Exercise: Convolutional Neural Network

Overview

In this exercise you will implement a convolutional neural network for digit classification. The architecture of the network will be a convolution and subsampling layer followed by a densely connected output layer which will feed into the softmax regression and cross entropy objective. You will use mean pooling for the subsampling layer. You will use the back-propagation algorithm to calculate the gradient with respect to the parameters of the model. Finally you will train the parameters of the network with stochastic gradient descent and momentum.

We have provided some MATLAB starter code (https://github.com/amaas/stanford_dl_ex/tree/master/cnn). You should write your code at the places indicated in the files "`YOUR CODE HERE`". You have to complete the following files: `cnnCost.m`, `minFuncSGD.m`. The starter code in `cnnTrain.m` shows how these functions are used.

Dependencies

Convolutional Network starter code (https://github.com/amaas/stanford_dl_ex/tree/master/cnn)

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

We strongly suggest that you complete the convolution and pooling (http://ufldl.stanford.edu/tutorial/supervised/ExerciseConvolutionAndPooling), multilayer supervised neural network (http://ufldl.stanford.edu/tutorial/supervised/ExerciseSupervisedNeuralNetwork) and softmax regression (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression) exercises prior to starting this one.

Step 0: Initialize Parameters and Load Data

In this step we initialize the parameters of the convolutional neural network. You will be using 10 filters of dimension 9x9, and a non-overlapping, contiguous 2x2 pooling region.

We also load the MNIST training data here as well.

Step 1: Implement CNN Objective

Implement the CNN cost and gradient computation in this step. Your network will have two layers. The first layer is a convolutional layer followed by mean pooling and the second layer is a densely connected layer into softmax regression. The cost of the network will be the standard cross entropy between the predicted probability distribution over 10 digit classes for each image and the ground truth distribution.

Step 1a: Forward Propagation

Convolve every image with every filter, then mean pool the responses. This should be similar to the implementation from the convolution and pooling (http://ufldl.stanford.edu/tutorial/supervised/ExerciseConvolutionAndPooling) exercise using MATLAB's `conv2` function. You will need to store the activations after the convolution but before the pooling for efficient back propagation later.

Following the convolutional layer, we unroll the subsampled filter responses into a 2D matrix with each column representing an image. Using the `activationsPooled` matrix, implement a standard softmax layer following the style of the softmax regression (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression) exercise.

Step 1b: Calculate Cost

Generate the ground truth distribution using MATLAB's `sparse` function from the `labels` given for each image. Using the ground truth distribution, calculate the cross entropy cost between that and the predicted distribution.

Note at the end of this section we have also provided code to return early after computing predictions from the probability vectors computed above. This will be useful at test time when we wish make predictions on each image without doing a full back propagation of the network which can be rather costly.

Step 1c: Back Propagation

First compute the error, δ_d, from the cross entropy cost function w.r.t. the parameters in the densely connected layer. You will then need to propagate this error through the subsampling and convolutional layer. Use MATLAB's `kron` function to upsample the error and propagate through the pooling layer.

Implementation tip: Using `kron` You can upsample the error from an incoming layer to propagate through a mean-pooling layer quickly using MATLAB's `kron` function. This function takes the Kroneckor Tensor Product of two matrices. For example, suppose the pooling region was 2x2 on a 4x4 image. This means that the incoming error to the pooling layer will be of dimension 2x2 (assuming non-overlapping and contiguous pooling regions). The error must be upsampled from 2x2 to be 4x4. Since mean pooling is used, each error value contributes equally to the values in the region from which it came in the original 4x4 image. Let the incoming error to the pooling layer be given by

$$delta = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

If you use `kron(delta, ones(2,2))`, MATLAB will take the element by element product of each element in `ones(2,2)` with `delta`, as below:

$$\begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} \rightarrow \text{kron}\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\right)$$

After the error has been upsampled, all that's left to be done to propagate through the pooling layer is to divide by the size of the pooling region. A basic implementation is shown below,

```
% Upsample the incoming error using kron
delta_pool = (1/poolDim^2) * kron(delta,ones(poolDim));
```

To propagate error through the convolutional layer, you simply need to multiply the incoming error by the derivative of the activation function as in the usual back propagation algorithm. Using these errors to compute the gradient w.r.t to each weight is a bit trickier since we have tied weights and thus many errors contribute to the gradient w.r.t. a single weight. We will discuss this in the next section.

Step 1d: Gradient Calculation

Compute the gradient for the densely connected weights and bias, `W_d` and `b_d` following the equations presented in multilayer neural networks (http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks).

In order to compute the gradient with respect to each of the filters for a single training example (i.e. image) in the convolutional layer, you must first convolve the error term for that image-filter pair as computed in the previous step with the original training image. Again, use MATLAB's `conv2` function with the 'valid' option to handle borders correctly. Make sure to flip the error matrix for that image-filter pair prior to the convolution as discussed in the simple convolution exercise (http://ufldl.stanford.edu/tutorial/supervised/ExerciseConvolutionAndPooling). The final gradient for a given filter is the sum over the convolution of all images with the error for that image-filter pair.

The gradient w.r.t to the bias term for each filter in the convolutional layer is simply the sum of all error terms corresponding to the given filter.

Make sure to scale your gradients by the inverse size of the training set if you included this scale in the cost calculation otherwise your code will not pass the numerical gradient check.

Step 2: Gradient Check

Use the `computeNumericalGradient` function to check the cost and gradient of your convolutional network. We've provided a small sample set and toy network to run the numerical gradient check on.

Once your code passes the gradient check you're ready to move onto training a real network on the full dataset. Make sure to switch the `DEBUG` boolean to `false` in order not to run the gradient check again.

Step 3: Learn Parameters

Using a batch method such as L-BFGS to train a convolutional network of this size even on MNIST, a relatively small dataset, can be computationally slow. A single iteration of calculating the cost and gradient for the full training set can take several minutes or more. Thus you will use stochastic gradient descent (SGD) to learn the parameters of the network.

You will use SGD with momentum as described in Stochastic Gradient Descent (http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent). Implement the velocity vector and parameter vector update in `minFuncSGD.m`.

In this implementation of SGD we use a relatively heuristic method of annealing the learning rate for better convergence as learning slows down. We simply halve the learning rate after each epoch. As mentioned in Stochastic Gradient Descent (http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent), we also randomly shuffle the data before each epoch, which tends to provide better convergence.

Step 4: Test

With the convolutional network and SGD optimizer in hand, you are now ready to test the performance of the model. We've provided code at the end of `cnnTrain.m` to test the accuracy of your networks predictions on the MNIST test set.

Run the full function `cnnTrain.m` which will learn the parameters of you convolutional neural network over 3 epochs of the data. This shouldn't take more than 20 minutes. After 3 epochs, your networks accuracy on the MNIST test set should be above 96%.

Congratulations, you've successfully implemented a Convolutional Neural Network!

Self-Taught Learning

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

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