Vectorization

For small jobs like the housing prices data we used for linear regression, your code does not need to be extremely fast. However, if your implementation for Exercise 1A or 1B used a for-loop as suggested, it is probably too slow to work well for large problems that are more interesting. This is because looping over the examples (or any other elements) sequentially in MATLAB is slow. To avoid for-loops, we want to rewrite our code to make use of optimized vector and matrix operations so that MATLAB will execute it quickly. (This is also useful for other languages, including Python and C/C++ — we want to re-use optimized operations when possible.)

Following are some examples for how to vectorize various operations in MATLAB.

Example: Many matrix-vector products

Frequently we want to compute matrix-vector products for many vectors at once, such as when we compute $\theta^\top x^{(i)}$ for each example in a dataset (where θ may be a 2D matrix, or a vector itself). We can form a matrix X containing our entire dataset by concatenating the examples $x^{(i)}$ to form the columns of X:

$$X = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & | & | \end{bmatrix}$$

With this notation, we can compute $y^{(i)} = Wx^{(i)}$ for all $x^{(i)}$ at once as:

$$\begin{bmatrix} | & | & | & | \\ y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ | & | & | & | \end{bmatrix} = Y = WX$$

So, when performing linear regression, we can use $\theta^\top X$ to avoid looping over all of our examples to compute $y^{(i)} = \theta^\top X^{(i)}$.

Example: normalizing many vectors

Suppose we have many vectors $x^{(i)}$ concatenated into a matrix X as above, and we want to compute $y^{(i)} = x^{(i)}/||x^{(i)}||_2$ for all of the $x^{(i)}$. This may be done using several of MATLAB's array operations:

```
X_norm = sqrt( sum(X.^2,1) );
Y = bsxfun(@rdivide, X, X_norm);
```

This code squares all of the elements of X, then sums along the first dimension (the rows) of the result, and finally takes the square root of each element. This leaves us with a 1-by-m matrix containing $||x^{(i)}||_2$. The `bsxfun` routine can be thought of as expanding or cloning $X\text{norm}$ so that it has the same dimension as X before applying an element-wise binary function. In the example above it divides every element $X_{ji} = x_j^{(i)}$ by the corresponding column in $X\text{norm}$, leaving us with $Y_{ji} = X_{ji}/X\text{norm}_i = x_j^{(i)}/||x^{(i)}||_2$ as desired. `bsxfun` can be used with almost any binary element-wise function (e.g., @plus, @ge, or @eq). See the `bsxfun` docs!

Example: matrix multiplication in gradient computations

In our linear regression gradient computation, we have a summation of the form:

$$\frac{\partial J(\theta; X, y)}{\partial \theta_j} = \sum_i x_j^{(i)}(y^{(i)} - y^{(i)}).$$

Supervised Learning and Optimization
Linear Regression (http://ufldl.stanford.edu/tutoria
Logistic Regression (http://ufldl.stanford.edu/tutoria
Vectorization (http://ufldl.stanford.edu/tutoria
Debugging: Gradient Checking (http://ufldl.stanford.edu/tutoria
Softmax Regression (http://ufldl.stanford.edu/tutoria
Debugging: Bias and Variance (http://ufldl.stanford.edu/tutoria
Debugging: Optimizers and Objectives (http://ufldl.stanford.edu/tutoria
Supervised Neural Networks
Multi-Layer Neural Networks (http://ufldl.stanford.edu/tutoria
Exercise: Supervised Neural Network (http://ufldl.stanford.edu/tutoria
Supervised Convolutional Neural Network
Feature Extraction Using Convolution (http://ufldl.stanford.edu/tutoria
Pooling (http://ufldl.stanford.edu/tutoria
Exercise: Convolution and Pooling (http://ufldl.stanford.edu/tutoria
Optimization: Stochastic Gradient Descent (http://ufldl.stanford.edu/tutoria
Convolutional Neural Network

Whenever we have a summation over a single index (in this case i) with several other fixed indices (in this case j) we can often rephrase the computation as a matrix multiply since $[AB]_{jk} = \sum_i A_{ji} B_{ik}$. If y and $y\hat{}$ are column vectors (so $y_i \equiv y^{(i)}$), then with this template we can rewrite the above summation as:

$$\frac{\partial J(\theta; X, y)}{\partial \theta_j} = \sum_i X_{ji}(y_i\hat{} - y_i) = [X(y\hat{} - y)]_j.$$

Thus, to perform the entire computation for every j we can just compute $X(y\hat{} - y)$. In MATLAB:

```
% X(j,i) = j'th coordinate of i'th example.
% y(i) = i'th value to be predicted;  y is a column vector.
% theta = vector of parameters

y_hat = theta'*X; % so y_hat(i) = theta' * X(:,i).  Note that y_hat is a *row-vec
tor*.
g = X*(y_hat' - y);
```

Exercise 1A and 1B Redux

Go back to your Exercise 1A and 1B code. In the `ex1a_linreg.m` file and `ex1b_logreg.m` file you will find commented-out code that calls `minFunc` using `linear_regression_vec.m` and `logistic_regression_vec.m` (respectively) instead of `linear_regression.m` and `logistic_regression.m`. For this exercise, fill in the `linear_regression_vec.m` and `logistic_regression_vec.m` files with a vectorized implementation of your previous solutions. Uncomment the calling code in `ex1a_linreg.m` and `ex1b_logreg.m` and compare the running times of each implementation. Verify that you get similar results to your original solutions!

(http://ufldl.stanford.edu/tutoria
Excercise: Convolutional Neural Network (http://ufldl.stanford.edu/tutoria
Unsupervised Learning
Autoencoders (http://ufldl.stanford.edu/tutoria
PCA Whitening (http://ufldl.stanford.edu/tutoria
Exercise: PCA Whitening (http://ufldl.stanford.edu/tutoria
Sparse Coding (http://ufldl.stanford.edu/tutoria
ICA (http://ufldl.stanford.edu/tutoria
RICA (http://ufldl.stanford.edu/tutoria
Exercise: RICA (http://ufldl.stanford.edu/tutoria
Self-Taught Learning
Self-Taught Learning (http://ufldl.stanford.edu/tutoria
Exercise: Self-Taught Learning (http://ufldl.stanford.edu/tutoria