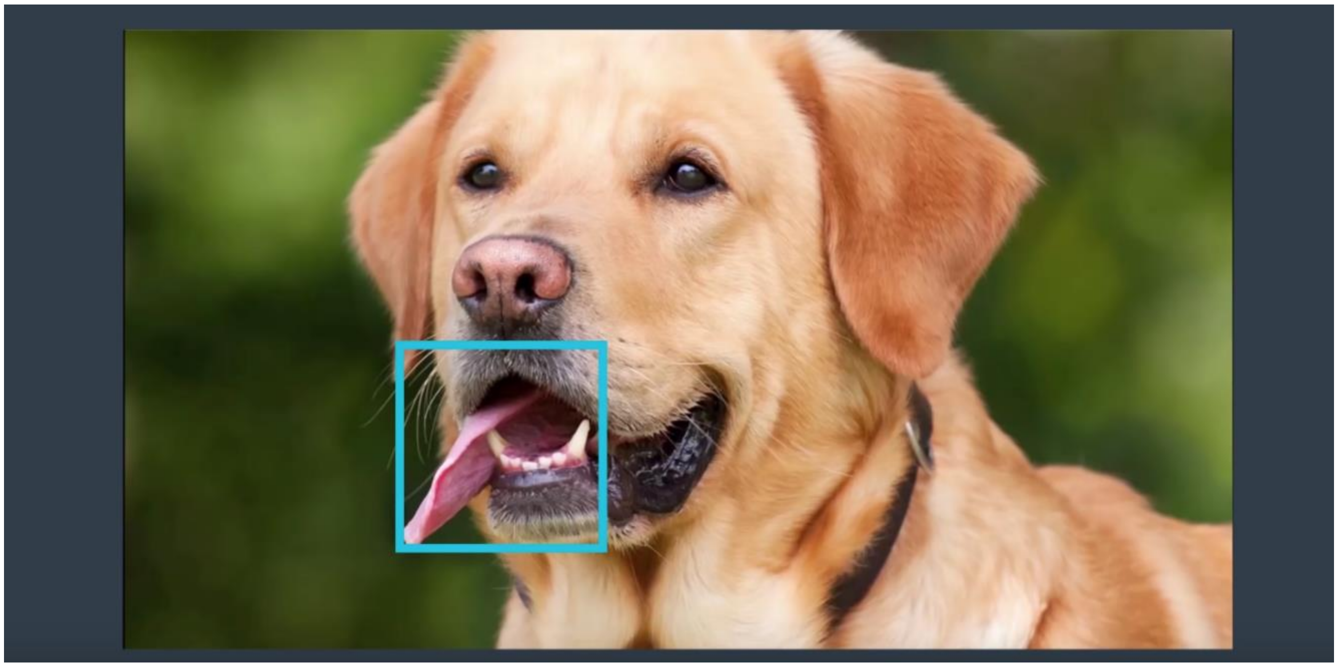
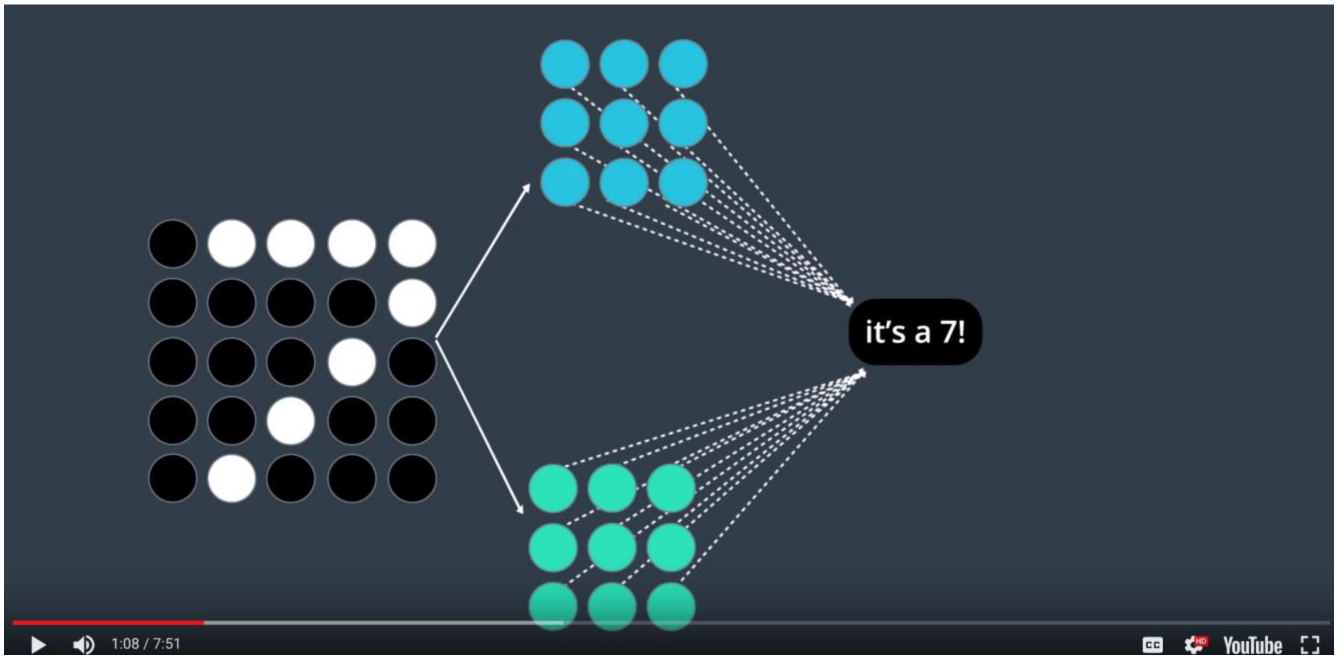
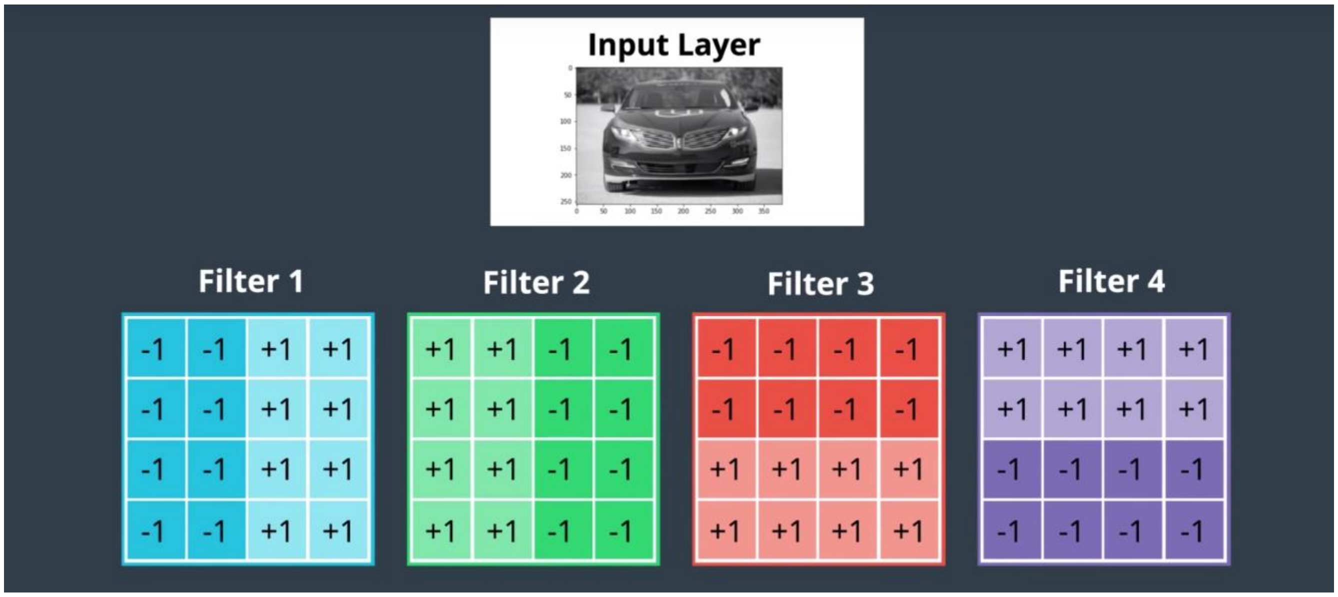


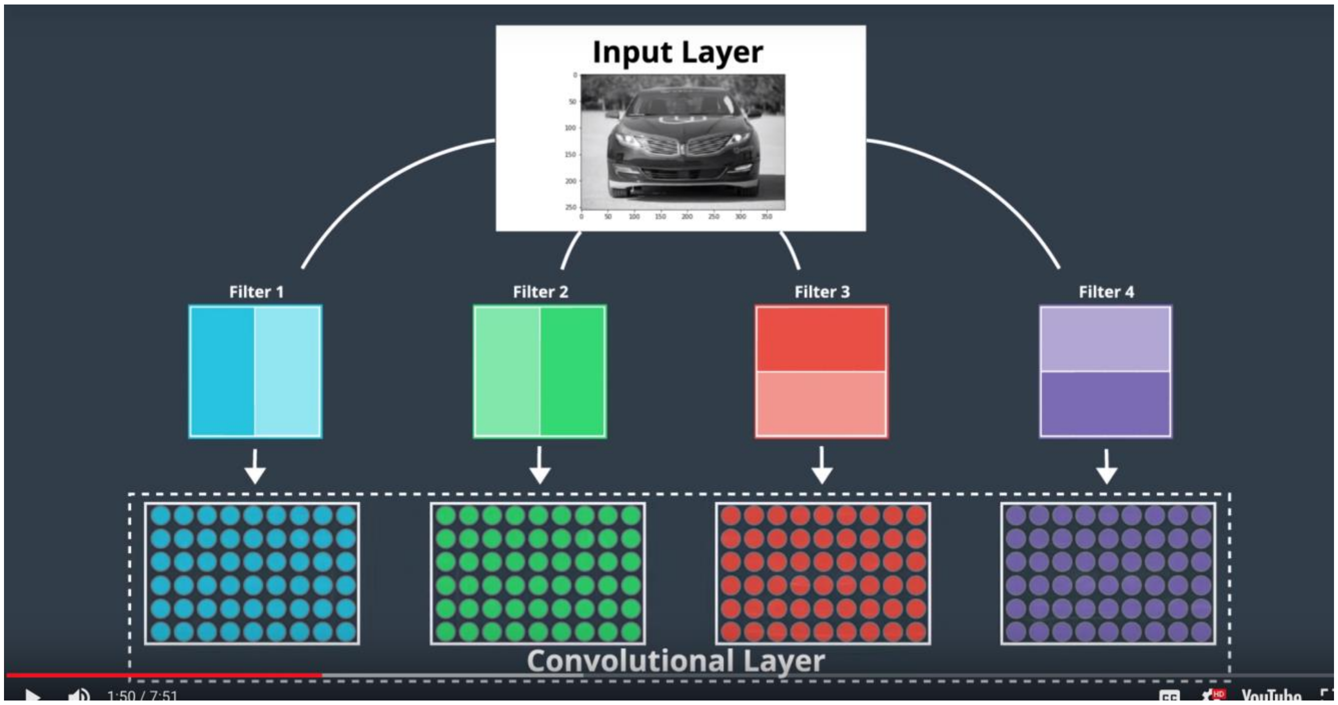
112. Consider this image of a dog. A single region in this image, may have many different patterns that we want to detect. Consider this region for instance, this region has teeth, some whiskers, and a tongue. In that case, to understand this image we need filters for detecting all three of these characteristics. <u>One for each of teeth, whiskers, and tongue</u>.



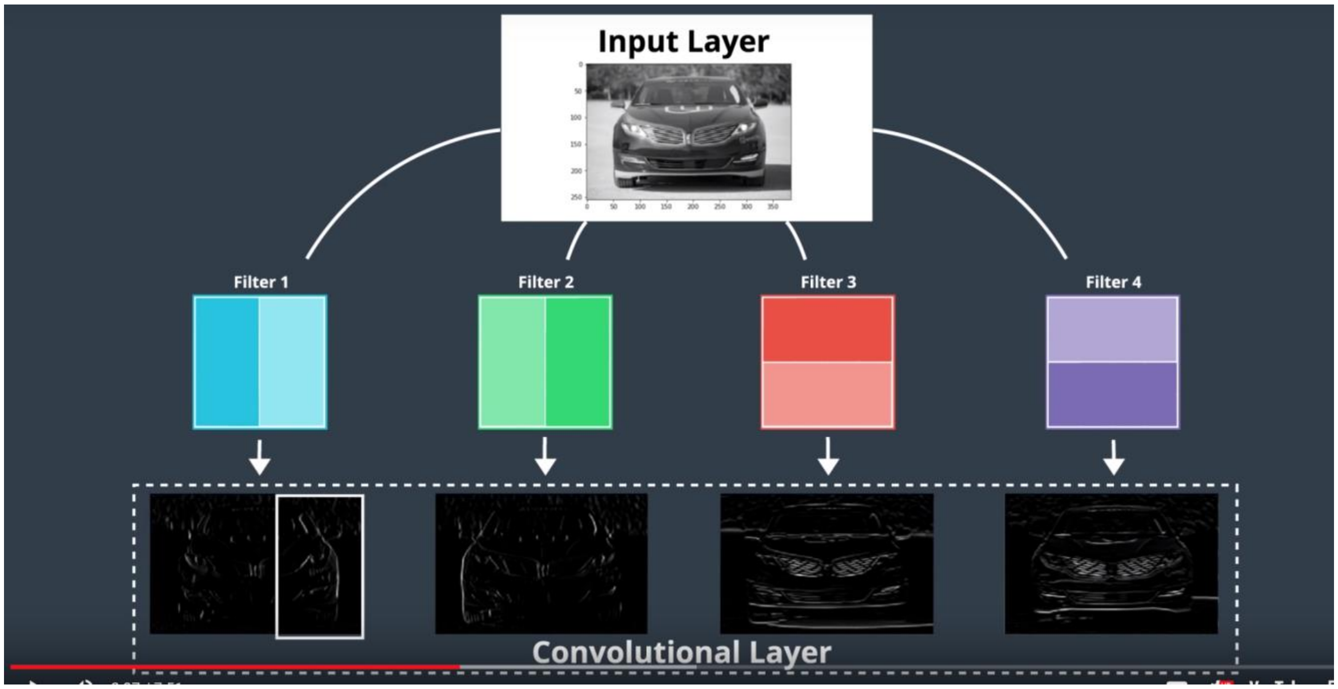
Recall the case of a single convolutional filter, adding another filter is probably exactly what you'd expect. Where we just populate an additional collection of nodes in the convolutional layer. This collection has its own shared set of weights that differ from the weights for the blue nodes above them. In fact, it's common to have tens to hundreds of these collections in a convolutional layer--each corresponding to their own filter. Let's now execute some code to see what these collections look like. After all, each is formatted in the same way as an image, namely as a matrix of values. We'll be visualizing the output of a Jupyter notebook. If you like and you can follow along with the link below.



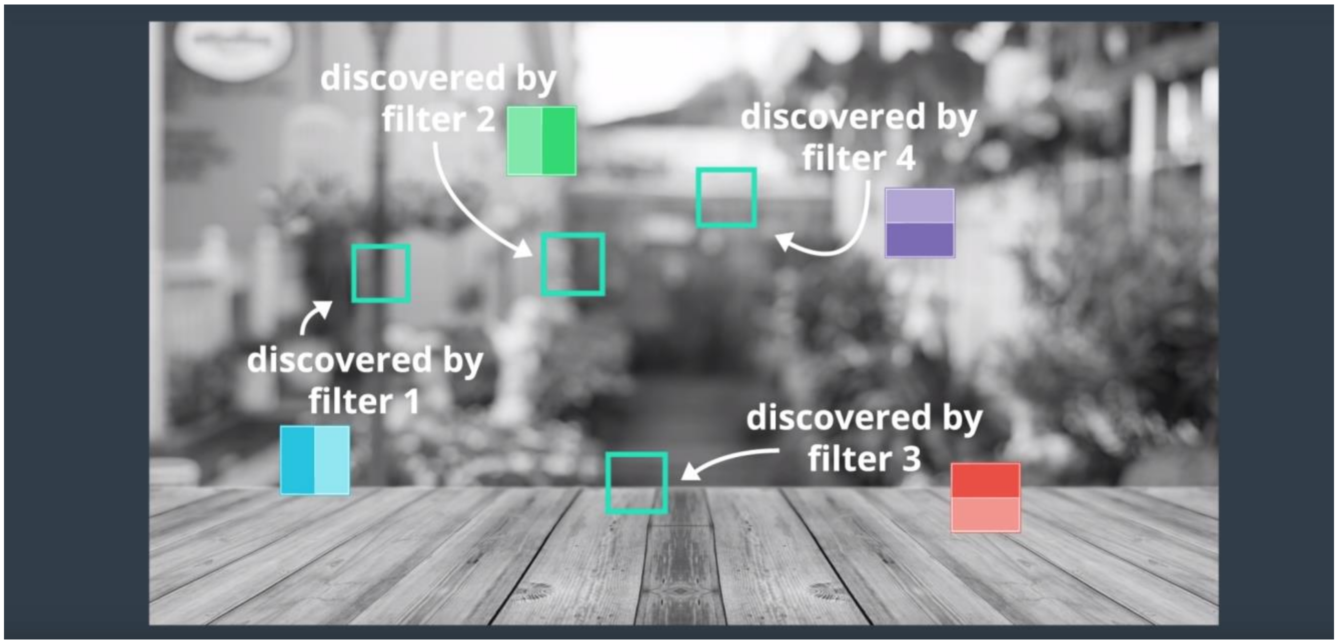
So, say we're working with an image of Udacity's self-driving car as input. Let's use four filters, each four pixels high and four pixels wide.



Recall each filter will be convolved across the height and width the image to produce an entire collection of nodes in the convolutional layer. In this case, since we have four filters, we'll have four collections of nodes. In practice of for to each of these four collections is either feature maps or as activation maps.



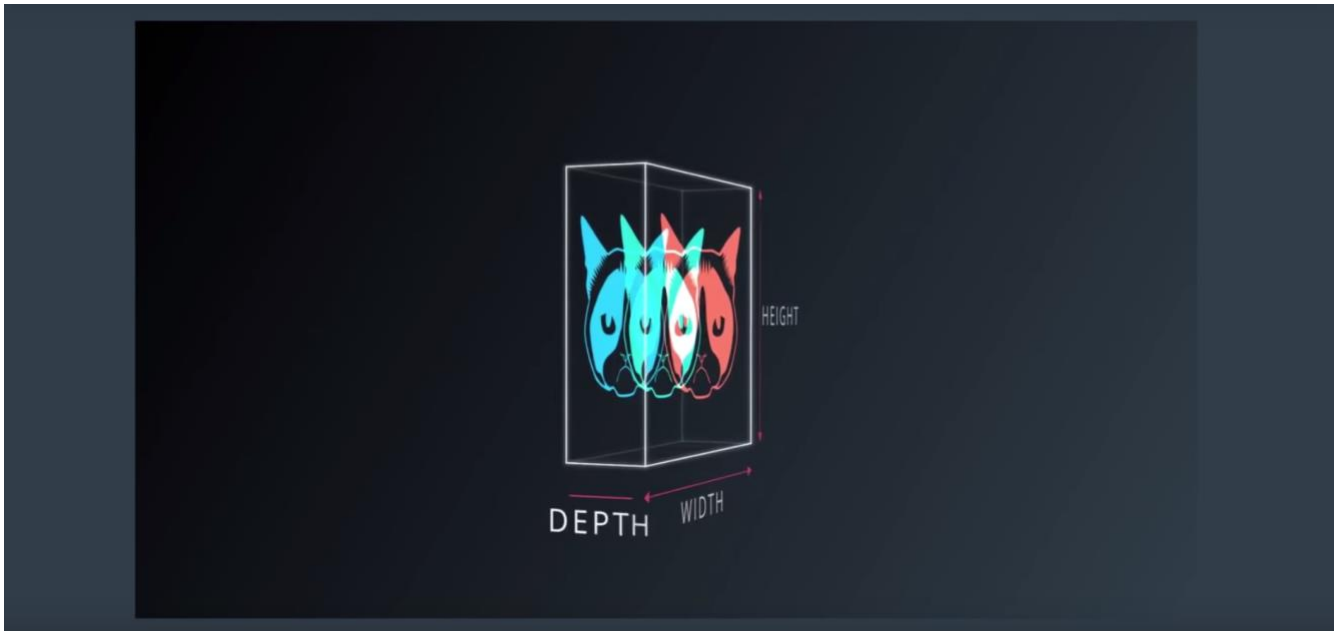
When we visualize these feature maps, we see that they look like filtered images. That is we've taken all of the complicated dense information in the original image and in each of these four cases outputted a much simpler image with less information. By peeking at the structure of the filters, you can see that the first two filters discover vertical edges, where the last two detect horizontal edges in the image. Remember that lighter values and the feature map mean that the pattern in the filter was detected in the image. So can you match the lighter regions in each feature map with their corresponding areas in the original image? In this activation map for instance, we can see a clear white line defining the right edge of the car.



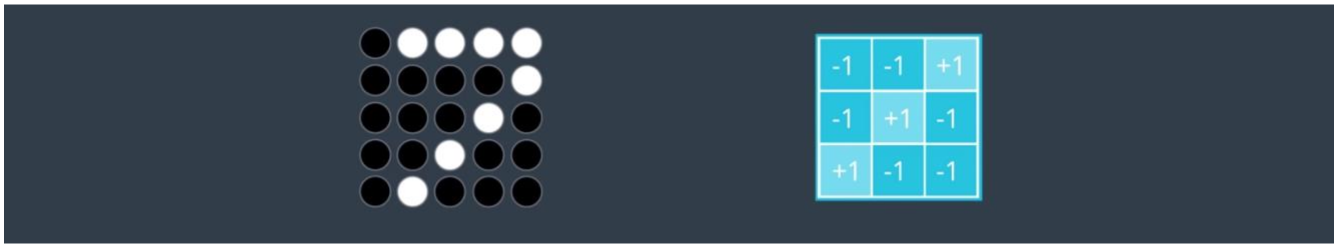
This is because all of the corresponding regions in the car image closely resemble the filter. Where we have a vertical line of dark pixels to the left of a vertical line of lighter pixels. If you think about it you'll notice that edges in images appear as a line of lighter pixels next to a line of darker pixels. This image for instance contains many regions that would be discovered or detected by one of the four filters we defined before. Filters that function as edge detectors are very important in CNNs and we'll revisit them later. So now we know how to understand convolutional layers that have a grayscale images input.

HEIGHT
WIDTH

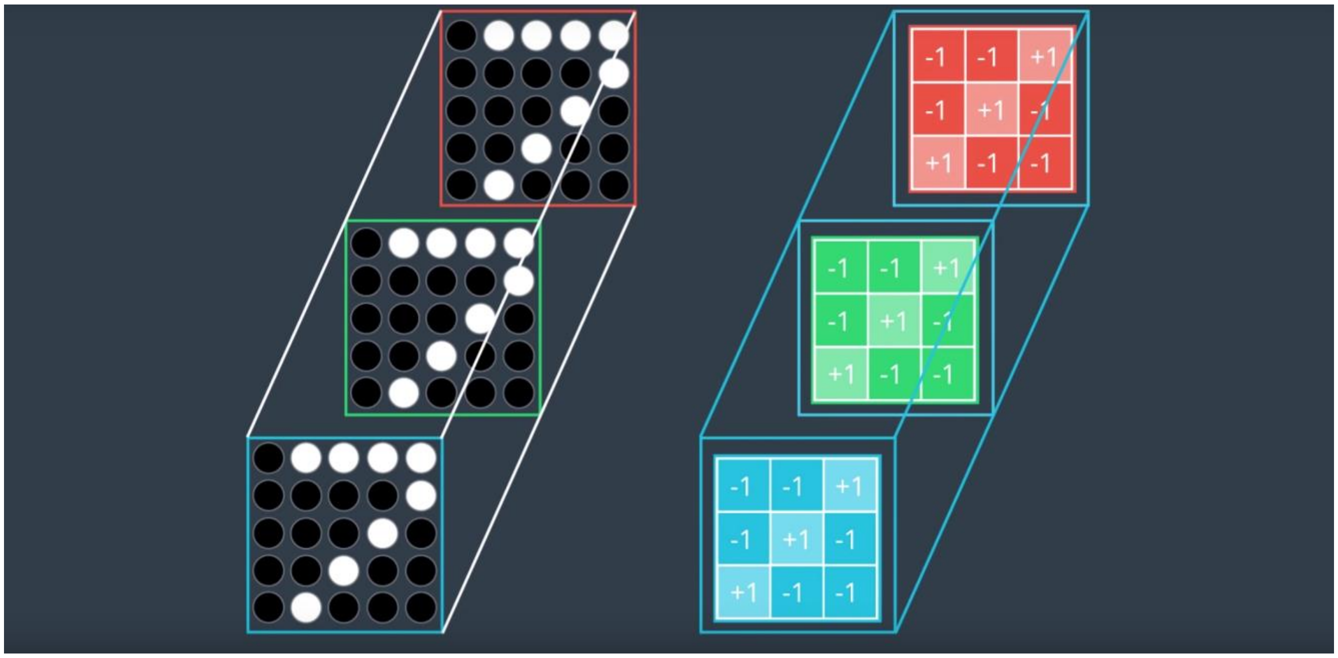
But what about color images? Well, we've seen that grayscale images are interpreted by the computer as a 2D array with height and width.



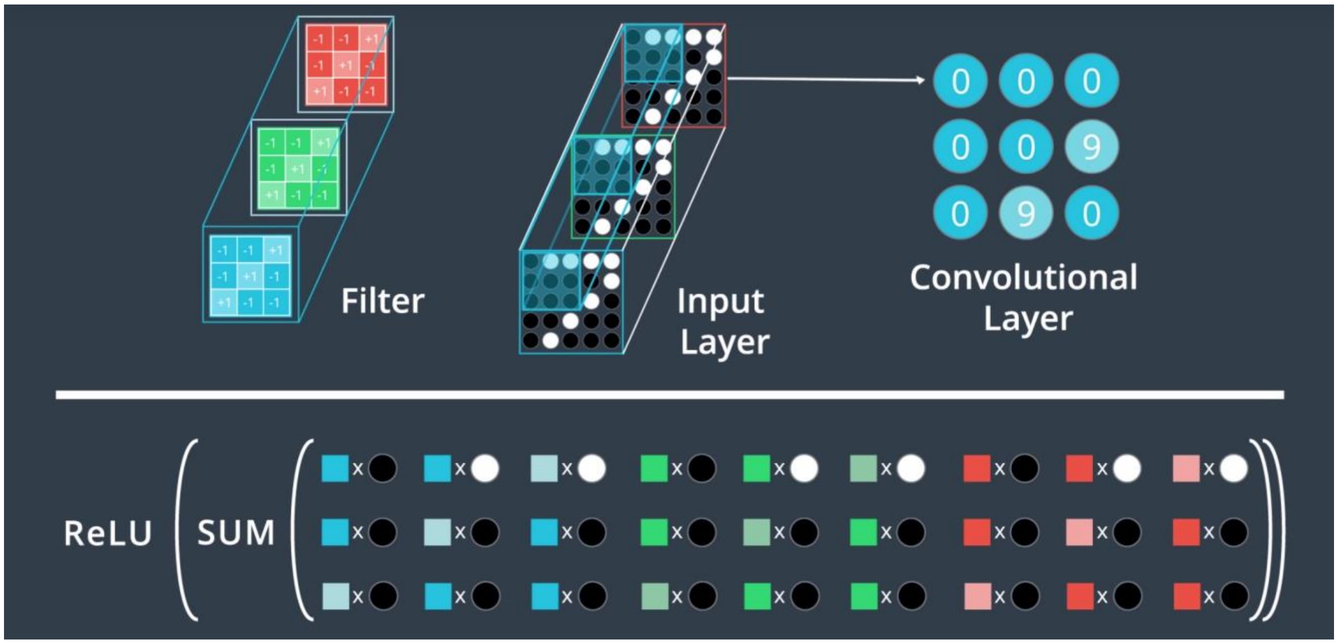
Color images are interpreted by the computer as a 3D array with height, width and depth. In the case of RGB images, the depth is three (tao: there are three numbers along the depth direction). This 3D array is best conceptualized as a stack of three two-dimensional matrices, where we have matrices corresponding to the red, green, and blue channels of the image.



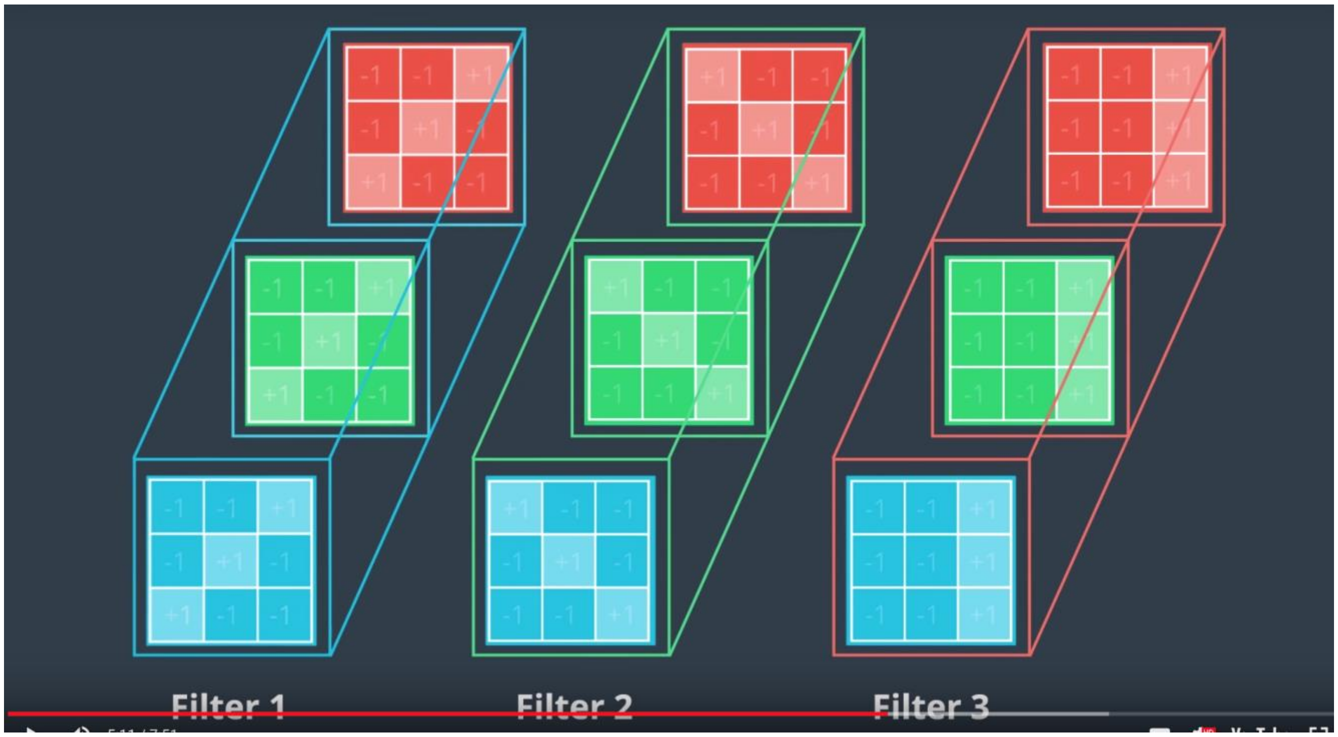
So how do we perform a convolution on a color image? As was the case with grayscale images, we still move a filter horizontally and vertically across the image.



