CSC321 Project 2: Handwritten Digit Recognition with Neural Networks

Due on Sunday, February 28, 2016

Wei Zhen Teoh, Ruiwen Li
1000960597 / 1001444136

February 28, 2016

Part 1

Dataset Overview

The dataset consists 60000 training set images and 10000 test set images of handwritten digits. 10 images of each digit are displayed below.

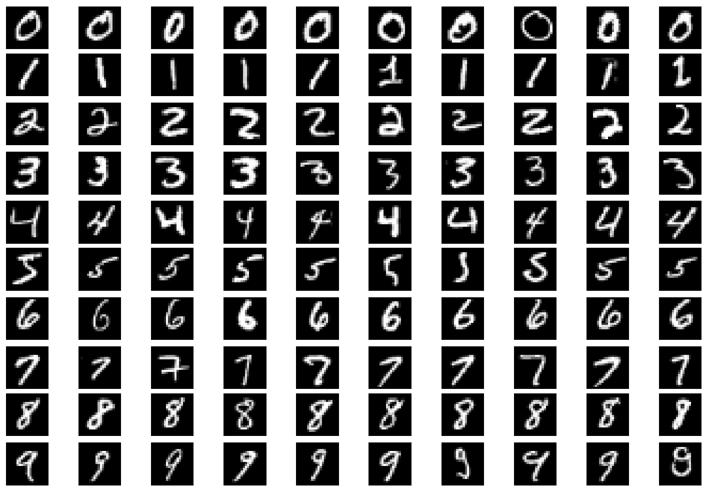


Figure 1: Sample images of all 10 digits from the dataset

Part 2

Linear Combination Function
The computation of the linear combination of the inputs x with given weights (W) and bias (b) for each output is implemented as follow:

```
def linComb(x, W, b):
    return dot(W.T,x)+b
    #W is weight matrix of dimension JxI,
    #each entry is w_ji, b is bias and has dimension Ix1
```

Part 3

2-layer Neural Network Derivative Computation

The following implementations yield the derivatives of the cost function with respect to the weight matrix and bias in the 2-layer neural network. The derivatives are the outputs of the function `deriv_2layer`.

```
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 cost(y, y_):
    '''Return the sum of negative log-probabilities of the correct answer for
    the M cases. y_ is the NxM one-hot encoded matrix and y is NxM matrix
    of the probabilities of the N outputs of all M cases.'''
    return -sum(y_*log(y))
#one-hot encoded matrix is one which has value 1 at entry corresponding
#to the correct digit for each case
#All the other entries have value 0

def deriv_2layer(x, W, b, y_):
    '''Return the derivative of the cost function with respect to each entry in
    the weight matrix W and the bias b. x is 784xM input matrix, M is the number
    of cases. y_ is the one-hot encoded matrix'''
    L = linComb(x, W, b)
    y = softmax(L)
    dCdL = y-y_
    dCdW = dot(x, dCdL.T )
    dCdb = dot(ones(x.shape[1]), dCdL.T).reshape(b.shape[0],1)
    return dCdW, dCdb
```

