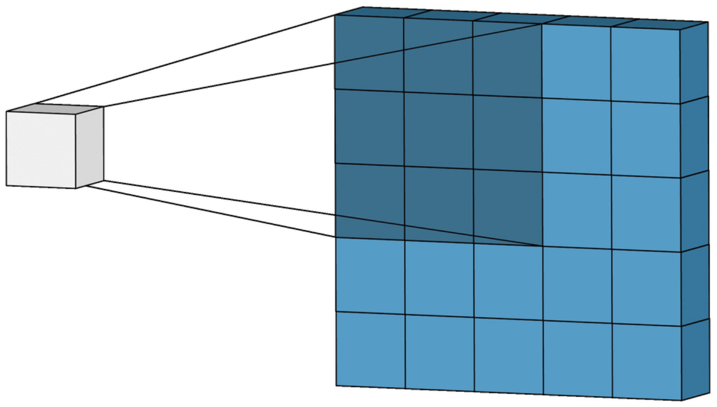
Irhum Shafkat Follow
Curious programmer, tinkers around in Python and deep learning.
Jun 1 · 15 min read

Intuitively Understanding Convolutions for Deep Learning

Exploring the strong visual hierarchies that makes them work

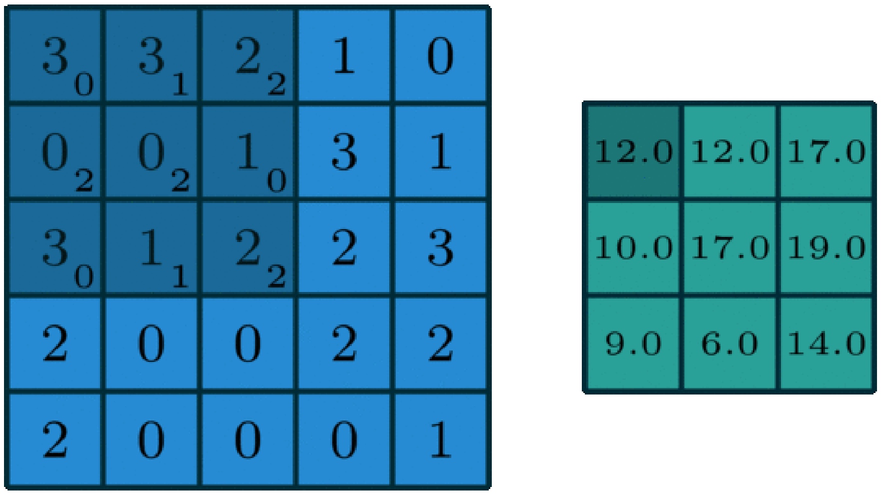


The advent of powerful and versatile deep learning frameworks in recent years has made it possible to implement convolution layers into a deep learning model an extremely simple task, often achievable in a single line of code.

However, understanding convolutions, especially for the first time can often feel a bit unnerving, with terms like kernels, filters, channels and so on all stacked onto each other. Yet, convolutions as a concept are fascinatingly powerful and highly extensible, and in this post, we'll break down the mechanics of the convolution operation, step-by-step, relate it to the standard fully connected network, and explore just how they build up a strong visual hierarchy, making them powerful feature extractors for images.

2D Convolutions: The Operation

The 2D convolution is a fairly simple operation at heart: you start with a kernel, which is simply a small matrix of weights. This kernel "slides" over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.



A standard convolution[1]

The kernel repeats this process for every location it slides over, converting a 2D matrix of features into yet another 2D matrix of features. The output features are essentially, the weighted sums (with the weights being the values of the kernel itself) of the input features located roughly in the same location of the output pixel on the input layer.

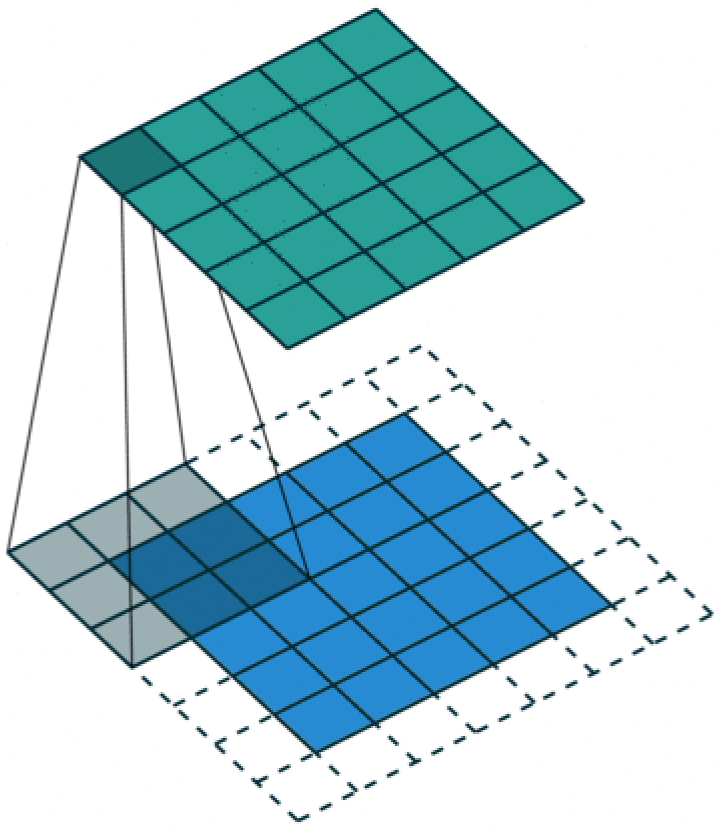
Whether or not an input feature falls within this "roughly same location", gets determined directly by whether it's in the area of the kernel that produced the output or not. This means the size of the kernel directly determines how many (or few) input features get combined in the production of a new output feature.

