ISTA 421/521 – Homework 5

Due: Monday, December 15, 5pm

16 points total

STUDENT NAME

Undergraduate / Graduate

Instructions

In this assignment you are required to work with the following set of python scripts: `utils.py, train_ml_class.py`. Be sure that the provided auxiliary scripts are in the same folder: `sample_images.py, gradient.py, load_MNIST.py, display_network.py`

You will run the script train_ml_class.py. Since some parts are not implemented, it will break at the beginning. We recommend you use the `sys.exit()` command after each part that you implement – this will gracefully exit the script at that point. You must first `import sys` in order to make the `sys` module available.

Some problems require you to include a plot of the weights that you found with the autoencoder.

To run the problems, be sure to download the MNIST dataset from:

- Training Images: `http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz`
- Training Labels: `http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz`

We have provided code to load these files into memory, you just need to set the right filepath when you call the function `load_MNIST_images`.

The problems are adapted from the UFDL demo at Stanford:
`http://ufldl.stanford.edu/wiki/index.php/Neural_Networks`.

Feel free to follow that tutorial to help you through your implementation.

1. [3 points] Exercise 1: Load and visualize MNIST:

   You will need to the files `display_network.py` and `load_MNIST.py` into the *same* directory. Run the `train_ml_class.py` code to load and visualize the MNIST dataset. Currently the code loads the training images, loads 10K images and visualizes 100.

   Modify the loading part in the train_ml_class.py so you display 10, 50 and 100 subsets of MNIST.

   Tip: Use the `sys.exit()` command to stop execution of the code, since it will break if you have not completed the other parts of the assignment. You must first `import sys` in order to make the `sys` module available.

   Solution:

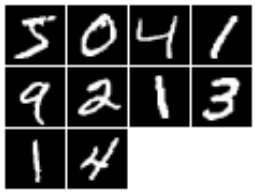

Figure 1: 10 subsets of MNIST

Figure 2: 50 subsets of MNIST

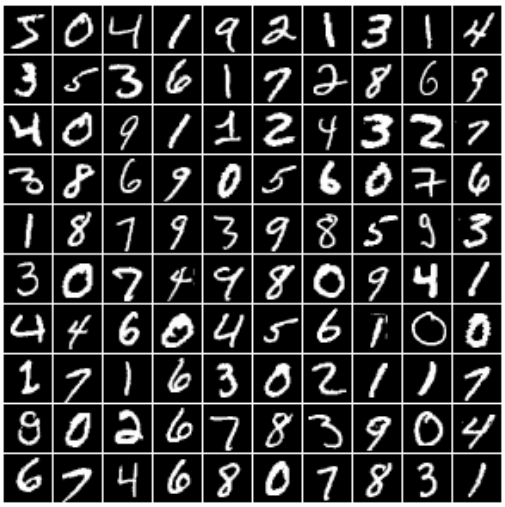

Figure 3: 100 subsets of MNIST

2. [4 point] Exercise 2: Write the initialization script for the parameters in an autoencoder with a single hidden layer:

   In class we learned that a NN'??s parameters are the weights $\mathbf{w}$ and the offset parameters b. Write a script where you initialize them given the size of the hidden layer and the visible layer.

   Then reshape them and concatenate them so they are all allocated in a single parameter vector.

   Example: For an autoencoder with visible layer size 2 and hidden size 3, we would have 6 weights ($\mathbf{w}_1$) from the visible layer to the hidden layer and 6 more weights ($\mathbf{w}_2$) from the hidden layer to the output layer; there are also one bias term (b_1) with 3 parameters, one for each of the hidden nodes, and another bias term (b_2) with 2 parameters to the output layer. This will make a total of 6+6+3+2 = 17 parameters. The output of your script should be a vector of 1x17 elements, with order [$\mathbf{w}_1$, $\mathbf{w}_2$, b_1, b_2].

   Tip: use the `np.concatenate` function to put the vectors together in the desired order, and the `np.reshape` function to put the result vector in the shape $1 \times size$.

