CSC411: Project #2

Due on February 19, 2017

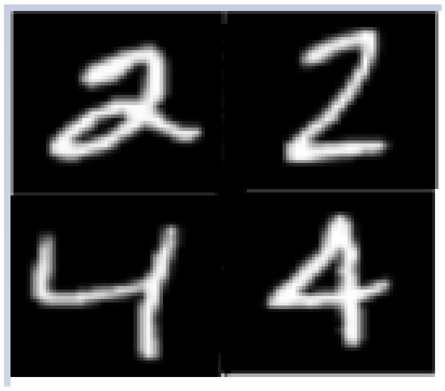
Angus Fung, Oleg Milyutin

March 17, 2018

Part 1

The following few pages contain 10 sample images of each digit in the MNIST dataset. In order to obtain an unbiased and accurate representation of the dataset, the images were randomly selected and displayed. Immediately apparent from the dataset are that some of the numbers appear in slanted orientations, which provide flexibility on testing sets in which the numbers are not perfectly oriented.

Furthermore, the dataset contains different notational variations of the same number. For example, in Fig. 1a, the number two is written with a swirl in the first image and without in the second image. Overall, the dataset seems accurate in that of the 100 images that were randomly chosen, there weren't any erroneous (e.g misplaced) images. However, not all images are in the best quality, such as the second image in Fig. 1b. The "seven" seems slightly faded away in parts. In addition, some images themselves are barely legible, such as the first image in Fig. 1a, which could be misconstrued as a "one" if there wasn't a slash through the number.


(a) Notational Variations

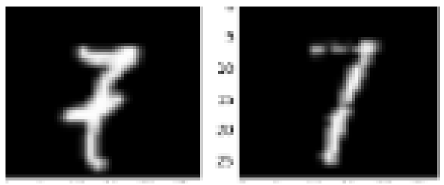
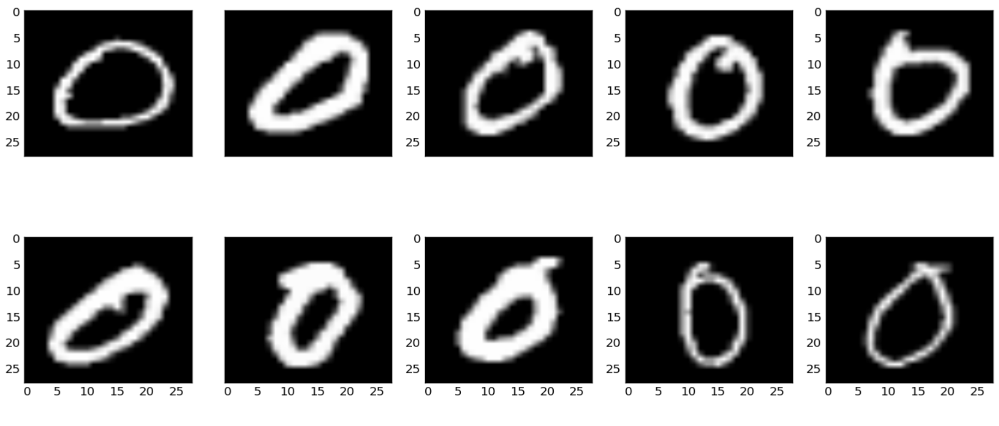
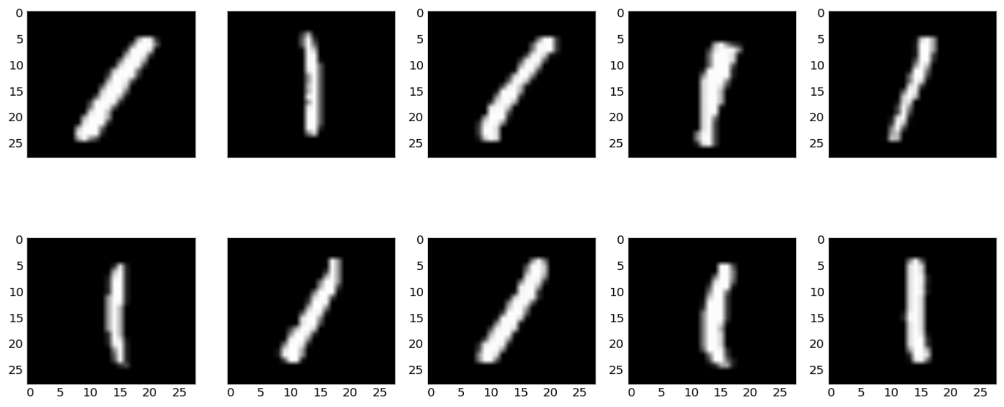
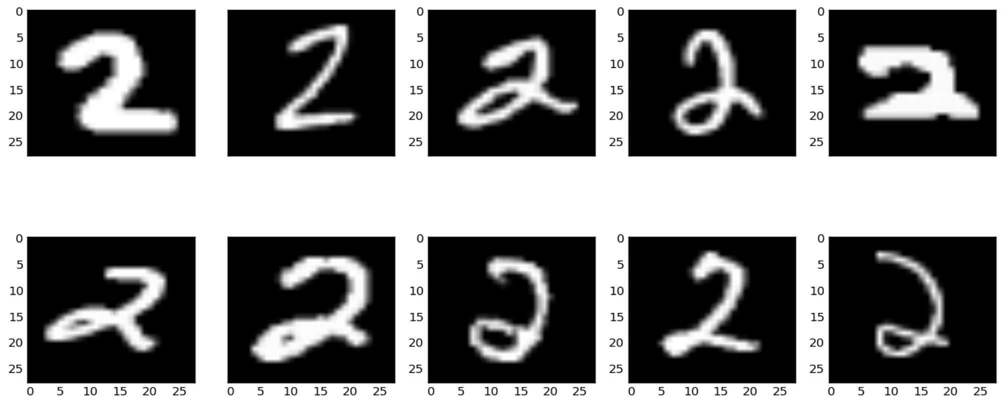
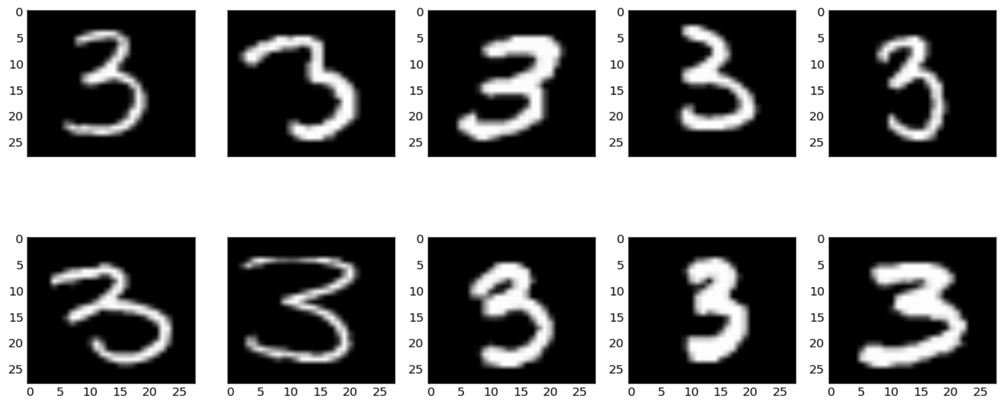

(b) Notational Variations

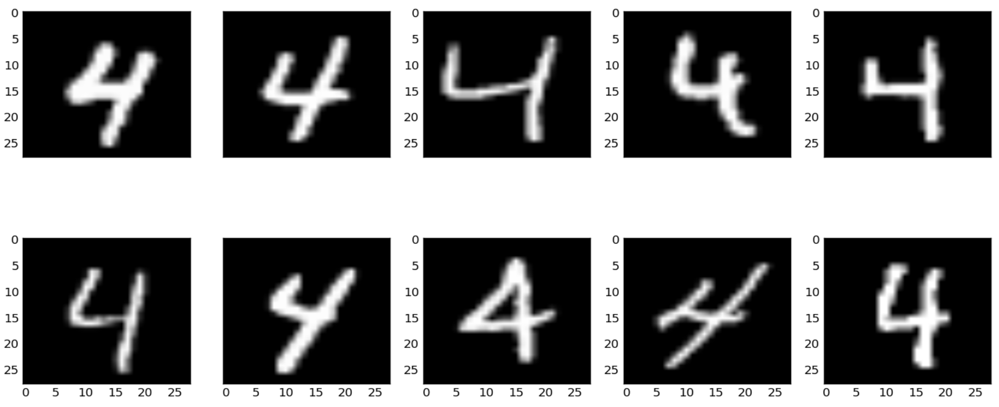
Figure 1: Description of dataset


(a) Zeros

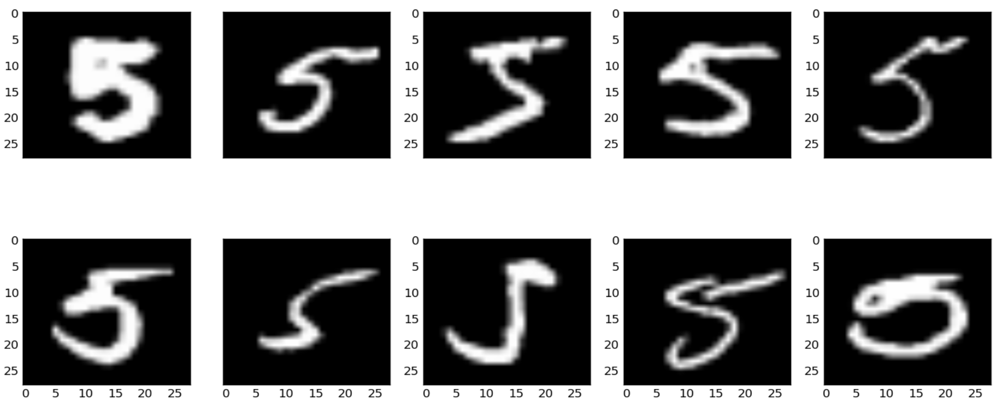

(a) Ones


(b) Twos

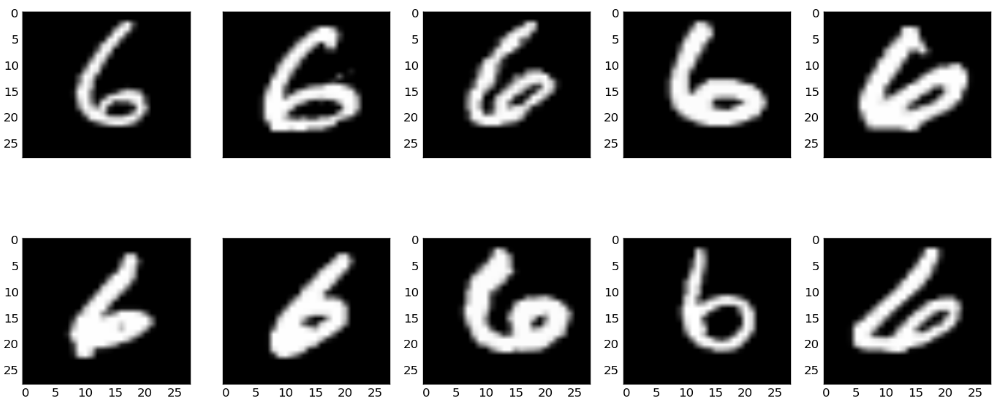

(c) Threes



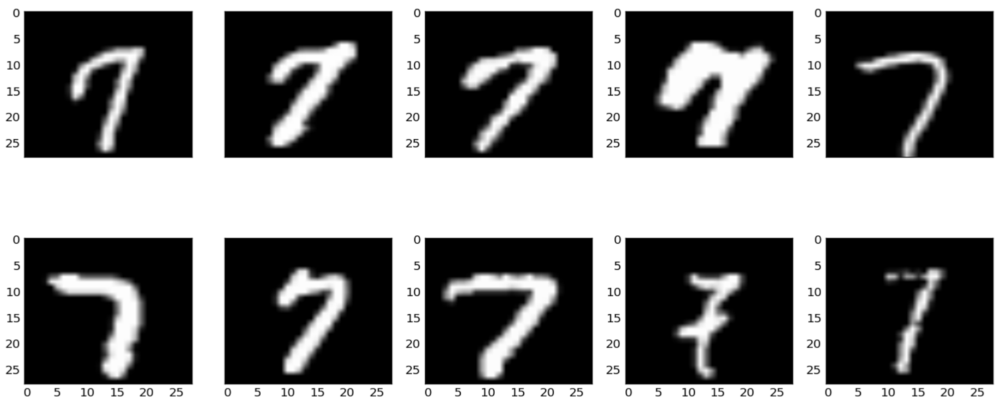
(a) Fours



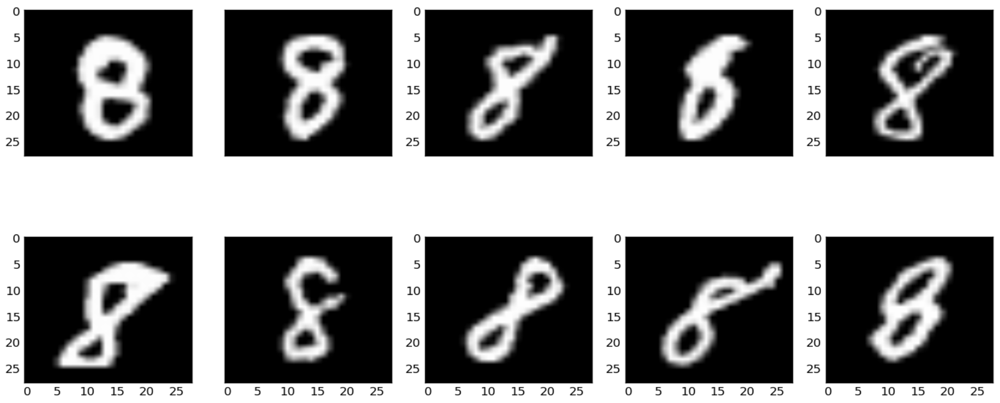
(b) Fives



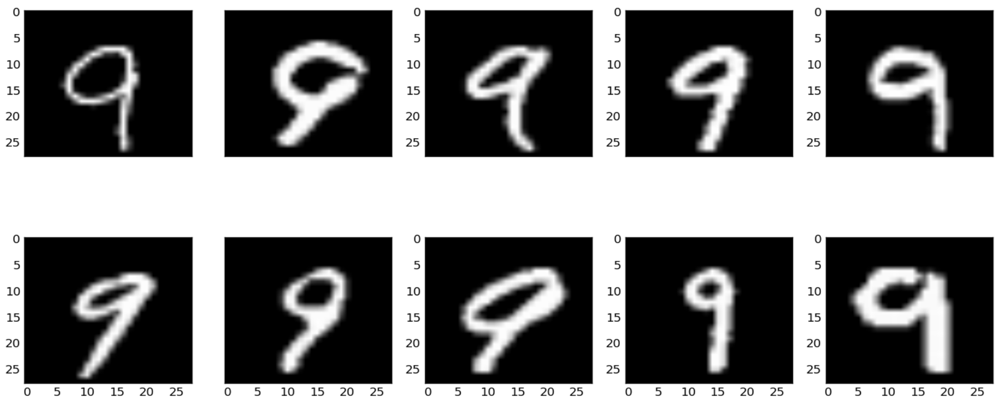
(c) Sixes



(a) Sevens



(b) Eights



(c) Nines

Part 2

Code

```
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 part2(x,weights):
    '''Return a matrix of probabiltiies, the output from the cost function
       bias is the first row of the weight matrix (785 x 10)
    '''
    x = x/255.0
    x = vstack( (ones((1, x.shape[1])), x)) #include x_0
    y = dot(weights.T, x) # + bias
    return softmax(y)
```

Above is an implementation of a softmax network with no hidden layers and identity as the activation function. The computation of $o_i = \sum_j w_{ji}x_j + b_i$ was vectorized so that the outputs of all the images could be done in one matrix multiplication. In particular, $Y = W^T X$, where W is the weight matrix of size 785 x 10 (the extra row vector is to include the bias term), X is the input image matrix of size 785 x m, where m is the size of the training set. Similarly, X is approximately normalized by division of 255.0 and includes a row of $x_0 = 1$ terms. The output Y is 10 by m, where 10 is the number of outputs for a particular image.

Part 3

Compute the gradient with respect to the weight w_{ij}

To compute the gradient, it suffices to examine the case for one training set, and thereafter sum the contributions of each training set to obtain the overall cost function. The interpretation of w_{ij} will be that i corresponds to x_i, the ith pixel of one particular training example and j, the jth output of the network. Therefore,

Proof.

$$\frac{\partial C}{\partial w_{ij}} = \sum_k \left(\frac{\partial C}{\partial p_k} \frac{\partial p_k}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}} \right), \tag{1}$$

where k sums over the outputs of the network. Examining each partial derivative separately:

$$\begin{aligned} \frac{\partial C}{\partial p_k} &= \frac{\partial}{\partial p_k} \left(- \sum_j y_j \log p_j \right) \\ &= -\frac{y_k}{p_k}, \end{aligned}$$

where $j \in \{1, ..., 784\}$ sums over the pixels of a particular image.

$$\begin{aligned} \frac{\partial p_k}{\partial o_j} &= \frac{\partial}{\partial o_j} \frac{e^{o_k}}{\sum_{j'} e^{o_{j'}}} \\ &\overset{*}{=} \begin{cases} p_j(1-p_j) & k = j \\ -p_k p_j & o.w \end{cases} \end{aligned}$$

where * denotes that the quotient rule was used.

$$\begin{aligned} \frac{\partial o_j}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \left(\sum_k w_{kj} x_k + b_j \right) \\ &= x_i \end{aligned}$$

Finally, combining everything together:

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}} &= \left(-\frac{y_0}{p_0}(-p_0 p_j) - \frac{y_1}{p_1}(-p_1 p_j) - ... - \frac{y_j}{p_j}(-p_j p_j) - ... - \frac{y_9}{p_9}(-p_9 p_j) \right) x_i \\ &= (p_j \underbrace{(y_1 + y_2 + ... + y_9)}_{=1 \text{ (One-Hot-Encoding)}} - y_j) x_i \end{aligned}$$

Summing over all the training sets:

$$\frac{\partial C}{\partial w_{ij}} = \sum_{k=1}^{m} \left(p_j^{(k)} - y_j^{(k)} \right) x_i^{(k)} \tag{2}$$

■

Finite Differences

Code

```
def f(x,y,weights):
    '''@input prob is a NxM matrix of probabilities
       @input y is a NxM matrix where each column is the one-hot-representation
              of an image
              N is the number of outputs for a single case, M is the number of
              cases (e.g images)
       @input weights is a 785 x 10 matrix
              '''
    prob=part2(x,weights)
    return -sum(y * log(prob))
def df(x, y, weights):

    prob=part2(x,weights)
    x=x/255.0
    x = vstack( (ones((1, x.shape[1])), x))
    return dot(x,(prob-y).T)
```

Recall, m is the number of training examples, n is the number of outputs in the network, and:

$$\mathbf{X} \in \mathbb{R}^{784\times m}$$
$$\mathbf{Y} \in \mathbb{R}^{10\times m}$$
$$\mathbf{P} \in \mathbb{R}^{10\times m}$$
$$\mathbf{W} \in \mathbb{R}^{784\times n}$$

Note: the above dimensions do not include the bias terms, etc.

To implement the cost function and its derivative, a few formulas were used:

Cost Function:

$$C = -\sum_{k=1}^{m}\sum_{j=1}^{n} y_j^{(k)} \log p_j^{(k)} \tag{3}$$

Vectorized Python Cost Function:

$$C = -\sum_{ij}(\mathbf{Y}\cdot\log\mathbf{P})_{ij}, \tag{4}$$

which sums all the elements in the matrix, and $\cdot$ denotes an element wise multiplication of the matrices.
Vectorized Gradient:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}} = \mathbf{X}(\mathbf{P}-\mathbf{Y})^T, \tag{5}$$

which can be proven analogously to the gradient function in Project 1.

```
run_part3 = False #finite differences
def part3():
    #initializing random variables to test
    m=200
    random.seed(0)
    x = reshape(random.rand(784*m), (784,m))
    random.seed(1)
    y=zeros((10,m)) #one-hot encoding matrix
    y[0,:]=1
    random.seed(2)
    weights = reshape(random.rand(785*10), (785,10))
    h = 0.00000001

    dh = zeros((785,10))
    dh[0,0]=h

    print("Finite difference approximation at (1,1)")
    print (f(x,y, weights+dh) - f(x,y, weights-dh))/(2*h)
    print df(x, y, weights)

    dh[0,0]=0
    dh[0,1]=h

    print("Finite difference approximation at (1,2)")
    print (f(x,y, weights+dh) - f(x,y, weights-dh))/(2*h)
    print df(x, y, weights)
    return
if run_part3:
    part3()
```

```
>>> part3()
Finite difference approximation at (1,1)
-178.618506652
[[ -1.78618509e+02   1.41366475e+01   2.40829494e+01 ...,   2.60508933e+01
    1.81162131e+01   1.83085535e+01]
 [ -3.50509502e-01   2.77476541e-02   4.72436648e-02 ...,   5.11430798e-02
    3.55464081e-02   3.59132698e-02]
 [ -3.44066486e-01   2.72160204e-02   4.64189153e-02 ...,   5.01883875e-02
    3.49169351e-02   3.52808330e-02]
 ...,

Finite difference approximation at (1,2)
14.1366456319
[[ -1.78618509e+02   1.41366475e+01   2.40829494e+01 ...,   2.60508933e+01
    1.81162131e+01   1.83085535e+01]
 [ -3.50509502e-01   2.77476541e-02   4.72436648e-02 ...,   5.11430798e-02
    3.55464081e-02   3.59132698e-02]
 [ -3.44066486e-01   2.72160204e-02   4.64189153e-02 ...,   5.01883875e-02
    3.49169351e-02   3.52808330e-02]
 ...,
```

Part 4

Fig. 1 displays the performance vs. iteration of a training set consisting of 60000 total images (roughly 6000 per digit). The graph is plotted *per 100 iterations*, which at the expense of time complexity, results in a very smooth curve. Some immediate observations are that the performance of the test set performs *better* than the training set, and that the learning curve is steepest in the beginning phases before leveling off. With $\alpha = 10^{-7}$ and 3000 iterations, the final performance for the training and test sets were 87.5 and 88.3%, respectively. Increasing α to 10^{-6} and running it for 5000 iterations yields performances of 91.7% and 92%, respectively. Ultimately, the graph was obtained under the former conditions since a smaller α and not-to-large[1] an iteration size would cause the program to learn slower and illustrate its development phase better.