Part 4

Verification of 2-layer Neural Network Derivative Computation
The function `appderiv_2layer` yields the derivatives of the cost function with respect to the weight matrix and bias in the 2-layer neural network using finite difference approximation. We compared the results of the two different implementations (this and `deriv_2layer`) to verify that the derivatives computed through Part 3 is correct.

```
#finite difference approximation
def appderiv_2layer(x, W, b, y_, d):
    '''Return the derivatives of the cost function with respect to the weight
    matrix W and bias b using finite difference approximation. x is 784xM input
    matrix, M is the number of cases. y_ is the one-hot encoded matrix. d is the
    size of run in the gradient approximation.'''
    y = softmax(linComb(x, W, b))
    fixpoint = cost(y, y_)

    dCdW = zeros(W.shape)
    dCdb = zeros(W.shape[1])
    for i in range(W.shape[1]):
        for j in range(W.shape[0]):
            Wd = copy(W).astype(float)
            Wd[j][i] += d
            y = softmax(linComb(x, Wd, b))
            dCdW[j][i] = (cost(y,y_)-fixpoint)/d
    for i in range(W.shape[1]):
        bd = copy(b).astype(float)
        bd[i] += d
        y = softmax(linComb(x, W, bd))
        dCdb[i] = (cost(y,y_)-fixpoint)/d
    return dCdW, dCdb.reshape(dCdb.shape[0], 1)


#generate random W, b, x for computing derivatives using both methods
print 'Verifying the gradient calculation using random coods of small dimension'
random.seed(0)
for n in range(3):
    W = random.rand(3,2)
    b = random.rand(2).reshape(2, 1)
    x = random.rand(3,3)
    y_ = array([[0, 0, 1], [1, 1, 0]])
    print "the actual gradient calculated"
    dCdW, dCdb = deriv_2layer(x, W, b, y_)
    print "dCdW = "+ str(dCdW)
    print "dCdb = " + str(dCdb)
    print "the gradient obtained by finite-difference approximation"
    dCdW, dCdb = appderiv_2layer(x, W, b, y_, 0.01)
    print "dCdW = "+ str(dCdW)
    print "dCdb = " + str(dCdb)
```

The output of the code above is

```
Verifying the gradient calculation using random coods of small dimension
the actual gradient calculated
dCdW = [[-0.01373891  0.01373891]
 [-0.18280839  0.18280839]
 [ 0.04522035 -0.04522035]]
dCdb = [[ 0.09983164]
 [-0.09983164]]
the gradient obtained by finite-difference approximation
dCdW = [[-0.01177271  0.015702  ]
 [-0.18111118  0.1845033 ]
 [ 0.04523551 -0.04520519]]
dCdb = [[ 0.10331657]
 [-0.0963529 ]]
the actual gradient calculated
dCdW = [[ 0.27590405 -0.27590405]
 [ 0.57771928 -0.57771928]
 [ 0.69640458 -0.69640458]]
dCdb = [[ 1.10299543]
 [-1.10299543]]
the gradient obtained by finite-difference approximation
dCdW = [[ 0.27727086 -0.27453408]
 [ 0.57825651 -0.57718139]
 [ 0.69757848 -0.69522857]]
dCdb = [[ 1.10613334]
 [-1.09984911]]
the actual gradient calculated
dCdW = [[-0.08046261  0.08046261]
 [ 0.29580366 -0.29580366]
 [-0.02597475  0.02597475]]
dCdb = [[ 0.31136347]
 [-0.31136347]]
the gradient obtained by finite-difference approximation
dCdW = [[-0.07933378  0.08159049]
 [ 0.29641802 -0.29518979]
 [-0.02529445  0.02665474]]
dCdb = [[ 0.31502928]
 [-0.3077007 ]]
```

The values of the derivatives `dCdW` and `dCdb` produced by both methods are very close. We thus concluded that our implementation in Part 3 is correct.

Part 5

2-layer Neural Network Weight Optimization - Mini Batch Gradient Descent
Using a learning rate of 0.01, and batch size of 50 cases, we implemented the mini batch gradient descent method to optimize the weight matrix and bias for handwritten digits recognition. The training cost was minimized after 4000 steps of gradient descent. The corresponding weight matrix and bias obtained yield a correct classification rate of 92.26% on the test set. The cost function and correct classification rate on the training set and test set after every 100 steps of gradient descent are graphed below:

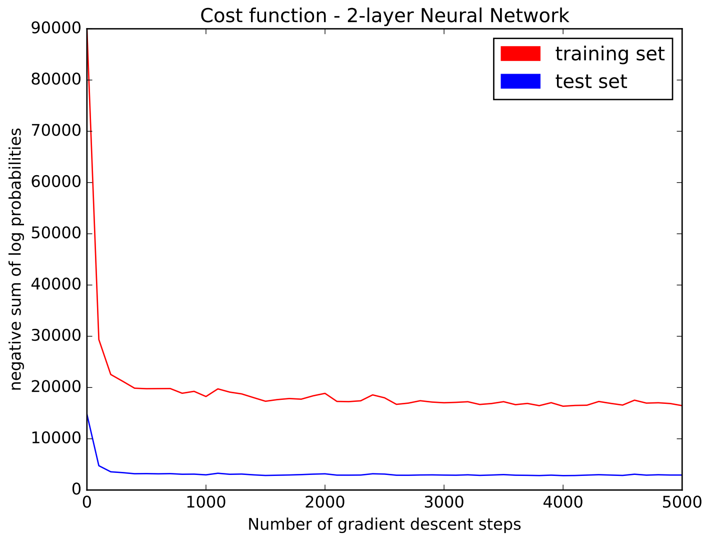


Figure 2: Cost function: sum of negative log probabilities of the correct answers

