# Sparse Autoencoder Exercise

Sai Ganesh and Shaowei Lin

13 Oct 2014

## 1 Instructions

Complete the exercise outlined below. For enquiries, please contact Sai or Shaowei.

1. Go to ULFDL tutorial stanford link given here:

   `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`

   Read the notes in the section on the Sparse Autoencoder.

2. For the exercise, *do not* use the Matlab starter files given in the website. Instead, please use the Python starter file `SparseAutoencoder.py`. Follow the instructions outlined in the remainder of this document to complete the exercise.

## 2 Writing machine learning algorithms in Python

1. For the Python implementation, we will use the `scikit-learn` format. In this format, the data comes in the form of a matrix where each row is a sample, while in the Matlab implementation, each column is a sample. The reason for this is that in JSON format (and many other text-based matrix formats), it is easier to add a row to a matrix than to add a column. As the number of samples increases, it is more efficient to use this convention. For this reason, the `IMAGES.npy` file contains a matrix that is the transpose of that in the `IMAGES.mat` file.

2. Because of the above transposed notation, many other matrix operations will be transposed. For instance, if the data matrix is $X$, and $A$ is a linear transformation that we want to apply to each row of $X$ and $Y$ is the matrix where each row is an output of the above transformation, then as a matrix product we have $Y = XA$. While this convention may seem strange at first, there are streams of mathematics which prefer this convention (e.g. postfix notation).

3. We break down the tasks into small atomic functions, and try to use classes and objects to bind variables and functions together where possible. The final solution can then be imported as a library in other projects. The *main program* which starts with

   ```
   if __name__ == "__main__":
   ```

   will not be imported with the rest of the library, and should only be used for tests.

# 3 Sparse autoencoder implementation

In this problem set, you will implement the sparse autoencoder algorithm, and show how it discovers that edges are a good representation for natural images. (Images provided by Bruno Olshausen.) The sparse autoencoder algorithm is described in the lecture notes found on the course website.

In the file `sparseae_exercise.zip`, we have provided some starter code in Python. You should write your code at the places indicated in the files ("YOUR CODE HERE"). You have to complete the following functions: `sampleIMAGES()`, `sparseAutoencoderCost()`, `computeNumericalGradient()`. The main program then shows how these functions are used.

Specifically, in this exercise you will implement a sparse autoencoder, trained with $8 \times 8$ image patches using the L-BFGS optimization algorithm.

## 3.1 Step 1: Generate training set

The first step is to generate a training set. To get a single training example $x$, randomly pick one of the 10 images, then randomly sample an $8 \times 8$ image patch from the selected image, and convert the image patch (either in row-major order or column-major order; it doesn't matter) into a 64-dimensional vector to get a training example $x \in \mathbb{R}^{64}$.

Complete the code in `sampleIMAGES()`. Your code should sample 10000 image patches and concatenate them into a $10000 \times 64$ matrix.

To make sure your implementation is working, run the code in Step 1 of the main program. This should result in a plot of a random sample of 200 patches from the dataset.

Implementational tip: When we run our implemented `sampleImages()`, it takes under 5 seconds. If your implementation takes over 30 seconds, it may be because you are accidentally making a copy of an entire $512 \times 512$ image each time you're picking a random image. By copying a $512 \times 512$ image 10000 times, this can make your implementation much less efficient. While this doesn't slow down your code significantly for this exercise (because we have only 10000 examples), when we scale to much larger problems later this quarter with 106 or more examples, this will significantly slow down your code. Please implement `sampleIMAGES()` so that you aren't making a copy of an entire $512 \times 512$ image each time you need to cut out an $8 \times 8$ image patch.

## 3.2 Step 2: Sparse autoencoder objective

Implement code to compute the sparse autoencoder cost function $J_{\text{sparse}}(W, b)$ (Section 3 of the lecture notes) and the corresponding derivatives of $J_{\text{sparse}}$ with respect to the different parameters. Use the sigmoid function for the activation function,

$$f(z) = \frac{1}{1 + e^{-z}}.$$

In particular, complete the code in `sparseAutoencoderCost()`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \mathbb{R}^{s_1 \times s_2}, W^{(2)} \in \mathbb{R}^{s_2 \times s_3}$ and vectors $b^{(1)} \in \mathbb{R}^{s_2}, b^{(2)} \in \mathbb{R}^{s_3}$. However, for subsequent notational convenience, we will "unroll" all of these parameters into a very long parameter vector $\theta$ with $s_1 s_2 + s_2 s_3 + s_2 + s_3$ elements. The code for converting between the $(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$ and the $\theta$ parameterization is already provided.

