CS 294A/W, Winter 2011
Programming Assignment: Sparse Autoencoder

All students taking CS294A/W are required to successfully complete this programming assignment by **5:30pm on Wednesday, January 12**. Please submit your solution via email to cs294a-qa@cs.stanford.edu.

Collaboration policy: This assignment should be done individually. It is okay to discuss sparse autoencoders and neural networks (e.g., the material in the lecture notes) with others. But please do not discuss anything specific to this programming assignment or to your implementation with anyone else. Please also do not look at anyone else's code (including source code found on the internet), or show your code to anyone else. If you have questions about the assignment or would like help, please email us at cs294a-qa@cs.stanford.edu.

The collaboration policy stated above applies only to this programming assignment. Once you've submitted your solution, for the research project you'll be doing you're welcome (and encouraged) to talk to anyone about your work and use any open-source/etc. code you find on the internet (with attribution).

1 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.[1] The sparse autoencoder algorithm is described in the lecture notes found on the course website.

In the file `cs294a_2011assgn.zip`, we have provided some starter code in Matlab. You should write your code at the places indicated in the files ("`YOUR CODE HERE`"). You have to complete the following files: `sampleIMAGES.m, sparseAutoencoderCost.m, computeNumericalGradient.m`. The starter code in `train.m` shows how these functions are used.

Specifically, in this exercise you will implement a sparse autoencoder, trained with 8x8 image patches using the L-BFGS optimization algorithm.

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 8x8 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.m`. Your code should sample 10000 image patches and concatenate them into a 64x10000 matrix.

To make sure your implementation is working, run the code in "Step 1" of `train.m`. 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

[1] Images provided by Bruno Olshausen.

making a copy of an entire 512x512 image each time you're picking a random image. By copying a 512x512 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 10^6 or more examples, this will significantly slow down your code. Please implement `sampleIMAGES` so that you aren't making a copy of an entire 512x512 image each time you need to cut out an 8x8 image patch.

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_{sparse} with respect to the different parameters. Use the sigmoid function for the activation function, $f(z) = 1/(1 + \exp(-z))$. In particular, complete the code in `sparseAutoencoderCost.m`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \mathbb{R}^{s_1 \times s_2}$, $W^{(2)} \in \mathbb{R}^{s_2 \times s_3}$ 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 θ 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 θ parameterization is already provided in the starter code.

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're 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.

Step 3: Gradient checking

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

We've also provided code in `checkNumericalGradient.m` 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 $x = (4, 10)^T$. It allows you to verify that your numerically evaluated gradient is very close to the true (analytically computed) gradient.

After using `checkNumericalGradient.m` to make sure your implementation is correct, next use `computeNumericalGradient.m` to make sure that your `sparseAutoencoderCost.m` is computing derivatives correctly. For details, see Steps 3 in `train.m`. 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 smaller training sets (e.g., using only 10 training examples and 1-2 hidden units) may speed things up.

Step 4: Train the sparse autoencoder

Now that you have code that computes J_{sparse} and its derivatives, we're ready to minimize J_{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 `minFunc`,[2] included in the starter code. (For the purpose of this assignment, you only need to call minFunc with the default parameters. You do not need to know how L-BFGS works.) We have already provided code in `train.m` (Step 4) to call `minFunc`. The `minFunc` code assumes that the parameters to be optimized are a long parameter vector; so we will use the "θ" 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_{in} is the fan-in (the number of inputs feeding into a node) and n_{out} is the fan-in (the number of units that a node feeds into).

The values we provided for the various parameters (λ, β, ρ, etc.) should work, but feel free to play with different settings of the parameters as well.