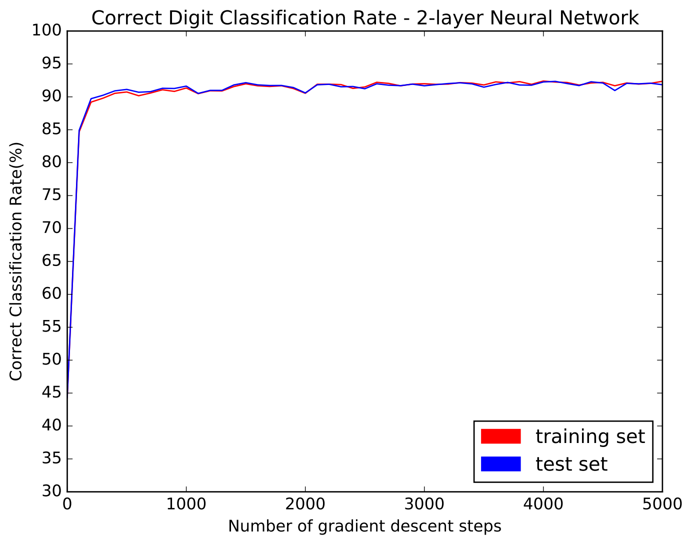


Figure 3: Correct digit classification rates

Using the weight matrix and bias for which the training cost is minimized, we identified 20 digit images that were classified correctly and 10 digit images that were classified incorrectly:

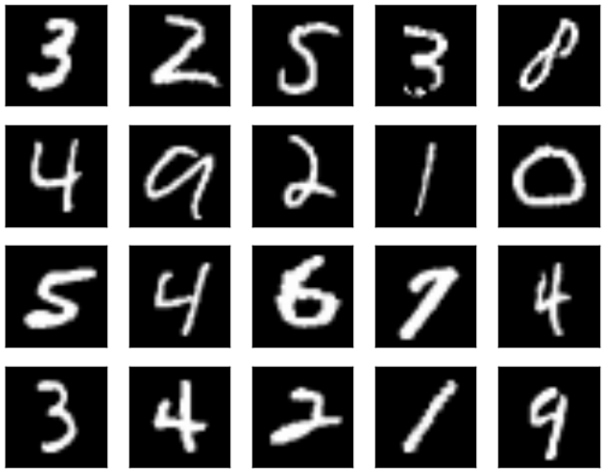


Figure 4: Correctly classified digit images

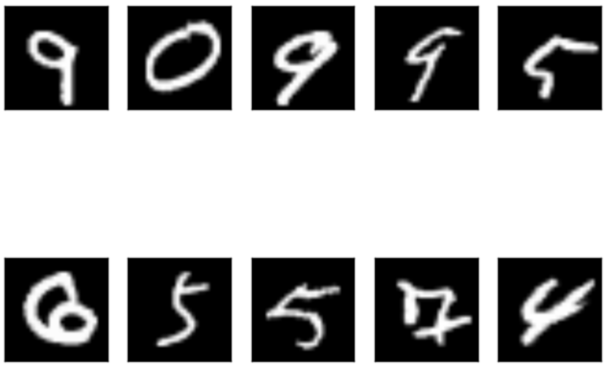


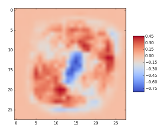
Figure 5: Incorrectly classified digit images

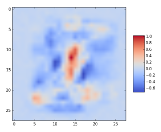
We noticed that the correctly classified digit images are also those that are easy to be identified visually by humans while most of those that were incorrectly identified are the exact opposite. It is unsurprising that those that look like the former ones are more common in the dataset as displayed in part 1. As a result, there is higher chance that the neural network had classified them correctly compared to the latter ones after being trained using images from the dataset.

