Convolutional Neural Networks

Monday, February 24, 2014

In the previous post (http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks/), we figured out how to do forward and backward propagation to compute the gradient for fully-connected neural networks, and used those algorithms to derive the Hessian-vector product algorithm for a fully connected neural network.

Next, let's figure out how to do the exact same thing for convolutional neural networks. While the mathematical theory should be *exactly* the same, the actual derivation will be slightly more complex due to the architecture of convolutional neural networks.

Convolutional Neural Networks

First, let's go over out convolutional neural network architecture. There are several variations on this architecture; the choices we make are fairly arbitrary. However, the algorithms will be very similar for all variations, and their derivations will look very similar.

A convolutional neural network consists of several layers. These layers can be of three types:

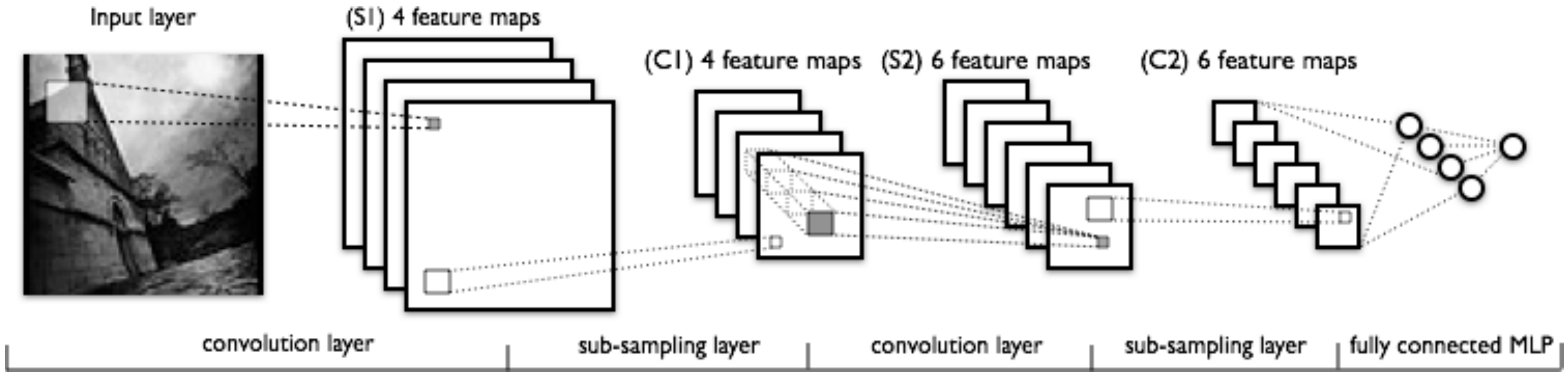
- **Convolutional:** Convolutional layers consist of a rectangular grid of neurons. It requires that the previous layer *also* be a rectangular grid of neurons. Each neuron takes inputs from a rectangular section of the previous layer; the weights for this rectangular section are the same for each neuron in the convolutional layer. Thus, the convolutional layer is just an image *convolution* of the previous layer, where the weights specify the convolution filter.

  In addition, there may be several grids in each convolutional layer; each grid takes inputs from *all* the grids in the previous layer, using potentially different filters.

- **Max-Pooling:** After each convolutional layer, there may be a pooling layer. The pooling layer takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block. There are several ways to do this pooling, such as taking the average or the maximum, or a learned linear combination of the neurons in the block. Our pooling layers will always be max-pooling layers; that is, they take the maximum of the block they are pooling.

- **Fully-Connected:** Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. A fully connected layer takes all neurons in the previous layer (be it fully connected, pooling, or convolutional) and connects it to every single neuron it has. Fully connected layers are not spatially located anymore (you can visualize them as one-dimensional), so there can be no convolutional layers after a fully connected layer.

