Sayyed Mohsen Vazirizade

23398312

smvazirizade@email.arizona.edu

INFO 521

Introduction to Machine Learning

Assignment 5

Problem 1

For problem the line 63 is modified as follows:

```
images = load_MNIST.load_MNIST_images('data/train-images.idx3-ubyte')
```

and using the following three commands we generate the three figures below.

```
visualize.plot_images(patches_train[:, 0:10], title='first 10')
visualize.plot_images(patches_train[:, 0:50], title='first 50')
visualize.plot_images(patches_train[:, 0:100], title='first 100')
```

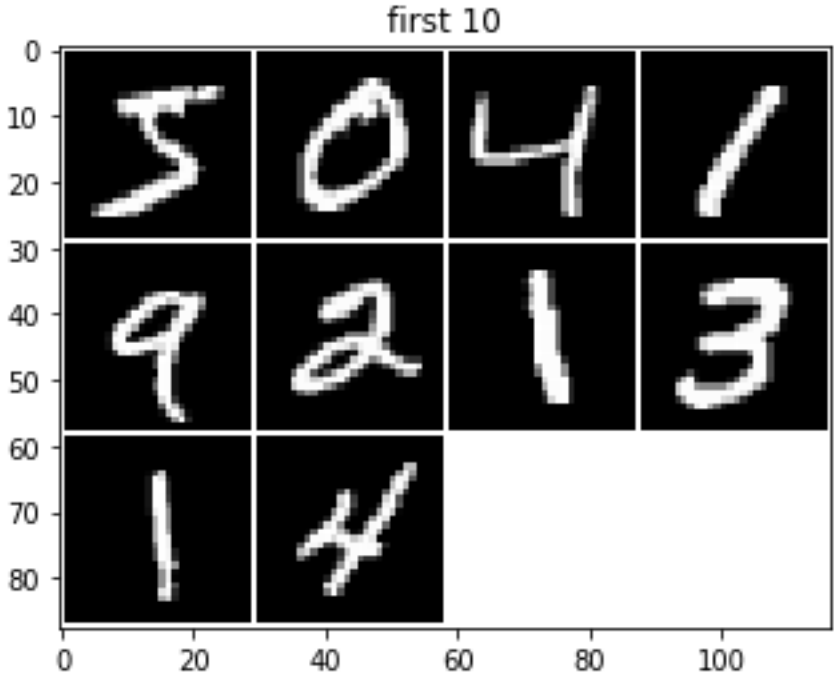

Figure 1 The first 10 figures for training

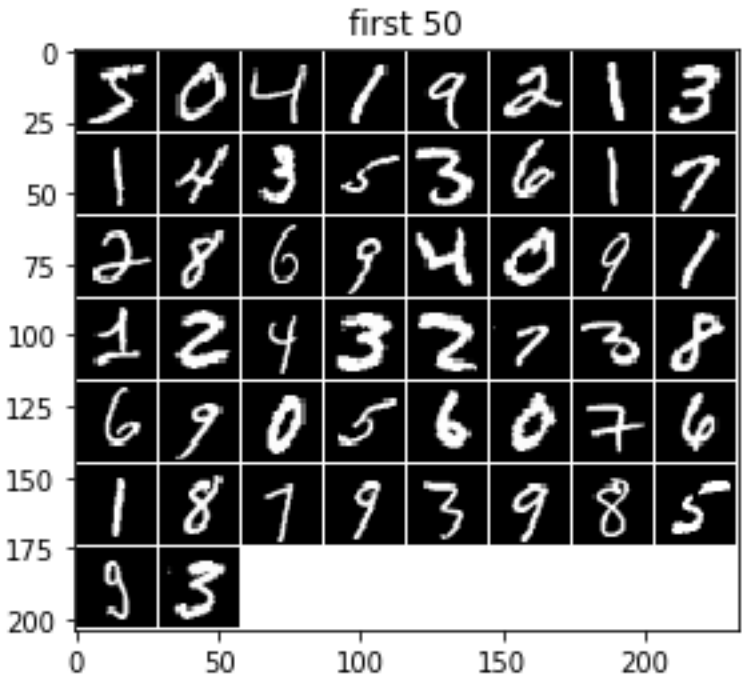

Figure 2The first 50 figures for training

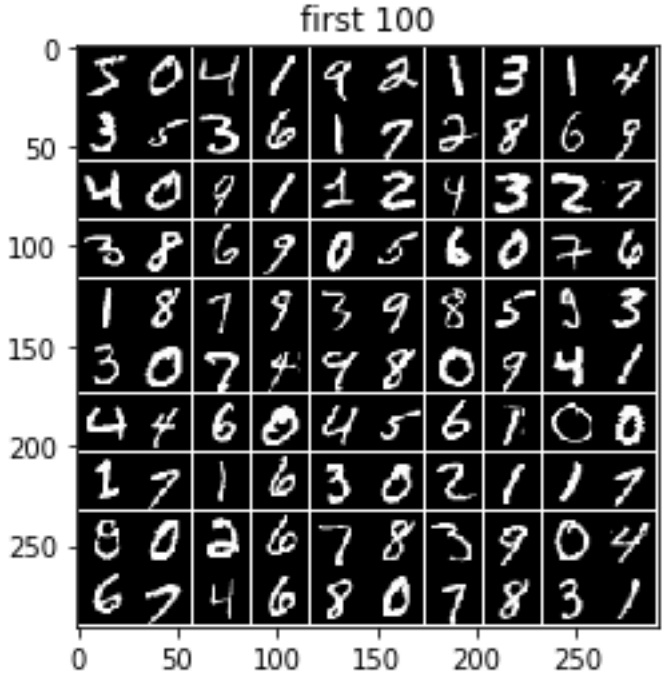

Figure 3The first 100 figures for training

Furthermore, for creating the following figure, showing the first 100 figures for test, we use the following command.

```
visualize.plot_images(patches_test[:, 0:100], title='first 100 test')
```

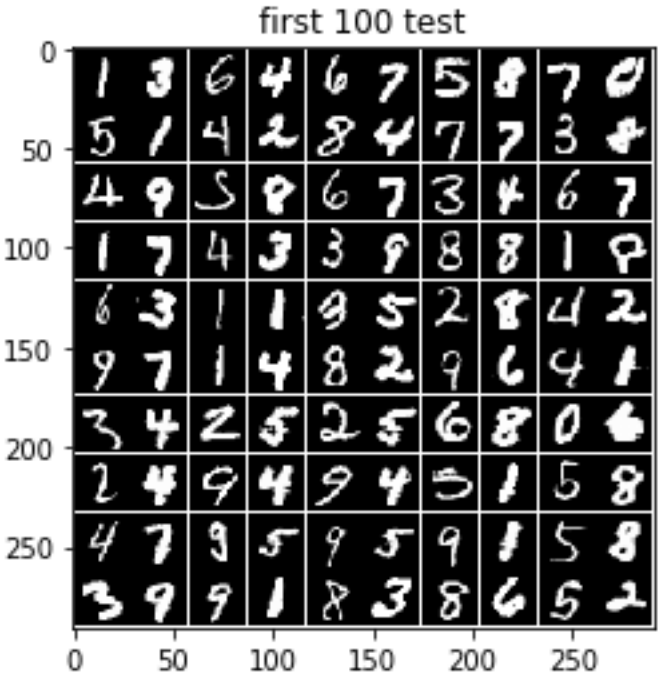

Figure 4The first 100 figures for test

Problem 2