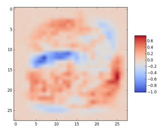
Part 6

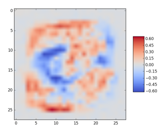
2-layer Neural Network Weight Heatmaps

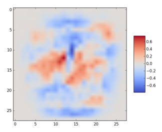
We produced the heatmaps of the weight components connected to each distinct output:

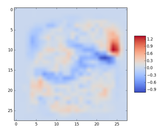

(a) Weight components for output 0

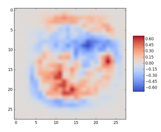

(b) Weight components for output 1

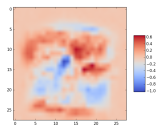

(c) Weight components for output 2

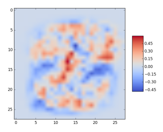

(d) Weight components for output 3


(e) Weight components for output 4


(f) Weight components for output 5


(g) Weight components for output 6


(h) Weight components for output 7


(i) Weight components for output 8

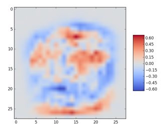

(j) Weight components for output 9

Figure 6: Heatmaps of the weight components connected to outputs for each digit

We noticed that the appearances of the heatmaps (the red areas) correspond to the appearances of the digits. This has to do with the training cost designed. In order to minimize the training cost, the weight is adjusted such that it gives the highest value of linear combination in the output channel for that digit. In general, this is done by adjusting the weight vector for that digit output such that it is 'closer' to its image pixel vectors. Consequently, the distribution of the weight components connected to each digit output is close to the pixel value distribution of images of the same digit.

Part 7

```
def deriv_multilayer(W0, b0, W1, b1, x, L0, L1, y, y_):
    '''Return the derivatives of the cost in a multilayer neural network with
    respect to weight matrices W0, W1 and biases b0, b1. x is the input cases of
    dimension 784xM, M is the number of cases. L0 and L1 are the outputs in
    second and third layers of the neural network. y is the matrix of
    probabilities for each digit and each case. y_ is the one-hot encoded matrix
    .'''
    dCdL1 =  y - y_
    dCdW1 =  dot(L0, ((1- L1**2)*dCdL1).T )
    dCdb1 = (dot(ones(L0.shape[1]), ((1- L1**2)*dCdL1).T)).reshape(len(b1),1)
    dCdL0 = dot(W1, ((1- L1**2)*dCdL1))
    dCdW0 = dot(x, ((1- L0**2)*dCdL0).T)
    dCdb0 = dot(ones(x.shape[1]), ((1- L0**2)*dCdL0).T).reshape(len(b0),1)
    return dCdW1, dCdb1, dCdW0, dCdb0
```