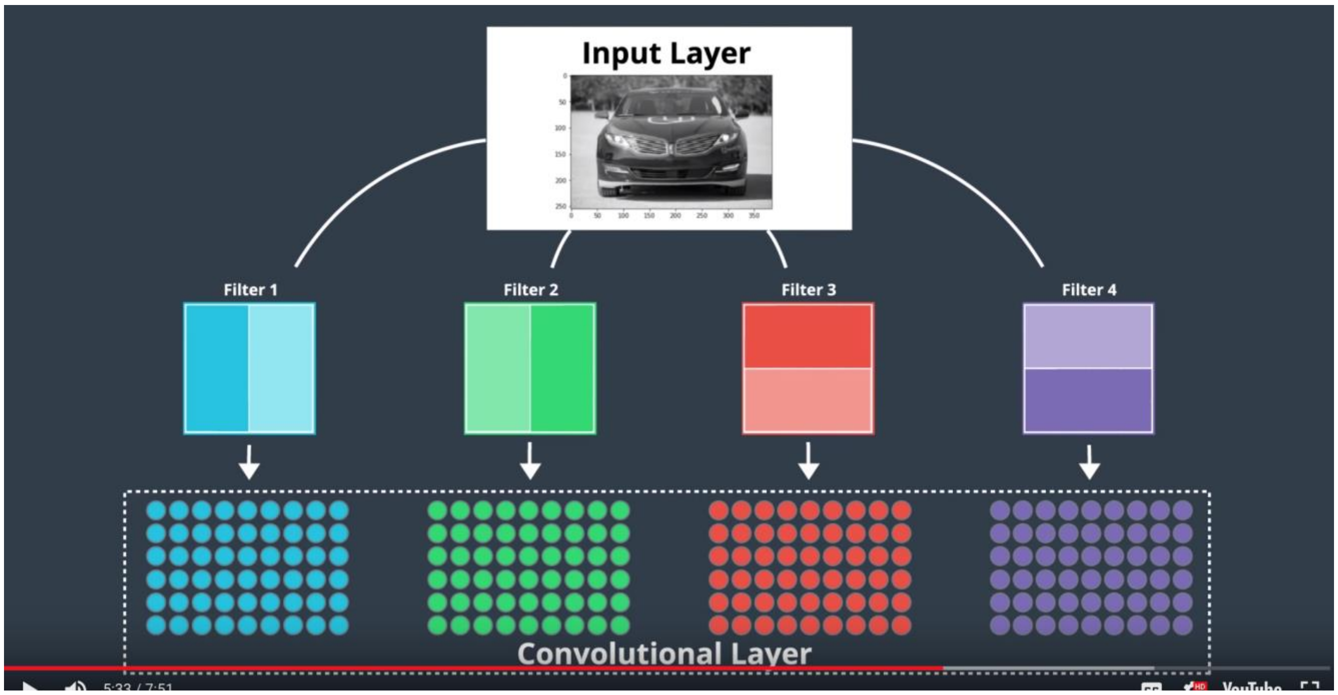
Only now the filter is itself three dimensional to have a value for each color channel at each horizontal and vertical location in the image array. Just as we think of the color image as a stack of three two-dimensional matrices, you can also think of the filter as a stack of three two-dimensional matrices. Both the color image and the filter have red, green, and blue channels.



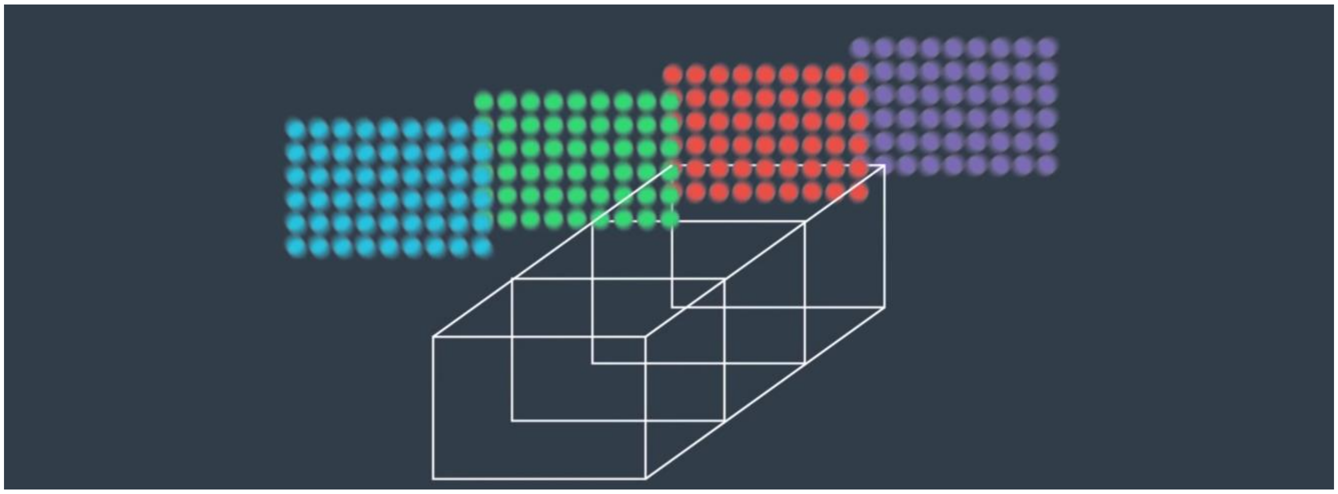
Now to obtain the values of the nodes in the feature map corresponding to this filter, we do pretty much the same thing we did before. Only now, our sum is over three times as many terms. We emphasize that here we've depicted the calculation of the value of a single node in a convolutional layer (tao: this sum is to get the value of a node in the convolutional layer, not the output layer) for one filter on a color image.

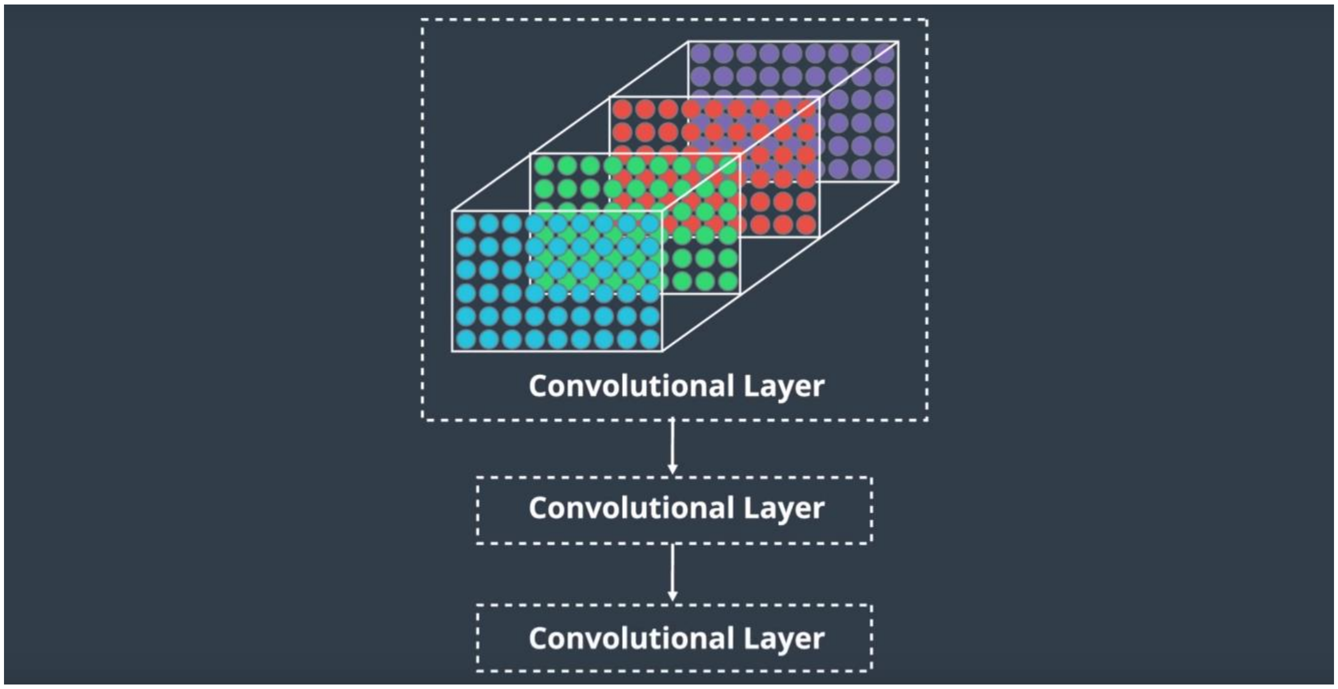


If we wanted to picture the case of a color image with multiple filters, instead of having a single 3D array, which corresponds to one filter, we would define multiple 3D arrays-- each defining a filter. Here we've depicted three filters, each is a 3D array that you can think of as a stack of three 2D arrays.