The resulting neural network will look like this (LeNet):



Note that we are not really constrained to two-dimensional convolutional neural networks. We can in the exact same way build one- or three- dimensional convolutional neural networks; our filters will just become appropriately dimensioned, and our pooling layers will change dimension as well. We may, for instance, want to use one-dimensional convolutional nets on audio or three-dimensional nets on MRI data.

Now that we've described the structure of our neural network, let's work through forward and backward propagation to do prediction and gradient computations in these neural networks.

Forward Propagation

Our neural networks now have three types of layers, as defined above. The forward and backward propagations will differ depending on what layer we're propagating through. We've already talked about fully connected networks in the previous post (http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks/), so we'll just look at the convolutional layers and the max-pooling layers.

Convolutional Layers

Suppose that we have some $N \times N$ square neuron layer which is followed by our convolutional layer. If we use an $m \times m$ filter ω, our convolutional layer output will be of size $(N - m + 1) \times (N - m + 1)$. In order to compute the pre-nonlinearity input to some unit x_{ij}^{ℓ} in our layer, we need to sum up the contributions (weighted by the filter components) from the previous layer cells:

$$x_{ij}^{\ell} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \omega_{ab} y_{(i+a)(j+b)}^{\ell-1}.$$

This is just a convolution, which we can express in Matlab via

```
conv2(x, w, 'valid')
```

Then, the convolutional layer applies its nonlinearity:

$$y_{ij}^{\ell} = \sigma(x_{ij}^{\ell}).$$

Max-Pooling Layers

The max-pooling layers are quite simple, and do no learning themselves. They simply take some $k \times k$ region and output a single value, which is the maximum in that region. For instance, if their input layer is a $N \times N$ layer, they will then output a $\frac{N}{k} \times \frac{N}{k}$ layer, as each $k \times k$ block is reduced to just a single value via the max function.

Backward Propagation

Next, let's derive the backward propagation algorithms for these two layer types.

Convolutional Layers

Let's assume that we have some error function, E, and we know the error values at our convolutional layer. What, then, are the error values at the layer before it, and what is the gradient for each weight in the convolutional layer?

Note that the error we know and that we need to compute for the previous layer is the partial of E with respect to each neuron output $\left(\frac{\partial E}{\partial y_{ij}^{\ell}}\right)$. Let's first figure out what the gradient component is for each weight by applying the chain rule. Note that in the chain rule, we must sum the contributions of *all* expressions in which the variable occurs.

$$\frac{\partial E}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^{\ell}} \frac{\partial x_{ij}^{\ell}}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^{\ell}} y_{(i+a)(j+b)}^{\ell-1}$$

In this case, we must sum over all x_{ij}^{ℓ} expressions in which ω_{ab} occurs. (This corresponds to weight-sharing in the neural network!) Note that we know that $\frac{\partial x_{ij}^{\ell}}{\partial \omega_{ab}} = y_{(i+a)(j+b)}^{\ell-1}$, just by looking at the forward propagation equations.

In order to compute the gradient, we need to know the values $\frac{\partial E}{\partial x_{ij}^\ell}$ (which are often called "deltas"). The deltas are fairly straightforward to compute, once more using the chain rule:

$$\frac{\partial E}{\partial x_{ij}^\ell} = \frac{\partial E}{\partial y_{ij}^\ell}\frac{\partial y_{ij}^\ell}{\partial x_{ij}^\ell} = \frac{\partial E}{\partial y_{ij}^\ell}\frac{\partial}{\partial x_{ij}^\ell}\left(\sigma(x_{ij}^\ell)\right) = \frac{\partial E}{\partial y_{ij}^\ell}\sigma'(x_{ij}^\ell)$$

As we can see, since we already know the error at the current layer $\frac{\partial E}{\partial y_{ij}^\ell}$, we can very easily compute the deltas $\frac{\partial E}{\partial x_{ij}^\ell}$ at the current layer by just using the derivative of the activation function, $\sigma'(x)$. Since we know the errors at the current layer, we now have everything we need to compute the gradient with respect to the weights used by this convolutional layer.