This is all in pretty stark contrast to a fully connected layer. In the above example, we have 5×5=25 input features, and 3×3=9 output features. If this were a standard fully connected layer, you'd have a weight matrix of 25×9 = 225 parameters, with every output feature being the weighted sum of *every single* input feature. Convolutions allow us to do this transformation with only 9 parameters, with each output feature, instead of “looking at” every input feature, only getting to “look” at input features coming from roughly the same location. Do take note of this, as it’ll be critical to our later discussion.

Some commonly used techniques

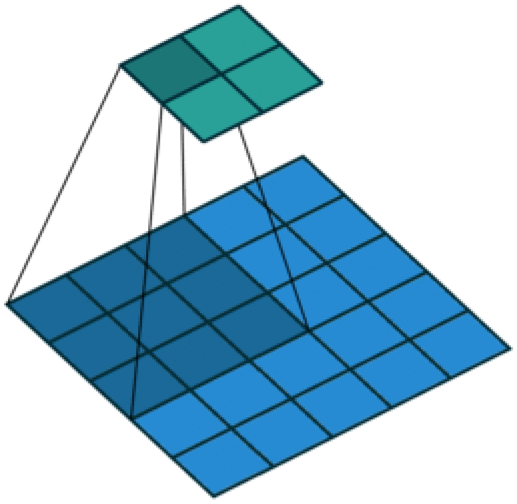
Before we move on, it’s definitely worth looking into two techniques that are commonplace in convolution layers: Padding and Strides.

- Padding: If you see the animation above, notice that during the sliding process, the edges essentially get “trimmed off”, converting a 5×5 feature matrix to a 3×3 one. The pixels on the edge are never at the center of the kernel, because there is nothing for the kernel to extend to beyond the edge. This isn’t ideal, as often we’d like the size of the output to equal the input.


Same padding[1]

Padding does something pretty clever to solve this: pad the edges with extra, “fake” pixels (usually of value 0, hence the oft-used term “zero padding”). This way, the kernel when sliding can allow the original edge pixels to be at its center, while extending into the fake pixels beyond the edge, producing an output the same size as the input.

- Striding: Often when running a convolution layer, you want an output with a lower size than the input. This is commonplace in convolutional neural networks, where the size of the spatial dimensions are reduced when increasing the number of channels. One way of accomplishing this is by using a pooling layer (eg. taking the average/max of every 2×2 grid to reduce each spatial dimensions in half). Yet another way to do is is to use a stride:



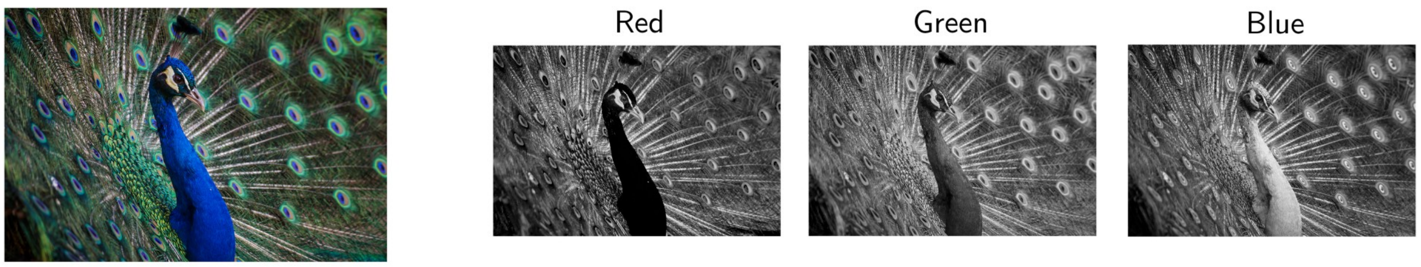
A stride 2 convolution[1]

The idea of the stride is to *skip* some of the slide locations of the kernel. A stride of 1 means to pick slides a pixel apart, so basically every single slide, acting as a standard convolution. A stride of 2 means picking slides 2 pixels apart, skipping every other slide in the process, downsizing by roughly a factor of 2, a stride of 3 means skipping every 2 slides, downsizing roughly by factor 3, and so on.

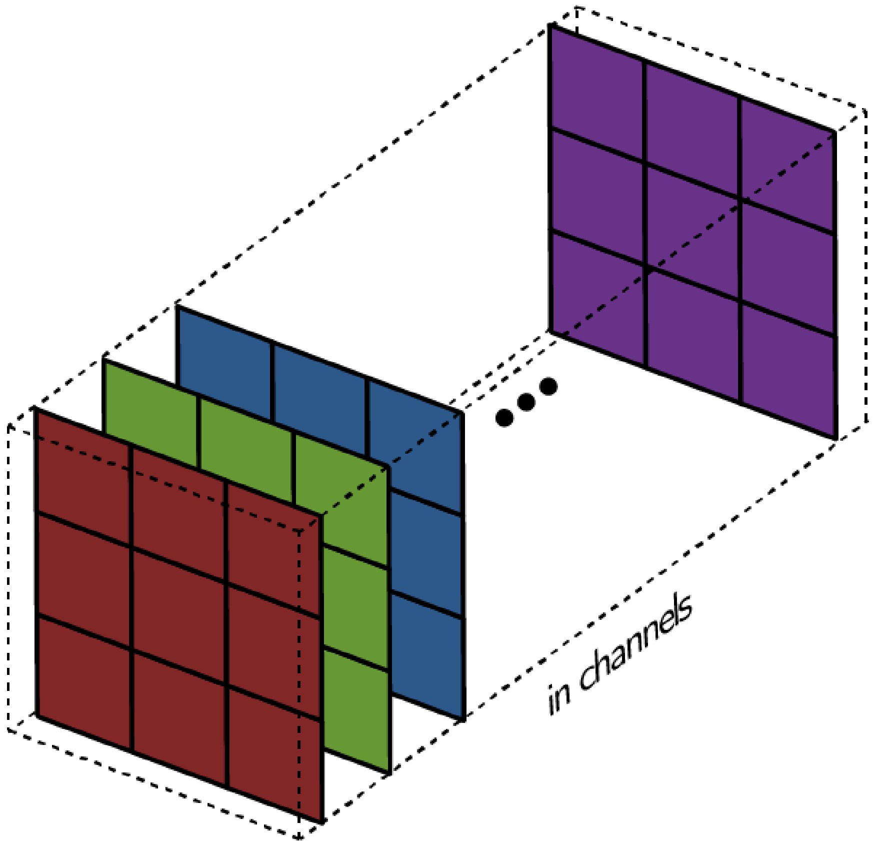
More modern networks, such as the ResNet architectures entirely forgo pooling layers in their internal layers, in favor of strided convolutions when needing to reduce their output sizes.

The multi-channel version

Of course, the diagrams above only deals with the case where the image has a single input channel. In practicality, most input images have 3 channels, and that number only increases the deeper you go into a network. It's pretty easy to think of channels, in general, as being a "view" of the image as a whole, emphasising some aspects, de-emphasising others.



Most of the time, we deal with RGB images with three channels. (Credit: Andre Mouton)

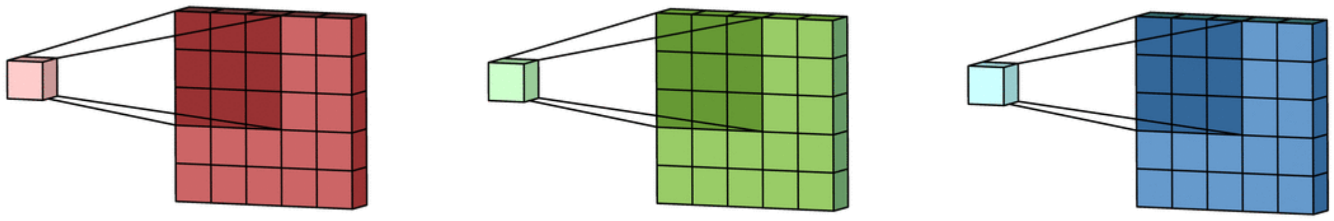


A filter: A collection of kernels

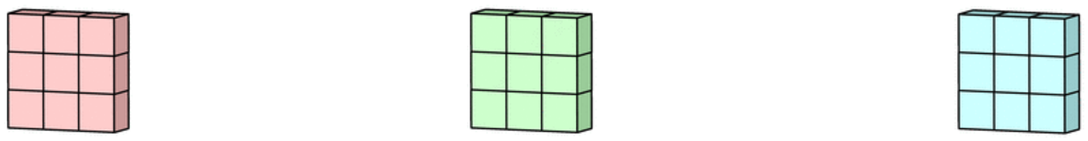
So this is where a key distinction between terms comes in handy: whereas in the 1 channel case, where the term filter and kernel are interchangeable, in the general case, they're actually pretty different. Each filter actually happens to be a *collection of kernels*, with there being one kernel for every single input channel to the layer, and each kernel being unique.

