Course 4: Convolutional Neural Networks

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/convolutional-neural-networks/home/welcome

****
Course 4: Convolutional Neural Networks
****
In this course we'll learn how to build convolutional neural networks and apply it to image data. Thanks to deep learning, computer vision is working far better than just two years ago, and this is enabling numerous exciting applications ranging from safe autonomous driving, to accurate face recognition, to automatic reading of radiology images.

You will:
- Understand how to build a convolutional neural network, including recent variations such as residual networks.
- Know how to apply convolutional networks to visual detection and recognition tasks.
- Know to use neural style transfer to generate art.
- Be able to apply these algorithms to a variety of image, video, and other 2D or 3D data.

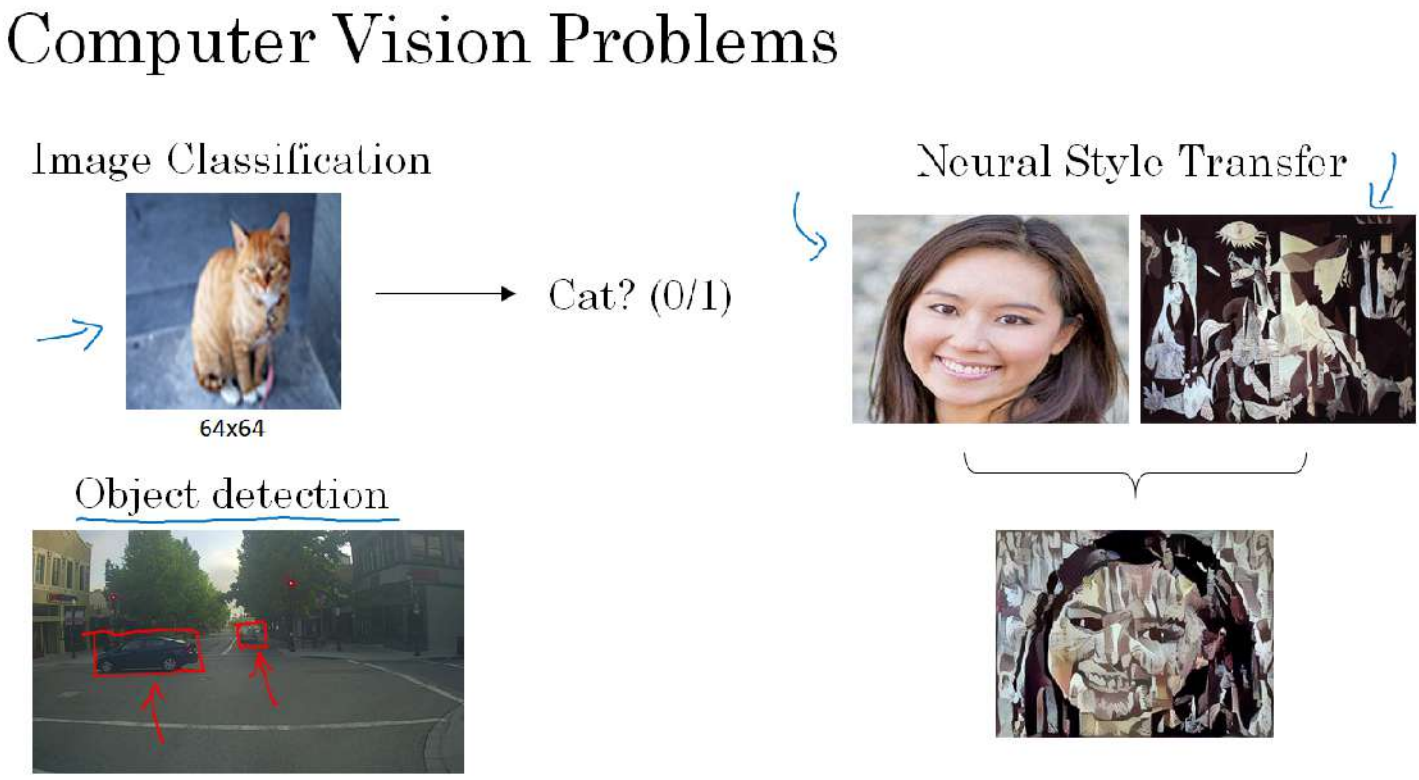
Week 1: Foundations of Convolutional Neural Networks
Learn to implement the foundation layers of CNN's (pooling, convolutions) and to stack them properly in a deep network to solve multi-class image classification problems.

Learning Objectives

- Understand the convolution operation
- Understand the pooling operation
- Remember the vocabulary used in convolutional neural network (padding, stride, filter, ...)
- Build a convolutional neural network for image multi-class classification

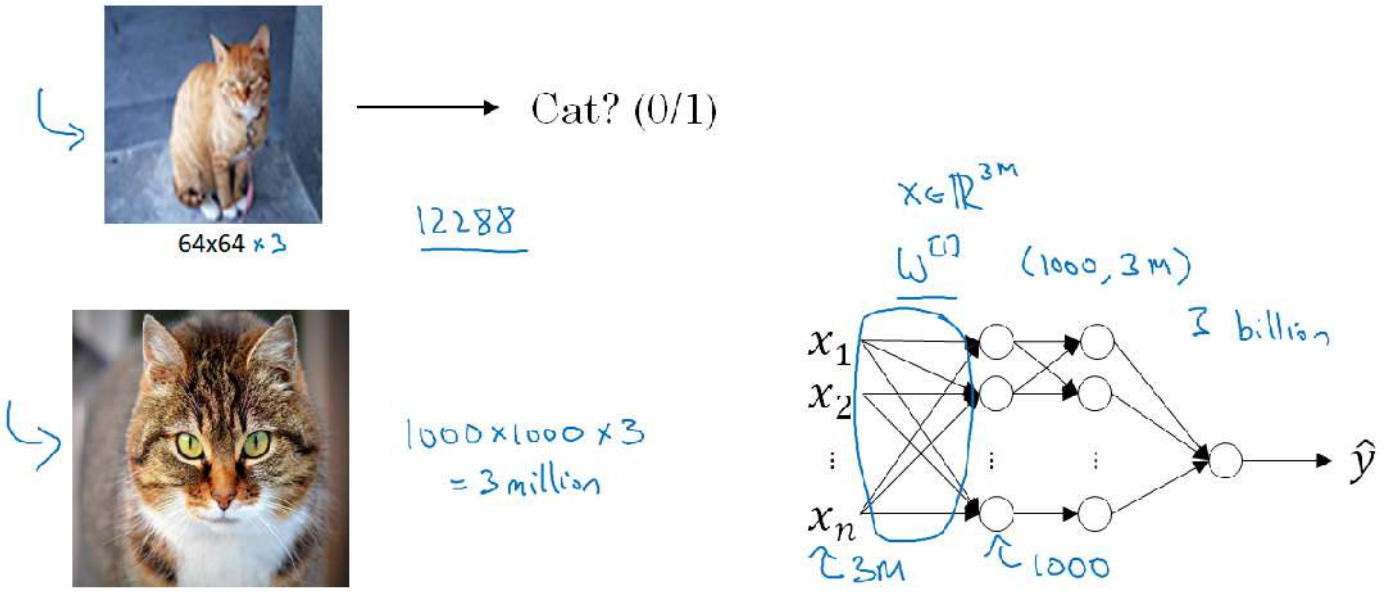
Convolutional Neural Networks
Computer Vision
Computer vision is one of the areas that's been advancing rapidly thanks to deep learning. Deep learning computer vision is now helping self-driving cars figure out where the other cars and pedestrians around so as to avoid them. Is making face recognition work much better than ever before, so that perhaps some of you will soon, or perhaps already, be able to unlock a phone, unlock even a door using just your face and if you look on your cell phone, I bet you have many apps that show you pictures of food, or pictures of a hotel, or just fun pictures of scenery. And some of the companies that build those apps are using deep learning to help show you the most attractive, the most beautiful, or the most relevant pictures and I think deep learning is even enabling new types of art to be created. So, I think the two reasons I'm excited about deep learning for computer vision and why I think you might be too. First, rapid advances in computer vision are enabling brand new applications to view, though they just were impossible a few years ago and by learning these tools, perhaps you will be able to invent some of these new products and applications. Second, even if you don't end up building computer vision systems per se, I found that because the computer vision research community has been so creative and so inventive in coming up with new neural network architectures and algorithms, is actually inspire that creates a lot cross-fertilization into other areas as well. For example, when I was working on speech recognition, I sometimes actually took inspiration from ideas from computer vision and borrowed them into the speech literature. So, even if you don't end up working on computer vision, I hope that you find some of the ideas you learn about in this course hopeful for some of your algorithms and your architectures. So with that, let's get started. Here are some examples of computer vision problems we'll study in this course. You've already seen image classifications, sometimes also

called image recognition, where you might take as input say a 64x64 image and try to figure out, is that a cat? Another example of the computer vision problem is object detection. So, if you're building a self-driving car, maybe you don't just need to figure out that there are other cars in this image. But instead, you need to figure out the position of the other cars in this picture, so that your car can avoid them. In object detection, usually, we have to not just figure out that these other objects say cars and picture, but also draw boxes around them. We have some other way of recognizing where in the picture are these objects. And notice also, in this example, that they can be multiple cars in the same picture, or at least every one of them within a certain distance of your car. Here's another example, maybe a more fun one is neural style transfer. Let's say you have a picture, and you want this picture repainted in a different style. So neural style transfer, you have a content image, and you have a style image. The image on the right is actually a Picasso. And you can have a neural network put them together to repaint the content image (that is the image on the left in diagram below), but in the style of the image on the right, and you end up with the image at the bottom. So, algorithms like these are enabling new types of artwork to be created.



One of the challenges of computer vision problems is that the inputs can get really big. For example, in previous section, we've worked with 64x64 images and so that's 64x64x3 because there are three color channels (RGB) and if you multiply that out, that's 12288. So **x the input features has dimension 12288** and that's not too bad but 64x64 is actually a very small image. If you work with larger images, maybe this is a 1000 pixel by 1000 pixel image, and that's actually just one megapixel but the dimension of the input features will be **1000x1000x3**, because you have three RGB channels, and that's three million. See below the example.

Deep Learning on large images



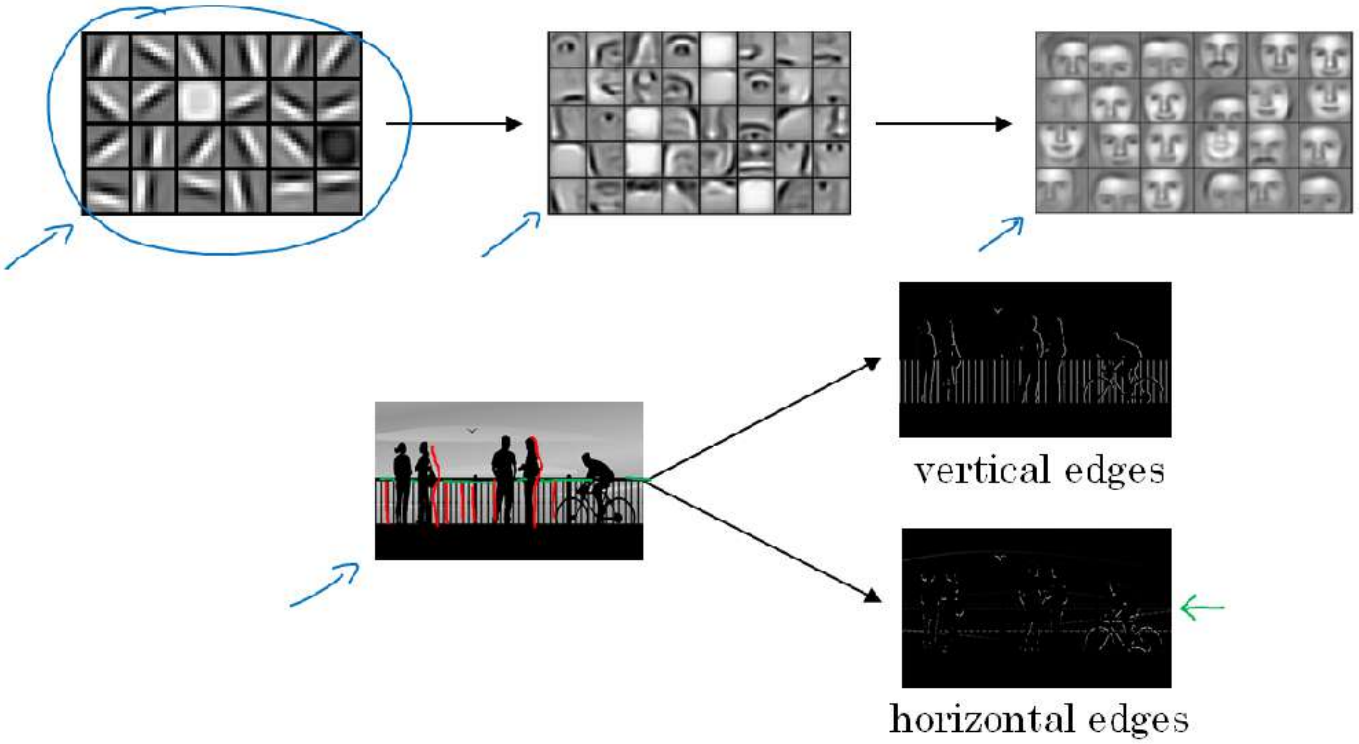
So, if you have three million input features, then this means that X here will be three million dimensional and so, if in the first hidden layer maybe you have just a 1000 hidden units, then the total number of weights that is the matrix W1, if you use a standard or fully connected network like we have seen before. This matrix will be a 1000x3 million dimensional matrix and this means that this matrix here will have three billion parameters which is just very, very large. and with that many parameters, it's difficult to get enough data to prevent a neural network from overfitting and also, the computational requirements and the memory requirements to train a neural network with three billion parameters is just a bit infeasible but for computer vision applications, you don't want to be stuck using only tiny little images. You want to use large images. To do that, you need to better implement the convolution operation, which is one of the fundamental building blocks of convolutional neural networks.

Edge Detection Example

The convolution operation is one of the fundamental building blocks of a convolutional neural network. Using edge detection as the motivating example in this section, you will see how the convolution operation works. In previous section, we've talked about how the **early layers of the neural network might detect edges and then the some later layers might detect cause of objects and then even later layers may detect cause of complete objects like people's faces in this case**. In this section, we'll see how we can detect edges in an image. Lets take an example.

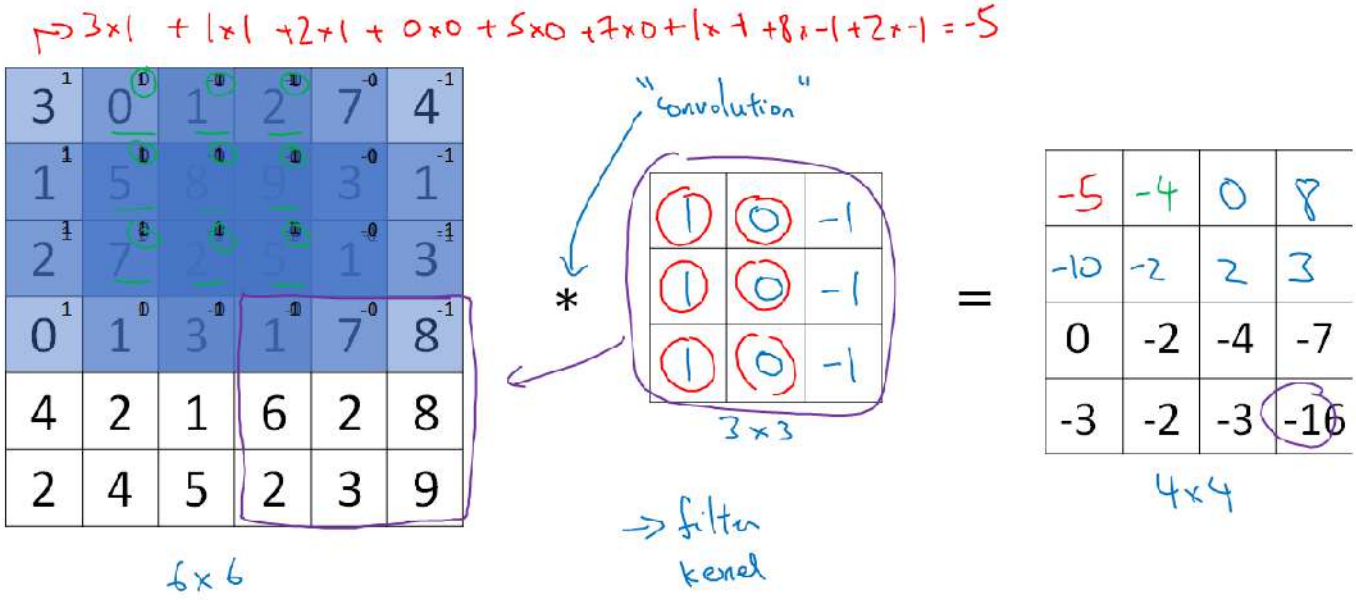
Given a picture like that for a computer to figure out what are the objects in this picture, the first thing you might do is maybe detect vertical edges in this image. For example, this image has all those vertical lines, where the buildings are, as well as kind of vertical lines idea all lines of these pedestrians and so those get detected in this vertical edge detector output. And you might also want to detect horizontal edges so for example, there is a very strong horizontal line where this railing is and that also gets detected sort of roughly here. How do you detect edges in image like this?

Computer Vision Problem



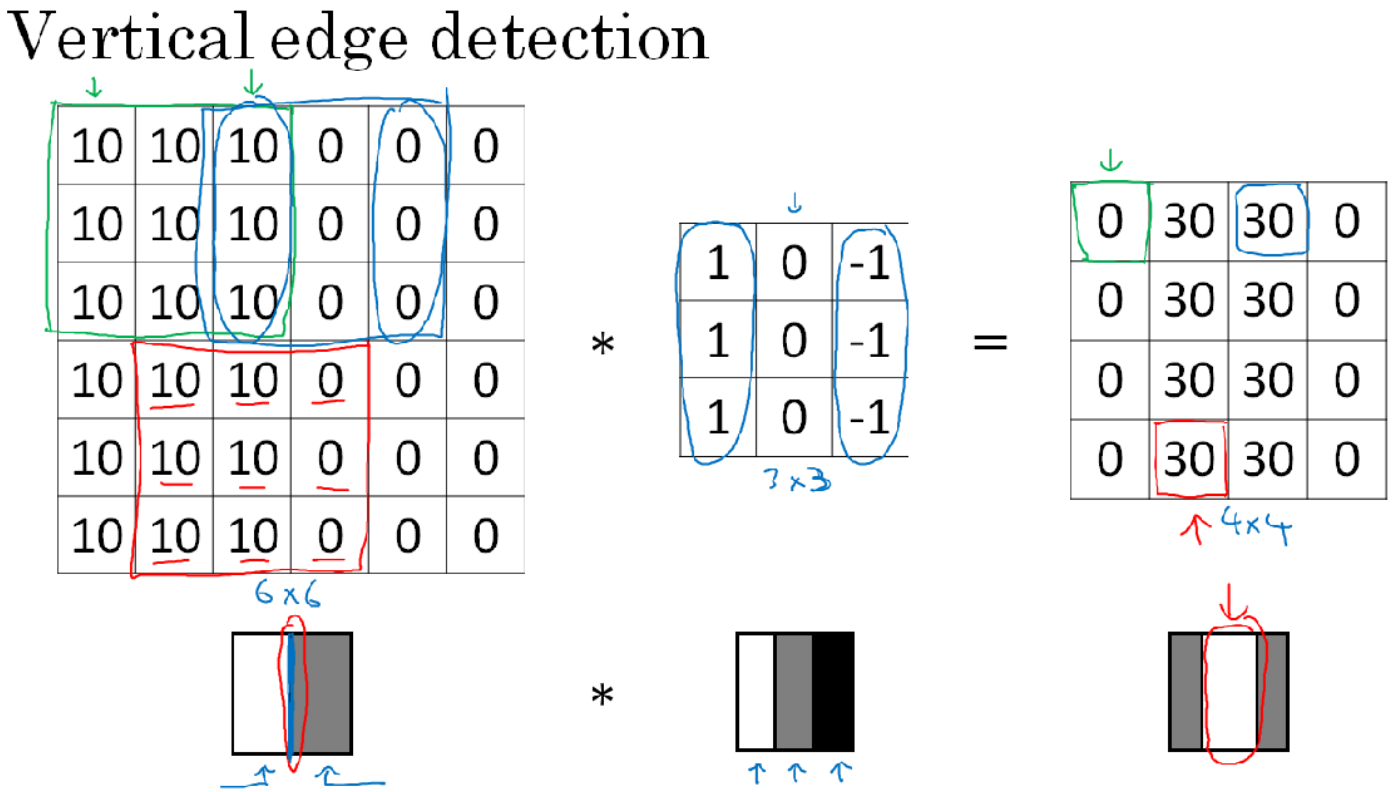
Let us look with an example. Here is a 6x6 grayscale image and because this is a grayscale image, this is just a 6x6x1 matrix rather than 6x6x3 because they are on a separate rgb channels. In order to detect edges or lets say vertical edges in his image, what you can do is construct a 3x3 matrix and in the terminology of convolutional neural networks, this is going to be called a **filter** and we're going to construct a 3x3 filter or 3x3 matrix and what you are going to do is take the 6x6 image and convolve it and the convolution operation is denoted by this asterisk and convolve it with the 3 x3 filter. The output of this convolution operator will be a 4x4 matrix, which you can interpret, which you can think of as a 4x4 image. The way you compute this 4 x4 output is shown in fig below.

Vertical edge detection



Looking on to the diagram to compute the first elements, the upper left element of this 4x4 matrix, what you are going to do is take the 3x3 filter and paste it on top of the 3x3 region of your original input image and what you should do is take the element wise product (check the calculation method at the top of diagram). Similarly we can calculate elements of 4x4 image. These are really just matrices of various dimensions. But the matrix on the left is convenient to interpret as image, and the one in the middle we interpret as a filter and the one on the right, you can interpret that as maybe another image. And this turns out to be a vertical edge detector.

Why is this doing vertical edge detection? Lets look at another example. To illustrate this, we are going to use a simplified image. Here is a simple 6x6 image where the left half of the image is 10 and the right half is zero. If you plot this as a picture, it might look like this, where the left half, the 10s, give you brighter pixel intensive values and the right half gives you darker pixel intensive values. I am using that shade of gray to denote zeros, although maybe it could also be drawn as black. But in this image, there is clearly a very strong vertical edge right down the middle of this image as it transitions from white to black or white to darker color. When you convolve this with the 3x3 filter and so this 3x3 filter can be visualized as follows, where is lighter, brighter pixels on the left and then this mid tone zeroes in the middle and then darker on the right. What you get is this matrix on the right.

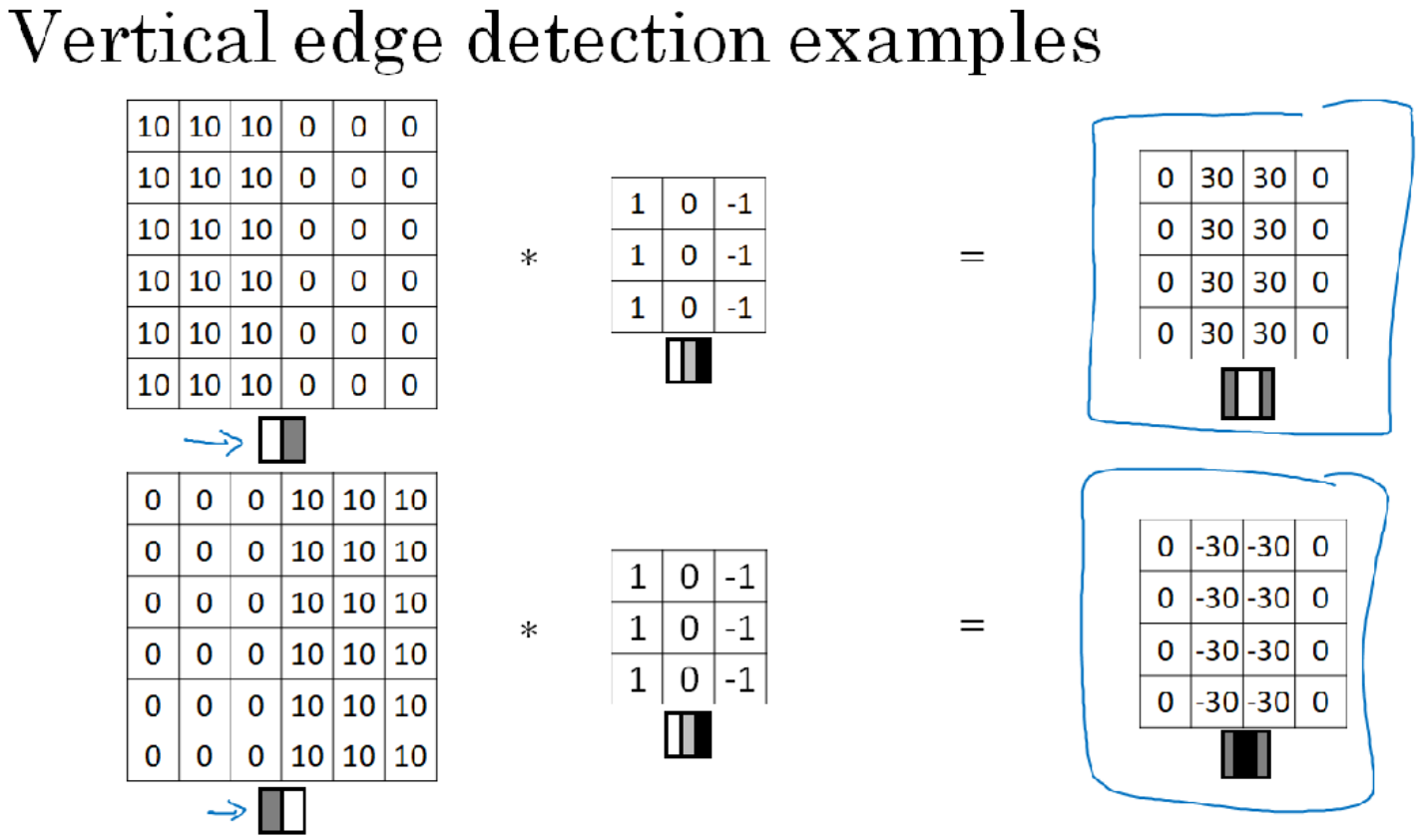


Now, if you plot this right most matrix's image it will look like that where there is this lighter region right in the middle and that corresponds to this having detected this vertical edge down the middle of your 6x6 image. In case the dimensions here seem a little bit wrong that the detected edge seems really thick, that's only because we are working with very small images in this example. And if you are using, say a 1000x1000 image rather than a 6x6 image then you find that this does a pretty good job, really detecting the vertical edges in your image. In this example, this bright region in the middle is just the output images way of saying that it looks like there is a strong vertical edge right down the middle of the image. Maybe one intuition to take away from vertical edge detection is that a vertical edge is a 3x3 region since we are using a 3x3 filter where there are bright pixels on the left, you do not care that much what is in the middle and dark pixels on the right. The middle in this 6x6 image is really where there could be bright pixels on the left and darker pixels on the right and that is why it thinks its a vertical edge over there. The

convolution operation gives you a convenient way to specify how to find these vertical edges in an image. You have now seen how the convolution operator works.