For each case m of the M cases, the derivative of cost with respect to probability for each digit i output is $\frac{\partial C_m}{\partial p_{mi}} = -\frac{1}{p_{mi}}$ if i is the correct digit and 0 otherwise. The derivative of the probability for the correct digit i with respect to outputs in L1 are: $\frac{\partial p_{mi}}{\partial L1_{mi}} = 1 - p_{mi}$ and $\frac{\partial p_{mi}}{\partial L1_{mj}} = -p_{mi}p_{mj}$
So applying chain rule: $\frac{\partial C_m}{\partial L1_{mj}} = \frac{\partial C_m}{\partial p_{mi}} * \frac{\partial p_{mi}}{\partial L1_{mj}} = p_{mj}$ and $\frac{\partial C_m}{\partial L1_{mi}} = \frac{\partial C_m}{\partial p_{mi}} * \frac{\partial p_{mi}}{\partial L1_{mi}} = p_{mi} - 1$ So the results for all M cases are the entries in the matrix `dCdL1`, which is just the probability matrix minus the one-hot encoded matrix.
The derivative for tanh function is: $1 - tanh^2(x)$ So the derivative of L1 with respect to the outputs prior to tanh function is $\frac{\partial L1_{mi}}{\partial o_{mi}} = 1 - L1_{mi}^2$
Applying chain rule again, $\frac{\partial C_m}{\partial o_{mi}} = (1 - L1_{mi}^2) * \frac{\partial C_m}{\partial L1_{mi}}$. Also we know that $\frac{\partial o_m i}{\partial W_(1,j,i)} = L0_{mj}$. Thus summing across M cases, $\frac{\partial C}{\partial W_(1,j,i)} = \sum_m \left(\frac{\partial\, C_m}{\partial\, o_{mi}}\right) \cdot \left(L0_{mj}\right).$
The results obtained is each j,i entry in the `dCdW1` matrix. And the implementation is performed through the operation in the second line.
By the same idea, since that $\frac{\partial o_{mi}}{\partial W_{(1,j,i)}} = 1$, $\frac{\partial C}{\partial b_(1,i)} = \sum_m \left(\frac{\partial\, C_m}{\partial\, o_{mi}}\right)$. This is exactly the operation to yield each entry i in `dCdb1`
The derivative of the pre-tanh L1 outputs with respect to the L0 layer outputs are $\frac{\partial o_m i}{\partial L0_{mj}} = W_{(1,j,i)}$ So for each case, $\frac{\partial C_m}{\partial L0_{mj}} = \sum_i \left(\frac{\partial\, C_m}{\partial\, o_{mi}}\right) \cdot \left(W_{(1,j,i)}\right)$
Each j,m enty in the `dCdL0` is produced through this operation. It has dimension JxM where J is the number of hidden units in the L0 layer.
Lastly, by the symmetry of the relationships of `(dCdL1, dCdW1, dCdb1)` and `(dCdL0, dCdW0, dCdb0)`, we make a symmetrical implementation to obtain `dCdW0` and `dCdb0`.

Part 8

Verification of Multi-layer Neural Network Derivative Computation
The function `appderiv_multilayer` yields the derivatives of the cost function with respect to the two weight matrices and biases in the Multi-layer neural network using finite difference approximation. We compared the results of the two different implementations (another one being the one in Part 7) to verify that the derivatives computed through Part 7 is correct.

```
#code for finite-difference approximation
def appderiv_multilayer(x, W0, b0, W1, b1, y_, d):
    L0, L1, y = forward(x, W0, b0, W1, b1)
    fixpoint = cost(y, y_)
    dCdW1 = zeros(W1.shape)
    dCdb1 = zeros(W1.shape[1])
    dCdW0 = zeros(W0.shape)
    dCdb0 = zeros(W0.shape[1])
    for i in range(W1.shape[1]):
        for j in range(W1.shape[0]):
            W1d = copy(W1).astype(float)
            W1d[j][i] += d
            y = forward(x, W0, b0, W1d, b1)[2]
            dCdW1[j][i] = (cost(y,y_)-fixpoint)/d
    for i in range(W0.shape[1]):
        for j in range(W0.shape[0]):
            W0d = copy(W0).astype(float)
            W0d[j][i] += d
            y = forward(x, W0d, b0, W1, b1)[2]
            dCdW0[j][i] = (cost(y,y_)-fixpoint)/d
    for i in range(W1.shape[1]):
        b1d = copy(b1).astype(float)
        b1d[i] += d
        y = forward(x, W0, b0, W1, b1d)[2]
        dCdb1[i] = (cost(y,y_)-fixpoint)/d
    for i in range(W0.shape[1]):
        b0d = copy(b0).astype(float)
        b0d[i] += d
        y = forward(x, W0, b0d, W1, b1)[2]
        dCdb0[i] = (cost(y,y_)-fixpoint)/d
    return dCdW1, dCdb1.reshape(dCdb1.shape[0],1), dCdW0, dCdb0.reshape(dCdb0.shape[0], 1)

print 'Verifying the gradient calculation
        using random coods of small dimension'
random.seed(0)
for n in range(3):
    W1 = random.rand(3,2)
    b1 = random.rand(2).reshape(2, 1)
    W0 = random.rand(4,3)
    b0 = random.rand(3).reshape(3, 1)
    x = random.rand(4,3)
    y_ = array([[0, 0, 1], [1, 1, 0]])
    print "the actual gradient calculated"
    L0, L1, y = forward(x, W0, b0, W1, b1)
    dCdW1, dCdb1, dCdW0, dCdb0 = deriv_multilayer(W0, b0, W1, b1, x, L0, L1, y, y_)
    print "dCdW1 = "+ str(dCdW1)
    print "dCdb1 = " + str(dCdb1)
    print "dCdW0 = "+ str(dCdW0)
    print "dCdb0 = " + str(dCdb0)

    print "the gradient obtained by finite-difference approximation"
    dCdW1, dCdb1, dCdW0, dCdb0 = appderiv_multilayer(x, W0, b0, W1, b1, y_, 0.01)
    print "dCdW1 = "+ str(dCdW1)
    print "dCdb1 = " + str(dCdb1)
    print "dCdW0 = "+ str(dCdW0)
    print "dCdb0 = " + str(dCdb0)
```