Each filter in a convolution layer produces one and only one output channel, and they do it like so:

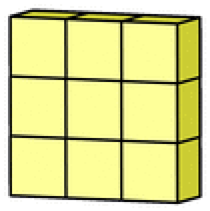
Each of the kernels of the filter "slides" over their respective input channels, producing a processed version of each. Some kernels may have stronger weights than others, to give more emphasis to certain input channels than others (eg. a filter may have a red kernel channel with stronger weights than others, and hence, respond more to differences in the red channel features than the others).



Each of the per-channel processed versions are then summed together to form *one* channel. The kernels of a filter each produce one version of each channel, and the filter as a whole produces one overall output channel.



Finally, then there's the bias term. The way the bias term works here is that each output filter has one bias term. The bias gets added to the output channel so far to produce the final output channel.



And with the single filter case down, the case for any number of filters is identical: Each filter processes the input with its own, different set of kernels and a scalar bias with the process described above, producing a single output channel. They are then concatenated together to produce the overall output, with the number of output channels being the number of filters. A nonlinearity is then usually applied before passing this as input to another convolution layer, which then repeats this process.

2D Convolutions: The Intuition

Convolutions are still linear transforms

Even with the mechanics of the convolution layer down, it can still be hard to relate it back to a standard feed-forward network, and it still doesn't explain why convolutions scale to, and work so much better for image data.

Suppose we have a 4×4 input, and we want to transform it into a 2×2 grid. If we were using a feedforward network, we'd reshape the 4×4 input into a vector of length 16, and pass it through a densely connected layer with 16 inputs and 4 outputs. One could visualize the weight matrix **W** for a layer:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} & w_{1,9} & w_{1,10} & w_{1,11} & w_{1,12} & w_{1,13} & w_{1,14} & w_{1,15} & w_{1,16} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} & w_{2,9} & w_{2,10} & w_{2,11} & w_{2,12} & w_{2,13} & w_{2,14} & w_{2,15} & w_{2,16} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} & w_{3,9} & w_{3,10} & w_{3,11} & w_{3,12} & w_{3,13} & w_{3,14} & w_{3,15} & w_{3,16} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} & w_{4,9} & w_{4,10} & w_{4,11} & w_{4,12} & w_{4,13} & w_{4,14} & w_{4,15} & w_{4,16} \end{bmatrix}$$

All in all, some 64 parameters

And although the convolution kernel operation may seem a bit strange at first, it is still a linear transformation with an equivalent transformation matrix. If we were to use a kernel **K** of size 3 on the reshaped 4×4 input to get a 2×2 output, the equivalent transformation matrix would be:

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

There's really just 9 parameters here.

(Note: while the above matrix is an *equivalent* transformation matrix, the actual operation is usually implemented as a very different matrix multiplication[2])

The convolution then, as a whole, is still a linear transformation, but at the same time it's also a dramatically different kind of transformation. For a matrix with 64 elements, there's just 9 parameters which themselves are reused several times. Each output node only gets to see a select number of inputs (the ones inside the kernel). There is no interaction with any of the other inputs, as the weights to them are set to 0.

It's useful to see the convolution operation as a *hard prior* on the weight matrix. In this context, by prior, I mean predefined network parameters. For example, when you use a pretrained model for image classification, you use the *pretrained network parameters* as your prior, as a feature extractor to your final densely connected layer.

In that sense, there's a direct intuition between why both are so efficient (compared to their alternatives). Transfer learning is efficient by orders of magnitude compared to random initialization, because you only really need to optimize the parameters of the final fully connected layer, which means you can have fantastic performance with only a few dozen images per class.

Here, you don't need to optimize all 64 parameters, because we set most of them to zero (and they'll stay that way), and the rest we convert to shared parameters, resulting in only 9 actual parameters to optimize. This efficiency matters, because when you move from the 784 inputs of MNIST to real world 224×224×3 images, thats over 150,000 inputs. A dense layer attempting to halve the input to 75,000 inputs would still require over 10 *billion* parameters. For comparison, the *entirety* of ResNet-50 has some 25 million parameters.