More Edge Detection

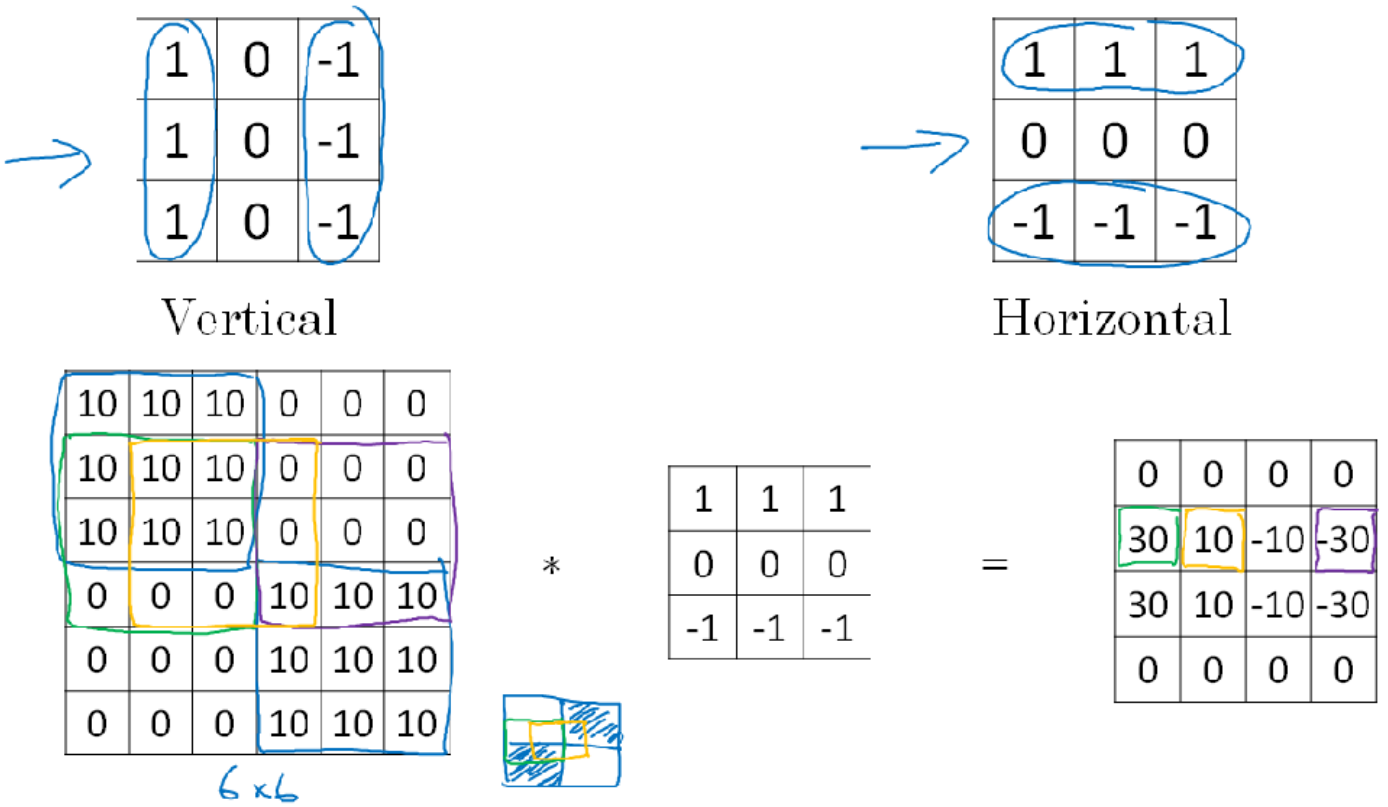
In this section, we'll learn the difference between positive and negative edges, that is, the difference between light to dark versus dark to light edge transitions and we'll also see other types of edge detectors, as well as how to have an algorithm learn, rather than have us hand code an edge detector as we've been doing so far. So let's get started. Here's the example you saw from the previous section, where you have a image, 6x6, there's light on the left and dark on the right, and convolving it with the vertical edge detection filter results in detecting the vertical edge down the middle of the image.



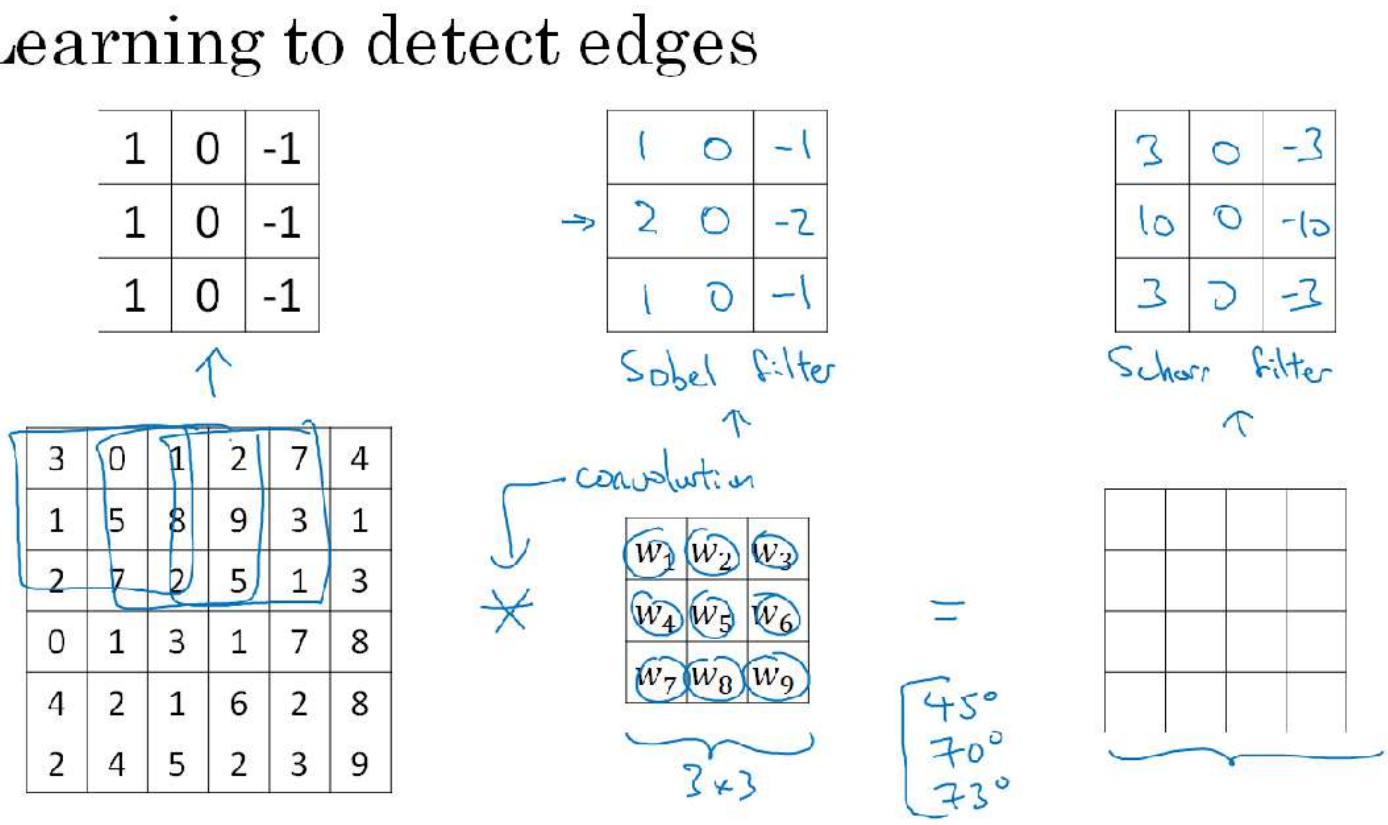
What happens in an image where the colors are flipped, where it is darker on the left and brighter on the right? So the 10s are now on the right half of the image and the 0s on the left. If you convolve it with the same edge detection filter, you end up with negative 30s, instead of 30 down the middle, and you can plot that as a picture that maybe looks like that. So because the shade of the transitions is reversed, the 30s now gets reversed as well and the negative 30s shows that this is a dark to light rather than a light to dark transition and if you don't care which of these two cases it is, you could take absolute values of this output matrix but this particular filter does make a difference between the light to dark versus the dark to light edges.

Let's see some more examples of edge detection. This 3x3 filter we've seen allows you to detect vertical edges. So maybe it should not surprise you too much that this 3x3 filter will allow you to detect horizontal edges. So as a reminder, a vertical edge according to this filter, is a 3x3 region where the pixels are relatively bright on the left part and relatively dark on the right part. So similarly, a horizontal edge would be a 3x3 region where the pixels are relatively bright on top and relatively dark in the bottom row. So here's one example, this is a more complex one, where you have here 10s in the upper left and lower right-hand corners. So if you draw this as an image, this would be an image which is going to be darker where there are 0s, so I'm going to shade in the darker regions, and then lighter in the upper left and lower right-hand corners. And if you convolve this with a horizontal edge detector, you end up with this.

Vertical and Horizontal Edge Detection



So in summary, different filters allow you to find vertical and horizontal edges. It turns out that the 3x3 vertical edge detection filter we've used is just one possible choice and historically, in the computer vision literature, there was a fair amount of debate about what is the best set of numbers to use. So here's something else you could use, which is maybe 1, 2, 1, 0, 0, 0, -1, -2, -1. This is called a Sobel filter. And the advantage of this is it puts a little bit more weight to the central row, the central pixel, and this makes it maybe a little bit more robust. But computer vision researchers will use other sets of numbers as well, like maybe instead of a 1, 2, 1, it should be a 3, 10, 3, right? And then -3, -10, -3. And this is called a Scharr filter. And this has yet other slightly different properties. And this is just for vertical edge detection. And if you flip it 90 degrees, you get horizontal edge detection. And with the rise of deep learning, one of the things we learned is that when you really want to detect edges in some complicated image, maybe you don't need to have computer vision researchers handpick these nine numbers. Maybe you can just learn them and treat the nine numbers of this matrix as parameters, which you can then learn using back propagation. And the goal is to learn nine parameters so that when you take the image, the six by six image, and convolve it with your three by three filter, that this gives you a good edge detector.

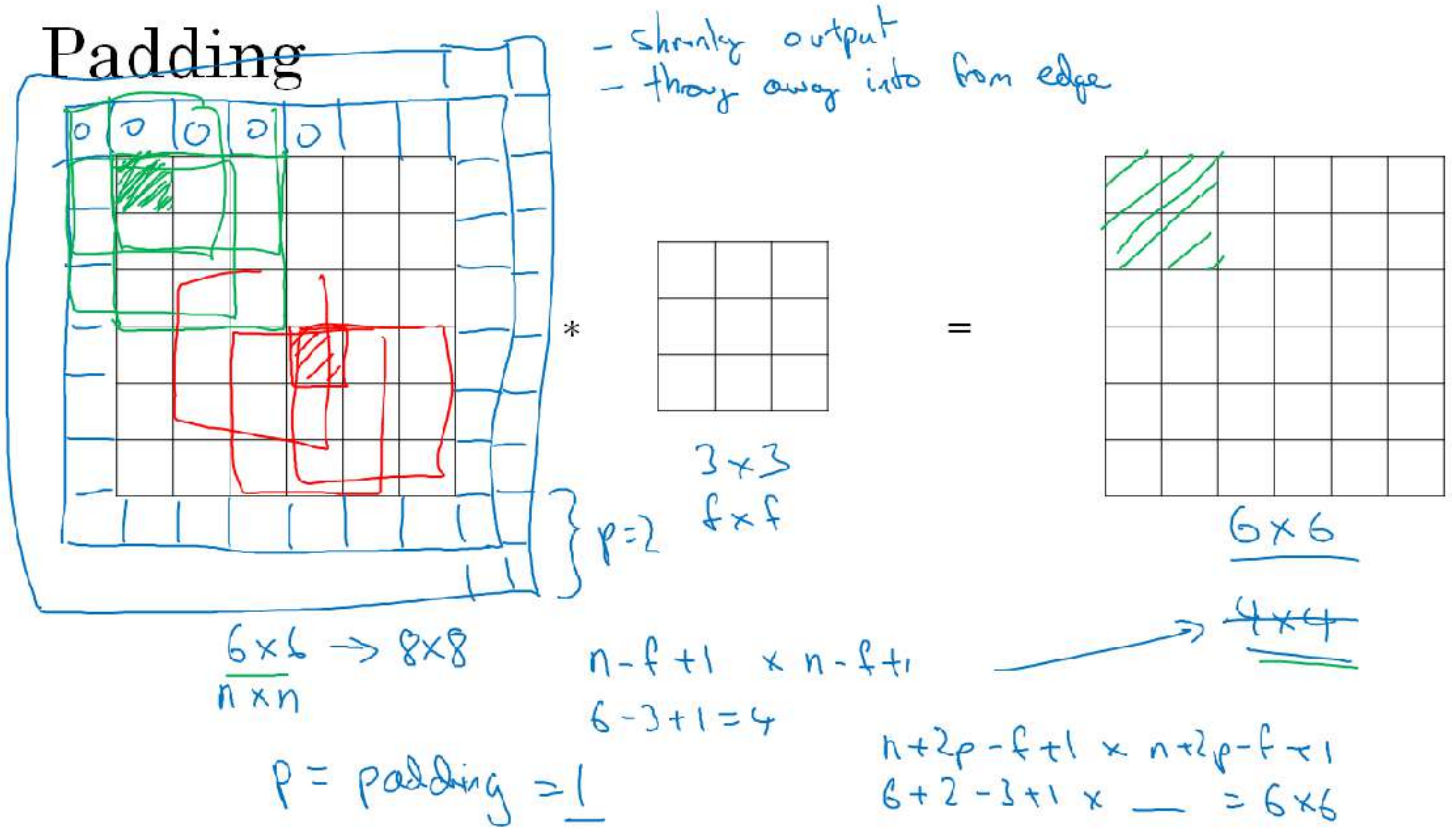


And what you see in later sections is that by just treating these nine numbers as parameters, the backprop can choose to learn 1, 1, 1, 0, 0, 0, -1,-1, if it wants, or learn the Sobel filter or learn the Scharr filter, or more likely learn something else that's even better at capturing the statistics of your data than any of these hand coded filters. And rather than just vertical and horizontal edges, maybe it can learn to detect edges that are at 45 degrees or 70 degrees or 73 degrees or at whatever orientation it chooses. And so by just letting all of these numbers be parameters and learning them automatically from data, we find that neural networks can actually learn low level features, can learn features such as edges, even more robustly than computer vision researchers are generally able to code up these things by hand. But underlying all these computations is still this convolution operation, Which allows back propagation to learn whatever three by three filter it wants and then to apply it throughout the entire image, at this position, at this position, at this position, in order to output whatever feature it's trying to detect. Be it vertical edges, horizontal edges, or edges at some other angle or even some other filter that we might not even have a name for in English.

So the idea you can treat these nine numbers as parameters to be learned has been one of the most powerful ideas in computer vision. And later in this section, we'll actually talk about the details of how you actually go about using back propagation to learn these nine numbers.

Padding

In order to build deep neural networks one modification to the basic convolutional operation that you need to really use is padding. Let's see how it works. What we saw in earlier sections is that if you take a 6x6 image and convolve it with a 3x3 filter, you end up with a 4x4 output with a 4x4 matrix, and that's because the number of possible positions with the 3x3 filter, there are only, sort of, 4x4 possible positions, for the 3x3 filter to fit in your 6x6 matrix and the math of this this turns out to be that if you have a end by end image and to involved that with an fxf filter, then the dimension of the output will be; **n- f +1 x n - f + 1**.



So the two downsides to this; one is that, if every time you apply a convolutional operator, your image shrinks, so you come from 6x6 down to 4x4 then, you can only do this a few times before your image starts getting really small, maybe it shrinks down to 1x1 or something, so maybe, you don't want your image to shrink every time you detect edges or to set other features on it, so that's one downside, and the second downside is that, if you look the pixel at the corner or the edge, this little pixel is touched as used only in one of the outputs, because this touches that 3x3 region. Whereas, if you take a pixel in the middle, say this pixel, then there are a lot of three by three regions that overlap that pixel and so, is as if pixels on the corners or on the edges are use much less in the output. So you're throwing away a lot of the information near the edge of the image. **So, to solve both of these problems, both the shrinking output, and when you build really deep neural networks, you see why you don't want the image to shrink on every step because if you have, maybe a hundred layer of deep net, then it'll shrinks a bit on every layer, then after a hundred layers you end up with a very small image. So that was one problem, the other is throwing away a lot of the information from the edges of the image.** So in order to fix both of these problems, what you can do is the full apply of convolutional operation. You can pad the image. So in this case, let's say you pad the image with an additional one border, with the additional border of one pixel all around the edges. So, if you do that, then instead of a 6x6 image, you've now padded this to 8x8 image and if you convolve an 8x8 with a 3x3 image you now get that out. Now, the 4x4 by the 6x6 image, so you managed to preserve the original input size of 6x6. So by convention when you pad, you padded with zeros and if p is the padding amounts. So in this case, p is equal to one, because we're padding all around with an extra boarder of one pixels, then the output becomes **n + 2p -f + 1 x n + 2p - f -1**. So we end up with a 6x6 image that preserves the size of the original image. So this being pixel actually influences all of these cells of the output and so this effective, maybe not by throwing away but counting less the information from the edge of the corner or the edge of the image is reduced. And we've shown here, the effect of padding deep border with just one pixel. If you want, you can also pad the border with two pixels, in which case I guess, you do add on another border here and they can pad it with even more pixels if you choose. So, I guess what I'm drawing here, this would be a padded equals to p plus two. In terms of how much to pad, it turns out there two common choices that are called, **Valid convolutions and Same convolutions.**

Valid and Same convolutions

"Valid": $n \times n \quad * \quad f \times f \quad \rightarrow \quad n-f+1 \times n-f+1$

$6 \times 6 \quad * \quad 3 \times 3 \quad \rightarrow \quad 4 \times 4$

"Same": Pad so that output size is the same as the input size.

$n+2p-f+1 \times n+2p-f+1$

$n+2p-f+1 = n \Rightarrow p = \frac{f-1}{2}$

$3 \times 3 \quad p = \frac{3-1}{2} = 1$

$5 \times 5 \quad p = 2$

f is usually odd

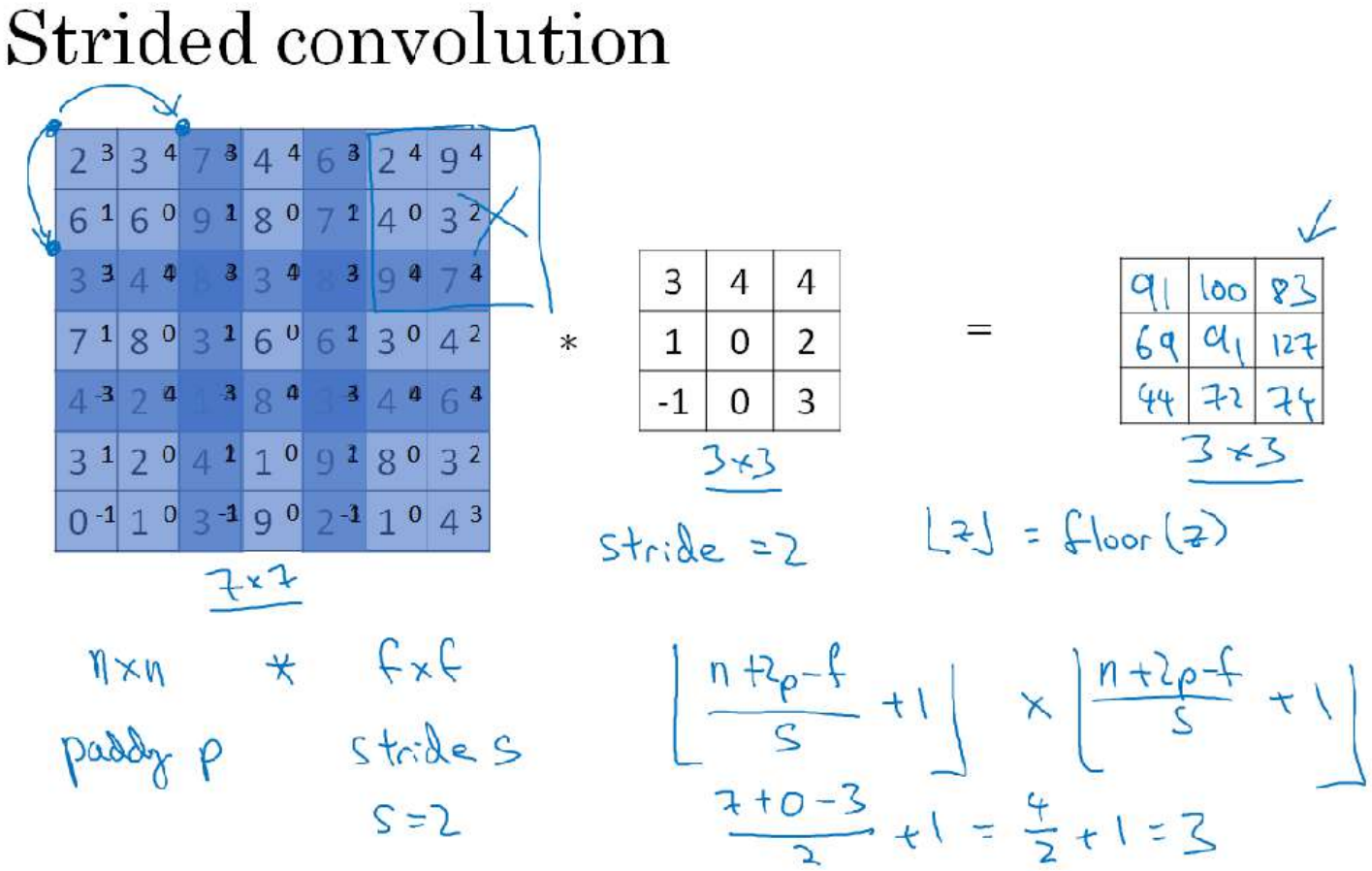
1×1
3×3
5×5
7×7

Not really is a great names but in a valid convolution, this basically means no padding. And so in this case you might have nxn image convolve with an fxf filter and this would give you an n minus f plus one by n minus f plus one dimensional output. So this is like the example we had previously on the previous sections where we had an n by n image convolve with the three by three filter and that gave you a four by four output. The other most common choice of padding is called the same convolution and that means when you pad, so the output size is the same as the input size. So if we actually look at this formula, when you pad by p pixels then, its as if n goes to n plus 2p and then you have from the rest of this, right? Minus f plus one. So we have an n by n image and the padding of a border of p pixels all around, then the output sizes of this dimension is xn plus 2p minus f plus one. And so, if you want n plus 2p minus f plus one to be equal to one, so the output size is same as input size, if you take this and solve for, I guess, n cancels out on both sides and if you solve for p, this implies that p is equal to f minus one over two. So when f is odd, by choosing the padding size to be as follows, you can make sure that the output size is same as the input size and that's why, for example, when the filter was three by three as this had happened in the previous slide, the padding that would make the output size the same as the input size was three minus one over two, which is one. And as another example, if your filter was five by five, so if f is equal to five, then, if you pad it into that equation you find that the padding of two is required to keep the output size the same as the input size when the filter is five by five. And by convention in computer vision, f is usually odd. It's actually almost always odd and you rarely see even numbered filters, filter works using computer vision. And I think that two reasons for that; one is that if f was even, then you need some asymmetric padding. So only if f is odd that this type of same convolution gives a natural padding region, had the same dimension all around rather than pad more on the left and pad less on the right, or something that asymmetric. And then second, when you have an odd dimension filter, such as three by three or five by five, then it has a central position and sometimes in computer vision its nice to have a distinguisher, it's nice to have a pixel, you can call the central pixel so you can talk about the position of the filter. Right, maybe none of this is a great reason for using f to be pretty much always odd but if you look a convolutional literature you see three by three filters are very common. You see some five by five, seven by sevens. And actually sometimes, later we'll also talk about one by one filters and that why that makes sense. But just by convention, I recommend you just use odd number filters as well. I think that you can probably get just fine performance even if you want to use an even

number value for f, but if you stick to the common computer vision convention, I usually just use odd number f. So you've now seen how to use padded convolutions. To specify the padding for your convolution operation, you can either specify the value for p or you can just say that this is a valid convolution, which means p equals zero or you can say this is a same convolution, which means pad as much as you need to make sure the output has same dimension as the input.

Strided Convolutions

Strided convolutions is another piece of the basic building block of convolutions as used in Convolutional Neural Networks. Example: Let's say you want to convolve a 7x7 image with 3x3 filter, except that instead of doing the usual way, we are going to do it with a stride of two.



What that means is you take the element Y's product as usual in this upper left three by three region and then multiply and add and that gives you 91. But then instead of stepping the blue box over by one step, we are going to step over by two steps. So, we are going to make it hop over two steps like so. Notice how the upper left hand corner has gone, jumping over one position. And then you do the usual element Y's product and summing it turns out 100. And now we are going to do they do that again, and make the blue box jump over by two steps. You end up there, and that gives you 83. Now, when you go to the next row, you again actually take two steps instead of one step so going to move the blue box over there. Notice how we are stepping over one of the positions and then this gives you 69, and now you again step over two steps, this gives you 91 and so on so 127. And then for the final row 44, 72, and 74. In this example, we convolve with a seven by seven matrix to this three by three matrix and we get a three by three outputs. The input and output dimensions turns out to be governed by the following formula, if you have an N by N image, they convolve with an F by F filter. And if you use padding P and stride S. In this example, S is equal to two then you end up with an output that is N plus two P minus F, and now because you're stepping S steps of the time, you step just one step of the time, you now divide by S plus one and then can apply the same thing. In our example, we have seven plus zero, minus three, divided by two S stride plus one equals let's see, that's four over two plus one equals three, which is why we wound up with this is three by three output. Now, just one last detail which is

what of this fraction is not an integer? In that case, we're going to round this down so this notation denotes the flow of something. This is also called the flow of Z. It means taking Z and rounding down to the nearest integer. The way this is implemented is that you take this type of blue box multiplication only if the blue box is fully contained within the image or the image plus to the padding and if any of this blue box kind of part of it hangs outside and you just do not do that computation. Then it turns out that if that's the convention that your three by three filter, must lie entirely within your image or the image plus the padding region before there's as a corresponding output generated that's convention. Then the right thing to do to compute the output dimension is to round down in case this N plus two P minus F over S is not an integer. Just to summarize the dimensions, if you have an N by N matrix or N by N image that you convolve with an F by F matrix or F by F filter with padding P N stride S, then the output size will have this dimension. It is nice we can choose all of these numbers so that there is an integer although sometimes you don't have to do that and rounding down is just fine as well. But please feel free to work through a few examples of values of N, F, P and S on yourself to convince yourself if you want, that this formula is correct for the output size.

