss function value. This is due to the fact that the distance from the linear boundary is penalized quadratically; thus a large distance from the boundary will result in large loss. The advantage of using logistic regression is that predictions are normalized to the range [0, 1] so that very incorrect predictions are more penalized, and correct predictions with high confidence have a penalty that approaches 0.

The experiment is a simulation of a binary classification problem. The training set is generated as above. The red and green colors are used to distinguish the two classes. 95% of the points are generated close to the ground-truth theta boundary line (orange-dotted line). A special 5% of the points are given the correct label, but are generated far from the boundary line to represent the correct predictions made with high confidence in the experiment. Gradient descent is used with logistic regression and linear regression to converge to the optimal boundary parameters. The resulting parameters are displayed as lines on both graph. Logistic regression approximates the ground truth boundary the best. Linear regression is sensitive to the points far from the boundary. To minimize its cost function, it rotates towards the outlier points which contribute large loss, at the expense of incorrectly classifying a few points at the tails, where the linear regression boundary diverges the most from the ground-truth. Thus, on the test set linear regression has a lower accuracy rate. The performance of linear regression is 76.7% while the performance of logistic regression is 100%.
The relevant code is below. The code for the gradient descent procedure, as well as the cost and gradient functions of Logistic regression and Linear regression are listed below, followed by the code that generates the actual experiment.

```
def linreg_f(x, y, theta):
    #adds a row of 1s for the bias term on top
    #each column of the X matrix corresponds to a training example
    x = vstack( (ones((1, x.shape[1])), x))
    N = x.shape[1]
    #linear regression cost function
    #the dot (mat-mult) creates a horizontal vector
    #thus the y vector should be horizontal as well
    return sum( (y - dot(theta.T,x)) ** 2) / N

def linreg_df(x, y, theta):
    #simple gradient of the multivariable linear sum of sqrs cost function
    #applied for multivariable calcs.
    x = vstack( (ones((1, x.shape[1])), x))
    N = x.shape[1]
    return -2*sum((y-dot(theta.T, x))*x, 1) / N #dot(x, (dot(theta, x) - y).T)/N#

def logreg_f(x, y, theta):
    x = vstack( (ones((1, x.shape[1])), x))
    N = x.shape[1]
    h = 1.0 / (1.0 + exp(-1.0*dot(theta, x)) + 1e-10)
    return -1 * sum( (log(h) * y) + (1.0 - y)*log(1.0 - h) )

def logreg_df(x, y, theta):
    x = vstack( (ones((1, x.shape[1])), x))
    N = x.shape[1]
    h = 1.0 / (1.0 + exp(-1.0*dot(theta, x))) #horizontal
    return -1 * sum((y - h)*x, 1) #dot(x, (h.T - y)) / N

def simple_grad_descent(f, df, x, y, init_t, alpha, max_iter):
    EPS = 1e-9   #EPS = 10**(-5)
    prev_t = init_t -10*EPS
    t = init_t.copy()
    iter  = 0
    while norm(t - prev_t) >  EPS and iter < max_iter:
        prev_t = t.copy()
        t -= alpha*df(x, y, t)
        # if iter % 500 == 0:
        #     print (iter)
        #     print (f(x, y, t))
        #     print (df(x, y, t))
        iter += 1
    return t   #return optimal theta


def plot_line(ax, theta, x_min, x_max, color, label, linewidth=1, ls='-'):
    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)
    ax.plot(x_grid[1,:], y_grid, linestyle=ls, color=color, label=label, linewidth=linewidth)


def gen_lin_data_1d(theta, train_N, test_N, sigma_low, sigma_high, show_plot=True):

    random.seed(0)
```

```
#######################################################
# Set up Plots

fig, (ax1, ax2) = plt.subplots(2, sharex=True, sharey=True)

#######################################################
# Actual data
# 2D data points scattered in either halfspace of theta line

x_train_raw = 100*(random.random((train_N))-.5)
x_test_raw = 100*(random.random((test_N))-.5)

x_train = vstack((     ones_like(x_train_raw),
                x_train_raw,
                ))
x_test = vstack((     ones_like(x_test_raw),
                x_test_raw,
                ))

#indices of points with "large outputs" - i.e. far from the theta line
#10% of points have large outputs
scatter_train = bernoulli.rvs(0.05, size=train_N) #1 for no flip
scatter_test = bernoulli.rvs(0.05, size=test_N)

