Exercise: Convolution and Pooling

Convolution and Pooling

In this exercise you will and test convolution (http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution) and pooling (http://ufldl.stanford.edu/tutorial/supervised/Pooling) functions. We have provided some starter code (https://github.com/amaas/stanford_dl_ex/tree/master/cnn). You should write your code at the places indicated "`YOUR CODE HERE`" in the files. For this exercise, you will need to modify `cnnConvolve.m` and `cnnPool.m`.

Dependencies

The following additional files are required for this exercise:

MNIST helper functions (https://github.com/amaas/stanford_dl_ex/tree/master/common)

Starter Code (https://github.com/amaas/stanford_dl_ex/tree/master/cnn)

Step 1: Implement and test convolution

In this step, you will implement the convolution function, and test it on a small part of the data set to ensure that you have implemented it correctly.

Step 1a: Implement convolution

Implement convolution, as described in ((Feature Extraction Using Convolution)), in the function `cnnConvolve` in `cnnConvolve.m`. Implementing convolution is somewhat involved, so we will guide you through the process below.

First, we want to compute $\sigma(Wx_{(r,c)} + b)$ for all valid (r, c) (valid meaning that the entire 8x8 patch is contained within the image; this is as opposed to a full convolution, which allows the patch to extend outside the image, with the area outside the image assumed to be 0), where W and b are the learned weights and biases from the input layer to the hidden layer, and $x_{(r,c)}$ is the 8x8 patch with the upper left corner at (r, c). To accomplish this, one naive method is to loop over all such patches and compute $\sigma(Wx_{(r,c)} + b)$ for each of them; while this is fine in theory, it can very slow. Hence, we usually use MATLAB's built in convolution functions, which are well optimized.

Observe that the convolution above can be broken down into the following three small steps. First, compute $Wx_{(r,c)}$ for all (r, c). Next, add b to all the computed values. Finally, apply the sigmoid function to the resulting values. This doesn't seem to buy you anything, since the first step still requires a loop. However, you can replace the loop in the first step with one of MATLAB's optimized convolution functions, `conv2`, speeding up the process significantly.

However, there are two important points to note in using `conv2`. First, `conv2` performs a 2-D convolution, but you have 4 "dimensions" - image number, filter (or feature) number, row of image and column of image - that you want to convolve over. Because of this, you will have to convolve each filter separately for each image, using the row and column of the image as the 2 dimensions you convolve over. This means that you will need two outer loops over the image number `imageNum` and filter number `filterNum`. Inside the two nested for-loops, you will perform a `conv2` 2-D convolution, using the weight matrix for the `filterNum`-th filter and the image matrix for the `imageNum`-th image.

Second, because of the mathematical definition of convolution, the filter matrix must be "flipped" before passing it to `conv2`. The following implementation tip explains the "flipping" of feature matrices when using MATLAB's convolution functions:

Implementation tip: Using `conv2` and `convn` Because the mathematical definition of convolution involves "flipping" the matrix to convolve with (reversing its rows and its columns), to use MATLAB's convolution functions, you must first "flip" the weight matrix so that when MATLAB "flips" it according to the mathematical definition the entries will be at the correct place. For example, suppose you wanted to convolve two matrices `image` (a large image) and `W` (the feature) using `conv2(image, W)`, and W is a 3x3 matrix as below: [

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

] If you use `conv2(image, W)`, MATLAB will first "flip" `W`, reversing its rows and columns, before convolving `W` with `image`, as below: [

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \xrightarrow{flip} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

] If the original layout of `W` was correct, after flipping, it would be incorrect. For the layout to be correct after flipping, you will have to flip `W` before passing it into `conv2`, so that after MATLAB flips `W` in `conv2`, the layout will be correct. For `conv2`, this means reversing the rows and columns, which can be done by rotating `W` 90 degrees twice with `rot90` as shown below:

```
% Flip W for use in conv2
W = rot90(W,2);
```

Next, to each of the `convolvedFeatures`, you should then add b, the corresponding bias for the `filterNum`-th filter.

Step 1b: Check your convolution

Step 1b: Check your convolution

We have provided some code for you to check that you have done the convolution correctly. The code randomly checks the convolved values for a number of (feature, row, column) tuples by computing the feature activations using randomly generated features and images from the MNIST dataset.

Step 2: Implement and test pooling

Step 2a: Implement pooling

Implement pooling in the function `cnnPool` in `cnnPool.m`. You should implement mean pooling (i.e., averaging over feature responses) for this part. This can be done efficiently using the `conv2` function as well. The inputs are the responses of each image with each filter computed in the previous step. Convolve each of these with a matrix of ones followed by a subsampling and averaging. Make sure to use the “valid” border handling convolution.

Step 2b: Check your pooling

We have provided some code for you to check that you have done the pooling correctly. The code runs `cnnPool` against a test matrix to see if it produces the expected result.

Self-Taught Learning

Self-Taught Learning (http://ufldl.stanford.edu/tutorial/selftaughtlearning/SelfTaughtLearning)

Exercise: Self-Taught Learning (http://ufldl.stanford.edu/tutorial/selftaughtlearning/ExerciseSelfTaughtLearning)