Summary of convolutions

$n \times n$ image $\quad f \times f$ filter

padding p $\quad$ stride s

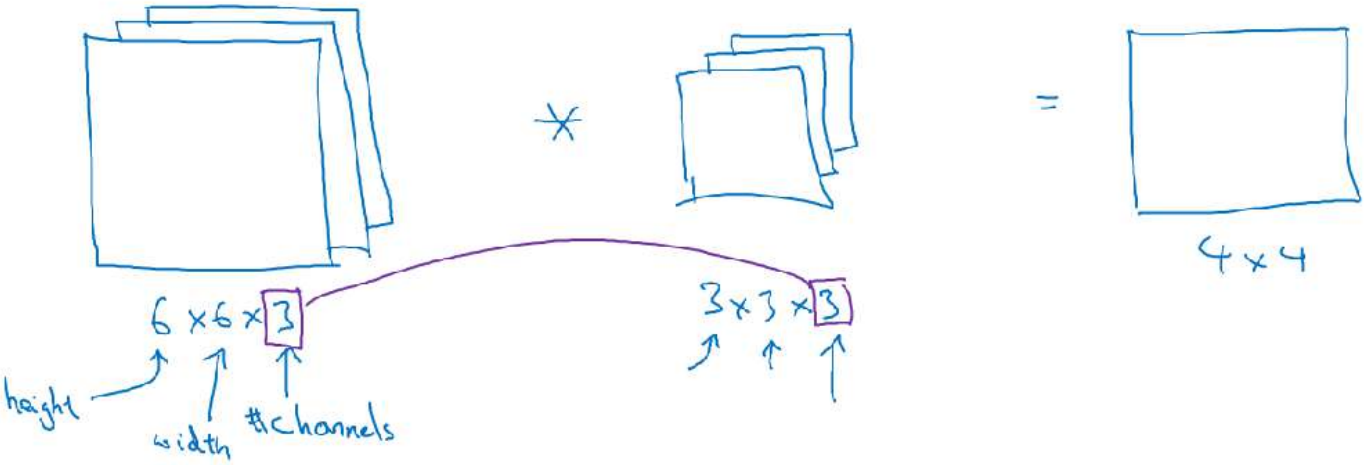
Output Size:

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

Convolutions Over Volume

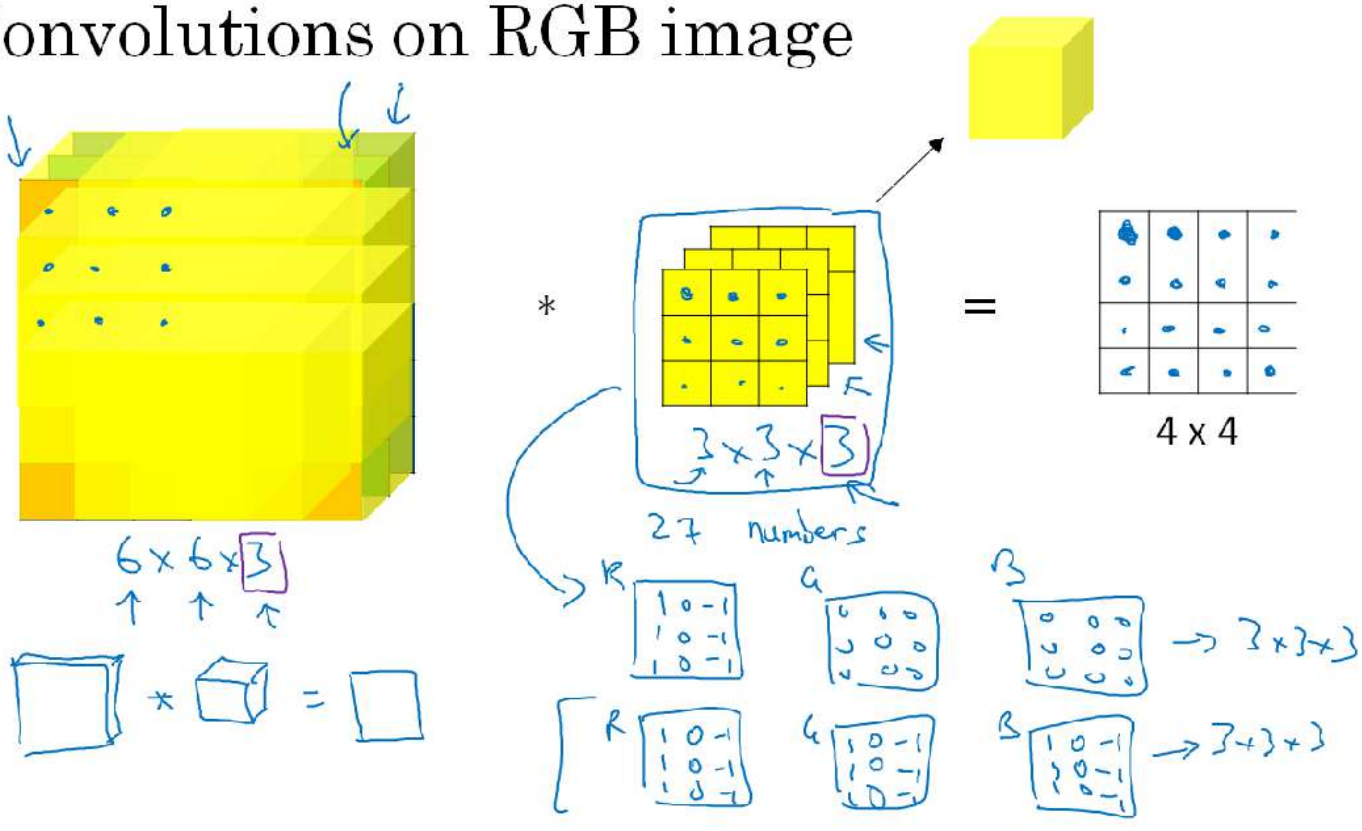
You've seen how convolutions over 2D images works. Now, let's see how you can implement convolutions over, not just 2D images, but over three dimensional volumes. Let's start with an example,

Convolutions on RGB images



let's say you want to detect features, not just in a great scale image, but in a RGB image. So, an RGB image might be instead of a six by six image, it could be six by six by three, where the three here responds to the three color channels. So, you think of this as a stack of three six by six images. In order to detect edges or some other feature in this image, you can vault this, not with a three by three filter, as we have previously, but now with also with a 3D filter, that's going to be 3x3x3. So the filter itself will also have three layers corresponding to the red, green, and blue channels. So to give these things some names, this first six here, that's the height of the image, that's the width, and this three is the number of channels. And your filter also similarly **has a height, a width, and the number of channels**. And the number of channels in your image must match the number of channels in your filter, so these two numbers have to be equal. We'll see on the next section how this convolution operation actually works, but the output of this will be a four by four image. And notice this is four by four by one, there's no longer a three at the end.

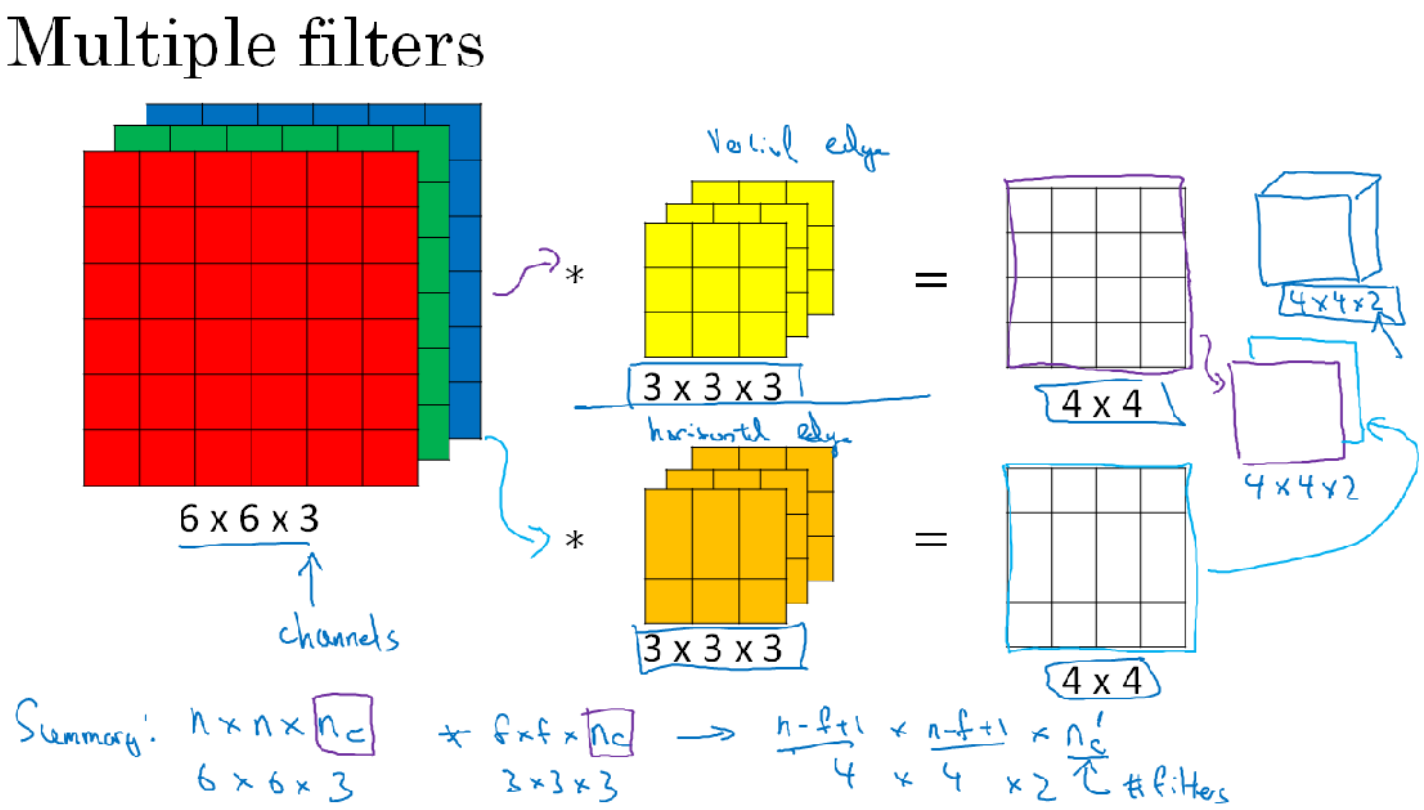
Convolutions on RGB image



Let's go through in detail how this works but let's use a more nicely drawn image. So here's the six by six by three image, and here's a three by three by three filter, and this last number, the number of channels matches the 3D image and the filter. So to simplify the drawing of this three by three by three filter, instead of joining it is a stack of the matrices, I'm also going to, sometimes, just draw it as this three dimensional cube, like that. So to compute the output of this convolutional operation, what you would do is take the three by three by three filter and first, place it in that upper left most position. So, notice that this three by three by three filter has 27 numbers, or 27 parameters, that's three cubes. And so, what you do is take each of these 27 numbers and multiply them with the corresponding numbers from the red, green, and blue channels of the image, so take the first nine numbers from red channel, then the three beneath it to the green channel, then the three beneath it to the blue channel, and multiply it with the corresponding 27 numbers that gets covered by this yellow cube show on the left. Then add up all those numbers and this gives you this first number in the output, and then to compute the next output you take this cube and slide it over by one, and again, due to 27 multiplications, add up the 27 numbers, that gives you this next output, do it for the next number over, for the next position over, that gives the third output and so on. That dives you the forth and then one row down and then the next one, to the next one, to the next one, and so on, you get the idea, until at the very end, that's the position you'll have for that final output.

So, what does this allow you to do? Well, here's an example, this filter is three by three by three. So, if you want to detect edges in the red channel of the image, then you could have the first filter, the one, one, one, one is one, one is one, one is one as usual, and have the green channel be all zeros, and have the blue filter be all zeros. And if you have these three stock together to form your three by three by three filter, then this would be a filter that detect edges, vertical edges but only in the red channel. Alternatively, if you don't care what color the vertical edge is in, then you might have a filter that's like this, whereas this one, one, one, minus one, minus one, minus one, in all three channels. So, by setting this second alternative, set the parameters, you then have a edge detector, a three by three by three edge detector, that detects edges in any color. And with different choices of these parameters you can get different feature detectors out of this three by three by three filter. And by convention, in computer vision, when you have an input with a certain height, a certain width, and a certain number of channels, then your filter will have a potential different height, different width, but the same number of channels. And in theory it's possible to have a filter that maybe only looks at the red channel or maybe a filter looks at only the green channel and a blue channel. And once again, you notice that convolving a volume, a six by six by three convolve with a three by three by three, that gives a four by four, a 2D output. Now that you know how to convolve on volumes, there is one last idea that will be crucial for building convolutional neural networks, which is what if we don't just wanted to detect vertical edges? What if we wanted to detect vertical edges and horizontal edges and maybe 45 degree edges and maybe 70 degree edges as well, but in other words, what if you want to use multiple filters at the

same time?

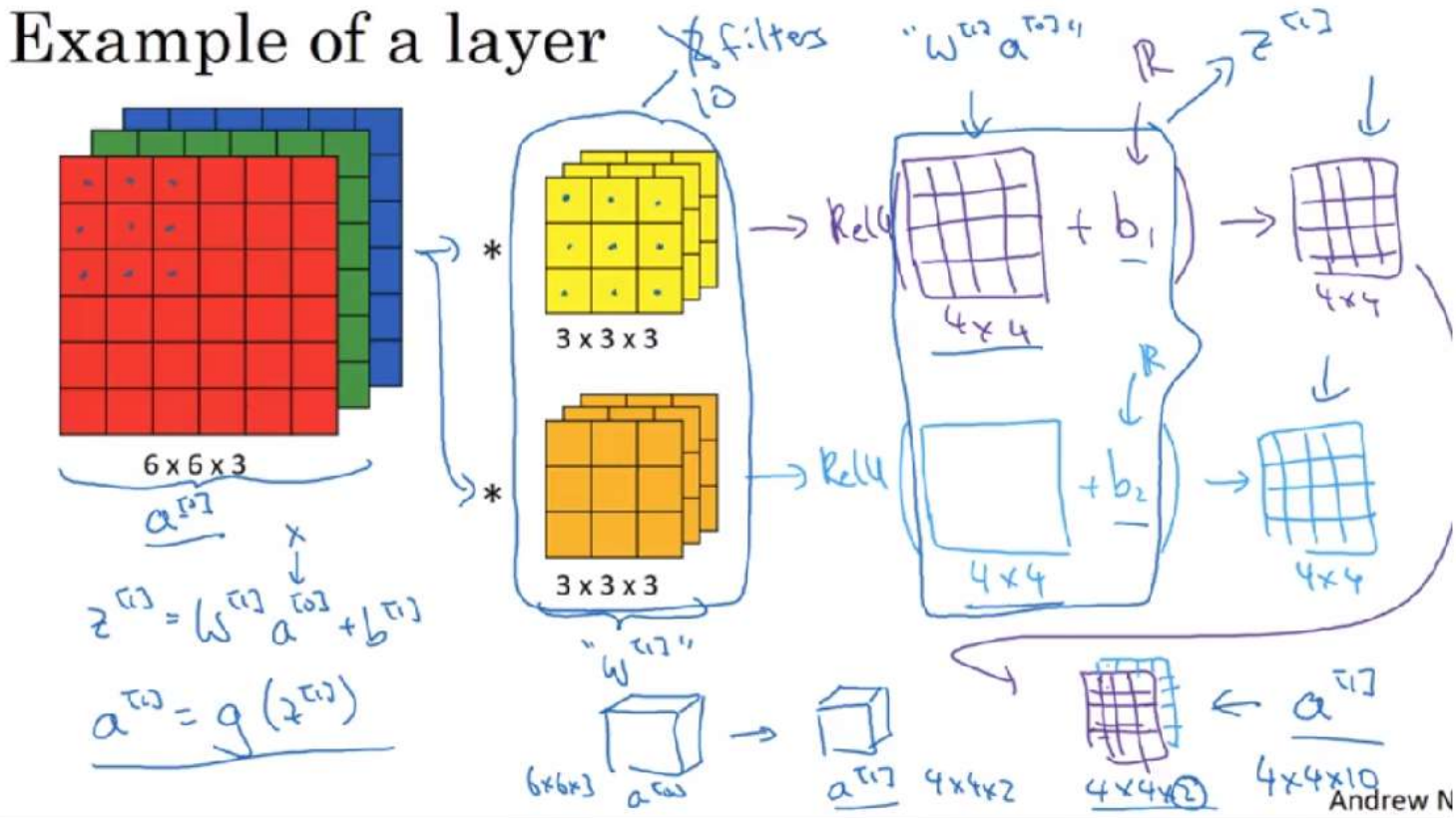


So, here's the picture we had from the previous section, we had six by six by three convolved with the three by three by three, gets four by four, and maybe this is a vertical edge detector, or maybe it's run to detect some other feature. Now, maybe a second filter may be denoted by this orange-ish color, which could be a horizontal edge detector. So, maybe convolving it with the first filter gives you this first four by four output and convolving with the second filter gives you a different four by four output. And what we can do is then take these two four by four outputs, take this first one within the front and you can take this second filter output and well, let me draw it here, put it at back as follows, so that by stacking these two together, you end up with a four by four by two output volume, right? And you can think of the volume as if we draw this is a box, I guess it would look like this. So this would be a four by four by two output volume, which is the result of taking your six by six by three image and convolving it or applying two different three by three filters to it, resulting in two four by four outputs that then gets stacked up to form a four by four by two volume. And the two here comes from the fact that we used two different filters.

So, let's just summarize the dimensions, if you have nxn number of channels input image, so an example, there's a 6x6x3, where n subscript C is the number of channels, and you convolve that with a f by f by, and again, this should be the same nC, so this was, three by three by three, and by convention this and this have to be the same number. Then, what you get is n minus f plus one by n minus f plus one by and you want to use this nC prime, or its really nC of the next layer, but this is the number of filters that you use. So this in our example would be be four by four by two. And I wrote this assuming that you use a stride of one and no padding. But if you used a different stride of padding than this n minus F plus one would be affected in a usual way, as we see in the previous videos. So this idea of convolution on volumes, turns out to be really powerful. Only a small part of it is that you can now operate directly on RGB images with three channels. But even more important is that you can now detect two features, like vertical, horizontal edges, or 10, or maybe a 128, or maybe several hundreds of different features. And the output will then have a number of channels equal to the number of filters you are detecting. And as a note of notation, I've been using your number of channels to denote this last dimension in the literature, people will also often call this the depth of this 3D volume and both notations, channels or depth, are commonly used in the literature.

One Layer of a Convolution Network

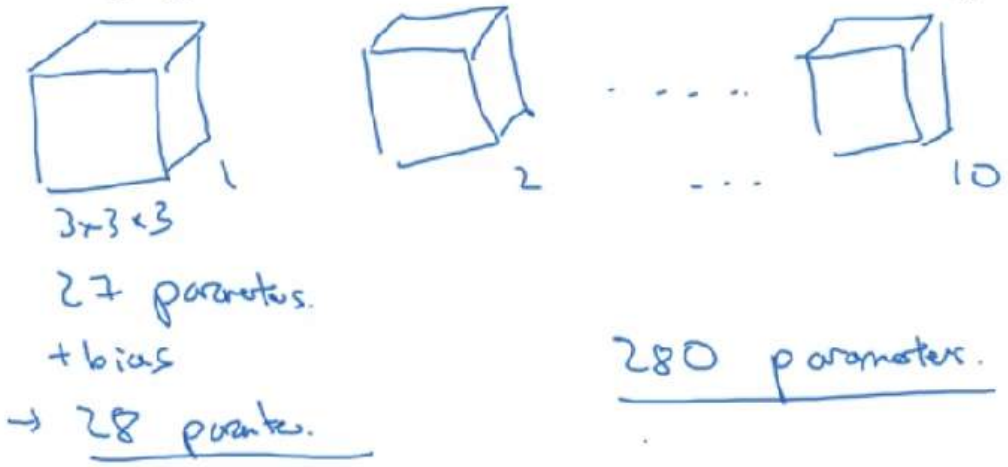
We are now ready to see how to build one layer of a convolutional neural network, let's go through the example. You've seen at the previous section how to take a 3D volume and convolve it with say two different filters. In order to get in this example to different 4 by 4 outputs. So let's say convolving with the first filter gives this first 4 by 4 output, and convolving with this second filter gives a different 4 by 4 output. The final thing to turn this into a convolutional neural net layer, is that for each of these we're going to add it bias, so this is going to be a real number. And where python broadcasting, you kind of have to add the same number so every one of these 16 elements. And then apply a non-linearity which for this illustration that says relative non-linearity, and this gives you a 4 by 4 output, all right? After applying the bias and the non-linearity. And then for this thing at the bottom as well, you add some different bias, again, this is a real number. So you add the single number to all 16 numbers, and then apply some non-linearity, let's say a real non-linearity. And this gives you a different 4 by 4 output. Then same as we did before, if we take this and stack it up as follows, so we ends up with a 4 by 4 by 2 outputs. Then this computation where you come from a 6 by 6 by 3 to 4 by 4 by 4, this is one layer of a convolutional neural network. So to map this back to one layer of four propagation in the standard neural network, in a non-convolutional neural network. Remember that one step before the prop was something like this, right? z1 = w1* a0, a0 was also equal to x, and then plus b[1]. And you apply the non-linearity to get a[1], so that's g(z[1]). So this input here, in this analogy this is a[0], this is x3 and these filters here, this plays a role similar to w1. And you remember during the convolution operation, you were taking these 27 numbers, or really well, 27 times 2, because you have two filters. You're taking all of these numbers and multiplying them. So you're really computing a linear function to get this 4 x 4 matrix. So that 4 x 4 matrix, the output of the convolution operation, that plays a role similar to w1 * a0. That's really maybe the output of this 4 x 4 as well as that 4 x 4. And then the other thing you do is add the bias. So, this thing here before applying value, this plays a role similar to z. And then it's finally by applying the non-linearity, this kind of this I guess. So, this output plays a role, this really becomes your activation at the next layer. So this is how you go from a0 to a1, as far as the linear operation and then convolution has all these multiples. So the convolution is really applying a linear operation and you have the biases and the applied value operation and you've gone from a 6 by 6 by 3, dimensional a0, through one layer of neural network to, I guess a 4 by 4 by 2 dimensional a(1). And so 6 by 6 by 3 has gone to 4 by 4 by 2, and so that is one layer of convolutional net.



Now in this example we have two filters, so we had two features of you will, which is why we round up with our output 4 x4x2. But if for example we instead had 10 filters instead of 2, then we would have round up with the 4x4x10 dimensional output volume. Because we'll be taking 10 of these not just two of them, and stacking them up to form a 4 x4x10 output volume, and that's what a1 would be. So, to make sure you understand this, let's go through an exercise. Let's suppose you have 10 filters, not just two filters, that are 3x3x3 and 1 layer of a neural network, how many parameters does this layer have? Well, let's figure this out. Each filter, is a 3 x 3 x 3 volume, so 3 x 3 x 3, so each fill has 27 parameters, all right? There's 27 numbers to be run, and plus the bias. So that was the b parameter, so this gives you 28 parameters.

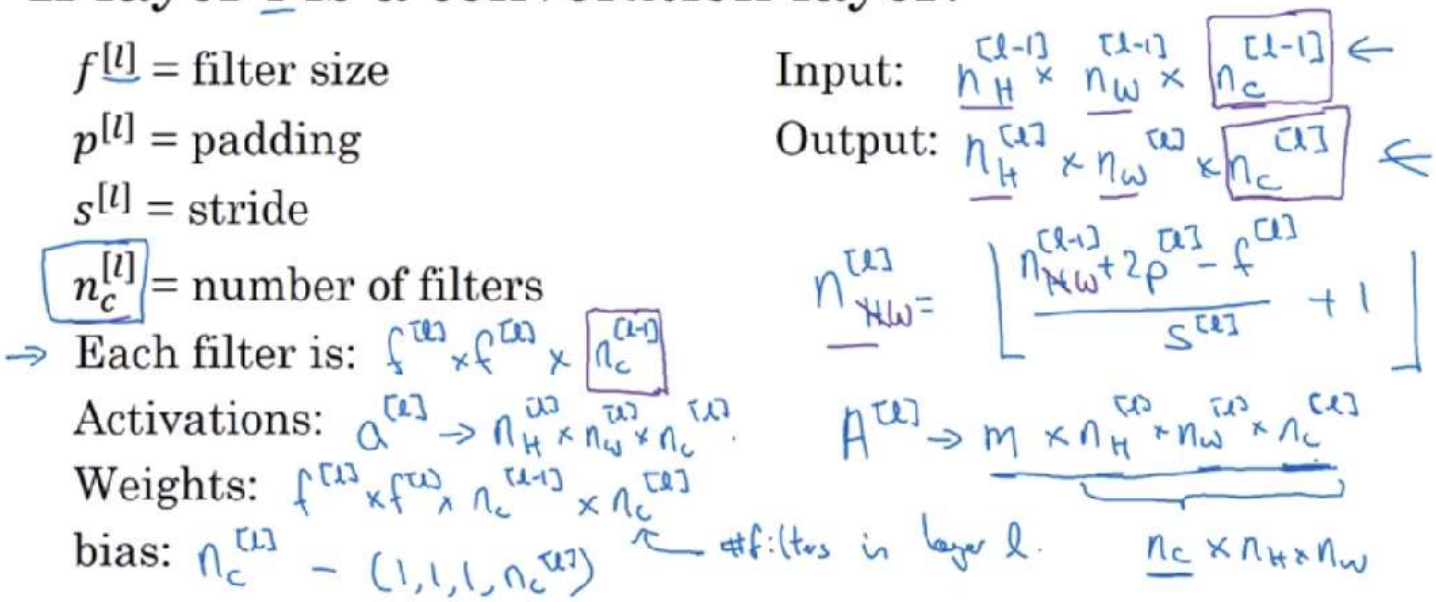
Number of parameters in one layer

If you have 10 filters that are 3 x 3 x 3 in one layer of a neural network, how many parameters does that layer have?



and then if you imagine that on the previous section we had drawn two filters, but now if you imagine that you actually have ten of these, right? 1, 2..., 10 of these, then all together you'll have 28 times 10, so that will be 280 parameters. Notice one nice thing about this, is that no matter how big the input image is, the input image could be 1,000x1,000 or 5,000x5,000, but the number of parameters you have still remains fixed as 280. And you can use these **ten filters to detect features, vertical edges, horizontal edges maybe other features anywhere even in a very, very large image is just a very small number of parameters.** So these is really one property of convolution neural network that makes less prone to overfitting then if you could. So once you've learned 10 feature detectors that work, you could apply this even to large images. And the number of parameters still is fixed and relatively small, as 280 in this example. Let's just summarize the notation we are going to use to describe one layer to describe a covolutional layer in a convolutional neural network.

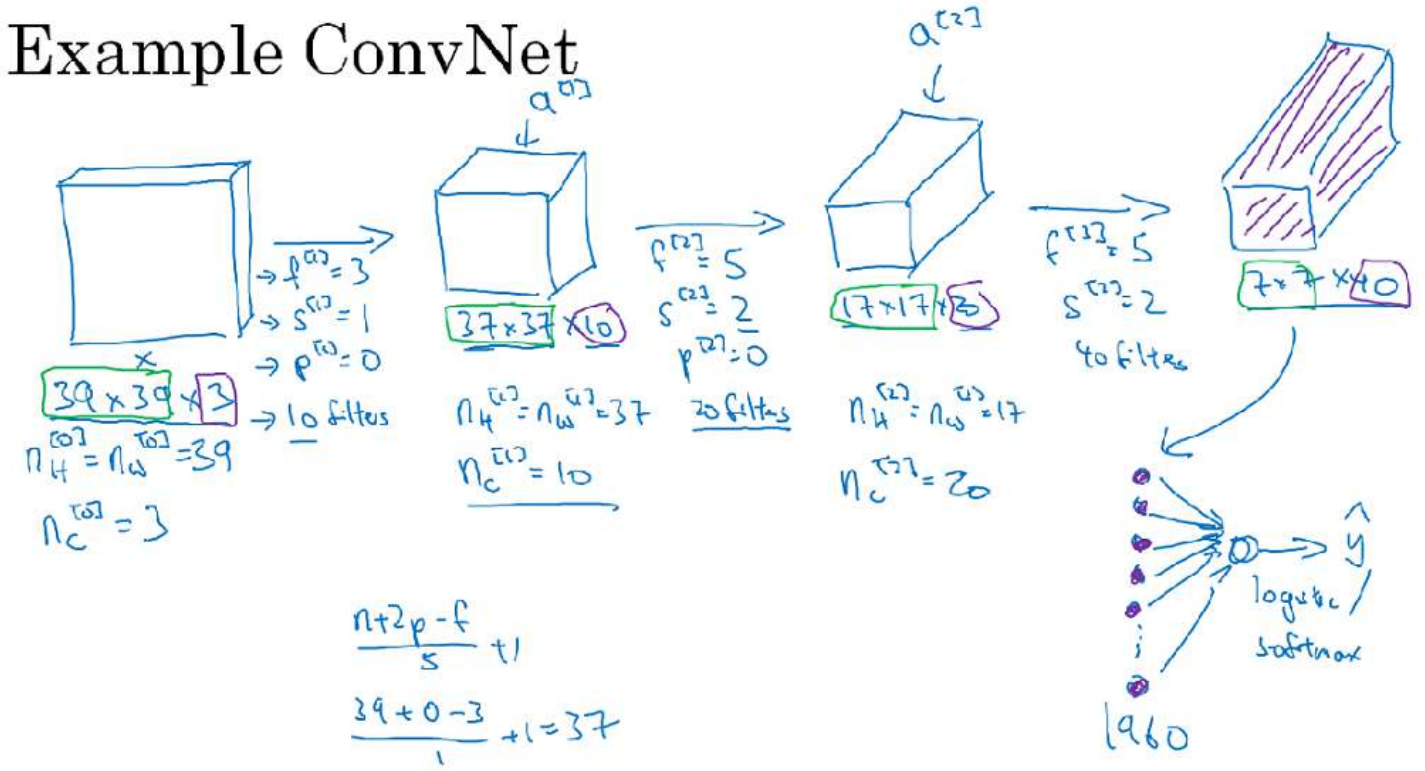
Summary of notation



Simple Convolutions Network Example

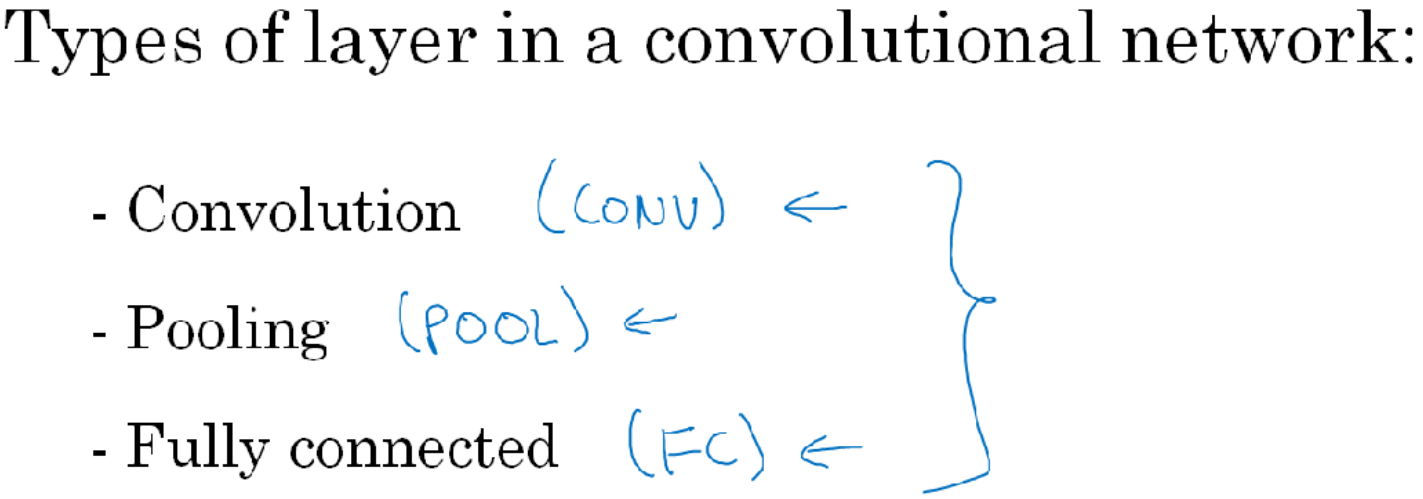
In the last section, we saw the building blocks of a single layer, of a single convolution layer in the ConvNet. Now let's go through a concrete example of a deep convolutional neural network and this will give you some practice with the notation that we introduced toward the end of the last video as well.

Let's say you have an image, and you want to do image classification, or image recognition. Where you want to take as input an image, x, and decide is this a cat or not, 0 or 1, so it's a classification problem. Let's build an example of a ConvNet you could use for this task. For the sake of this example, I'm going to use a fairly small image. Let's say this image is 39 x 39 x 3. This choice just makes some of the numbers work out a bit better. And so, nH in layer 0 will be equal to nw height and width are equal to 39 and the number of channels and layer 0 is equal to 3. Let's say the first layer uses a set of 3 by 3 filters to detect features, so f = 3 or really f1 = 3, because we're using a 3 by 3 process. And let's say we're using a stride of 1, and no padding. So using a same convolution, and let's say you have 10 filters.



Then the activations in this next layer of the neutral network will be 37 x 37 x 10, and this 10 comes from the fact that you use 10 filters. And 37 comes from this formula n + 2p- f over s + 1. Right, then I guess you have 39 + 0- 3 over 1 + 1 that's = to 37. So that's why the output is 37 by 37, it's a valid convolution and that's the output size. So in our notation you would have nh[1] = nw[1] = 37 and nc[1] = 10, so nc[1] is also equal to the number of filters from the first layer. And so this becomes the dimension of the activation at the first layer. Let's say you now have another convolutional layer and let's say this time you use 5 by 5 filters. So, in our notation f[2] at the next neural network = 5, and let's say use a stride of 2 this time. And maybe you have no padding and say, 20 filters. So then the output of this will be another volume, this time it will be 17 x 17 x 20. Notice that, because you're now using a stride of 2, the dimension has shrunk much faster. 37 x 37 has gone down in size by slightly more than a factor of 2, to 17 x 17. And because you're using 20 filters, the number of channels now is 20. So it's this activation a2 would be that dimension and so nh[2] = nw[2] = 17 and nc[2] = 20. All right, let's apply one last convolutional layer. So let's say that you use a 5 by 5 filter again, and again, a stride of 2. So if you do that, I'll skip the math, but you end up with a 7 x 7, and let's say you use 40 filters, no padding, 40 filters. You end up with 7 x 7 x 40. So now what you've done is taken your 39 x 39 x 3 input image and computed your 7 x 7 x 40 features for this image. And then finally, what's commonly done is if you take this 7 x 7 x 40, 7 times 7 times 40 is actually 1,960. And so what we can do is take this volume and flatten it or unroll it into just 1,960 units, right? Just flatten it out into a vector, and then feed this to a **logistic regression unit, or a softmax** unit. Depending on whether you're trying to recognize or trying to recognize any one of key different objects and then just have this give the final predicted output for the neural network. So just be clear, this last step is just taking all of these numbers, all 1,960 numbers, and unrolling them into a very long vector. So then you just have one long vector that you can feed into softmax until it's just a regression in order to make prediction for the final output. So this would be a pretty typical example of a ConvNet. A lot of the work in designing convolutional neural net is selecting hyperparameters like these, deciding what's the total size? What's the stride? What's the padding and how many filters are used? and both later this week as well as next week, we'll give some suggestions and some guidelines on how to make these choices. But for now, maybe one thing to take away from this is that as you go deeper in a neural network, typically you start off with larger images, 39 by 39. And then the height and width will stay the same for a while and gradually trend down as you go deeper in the neural network. It's gone from 39 to 37 to 17 to 14. Excuse me, it's gone from 39 to 37 to 17 to

7. Whereas the number of channels will generally increase. It's gone from 3 to 10 to 20 to 40, and you see this general trend in a lot of other convolutional neural networks as well. So we'll get more guidelines about how to design these parameters in later sections. So it turns out that in a typical ConvNet, there are usually three types of layers. One is the **convolutional layer**, and often we'll denote that as a **Conv layer** and that's what we've been using in the previous network. It turns out that there are two other common types of layers that you haven't seen yet but we'll talk about in the next couple of videos. One is called a pooling layer, often I'll call this pool. And then the last is a fully connected layer called FC. And although it's possible to design a pretty good neural network using just convolutional layers, most neural network architectures will also have a few pooling layers and a few fully connected layers.

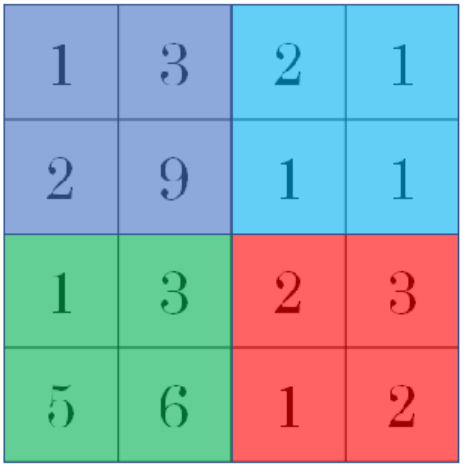


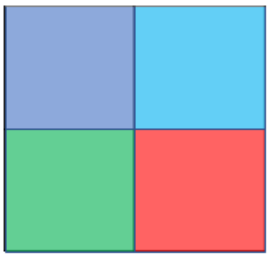
Fortunately pooling layers and fully connected layers are a bit simpler than convolutional layers to define. So we'll do that quickly in the next two videos and then you have a sense of all of the most common types of layers in a convolutional neural network. And you will put together even more powerful networks than the one we just saw.

Pooling Layers

Other than convolutional layers, ConvNets often also use pooling layers to reduce the size of the representation, to speed the computation, as well as make some of the features that detects a bit more robust. Let's take a look. Let's go through an example of pooling, and then we'll talk about why you might want to do this. Suppose you have a four by four input, and you want to apply a type of pooling called max pooling. And the output of this particular implementation of max pooling will be a two by two output. And the way you do that is quite simple.

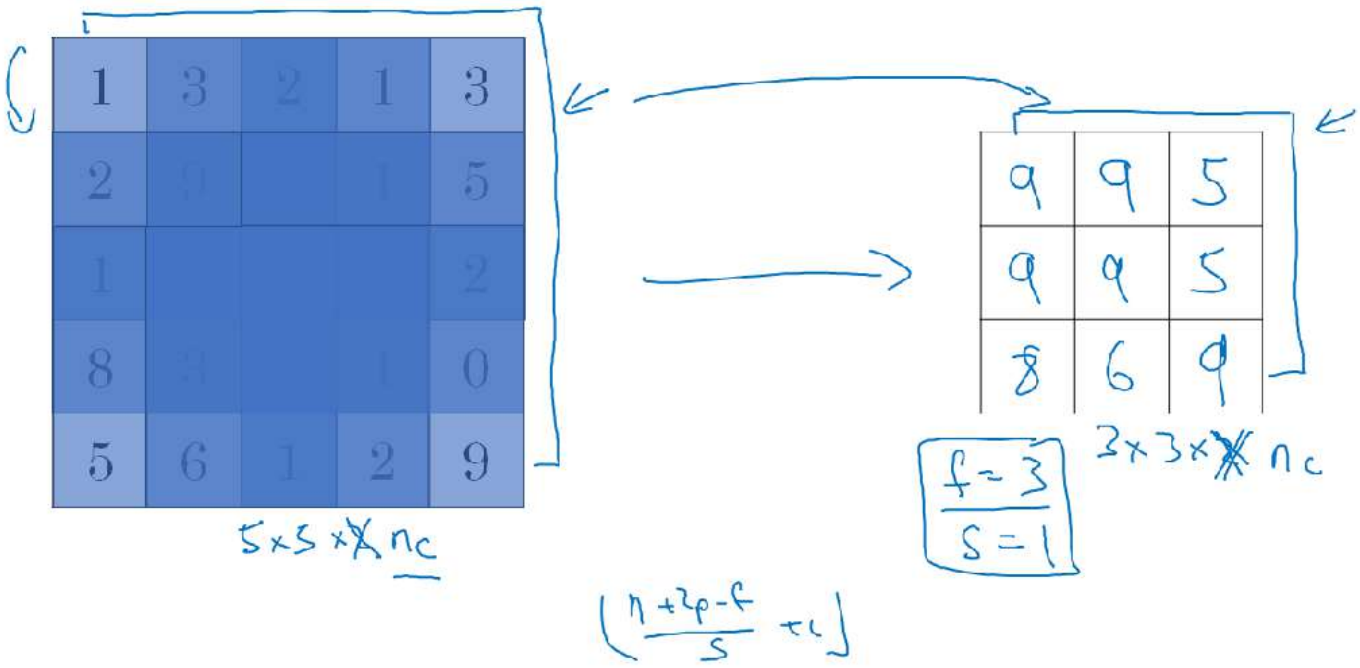
Pooling layer: Max pooling





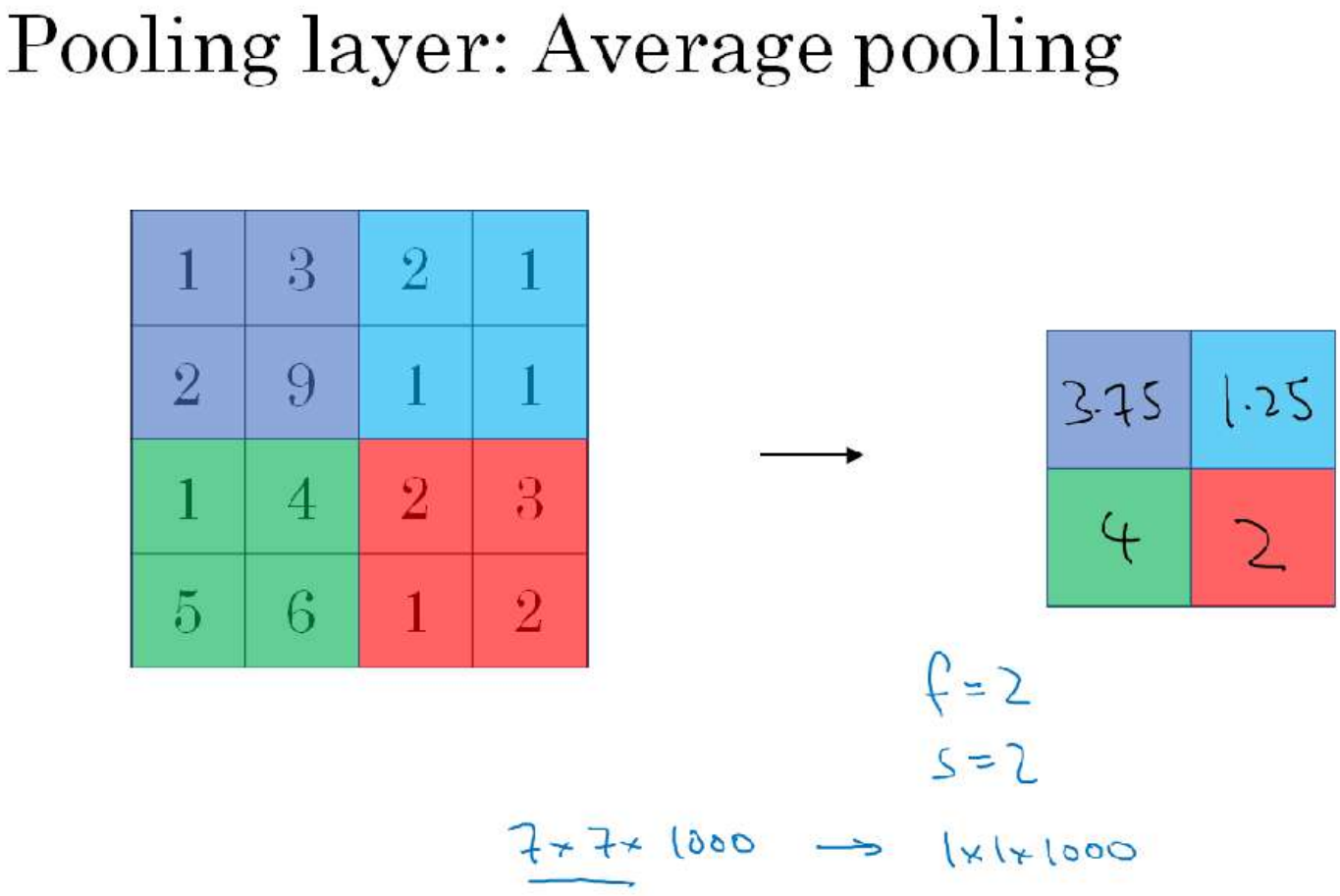
Take your four by four input and break it into different regions and I'm going to color the four regions as follows. And then, in the output, which is two by two, each of the outputs will just be the max from the corresponding reshaded region. So the upper left, I guess, the max of these four numbers is nine. On upper right, the max of the blue numbers is two. Lower left, the biggest number is six, and lower right, the biggest number is three. So to compute each of the numbers on the right, we took the max over a two by two regions.

Pooling layer: Max pooling

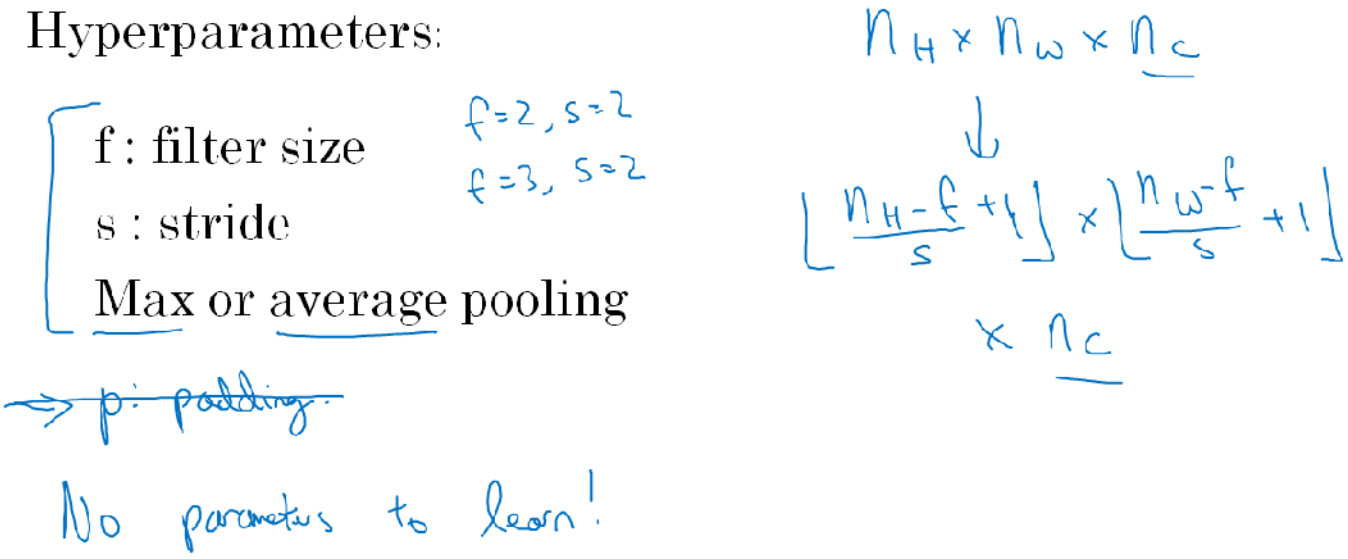


So, this is as if you apply a filter size of two because you're taking a two by two regions and you're taking a stride of two. So, these are actually the hyperparameters of max pooling because we start from this filter size. It's like a two by two region that gives you the nine. And then, you step all over two steps to look at this region, to give you the two, and then for the next row, you step it

down two steps to give you the six, and then step to the right by two steps to give you three. So because the squares are two by two, f is equal to two, and because you stride by two, s is equal to two. So here's the intuition behind what max pooling is doing. If you think of this four by four region as some set of features, the activations in some layer of the neural network, then a large number, it means that it's maybe detected a particular feature. So, the upper left-hand quadrant has this particular feature. It maybe a vertical edge or maybe a higher or whisker if you trying to detect clearly, that feature exists in the upper left-hand quadrant. Whereas this feature, maybe it isn't cat eye detector. Whereas this feature, it doesn't really exist in the upper right-hand quadrant. So what the max operation does is a lots of features detected anywhere, and one of these quadrants , it then remains preserved in the output of max pooling. So, what the max operates to does is really to say, if these features detected anywhere in this filter, then keep a high number. But if this feature is not detected, so maybe this feature doesn't exist in the upper right-hand quadrant. **Then the max of all those numbers is still itself quite small**. So maybe that's the intuition behind max pooling. But I have to admit, I think the **main reason people use max pooling is because it's been found in a lot of experiments to work well**, and the intuition I just described, despite it being often cited, I don't know of anyone fully knows if that is the real underlying reason. I don't have anyone knows if that's the real underlying reason that max pooling works well in ConvNets. **One interesting property of max pooling is that it has a set of hyperparameters but it has no parameters to learn. There's actually nothing for gradient descent to learn**. Once you fix f and s, it's just a fixed computation and gradient descent doesn't change anything.



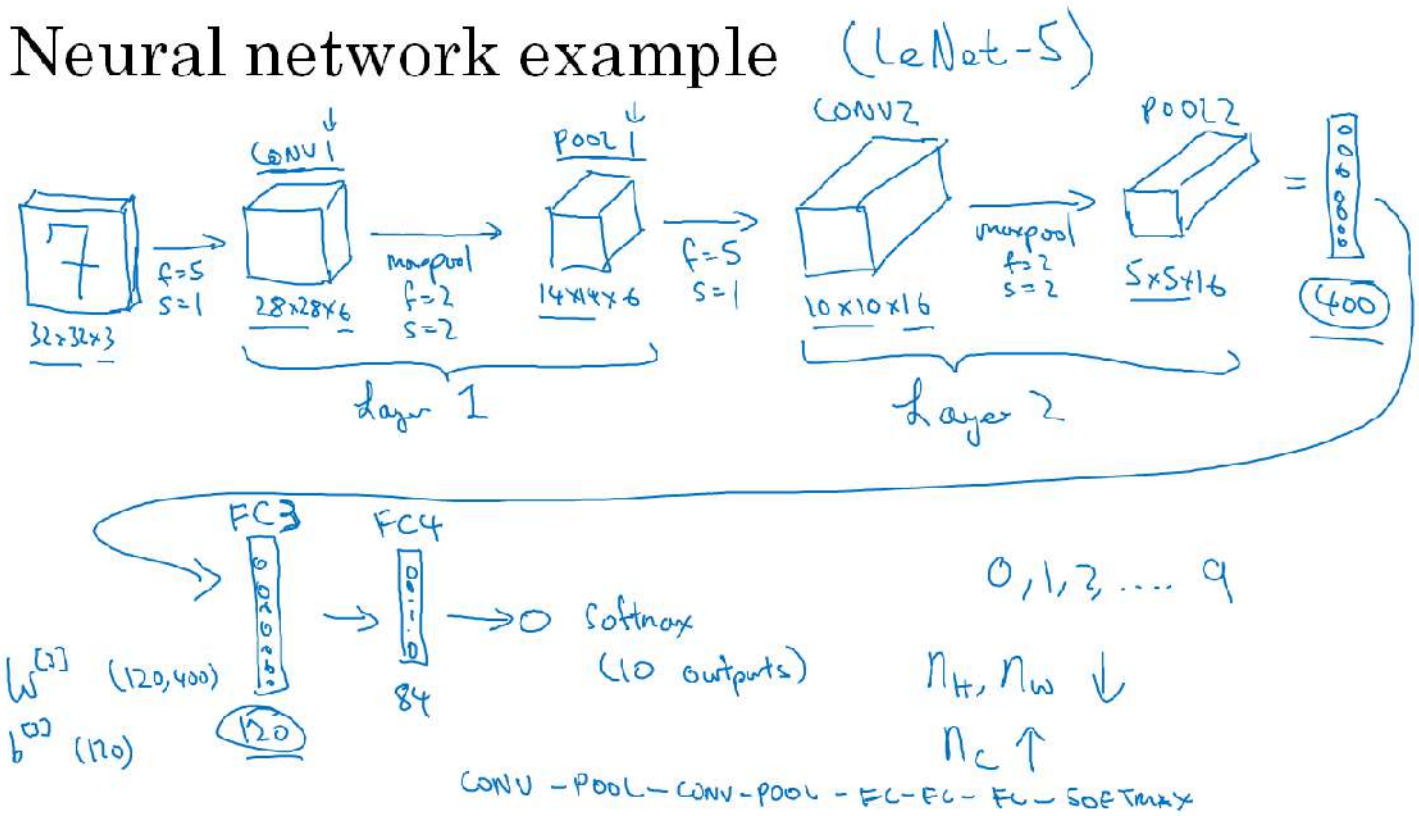
Summary of pooling



CNN Example

You now know pretty much all the building blocks of building a full convolutional neural network. Let's look at an example. Let's say you're inputting an image which is 32 x 32 x 3, so it's an RGB image and maybe you're trying to do handwritten digit recognition. So you have a number like 7 in a 32 x 32 RGB initiate trying to recognize which one of the 10 digits from zero to nine is this. Let's throw the neural network to do this. And what I'm going to use in this slide is inspired, it's actually quite similar to one of the classic neural networks called LeNet-5, which is created by Yann LeCun many years ago. What I'll show here isn't exactly LeNet-5 but it's inspired by it, but many parameter choices were inspired by it. So with a 32 x 32 x 3 input let's say that the first layer uses a 5 x 5 filter and a stride of 1, and no padding. So the output of this layer, if you use 6 filters would be 28 x 28 x 6, and we're going to call this layer conv 1. So you apply 6 filters, add a bias, apply the non-linearity, maybe a real non-linearity, and that's the conv 1 output. Next, let's apply a pooling layer, so I am going to apply mass pooling here and let's use a f=2, s=2. When I don't write a padding use a pad easy with a 0. Next let's apply a pooling layer, I am going to apply, let's see max pooling with a 2 x 2 filter and the stride equals 2. So this is should reduce the height and width of the representation by a factor of 2. So 28 x 28 now becomes 14 x 14, and the number of channels remains the same so 14 x 14 x 6, and we're going to call this the Pool 1 output. So, it turns out that in the literature of a ConvNet there are two conventions which are inside the inconsistent about what you call a layer. One convention is that this is called one layer. So this will be layer one of the neural network, and now the conversion will be to call they convey layer as a layer and the pool layer as a layer. When people report the number of layers in a neural network usually people just record the number of layers that have weight, that have parameters. And because the pooling layer has no weights, has no parameters, only a few hyper parameters, I'm going to use a convention that Conv 1 and Pool 1 shared together. I'm going to treat that as Layer 1, although sometimes you see people maybe read articles online and read research papers, you hear about the conv layer and the pooling layer as if they are two separate layers. But this is maybe two slightly inconsistent notation terminologies, but when I count layers, I'm just going to count layers that have weights. So achieve both of this together as Layer 1. And the name Conv1 and Pool1 use here the 1 at the end also refers the fact that I view both of this is part of Layer 1 of the neural network. And Pool 1 is grouped into Layer 1 because it doesn't have its own weights. Next, given a 14 x 14 bx 6 volume, let's apply another convolutional layer to it, let's use a filter size that's 5 x 5, and let's use a stride of 1, and let's use 10 filters this time. So now you end up with, A 10 x 10 x 10 volume, so I'll call this Comv 2, and then in this network let's do max pulling

with f=2, s=2 again. So you could probably guess the output of this, f=2, s=2, this should reduce the height and width by a factor of 2, so you're left with 5 x 5 x 10. And so I'm going to call this Pool 2, and in our convention this is Layer 2 of the neural network. Now let's apply another convolutional layer to this. I'm going to use a 5 x 5 filter, so f = 5, and let's try this, 1, and I don't write the padding, means there's no padding. And this will give you the Conv 2 output, and that's your 16 filters. So this would be a 10 x 10 x 16 dimensional output. So we look at that, and this is the Conv 2 layer. And then let's apply max pooling to this with f=2, s=2. You can probably guess the output of this, we're at 10 x 10 x 16 with max pooling with f=2, s=2. This will half the height and width, you can probably guess the result of this, right? Left pooling with f = 2, s = 2. This should halve the height and width so you end up with a 5 x 5 x 16 volume, same number of channels as before. We're going to call this Pool 2. And in our convention this is Layer 2 because this has one set of weights and your Conv 2 layer. Now 5 x 5 x 16, 5 x 5 x 16 is equal to 400. So let's now fatten our Pool 2 into a 400 x 1 dimensional vector. So think of this as fatting this up into these set of neurons, like so. And what we're going to do is then take these 400 units and let's build the next layer, As having 120 units. So this is actually our first fully connected layer. I'm going to call this FC3 because we have 400 units densely connected to 120 units.