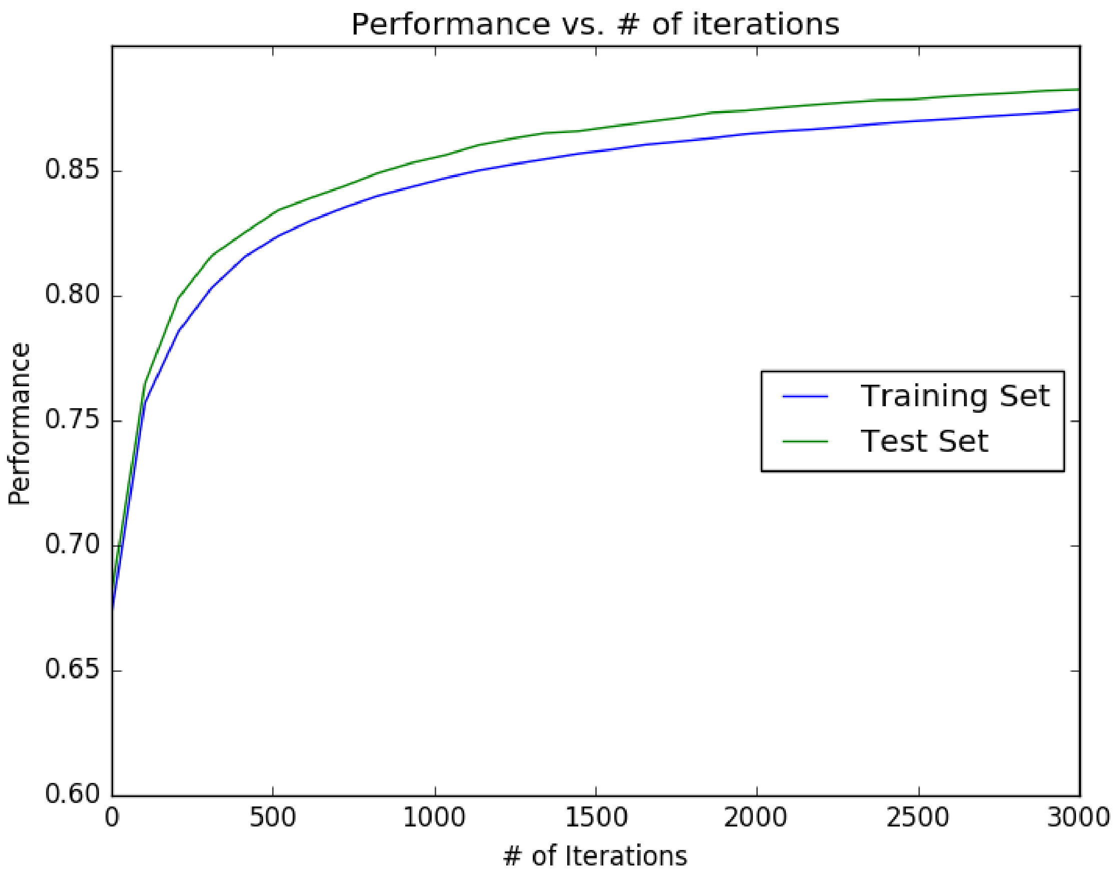


Figure 6: Performance vs. Iteration

As always, the optimization procedure required four parameters to be varied: (1) α, (2) initial conditions, (3) ϵ, the threshold, and (4) the number of iterations. As was mentioned earlier, $\alpha = 10^{-6}$ was chosen over $\alpha = 10^{-7}$ as a smaller convergence rate was preferred to demonstrate learning. The initial condition was chosen arbitrary to be the zero matrix, although any array of small weights would suffice. The threshold was chosen to be $\epsilon = 10^{-10}$. Finally, the number of iterations was determined through trial and error and 3000 seemed reasonable enough to demonstrate the full learning cycle, without too much over-fitting.

[1]an iteration size that demonstrates reasonable change in function behaviour over time

Below are the visualization of the weights going into each of the output units; one for each digit.

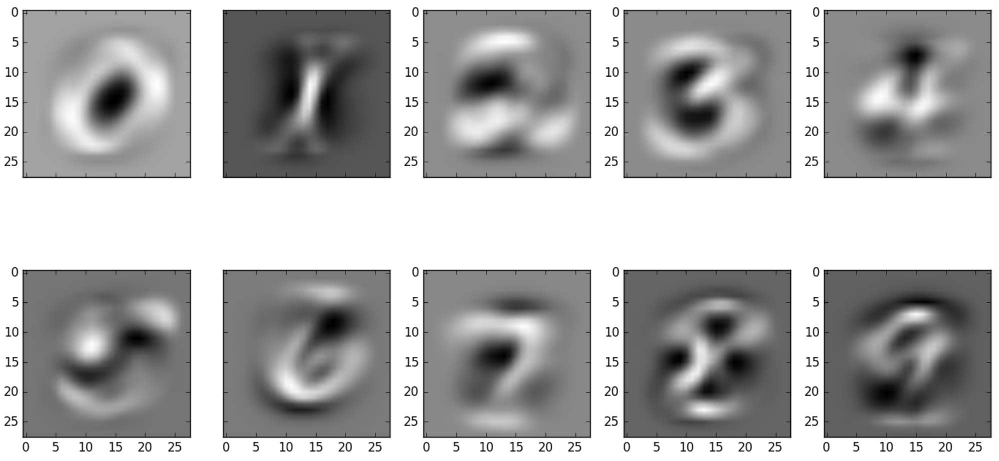


Figure 7: Visualizing the weights

Part 5

As it was done in lectures, training data was generated using the normal distribution function. The classification is binary so the objective here is to find the correct prediction horizon.
A key that was feature used in the generation is a **large** standard deviation ($\sigma = 80$, N = 100). This way, training set contained a significant numbers of outliers. We expect to see the solutions for least square method being less accurate compared to the logistic regression solution. As it was noted in the handout, logistic regression forces single training points not to adjust the weights too much. Thus, outliers can't affect the accuracy too much.
The prediction lines solved by both methods were plotted together along with the actual line that was used for generation. The actual line served its purpose as the decision boundary, where points above the line are classified as $y = 1$, and points below the line are classified as $y = 0$. Using both θ_{mle} and θ_{log}, a performance on the test set was generated with a 93% accuracy on the MLE solution and a 100% accuracy on the logistic regression solution. The 7% discrepancy confirms that logistic regression performs better when there are outliers in the dataset.