Implementational tip: The objective $J_{\text{sparse}}(W, b)$ contains 3 terms, corresponding to the squared error term, the weight decay term, and the sparsity penalty. You are welcome to implement this however you want, but for ease of debugging, you might implement the cost function and derivative

computation (backpropagation) only for the squared error term first (this corresponds to setting $\lambda = \beta = 0$), and implement the gradient checking method in the next section to first verify that this code is correct. Then only after you have verified that the objective and derivative calculations corresponding to the squared error term are working, add in code to compute the weight decay and sparsity penalty terms and their corresponding derivatives.

### 3.3 Step 3: Gradient checking

Following Section 2.3 of the lecture notes, implement code for gradient checking. Specifically, complete the code in `computeNumericalGradient()`. Please use `EPSILON` $= 10^{-4}$ as described in the notes.

We have also provided code in `checkNumericalGradient()` for you to test your code. This code defines a simple quadratic function $h : \mathbb{R}^2 \mapsto \mathbb{R}$ given by $h(x) = x_1^2 + 3x_1x_2$, and evaluates it at the point $x = (4, 10)$. It allows you to verify that your numerically evaluated gradient is very close to the true (analytically computed) gradient.

After using `checkNumericalGradient()` to make sure that your implementation is correct, next use `computeNumericalGradient()` to make sure that your `sparseAutoencoderCost()` is computing derivatives correctly. For details, see Steps 3 in the main program. We strongly encourage you not to proceed to the next step until you've verified that your derivative computations are correct.

Implementational tip: If you are debugging your code, performing gradient checking on smaller models and training sets (e.g., 10 training examples and 1-2 hidden units) may speed things up.

### 3.4 Step 4: Train the sparse autoencoder

Now that you have code that computes $J_{\text{sparse}}$ and its derivatives, we're ready to minimize $J_{\text{sparse}}$ with respect to its parameters, and thereby train our sparse autoencoder.

We will use the L-BFGS algorithm. This is provided to you in a function called

`scipy.optimize.fmin_l_bfgs_b`

which we import as `minimize` in the starter code. (For the purpose of this assignment, you only need to call `minimize` with the default parameters. You do not need to know how L-BFGS works.) We have already provided code in the main program (Step 4) to call `minimize`. The `minimize` function assumes that the parameters to be optimized are in a long vector; so we will use the $\theta$ parameterization rather than the $(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$ parameterization when passing our parameters to it.

Train a sparse autoencoder with 64 input units, 25 hidden units, and 64 output units. In our starter code, we have provided a function for initializing the parameters. We initialize the biases $b_i^{(l)}$ to zero, and the weights $W_{ij}^{(l)}$ to random numbers drawn uniformly from the interval

$$\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}\right],$$

where $n_{\text{in}}$ is the fan-in (the number of inputs feeding into a node) and $n_{\text{out}}$ is the fan-in (the number of units that a node feeds into).

The values we provided for the various parameters $(\lambda, \beta, \rho, \ldots)$ should work, but feel free to play with different settings of the parameters as well.

Implementational tip: Once you have your backpropagation implementation correctly computing the derivatives (as verified using gradient checking in Step 3), when you are now using it with L-BFGS to optimize $J_{\text{sparse}}(W, b)$, make sure you're not doing gradient-checking on every step. Backpropagation

can be used to compute the derivatives of $J_{\text{sparse}}(W, b)$ fairly efficiently, and if you were additionally computing the gradient numerically on every step, this would slow down your program significantly.

### 3.5 Step 5: Visualization

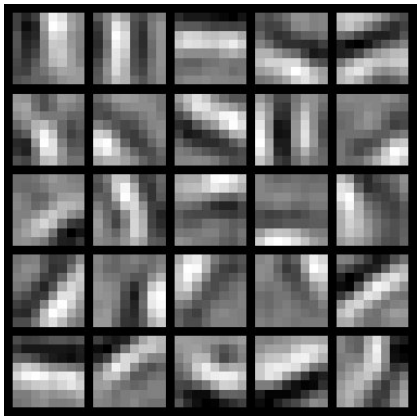
After training the autoencoder, use `displayNetwork()` to visualize the learned weights (Step 5 of the main program) and to save the visualization to a file `weights.jpg` (which you will submit together with your code to show that your code is performing correctly).

### 3.6 Step 6: Classes and objects

Now that we have tested the individual functions, we may bind them together in a `SparseAutoencoder` class that will perform the necessary book-keeping of the parameters and hyperparameters. Our class will follow the `scikit-learn` format by implementing `fit` and `predict` functions for training and performing estimation with the algorithm.

## 4 Results

To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:



Our implementation took around 5 minutes to run on a fast computer. In case you end up needing to try out multiple implementations or different parameter values, be sure to budget enough time for debugging and to run the experiments you'll need.

Also, by way of comparison, here are some visualizations from implementations that we do not consider successful (either a buggy implementation, or where the parameters were poorly tuned):