So this fully connected unit, this fully connected layer is just like the single neural network layer that you saw in Courses 1 and 2. This is just a standard neural network where you have a weight matrix that's called W3 of dimension 120 x 400. And this is fully connected because each of the 400 units here is connected to each of the 120 units here, and you also have the bias parameter, yes that's going to be just a 120 dimensional, this is 120 outputs. And then lastly let's take 120 units and add another layer, this time smaller but let's say we had 84 units here, I'm going to call this fully connected Layer 4. And finally we now have 84 real numbers that you can fit to a [INAUDIBLE] unit. And if you're trying to do handwritten digital recognition, to recognize this hand it is 0, 1, 2, and so on up to 9. Then this would be a softmax with 10 outputs. So this is a vis-a-vis typical example of what a convolutional neural network might look like. And I know this seems like there a lot of hyper parameters. We'll give you some more specific suggestions later for how to choose these types of hyper parameters. Maybe one common guideline is to actually not try to invent your own settings of hyper parameters, but to look in the literature to see what hyper parameters you work for others. And to just choose an architecture that has worked well for someone else, and there's a chance that will work for your application as well. We'll see more

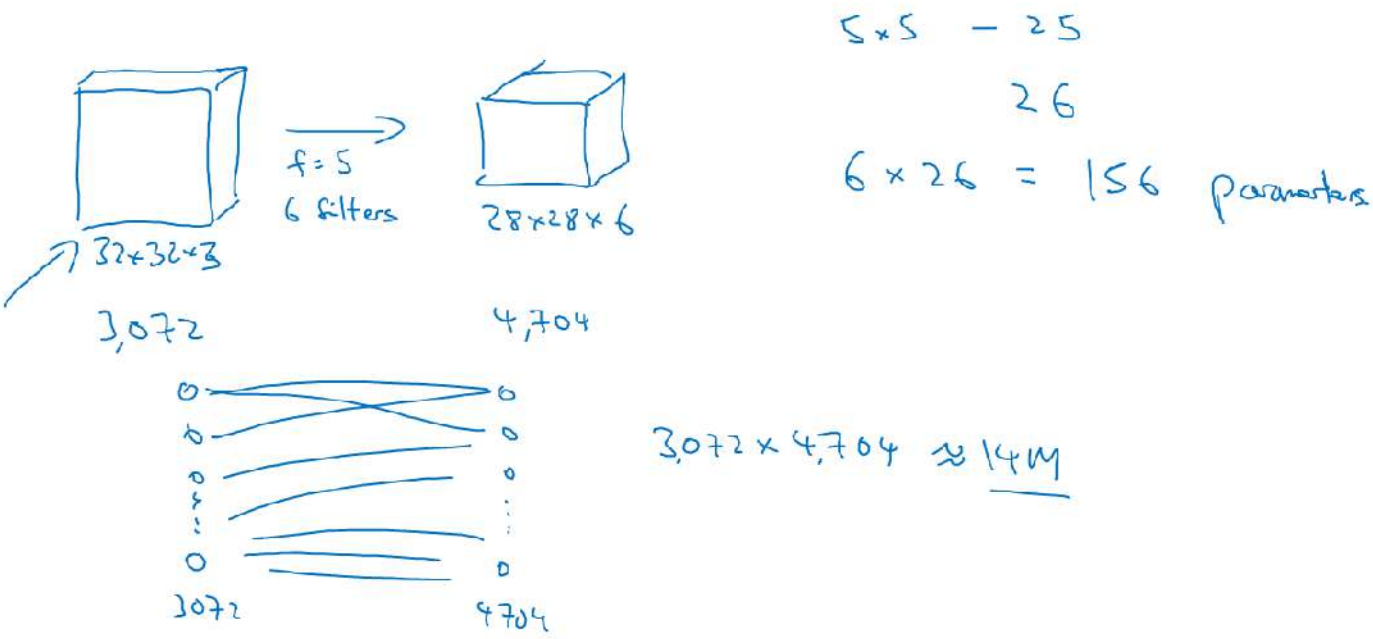
about that next week. But for now I'll just point out that as you go deeper in the neural network, usually nh and nw to height and width will decrease. Pointed this out earlier, but it goes from 32 x 32, to 20 x 20, to 14 x 14, to 10 x 10, to 5 x 5. So as you go deeper usually the height and width will decrease, whereas the number of channels will increase. It's gone from 3 to 6 to 16, and then your fully connected layer is at the end. And another pretty common pattern you see in neural networks is to have conv layers, maybe one or more conv layers followed by a pooling layer, and then one or more conv layers followed by pooling layer. And then at the end you have a few fully connected layers and then followed by maybe a softmax. And this is another pretty common pattern you see in neural networks. So let's just go through for this neural network some more details of what are the activation shape, the activation size, and the number of parameters in this network. So the input was 32 x 30 x 3, and if you multiply out those numbers you should get 3,072. So the activation, a0 has dimension 3072. Well it's really 32 x 32 x 3. And there are no parameters I guess at the input layer. And as you look at the different layers, feel free to work out the details yourself. These are the activation shape and the activation sizes of these different layers.

Why Convolutions?

For this final section for this week, let's talk a bit about why convolutions are so useful when you include them in your neural networks. And then finally, let's briefly talk about how to put this all together and how you could train a convolution neural network when you have a label training set.

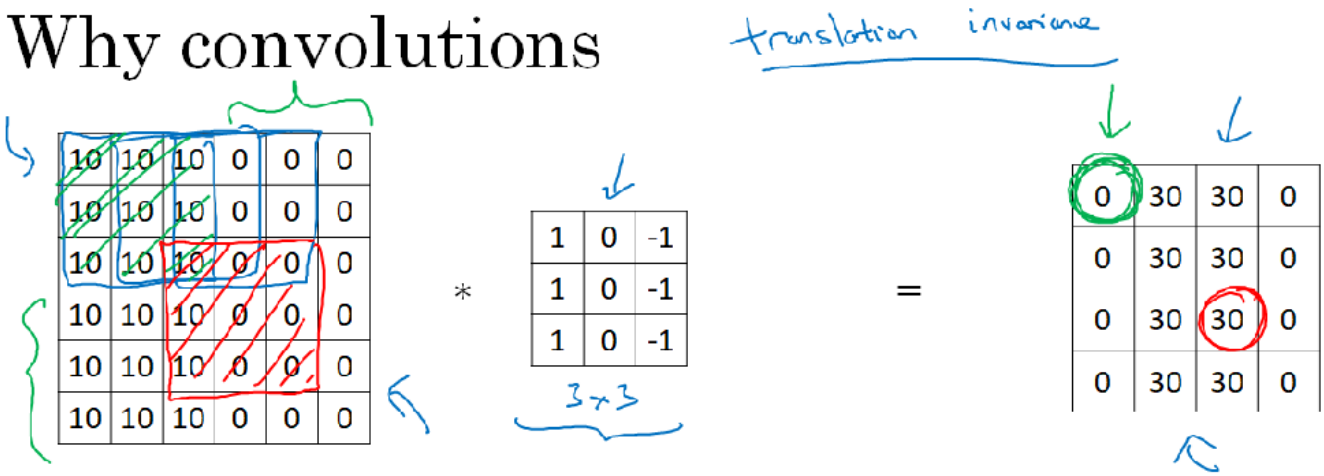
I think there are two main advantages of convolutional layers over just using fully connected layers and the advantages are **parameter sharing and sparsity of connections**. Let me illustrate with an example. Let's say you have a 32x32x3 dimensional image, and this actually comes from the example from the previous section, but let's say you use five by five filter with six filters and so, this gives you a 28 by 28 by 6 dimensional output. So, 32 by 32 by 3 is 3,072, and 28 by 28 by 6 if you multiply all those numbers is 4,704. And so, if you were to create a neural network with 3,072 units in one layer, and with 4,704 units in the next layer, and if you were to connect every one of these neurons, then the weight matrix, the number of parameters in a weight matrix would be 3,072 times 4,704 which is about 14 million. So, that's just a lot of parameters to train. And today you can train neural networks with even more parameters than 14 million, but considering that this is just a pretty small image, this is a lot of parameters to train. And of course, if this were to be 1,000 by 1,000 image, then your display matrix will just become invisibly large. But if you look at the number of parameters in this convolutional layer, each filter is five by five. So, each filter has 25 parameters, plus a bias parameter miss of 26 parameters per a filter, and you have six filters, so, the total number of parameters is that, which is equal to 156 parameters. And so, the number of parameters in this conv layer remains quite small. And the reason that a consonant has run to these small parameters is really two reasons.

Why convolutions



One is parameter sharing. And parameter sharing is motivated by the observation that feature detector such as vertical edge detector, that's useful in one part of the image is probably useful in another part of the image. And what that means is that, if you've figured out say a three by three filter for detecting vertical edges, you can then apply the same three by three filter over here, and then the next position over, and the next position over, and so on. And so, each of these feature detectors, each of these aqua's can use the same parameters in lots of different positions in your input image in order to detect say a vertical edge or some other feature. And I think this is true for low-level features like edges, as well as the higher level features, like maybe, detecting the eye that indicates a face or a cat or something there. But being with a share in this case the same nine parameters to compute all 16 of these aquas, is one of the ways the number of parameters is reduced. And it also just seems intuitive that a feature detector like a vertical edge detector computes it for the upper left-hand corner of the image. The same feature seems like it will probably be useful, has a good chance of being useful for the lower right-hand corner of the image. So, maybe you don't need to learn separate feature detectors for the upper left and the lower right-hand corners of the image. And maybe you do have a dataset where you have the upper left-hand corner and lower right-hand corner have different distributions, so, they maybe look a little bit different but they might be similar enough, they're sharing feature detectors all across the image, works just fine. The second way that consonants get away with having relatively few parameters is by having sparse connections. So, here's what I mean, if you look at the zero, this is computed via three by three convolution. And so, it depends only on this three by three inputs grid or cells. So, it is as if this output units on the right is connected only to nine out of these six by six, 36 input features. And in particular, the rest of these pixel values, all of these pixel values do not have any effects on the other output. So, that's what I mean by sparsity of connections.

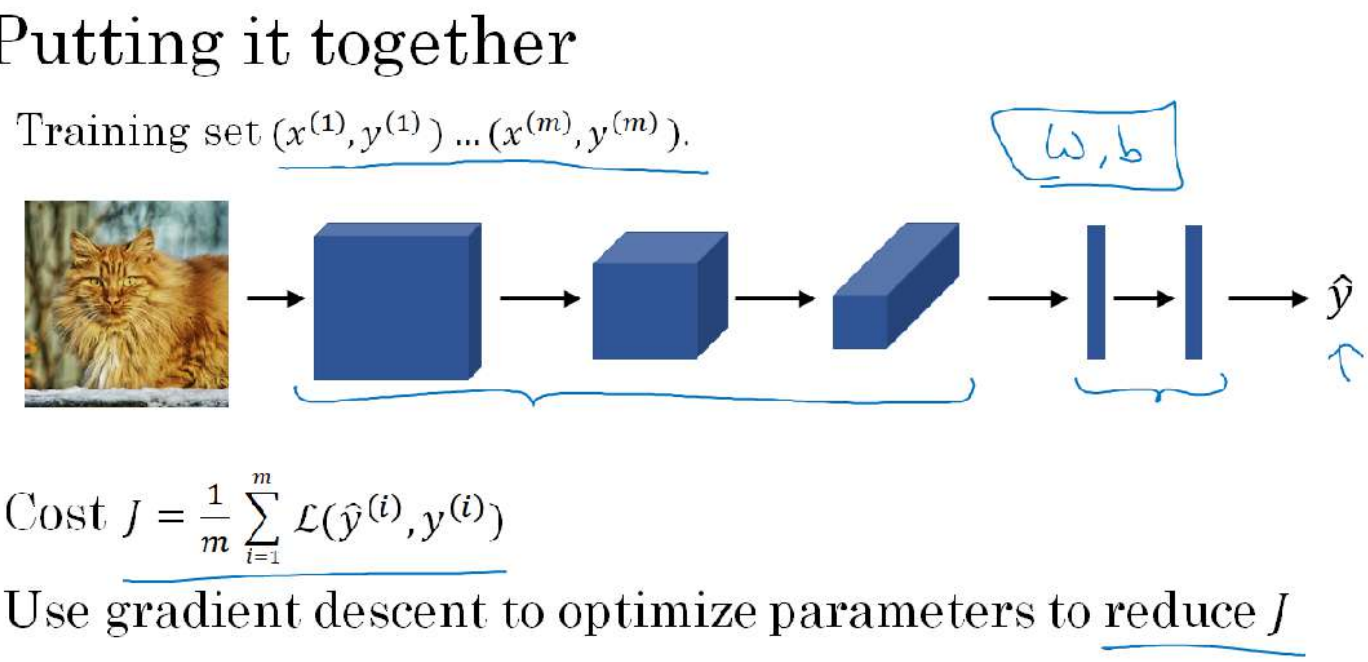
Why convolutions



Parameter sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

→ **Sparsity of connections:** In each layer, each output value depends only on a small number of inputs.

As another example, this output depends only on these nine input features. And so, it's as if only those nine input features are connected to this output, and the other pixels just don't affect this output at all. And so, through these two mechanisms, a neural network has a lot fewer parameters which allows it to be trained with smaller training cells and is less prone to be over 30. And so, sometimes you also hear about convolutional neural networks being very good at capturing translation invariance. And that's the observation that a picture of a cat shifted a couple of pixels to the right, is still pretty clearly a cat. And convolutional structure helps the neural network encode the fact that an image shifted a few pixels should result in pretty similar features and should probably be assigned the same oval label. And the fact that you are applying to same filter, knows all the positions of the image, both in the early layers and in the late layers that helps a neural network automatically learn to be more robust or to better capture the desirable property of translation invariance. So, these are maybe a couple of the reasons why convolutions or convolutional neural network work so well in computer vision. Finally, let's put it all together and see how you can train one of these networks. Let's say you want to build a cat detector and you have a labeled training sets as follows, where now, X is an image. And the y's can be binary labels, or one of K causes. And let's say you've chosen a convolutional neural network structure, may be inserted the image and then having neural convolutional and pulling layers and then some fully connected layers followed by a software output that then operates Y hat. The conv layers and the fully connected layers will have various parameters, W, as well as bias's B. And so, any setting of the parameters, therefore, lets you define a cost function similar to what we have seen in the previous courses, where we've randomly initialized parameters W and B. You can compute the cause J, as the sum of losses of the neural networks predictions on your entire training set, maybe divide it by M. So, to train this neural network, all you need to do is then use gradient descents or some of the algorithm like, gradient descent momentum, or RMSProp or Adam, or something else, in order to optimize all the parameters of the neural network to try to reduce the cost function J. And you find that if you do this, you can build a very effective cat detector or some other detector.



Week 2: Deep convolutional models: case studies

Learn about the practical tricks and methods used in deep CNNs straight from the research papers.

Learning Objectives

- Understand multiple foundational papers of convolutional neural networks
- Analyze the dimensionality reduction of a volume in a very deep network
- Understand and Implement a Residual network
- Build a deep neural network using Keras
- Implement a skip-connection in your network
- Clone a repository from github and use transfer learning

Case studies

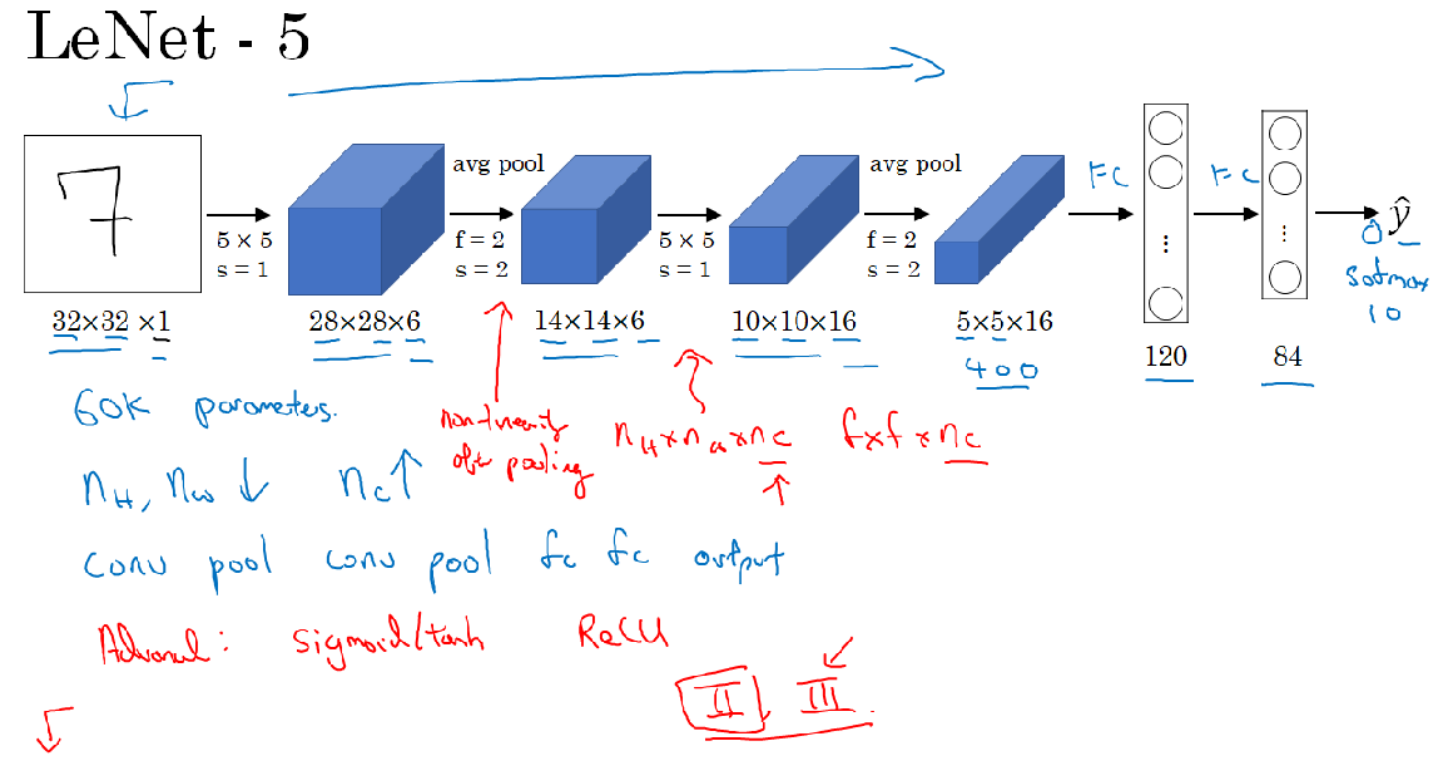
Why look at case studies?

In this section the first thing we'll do is show you a number of case studies of the factor convolutional neural networks. So why look at case studies? In last section we learned about the basic building blocks such as convolutional layers, pooling layers and fully connected layers of conv nets. It turns out a lot of the past few years of computer vision research has been on how to put together these basic building blocks to form effective convolutional neural networks and one of the best ways for you to get intuition yourself is to see some of these examples. I think just as many of you may have learned to write codes by reading other people's codes, I think that a good way to get intuition on how to build conv nets is to read or to see other examples of effective conv nets and it turns out that a neural network architecture that works well on one computer vision task often works well on other tasks as well such as maybe on your task. So if someone else is training neural network as speak it out in your network architecture is very good at recognizing cats and dogs and people but you have a different computer vision task like maybe you're trying to sell self-driving car. You might well be able to take someone else's neural network architecture and apply that to your problem. Also, in upcoming sections we'll be able to read some of the research papers from the computer vision and I hope that you might find it satisfying as well. You don't have to do this as a class but I hope you might find it satisfying to be able to read some of these seminal computer vision research paper and see yourself able to understand them. So with that, let's get started. As an outline of what we'll do in the next few sections, we'll first show you a few classic networks. The LeNet-5 network which came from, I guess, in 1980s, AlexNet which is often cited and the VGG network and these are examples of pretty effective neural networks and

some of the ideas lay the foundation for modern computer vision. Then we'll see the ResNet or conv residual network and you might have heard that neural networks are getting deeper and deeper. The ResNet neural network trained a very, very deep 152-layer neural network that has some very interesting tricks, interesting ideas how to do that effectively and then finally you also see a case study of the Inception neural network. After seeing these neural networks, I think you have much better intuition about how to built effective convolutional neural networks and even if you end up not working computer vision yourself, I think you find a lot of the ideas from some of these examples, such as ResNet Inception network, many of these ideas are cross-fertilizing on making their way into other disciplines. So even if you don't end up building computer vision applications yourself, I think you'll find some of these ideas very interesting and helpful for your work.

Classic Networks

In this section, we'll learn about some of the classic neural network architecture starting with LeNet-5, and then AlexNet, and then VGGNet. Let's take a look. Here is the LeNet-5 architecture.

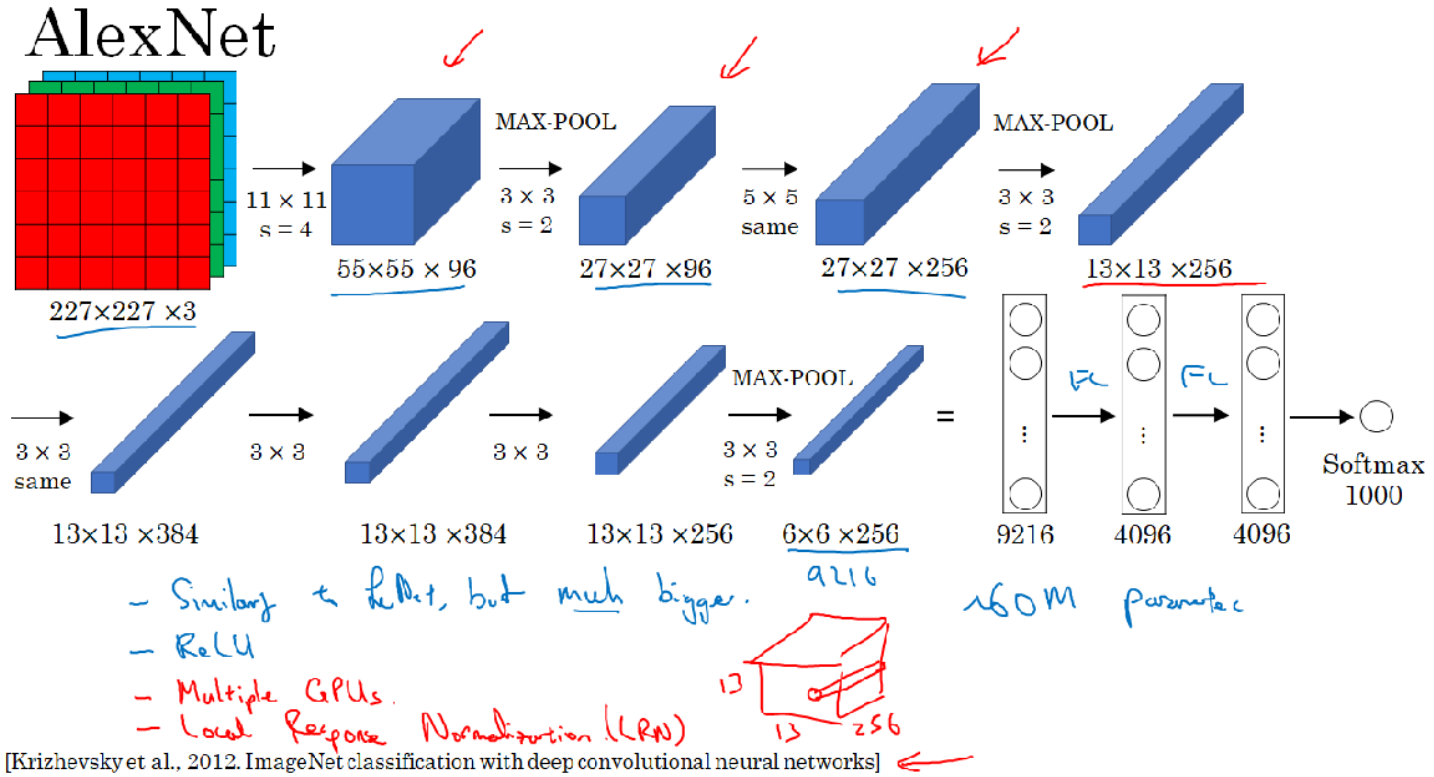


[LeCun et al., 1998. Gradient-based learning applied to document recognition]

You start off with an image which say, 32x32x1 and the goal of LeNet-5 was to **recognize handwritten digits**, so maybe an image of a digits like shown in diagram and **LeNet-5** was trained on grayscale images, which is why it's 32x32x1. In the first step, you use a set of six, 5x5 filters with a stride of one because you use six filters you end up with a 20x20x6 over there and with a stride of one and no padding, the image dimensions reduces from 32x32 down to 28x28. Then the LeNet neural network applies pooling and back then when this paper was written, people use average pooling much more. If you're building a modern variant, you probably use max pooling instead. But in this example, you average pool and with a filter width two and a stride of two, you wind up reducing the dimensions, the height and width by a factor of two, so we now end up with a 14x14x6 volume. I guess the height and width of these volumes aren't entirely drawn to scale. Now technically, if I were drawing these volumes to scale, the height and width would be stronger by a factor of two. Next, you apply another convolutional layer. This time you use a set of 16 filters, the 5x5, so you end up with 16 channels to the next volume and back when this paper was written in 1998, people didn't really use padding or you always using valid convolutions, which is why every time you apply convolutional layer, they heightened with strengths. So that's why, here, you go from 14x14 down to 10x10. Then another pooling layer, so that reduces the height

and width by a factor of two, then you end up with 5x5 over there and if you multiply all these numbers 5x5x16, this multiplies up to 400. That's 25 times 16 is 400. And the next layer is then a fully connected layer that fully connects each of these 400 nodes with every one of 120 neurons, so there's a fully connected layer and sometimes, that would draw out exclusively a layer with 400 nodes. There's a fully connected layer and then another a fully connected layer. And then the final step is it uses these essentially 84 features and uses it with one final output. I guess you could draw one more node here to make a prediction for ŷ. And ŷ took on 10 possible values corresponding to recognising each of the digits from 0 to 9. A modern version of this neural network, we'll use a **softmax** layer with a 10 way classification output. Although back then, LeNet-5 actually use a different classifier at the output layer, one that's useless today. So this neural network was small by modern standards, had about 60,000 parameters. And today, you often see neural networks with anywhere from 10 million to 100 million parameters, and it's not unusual to see networks that are literally about a thousand times bigger than this network. But one thing you do see is that as you go deeper in a network, so as you go from left to right, the height and width tend to go down. So you went from 32 by 32, to 28 to 14, to 10 to 5, whereas the number of channels does increase. It goes from 1 to 6 to 16 as you go deeper into the layers of the network. One other pattern you see in this neural network that's still often repeated today is that you might have some one or more conv layers followed by pooling layer, and then one or sometimes more than one conv layer followed by a pooling layer, and then some fully connected layers and then the outputs. So this type of arrangement of layers is quite common. Now finally, this is maybe only for those of you that want to try reading the paper. So it turns out that if you read the original paper, back then, people used sigmoid and tanh nonlinearities, and people weren't using value nonlinearities back then. So if you look at the paper, you see sigmoid and tanh referred to and there are also some funny ways about this network was wired that is funny by modern standards. So for example, you've seen how if you have a nh by nw by nc network with nc channels then you use f by f by nc dimensional filter, where everything looks at every one of these channels. But back then, computers were much slower. And so to save on computation as well as some parameters, the original LeNet-5 had some crazy complicated way where different filters would look at different channels of the input block. And so the paper talks about those details, but the more modern implementation wouldn't have that type of complexity these days. And then one last thing that was done back then I guess but isn't really done right now is that the original LeNet-5 had a non-linearity after pooling, and I think it actually uses sigmoid non-linearity after the pooling layer.