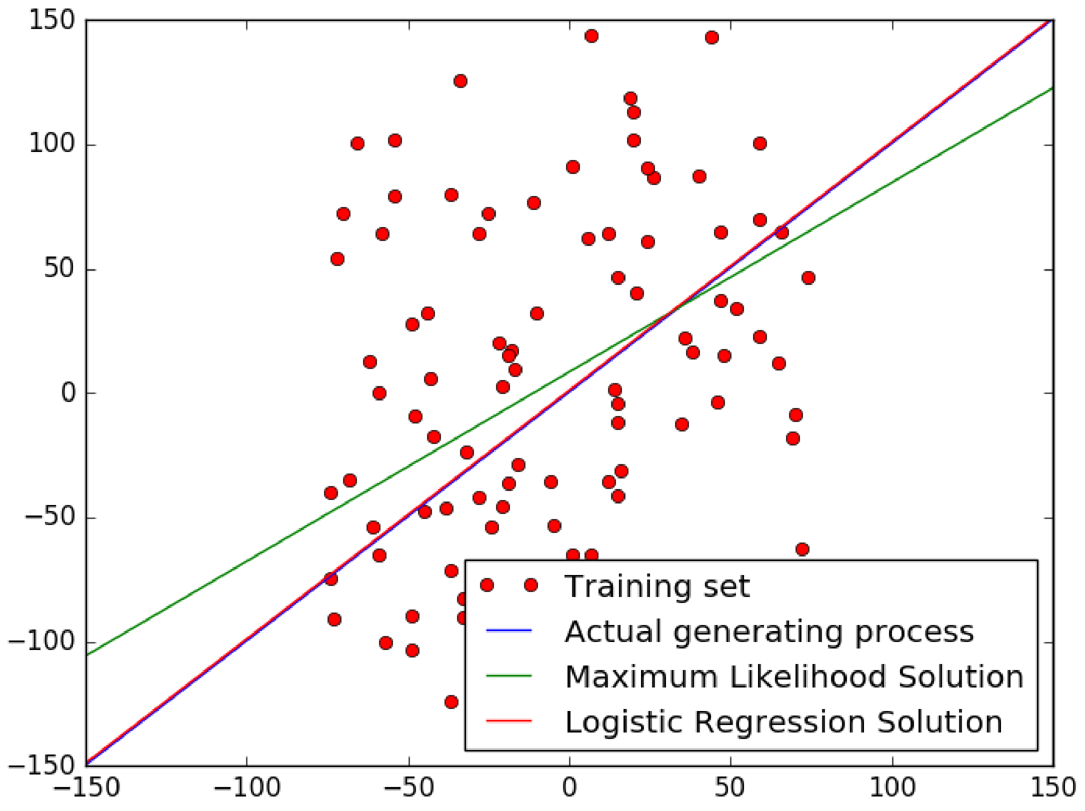


Figure 8: Logistic regression and Least squares methods comparison

The code for this part of the project is given on the next page. Note that due to the lack of space, some comments were omitted. For the full version, please refer to the digits.py file submitted.

```
def part5():
    N = 100
    sigma = 80
    theta = array([0.3, 1])
    gen_lin_data_1d(theta, N, sigma)
    show()
    return

def gen_lin_data_1d(theta, N, sigma):
    # Data limits
    x_limit = 150
    y_limit = 150

    # Actual data generation
    random.seed(N)
    x_train = rint(x_limit * (random.random((N)) - .5)).astype(int64)
    x_test = rint(x_limit * (random.random((N)) - .5)).astype(int64)
    x = vstack((ones_like(x_train), x_train,))
    x_test = vstack((ones_like(x_test), x_test,))
    y = dot(theta, x) + norm.rvs(scale=sigma, size=N)
    y_test = dot(theta, x_test) + norm.rvs(scale=sigma, size=N)

    plot(x[1, :], y, "ro", label="Training set")

    # Actual generating process
    plot_line(theta, -x_limit, x_limit, "b", "Actual generating process")

    # Least squares solution
    theta_hat = dot(linalg.inv(dot(x, x.T)), dot(x, y.T))
    plot_line(theta_hat, -x_limit, x_limit, "g", "Maximum Likelihood Solution")

    # Logistic regression solution
    labels = zeros(y.shape[0])
    for i in range(y.shape[0]):
        if y[i] > x_train[i]*theta[1] + theta[0]: labels[i] = 1
    theta_log = grad_descent(f, df, np.reshape(hstack((x_train, y)), (2,N)),
                np.reshape(labels, (1,N)), ones((3, 1)), 0.00000001)[0].T
    theta_log = np.array((theta_log[0,0]/theta_log[0,2],theta_log[0,1]/theta_log[0,2]))

    plot_line(theta_log, -x_limit, y_limit, "r", "Logistic Regression Solution")

    legend(loc=4)
    xlim([-x_limit, x_limit])
    ylim([-y_limit, y_limit])
    plt.savefig("logistic_vs_least_squares.png")

    score_log = 0.
    score_mle = 0.
    for i in range(y_test.shape[0]):

        correct_label = 0
        line = dot(theta, x_test)[i]
        if y_test[i] > line:
            correct_label = 1

        if correct_label == 1 and y_test[i] > dot(theta_hat, x_test)[i]:
            score_mle += 1
        if correct_label == 1 and y_test[i] > dot(theta_log, x_test)[i]:
            score_log += 1
        if correct_label == 0 and y_test[i] < dot(theta_hat, x_test)[i]:
            score_mle += 1
        if correct_label == 0 and y_test[i] < dot(theta_log, x_test)[i]:
            score_log += 1

    print("Performance on test set for MLE solution: " +str(score_mle/y_test.shape[0]))
    print("Performance on test set for Logistic solution: "
    + str(score_log / y_test.shape[0]))

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)
```

Part 6

No back-propagation versus vectorized back-propagation

To compare no-back propagation with vectorized back-propagation, the proof will be split in two parts: (1) compare how much faster storing intermediate results are and, (2) compare how much faster vectorizing the calculations are.

General Assumptions:

- All layers are connected.
- There are N *hidden* layers, K neurons per layer.

Part 1 - Storing intermediate results

For a motivation, we can look at the case for an ANN with two-hidden layers, each layer containing two neurons ($N = 2, K = 2$). It's straightforward to see that the partial derivative of the cost function with respect to weights on the *same* level have the same complexity. The partial derivatives of the cost function with respect to a weight on the second and first layer, respectively:

$$\frac{\partial C}{\partial W^{(2,1,1)}} = \frac{\partial C}{\partial o_1}\frac{\partial o_1}{\partial W^{(2,1,1)}}$$
$$\frac{\partial C}{\partial W^{(1,1,1)}} = \frac{\partial C}{\partial o_1}\frac{\partial o_1}{\partial h_1}\frac{\partial h_1}{\partial W^{(1,1,1)}} + \frac{\partial C}{\partial o_2}\frac{\partial o_2}{\partial h_1}\frac{\partial h_1}{\partial W^{(1,1,1)}}$$

Immediately, we see that the calculation $\partial h_1/\partial W^{(1,1,1)}$ is repeated twice, and it would be more efficient if said calculation is stored. Intuitively, a change in h_1 affects both o_1 and o_2, since it branches to them (hence two terms). This becomes more problematic as the number of layers increase, and we will term *branching factor* to indicate the number of parent nodes a node branches off to (which in this case, is 2). More generally, this branching factor is k, as there are k neurons and all the neurons are interconnected. As we will see later, storing $\partial h_i/\partial W^{(i,j,k)}$ for each weight (edge) will significantly improve run-time, which can be vectorized for each layer as $\mathbf{X}\left(\mathbf{h}_i \cdot (1 - \mathbf{h}_i)\right)^T$ for the sigmoid activation function ($\cdot$ denotes point-wise multiplication).

For the purposes of Part 1, the computational cost at each node will be assumed to be roughly equal (e.g the evaluation of the activation function on linear combinations of children nodes). Therefore, it can be assigned a cost of 1.

In an ANN with N hidden layers and K neurons, a change in $h_{(i,j)}$ directly affects k other neurons $h_{(i+1,0)}$, $h_{(i+1,1)}$, ..., $h_{(i+1,k)}$, each of which will affect k other neurons in the layer above (branching factor of k). Therefore, the calculation of $\partial h_i/\partial W^{(i,j,k)}$ will be repeated once for every branching path to the output. For example, in the computation of computation of $\partial C/\partial W^{(1,1,1)}$, neuron h_1 will branch off k times in the 2nd layer, each of which will branch off k times in the 3rd layer, etc., leading to k^N repetitions of the computation $\partial h_1/\partial W^{(1,1,1)}$. The cost of computing the gradient is cost of computing the gradient with respect to each weight in the ANN. That is,

$$\begin{aligned}\text{Cost} &= \text{Cost (while storing values)} + \text{Cost (repeated terms)}\\ &= \text{Cost (while storing values)} + \sum_{\forall i,j,k} \text{Cost}_{w^{(i,j,k)}}\end{aligned}$$

As discussed earlier, since the cost of the partial derivative with respect to the weights on the same level have the same complexity, the triple summation can be simplified to just *one*, which indexes over the *layers*.

Letting C' be the cost while storing the values,

$$\begin{aligned}\text{Cost} &= C' + \sum_{\forall i,j,k} \text{Cost}_{w^{(i,j,k)}} \\ &= C' + \sum_{\forall i} \text{Cost}_{w^{(i)}} \\ &= C' + N \cdot k^N + N \cdot k^{N-1} + ... + N \cdot k \\ &= C' + N\frac{1-k^{N+1}}{1-k} \\ &= C' + \mathcal{O}(N \cdot k^N),\end{aligned}$$

where N is multiplied to each term since there are N neurons per layer with cost k^i, i being the number of layers to the output. The added complexity $\mathcal{O}(N \cdot k^N)$ is why it is essential to store the partial derivatives at each neuron!

Part 2 - Benefits of Vectorization

Assumptions:

- Naive matrix multiplication with complexity $\mathcal{O}(n^3)$.
- Bias terms will be ignored as they are computational inexpensive in comparison to matrix multiplication.

One standard way of looking at back-propagation, that is, the process of computing $dC/d\mathbf{W}$, is by looking at the error signals, which can be defined for each neuron (in the hidden layers) as $\delta^{(l,j)} \equiv \partial C/\partial h^{(l,j)} = g'(h_j)\mathbf{W^{(l+1)}}^T * \delta^{(l+1)}$, which is vectorized for the jth neuron in the lth layer, where * denotes element-wise multiplication. As can be seen, this algorithm is applied recursively from the neurons in the outer layer all the way down to the input layer. For the neurons at the output layer, $\delta^{(l,j)} = g'(o_j)(p_j - y_j)$. Note that g is the activation function, where $g'(h_j) = g(h_j)(1 - g(h_j))$ for the sigmoid activation function. Further vectorizing can be done to calculate all the error terms in a given layer:

$$\boldsymbol{\delta^{(l)}} = \mathbf{g}'(\mathbf{h^{(l)}}) * \mathbf{W}^T \boldsymbol{\delta^{(l+1)}}$$

After which, we can calculate $\partial C/\partial W^{(l,j,k)}$ for each neuron by the following: $\partial C/\partial W^{(l,j,k)} = o^{(l-1,j,k)}\delta^{(l,j)}$, where $o^{(l-1,j,k)}$ is the output from the previous layer going to the current neuron. We can vectorize this entire expression for a given layer l, yielding:

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot \left(\mathbf{o}^{(l-1)}\right)^T \tag{6}$$

That is the gradient of the cost function for a weight matrix of size k by k (the number of edges) for a given layer, and since $\boldsymbol{\delta}^{(l)}$ is a k by 1 matrix (one per neuron) and $\mathbf{o}^{(l-1)}$ is k by 1 (one per ouput from the previous layer), we obtain a k by k result, as expected.

From the above equations, the most computationally expensive step of back-propagation is matrix multiplication, which has complexity $\mathcal{O}(k^3)$, where k is the number of neurons in a given layer. This is much faster compared to the exponential run-time in the previous discussion. Of course, for smaller neural networks without back-propagation, e.g $N = 3$, $\mathcal{O}(N \cdot k^N) = 3k^3$, the run-time is comparable but still bit slower. In practice, better algorithms for matrix multiplication exist (such as Strassen's Algorithm) and the fact that GPU's handles parallel computing much faster makes vectorization much more ideal.