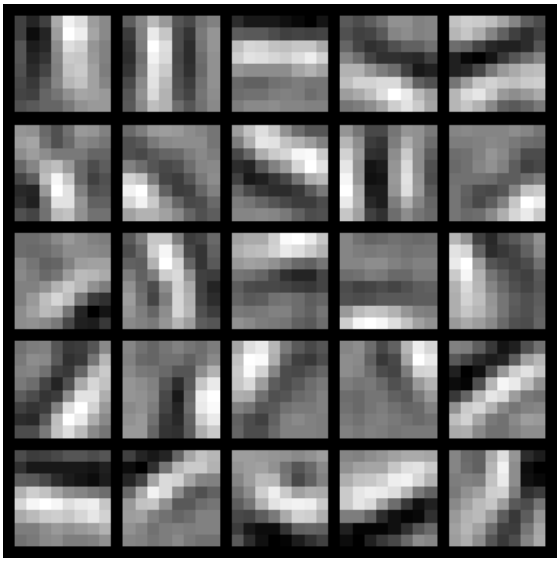
Step 5: Visualization

After training the autoencoder, use `display_network.m` to visualize the learned weights. (See `train.m`, Step 5.) Run "`print -djpeg weights.jpg`" to save the visualization to a file "`weights.jpg`" (which you will submit together with your code).

2 Results

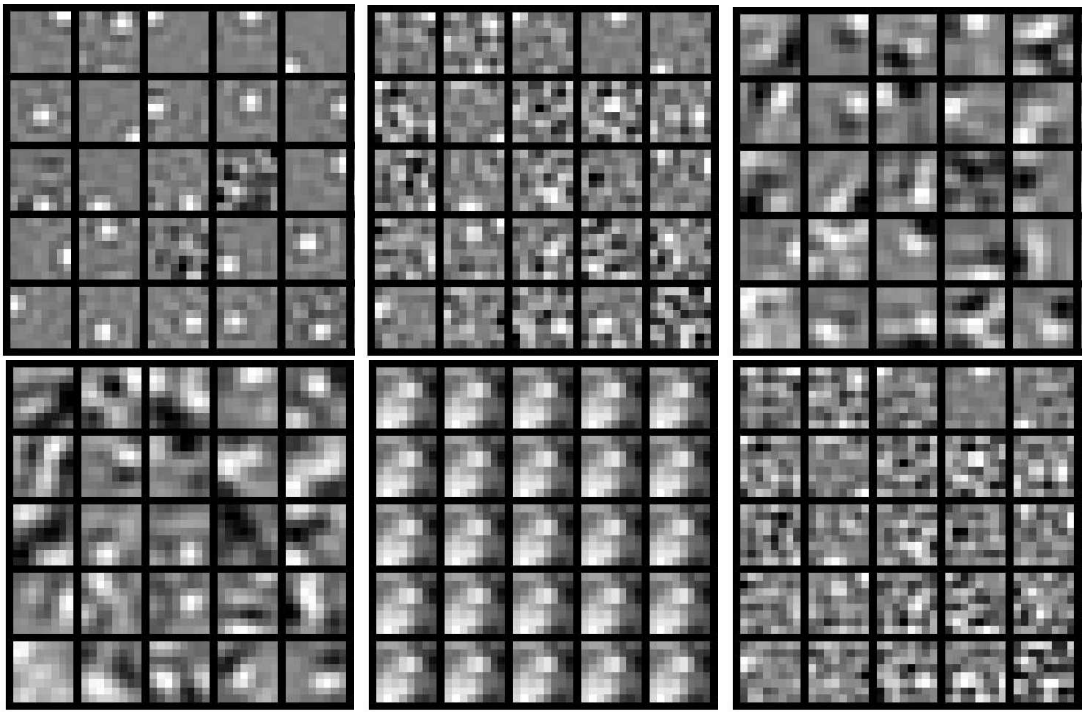
To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:

[2] Code provided by Mark Schmidt.



Our implementation took around 10 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):



3 What to submit

Please submit your solution by emailing a tar file to cs294a-qa@cs.stanford.edu.

Your submission should include `sampleIMAGES.m`, `sparseAutoencoderCost.m`, `computeNumericalGradient.m`, `weights.jpg` (and, if you wish, other files as well). You can also include a `README` if there're notes that you'd like us to look at.

A good solution should should run and produce weights that look like "edge detectors" (as in the image on the previous page).

4 Contact

This programming assignment is (by design) more open-ended than most assignments you might have seen in other classes (including CS221 and CS229). If you have questions, don't understand parts of it, find parts of it ambiguous, or need help with Matlab, don't hesitate to email us at cs294a-qa@cs.stanford.edu to ask for help.

In case you have questions about the lecture notes or want clarifications pertaining to the programming assignment, we will also have office hours from 9-10am on Friday, Jan 7th and 9-10am on Monday, Jan 10th in Gates 110.