The following lines are just added to the script. It is worth mentioning that in the question statement the number of neurons in output is considered two, and then that number 17 is calculated.

```
A1=visible_size*hidden_size
B1=hidden_size
A2=hidden_size*2
B2=2

w1=numpy.random.uniform(-1,1,(A1,1))
w2=numpy.random.uniform(-1,1,(A2,1))
b1=numpy.random.uniform(-1,1,(B1,1))
b2=numpy.random.uniform(-1,1,(B2,1))
print('w1')
theta=numpy.concatenate((w1,w2))
theta=numpy.concatenate((theta,b1))
theta=numpy.concatenate((theta,b2))
print(theta)
```

Problem 3

The first step is converting the data from array to list format using the following script:

```
w1 = theta[:hidden_size*visible_size].reshape(hidden_size, visible_size)

w2 = theta[hidden_size*visible_size:(hidden_size*visible_size)*2].reshape(visible_size, hidden_size)

b1 = theta[(hidden_size*visible_size)*2:((hidden_size*visible_size)*2)+hidden_size]

b2 = theta[((hidden_size*visible_size)*2)+hidden_size:]
```

Then we have to use w1, b1, and input data to calculate the value in the hidden layer (z1) and then using these calculated values, b2 and w2 the output values (z2) are calculated.