In addition to compute the weights for this convolutional layer, we need to propagate errors back to the previous layer. We can once more use the chain rule:

$$\frac{\partial E}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^\ell}\frac{\partial x_{(i-a)(j-b)}^\ell}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^\ell}\omega_{ab}$$

Looking back at the forward propagation equations, we can tell that $\frac{\partial x_{(i-a)(j-b)}^\ell}{\partial y_{ij}^{\ell-1}} = \omega_{ab}$. This gives us the above value for the error at the previous layer. As we can see, that looks *slightly* like a convolution! We have our filter ω being applied somehow to the layer; however, instead of having $x_{(i+a)(j+b)}$ we have $x_{(i-a)(j-b)}$. In addition, note that the expression above only makes sense for points that are at least m away from the top and left edges. In order to fix this, we must pad the top and left edges with zeros. If we do that, then this is simply a convolution using ω which has been flipped along both axes!

Max-Pooling Layers

As noted earlier, the max-pooling layers do not actually do any learning themselves. Instead, then reduce the size of the problem by introducing sparseness. In forward propagation, $k \times k$ blocks are reduced to a single value. Then, this single value acquires an error computed from backwards propagation from the previous layer. This error is then just forwarded to the place where it came from. Since it only came from one place in the $k \times k$ block, the backpropagated errors from max-pooling layers are rather sparse.

Conclusion

Convolutional neural networks are an architecturally different way of processing dimensioned and ordered data. Instead of assuming that the location of the data in the input is irrelevant (as fully connected layers do), convolutional and max pooling layers enforce weight sharing translationally. This models the way the human visual cortex works, and has been shown to work incredibly well for object recognition (https://www.cs.toronto.edu/~hinton/absps/imagenet.pdf) and a number of other tasks. We can learn convolutional networks through traditional stochastic gradient descent; in addition, we could apply the $\mathcal{R}_v\{\cdot\}$ operator from the previous post (http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks/) to convolutional network gradient computation to get a Hessian-vector product algorithm, which would enable use to use Hessian-free optimization instead.

Monday, February 24, 2014 - Posted in machine-learning (http://andrew.gibiansky.com/blog/categories/machine-learning)

« Fully Connected Neural Network Algorithms (http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks)

Gauss Newton Matrix » (http://andrew.gibiansky.com/blog/machine-learning/gauss-newton-matrix)

Contact

If you've got questions, comments, suggestions, or just want to talk, feel free to email me at andrew.gibiansky on Gmail.

Recent Posts (RSS)

- NRAM: Theano Implementation (http://andrew.gibiansky.com/blog/machine-learning/nram-2)
- NRAM: Neural Random Access Memory (http://andrew.gibiansky.com/blog/machine-learning/nram-1)
- jq Primer: Munging JSON Data (http://andrew.gibiansky.com/blog/command-line/jq-primer)
- Creating a Culture of Good Engineering (http://andrew.gibiansky.com/blog/thoughts/engineering-practices)
- Common Techniques in Molecular Biology (http://andrew.gibiansky.com/blog/genetics/technique-primers)
- CRISPR Gene Editing (http://andrew.gibiansky.com/blog/genetics/crispr)
- Quick Coding Intro to Neural Networks (http://andrew.gibiansky.com/blog/machine-learning/coding-intro-to-nns)
- Writing a SAT Solver (http://andrew.gibiansky.com/blog/verification/writing-a-sat-solver)
- Lattice Boltzmann Method (http://andrew.gibiansky.com/blog/physics/lattice-boltzmann-method)
- Finger Trees (http://andrew.gibiansky.com/blog/haskell/finger-trees)
- Abstraction in Haskell (Monoids, Functors, Monads) (http://andrew.gibiansky.com/blog/haskell/haskell-abstractions)
- Typeclasses: Polymorphism in Haskell (http://andrew.gibiansky.com/blog/haskell/haskell-typeclasses)
- Your First Haskell Application (with Gloss) (http://andrew.gibiansky.com/blog/haskell/haskell-gloss)
- Intro to Haskell Syntax (http://andrew.gibiansky.com/blog/haskell/haskell-syntax)
- Linguistics and Syntax (http://andrew.gibiansky.com/blog/linguistics/why-syntax)
- Speech Recognition with Neural Networks

