Logistic Regression Vectorization Example

From Ufldl

Consider training a logistic regression model using batch gradient ascent. Suppose our hypothesis is

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)},$$

where (following the notational convention from the OpenClassroom videos and from CS229) we let $x_0 = 1$, so that $x \in \Re^{n+1}$ and $\theta \in \Re^{n+1}$, and θ_0 is our intercept term. We have a training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ of m examples, and the batch gradient ascent update rule is $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$, where $\ell(\theta)$ is the log likelihood and $\nabla_\theta \ell(\theta)$ is its derivative.

[Note: Most of the notation below follows that defined in the OpenClassroom videos or in the class CS229: Machine Learning. For details, see either the OpenClassroom videos (http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning) or Lecture Notes #1 of http://cs229.stanford.edu/ .]

We thus need to compute the gradient:

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}.$$

Suppose that the Matlab/Octave variable `x` is a matrix containing the training inputs, so that `x(:,i)` is the i-th training example $x^{(i)}$, and `x(j,i)` is $x_j^{(i)}$. Further, suppose the Matlab/Octave variable `y` is a *row* vector of the labels in the training set, so that the variable `y(i)` is $y^{(i)} \in \{0, 1\}$. (Here we differ from the OpenClassroom/CS229 notation. Specifically, in the matrix-valued `x` we stack the training inputs in columns rather than in rows; and `y`$\in \Re^{1\times m}$ is a row vector rather than a column vector.)

Here's truly horrible, extremely slow, implementation of the gradient computation:

```
% Implementation 1
grad = zeros(n+1,1);
for i=1:m,
  h = sigmoid(theta'*x(:,i));
  temp = y(i) - h;
  for j=1:n+1,
    grad(j) = grad(j) + temp * x(j,i);
  end;
end;
```

The two nested for-loops makes this very slow. Here's a more typical implementation, that partially vectorizes the algorithm and gets better performance:

```
% Implementation 2
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + (y(i) - sigmoid(theta'*x(:,i)))* x(:,i);
end;
```

However, it turns out to be possible to even further vectorize this. If we can get rid of the for-loop, we can significantly speed up the implementation. In particular, suppose `b` is a column vector, and `A` is a matrix. Consider the following ways of computing `A * b`:

```
% Slow implementation of matrix-vector multiply
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + b(i) * A(:,i);  % more commonly written A(:,i)*b(i)
end;

% Fast implementation of matrix-vector multiply
grad = A*b;
```

We recognize that Implementation 2 of our gradient descent calculation above is using the slow version with a for-loop, with `b(i)` playing the role of `(y(i) - sigmoid(theta'*x(:,i)))`, and `A` playing the role of `x`. We can derive a fast implementation as follows:

```
% Implementation 3
grad = x * (y- sigmoid(theta'*x))';
```

Here, we assume that the Matlab/Octave `sigmoid(z)` takes as input a vector `z`, applies the sigmoid function component-wise to the input, and returns the result. The output of `sigmoid(z)` is therefore itself also a vector, of the same dimension as the input `z`

When the training set is large, this final implementation takes the greatest advantage of Matlab/Octave's highly optimized numerical linear algebra libraries to carry out the matrix-vector operations, and so this is far more efficient than the earlier implementations.

Coming up with vectorized implementations isn't always easy, and sometimes requires careful thought. But as you gain familiarity with vectorized operations, you'll find that there are design patterns (i.e., a small number of ways of vectorizing) that apply to many different pieces of code.

Vectorization | **Logistic Regression Vectorization Example** | Neural Network Vectorization | Exercise:Vectorization

Language : 中文