   Solution:

   Also, the solution for this problem is available in the utils_hw.py file, which has been attached with this pdf file.

```
def initialize(hidden_size, visible_size):
    # we'll choose weights uniformly from the interval [-r, r] following what we saw in class

    #as we saw in the class: n_in + n_out = hidden_size + visible_size
    my_guess_range = math.sqrt(6) / math.sqrt(hidden_size + visible_size + 1)

    #In our example, hidden_size = 3 + visible_size = 2 => it will generates
    #number between -1 and 1 since myguess_range = 1 (in this case)
    my_w1 = 2 * my_guess_range * np.random.rand((hidden_size * visible_size)) - my_guess_range
    #we will reshape, so it will be 1x(hidden_size + visible_size)
    my_w1 = my_w1.reshape(hidden_size * visible_size)

    #the same thing for w2
    my_w2 = 2 * my_guess_range * np.random.rand((hidden_size * visible_size)) - my_guess_range
    #we will reshape, so it will be 1x(hidden_size + visible_size)
    my_w2 = my_w2.reshape(hidden_size * visible_size)

    #b_1 is our first bias unit ( so it is the input from the visible layer to the hidden layer)
    #it will have "hidden_size" arrows
    my_b1 = np.ones(hidden_size)
    #reshapping -> 1xlength array
    my_b1 = my_b1.reshape(hidden_size)

    #on the other hand b_2 will have visible_size arrows since it is the
    #input from the hidden layer to visible layer
    my_b2 = np.ones(visible_size)
    #reshapping -> 1xlength array
    my_b2 = my_b2.reshape(visible_size)

    #concatenating in order to we have a 1X(length) (ONE LINE) array
    #we will have [w1, w2, b1, b2]
    theta = np.concatenate((my_w1, my_w2, my_b1, my_b2))
    #print(len(theta))

    #print("w1 shape ", my_w1.shape(0))

    return theta
```

3. [4 point] Exercise 3: Write the cost function for an autoencoder as well as the gradient for each of the parameters.

   In class we learned that we can use gradient descent to train a NN. In this exercise we will use a more refined version of this called LBFGS (`http://en.wikipedia.org/wiki/Limited-memory_BFGS`), which is readily implemented in the optimization library of scipy. For your convenience the implementation is ready to run.

   To use it, you need to define functions for the cost and for the gradient of each parameter. Do this based on the error functions δ that we defined in the slides in class.

   The functions use the data to do a forward pass of the network, calculates the overall error, and then calculate the gradient per each parameter.

   Tip: In the code, there is a flag called `debug`; if set to `True`, it will run a debugging code to check if your gradient is correct.

   You might want to load fewer images in this step, so you do not spend too much time waiting for all the examples.

   The gradient has to be a matrix of the same size as the parameter matrix, while the cost has to be the evaluation of the cost after data has passed through.