#95% of y points scattered closer to theta line
y_train = dot(theta, x_train) + (1.0-scatter_train)*scipy.stats.norm.rvs(
scale=sigma_low,size=train_N) + (scatter_train)*scipy.stats.norm.rvs(
scale=sigma_high,size=train_N)
y_test = dot(theta, x_test) + (1.0-scatter_test)*scipy.stats.norm.rvs(
scale=sigma_low,size=test_N) + (scatter_test)*scipy.stats.norm.rvs(
scale=sigma_high,size=test_N)

#######################################################
# Label generating process (Two labels: 0 or 1)
#

#threshold 0, assign label 1 or 0
t_train = 1  * ((y_train - dot(theta, x_train)) >= 0)
t_test = 1  * ((y_test - dot(theta, x_test)) >= 0)


X_train = vstack((x_train, y_train))
X_test = vstack((x_test, y_test))

if show_plot:
    plot_line(ax1, theta, -70, 70, 'orange', "Actual generating process",
    linewidth=3, ls=':')
    plot_line(ax2, theta, -70, 70, 'orange', "Actual generating process",
    linewidth=3, ls=':')

    positive_samples_train = X_train[:, where(t_train == 1)]
    negative_samples_train = X_train[:, where(t_train == 0)]
    positive_samples_test = X_test[:, where(t_test == 1)]
    negative_samples_test = X_test[:, where(t_test == 0)]

    ax1.plot(negative_samples_train[1,:], negative_samples_train[2,:], "ro")
```

```
    ax1.plot(positive_samples_train[1,:], positive_samples_train[2,:], "go")
    ax2.plot(negative_samples_test[1,:], negative_samples_test[2,:], "ro")
    ax2.plot(positive_samples_test[1,:], positive_samples_test[2,:], "go")


#########################################################
# Least squares solution
#
theta0 = array([0.0, 0.0, 0.0])
theta_lin = simple_grad_descent(linreg_f, linreg_df, X_train[1:,:], t_train,
theta0, 0.00010, 100000)

if show_plot:
    lin_t = array([(0.5 - theta_lin[0])/theta_lin[2], -1*theta_lin[1]/theta_lin[2]])
    plot_line(ax1, lin_t, -70, 70, "k", "Linear Regression Soln")
    plot_line(ax2, lin_t, -70, 70, "k", "Linear Regression Soln")

#########################################################
# Logistic Regression solution
#

theta_log = simple_grad_descent(logreg_f, logreg_df, X_train[1:,:], t_train,
theta0, 0.000010, 10000)
if show_plot:
    log_t = array([(0.5 - theta_log[0])/theta_log[2], -1*theta_log[1]/theta_log[2]])
    plot_line(ax1, log_t, -70, 70, "b", "Logistic Regression Soln")
    plot_line(ax2, log_t, -70, 70, "b", "Logistic Regression Soln")
#print(logreg_f(X[1:,:], t, theta_log))

#########################################################
# Classification Results
#
result_lin = 1 * (dot(theta_lin,X_test) >= 0.5)
result_log = 1 * (dot(theta_log,X_test) >= 0.5)

classif_lin = asarray([1 if result_lin[i] == t_test[i] else 0 for i in range(len(result_lin))]
classif_log = asarray([1 if result_log[i] == t_test[i] else 0 for i in range(len(result_lin))]

if show_plot:
    ax1.plot(negative_samples_train[1,0], negative_samples_train[2,0], "ro", label="Negative")
    ax1.plot(positive_samples_train[1,0], positive_samples_train[2,0], "go", label="Positive")
    negative_samples_lin = X_test[:, where(classif_lin == 0)]
    negative_samples_log = X_test[:, where(classif_log == 0)]
    ax2.plot(negative_samples_lin[1,:] + 1e-1, negative_samples_lin[2,:] + 1e-1, "ko", mfc='no
    ax2.plot(negative_samples_lin[1,0] + 1e-1, negative_samples_lin[2,0] + 1e-1, "ko", mfc='no
    ax2.plot(negative_samples_log[1,:] - 1e-1, negative_samples_log[2,:] - 1e-1, "bo", mfc='no
    ax2.plot(negative_samples_log[1,0] + 1e-1, negative_samples_log[2,0] + 1e-1, "bo", mfc='no

    ax1.set_title('Training Set')
    ax2.set_title('Test Set')

    # Adds the legend with some customizations.
    legend1 = ax1.legend(bbox_to_anchor=(1.05, 1), loc=2, shadow=True)
    legend2 = ax2.legend(bbox_to_anchor=(1.05, 1), loc=2, shadow=True)
```

```
    frame1 = legend1.get_frame()
    frame1.set_facecolor('0.90')
    frame2 = legend2.get_frame()
    frame2.set_facecolor('0.90')

    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()
    savefig('part5_experiment', bbox_inches='tight')


lin_hits = sum(classif_lin)
log_hits = sum(classif_log)

print("\n----------------------------------------------------------------")
print("Results for Part 5 experiment (2 labels for the target 0/1):")
print("Experiment of training set size " + str(train_N) + " and test set size " + str(test_N)
print("Linear Regression correctly classified " + str(lin_hits/float(test_N) * 100) + "% of te
print("Logistic Regression correctly classified " + str(log_hits/float(test_N) * 100) + "% of
```