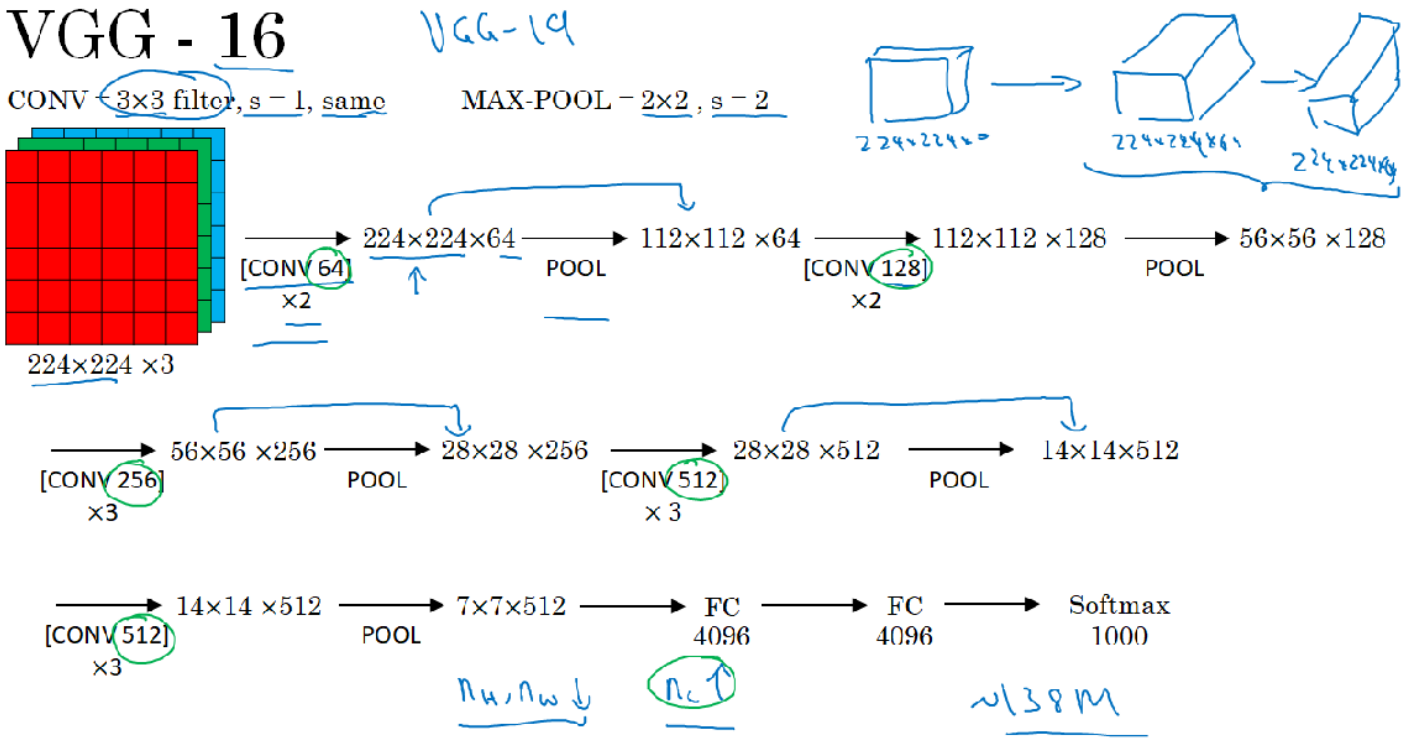
The second example of a neural network I want to show you is **AlexNet**, named after Alex Krizhevsky, who was the first author of the paper describing this work. The other author's were Ilya Sutskever and Geoffrey Hinton.



So, AlexNet input starts with 227x227x3 images and if you read the paper, the paper refers to 224x224x3 images. But if you look at the numbers, I think that the numbers make sense only of actually 227x227 and then the first layer applies a set of 96, 11x11 filters with a stride of four and because it uses a large stride of four, the dimensions shrinks to 55x55. So roughly, going down by a factor of 4 because of a large stride and then it applies max pooling with a 3x3 filter. So f equals three and a stride of two. So this reduces the volume to 27x27x96, and then it performs a 5x5 same convolution, same padding, so you end up with 27x27x276. Max pooling again, this reduces the height and width to 13. And then another same convolution, so same padding. So it's 13 by 13 by now 384 filters and then 3 by 3, same convolution again, gives you that. Then 3 by 3, same convolution, gives you that. Then max pool, brings it down to 6 by 6 by 256. If you multiply all these numbers,6 times 6 times 256, that's 9216. So we're going to unroll this into 9216 nodes. And then finally, it has a few fully connected layers. And then finally, it uses a softmax to output which one of 1000 causes the object could be. So this neural network actually had a lot of similarities to LeNet, but it was much bigger. So whereas the **LeNet-5 from previous slide had about 60,000 parameters, this AlexNet that had about 60 million parameters** and the fact that they could take pretty similar basic building blocks that have a lot more hidden units and training on a lot more data, they trained on the image that dataset that allowed it to have a just remarkable performance. Another aspect of this architecture that made it much better than LeNet was using the value activation function. And then again, just if you read the bay paper some more advanced details that you don't really need to worry about if you don't read the paper, one is that, when this paper was written, GPUs was still a little bit slower, so it had a complicated way of training on two GPUs. And the basic idea was that, a lot of these layers was actually split across two different GPUs and there was a thoughtful way for when the two GPUs would communicate with each other. And the paper also, the original AlexNet architecture also had another set of a layer called a **Local Response Normalization** and this type of layer isn't really used much, which is why I didn't talk about it. But the basic idea of Local Response Normalization is, if you look at one of these blocks, one of these volumes that we have on top, let's say for the sake of argument, this one, 13x13x256, what Local Response Normalization, (LRN) does, is you look at one position. So one position height and width, and look down this across all the channels, look at all 256 numbers and normalize them. And the motivation for this Local Response Normalization was that for each position in this 13x13 image, maybe you don't want too many neurons with a very high

activation. But subsequently, many researchers have found that this doesn't help that much so this is one of those ideas I guess I'm drawing in red because it's less important for you to understand this one. And in practice, I don't really use local response normalizations really in the networks language trained today. So if you are interested in the history of deep learning, I think even before AlexNet, deep learning was starting to gain attraction in speech recognition and a few other areas, but it was really just paper that convinced a lot of the computer vision community to take a serious look at deep learning to convince them that deep learning really works in computer vision. And then it grew on to have a huge impact not just in computer vision but beyond computer vision as well. And if you want to try reading some of these papers yourself and you really don't have to for this course, but if you want to try reading some of these papers, this one is one of the easier ones to read so this might be a good one to take a look at. So whereas AlexNet had a relatively complicated architecture, there's just a lot of hyperparameters, right? Where you have all these numbers that Alex Krizhevsky and his co-authors had to come up with.

Let me show you a third and final example on this section called the **VGG** or **VGG-16** network and a remarkable thing about the VGG-16 net is that they said, instead of having so many hyperparameters, let's use a much simpler network where you focus on just having conv-layers that are just three-by-three filters with a stride of one and always use same padding and make all your max pulling layers two-by-two with a stride of two. And so, one very nice thing about the VGG network was it really simplified this neural network architectures. So, let's go through the architecture.



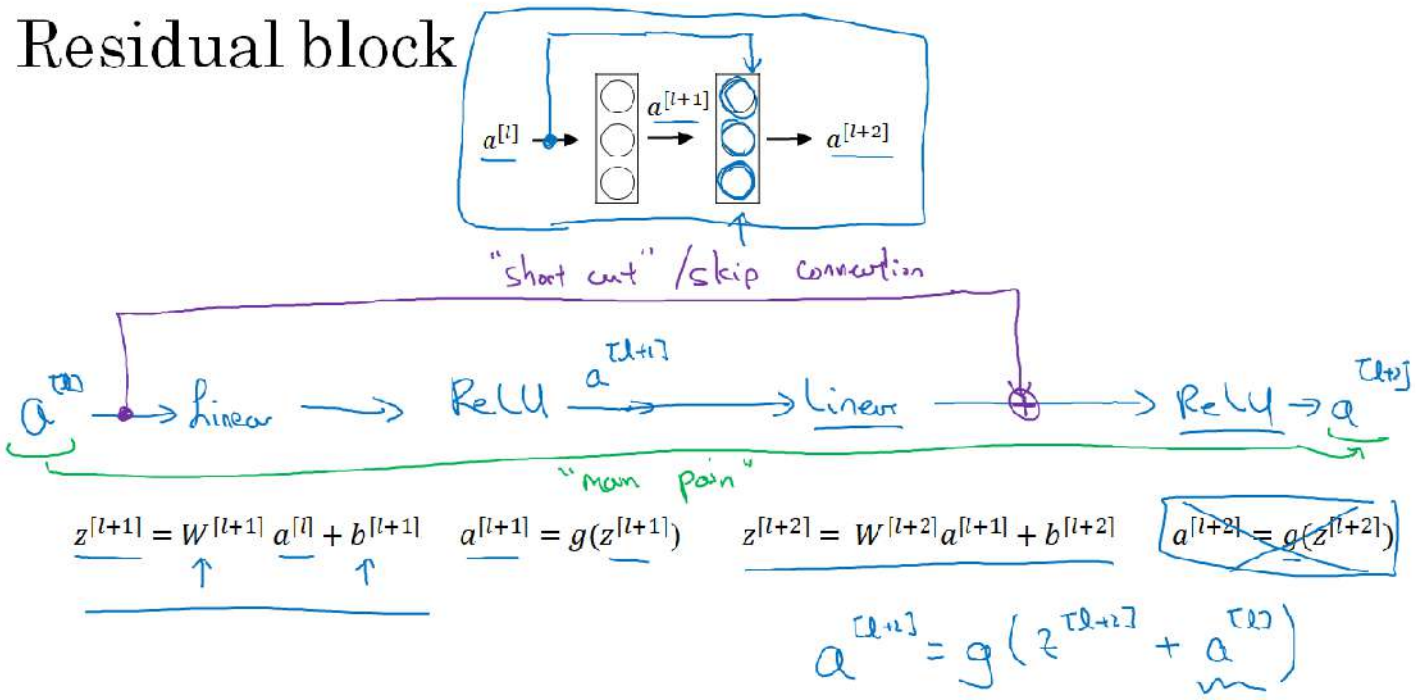
[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

So, you solve up with an image for them and then the first two layers are convolutions, which are therefore these three-by-three filters and in the first two layers use 64 filters. You end up with a 224x224 because using same convolutions and then with 64 channels. So because VGG-16 is a relatively deep network, am going to not draw all the volumes here. So what this little picture denotes is what we would previously have drawn as this 224x224x3 and then a convolution that results in I guess a 224x224x64 is to be drawn as a deeper volume, and then another layer that results in 224x224x64. So this conv64 times two represents that you're doing two conv-layers with 64 filters and as I mentioned earlier, the filters are always three-by-three with a stride of one and they are always same convolutions. So rather than drawing all these volumes, am just going to use text to represent this network. Next, then uses are pooling layer, so the pooling layer will reduce. I think it goes from 224x224 down to what? Right. Goes to 112x112x64 and then it has a

couple more conv-layers. So this means it has 128 filters and because these are the same convolutions, let's see what is the new dimension. Right? It will be 112 by 112 by 128 and then pulling layer so you can figure out what's the new dimension of that. And now, three conv-layers with 256 filters to the pulling layer and then a few more conv-layers, pooling layer, more conv-layers, pooling layer. And then it takes this final 7x7x512 these in to fully connected layer, fully connected with four thousand ninety six units and then a softmax output one of a thousand classes. By the way, the 16 in the VGG-16 refers to the fact that this has 16 layers that have weights and this is a pretty large network, this network has a total of about 138 million parameters. And that's pretty large even by modern standards. But the simplicity of the VGG-16 architecture made it quite appealing. You can tell his architecture is really quite uniform. There is a few conv-layers followed by a pooling layer, which reduces the height and width, right? So the pooling layers reduce the height and width. You have a few of them here. But then also, if you look at the number of filters in the conv-layers, here you have 64 filters and then you double to 128 double to 256 doubles to 512. And then I guess the authors thought 512 was big enough and did double on the game here. But this sort of roughly doubling on every step, or doubling through every stack of conv-layers was another simple principle used to design the architecture of this network. And so I think the relative uniformity of this architecture made it quite attractive to researchers. The main downside was that it was a pretty large network in terms of the number of parameters you had to train. And if you read the literature, you sometimes see people talk about the VGG-19, that is an even bigger version of this network. And you could see the details in the paper cited at the bottom by Karen Simonyan and Andrew Zisserman. But because VGG-16 does almost as well as VGG-19. A lot of people will use VGG-16. But the thing I liked most about this was that, this made this pattern of how, as you go deeper and height and width goes down, it just goes down by a factor of two each time for the pulling layers whereas the number of channels increases. And here roughly goes up by a factor of two every time you have a new set of conv-layers. So by making the rate at which it goes down and that go up very systematic, I thought this paper was very attractive from that perspective. So that's it for the three classic architecture's. If you want, you should really now read some of these papers. I recommend starting with the AlexNet paper followed by the VGG net paper and then the LeNet paper is a bit harder to read but it is a good classic once you go over that. But next, let's go beyond these classic networks and look at some even more advanced, even more powerful neural network architectures.

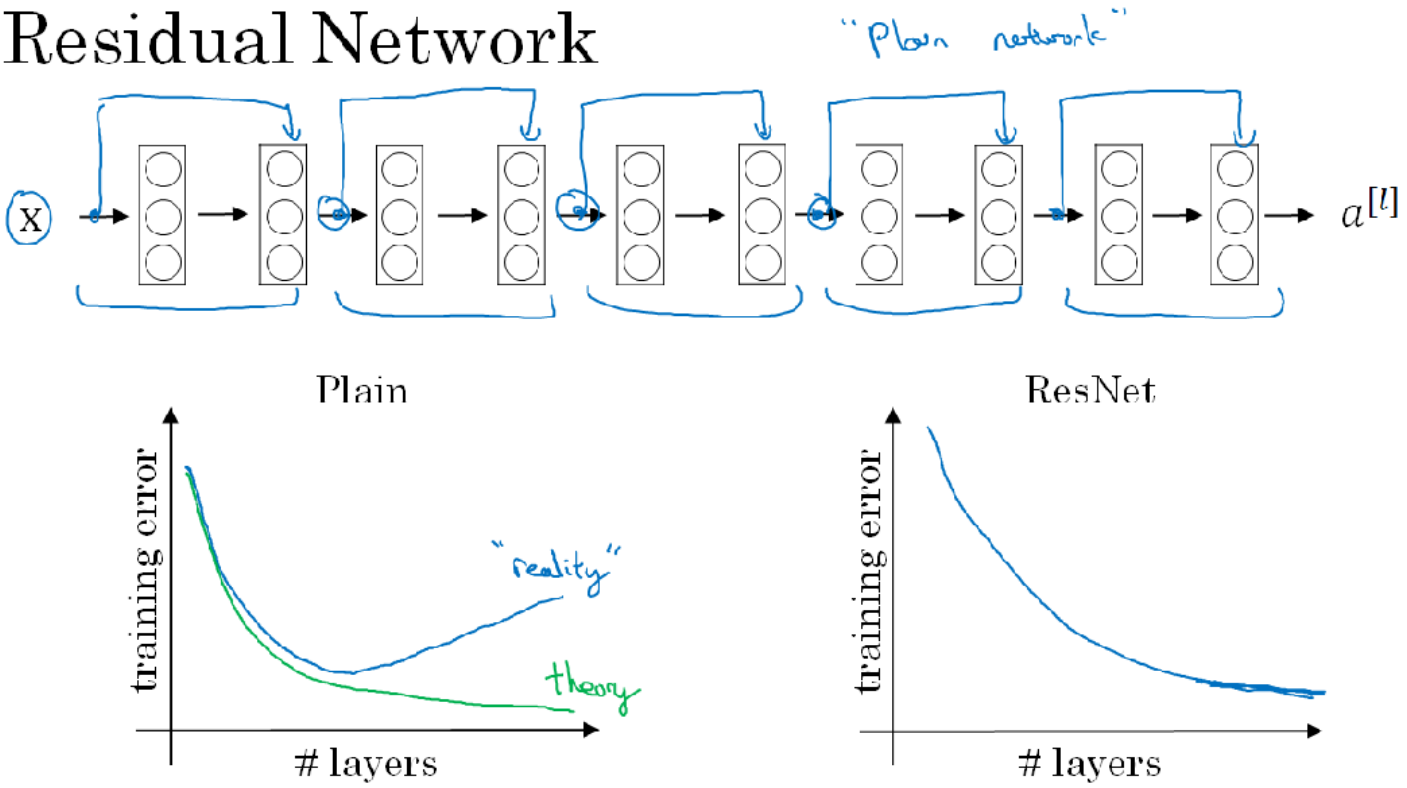
ResNets

Very, very deep neural networks are difficult to train because of vanishing and exploding gradient types of problems. In this section, we'll learn about skip connections which allows you to take the activation from one layer and suddenly feed it to another layer even much deeper in the neural network and using that, we'll build ResNet which enables you to train very, very deep networks. Sometimes even networks of over 100 layers. Let's take a look. ResNets are built out of something called a residual block, let's first describe what that is.



[He et al., 2015. Deep residual networks for image recognition]

Here are two layers of a neural network where you start off with some activations in layer $a^{[l]}$, then goes $a^{[l+1]}$ and then deactivation two layers later is $a^{[l+2]}$. So let's to through the steps in this computation you have $a^{[l]}$, and then the first thing you do is you apply this linear operator to it, which is governed by this equation. So you go from $a^{[l]}$ to compute $z^{[l+1]}$ by multiplying by the weight matrix and adding that bias vector. After that, you apply the ReLU nonlinearity, to get $a^{[l+1]}$ and that's governed by this equation where $a^{[l+1]}$ is $g(z^{[l+1]})$. Then in the next layer, you apply this linear step again, so is governed by that equation. So this is quite similar to this equation we saw on the left. And then finally, you apply another ReLU operation which is now governed by that equation where G here would be the ReLU nonlinearity and this gives you $a^{[l+2]}$. So in other words, for information from $a^{[l]}$ to flow to $a^{[l+2]}$, it needs to go through all of these steps which I'm going to call the main path of this set of layers. In a residual net, we're going to make a change to this. We're going to take $a^{[l]}$, and just first forward it, copy it, match further into the neural network to here, and just at $a^{[l]}$, before applying to non-linearity, the ReLU non-linearity and I'm going to call this the shortcut. So rather than needing to follow the main path, the information from $a^{[l]}$ can now follow a shortcut to go much deeper into the neural network. and what that means is that this last equation goes away and we instead have that the output $a^{[l+2]}$ is the ReLU non-linearity g applied to $z^{[l+2]}$ as before, but now plus $a^{[l]}$. So, the addition of this $a^{[l]}$ here, it makes this a residual block. And in pictures, you can also modify this picture on top by drawing this picture shortcut to go here. And we are going to draw it as it going into this second layer here because the short cut is actually added before the ReLU non-linearity. So each of these nodes here, where there applies a linear function and a ReLU. So $a^{[l]}$ is being injected after the linear part but before the ReLU part and sometimes instead of a term short cut, you also hear the term skip connection, and that refers to $a^{[l]}$ just skipping over a layer or kind of skipping over almost two layers in order to process information deeper into the neural network. So, what the inventors of ResNet, so that'll will be Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. **What they found was that using residual blocks allows you to train much deeper neural networks. And the way you build a ResNet is by taking many of these residual blocks, blocks like these, and stacking them together to form a deep network.** So, let's look at this network. This is not the residual network, this is called as a plain network. This is the terminology of the ResNet paper.



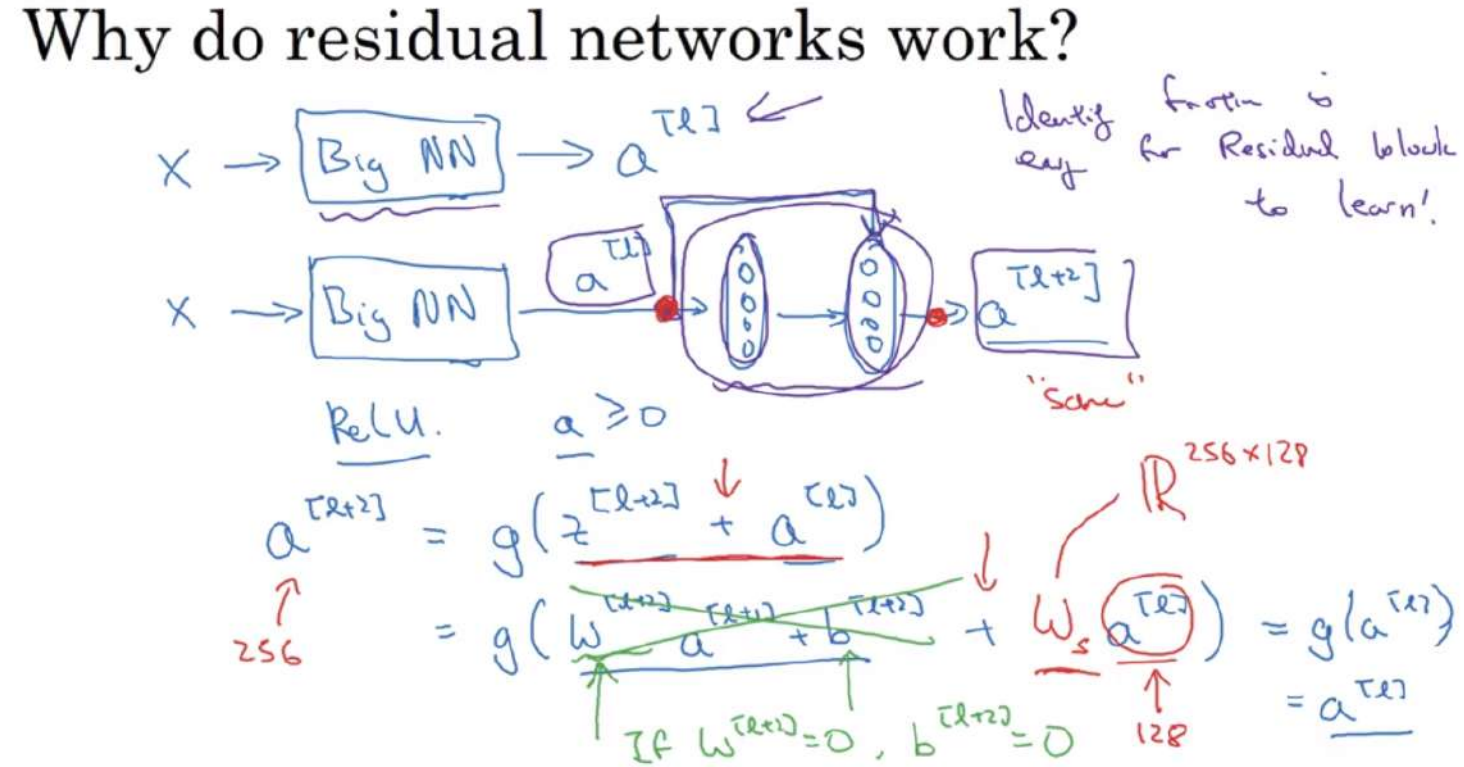
[He et al., 2015. Deep residual networks for image recognition]

To turn this into a ResNet, what you do is you add all those skip connections although those short like a connections like so. So every two layers ends up with that additional change that we saw on the previous slide to turn each of these into residual block. So this picture shows five residual blocks stacked together, and this is a residual network and it turns out that if you use your standard optimization algorithm such as a gradient descent or one of the fancier optimization algorithms to the train or plain network. So without all the extra residual, without all the extra short cuts or skip connections I just drew in. Empirically, you find that as you increase the number of layers, the training error will tend to decrease after a while but then they'll tend to go back up. And in theory as you make a neural network deeper, it should only do better and better on the training set. So, in theory, having a deeper network should only help. But in practice or in reality, having a plain network, so no ResNet, having a **plain network that is very deep means that all your optimization algorithm just has a much harder time training**. And so, in reality, your training error gets worse if you pick a network that's too deep. But what happens with ResNet is that even as the number of layers gets deeper, you can have the performance of the training error kind of keep on going down. Even if we train a network with over a hundred layers. And then now some people experimenting with networks of over a thousand layers although I don't see that it used much in practice yet. But by taking these activations be it X of these intermediate activations and allowing it to go much deeper in the neural network, this really helps with the vanishing and exploding gradient problems and allows you to train much deeper neural networks without really appreciable loss in performance, and maybe at some point, this will plateau, this will flatten out, and it doesn't help that much deeper and deeper networks. **But ResNet is not even effective at helping train very deep networks**.