Part 7

For this part of the project, various parameters were tested. But most importantly, two different network types were built using the original grayscale 32x32 images from the project 1 and higher quality 64x64 colored images. In both cases, images were flattened into 1024 or 12288 vectors (alpha channel ignored for the colored set). However, significant difference in performance wasn't observed. In fact, top performance was achieved on grayscale images, which wasn't expected. For the images, test set consisted of 30 images, validation set had 15 images and the training set had 90 images (all numbers are given per actor).
As it was required, the network was built with a single fully-connected hidden layer. The weight initialization was done through a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$. Tanh activation function was used and the training was done using mini-batches (though ReLU was also tried with a slightly worse peak performance). Additionally, a squared (L2) weight penalty function was introduced with a variable parameter λ.
This architecture gives rise to the following hyperparameters: number of hidden units, mini-batch size, λ. Training size and a number of iterations can also be tuned for experimenting.
The best performance on validation set was observed when using a relatively small number of hidden units (30), mini-batches of size 180, λ of 0.0001. The full training set was used to achieve this. Performance on the test set is **0.9166**, on the validation set: **0.9111**, and the training set: 1.0. The learning curve can be seen below:

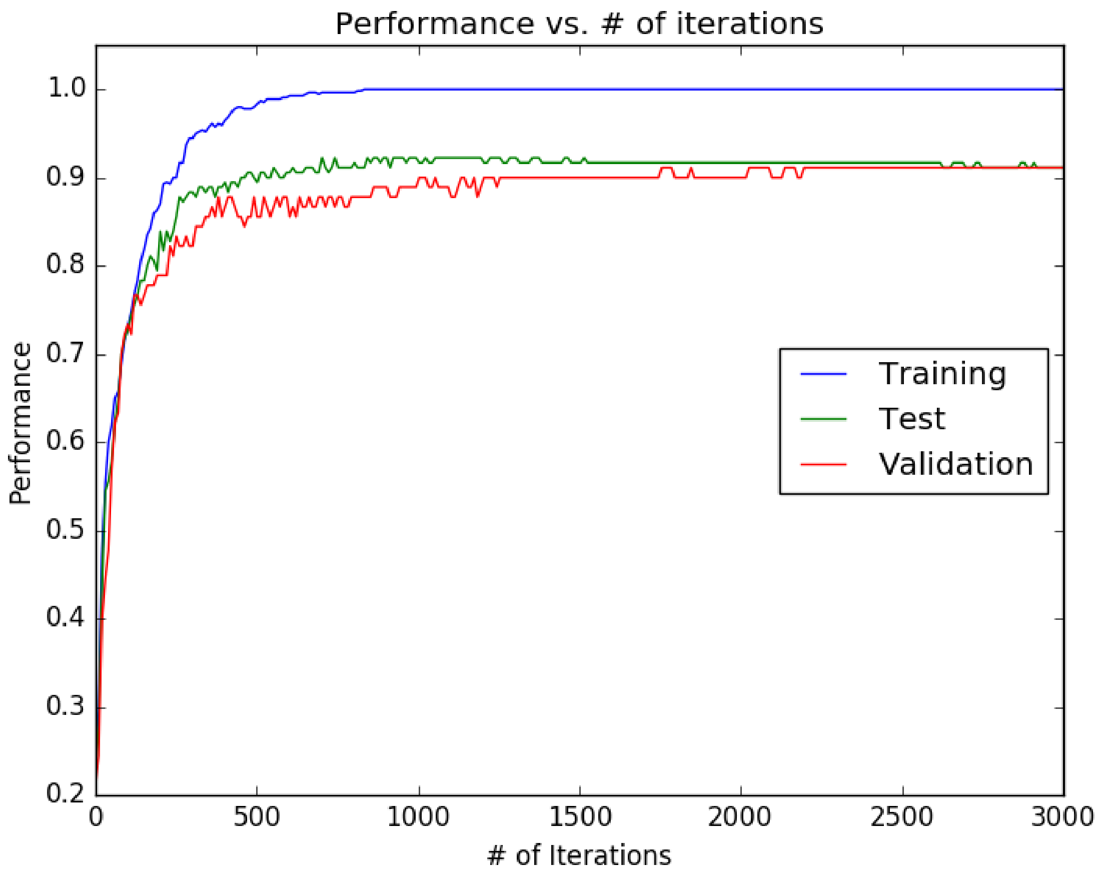


Figure 9: Learning curve for the best performing parameters

Part 8

Since introducing lambda penalty is one of the regularization methods that prevent overfitting, we expect to see it playing a significant role where there is an opportunity for overtraining a network. For a single hidden layer network, that can happen when a number of hidden units is too large. Training set can also be reduced in size so that a network adjusts its weights too specifically. One more parameter is having a large number of outliers in a dataset. In case of face images, these can be represented by pictures of faces turned away, for instance. There may be a large shadow or hair covering faces.
The tests were run for many different configurations. Additionally, special dataset (40 pictures per actor) was built so that half of the training set pictures weren't standard. The dataset is attached in the **data_outliers.zip** file. Few examples of outliers that were selected:

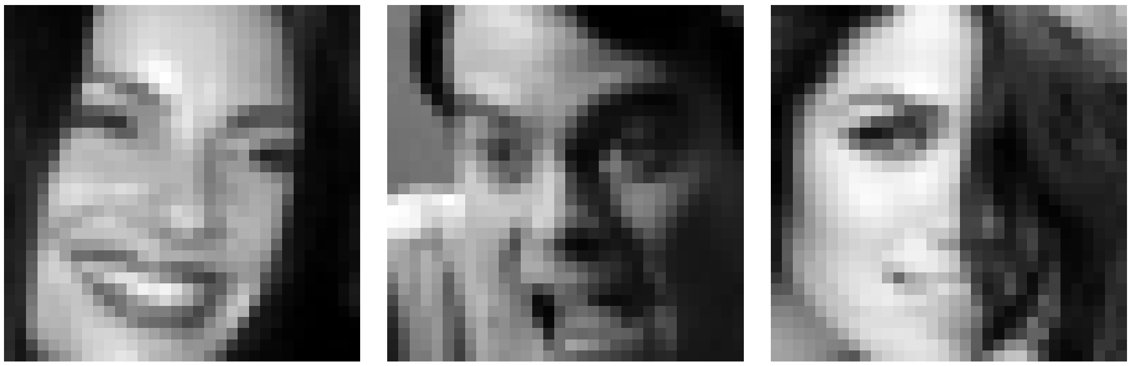


Figure 10: Examples of non standard face images that can be treated as outliers

It was also noted that a larger value of λ is required when the training set size is small. Correlation with the number of neurons is less visible, but also present - λ usually needs to regularize larger number of hidden units. Some sample results are given below (full results are in **results_part7.txt**):

```
Using 500 hidden units
0.0 lambda
40 Training size (outliers)
Final validation accuracy: 0.722222
=======
Using 500 hidden units
0.0001 lambda
40 Training size (outliers)
Final validation accuracy: 0.722222
=======
Using 500 hidden units
0.0003 lambda
40 Training size (outliers)
Final validation accuracy: 0.733333
=======
Using 500 hidden units
0.001 lambda
40 Training size (outliers)
Final validation accuracy: 0.744444
=======
Using 500 hidden units
0.003 lambda
40 Training size (outliers)
```

```
Final validation accuracy: 0.744444
=======
Using 500 hidden units
0.01 lambda
40 Training size (outliers)
Final validation accuracy: 0.788889
=======
Using 200 hidden units
0 lambda
40 Training size (outliers)
Final validation accuracy: 0.844444
=======
Using 200 hidden units
0.0001 lambda
40 Training size (outliers)
Final validation accuracy: 0.855556
=======
Using 200 hidden units
0.0003 lambda
40 Training size (outliers)
Final validation accuracy: 0.855556
=======
Using 200 hidden units
0.001 lambda
40 Training size (outliers)
Final validation accuracy: 0.855556
=======
Using 50 hidden units
0.0001 lambda
90 Training size (full set)
Final validation accuracy: 0.888889
=======
Using 50 hidden units
0.001 lambda
90 Training size (full set)
Final validation accuracy: 0.888889
=======
Using 50 hidden units
0.01 lambda
90 Training size (full set)
Final validation accuracy: 0.888889
=======
Using 50 hidden units
0.1 lambda
90 Training size (full set)
Final validation accuracy: 0.844444
```

It was also worth saying that while the results above do show the expected trend, sometimes this might not be a case. Sometimes, further increase of λ after a certain value resulted in worse performance. Or there might be another increase following a decrease. When the dataset contained many outliers, larger λ's may perform equally compared to smaller numbers. However, in case of the full dataset values close to 0.1 performed much worse.
But in general, some λ in the interval of [0.0001,0.05] gave up to 5% performance almost in all cases.