Part 6

Assumptions & notation:
Comparison is for a single sample in the training set.
Let l denote the layer of the network.
Let j denote the index of the neuron in layer l.
Let i denote the index of the neuron in layer $l-1$ (the input neurons for layer l).
Let $h_{l,j}$ refer to a hidden unit in layer l, at index j, except for the output units, denoted o_j
Let $w_{l,i,j}$ refer to a weight in layer l connecting unit $h_{l,j}$ in layer l to $h_{l-1,i}$ in layer $l-1$.
Let x_i denote the i^{th} input unit.
We assume for this question that the cost model is the same as in Part 3, where each output unit o_j is transformed via softmax (p_j) and the cost function C is log-loss (i.e. the set up for multinomial logistic regression). We assume each hidden unit is computed with some activation function:

$$h_{l,j} = \sigma(\sum_i w_{l,i,j} h_{l-1,i} + b_{l,j})$$

For the vectorization, let W_l denote the weight matrix of layer l of dimension $k \times k$, the entry of row i, column j corresponding to $w_{l,i,j}$ as above.
Let H_l denote the vector of hidden units of dimension $k \times 1$, a single entry is referred to as $h_{l,j}$ as above.
Let X denote the vector of input units of unknown dimension $m \times 1$. Let O denote the vector of output units (dimension $k \times 1$).

We compare runtimes in terms of the number of multiplication operations.
In Backpropagation, we do 2 computations per layer l:
$\frac{\partial C}{\partial W_l}$, the partial derivatives of weight matrix of layer l.
$\frac{\partial C}{\partial H_{l-1}}$, the partials of hidden units which are input to layer l, in preparation for calculating $\frac{\partial C}{\partial W_{l-1}}$.
The two exceptions to the general computation framework are the output and input network layer, which we express calculations for separately.

1. Output Layer

$$\frac{\partial C}{\partial w_{n,i,j}} = \frac{\partial C}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{n,i,j}} = (p_j - y_j) \cdot h_{n-1,i}$$

Vectorized:

$$\frac{\partial C}{\partial W_n} = (H_{n-1})(P-Y)^T$$

H, P, and Y are $k \times 1$. Total multiplications: k^2.

$$\frac{\partial C}{\partial h_{n-1,i}} = \sum_j \frac{\partial C}{\partial o_j} \cdot \frac{\partial o_j}{\partial h_{n-1,i}} = \sum_j \frac{\partial C}{\partial o_j} \cdot w_{n,i,j}$$

Vectorized:

$$\frac{\partial C}{\partial H_{n-1}} = W_n \Big[\frac{\partial C}{\partial O}\Big]$$

W_n is $k \times k$. $\frac{\partial C}{\partial O}$ is $k \times 1$. Total multiplications: k^2.
Total multiplications in the output layer: $2k^2$.