   Solution: Also, the solution for this problem is available in the utils_hw.py file, which has been attached with this pdf file.

```
def sparse_autoencoder_cost(theta, visible_size, hidden_size,
                            lambda_, data):
    # The input theta is a vector (because minFunc expects the parameters to be a vector).
    # We first convert theta to the (W1, W2, b1, b2) matrix/vector format, so that this
    # follows the notation convention of the lecture notes.
        # The input theta is a vector (because minFunc expects the parameters to be a vector).
    # We first convert theta to the (W1, W2, b1, b2) matrix/vector format, so that this
    # follows the notation convention of the lecture notes.

    weights_length = hidden_size * visible_size
    b1_lenth = hidden_size
    b2_length = visible_size

    #unwrapping our arrays
    my_W1 = theta[0:weights_length].reshape(hidden_size, visible_size)

    my_W2 = theta[weights_length:2*weights_length].reshape(visible_size, hidden_size)

    my_b1 = theta[2*weights_length:2*weights_length + b1_lenth]

    my_b2 = theta[2*weights_length + b1_lenth:] # = b2_length

    # total of columns (data to be trained)
    m = data.shape[1]


    #FP
    my_z2 = my_W1.dot(data) + np.tile(my_b1, (m, 1)).transpose() #np.tile(my_b1, (m, 1)).transpose() =
    my_a2 = sigmoid(my_z2)
    my_z3 = my_W2.dot(my_a2) + np.tile(my_b2, (m, 1)).transpose()
    my_h = sigmoid(my_z3)



    #Our cost func.
    cost = (0.5 * np.sum(np.power((my_h - data), 2)) / m) + ((lambda_ / 2.0) * \
        ( np.sum(np.power(my_W1, 2)) + np.sum(np.power(my_W2, 2)) ) )
```

```
    #deltas
    delta_output_layer = -(data - my_h) * sigmoid_derivative(my_z3)
    delta_output_layer = -(data - my_h) * (my_h *(1 - my_h))
    delta_middle_layer = my_W2.transpose().dot(delta_output_layer) * sigmoid_derivative(my_z2)
    delta_middle_layer = (my_W2.transpose().dot(delta_output_layer)) * (my_a2*(1 - my_a2))


  #grad arrays
    my_W2_grad = delta_output_layer.dot(my_a2.transpose())
    my_W1_grad = delta_middle_layer.dot(data.transpose())
    my_b1_grad =  np.sum(delta_middle_layer, axis=1) / m
    my_b2_grad =  np.sum(delta_output_layer, axis=1) / m


    #shapping grad arrays
    my_W1_grad = my_W1_grad.reshape(hidden_size * visible_size)
    my_W2_grad = my_W2_grad.reshape(hidden_size * visible_size)
    my_b1_grad = my_b1_grad.reshape(hidden_size)
    my_b2_gra = my_b2_grad.reshape(visible_size)


    #concatenating gra arrays
    grad = np.concatenate((my_W1_grad, my_W2_grad, \
                    my_b1_grad, my_b2_grad))



    return cost, grad
```

4. [3 point] Exercise 4:

   If your gradient is correct, now load the 10,000 images and run the code.

   Test the code and change the size of the hidden layer to 50, 100, 150 and 250. By the end, the code creates an image called `weights` (which overwrites the one in ex. 1) and it prints the weights obtained after training.

   Report those weights and comment of the difference between using different sizes in the hidden layer.

   Solution:

   Using hidden layers sizes smaller than the size of the visible layers we are removing some of the redundant information contained in images (e.g., requiring less storage space and less time to transmit). On the other hand, if we use too small sizes of hidden layers we, sometimes, cannot "restore"/"uncompress" the images.