Part 9

2 methods of visualizing hidden units were tried. First, the most sensitive neuron for a particular actor was found by iterating through all of them and observing the weights. The criteria was the highest sum of correct scores across across a test set. This visualization method gave the following results for Bill Hader and Fran Drescher.

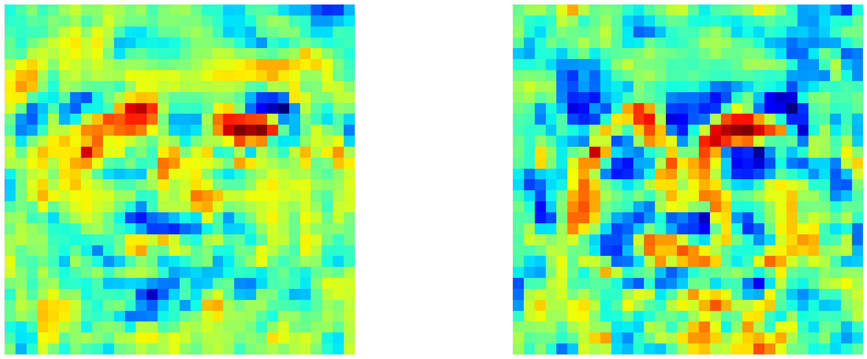


Figure 11: Visualizing weights for the most sensitive neurons recognizing Bill Hader and Fran Drescher

Secondly, a dot product was performed between input weights W0 and output weights W1. The results of this product is a 1024 by 6 matrix. By selecting a correct column, we can visualize a visualization for a specific actor.

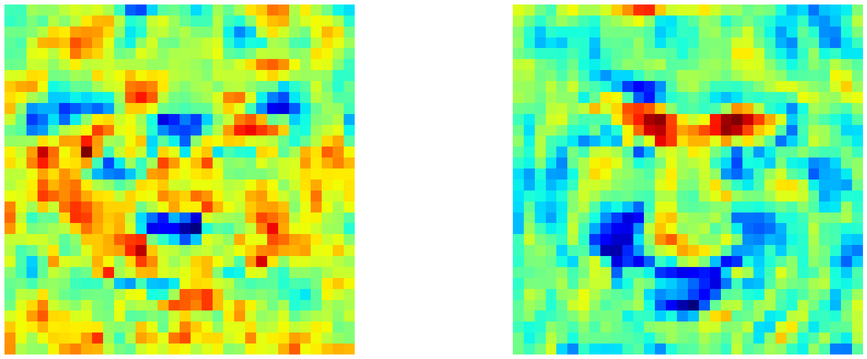


Figure 12: Visualizing dot product of $\mathbf{W}_0$ and $\mathbf{W}_1$

We can interpret the second method as more general description of a person compared to the first method. This way, all the features are combined in one picture. On the other hand, the first method highlighted just one unique feature of a person. It was also observed that the second method gave more ”symmetrical” faces in general.

Part 10

A network similar to the one built in part 7 was built on top of of the conv4 of the AlexNet network provided. For that, 227x227 colored pictures of actors were fed into the network, which produced 13x13x384 output (per image). The output was flattened for further analysis in a way similar to the previous part (i.e. via the dictionary). Thus, combined feature vector was of size 64896.
A fully-connected layer on top of this architecture was built. The layer used tanh activation function. Again, training was done in mini-batches. Lambda parameter was introduced with the penalty based on squared weights (L2).
The best performance on validation set was 0.9556 with the optimal parameters of 400 hidden units and $\lambda = 0.0001$. In general, it was noted that larger number of neurons should be used for this architecture in order to achieve better results. The test performance was **0.95** - 40% error reduction compared to the peak performance of the single FC network trained previously. A learning curve for the training is presented below. Right away we see that the learning is more rapid with less number of iterations required to achieve a maximum performance.

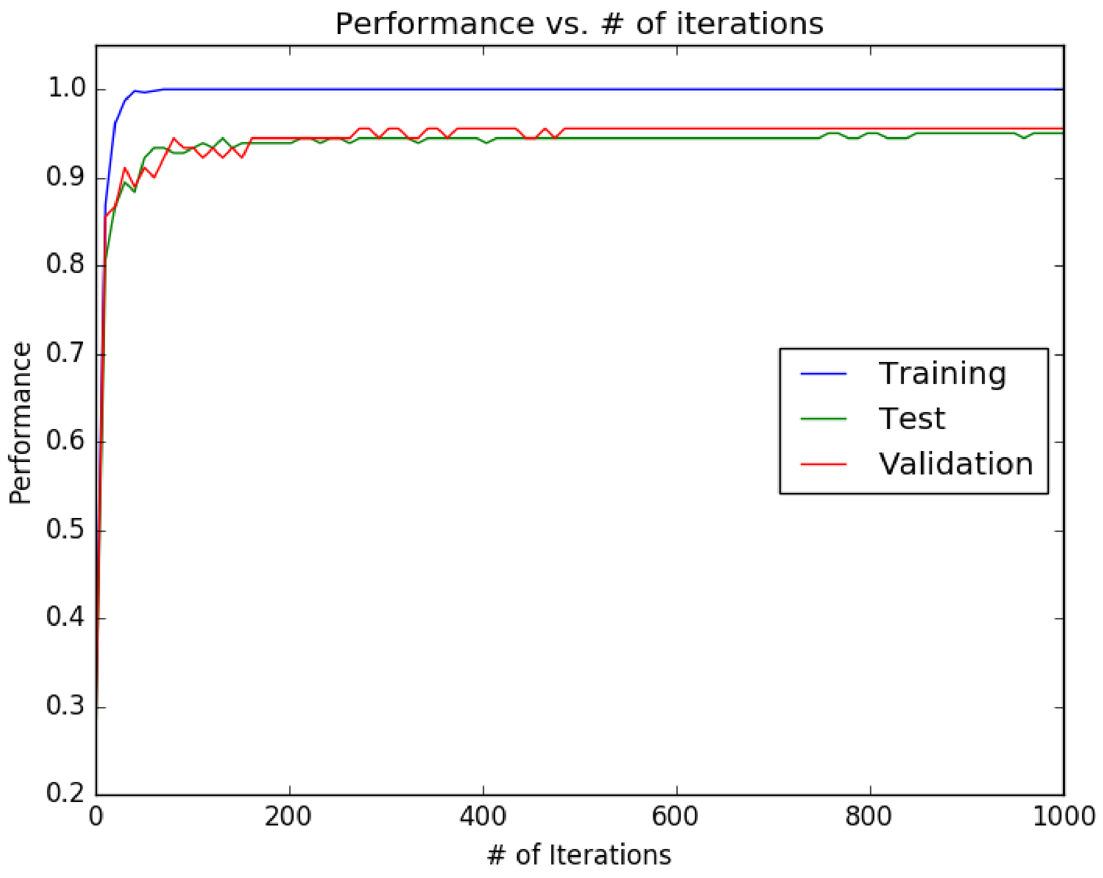


Figure 13: Learning curve for the best performing parameters

Part 11

The same method of visualizing was tried on the FC layer trained on top of the conv4 AlexNet layer. This visualization doesn't resemble faces in any way - this is something expected since features extracted by conv4 layer shouldn't look like faces. But this method doesn't highlight any features as well, and overall looks very random:

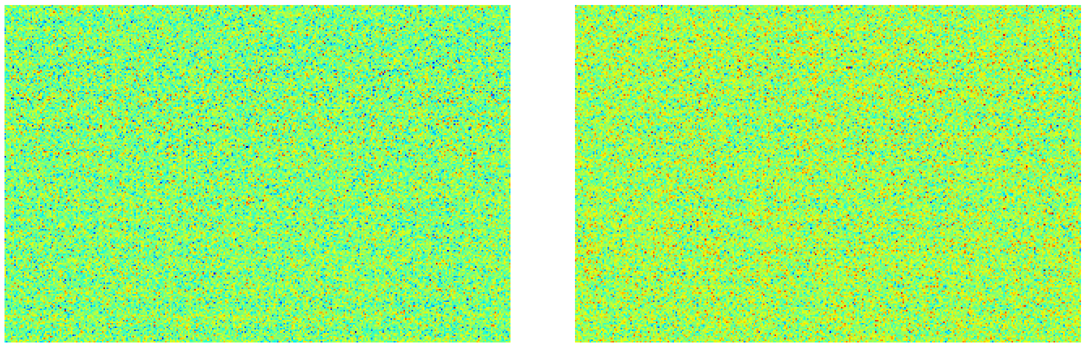


Figure 14: Visualizing deep neural net weights for the most sensitive neurons recognizing Bill Hader and Fran Drescher

Instead, we need to focus on visualizing separate features vectors. One method is shown below - simple summing across the weights for each feature vector (384 of them). Features are separated by dark blue pixel strips of size 5. This visualization is much better and does what is expected.