2. Input Layer
We need only the partials of the weights in this layer.

$$\frac{\partial C}{\partial w_{1,i,j}} = \frac{\partial C}{\partial h_{1,i}} \cdot \frac{\partial h_{1,i}}{\partial w_{1,i,j}} = \frac{\partial C}{\partial h_{1,i}} \cdot x_i$$

Vectorized:

$$\frac{\partial C}{\partial W_1} = X\Big[\frac{\partial C}{\partial H_1}\Big]^T$$

X is $m \times 1$. Total multiplications in the input layer: mk.

3. General Layer

$$\frac{\partial C}{\partial w_{l,i,j}} = \frac{\partial C}{\partial h_{l,j}} \cdot \frac{\partial h_{i,j}}{\partial w_{l,i,j}} = \frac{\partial C}{\partial h_{l,j}} \cdot h_{l-1,i}$$

Vectorized:

$$\frac{\partial C}{\partial W_l} = (H_{l-1})\Big[\frac{\partial C}{\partial H_l}\Big]^T$$

H_l, $\frac{\partial C}{\partial H_l}$ are $k \times 1$. Total multiplications: k^2.

$$\frac{\partial C}{\partial h_{l-1,i}} = \sum_j \frac{\partial C}{\partial h_{l,j}} \cdot \frac{\partial h_{l,j}}{\partial h_{l-1,i}} = \sum_j \frac{\partial C}{\partial h_{l,j}} \cdot w_{l,i,j}$$

Vectorized:

$$\frac{\partial C}{\partial H_{l-1}} = W_l\Big[\frac{\partial C}{\partial H_l}\Big]$$

W_l is $k \times k$. $\frac{\partial C}{\partial H_l}$ is $k \times 1$. Total multiplications: k^2.
Total multiplications in a general layer: $2k^2$.

In total, we have $2k^2(n-1) + mk$ multiplications for Backpropagation, or $\mathcal{O}(k^2 n)$. This is polynomial time with respect to n, the number of layers in the network.

If we don't use Backpropagation or vectorized operations, our computation time is as follows:
The derivative of a single weight $w_{l,i,j}$ is expressed as:

$$\frac{\partial C}{\partial w_{l,i,j}} = \frac{\partial C}{\partial h_{l,j}} \cdot \frac{\partial h_{i,j}}{\partial w_{l,i,j}} = \frac{\partial C}{\partial h_{l,j}} \cdot h_{l-1,i}$$

We need to re-derive the partial of the hidden unit $\frac{\partial C}{\partial h_{l,j}}$. There are two cases:

1. $w_{n,i,j}$ is a weight in the output layer.
   Then, $\frac{\partial C}{\partial o_j}$ does not depend on any layer above. There are $k \times k$ weights $w_{n,,,}$ in this layer, one multiplication operation per each. Thus k^2 operations in the output layer.

2. $w_{l,i,j}$ is a weight in any other layer.
   Computing $\frac{\partial C}{\partial h_{l,j}}$ requires $k \cdot (k^2)^{n-i-1}$ multiplication operations.
   Computing $\frac{\partial C}{\partial w_{l,i,j}} = \frac{\partial C}{\partial h_{l,j}} \cdot h_{l-1,i}$ requires $k \cdot (k^2)^{n-i-1} + 1$ multiplication operations.
   There are k^2 weights in every fully connected layer, thus $k \cdot (k^2)^{n-i} + k^2$ operations per layer.

In conclusion, if we compute the gradient with respect to each weight individually without Backpropogation, the total number of multiplication operations for all the weights is:

$$= k^2 + (k + k^2) + (k(k^2) + k^2) + (k(k^2)^2 + k^2) + \ldots + (k(k^2)^{n-2} + k^2)$$

$$= nk^2 + k\sum_{i=0}^{n} k^2 \in \mathcal{O}(k^{2n})$$

The runtime is exponential, compared to the polynomial runtime with Backpropogation.

Part 7

To preprocess the input, the images are downloaded similarly to Project 1 and the hash of each image is checked. The images are cropped to their bounding box, and are grayscaled and resized to 64×64. The dataset is randomly shuffled and split into the three sets, training, test, and validation, in the following way:

- 75 images per actor in the training set.
- 15 images per actor in the validation set.
- 30 images per actor in the test set.