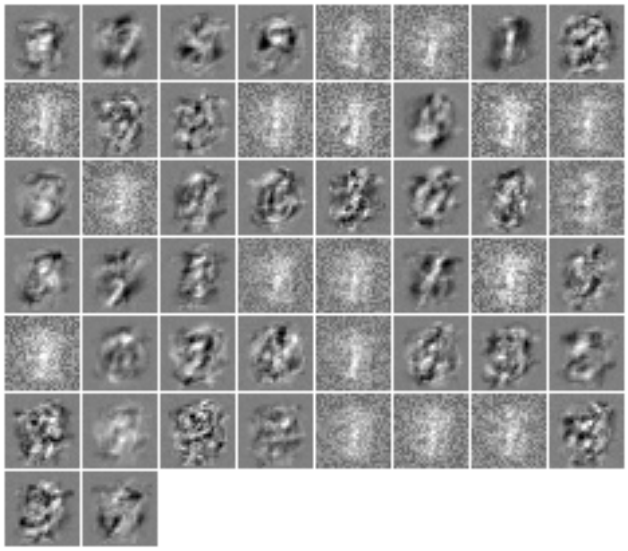

Figure 4: 50 hidden layers

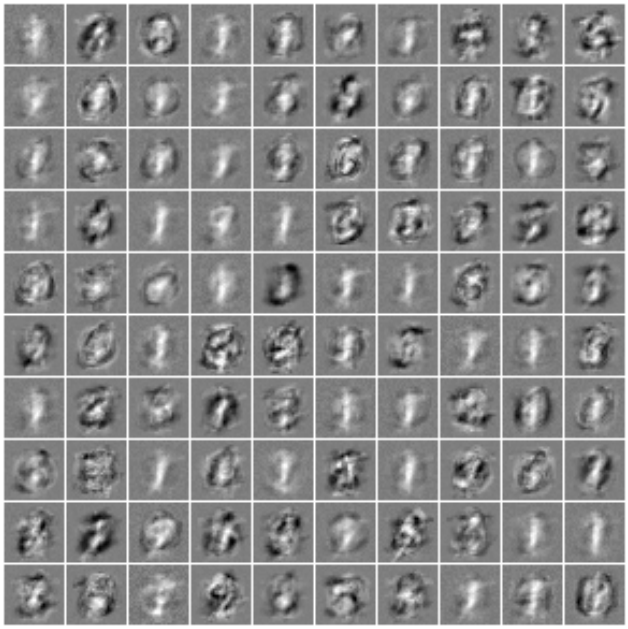

Figure 5: 100 hidden layers

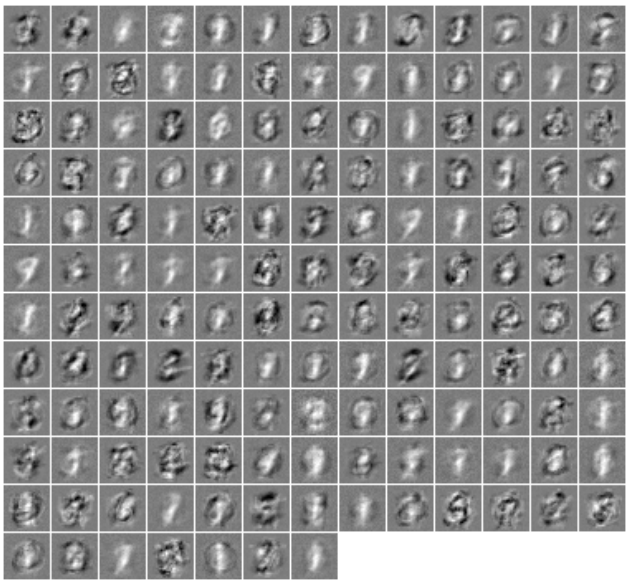

Figure 6: 150 hidden layers

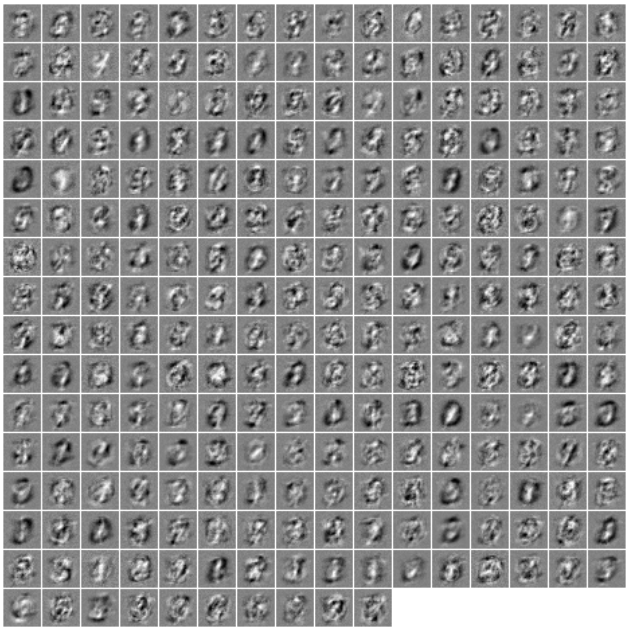

Figure 7: 250 hidden layers

5. [2 point] Exercise 5:

   Once everything is running, run the code `stacked_ae_hw.py`, which implements the stacked autoencoder concept that we saw in class on Thursday. At the end you should have a report of your results accuracy. Change the number of training examples to 100, 500, 1000, 10000 and report the results here.

   Tip: You might want to check how much time it takes to run on your computer before-hand and consider leaving it overnight.

   Solution:

   Results for 100 images

   ```
   Before fine-tuning accuracy: 39.68%
   After fine-tuning accuracy: 45.44%
   ```

   Results for 500 images

   ```
   Before fine-tuning accuracy: 42.75%
   After fine-tuning accuracy: 67.02%
   ```

   Results for 1000 images

   ```
   Before fine-tuning accuracy: 34.02%
   After fine-tuning accuracy: 68.22%
   ```

   Results for 10000 images

   ```
   Before fine-tuning accuracy: 46.57%
   After fine-tuning accuracy: 71.00%
   ```