(http://andrew.gibiansky.com/blog/machine-learning/speech-recognition-neural-networks)

- Matrix Multiplication (http://andrew.gibiansky.com/blog/mathematics/matrix-multiplication)
- Recurrent Neural Networks (http://andrew.gibiansky.com/blog/machine-learning/recurrent-neural-networks)
- Gauss Newton Matrix (http://andrew.gibiansky.com/blog/machine-learning/gauss-newton-matrix)
- Convolutional Neural Networks (http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks)
- Fully Connected Neural Network Algorithms (http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks)
- Hessian Free Optimization (http://andrew.gibiansky.com/blog/machine-learning/hessian-free-optimization)
- Conjugate Gradient (http://andrew.gibiansky.com/blog/machine-learning/conjugate-gradient)
- Gradient Descent Typeclasses in Haskell (http://andrew.gibiansky.com/blog/machine-learning/gradient-descent)
- Homophony Groups in Haskell (http://andrew.gibiansky.com/blog/linguistics/homophony-groups)
- Creating Language Kernels for IPython (http://andrew.gibiansky.com/blog/ipython/ipython-kernels)
- Detecting Genetic Copynumber with Gaussian Mixture Models (http://andrew.gibiansky.com/blog/machine-learning/qpcr-blog-post)
- K Nearest Neighbors: Simplest Machine Learning (http://andrew.gibiansky.com/blog/machine-learning/k-nearest-neighbors-simplest-machine-learning)
- Cool Linear Algebra: Singular Value Decomposition (http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition)
- Accelerating Options Pricing via Fourier Transforms (http://andrew.gibiansky.com/blog/economics/accelerating-options-pricing-via-fourier-transforms)
- Pricing Stock Options via the Binomial Model (http://andrew.gibiansky.com/blog/economics/binomial-options-pricing-model)
- Your Very First Microprocessor (http://andrew.gibiansky.com/blog/electrical-engineering/your-very-first-microprocessor)
- Circuits and Arithmetic (http://andrew.gibiansky.com/blog/electrical-engineering/circuits-and-arithmetic)
- Digital Design Tools: Verilog and HDLs (http://andrew.gibiansky.com/blog/electrical-engineering/digital-design-tools-verilog-and-hdls)
- Quadcopter Dynamics and Simulation (http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics)
- The Digital State (http://andrew.gibiansky.com/blog/electrical-engineering/the-digital-state)
- Computing with Transistors (http://andrew.gibiansky.com/blog/electrical-engineering/computing-with-transistors)
- Machine Learning: Neural Networks (http://andrew.gibiansky.com/blog/machine-learning/machine-learning-neural-networks)
- Machine Learning: the Basics (http://andrew.gibiansky.com/blog/machine-learning/machine-learning-the-basics)
- Iranian Political Embargoes, and their Non-Existent Impact on Gasoline Prices (http://andrew.gibiansky.com/blog/economics/iranian-political-embargoes-and-their-non-existent-impact-on-gasoline-prices)
- Computational Fluid Dynamics (http://andrew.gibiansky.com/blog/physics/computational-fluid-dynamics)
- Fluid Dynamics: The Navier-Stokes Equations

(http://andrew.gibiansky.com/blog/physics/fluid-dynamics-the-navier-stokes-equations)

- Image Morphing (http://andrew.gibiansky.com/blog/image-processing/image-morphing)