The sets were saved to a dictionary in the same format as the M dictionary loaded from mnist_all.mat containing the digits datasets. For example, the training set for actor 6 (harmon) is stored in the dictionary under the key 'train5'. Each image was flattened to a 1×4096 NumPy array, and the images are normalized to the range [0, 1] by dividing each pixel value by 255.0.

The architecture of the network is as follows: the input layer is of size 4096, same as the size of each image. The single hidden layer has 300 units, and the activation function of the hidden units is tanh. The output layer has 6 units, each of which is passed through the softmax function. Each layer is fully connected. The cost model is negative log loss (cross-entropy). The weights were initialized to random values to prevent symmetrical updates in the gradient descent procedure. They were assigned random values using a normal distribution with a standard deviation of 0.01 and mean 0. Mini-batch gradient descent was used with a batch size of 50 training images. Mini-batch helps speed up the runtime of the descent algorithm by using a smaller training set to compute gradient updates on with each iteration, and also helps the algorithm break out of sub-optimal local minima of the cost surface.

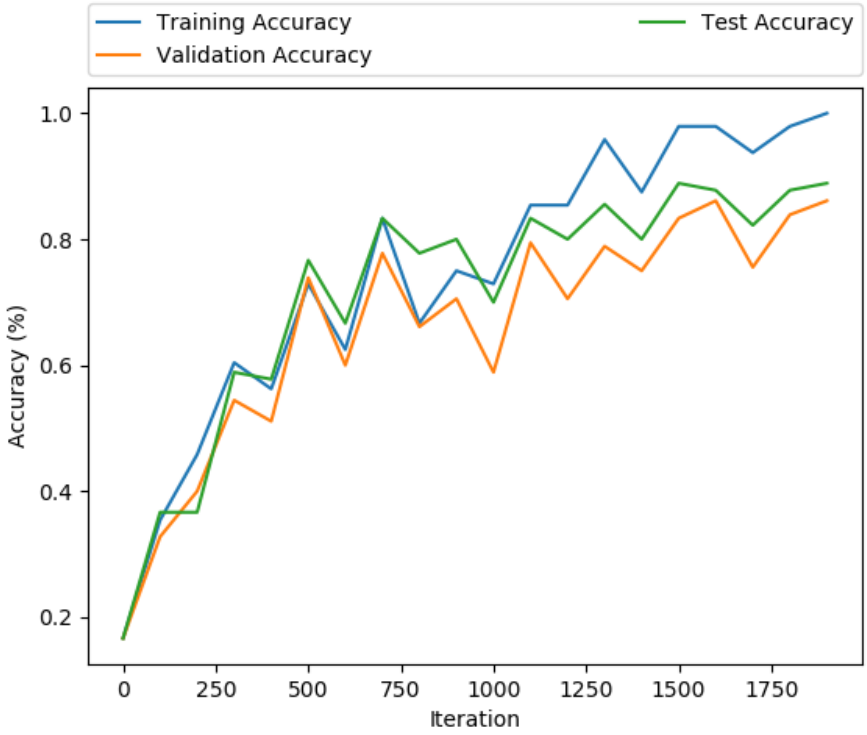
The Mini-batch gradient descent parameters are as follows:

- alpha = 0.05
- max_iter = 1000
- decay penalty parameter = 0.0

The final performance on the test set is 90.0***%.***

```
================== RUNNING PART 7 ===================
i= 0
i= 200
i= 400
i= 600
i= 800
i= 1000
i= 1200
i= 1400
i= 1600
i= 1800
The final performance on the training set is:  0.991111
The final validation set accuracy is:  0.872222
The final performance on the test set is:  0.9
```