So fixing some parameters to 0, and tying parameters increases efficiency, but unlike the transfer learning case, where we know the prior is good because it works on a large general set of images, how do we know *this* is any good?

The answer lies in the feature combinations the prior leads the parameters to learn.

Locality

Early on in this article, we discussed that:

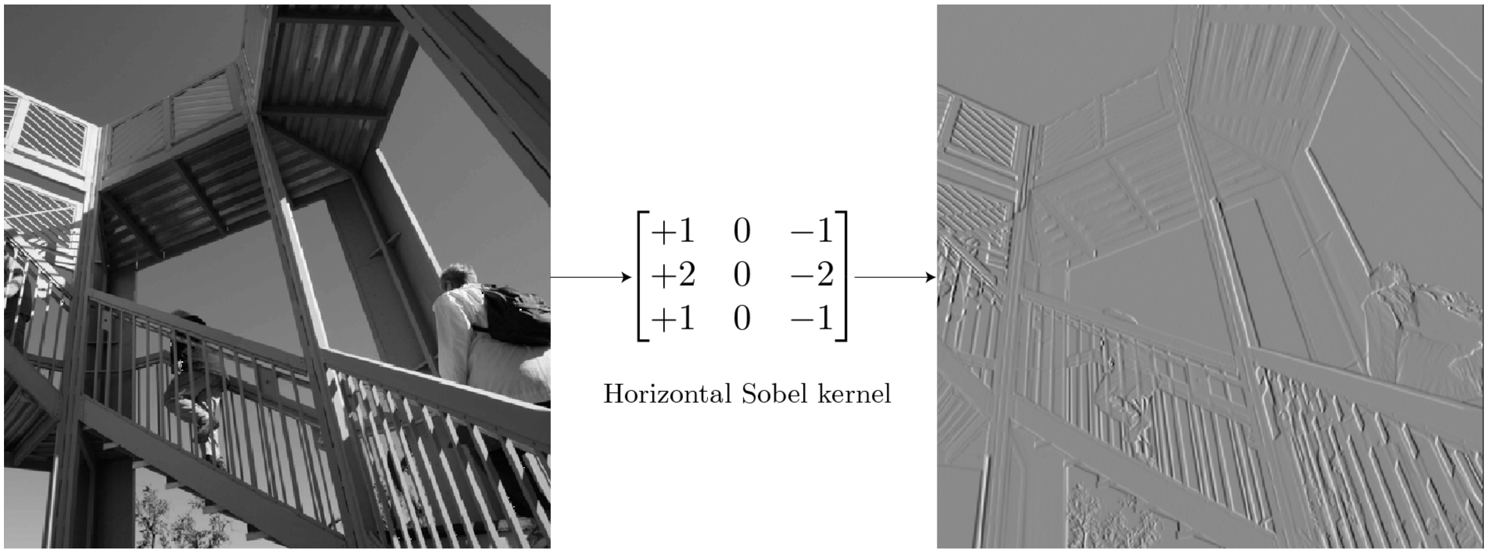
- Kernels combine pixels only from a small, local area to form an output. That is, the output feature only "sees" input features from a small local area.
- The kernel is applied globally across the whole image to produce a matrix of outputs.

So with backpropagation coming in all the way from the classification nodes of the network, the kernels have the interesting task of learning weights to produce features only from a set of local inputs. Additionally, because the kernel itself is applied across the entire image, the features the kernel learns must be general enough to come from any part of the image.

If this were any other kind of data, eg. categorical data of app installs, this would've been a disaster, for just because your number of *app installs* and *app type* columns are next to each other doesn't mean they have any "local, shared features" common with *app install dates* and *time used*. Sure, the four may have an underlying higher level feature (eg. which apps people want most) that can be found, but that gives us no reason to believe the parameters for the first two are exactly the same as the parameters for the latter two. The four could've been in any (consistent) order and still be valid!

Pixels however, always appear in a consistent order, and nearby pixels influence a pixel e.g. if all nearby pixels are red, it's pretty likely the pixel is also red. If there are deviations, that's an interesting anomaly that could be converted into a feature, and all this can be detected from comparing a pixel with its neighbors, with other pixels in its locality.