Here is the script that we need to address this issue:

```
z2 = numpy.dot(w1, data) + numpy.transpose(numpy.tile(b1, (MNumTrainingExamples, 1)))

a2 = sigmoid(z2)
a=numpy.dot(w2, a2)
b= numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))
print('a=',a.shape)
print('b=',b.shape)
z3 = numpy.dot(w2, a2) + numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))

a3 = sigmoid(z3)
```

it is worth mentioning that, the transfer function sigmoid is call twice in this process.

For the next step we calculate the cost value:

```
JsumOfSquaredError = (1 / 2) * numpy.sum(numpy.power((a3 - data), 2)) / data.shape[1]

weightDecay = (lambda_ / 2) * (numpy.sum(numpy.power(w1, 2)) + numpy.sum(numpy.power(w2, 2)))
cost = JsumOfSquaredError + weightDecay
```

and gradient:

```
delta_a3 = numpy.multiply(-(data - a3), sigmoid_prime(z3))
```

```
    delta_a2 = numpy.multiply(numpy.dot(numpy.transpose(w2), delta_a3), sigmoid_prime(z2))

    w1_gradient = (numpy.multiply((numpy.dot(delta_a2, numpy.transpose(data)) / MNumTrainingExamples + lambda_), w1)).flatten()
    w2_gradient = (numpy.multiply((numpy.dot(delta_a3, numpy.transpose(a2)) / MNumTrainingExamples + lambda_), w2)).flatten()
    b1_gradient = (numpy.sum(delta_a2, axis=1) / MNumTrainingExamples).flatten()
    b2_gradient = (numpy.sum(delta_a3, axis=1) / MNumTrainingExamples).flatten()

    grad = numpy.concatenate((w1_gradient, w2_gradient, b1_gradient, b2_gradient))
```

For calculation of numeral estimation of the gradient the following script is used:

```
def compute_gradient_numerical_estimate(J, theta, epsilon=0.0001):
    """
    :param J: a loss (cost) function that computes the real-valued loss given parameters and data
    :param theta: array of parameters
    :param epsilon: amount to vary each parameter in order to estimate
                the gradient by numerical difference
    :return: array of numerical gradient estimate
    """

    gradient = numpy.zeros(theta.shape)
    numThetaRows = theta.shape[0]
    for i in(range(numThetaRows)):
     thetaPlusEpsilon = numpy.array(theta, dtype=numpy.float64)
     thetaPlusEpsilon[i] = theta[i] + epsilon
     thetaMinusEpsilon = numpy.array(theta, dtype=numpy.float64)
     thetaMinusEpsilon[i] = theta[i] - epsilon

     gradient[i] = (J(thetaPlusEpsilon)[0] - J(thetaMinusEpsilon)[0]) / (2 * epsilon)
    return gradient
```

And when we run debug checking we see this message which corroborates that the script is correct

```
========== DEBUG: checking gradient ==========
test_compute_gradient_numerical_estimate(): Start Test
    Testing that your implementation of
        compute_gradient_numerical_estimate()
        is correct
    Computing the numerical and actual gradient for 'simple_quadratic_function'
    The following two 2d arrays should be very similar:
        [ 38.  12.] [ 38.  12.]
    (Left: numerical gradient estimate; Right: analytical gradient)
    Norm of the difference between numerical and analytical num_grad:
        1.70974269866e-10
    (should be < 1.0e-09 ; I get about 1.7e-10)
test_compute_gradient_numerical_estimate(): DONE

Now test autoencoder_cost_and_grad() gradient against numerical estimate:
    Total number of parameters, theta.shape= (2353,)
```