Figure 2: Part 7 Learning curves



Part 8

To demonstrate the positive effects of using regularization on the test performance, the training set was modified to contain 70% noise. A distribution of locations of noisy pixels is chosen and fixed for all training images. Then, random noise (values from [0, 255]) is generated at these pixel locations for each image. The idea of this experiment is that without regularization, the chance of overfitting to random noisy pixels in the training set is higher, thus the performance on the test and validation sets with no noise applied will be lower. The ideal weight parameters in the first layer would be zero where they are connected to the noisy pixels since they contain no useful information with regards to classification, and maximized for pixels or groups of pixels which represent genuine features in the image that are the most useful for identifying each class of actor. The motivation for regularization is prevent overfitting to the noise by keeping the weights low, thus the descent algorithm becomes more selective of the pixel groupings for which to raise the weights, and which to disregard.
For this experiment I found that introducing a large amount of noise in every training image (70%) produced a greater effect in performance with regularization. The architecture of the network for this experiment follows Part 7, except the number of hidden units is decreased to 150.

The optimal mini-batch gradient descent parameters are as follows:

- alpha = 0.05
- max_iter = 2000
- decay penalty parameter = 0.23

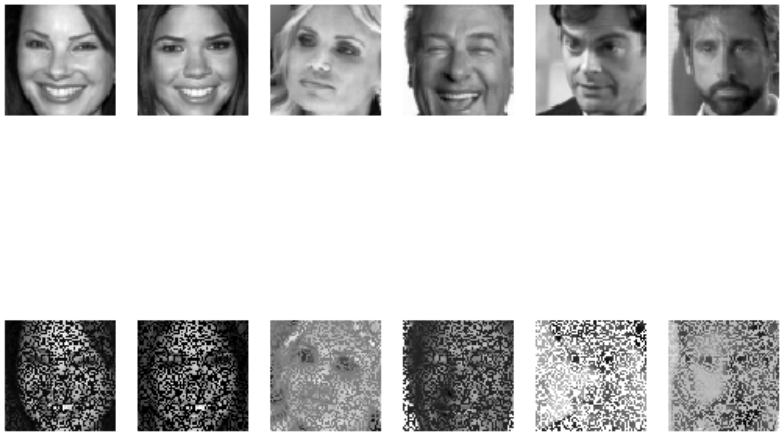
With the regularization parameter set to 0.23 the performance on the test set increased from 72.67% to 84.0%, a difference of 11.33%.

The code to generate the noise in the training set, a sample of the noisy generated images from each actor class, and the printed output result from the experiment is listed below:

```
def apply_noise_to_training(M, display=False):
    train_k =  ["train"+str(i) for i in range(N_LABELS)]

    im = M[train_k[0]][0, :]

    amount = 0.7
    num_salt = np.ceil(amount * im.size)
    coords = [np.random.randint(0, i - 1, int(num_salt)) for i in im.shape]

    for k in range(N_LABELS):
        for i in range(M[train_k[k]].shape[0]):
            image = M[train_k[k]][i, :]
            values = [np.random.randint(0, 256) for i in image.shape]
            M[train_k[k]][i, :][coords] = values

    if display:
        M1 = loadmat("faces.mat")
        display_training(M1, M)
    return M
```

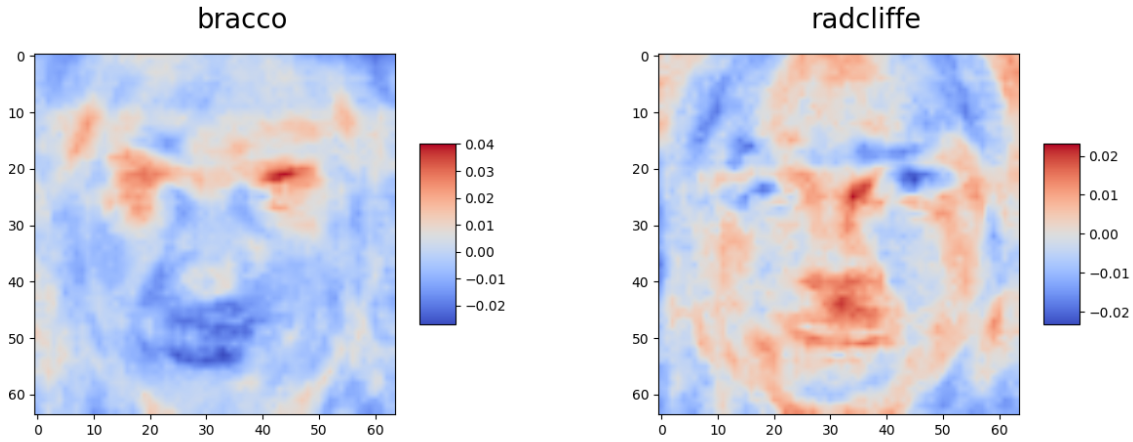
```
================== RUNNING PART 8 ==================
Modifying the Training Set to contain 70% noise
Result of Training for 2000 iterations with NO regularization (lam=0.00)
i= 0
i= 200
i= 400
i= 600
i= 800
i= 1000
i= 1200
i= 1400
i= 1600
i= 1800
The final performance on the training set is:  0.811905
The final validation set accuracy is:  0.686667
The final performance on the test set is:  0.726667
Result of Training for 2000 iterations WITH regularization (lam=0.23)
i= 0
i= 200
i= 400
i= 600
i= 800
i= 1000
i= 1200
i= 1400
i= 1600
i= 1800
The final performance on the training set is:  0.852381
The final validation set accuracy is:  0.78
The final performance on the test set is:  0.84
```