And this idea is really what a lot of earlier computer vision feature extraction methods were based around. For instance, for edge detection, one can use a Sobel edge detection filter, a kernel with fixed parameters, operating just like the standard one-channel convolution:



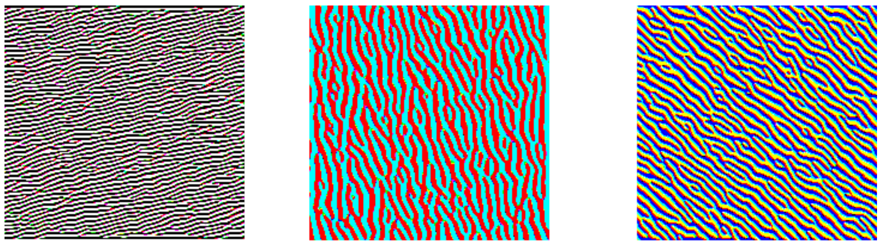
$$\xrightarrow{} \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \xrightarrow{}$$

Applying a vertical edge detector kernel

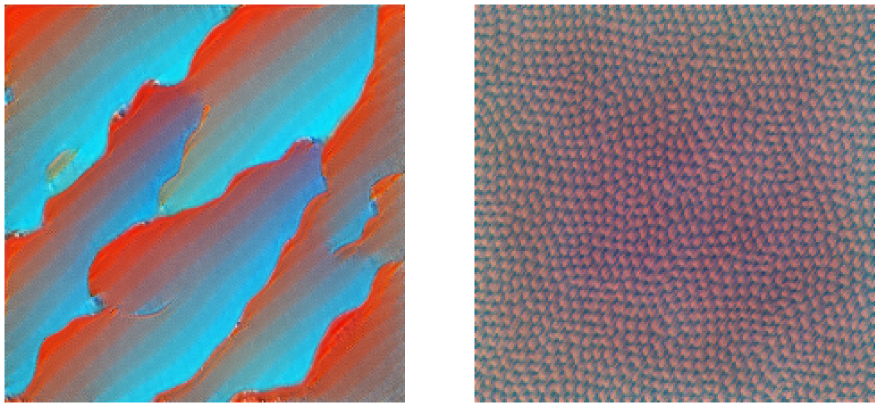
For a non-edge containing grid (eg. the background sky), most of the pixels are the same value, so the overall output of the kernel at that point is 0. For a grid with an vertical edge, there is a difference between the pixels to the left and right of the edge, and the kernel computes that difference to be non-zero, activating and revealing the edges. The kernel only works only a 3×3 grids at a time, detecting anomalies on a local scale, yet when applied across the entire image, is enough to detect a certain feature on a global scale, anywhere in the image!

So the key difference we make with deep learning is ask this question: Can useful kernels be learnt? For early layers operating on raw pixels, we could reasonably expect feature detectors of fairly low level features, like edges, lines, etc.

There's an entire branch of deep learning research focused on making neural network models interpretable. One of the most powerful tools to come out of that is Feature Visualization using optimization[3]. The idea at core is simple: optimize a image (usually initialized with random noise) to activate a filter as strongly as possible. This does make intuitive sense: if the optimized image is completely filled with edges, that's strong evidence that's what the filter itself is looking for and is activated by. Using this, we can peek into the learnt filters, and the results are stunning:



Feature visualization for 3 different channels from the 1st convolution layer of GoogLeNet[3]. Notice that while they detect different types of edges, they're still low-level edge detectors.



Feature Visualization of channel 12 from the 2nd and 3rd convolutions[3]

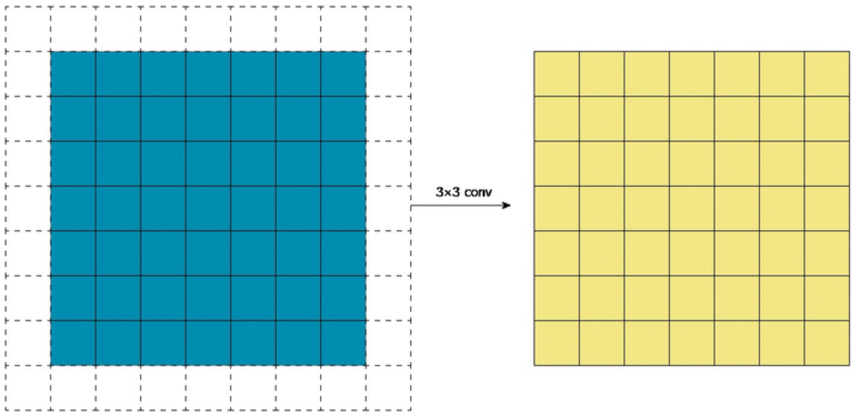
One important thing to notice here is that *convolved images are still images*. The output of a small grid of pixels from the top left of an image will still be on the top left. So you can run another convolution layer on top of another (such as the two on the left) to extract deeper features, which we visualize.

Yet, however deep our feature detectors get, without any further changes they'll still be operating on very small patches of the image. No matter how deep your detectors are, you can't detect faces from a 3×3 grid. And this is where the idea of the receptive field comes in.