Why ResNet Work

So, why do ResNets work so well? Let's go through one example that illustrates why ResNets work so well, at least in the sense of how you can make them deeper and deeper without really hurting your ability to at least get them to do well on the training set. And hopefully as you've understood from the third course in this sequence, doing well on the training set is usually a prerequisite to doing well on your hold up or on your depth or on your test sets. So, being able to

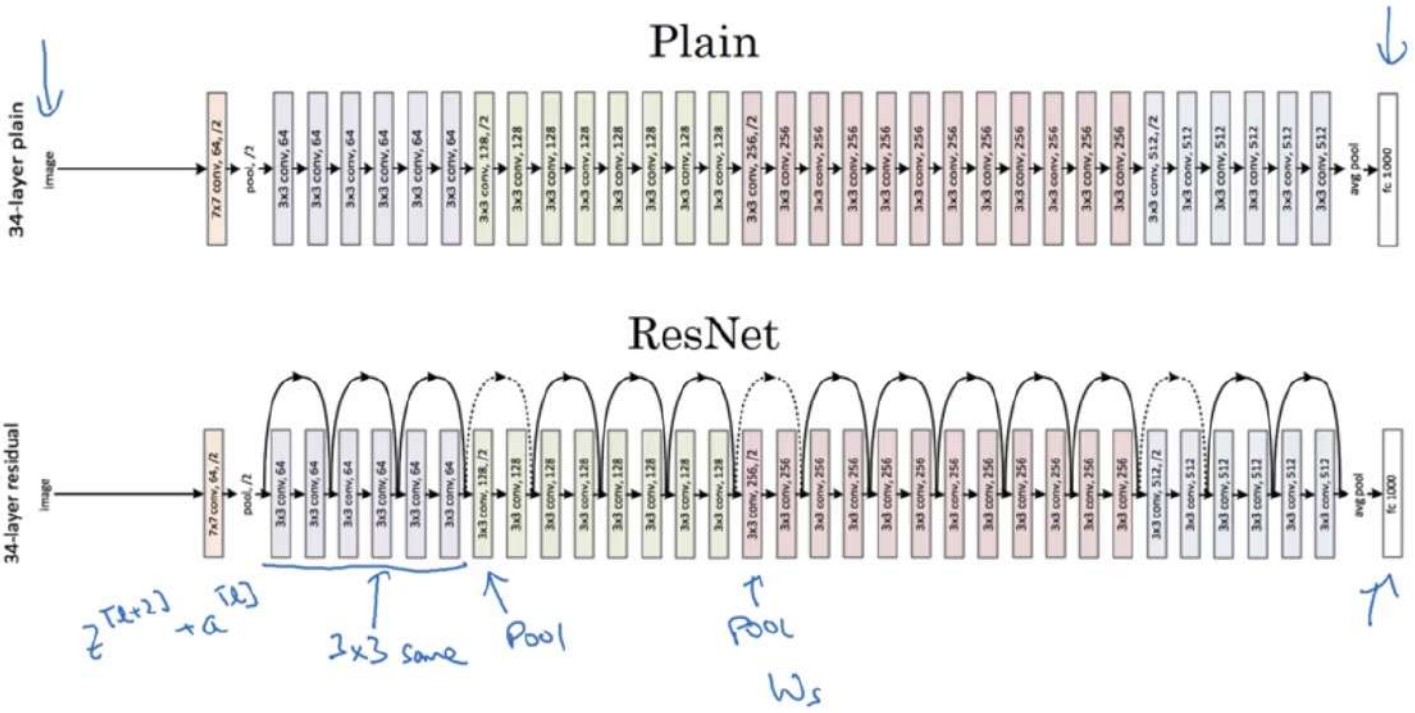
at least train ResNet to do well on the training set is a good first step toward that. Let's look at an example. What we saw on the last section was that if you make a network deeper, it can hurt your ability to train the network to do well on the training set and that's why sometimes you don't want a network that is too deep. But this is not true or at least is much less true when you training a ResNet. So let's go through an example.



I think the main reason the residual network works is that it's so easy for these extra layers to learn the identity function that you're kind of guaranteed that it doesn't hurt performance and then a lot the time you maybe get lucky and then even helps performance. At least is easier to go from a decent baseline of not hurting performance and then great in decent can only improve the solution from there.

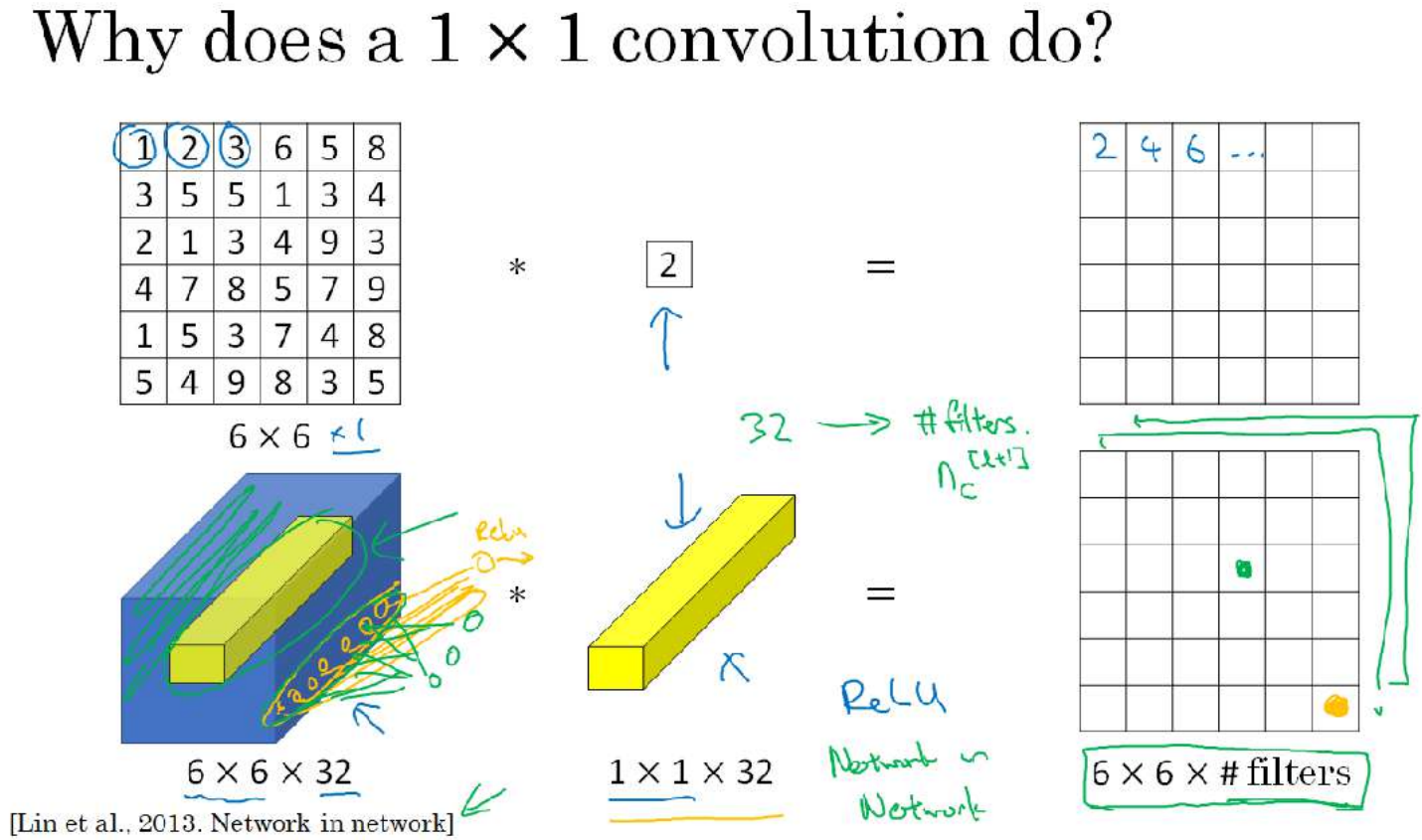
So finally, let's take a look at ResNets on images. So these are images I got from the paper by Harlow. This is an example of a plain network and in which you input an image and then have a number of conv layers until eventually you have a softmax output at the end. To turn this into a ResNet, you add those extra skip connections. And I'll just mention a few details, there are a lot of three by three convolutions here and most of these are three by three same convolutions and that's why you're adding equal dimension feature vectors.

ResNet



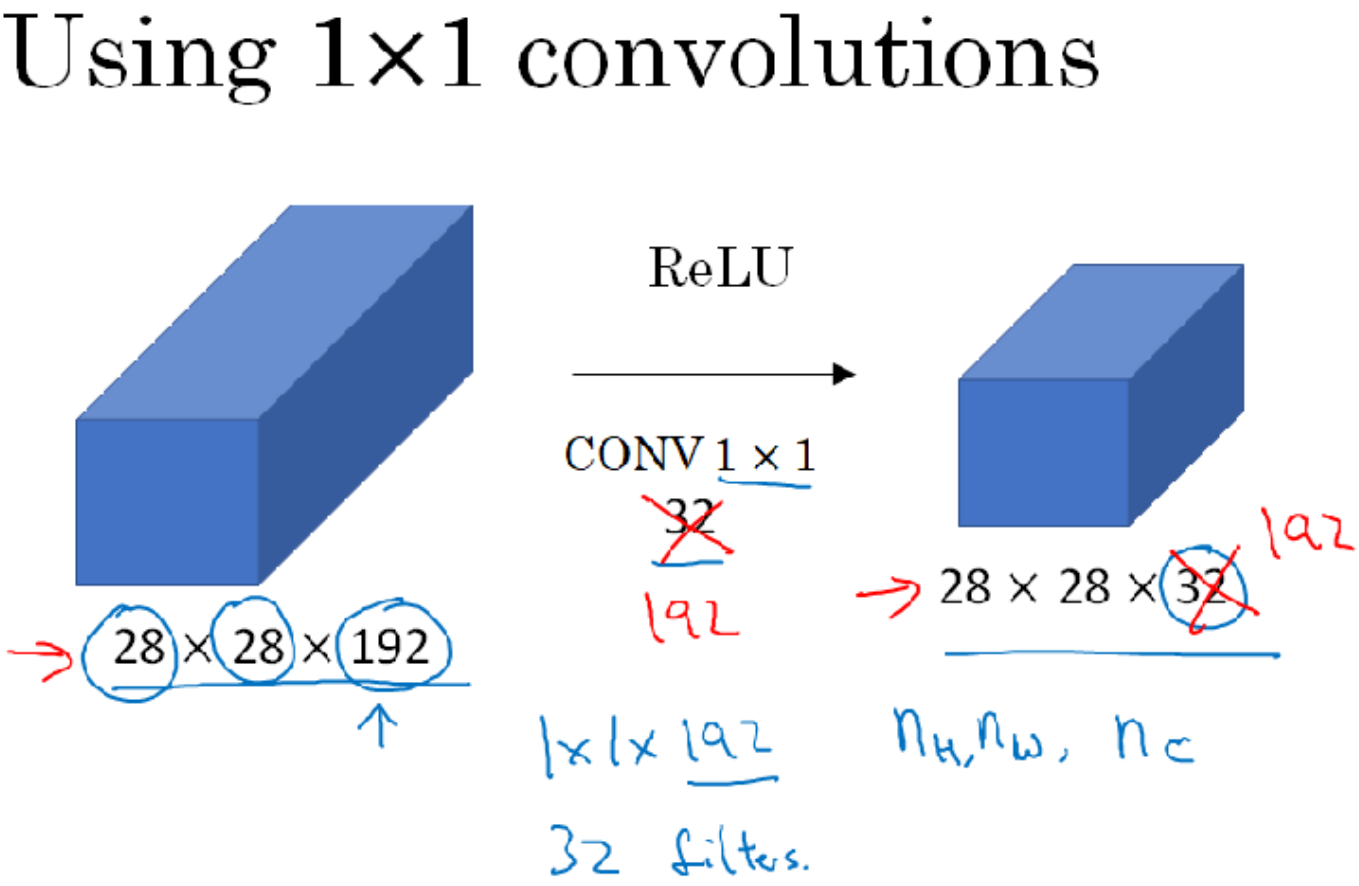
Networks in Networks and 1x1 Convolutions

In terms of designing content architectures, one of the ideas that really helps is using a 1x1 convolution. Now, you might be wondering, what does a 1x1 convolution do? Isn't that just multiplying by numbers? That seems like a funny thing to do. Turns out it's not quite like that.



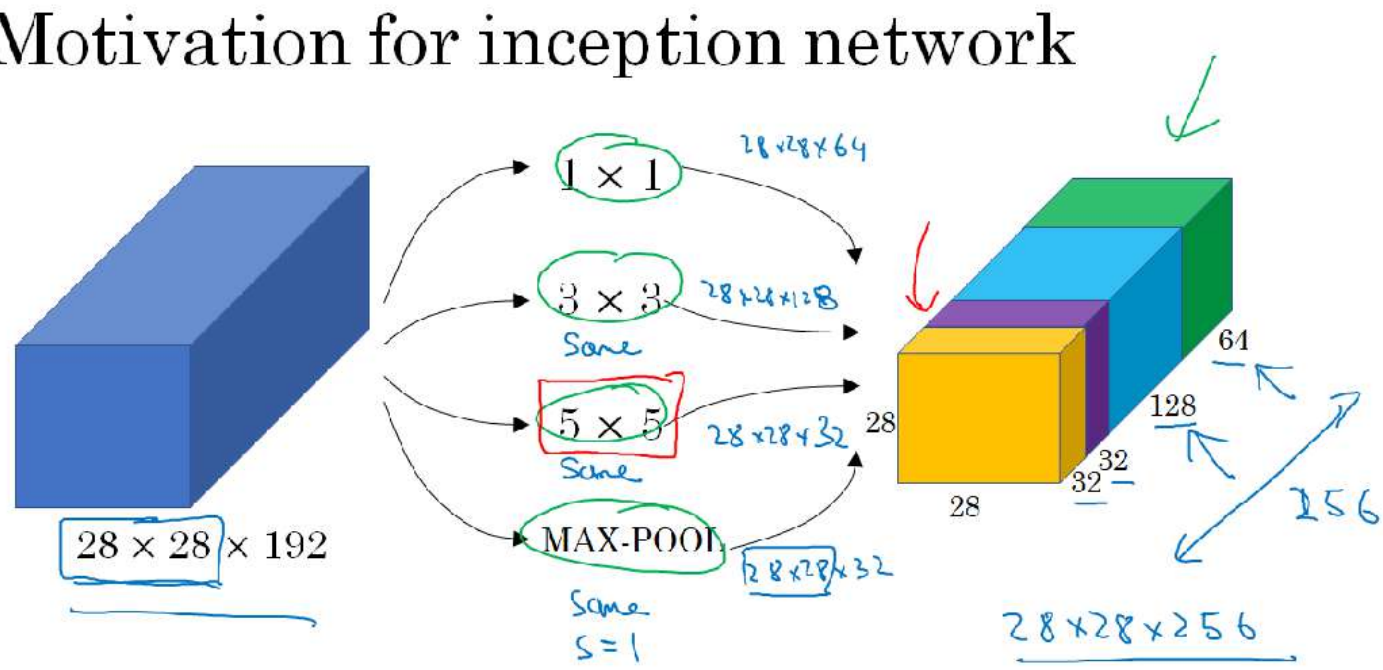
Let's take a look at the diagram. So you'll see one by one filter, we'll put in number two there, and if you take the six by six image, six by six by one and convolve it with this one by one by one

filter, you end up just taking the image and multiplying it by two. So, one, two, three ends up being two, four, six, and so on. And so, a convolution by a one by one filter, doesn't seem particularly useful. You just multiply it by some number. But that's the case of six by six by one channel images. If you have a 6 by 6 by 32 instead of by 1, then a convolution with a 1 by 1 filter can do something that makes much more sense and in particular, what a one by one convolution will do is it will look at each of the 36 different positions here, and it will take the element wise product between 32 numbers on the left and 32 numbers in the filter. And then apply a ReLU non-linearity to it after that. So, to look at one of the 36 positions, maybe one slice through this value, you take these 36 numbers multiply it by 1 slice through the volume like that, and you end up with a single real number which then gets plotted in one of the outputs like that. And in fact, one way to think about the 32 numbers you have in this 1x1x32 filters is that, it's as if you have neuron that is taking as input, 32 numbers multiplying each of these 32 numbers in one slice of the same position heightened with by these 32 different channels, multiplying them by 32 weights and then applying a ReLU non-linearity to it and then outputting the corresponding thing over there. And more generally, if you have not just one filter, but if you have multiple filters, then it's as if you have not just one unit, but multiple units, taken as input all the numbers in one slice, and then building them up into an output of six by six by number of filters. So one way to think about the **1x1 convolution is that, it is basically having a fully connected neuron network, that applies to each of the 62 different positions** and what does fully connected neural network does? Is it puts 32 numbers and outputs number of filters outputs. So I guess the point on notation, this is really a $\mathbf{n_C^{[l+1]}}$, if that's the next layer and by doing this at each of the 36 positions, each of the six by six positions, you end up with an output that is six by six by the number of filters. and this can carry out a pretty non-trivial computation on your input volume and this idea is often called a 1x1 convolution but it's sometimes also called Network in Network, and is described in this paper, by Min Lin, Qiang Chen, and Schuicheng Yan. And even though the details of the architecture in this paper aren't used widely, this idea of a one by one convolution or this sometimes called Network in Network idea has been very influential, has influenced many other neural network architectures including the inception network which we'll see in the next section. But to give you an example of where one by one convolution is useful, here's something you could do with it. Let's say you have a 28 by 28 by 192 volume. If you want to shrink the height and width, you can use a pooling layer. So we know how to do that. But one of a number of channels has gotten too big and we want to shrink that. How do you shrink it to a 28 by 28 by 32 dimensional volume? Well, what you can do is use 32 filters that are one by one. And technically, each filter would be of dimension 1 by 1 by 192, because the number of channels in your filter has to match the number of channels in your input volume, but you use 32 filters and the output of this process will be a 28 by 28 by 32 volume. So this is a way to let you shrink nc as well, whereas pooling layers, I used just to shrink nH and nW, the height and width these volumes. and we'll see later how this idea of one by one convolutions allows you to shrink the number of channels and therefore, save on computation in some networks. But of course, if you want to keep the number of channels at 192, that's fine too and the effect of the one by one convolution is it just adds non-linearity. It allows you to learn the more complex function of your network by adding another layer that inputs 28 by 28 by 192 and outputs 28 by 28 by 192. So, that's how a one by one convolutional layer is actually doing something pretty non-trivial and adds non-linearity to your neural network and allow you to decrease or keep the same or if you want, increase the number of channels in your volumes.



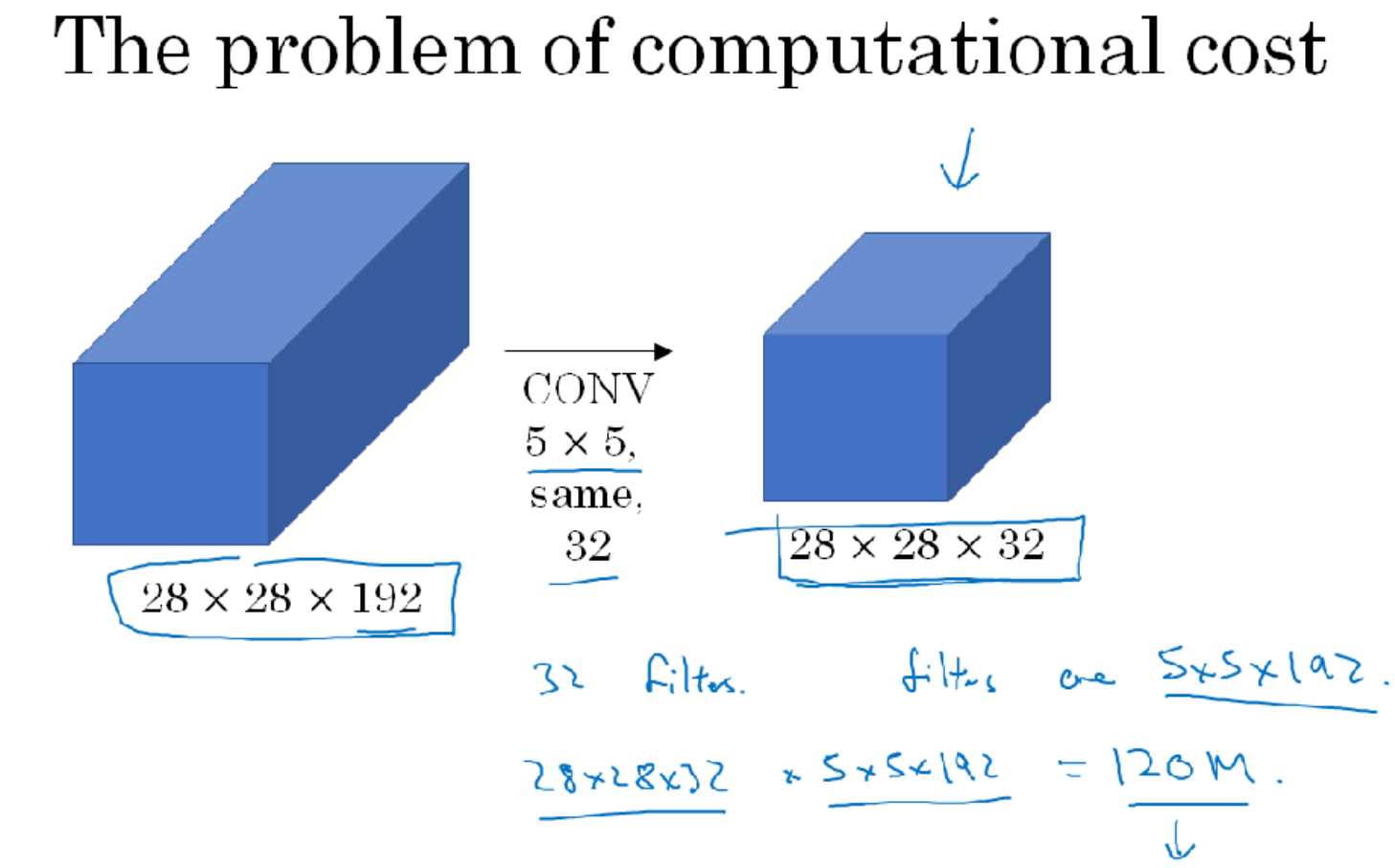
Inception Network Motivation

When designing a layer for a ConvNet, you might have to pick, do you want a 1x3 filter, or 3x3, or 5x5, or do you want a pooling layer? What the inception network does is it says, why should you do them all? and this makes the network architecture more complicated, but it also works remarkably well.



Let's see how this works. Let's say for the sake of example that you have inputted a 28 by 28 by 192 dimensional volume. So what the inception network or what an inception layer says is,

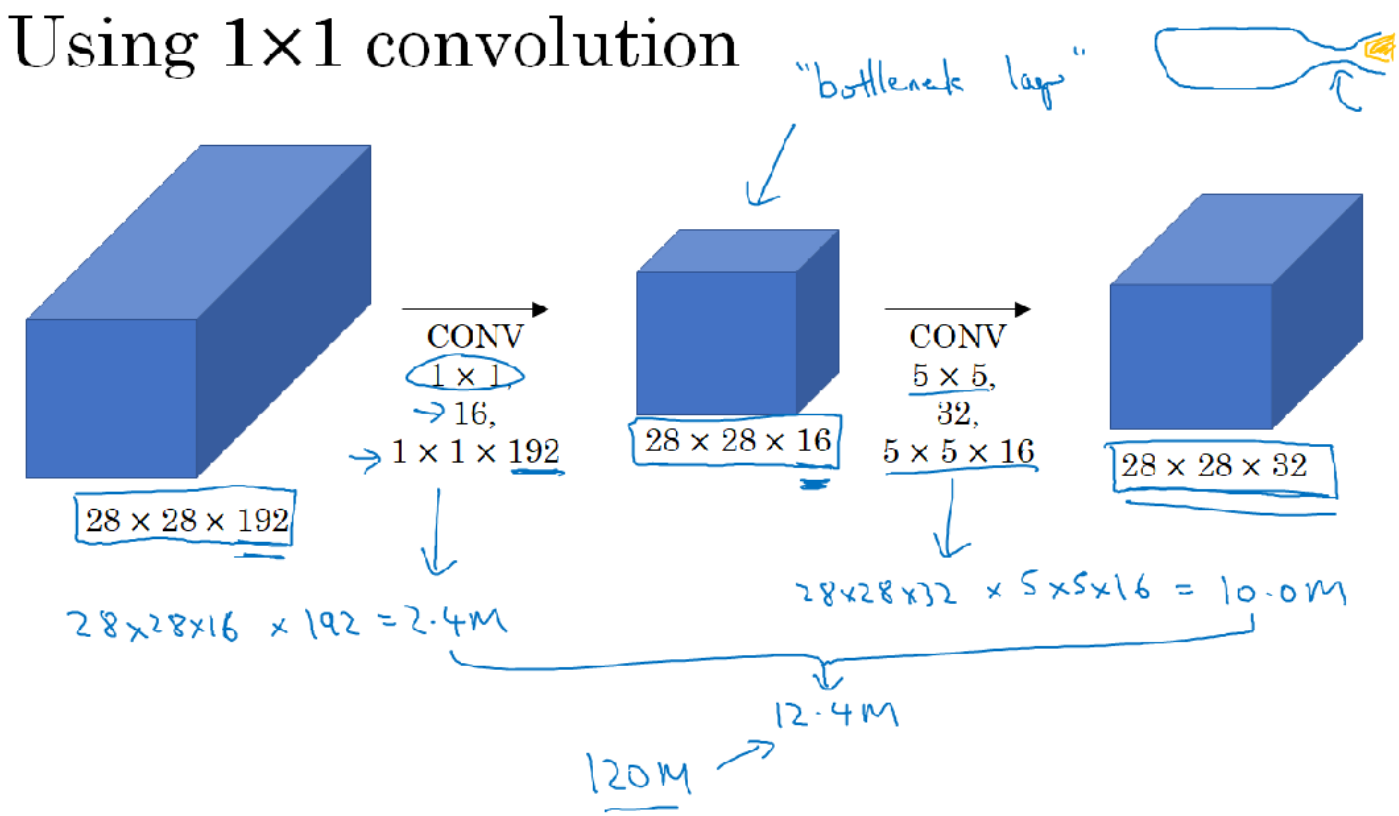
instead choosing what filter size you want in a Conv layer, or even do you want a convolutional layer or a pooling layer? Let's do them all. So what if you can use a 1 by 1 convolution, and that will output a 28 by 28 by something. Let's say 28 by 28 by 64 output, and you just have a volume there. But maybe you also want to try a 3 by 3 and that might output a 20 by 20 by 128. And then what you do is just stack up this second volume next to the first volume. And to make the dimensions match up, let's make this a same convolution. So the output dimension is still 28 by 28, same as the input dimension in terms of height and width. But 28 by 28 by in this example 128. Maybe a 5 by 5 filter works better. So let's do that too and have that output a 28 by 28 by 32. And again you use the same convolution to keep the dimensions the same. And maybe you don't want to convolutional layer. Let's apply pooling, and that has some other output and let's stack that up as well. And here pooling outputs 28 by 28 by 32. Now in order to make all the dimensions match, you actually need to use padding for max pooling. So this is an unusual formal pooling because if you want the input to have a higher than 28 by 28 and have the output, you'll match the dimension everything else also by 28 by 28, then you need to use the same padding as well as a stride of one for pooling. So this detail might seem a bit funny to you now, but let's keep going. And we'll make this all work later. But with a inception module like this, you can input some volume and output. In this case I guess if you add up all these numbers, 32 plus 32 plus 128 plus 64, that's equal to 256. So you will have one inception module input 28 by 28 by 129, and output 28 by 28 by 256. And this is the heart of the inception network which is due to Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. And the basic idea is that instead of you needing to pick one of these filter sizes or pooling you want and committing to that, you can do them all and just concatenate all the outputs, and let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants. Now it turns out that there is a problem with the inception layer as we've described it here, which is computational cost. Let's figure out what's the computational cost of this 5 by 5 filter resulting in this block over here.



