Machine Learning Course - CS-433

Neural Nets – Convolutional Nets

Dec 13, 2016

Basic Structure of Convolutional Nets

Recall that for the standard neural network structure we introduced, every node at level l is connected to every node at level $l-1$. The advantage of this structure is that it is very general and powerful. The disadvantage is that such a network has a large number of parameters and so generically lots of data is needed to train it.

In some scenarios it is intuitive that "local" processing of data should suffice. E.g., consider an audio stream given by samples $x^{(0)}[n]$. It is natural to process such a stream by running it through a linear time-invariant *filter*, whose output, call it $x^{(1)}[n]$, is given by the *convolution* of the input $x^{(0)}[n]$ and the filter $f[n]$,

$$x^{(1)}[n] = \sum_k f[k]x^{(0)}[n-k].$$

The filter $f[n]$ is often "local", i.e., $f[k] = 0$ for $|k| \geq K$. Much of signal processing is based on this type of operation. By choosing an appropriate type of filter we can bring out various aspects of the underlying signal. E.g., we can *smooth features* by averaging or we can *enhance differences* between neighboring elements by taking a so-called "high-pass" filter.

We have essentially the same scenario if we think of a picture. Now the signal $x^{(0)}[n,m]$ is two-dimensional and the convolution takes the form

$$x^{(1)}[n,m] = \sum_{k,l} f[k,l]x^{(0)}[n-k,m-l].$$

As before, we can take filters $f[n, m]$ that are "local" i.e., only have non-zero coefficients for small values of $|n|$ and $|m|$.

There are two important aspects about this structure. First, the output $x^{(1)}$ at position $[n, m]$ only depends on the value of the input $x^{(0)}$ at positions close to $[n, m]$. If we use this structure in a nn then we no longer get a fully connected network but the structure is much more sparse and local. This implies that we have significantly fewer parameters to deal with.

Second, this structure implies that we should use the *same* filter (e.g., not only the same connection-pattern but also the same weights) at every position! This is called *weight sharing*. Weight sharing drastically reduces the number of parameters further.

Figure 1 shows two layers of a very small nn where we see the difference between a fully connected network and one where connections are sparse and only local.

Why is it meaningful to use the same filter at every position in the network? Consider e.g. a photo. Photos are inherently "shift invariant." We can imagine that there is an essentially infinite-sized 2D photo in reality and we are seeing a small portion of it. The portion we are seeing is more or less random and so the exact position of any object in this photo is more or less random as well. It therefore makes sense that we treat each position essentially equally.

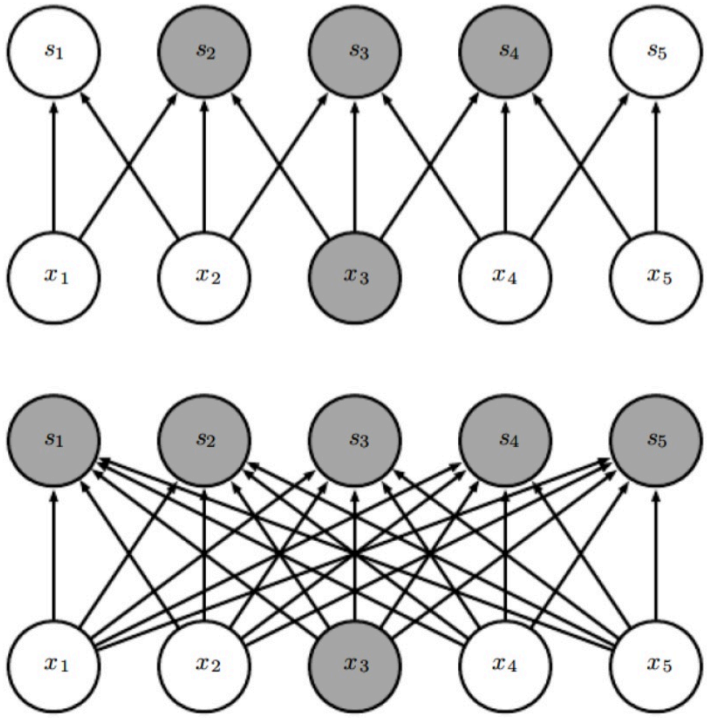

Figure 1: Fully connected versus sparse connections. In the sparse case a node on the bottom layer only influences a limited number of nodes on the top layer. (Figure 9.2 from http://www.deeplearningbook.org/)

Layout

It is common to lay out the data in a convolutional network according to its "natural" layout. E.g., if the input is a photo then it is natural to use a 2D layout, whereas if the data is naturally one-dimensional then it makes sense to use a 1D layout.