The output of the code above is

```
Verifying the gradient calculation using random coods of small dimension
the actual gradient calculated
dCdW1 = [[ 0.03371435 -0.00725806]
 [ 0.03373228 -0.00727393]
 [ 0.03376632 -0.00727908]]
dCdb1 = [[ 0.03402525]
 [-0.00731837]]
dCdW0 = [[ -1.99767991e-04  -2.30852006e-04  -9.12965063e-05]
 [  2.38125414e-04  -2.33494381e-04  -1.52408333e-04]
 [ -5.22643223e-04  -1.71426318e-03  -8.84393763e-04]
 [  8.22673041e-04   2.33323977e-03   1.21099503e-03]]
dCdb0 = [[ 0.00026061]
 [ 0.00038682]
 [ 0.00020481]]
the gradient obtained by finite-difference approximation
dCdW1 = [[ 0.03341823 -0.00718595]
 [ 0.03343508 -0.00720199]
 [ 0.03346848 -0.00720693]]
dCdb1 = [[ 0.0337245 ]
 [-0.00724495]]
dCdW0 = [[ -1.98753593e-04  -2.31601814e-04  -9.20521821e-05]
 [  2.33733681e-04  -2.37480697e-04  -1.54302044e-04]
 [ -5.16349219e-04  -1.69379521e-03  -8.74453691e-04]
 [  8.18262621e-04   2.32105770e-03   1.20491341e-03]]
dCdb0 = [[ 0.00025802]
 [ 0.0003832 ]
 [ 0.00020249]]
the actual gradient calculated
dCdW1 = [[ 0.02751582 -0.01600964]
 [ 0.04204579 -0.0240172 ]
 [ 0.03548532 -0.0202872 ]]
dCdb1 = [[ 0.05470911]
 [-0.03113046]]
dCdW0 = [[ 0.00766466  0.00126956  0.01035589]
 [ 0.00212959  0.00063249  0.00535499]
 [-0.00530042 -0.00059094 -0.00605846]
 [ 0.00285727  0.00082527  0.00711856]]
dCdb0 = [[ 0.00960756]
 [ 0.00194673]
 [ 0.01495346]]
```

```
the gradient obtained by finite-difference approximation
dCdW1 = [[ 0.02734073 -0.01589296]
 [ 0.04178699 -0.02383035]
 [ 0.03528995 -0.02015129]]
dCdb1 = [[ 0.05426131]
 [-0.03081776]]
dCdW0 = [[ 0.0076315   0.00126713  0.01030063]
 [ 0.00211865  0.00063138  0.00531966]
 [-0.00523262 -0.00058472 -0.00595834]
 [ 0.0028386   0.00082335  0.00705974]]
dCdb0 = [[ 0.00953799]
 [ 0.00194274]
 [ 0.01479898]]
the actual gradient calculated
dCdW1 = [[ 0.00951473 -0.17820615]
 [ 0.01411871 -0.24855832]
 [ 0.01402793 -0.25030584]]
dCdb1 = [[ 0.01397154]
 [-0.25897271]]
dCdW0 = [[-0.04197098 -0.0257332   0.00023657]
 [ 0.0015298   0.01644763 -0.00095736]
 [-0.02018402 -0.00390268 -0.00019634]
 [-0.07052709 -0.03975845  0.00037153]]
dCdb0 = [[-0.05658585]
 [-0.01513997]
 [-0.00064936]]
the gradient obtained by finite-difference approximation
dCdW1 = [[ 0.00945257 -0.17690077]
 [ 0.01398599 -0.24628446]
 [ 0.01389483 -0.24792971]]
dCdb1 = [[ 0.01383751]
 [-0.25625876]]
dCdW0 = [[-0.04173252 -0.02553769  0.00023241]
 [ 0.00146565  0.01629045 -0.00094458]
 [-0.02002947 -0.00382286 -0.00019721]
 [-0.06996222 -0.03936335  0.00036312]]
dCdb0 = [[-0.05608496]
 [-0.01494601]
 [-0.00064066]]
```