Receptive field

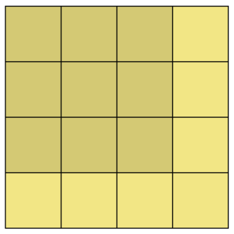
A essential design choice of any CNN architecture is that the input sizes grow smaller and smaller from the start to the end of the network, while the number of channels grow deeper. This, as mentioned earlier, is often done through strides or pooling layers. Locality determines what inputs from the previous layer the outputs get to see. The receptive field determines what area of the *original input* to the entire network the output gets to see.

The idea of a strided convolution is that we only process slides a fixed distance apart, and skip the ones in the middle. From a different point of view, we only keep outputs a fixed distance apart, and remove the rest[1].

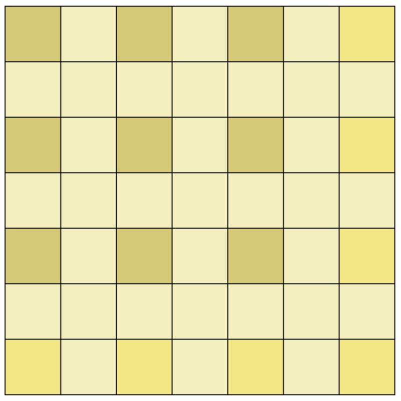


3×3 convolution, stride 2

We then apply a nonlinearity to the output, and per usual, then stack another new convolution layer on top. And this is where things get interesting. Even if were we to apply a kernel of the same size (3×3), having the same local area, to the output of the strided convolution, the kernel would have a larger effective receptive field:



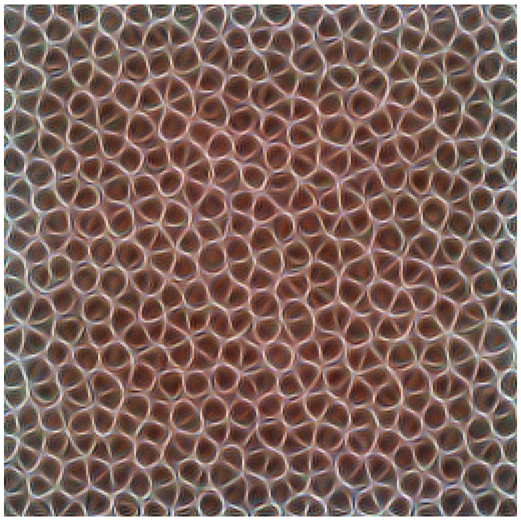
Operation on the true strided output



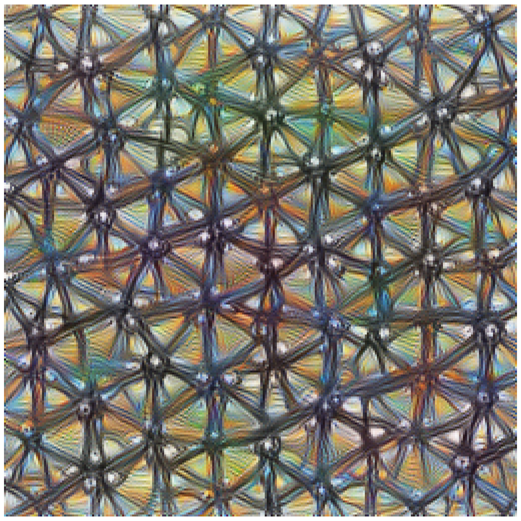
Operation on the output with the removed pixels

This is because the output of the strided layer still does represent the same image. It is not so much cropping as it is resizing, only thing is that each single pixel in the output is a "representative" of a larger area (of whose other pixels were discarded) from the same rough location from the original input. So when the next layer's kernel operates on the output, it's operating on pixels collected from a larger area.

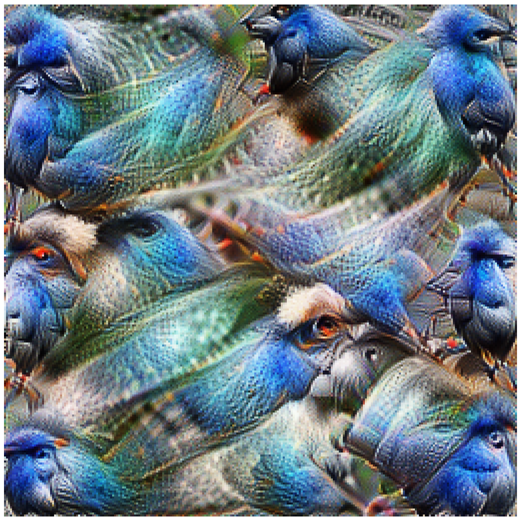
(Note: if you're familiar with dilated convolutions, note that the above is *not* a dilated convolution. Both are methods of increasing the receptive field, but dilated convolutions are a single layer, while this takes place on a regular convolution following a strided convolution, with a nonlinearity inbetween)



mixed3a, channel 31



mixed4a, channel 11



mixed5a, channel 14

Feature visualization of channels from each of the major collections of convolution blocks, showing a progressive increase in complexity[3]

This expansion of the receptive field allows the convolution layers to combine the low level features (lines, edges), into higher level features (curves, textures), as we see in the mixed3a layer.