Handling of Borders

Consider a 2D case. Assume that we have an input of dimension $n_1 \times n_2$ and a $k_1 \times k_2$ kernel. There are several natural ways of dealing with the boundary.

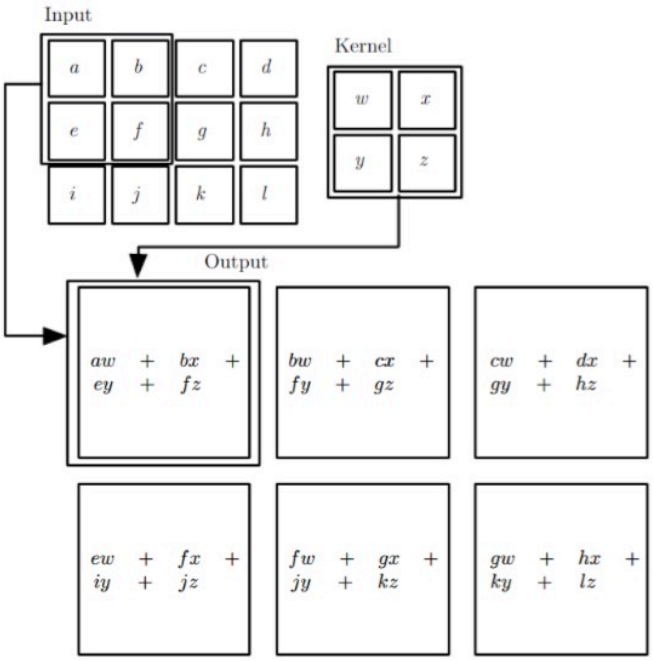

Figure 2: Handling the boundary using the *valid packing* method. (Figure 9.1 from http://www.deeplearningbook.org/)

The first is to *padd* the original input data with zeros at the boundary. More precisely, instead of considering the input of size $n_1 \times n_2$, consider the input to be of size $(n_1 + 2(k_1 - 1)) \times (n_2 + 2(k_2 - 1))$ where the "center piece" is the original data and there is an additional boundary of width $k_1 - 1$ and $k_2 - 1$ respectively which is set to 0. If we now perform the convolution then the output will again be of size $n_1 \times n_2$. For obvious reasons this is called *zero padding*.
The second option is to compute an output which is only of size $(n_1 - k_1 + 1) \times (n_2 - k_2 + 1)$, i.e., to perform the convolution only for positions so that the whole filter lies inside the original data. This is called *valid padding*. Figure 2 shows this second method.

Multiple Channels

Assume we start with picture, i.e., a 2D structure. It is common to not only compute the output of a single filter but to use multiple filters. The various outputs are called *channels*. This introduces some additional parameters into the model.
If we add several channels we do not end up with a 2D output in the next level but in fact with a 3D output. We proceed in the same fashion in each further stage, computing several channels per input layer in the previous stage. In this manner, we will get more and more channels as we get deeper and deeper into the network. On the other hand the "size" of each picture typically gets smaller and smaller as we proceed through the layers, either due to the handling of the boundary or because we might perform subsampling. The whole structure then looks a little bit like a pyramid. It gets thinner towards the top but the sections become longer and longer (more and more channels). This is shown in Figure 3.