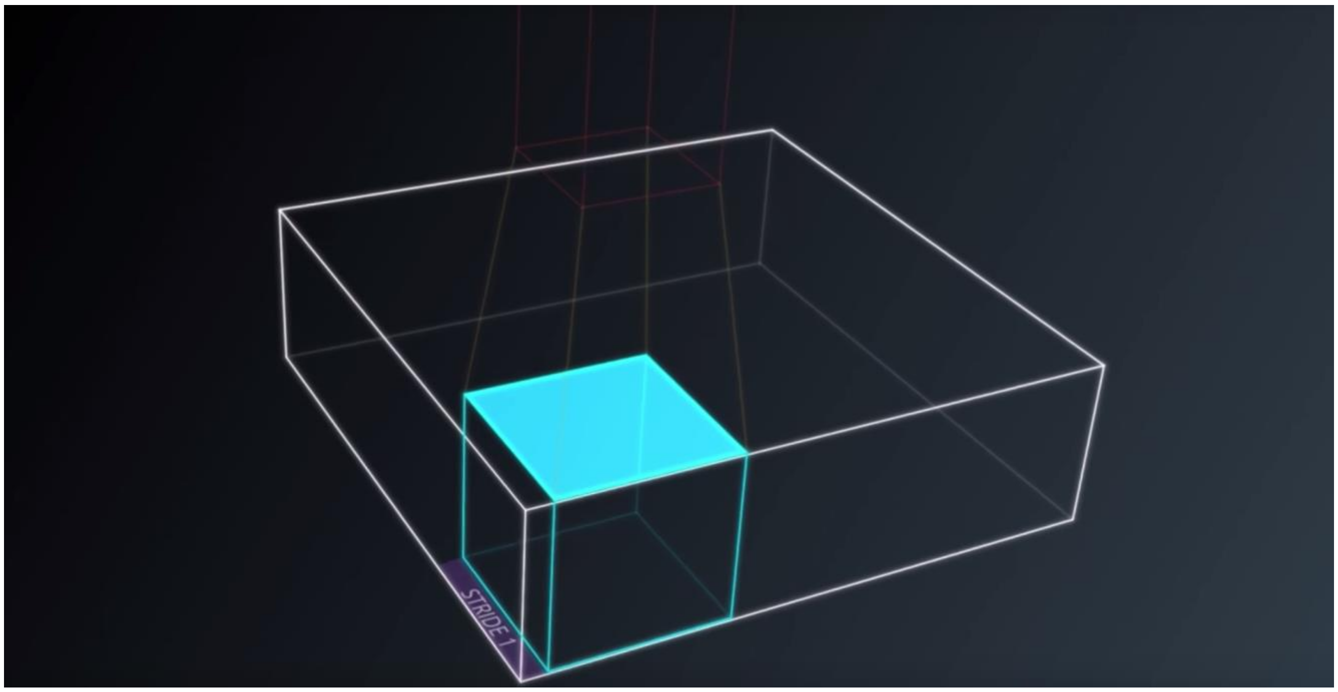


Here's where it starts to get really cool. You can think about each of the feature maps in a convolutional layer along the same lines as an image channel and stack them to get a 3D array.

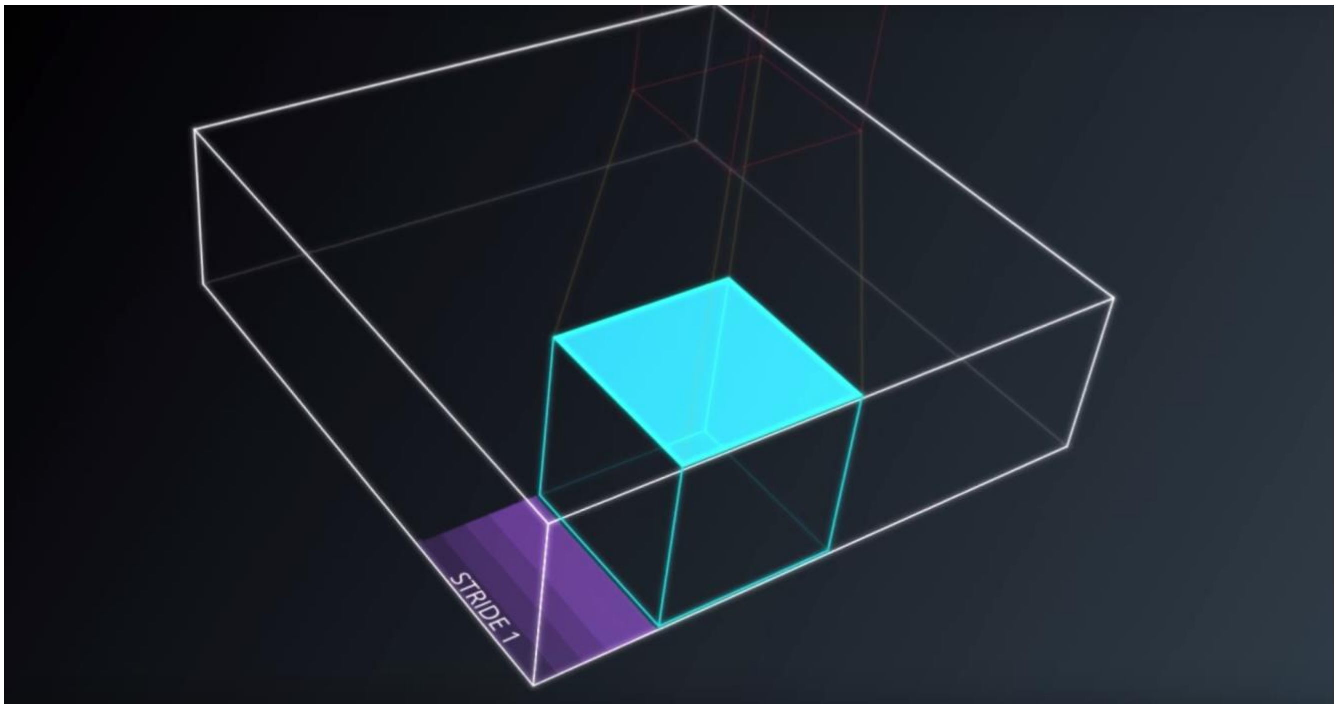




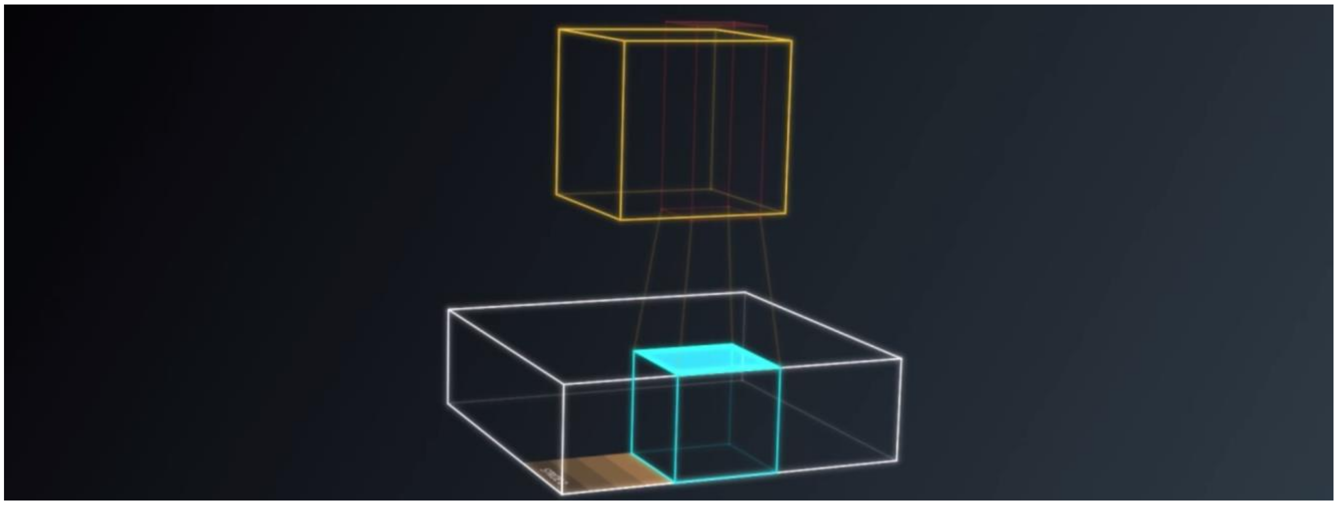
Then, we can use this 3D array as input to still another convolutional layer to discover patterns within the patterns that we discovered in the first convolutional layer. We can then do this again to discover patterns within patterns within patterns. Remember that in some sense convolutional layers aren't too different from the dense layers that you saw in the previous section. Dense layers are fully connected meaning that the nodes are connected to every node in the previous layer. Convolutional layers are locally connected where their nodes are connected to only a small subset of the previous layers' nodes. Convolutional layers also had this added perimeter sharing. But in both cases, with convolutional and dense layers, inference works the same way. Both have weights and biases that are initially randomly generated. So in the case of CNNs where the weights take the form of convolutional filters, those filters are randomly generated and so are the patterns that they're initially designed to detect. As with MLPs, when we construct to CNN we will always specify a loss function. In the case of multiclass classification, this will be categorical cross-entropy loss. Then as we train the model through back propagation, the filters are updated at each epic to take on values that minimize the loss function. In other words, the CNN determines what kind of patterns it needs to detect based on the loss function. We'll visualize these patterns later and see that for instance, if our dataset contains dogs, the CNN is able to, on its own, learn filters that look like dogs. So with CNNs to emphasize, we won't specify the values of the filters or tell the CNN what kind of patterns it needs to detect. These will be learned from the data.