Problem 4

We just copy some parts of problem 3 into this part.

```
def autoencoder_feedforward(theta, visible_size, hidden_size, data):
    """
    Given a definition of an autoencoder (including the size of the hidden
    and visible layers and the theta parameters) and an input data matrix
    (each column is an image patch, with 1 or more columns), compute
    the feedforward activation for the output visible layer for each
    data column, and return an output activation matrix (same format
    as the data matrix: each column is an output activation "image"
    corresponding to the data input).

    Once you have implemented the autoencoder_cost_and_grad() function,
    simply copy the part of the code that computes the feedforward activations
    up to the output visible layer activations and return that activation.
    You do not need to include any of the computation of cost or gradient.
    It is likely that your implementation of feedforward in your
    autoencoder_cost_and_grad() is set up to handle multiple data inputs,
    in which case your only task is to ensure the output_activations matrix
    is in the same corresponding format as the input data matrix, where
    each output column is the activation corresponding to the input column
    of the same column index.

    :param theta: the parameters of the autoencoder, assumed to be in this format:
            { W1, W2, b1, b2 }
            W1 = weights of layer 1 (input to hidden)
            W2 = weights of layer 2 (hidden to output)
            b1 = layer 1 bias weights (to hidden layer)
            b2 = layer 2 bias weights (to output layer)
    :param visible_size: number of nodes in the visible layer(s) (input and output)
    :param hidden_size: number of nodes in the hidden layer
    :param data: input data matrix, where each column is an image patch,
            with one or more columns
    :return: output_activations: an matrix output, where each column is the
            vector of activations corresponding to the input data columns
    """

    w1 = theta[:hidden_size*visible_size].reshape(hidden_size, visible_size)
    w2 = theta[hidden_size*visible_size:(hidden_size*visible_size)*2].reshape(visible_size, hidden_size)
    b1 = theta[(hidden_size*visible_size)*2:((hidden_size*visible_size)*2)+hidden_size]
    b2 = theta[((hidden_size*visible_size)*2)+hidden_size:]

    MNumTrainingExamples = data.shape[1]

    z2 = numpy.dot(w1, data) + numpy.transpose(numpy.tile(b1, (MNumTrainingExamples, 1)))

    a2 = sigmoid(z2)
    a=numpy.dot(w2, a2)
    b= numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))
    print('a=',a.shape)
    print('b=',b.shape)
    z3 = numpy.dot(w2, a2) + numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))

    a3 = sigmoid(z3)
    output_activations = a3   # implement

    return output_activations
```

Problem 5

We just copy some parts of problem 3 into this part.

```
def autoencoder_feedforward(theta, visible_size, hidden_size, data):
    """
    Given a definition of an autoencoder (including the size of the hidden
    and visible layers and the theta parameters) and an input data matrix
    (each column is an image patch, with 1 or more columns), compute
    the feedforward activation for the output visible layer for each
    data column, and return an output activation matrix (same format
    as the data matrix: each column is an output activation "image"
    corresponding to the data input).

    Once you have implemented the autoencoder_cost_and_grad() function,
    simply copy the part of the code that computes the feedforward activations
    up to the output visible layer activations and return that activation.
    You do not need to include any of the computation of cost or gradient.
    It is likely that your implementation of feedforward in your
    autoencoder_cost_and_grad() is set up to handle multiple data inputs,
    in which case your only task is to ensure the output_activations matrix
    is in the same corresponding format as the input data matrix, where
    each output column is the activation corresponding to the input column
    of the same column index.

    :param theta: the parameters of the autoencoder, assumed to be in this format:
            { W1, W2, b1, b2 }
            W1 = weights of layer 1 (input to hidden)
            W2 = weights of layer 2 (hidden to output)
            b1 = layer 1 bias weights (to hidden layer)
            b2 = layer 2 bias weights (to output layer)
    :param visible_size: number of nodes in the visible layer(s) (input and output)
    :param hidden_size: number of nodes in the hidden layer
    :param data: input data matrix, where each column is an image patch,
            with one or more columns
    :return: output_activations: an matrix output, where each column is the
            vector of activations corresponding to the input data columns
    """

    w1 = theta[:hidden_size*visible_size].reshape(hidden_size, visible_size)
    w2 = theta[hidden_size*visible_size:(hidden_size*visible_size)*2].reshape(visible_size, hidden_size)
    b1 = theta[(hidden_size*visible_size)*2:((hidden_size*visible_size)*2)+hidden_size]
    b2 = theta[((hidden_size*visible_size)*2)+hidden_size:]

    MNumTrainingExamples = data.shape[1]

    z2 = numpy.dot(w1, data) + numpy.transpose(numpy.tile(b1, (MNumTrainingExamples, 1)))

    a2 = sigmoid(z2)
    a=numpy.dot(w2, a2)
    b= numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))
    print('a=',a.shape)
    print('b=',b.shape)
    z3 = numpy.dot(w2, a2) + numpy.transpose(numpy.tile(b2, (MNumTrainingExamples, 1)))

    a3 = sigmoid(z3)
    output_activations = a3   # implement

    return output_activations
```

