Convolutional Neural Networks

Sylesh Suresh

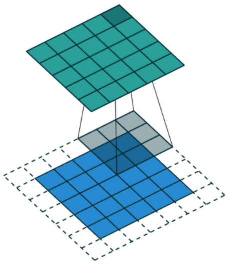
November 2018

1 Introduction

Convolutional neural networks (CNNs) are powerful neural network variants that are especially designed to perform well on data with features that are spatially dependent on each other. Thus, CNNs are one of the best algorithms for image analysis to date.

2 Convolutional Layers

Convolutional layers are similar to fully-connected/dense layers (standard neural network layers) in that both have neurons with weights that are attached to neurons from the previous layer. However, fully-connected layers connect each neuron to every single previous input with a weight. The disadvantage of the fully-connected approach is apparent when the input size is very large. This is generally the case with images. Consider a 200x200 pixel image with RGB values. Our input would have dimensions 200x200x3. A standard fully-connected layer would then have 120,000 weights per neuron, which would be incredibly computationally expensive to train.



The solution is to limit the number of input elements each neuron is connected to. Neurons in convolutional layers, specifically, are connected only to the inputs in their *receptive field*. In the image above, the green grid represents a convolutional layer while the blue grid represents the input image. Each square in the green grid represents a neuron, and each square in the blue grid represents an input element, or a pixel in our case. Notice we are dealing with a two-dimensional input for the sake of simplicity.

Each convolutional neuron is connected to the elements in its receptive field, illustrated by the "shadow" cast by the green square on the blue grid. The convolutional neuron only has weights connected to these input elements, being completely disconnected from all the other input elements. The next neuron is connected to the elements in its own receptive field, which is the same as the previous neuron's receptive field except shifted one over to the right. There are nine connections per neuron in this case. Note that the input layer is padded with a border of zeros (represented by the white squares) in order to ensure that the convolutional neurons on the border of the layer have the same receptive field size as the other convolutional neurons. This is called *zero-padding*.

To reduce the number of weights even further, we make every neuron share their weights. The nine weights associated with each receptive field are the same. Another way of looking at this: there is only one actual neuron. This one neuron has a specific set of nine attached weights, and it computes the output by scanning the input image left-to-right and top-to-bottom.

2.1 Filters

This will become more clear with a specific mathematical example. For the sake of simplicity, let us imagine a simpler situation where the input layer is a 2x2 (with zero-padding, making it a 4x4 including the zeros) and a convolutional layer of size 3x3 whose receptive field size is 2x2. In terms of the picture, imagine that the green grid is of the size 3x3, the blue grid is of the size 4x4 (including the zero-padding), and the shadow cast upon the blue grid is of size 2x2.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The matrix on the left represents the input (green grid), while the matrix on the right represents the weights of the convolutional connections (shadow) to the input layer. In CNNs, weight matrices are often referred to as *filters* or *kernels*. The actual convolutional layer is not shown here in matrix form - that is, there is no matrix representation of what the green square represented in the picture before. The input layer and the connection weights of the filter are all we need to compute the output of the convolutional layer. The operation we perform is fittingly called the convolution, where we stride the filter across the input matrix. At each stride step, we compute the Hadamard (element-wise) product of the weight matrix and the subset of the input matrix to calculate the corresponding element of the output matrix. Thus, performing the convolution operation:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 5 & 2 \\ 7 & 14 & 6 \\ 3 & 7 & 4 \end{bmatrix}$$

For the top-left element of the output, (working left-to-right and top-to-bottom) $0*1+0*1+0*1+1*2 = 2$. For the element directly to the right, 0*1 + 0*1 + 1*1 + 2*2 = 5. Next, $0*1+0*1+2*1+0*2 = 2$. This process continues until we fill out the entire 3x3 output matrix. Generally, in convolutional neural networks, we not only perform this convolution operation but also add a bias matrix to the final matrix output. This bias matrix has the same dimensions as the final matrix output, and all the bias matrix's elements are always the same. This means that the output of a convolutional layer (where matrix $\boldsymbol{X}$ is the input matrix, $\boldsymbol{W}$ is the filter, and $\boldsymbol{b}$ is the bias matrix) is $X * W + b$, which should look like a very familiar expression, as it looks just like the output of a fully-connected neuron before the activation function, except instead of standard matrix multiplication, we use the convolution operator. The result of applying the filter to the input matrix (including the addition of bias) is called a *feature map*.

2.2 Stride

Notice that convolutional layers, by nature, reduce the dimensionality of the input. We can further reduce the dimensionality by increasing the stride when we perform the convolution operation. Instead of moving the filter across the input matrix one column to the right until we reach the end and then one row down, we can increase the horizontal and/or vertical stride. That is, we can move more than one column/row at each step.

Again, this is best illustrated by a worked example. Let us introduce stride to the previously worked out convolution operation. We will first use a horizontal stride of 2 and keep the vertical stride at 1. This means that we move the filter horizontally across the input matrix two columns at a time. Computing the result, we arrive at:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 7 & 6 \\ 3 & 4 \end{bmatrix}$$

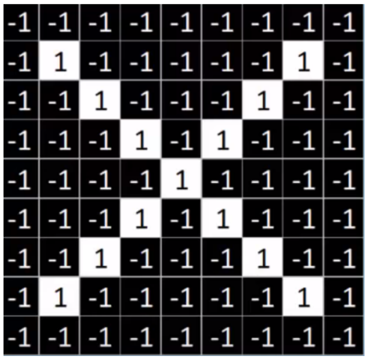
Notice that the dimensionality of the feature map is reduced. Using a horizontal and vertical stride of 1, the dimensionality was 3x3. But by increasing the horizontal stride, we have reduced it to 3x2. We can also increase the vertical stride. Let us now work out an example where we use a horizontal stride of 2 as well as a vertical stride of 2. This would result in:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix}$$

Notice that the dimensionality is even further reduced to 2x2.

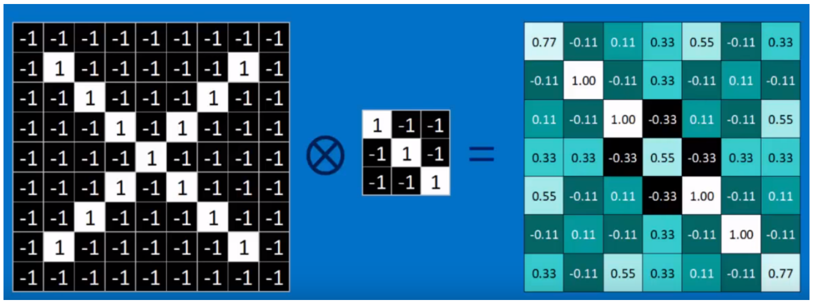
2.3 Feature Maps

When we apply a filter to an input image, we produce a feature map. This transforms the original input according to the filter. This means that filters can highlight and amplify parts of the image. Consider the following input image:



-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Different filters will produce different feature maps, each highlighting parts of the image that look most similar to the filter applied. Consider the three following feature maps:



-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

⊗

1	-1	-1
-1	1	-1
-1	-1	1

=

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

You can see that the high values of the feature map correspond to locations on the original input image that look very similar to the filter. A filter consisting of a diagonal line running from the top left to the bottom right will highlight parts of the image where there is a similarly oriented diagonal line. It makes sense, then, to use multiple filters to create multiple feature maps in order to amplify/detect more parts of the image. This is exactly what convolutional layers do. A convolutional layer can use k filters to create k feature maps. This means that for 2D images, a convolutional layer's output creates k 2D feature maps, or a 3D matrix. Subsequent convolutional layers create their output/feature maps by performing convolutions in each feature map that the last layer created. This is a bit harder to visualize, but mathematically, we are simply extending into another dimension. To wrap our heads around this, let us first put the standard, single filter convolutional case in indexed form - where we can calculate, at a particular row i and column j, the output of the nth convolutional layer - $z_{i,j,n}$.

$$z_{i,j,n} = b + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i',j',n-1} w_{u,v,n}$$

Here, b is the bias (which is the same no matter the row and column, which is why it has no subscript). f_h is the height of the receptive field, while f_w is the width of it. $i' = i * s_h + u$ where s_h is the vertical stride and $j' = j * s_w + v$ where s_w is the horizontal stride. $x_{i',j',n-1}$ is the value of the $(n-1)$th layer at row i' and column j'. $w_{u,v,n}$ is the value of the nth layer's filter at row u and column v. We are simply putting the convolution operator (as well as bias) into indexed terms in this equation.

From this, the jump to using multiple filters/feature maps should seem more understandable. In indexed terms, the equation is:

$$z_{i,j,k,n} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{m-1} x_{i',j',k',n-1} w_{u,v,n}$$

3 Activation Function

Before we go into the mathematics of convolutional layers, we will firstly discuss the activation functions used. As we know from before:

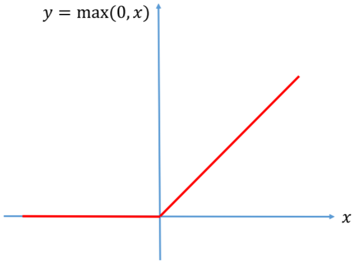
$$n = \sigma(w_1 n_1 + w_2 n_2 + b)$$

Previously, we have used the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

As we covered in our fourth neural networks lecture on hyper parameters, neural networks suffer from the vanishing gradient problem. To briefly recap, the gradient rapidly approaches 0 as we go deeper into the network as the derivative of the sigmoid is always less than 1. This is because when the delta for subsequent layers is calculated, the derivative of the activation function is multiplied to the values.

$$\delta_i = W_{i+1}^T \delta_{i+1} \odot \sigma'(W_i x_{i-1})$$

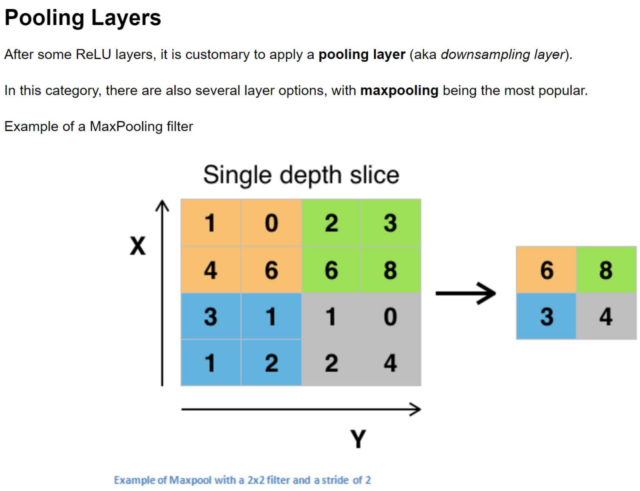


To account for this, we use the relu function. As the function is a line with a slope of 1, the derivative of the activation function is 1 and the gradient no longer approaches 0 for deeper layers. This allows us to build far greater networks. The reason the left side of the function is 0 is because it makes it non-linear. Non-linearity allows the network to detect the presence of something. In this function, it either says a feature is not present or it is present and here is the confidence, as given by the sum of the weights and biases. Theoretically, a derivative is not present if x is 0. However, in practice x is never exactly 0 so this is not an issue.

4 Pooling Layers

Even with sparse connections, convolutional layers produce a lot of data. If we look for many features, we greatly increase the size of the subsequent hidden layer. However, a lot of this information isn't very useful. To remove this extra information, we use pooling layers.

4.1 Max Pooling



Pooling Layers

After some ReLU layers, it is customary to apply a **pooling layer** (aka *downsampling layer*).

In this category, there are also several layer options, with **maxpooling** being the most popular.

Example of a MaxPooling filter

Single depth slice

1	0	2	3
4	6	6	8
3	1	1	0
1	2	2	4

6	8
3	4

Example of Maxpool with a 2x2 filter and a stride of 2

Pooling layers take the highest value in each region and only track that. This removes the exact pixel location information and actually helps in making the network more generalizable while decreasing data by 1/4. This greatly increases speed while increasing robustness. The most common method for this technique is max pooling. Max pooling takes the size of the pooling region, which is typically 2 x 2, and takes the highest value from that region. The intuition is that each kernel is looking for a shape, so if the value is highest in one part, that is closest to where the shape is present and therefore is the significant value.