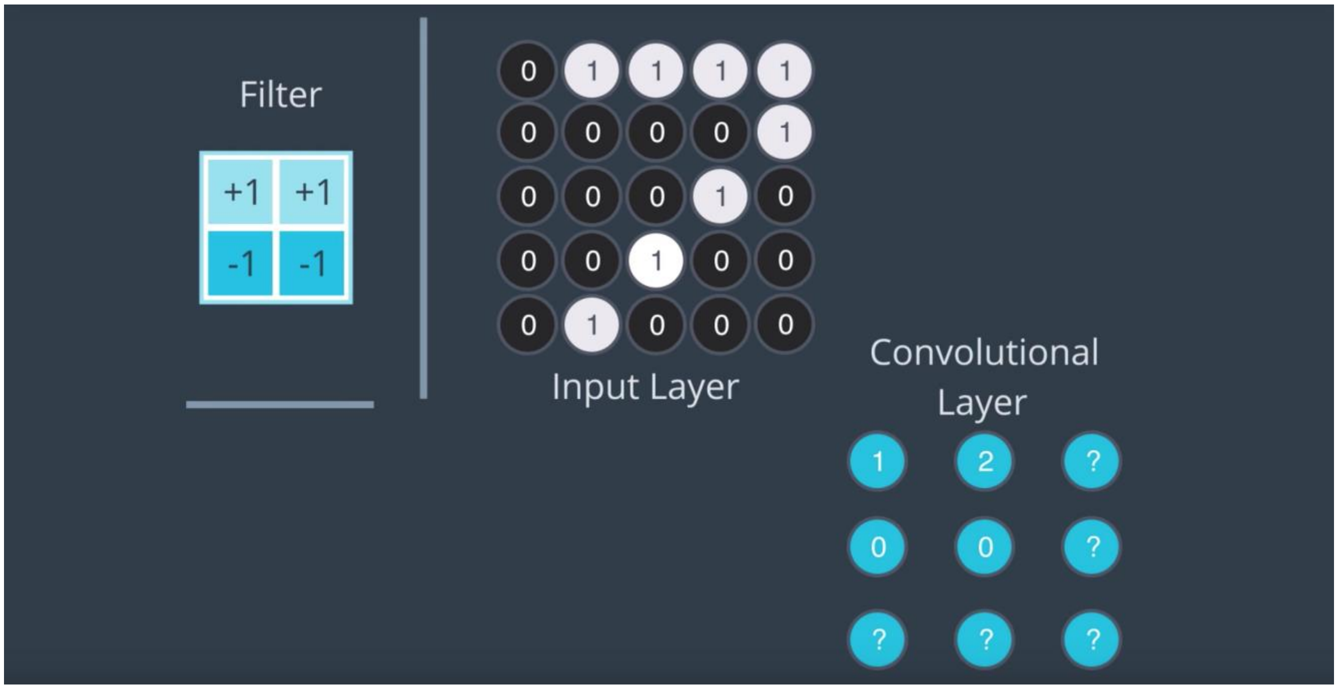
113. We've seen that you can control the behavior of a convolutional layer by specifying the number of filters and the size of each filter, for instance. To increase the number of nodes in a convolutional layer, you could increase the number of filters. To increase the size of the detected patterns, you could increase the size of your filter. But there are even more hyperparameters that you can do. One of these hyperparameters is referred to as the stride (大步走) of the convolution. The stride is just the amount by which the filter slides over the image.



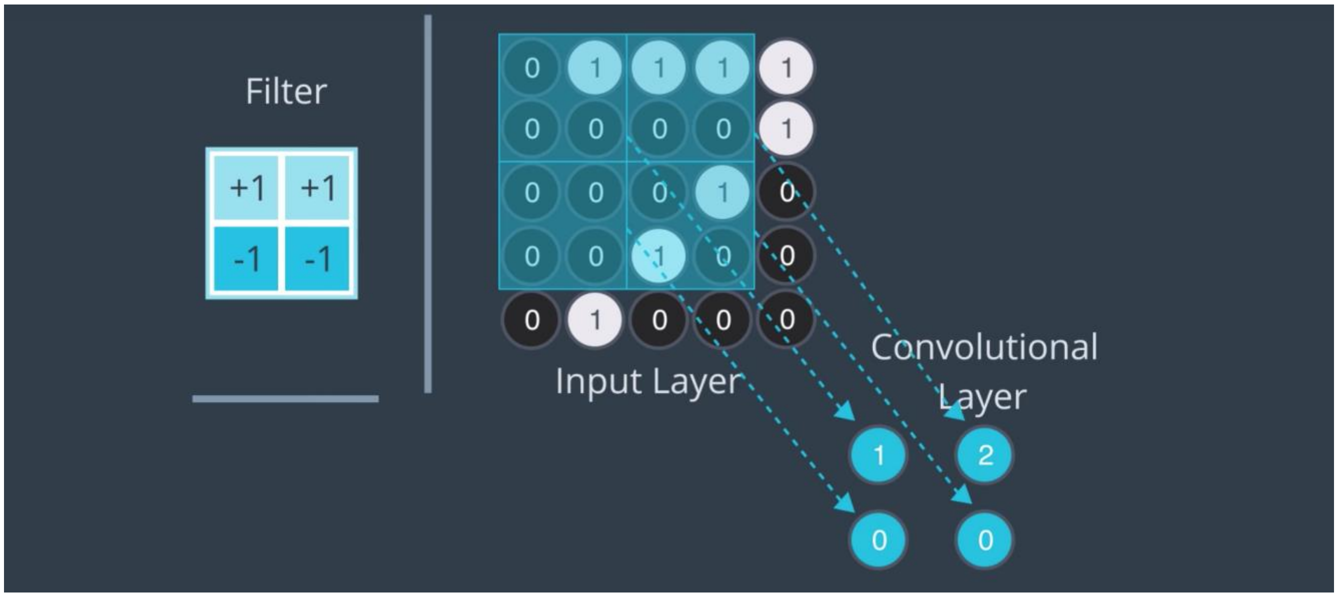
In the example in the previous video, the stride was one. We move the convolution window horizontally and vertically across the image one pixel at a time.



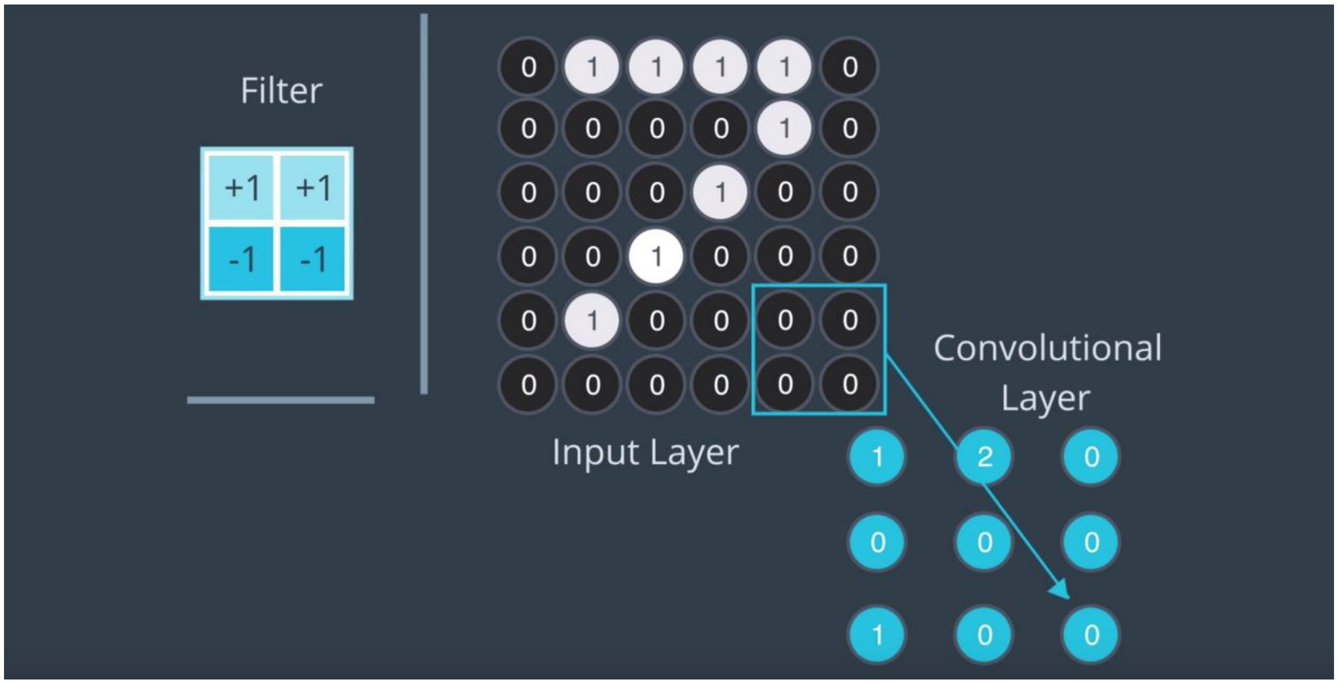
A stride of one makes the convolutional layer roughly the same width and height as the input image. In this animation, we've drawn the purple convolutional layer as stacked feature maps. If we instead make the stride equal to two, the convolutional layer is about half the width and height of the image (tao: same depth). I save roughly because it depends on what you do at the edge of your image.



To see how the treatment of the edges will matter, consider our toy example of a five-by-five grey scale image. Say, we have a different filter now with the height and width of two. Say, the stride is also two (tao: see picture below). Then, as before, we start with the filter in the top left corner of the image and calculate the value for the first node in the convolutional layer. We then move the filter two units to the right and do the same. But when we move the filter two more units to the right, the filter extends outside the image. What do we do now? Do we still want to keep the corresponding convolutional node? For now, let's just populate the places where the filter extends outside with a question mark and proceed as planned. So now, how do we deal with these nodes where the filter extended outside the image? We could as a first option, just get rid of them. Note that if we choose this option, it's possible that our convolutional layer has no information about some regions of the image.



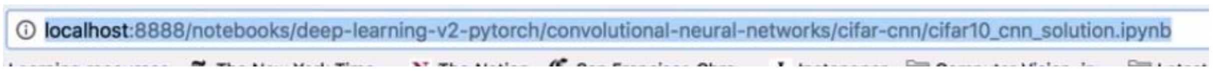
This is the case here for the right and bottom edges of the image.