Problem 6

This is almost the same as problem 3 but we have to just add some lines to our script. This new function actually has more two inputs, rho_ : the target sparsity limit for each hidden node activation, beta_ : controls the weight of the sparsity penalty term relative to other loss components

```
def autoencoder_cost_and_grad_sparse(theta, visible_size, hidden_size, lambda_, rho_, beta_, data):
    """
    Version of cost and grad that incorporates the hidden layer sparsity constraint
        rho_ : the target sparsity limit for each hidden node activation
        beta_ : controls the weight of the sparsity penalty term relative
              to other loss components

    The input theta is a 1-dimensional array because scipy.optimize.minimize expects
    the parameters being optimized to be a 1d array.
    First convert theta from a 1d array to the (W1, W2, b1, b2)
    matrix/vector format, so that this follows the notation convention of the
    lecture notes and tutorial.
    You must compute the:
        cost : scalar representing the overall cost J(theta)
        grad : array representing the corresponding gradient of each element of theta
    """

    ### YOUR CODE HERE ###
    W1 = theta[0:hidden_size * visible_size].reshape(hidden_size, visible_size)
    W2 = theta[hidden_size * visible_size:2 * hidden_size * visible_size].reshape(visible_size, hidden_size)
    b1 = theta[2 * hidden_size * visible_size:2 * hidden_size * visible_size + hidden_size]
    b2 = theta[2 * hidden_size * visible_size + hidden_size:]

    # Number of training examples
    m = data.shape[1]

    # Forward propagation
    z2 = W1.dot(data) + numpy.tile(b1, (m, 1)).transpose()
    a2 = sigmoid(z2)
    z3 = W2.dot(a2) + numpy.tile(b2, (m, 1)).transpose()
    h = sigmoid(z3)

    # Sparsity
    rho_hat = numpy.sum(a2, axis=1) / m
    rho = numpy.tile(rho_, hidden_size)

    # Cost function
    cost = numpy.sum((h - data) ** 2) / (2 * m) + \
        (lambda_ / 2) * (numpy.sum(W1 ** 2) + numpy.sum(W2 ** 2)) + \
        beta_ * numpy.sum(KL_divergence(rho, rho_hat))

    # Backprop
    rho_delta = numpy.tile(- rho / rho_hat + (1 - rho) / (1 - rho_hat), (m, 1)).transpose()

    delta3 = -(data - h) * sigmoid_prime(z3)
    delta2 = (W2.transpose().dot(delta3) + beta_ * rho_delta) * sigmoid_prime(z2)
    W1grad = delta2.dot(data.transpose()) / m + lambda_ * W1
    W2grad = delta3.dot(a2.transpose()) / m + lambda_ * W2
    b1grad = numpy.sum(delta2, axis=1) / m
    b2grad = numpy.sum(delta3, axis=1) / m

    # After computing the cost and gradient, we will convert the gradients back
    # to a vector format (suitable for minFunc).  Specifically, we will unroll
    # your gradient matrices into a vector.
    grad = numpy.concatenate((W1grad.reshape(hidden_size * visible_size),
                              W2grad.reshape(hidden_size * visible_size),
```

```
                b1grad.reshape(hidden_size),
                b2grad.reshape(visible_size)))

    return cost, grad
```