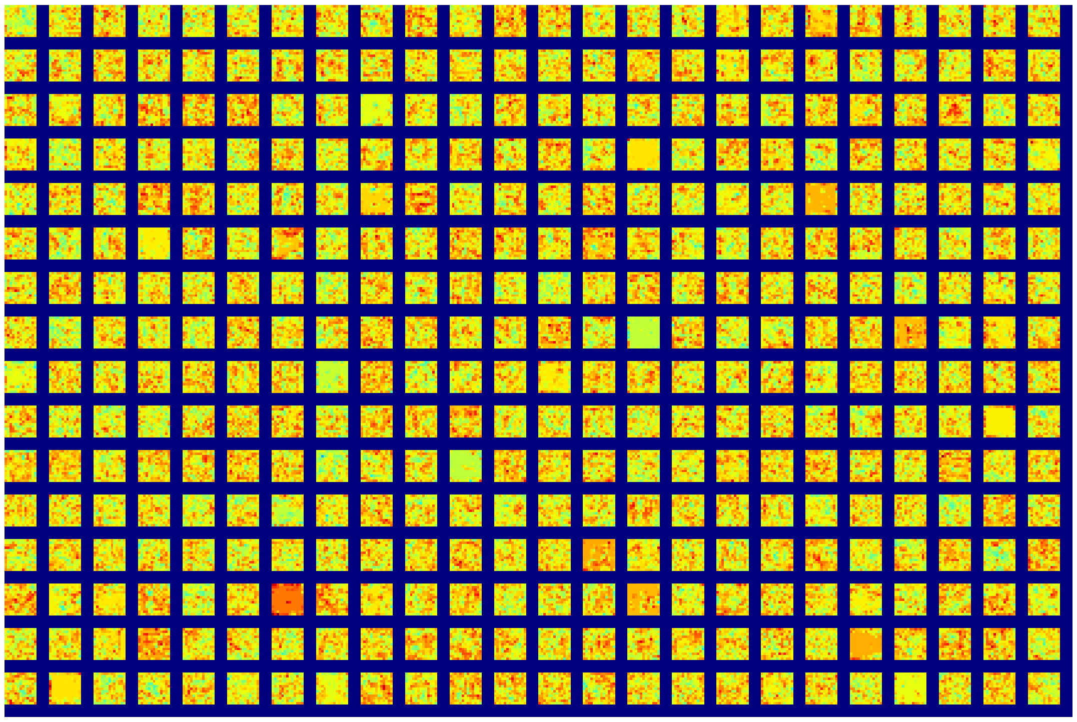


Figure 15: Visualizing deep neural net weights for each conv4 feature vector

Finally, we can combine these methods - determine most sensitive neurons for different actors and visualize feature vector weights separately - see the next page. Since features are small (just 13x13 pixels), we can't really interpret what the network is looking for. However, it is clear that each feature corresponds to some detail of the original images. Most of the features seem smooth, which is a good indication of well-trained parameters. Additionally, for each actor we see some almost empty squares - that means that some features were not found in the actor images.

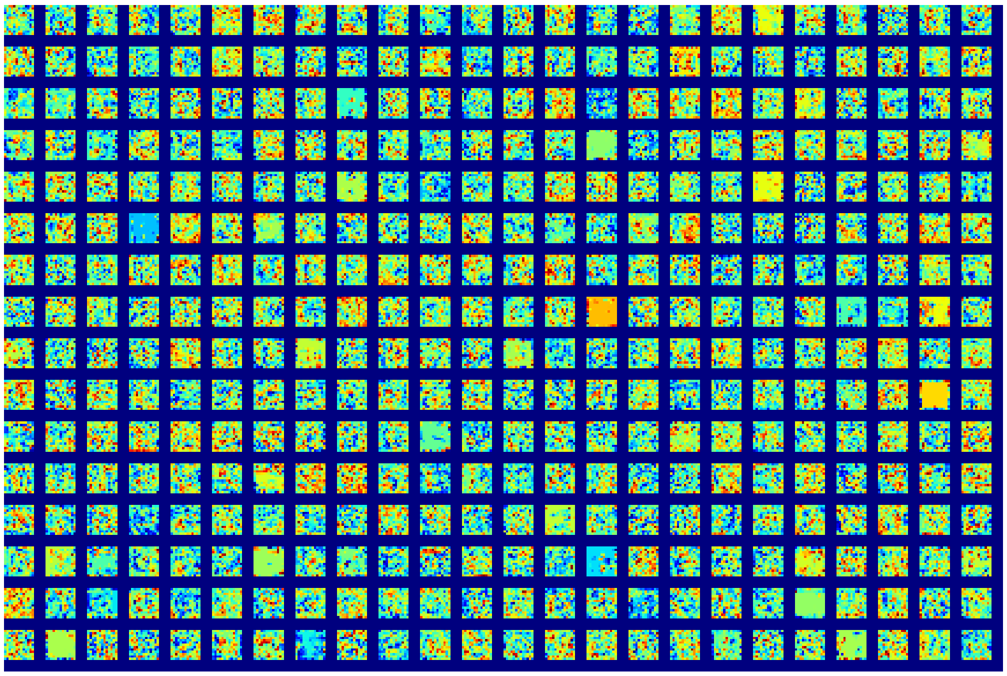


Figure 16: Visualizing deep neural net weights for the most sensitive neurons recognizing Bill Hader separated by features

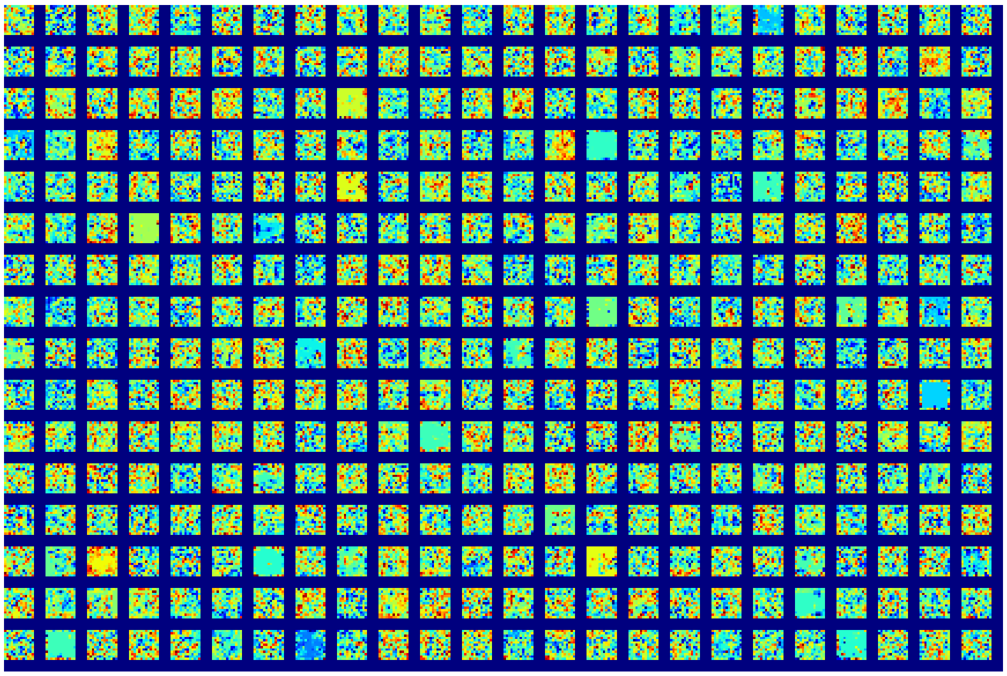


Figure 17: Visualizing deep neural net weights for the most sensitive neurons recognizing Fran Drescher separated by features