ISTA 421/521 – Homework 5

Due: Monday, November 30, 5pm

16 points total

STUDENT NAME

Undergraduate / Graduate

Instructions

In this assignment you will work directly with the following three python scripts: `utils.py, train_ml_class.py, and gradient.py`. You will fill in portions of these files that are currently unimplemented, following the instructions in the exercises, and you will include all three in your final submission.

Three additional files implement the stacked autoencoder: `softmax.py`, `stacked_autoencoder.py`, and `stacked_ae_hw.py`. You will need to complete Exercises 1-4 before you can use these, but you will not need to implement any additional code for these and do not need to include them in your final submission. Instructions for their use are in Exercise 5. It is worth looking through them to make sure you understand everything that is going on (especially after completing Exercises 1-4).

Finally, also included in the released code are the following auxiliary files, which should appear in the same directory: `load_MNIST.py` and `display_network.py`. You will not make any changes to these files and do not need to include these in your final submission.

`display_network.py` relies on the Python Image Library (PIL); you will need to install this for display to work. I recommend installing `pillow`, a fork of the PIL project: `https://python-pillow.github.io/`. You can install pillow using `pip`: `$ pip install pillow`

In this assignment, we will use two databases from the MNIST dataset, which you can get here:

- 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`

The top-level script to run is `train_ml_class.py`. Initially, many parts of the script are not implemented. On line 47 there is the command `sys.exit()`, which will gracefully exit the script at that point. As you implement more of the assignment, you can move this command further down the script file. When you have completed everything, you can remove the command entirely.

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

We have provided the function `load_MNIST_images` in `load_MNIST.py` to load these files into memory. Be sure to adjust the filepaths so that they point to where you have saved the MNIST data.

These problems are adapted from the UFDL demo/tutorial from Stanford:
`http://ufldl.stanford.edu/wiki/index.php/Neural_Networks`.
You can follow the tutorial to walk you through your implementation.

NOTE: You will see that in the code the steps of this tutorial are numbered, to indicate the steps of implementing the NN framework; these steps do not necessarily correspond to the exercise numbers in this document. It should be clear from the exercises below which part of the code you'll be working on.

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

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

   Modify the script `train_ml_class.py` so you display 10, 50 and 100 subsets of MNIST. The function `display_network` takes an optional filename (string) as the second argument. Include in your written solution a figure for each of the three subsets.

   Solution:

2. [3 points] Exercise 2: Write the initialization script for the parameters in an autoencoder with a single hidden layer:

   You will implement this functionality in the function `initialize` in `utils_hw.py`.

   In class we learned that a NN's parameters are the weights $\mathbf{w}$ and the offset (bias) 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 bias terms (b) with 3 parameters each, one (b_1) 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:

3. [8 points] Exercise 3: Write the cost function for an autoencoder as well as the gradient for each of the parameters.

   In this exercise you will work on implementing two functions: `sparse_autoencoder_cost` in `utils_hw.py` and `compute_gradient` in `gradient.py`. (In the script `traim_ml_class.py`, this exercise comprises Steps 2 and 3)

   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 gradient descent 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 (`utils_hw.sparse_autoencoder_cost`) and the gradient (`gradient.compute_gradient`) of each parameter. Do this based on the δ error functions 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.

   NOTE: On line 28 of `train_ml_class.py`, 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:

4. [2 points] Exercise 4:

   If your gradient, as tested in Exercise 3, is correct, now load the 10,000 images and run the code.

   Now run the code with different sizes of hidden layer. In particular, run the `train_ml_class.py` script with hidden layers of (a) 50, (b) 100, (c) 150 and (d) 250.

   After each run, the code saves an image in your working directory called `weights.png` (this occurs in Step 5 of `traim_ml_class.py`; NOTE: this overwrites the weights.png file produced by default in Exercise 1 and will overwrite after each execution of the script, so you will need to save/rename the weights image for each of your hidden-layer sizes or use the second optional argument to specify the filename). This generated image represents the weight magnitudes for each parameter, as obtained after training.

   Report those weights (include figures for each hidden layer size) and *comment* of the difference between using different sizes in the hidden layer. Do you see any patterns?

   Solution:

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.

   NOTE: This computation is expensive and will take a long time to execute. Expect to have to run on your computer (possibly overnight).

   Solution: