STA 414/2104, Spring 2013 — Assignment #4 (now with addendum)

Due at the start of class on April 4. Please hand it in on 8 1/2 by 11 inch paper, stapled in the upper left, with no other packaging.

This assignment is to be done by each student individually. You may discuss it in general terms with other students, but the work you hand in should be your own. In particular, you should not leave any discussion with someone else with any written notes (either on paper or in electronic form).

In this assignment, you will modify the R functions for training multilayer perceptron networks that were demonstrated in class so that they can be used to train auto-encoder networks for reducing the dimensionality of data. You will then apply auto-encoder networks that reduce to one dimension to the first two synthetic datasets you used for Assignment 3, and apply an auto-encoder network that reduces to two dimensions to a real dataset of images of handwritten digits.

The multilayer perceptron functions provided on the webpage are for networks with one hidden layer, and a single output unit. You will need to modify these functions to instead handle networks with three hidden layers, which have multiple outputs. These more complex networks have matrices of weights, which you should call w1, w2, w3, and w4, that connect the input layer to hidden layer 1, hidden layer 1 to hidden layer 2, hidden layer 2 to hidden layer 3, and hidden layer 3 to the outputs. There are also vectors of "biases" (intercepts) for the hidden and output units, which you should call w1_0, w2_0, w3_0, and w4_0. The function you should try to minimize during training is total squared error, over all training cases and all outputs.

You will need to modify the mlp_skeleton function to work with the new groups of parameters. Its "m" argument will now be a vector of three numbers, giving the number of hidden units in each of the three hidden layers, and it in addition to its "p" argument giving the number of inputs, it will need a "q" argument giving the number of outputs. The "m" argument for the mlp_train function will also now be a vector of three numbers, and the argument for mlp_train giving the responses in training cases will now be a matrix. In the demo version on the webpage, mlp_train returns a list that includes a matrix ("P") of outputs for all training cases using the networks at every training iteration. You may omit this in your program — it would need to become a three-dimensional array rather than a matrix, and might occupy an excessive amount of memory. The list returned should still include the "E" and "W" elements.

When training an auto-encoder network, the same matrix will be used for the inputs and the outputs. The middle layer (hidden layer 2) will be the "bottleneck" when these networks are used as auto-encoders. You should use the identity function as the activation function for this layer (that is, the output of a hidden unit in this layer should be the same as its input). You should use tanh as the activation function for the other two hidden layers. As in the demo functions on the webpage, the output layer should use the identity activation function. (Using tanh for the bottleneck layer would produce a reasonable result, but it seems better to me if values of bottleneck units are not limited to the range $(-1, +1)$.)

You should initialize the weights as in the demo functions on the webpage, except that you should use a standard deviation of 0.1 rather than 0.01. This change is necessary because when there are many layers, the gradient of the error when the weights are very small is tiny, and hence training is very slow at the start.

You should first apply auto-encoder networks to the first two datasets that you used for assignment 3. For convenience, I created new data files for this assignment with just the first 200 cases from these datasets (which are all you used for training in assignment 3). You should try reducing dimensionality from two to one for these datasets, using an auto-encoder network with just one

hidden unit in the bottleneck (hidden layer 2). You should use five hidden units in hidden layers 1 and 3. You will need to experiment to find a learning rate that is stable, and to see how many training iterations are needed. You should hand in a plot of the squared error over the training run (from the "E" element in the list returned by mlp_train), and a scatterplot that shows the training points (in black), the reconstructed training points (the auto-encoder outputs) in red, and lines connecting each training point to its reconstruction. (You can use some similar black-and-white scheme if you don't have a colour printer.)

You should then apply an auto-encoder network to a dataset of 600 images of handwritten digits (from US zip codes). I derived these images from the well-known MNIST dataset, by randomly selecting 600 out of the total 60000 training cases, and reducing the resolution of the images from 28×28 to 14×14 by averaging 2×2 blocks of pixel values. A data file with 600 lines each containing 196 pixel values (scaled to the interval $[0, 1]$) is provided on the course webpage. Another file containing the identity of each of these 600 digits (0 to 9) is also provided.

You should try to reduce the dimensionality of this data from 196 to two, using an auto-encoder network with two bottleneck units. You should use ten hidden units in hidden layers 1 and 3. You will need to find a suitable learning rate, and then train for a suitable number of iterations. You should hand in a plot of the squared error over the training run, and a scatterplot of the hidden unit values for each training case, with each point identified by its label (0-9). You can set plot characters by using a vector for the "pch" option for plot (converting the labels with as.character if you are storing them as numbers). You may also find it helpful to plot different digits in different colours. You can use the "col" option for this, using a vector of numbers from 1 to 10 to identify colours (colour 0 is invisible).

I have provided a function on the course webpage that displays an image of a digit, given the vector of 196 pixel values, which will be useful if you are curious what the digit images actually look like.

Note that for all datasets, it may be desirable to try more than one training run, with different random initializations of the weights.

You should hand in the plots mentioned above, a listing of your modified mlp functions and your scripts for running them and producing plots, and a discussion of the results (which may include other ouput or plots if they help support your conclusions). You should discuss how well the auto-encoder networks worked for the first two datasets in terms of how well they seem to have captured the one-dimensional structure that is present. For the digit dataset, you should discuss whether the reduction to two dimensions seems to have preserved the information needed for classifying an image as a digit from 0 to 9. You can discuss other things as well, such as how easy or hard it was to find good learning rates.

Addendum: Here in detail are the equations defining the function computed by the multilayer perceptron network with three hidden layers that you should use:

$$
\begin{aligned}
h_j^{(1)} &= \tanh\left(w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k\right), \quad \text{for } j = 1, \ldots, m_1 \\
h_j^{(2)} &= w_{0j}^{(2)} + \sum_{k=1}^{m_1} w_{kj}^{(2)} h_k^{(1)}, \quad \text{for } j = 1, \ldots, m_2 \\
h_j^{(3)} &= \tanh\left(w_{0j}^{(3)} + \sum_{k=1}^{m_2} w_{kj}^{(3)} h_k^{(2)}\right), \quad \text{for } j = 1, \ldots, m_3 \\
f_j &= w_{0j}^{(4)} + \sum_{k=1}^{m_3} w_{kj}^{(4)} h_k^{(3)}, \quad \text{for } j = 1, \ldots, q
\end{aligned}
$$