Training

As we discussed, there are two aspects that make CNN special. First, only some of the edges are present. This means that the weight matrices are sparse and this does not require any change in the learning steps when we use SGD and backpropagation.
Second, weight sharing is used, i.e., many edges use weights that are the same. As we already mentioned in the previous

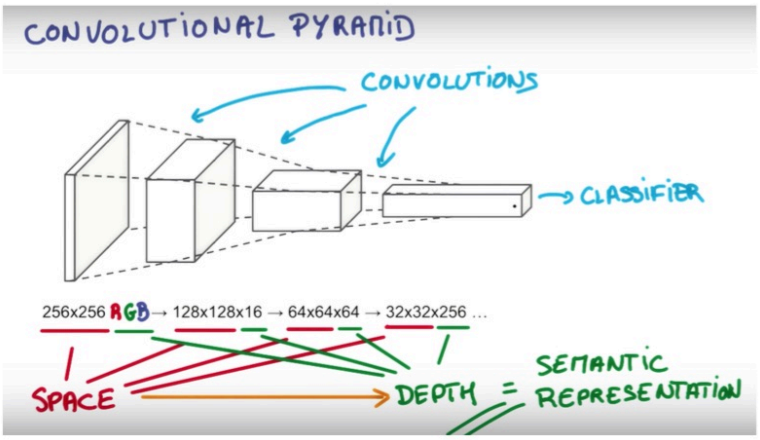

Figure 3: Structure of a typical CNN. Per layer we have increasingly more channels but a smaller "footprint." (from www.udacity.com/course/deep-learning–ud730)

lecture, with a small modificaiton the back propagation algorithm can still be used to train a cnn with weight sharing: run backpropagation ignoring that some weights are shared, considering each weight on each edge to be an independent variable. Once the gradient has been computed for this network with independent weights, just sum up the gradients of all edges that share the same weight. This gives us the gradient for the network with weight sharing.
To see this, consider a simple example. Let $f(x, y, z)$ be a function from $\mathbb{R}^3 \to \mathbb{R}$. Let $g(x, y) = f(x, y, x)$. In words, z is no longer an independent variable but we set $z = x$. If we now want to compute the gradient

$$(\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y})$$

then we can compute this by first computing

$$(\frac{\partial f(x,y,z)}{\partial x}, \frac{\partial f(x,y,z)}{\partial y}, \frac{\partial f(x,y,z)}{\partial z})$$

and then realizing that

$$(\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}) = (\frac{\partial f(x,y,z)}{\partial x} + \frac{\partial f(x,y,z)}{\partial z}, \frac{\partial f(x,y,z)}{\partial y}).$$

FFT

The convolutional structure can be used in order to compute the output of a CNN very efficiently. Wether this is more efficient than computing the output by directly implementing the sum depends on the size of the filter/kernel.
To keep things simple, assume that our data layout is one-dimensional and that each layer is connected via a convolution where we use zero-padding at the boundaries. Let us consider the first layer to be specific. We then have

$$x^{(1)}[n] = \sum_k f[k] x^{(0)}[n-k].$$

Assume that $|f[k]| > 0$ for $k = 0, \cdots K-1$. I.e., the filter has K non-zero coeffients. And assume that the signal $x^{(0)}[n]$ has length N. Zero-pad both the signal and the filter to make them of length L, where $L \geq N + K - 1$. More precisely, for the filter the first K positions are the non-zero positions of the original filter and the remaining $L - K$ are zeros. For the signal the first N positions are the non-zero coefficients of the signal and the remaining L_N are zeros.
Call the resulting quantities $\tilde{f}$ and $\tilde{x}^{(0)}$ respectively. Compute the *cyclic* convolution of these two signals, i.e., compute

$$\tilde{x}^{(1)}[n] = \sum_{k=0}^{L} f[k] x^{(0)}[n-k \bmod L].$$

This convolution is just like the regular one except that we compute indices modulo L. A quick check shows that $x^{(1)}[n]$ (where we used zero-padding to deal with the boundary) and $\tilde{x}^{(1)}[n]$ are identical for the first $N - K + 1$ positions.
But this cyclic convolution can be computed by transforming both the signal as well as the filter into the Fourier domain, multiplying the two, and then converting the result back. This in turn can be accomplished efficiently by means of the Fast Fourier Transform (FFT) in $cL \log_2(L)$ operations, where c is a small constant.
More precisely, for a signal $x[n]$ of length L its discrete Fourier Transform (DFT) (also of length L) and its inverse are given by

$$\hat{x}[k] = \sum_{n=0}^{L} x[n] e^{-\frac{2\pi}{L} kn},$$

$$x[n] = \frac{1}{L} \sum_{k=0}^{L} \hat{x}[k] e^{\frac{2\pi}{L} kn}.$$