As a second option, we could plan ahead for this case by padding the image with zeros to give the filter more space to move. Now, when we populate the convolutional layer, we get contributions from every region in the image.

Not_read_begin

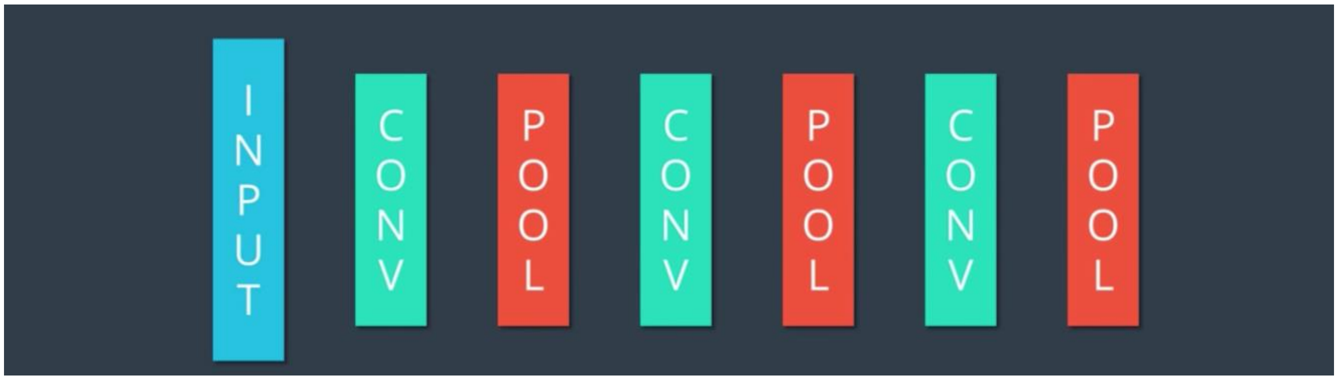
The following paragraph is about the following notebook:


localhost:8888/notebooks/deep-learning-v2-pytorch/convolutional-neural-networks/cifar-cnn/cifar10_cnn_solution.ipynb

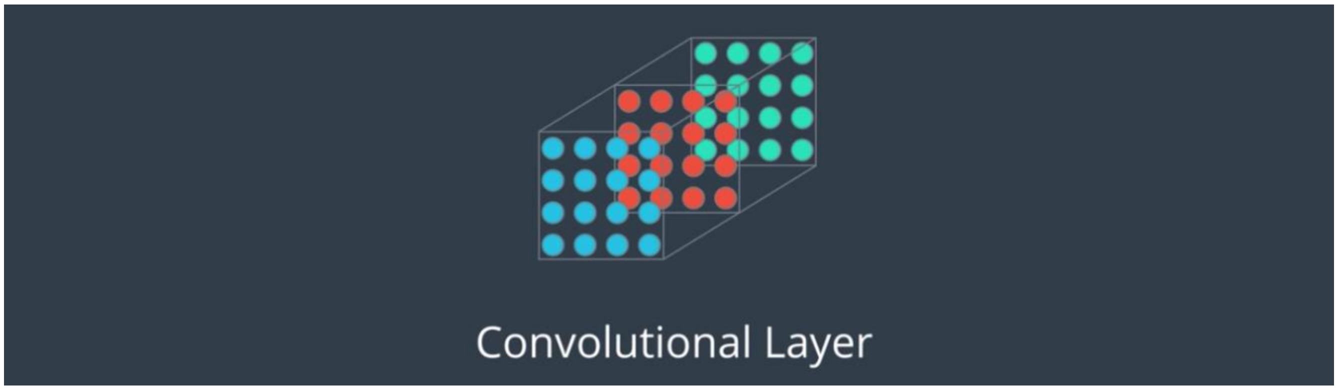
114. Okay. Now that you've tried to define and train a classifier of your own, I'll show you a solution that I created. After loading in my data and processing it, I defined a complete CNN. I've kept my initial convolutional layer, conv1, that sees our input image and outputs a stack of 16 feature maps. I'm defining all my convolutional layers first, defining two more. Each of which doubles the depth of the output until we get to a layer with a depth of 64. So, we're starting with an image depth of three, then moving to 16, then 32, and finally 64. Each of these layers uses a convolutional kernel of size three by three, and has a padding of one. You can also see that I've kept my one max pooling layer, which will down-sample any XY size by two. I've also included a dropout layer with a probability of 0.25 to prevent overfitting. Finally, I also have a couple of fully connected layers. This first layer will be responsible for taking as input my final downside stack of feature maps. I know that my original input image which is 32 by 32 by three is getting squished in the x and y dimension and stretched in the depth dimension as it moves through each convolutional and pooling layer. In the forward function, you can see that I apply a pooling layer after each convolutional layer. So, this image will reduce in size to 16 by 16, then eight by eight, and finally four by four after the last pooling layer. The third convolutional layer will have produced a depth of 64, and that's how I get these values. Four by four, for my final XY size, and 64 for my depth. That's the number of inputs that this first fully connected layer should see. Then I'm having that produced 500 outputs. These 500 outputs will feed as input into my final classification layer, which will see these as inputs and produce 10 class scores as outputs. So, let's see how all these layers are used in the forward function. First, I'm adding a sequence of convolutional and pooling layers in sequence, and passing our input image into our first convolutional layer, applying an activation function than a pooling layer. I'm doing the same thing for our second and third convolutional layers. Finally, this resultant x, I'm going to flatten into a vector shape. This will allow me to pass it as input into a fully connected layer. In between this flattening layer and each fully-connected layer, I'm adding a dropout layer to prevent overfitting. But then in passing my flattened image input x into my first fully connected layer. As with all my hidden layers, I'm applying a relu activation function. Finally, one more dropout layer and my last fully connected layer. The resultant x should be a list of 10 class scores. Finally, I instantiate this model and I move it to GPU. Below you can see that I've printed out each of the layers in my unit function to make sure they're as I expect. This shows me the number of inputs and outputs for each layer, the kernel size, the stride, and the padding, and they all checkout. So, just to summarize, for every convolutional layer that I defined, I apply a relu function and a max pooling layer right after that. After that series of layers, I flatten that representation and pass it to my fully-connected layer. Adding some dropout, I'm finally passing it so that it produces 10 class scores. So, I have my complete model, and then I'm moving onto training. I'm going to use a standard cross entropy loss, which is useful for classification tasks like this, and Stochastic Gradient Descent. Then I'm actually training the network, and I decided on training for 30 epochs. I decided on this value after watching the training and validation loss decrease over time. I can see that I might have stopped training even earlier around epoch 20, which is where the validation loss stops decreasing. I saved my model here, the one that got the best validation loss, and then I loaded and tested it out. I can see

when I test this, that I get an overall accuracy of about 70 percent. That's not bad, it's much better than guessing for example. If you've given this task in honest attempt, I can say congratulations on getting this far. You've really learned a lot about programming your own neural networks. Of course we can see that it does better on some classes than others and it's always interesting to think about why that might be the case. In general, it seems like my model does better on vehicles rather than animals. It's probably because animals really vary in color and size, and so I might be able to improve this model if I had more images in that particular data set. It may also help to add another convolutional layer and see if I could suss out more complex patterns in these images. Here I'm just further visualizing which images it gets right or wrong. There's plenty of room to tinker with and optimize the CNN further, and I encourage you to do this, it's a great learning experience. I should mention that in 2015, there was an online competition where data scientists competed to classify images in this database. The winning architecture was a CNN, and it achieved over 95 percent test accuracy. It took about 90 hours to train on a GPU, which I do not encourage in this classroom. But it does demonstrate that good accuracy is not a trivial task.