Followed by a pooling/strided layer, the network continues to create detectors for even higher level features (parts, patterns), as we see for mixed4a.

The repeated reduction in image size across the network results in, by the 5th block on convolutions, input sizes of just 7×7, compared to inputs of 224×224. At this point, each *single* pixel represents a grid of 32×32 pixels, which is huge.

Compared to earlier layers, where an activation meant detecting an edge, here, an activation on the tiny 7×7 grid is one for a very high level feature, such as for birds.

The network as a whole progresses from a small number of filters (64 in case of GoogLeNet), detecting low level features, to a very large number of filters(1024 in the final convolution), each looking for an extremely specific high level feature. Followed by a final pooling layer, which collapses each 7×7 grid into a single pixel, each channel is a feature detector with a receptive field equivalent to the *entire* image.

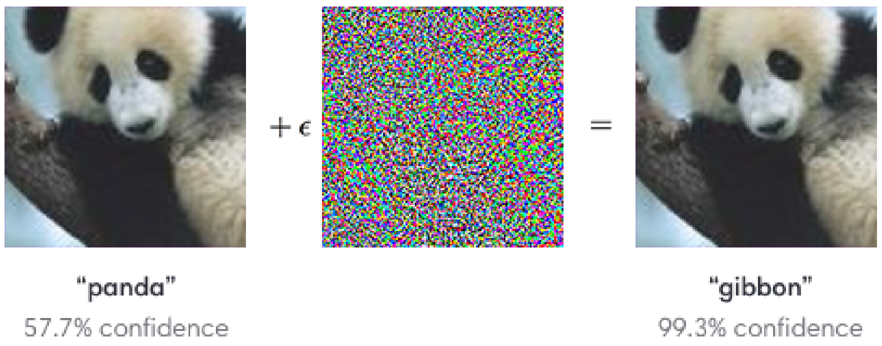
Compared to what a standard feedforward network would have done, the output here is really nothing short of awe-inspiring. A standard feedforward network would have produced abstract feature vectors, from combinations of every single pixel in the image, requiring intractable amounts of data to train.

The CNN, with the priors imposed on it, starts by learning very low level feature detectors, and as across the layers as its receptive field is expanded, learns to combine those low-level features into progressively higher level features; not an abstract combination of every single pixel, but rather, a strong *visual hierarchy* of concepts.

By detecting low level features, and using them to detect higher level features as it progresses up its visual hierarchy, it is eventually able to detect entire visual concepts such as faces, birds, trees, etc, and that's what makes them such powerful, yet efficient with image data.

A final note on adversarial attacks

With the visual hierarchy CNNs build, it is pretty reasonable to assume that their vision systems are similar to humans. And they're really great with real world images, but they also fail in ways that strongly suggest their vision systems aren't entirely human-like. The most major problem: Adversarial Examples[4], examples which have been specifically modified to fool the model.



To a human, both images are obviously pandas. To the model, not so much.[4]

Adversarial examples would be a non-issue if the only tampered ones that caused the models to fail were ones that even humans would notice. The problem is, the models are susceptible to attacks by samples which have only been tampered with ever so slightly, and would clearly not fool any human. This opens the door for models to silently fail, which can be pretty dangerous for a wide range of applications from self-driving cars to healthcare.

Robustness against adversarial attacks is currently a highly active area of research, the subject of many papers and even competitions, and solutions will certainly improve CNN architectures to become safer and more reliable.

Conclusion

CNNs were the models that allowed computer vision to scale from simple applications to powering sophisticated products and services, ranging from face detection in your photo gallery to making better medical diagnoses. They might be the key method in computer vision going forward, or some other new breakthrough might just be around the corner. Regardless, one thing is for sure: they're nothing short of amazing, at the heart of many present-day innovative applications, and are most certainly worth deeply understanding.

References

1. A guide to convolution arithmetic for deep learning
2. CS231n Convolutional Neural Networks for Visual Recognition—Convolutional Neural Networks
3. Feature Visualization—How neural networks build up their understanding of images (of note: the feature visualizations here were produced with the Lucid library, an open source implementation of the techniques from this journal article)
4. Attacking Machine Learning with Adversarial Examples

Further Resources

1. fast.ai—Lesson 3: Improving your Image Classifier
2. Conv Nets: A Modular Perspective
3. Building powerful image classification models using very little data

. . .

Hope you enjoyed this article! If you'd like to stay connected, you'll find me on Twitter here. If you have a question, comments are welcome!—I find them to be useful to my own learning process as well.