Figure 3: Part 8 Noisy Training Set image samples



Part 9

Figure 4: Images of weights of most interesting Hidden units for 2 actors



From Part 7, I saved the $W0$, $b0$ parameter values in a Pickle file to reuse in Part 9. Then I did a forward pass of the network up to only the Hidden layer with the entire Training set consisting of 70 images per 6 actors. From the forward pass, I saved the values of the tanh activations of the 300 Hidden units in an matrix, one row per training sample. To be able to highlight the difference in activations per training image, the activations for each neuron across the entire dataset are normalized (each column of the matrix). This is because it is possible that some neurons may fire high for any input, and when just comparing absolute magnitudes of activations, these neurons may be falsely flagged as important. Thus, the activations of a single neuron across the dataset are normalized by their mean and standard devation. This is done for each of the 300 neurons.

After normalizing, activations for the training sample belonging to a single actor class are observed. For each training sample (one row of the activation matrix), the neuron with the highest activation value is recorded. The neuron with the greatest frequency of highest value amongst all training samples for the actor is taken as the most sensitive/interesting hidden unit, and weights connected to that unit are displayed as an image.

Part 10

The images for this part are the same images from Part 7, 8, 9 with the exception that they are RGB and they are cropped and resized to $227 \times 227 \times 3$ dimension. The images are normalized by their mean value, and are flipped to BGR as in the example with the Laska image. The image dataset is split in the same way as in Part 7; 70, 15, 30 for the training, validation, and test set respectively. Each set is fed forward into the unmodified AlexNet network using the trained parameters provided in `bvlc_alexnet.npy` up to the conv4 activations, where is it extracted. As a result of feeding it through the network, each sample is transformed to $13 \times 13 \times 384$ dimension.
A fully connected single layer network is trained on the conv4 activation inputs. The training, validation and test sets composed of conv4 activations are maintained for this next part. The input layer of the network consists of $13 \times 13 \times 384 = 64896$ units (matching the size of the AlexNet output). The output layer consists of 6 units that are fully connected to the 64896 input units. The output layer is passed into a softmax. The softmax output is used to calculate the negative log loss cost.
The weights were initialized to random values to prevent symmetry in the gradient descent procedure. They were assigned random values using a normal distribution with a standard deviation of 0.01 and mean 0. Mini-batch gradient descent was used with a batch size of 50 training images for runtime efficiency and to break out sub-optimal local minima of the cost surface.

The training parameters of the fully connected output layer are as follows:

- alpha = 0.005
- lam = 0.0085 (L2 weight penalty coefficient)
- iterations = 3000

The performance accuracy on the Test set in Part 7 was 90.0%. Thus a 30% improvement on the error rate is calculated to be $90 + ((100 - 90) * 0.3) = 93.0\%$. The network with the conv4 AlexNet outputs improves on the performance of the Test set in Part 7 by 44.44%, with a final Test set performance accuracy of 94.44%.

```
The final performance on the training set is:  1.0
The final validation set accuracy is:  0.9
The final performance on the test set is:  0.944444
```