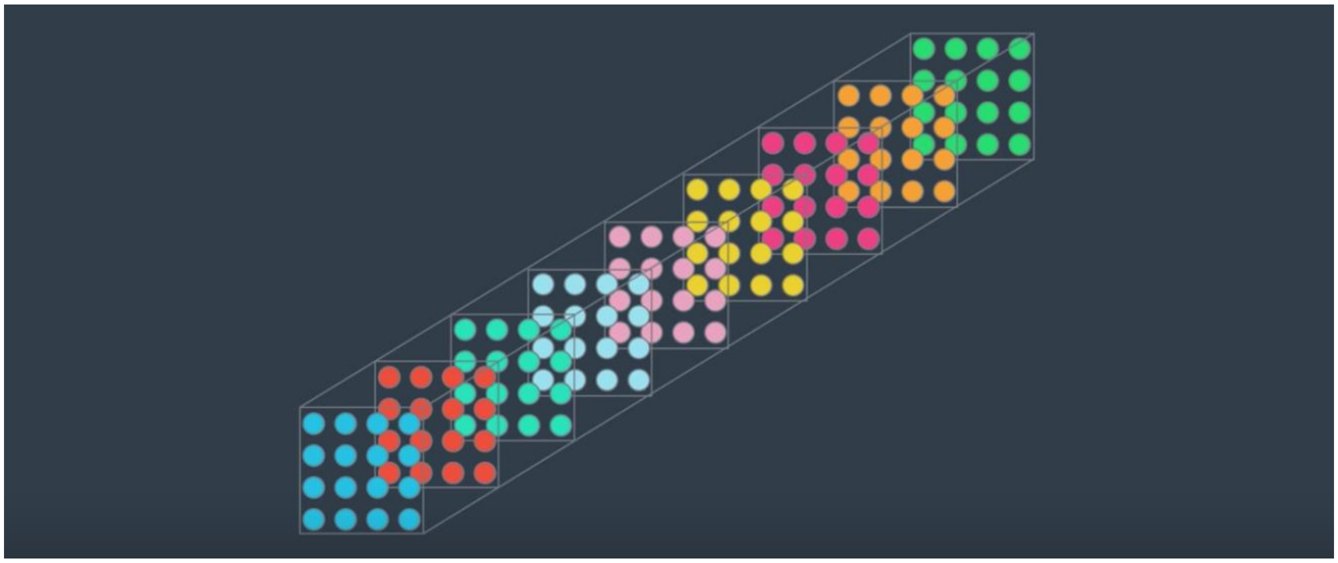
Not_read_end



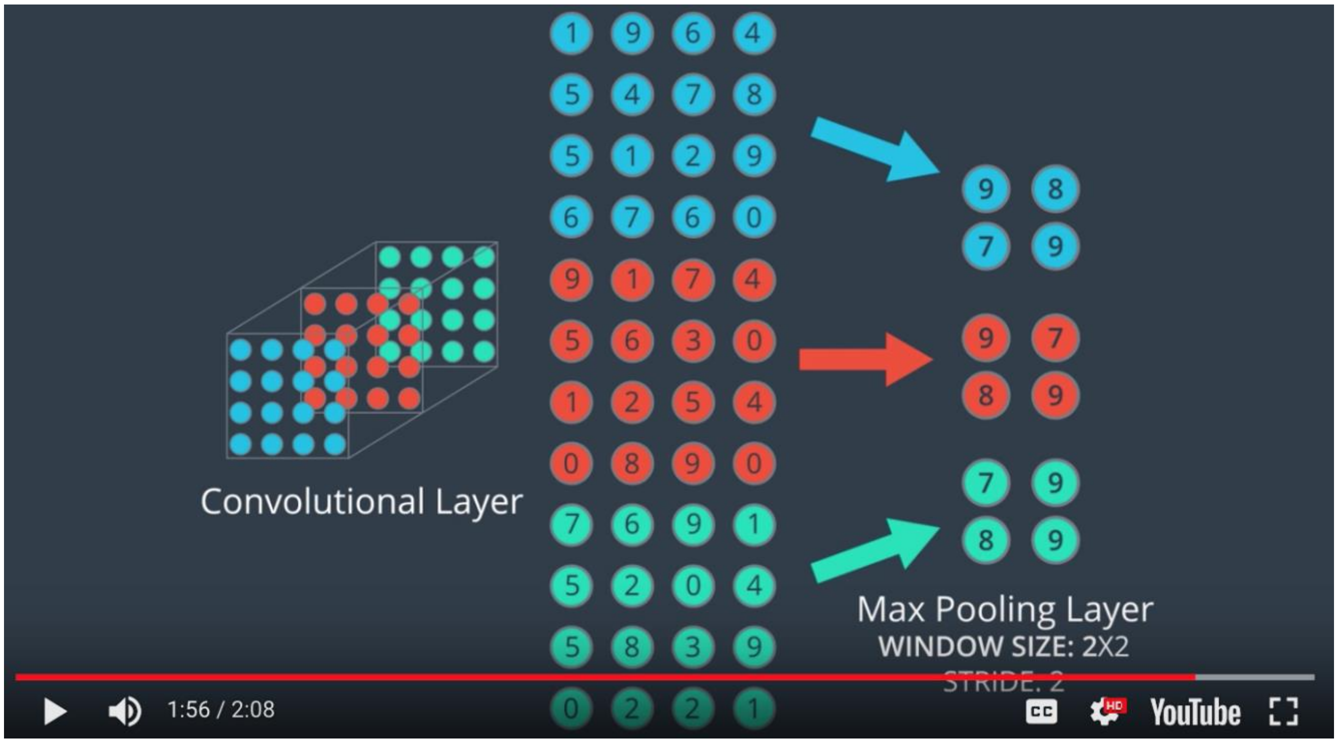
115. We're now ready to introduce you to the second and final type of layer that we'll need to introduce before building our own convolutional neural networks. These so-called pooling layers often take convolutional layers as input.



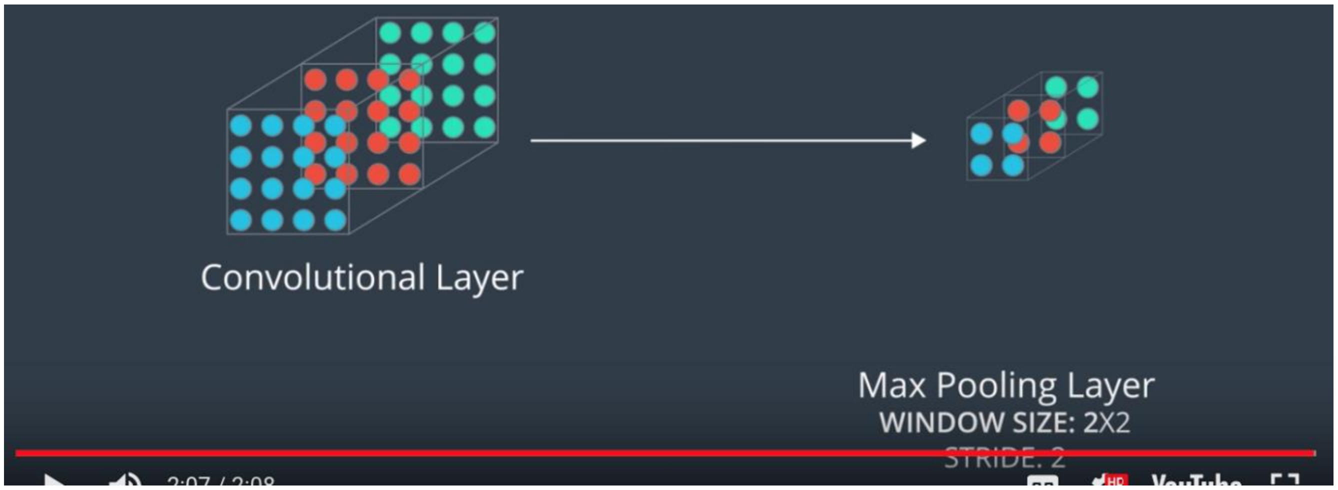
Recall that a convolutional layer is a stack of feature maps where we have one feature map for each filter.



A complicated dataset with many different object categories will require a large number of filters, each responsible for finding a pattern in the image. More filters means a bigger stack, which means that the dimensionality of our convolutional layers can get quite large. Higher dimensionality means, we'll need to use more parameters, which can lead to over-fitting. Thus, we need a method for reducing this dimensionality. This is the role of pooling layers within a convolutional neural network. We'll focus on two different types of pooling layers.



The first type is a max pooling layer, max pooling layers will take a stack of feature maps as input. Here, we've enlarged and visualized all three of the feature maps. As with convolutional layers, we'll define a window size and stride (for calculating the pooling layers, tao). In this case, we'll use a window size of two and a stride of two. To construct the max pooling layer, we'll work with each feature map separately. Let's begin with the first feature map, we start with our window in the top left corner of the image. The value of the corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window. In this case, we had a one, nine, five, and four in our window, so nine was the maximum. If we continue this process and do it for all of our feature maps, the output is a stack with the same number of feature maps,



but each feature map has been reduced in width and height. In this case, the width and height are half of that of the previous convolutional layer.

Alternatives to Pooling

It's important to note that pooling operations do throw away some image information. That is, they discard pixel information in order to get a smaller, feature-level representation of an image. This works quite well in tasks like image classification, but it can cause some issues.

Consider the case of facial recognition. When you think of how you identify a face, you might think about noticing features; two eyes, a nose, and a mouth, for example. And those pieces, together, form a complete face! A typical CNN that is trained to do facial recognition, should also learn to identify these features. Only, by distilling an image into a feature-level representation, you might get a weird result:

Given an image of a face that has been photoshopped to include three eyes or a nose placed above the eyes, a feature-level representation will identify these features and still recognize a face! Even though that face is fake/contains too many features in an atypical orientation.
So, there has been research into classification methods that do not discard spatial information (as in the pooling layers), and instead learn to spatial relationships between parts (like between eyes, nose, and mouth).

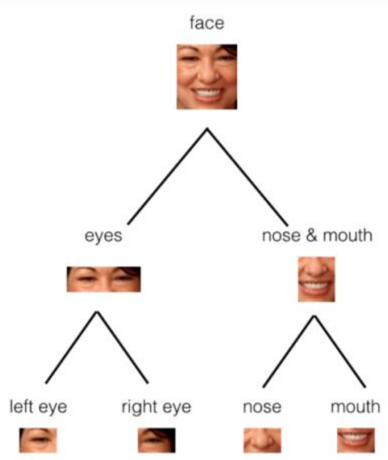
One such method, for learning spatial relationships between parts, is the capsule network.

Capsule Networks

Capsule Networks provide a way to detect parts of objects in an image and represent spatial relationships between those parts. This means that capsule networks are able to recognize the same object, like a face, in a variety of different poses and with the typical number of features (eyes, nose , mouth) even if they have not seen that pose in training data.

Capsule networks are made of parent and child nodes that build up a complete picture of an object.

Parts of a face, making up a whole image.



In the example above, you can see how the parts of a face (eyes, nose, mouth, etc.) might be recognized in leaf nodes and then combined to form a more complete face part in parent nodes.

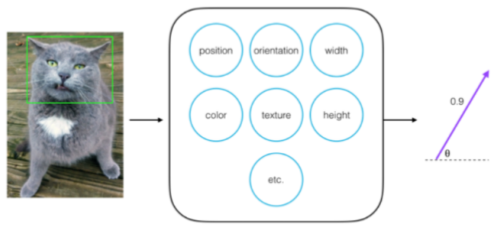
What are Capsules?

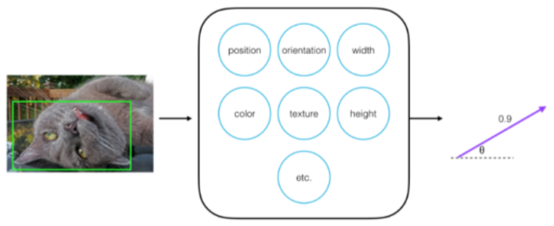
Capsules are essentially a collection of nodes, each of which contains information about a specific part; part properties like width, orientation, color, and so on. The important thing to note is that each capsule outputs a vector with some magnitude and orientation.

Magnitude (m) = the probability that a part exists; a value between 0 and 1.
Orientation (theta) = the state of the part properties.

These output vectors allow us to do some powerful routing math to build up a parse tree that recognizes whole objects as comprised of several, smaller parts!

The magnitude is a special part property that should stay very high even when an object is in a different orientation, as shown below.





Cat face, recognized in a multiple orientations, co: this blog post.

Resources

You can learn more about capsules, in this blog post.
And experiment with an implementation of a capsule network in PyTorch, at this github repo.
Supporting Materials
Dynamic routing between capsules, hinton et al