The values of the derivatives `dCdW1`, `dCdW0` and `dCdb1`, `dCdb0` produced by both methods are very close. We thus concluded that our implementation in Part 7 is correct.

Part 9

Multi-layer Neural Network Weight Optimization - Mini Batch Gradient Descent
Using a learning rate of 0.01, and batch size of 50 cases, we implemented the mini batch gradient descent method to optimize the weight matrices and biases for handwritten digits recognition. The training cost was minimized after 9500 steps of gradient descent. The corresponding weight matrices and biases obtained yield a correct classification rate of 97.21% on the test set. The cost function and correct classification rate on the training set and test set after every 100 steps of gradient descent are graphed below. we can see that compared to part 5, the learning curve grows slower at the beginning but it eventually reaches higher performance. The performance on training set and test set start to diverge after about 6000 steps of gradient descent.

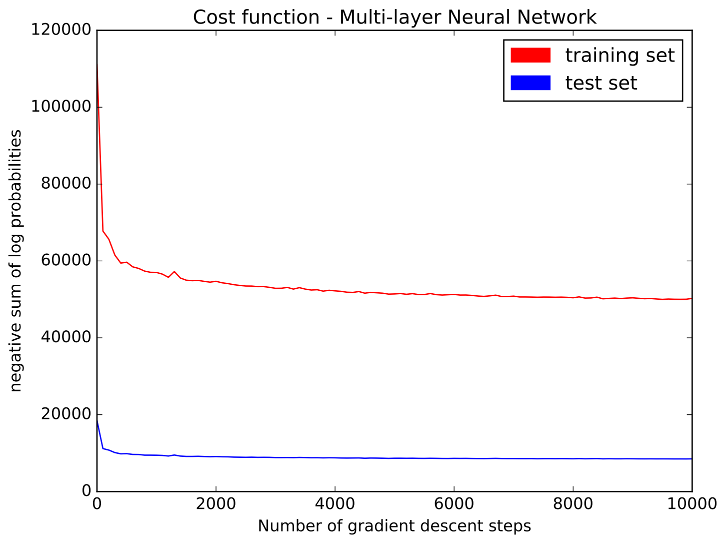


Figure 7: Cost function: sum of negative log probabilities of the correct answers

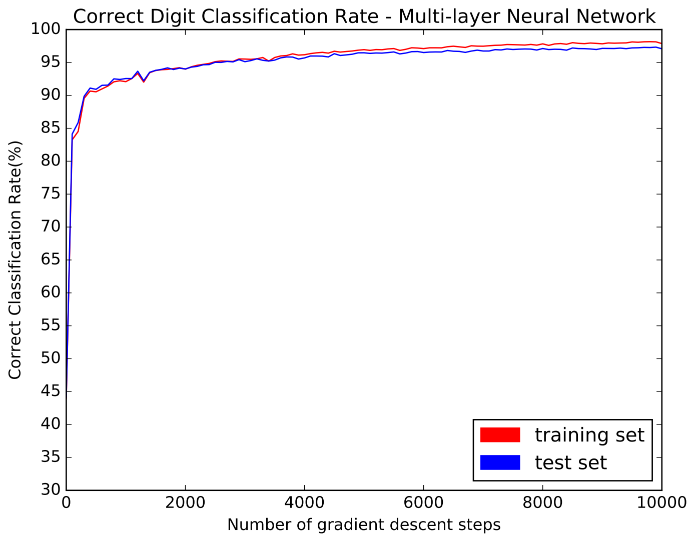


Figure 8: Correct digit classification rates

Some examples of correctly and incorrectly classified cases are displayed here:

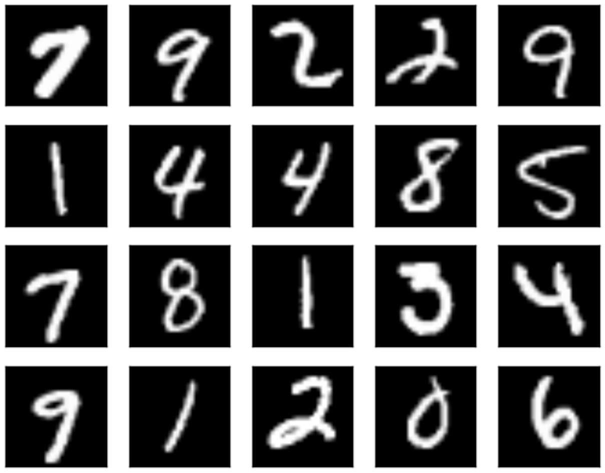


Figure 9: Correctly classified digit images

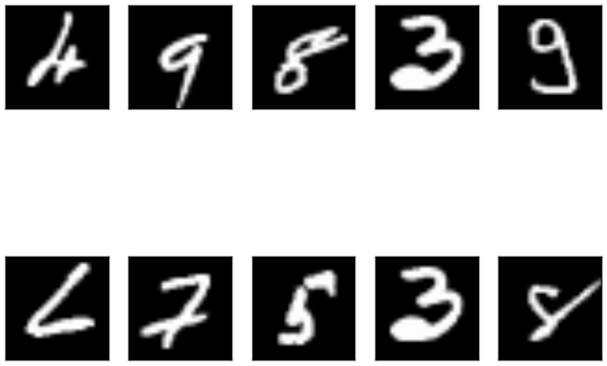
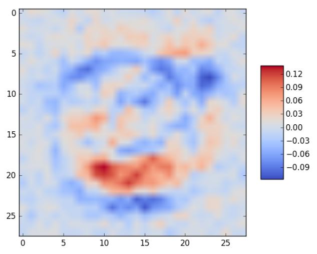


Figure 10: Incorrectly classified digit images

Part 10

Multi-layer Neural Network Weight Heatmaps
Using the best weight matrices that minimized the training cost obtained from part 9, we produced the heatmaps of the weight components connected to 2 different hidden units:


(a) Weight components connected to hidden unit 150

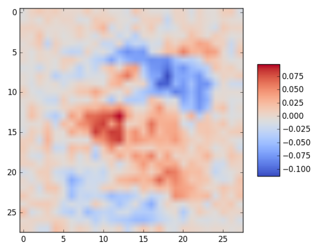

(b) Weight components connected to hidden unit 270

Figure 11: Heatmaps of the weight components connected to the hidden units.

The weight components connecting the first hidden unit to the each of the 10 digit outputs are [0.2241291, 0.07528672, 0.21050657, 0.17347277, -0.011706, 0.05184363, 0.30230802, -0.0927176, 0.07148419, 0.07864353]. Notice that the weights for digit 0 and digit 6 output are significantly larger than the rest. Thus, if the weight components connected to this hidden unit is to help producing good prediction, it should gives higher outputs when the digit images are indeed 0, 6. And by the same reason stated in part 5, the heatmap should appear to mimick a combination of digits 0 and 6, which is what we observed.
The weight components connecting the second hidden unit to each of the digit output is [0.03861332, 0.04861392, 0.16066871, 0.03146538, 0.12853166, 0.11023266, 0.14077693, -0.01080319, 0.0585668, -0.08794519]. The weight components are small for each digit output and we thus do not expect the heatmap to look like any specific digit. Indeed, that is what we observed.