So just focusing on the 5 by 5 pot on the previous diagram, we had as input a 28 by 28 by 192 block, and you implement a 5 by 5 same convolution of 32 filters to output 28 by 28 by 32. On the

previous diagram we had drawn this as a thin purple slide. So I'm just going draw this as a more normal looking blue block here. So let's look at the computational costs of outputting this 20 by 20 by 32. So you have 32 filters because the outputs has 32 channels, and each filter is going to be 5 by 5 by 192. And so the output size is 20 by 20 by 32, and so you need to compute 28 by 28 by 32 numbers. And for each of them you need to do these many multiplications, right? 5 by 5 by 192. So the total number of multiplies you need is the number of multiplies you need to compute each of the output values times the number of output values you need to compute. And if you multiply all of these numbers, this is equal to 120 million. And so, while you can do 120 million multiplies on the modern computer, this is still a pretty expensive operation. On the next diagram you see how using the idea of 1 by 1 convolutions, which you learnt about in the previous section, you'll be able to reduce the computational costs by about a factor of 10. To go from about 120 million multiplies to about one tenth of that. So please remember the number 120 so you can compare it with what you see on the next diagram, 120 million. Here is an alternative architecture for inputting 28 by 28 by 192, and outputting 28 by 28 by 32, which is falling. You are going to input the volume, use a 1 by 1 convolution to reduce the volume to 16 channels instead of 192 channels, and then on this much smaller volume, run your 5 by 5 convolution to give you your final output. So notice the input and output dimensions are still the same. You input 28 by 28 by 192 and output 28 by 28 by 32, same as the previous diagram. But what we've done is we're taking this huge volume we had on the left, and we shrunk it to this much smaller intermediate volume, which only has 16 instead of 192 channels. Sometimes this is called a **bottleneck layer**. I guess because a bottleneck is usually the smallest part of something. So I guess if you have a glass bottle that looks like this, then you know this is I guess where the cork goes. And then the bottleneck is the smallest part of this bottle. So in the same way, the bottleneck layer is the smallest part of this network. We shrink the representation before increasing the size again. Now let's look at the computational costs involved. To apply this 1 by 1 convolution, we have 16 filters. Each of the filters is going to be of dimension 1 by 1 by 192, this 192 matches that 192. And so the cost of computing this 28 by 28 by 16 volumes is going to be well, you need these many outputs, and for each of them you need to do 192 multiplications. I could have written 1 times 1 times 192, right? Which is this. And if you multiply this out, this is 2.4 million, it's about 2.4 million. How about the second? So that's the cost of this first convolutional layer. The cost of this second convolutional layer would be that well, you have these many outputs. So 28 by 28 by 32. And then for each of the outputs you have to apply a 5 by 5 by 16 dimensional filter. And so by 5 by 5 by 16. And you multiply that out is equals to 10.0. And so the total number of multiplications you need to do is the sum of those which is 12.4 million multiplications and you compare this with what we had on the previous slide, you reduce the computational cost from about 120 million multiplies, down to about one tenth of that, to 12.4 million multiplications and the number of additions you need to do is about very similar to the number of multiplications you

need to do. So that's why I'm just counting the number of multiplications.

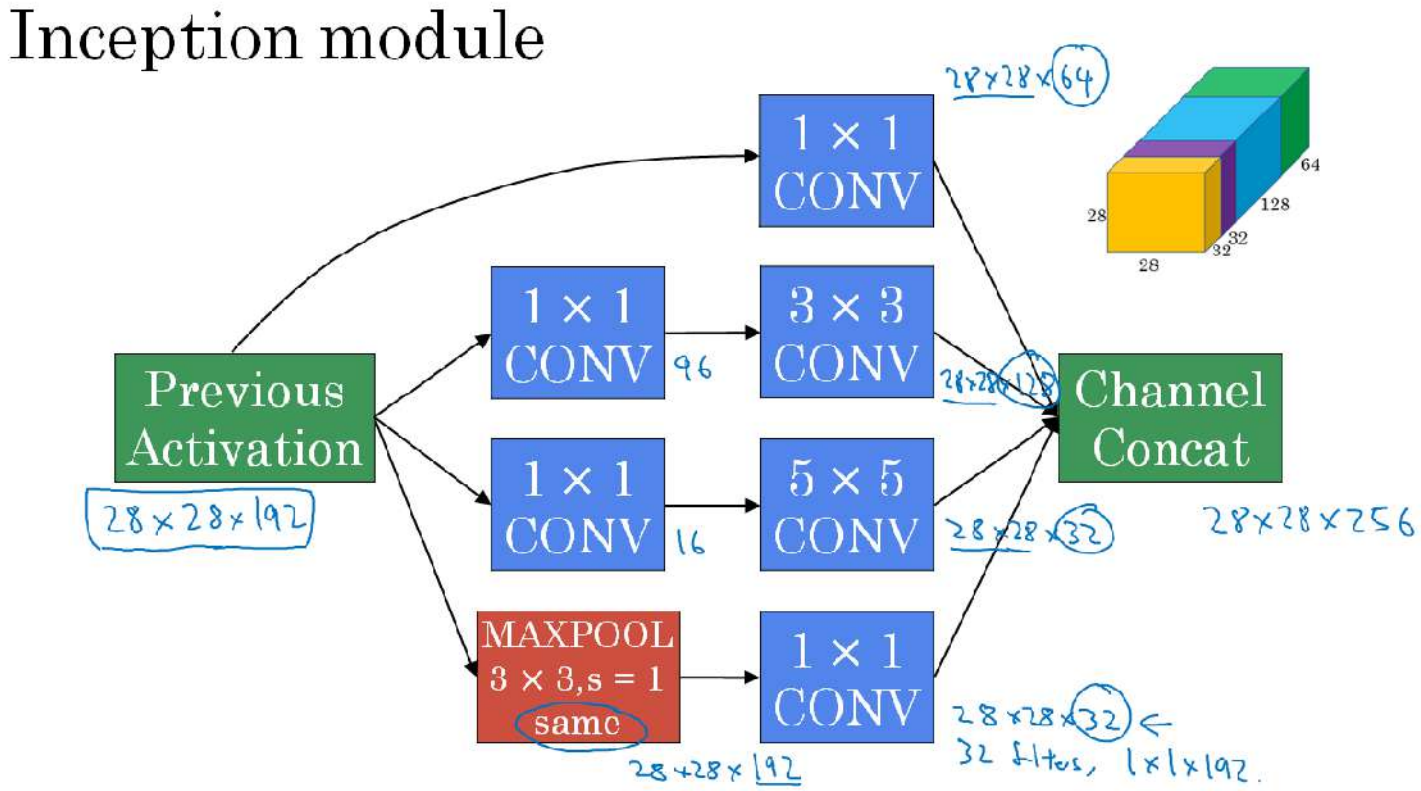


So to summarize, if you are building a layer of a neural network and you don't want to have to decide, do you want a 1 by 1, or 3 by 3, or 5 by 5, or pooling layer, the inception module let's you say let's do them all, and let's concatenate the results. And then we run to the problem of computational cost and what you saw here was how using a 1 by 1 convolution, you can create this bottleneck layer thereby reducing the computational cost significantly. Now you might be wondering, does shrinking down the representation size so dramatically, does it hurt the performance of your neural network? It turns out that so long as you implement this bottleneck layer so that within reason, you can shrink down the representation size significantly, and it doesn't seem to hurt the performance, but saves you a lot of computation. So these are the key ideas of the inception module. Let's put them together and in the next section will show you what the full inception network looks like.

Inception Network

In the previous section, we've already seen all the basic building blocks of the Inception network. In this section, let's see how we can put these building blocks together to build your own Inception network. So the Inception module takes as input the activation or the output from some previous layer. So let's say for the sake of argument, this is 28 by 28 by 192, same as our previous section. The example we worked through in depth was the one by one followed by five by five layer. So maybe the one by one has 16 channels, and then the five by five will output a 28 by 28 by let say, 32 channels and this was the example we worked through on the last diagram of the previous section. Then, to save computation on your three by three convolution, you can also do the same here and then the three by three outputs 28 by 28 by 128 and then, maybe you want to consider a one by one convolution as well. There's no need to do a one by one conv followed by another one by one conv. So there's just one step here. And let's say this outputs 28 by 28 by 64 and then finally is the pooling layer. And here we're going to do something funny. So we are going to use- we want to use a same padding for max pooling. So the output is 28 by 28 and by- so here we're going to do something funny. In order to deliver concatenate all of these outputs at the end, we are going to use same type of padding for pooling so that the output item with is still 28 by 28. So we can concatenate it with these other outputs. But notice that if you do max pooling, even with same padding three by three filter is right at one. The output here will be twenty 28 by 28 by 192.

You will have the same number of channels and the same depth as the input that we had here. So this seems like it has a lot of channels. So what we're going to do is actually add one more one by one conv layer to then do what we saw in the one by one convolutional layer to shrink the number of channels so as to get this down to 28 by 28 by, let's say, 32 and the way you do that is to use 32 filters of dimension one by one by 192. So that's why the output dimension has the number of channels shrunk down to 32, so then you don't end up with the pooling layer taking up all the channels in the final output and then finally, you take these all of these blocks and you do channel concatenation, just concatenate across this 64 plus 128 plus 32 plus 32, and this if you add it up, this gives you a 28 by 28 by 256 dimensional output. Channel concat is just concatenating the blocks that we saw in the previous section. So this is one Inception module and **what the Inception network does is more or less put a lot of these modules together**. Below id the picture of the Inception that were taken from the paper by Szegety et al. and you notice a lot of repeated blocks in this.



Maybe this picture looks really complicated but you look at one of the blocks there, that block is basically the Inception module that you saw on the previous section. There's some extra Max pooling layers here to change the dimension of the height and width. But there's another Inception block, and then there's another max pool here to change the height and width but basically there's another Inception block. But the Inception network is just a lot of these blocks that you've learned about repeated to different positions of the network and so if you understand the Inception block from the previous section, then you understand the Inception network. It turns out that this one last detail to the Inception network if you read the original research paper, which is that there are these additional side branches that I just added. So what do they do? Well, the last few layers of the network is a fully connected layer followed by a softmax layer to try to make a prediction. What these side branches do is it take some hidden layer, and it tries to use that to make a prediction. So this is actually a softmax output, and so is that. And this other side branch, again takes a hidden layer passes through a few layers, a few fully connected layers, and then as a softmax tried to predict what's the output label. You should think of this as maybe just another detail of the Inception network, but what it does is it helps ensure that the features computes even in the hidden units, even at that intermediate layers that they're not too bad for predicting the output cause of a image and this appears to have a regularizing effect on the Inception network and helps prevent this network from overfitting.

So to summarize, **if you understand the Inception module, then you understand the Inception network, which is largely the Inception module repeated a bunch of times throughout the network**. Since the development of the original Inception module the authors and others have built on it and come up with other versions as well. So there are research papers on newer versions of the Inception algorithm and you sometimes see people use some of these later versions as well in their work, like Inception V2, Inception V3, Inception V4. There's also an Inception version that's combined with the resident idea of having skip connections and that sometimes works even better. But all of these variations are built on the basic idea that you learned about in this and the previous video of coming up with the Inception module and then stacking up a bunch of them together.

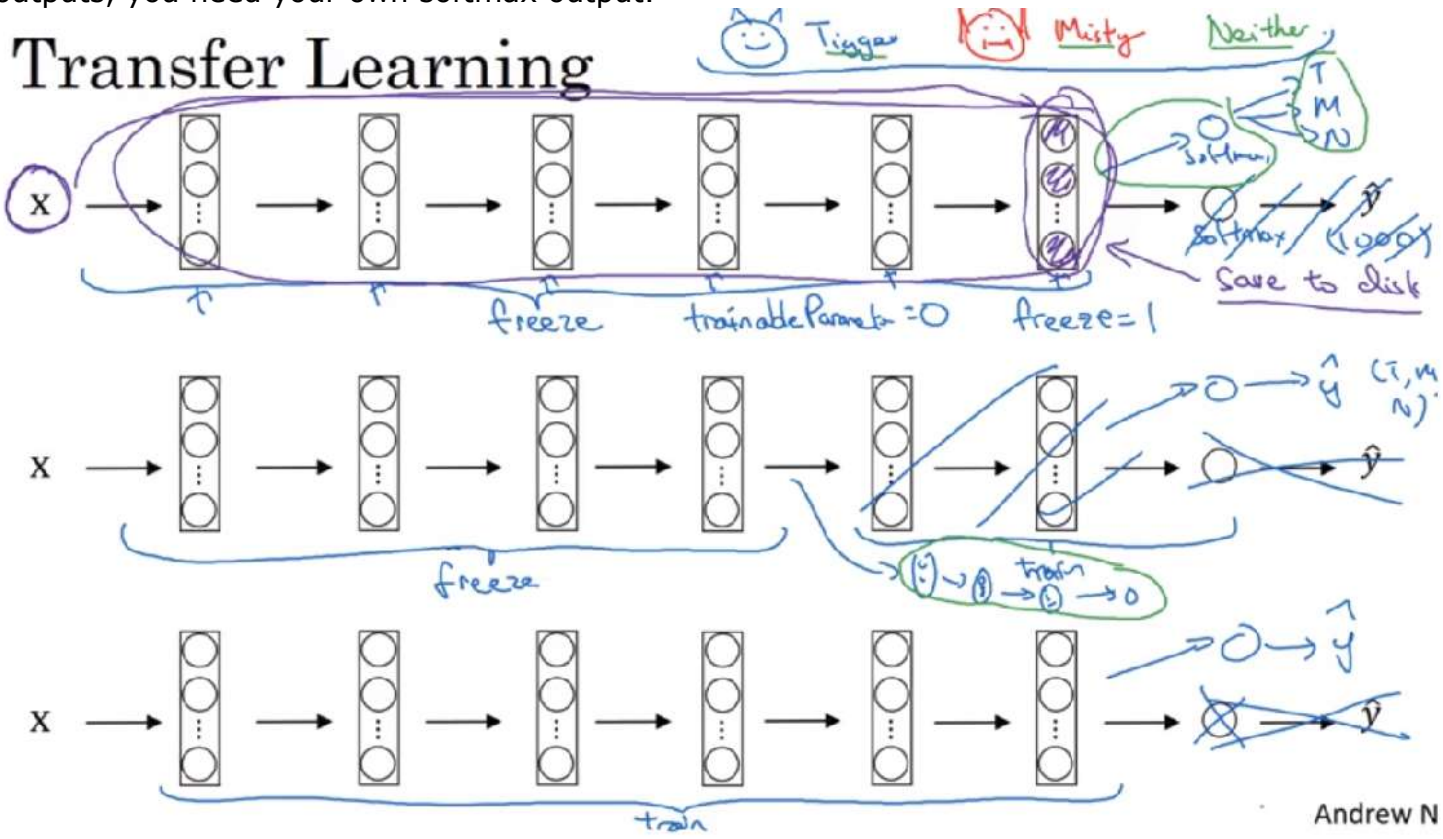
Practical advice for using ConvNets

Using Open-Source Implementation

Transfer Learning

If you're building a computer vision application rather than training the ways from scratch, from random initialization, you often make much faster progress if you download ways that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in. The computer vision research community has been pretty good at posting lots of data sets on the Internet so if you hear of things like Image Net, or MS COCO, or Pascal types of data sets, these are the names of different data sets that people have post online and a lot of computer researchers have trained their algorithms on. Sometimes these training takes several weeks and might take many GP use and the fact that someone else has done this and gone through the painful high-performance search process, means that you can often download open source ways that took someone else many weeks or months to figure out and use that as a very good initialization for your own neural network. And use transfer learning to sort of transfer knowledge from some of these very large public data sets to your own problem. Let's take a deeper look at how to do this. Let's start with the example, let's say you're building a cat detector to recognize your own pet cat. According to the internet, Tigger is a common cat name and Misty is another common cat name. Let's say your cats are called Tiger and Misty and there's also neither. You have a classification problem with three clauses. Is this picture Tigger, or is it Misty, or is it neither. And in all the case of both of you cats appearing in the picture. Now, you probably don't have a lot of pictures of Tigger or Misty so your training set will be small. What can you do? I recommend you go online and download some open source implementation of a neural network and download not just the code but also the weights. There are a lot of networks you can download that have been trained on for example, the Init Net data sets which has a thousand different clauses so the network might have a softmax unit that outputs one of a thousand possible clauses. What you can do is then get rid of the softmax layer and create your own softmax unit that outputs Tigger or Misty or neither. In terms of the network, I'd encourage you to think of all of these layers as frozen so you freeze the parameters in all of these layers of the network and you would then just train the parameters associated with your softmax layer. Which is the softmax layer with three possible outputs, Tigger, Misty or neither. By using someone else's free trade ways, you might probably get pretty good performance on this even with a small data set. Fortunately, a lot of people learning frameworks support this mode of operation and in fact, depending on the framework it might have things like trainable parameter equals zero, you might set that for some of these early layers. In others they just say, don't train those ways or sometimes you have a parameter like freeze equals one and these are different ways and different deep learning program frameworks that let you specify whether or not to train the ways associated with particular layer. In this case, you will train only the softmax layers ways but freeze all of the earlier layers ways. One other metric that may help for some implementations is that because all of these early leads are frozen, there are some fixed function that doesn't change because you're not changing it, you not training it that takes this input image acts and maps it to some set of activations in that layer. One of the trick that could speed up training is we just pre-compute that layer, the features of re-activations from that layer and just save them to disk. What you're doing is using this fixed function, in this first part of the neural network, to take this input any image X

and compute some feature vector for it and then you're training a shallow softmax model from this feature vector to make a prediction. One step that could help your computation as you just pre-compute that layers activation, for all the examples in training sets and save them to disk and then just train the softmax clause right on top of that. The advantage of the save to disk or the pre-compute method or the save to disk is that you don't need to recompute those activations everytime you take a epoch or take a post through a training set. This is what you do if you have a pretty small training set for your task. Whether you have a larger training set. One rule of thumb is if you have a larger label data set so maybe you just have a ton of pictures of Tigger, Misty as well as I guess pictures neither of them, one thing you could do is then freeze fewer layers. Maybe you freeze just these layers and then train these later layers. Although if the output layer has different clauses then you need to have your own output unit any way Tigger, Misty or neither. There are a couple of ways to do this. You could take the last few layers ways and just use that as initialization and do gradient descent from there or you can also lower weight of these last few layers and just use your own new hidden units and in your own final softmax outputs. Either of these matters could be worth trying. But maybe one pattern is if you have more data, the number of layers you've freeze could be smaller and then the number of layers you train on top could be greater and the idea is that if you pick a data set and maybe have enough data not just to train a single softmax unit but to train some other size neural network that comprises the last few layers of this final network that you end up using. Finally, if you have a lot of data, one thing you might do is take this open source network and ways and use the whole thing just as initialization and train the whole network. Although again if this was a thousand of softmax and you have just three outputs, you need your own softmax output.

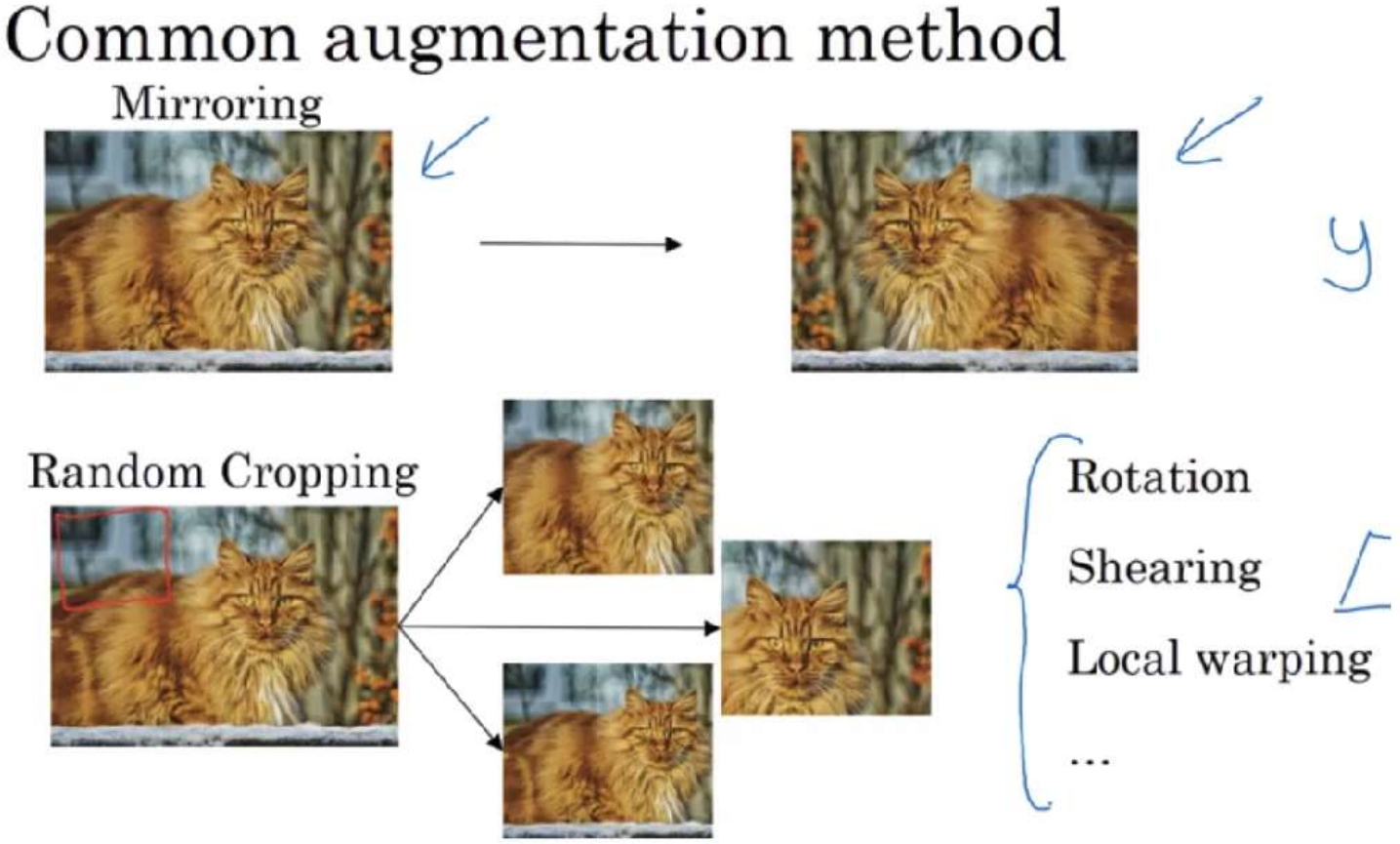


The output of labels you care about but the more label data you have for your task or the more pictures you have of Tigger, Misty and neither, the more layers you could train and in the extreme case, you could use the ways you download just as initialization so they would replace random initialization and then could do gradient descent, training updating all the ways and all the layers of the network. That's transfer learning for the training of ConvNets. In practice, because the open data sets on the internet are so big and the ways you can download that someone else has spent weeks training has learned from so much data, you find that for a lot of computer vision applications, you just do much better if you download someone else's open source ways and use that as initialization for your problem. In all the different disciplines, in all the different

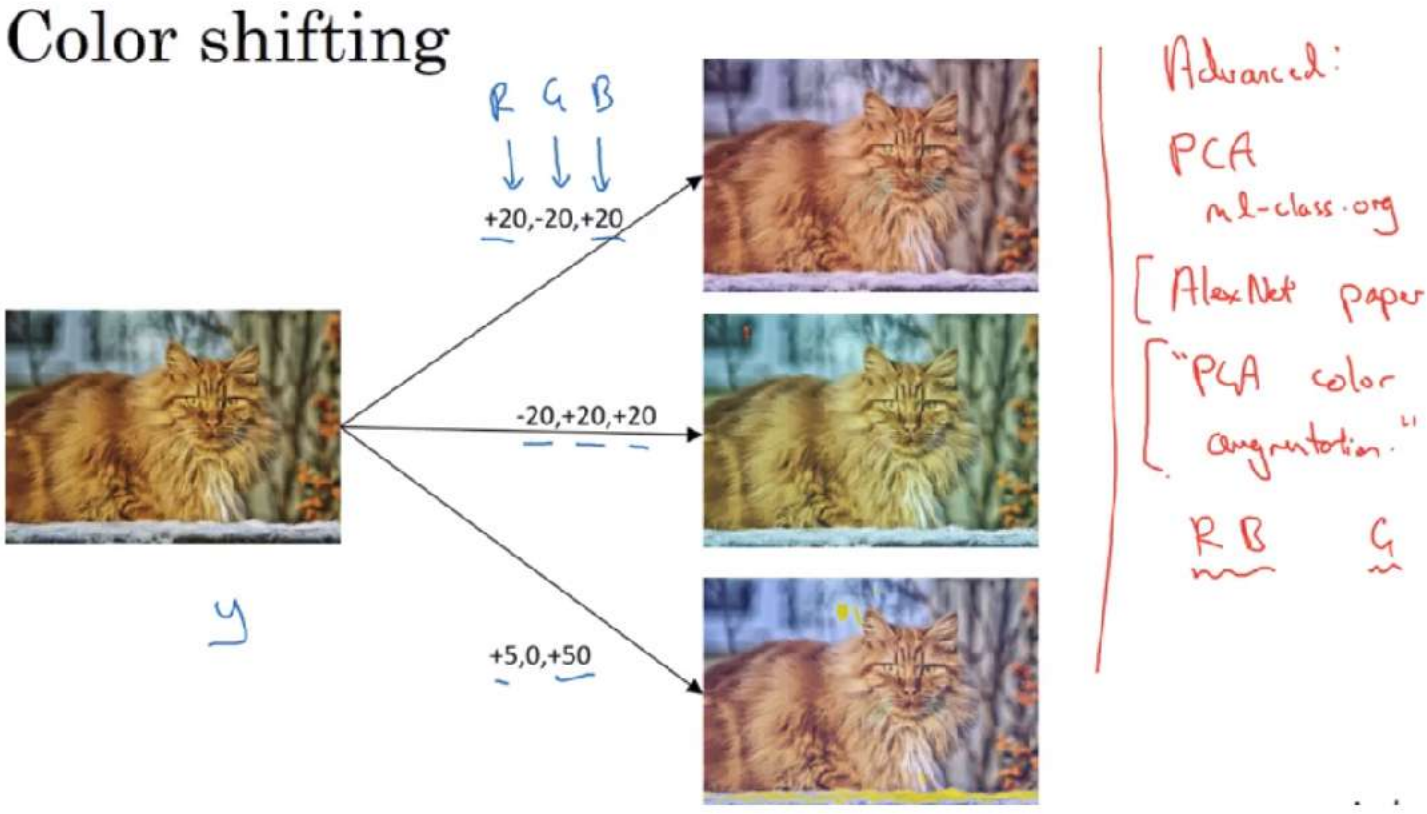
applications of deep learning, I think that computer vision is one where transfer learning is something that you should almost always do unless, you have an exceptionally large data set to train everything else from scratch yourself. But transfer learning is just very worth seriously considering unless you have an exceptionally large data set and a very large computation budget to train everything from scratch by yourself.

Data Augmentation

Most computer vision task could use more data. And so data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. I think that computer vision is a pretty complicated task. You have to input this image, all these pixels and then figure out what is in this picture and it seems like you need to learn the decently complicated function to do that and in practice, there almost all competing visions task having more data will help. This is unlike some other domains where sometimes you can get enough data, they don't feel as much pressure to get even more data. But I think today, this data computer vision is that, for the majority of computer vision problems, we feel like we just can't get enough data and this is not true for all applications of machine learning, but it does feel like it's true for computer vision. So, what that means is that when you're training in computer vision model, often **data augmentation** will help and this is true whether you're using transfer learning or using someone else's pre-trained ways to start, or whether you're trying to train something yourself from scratch. Let's take a look at the common data augmentation that is in computer vision. Perhaps the simplest data augmentation method is mirroring on the vertical axis, where if you have this example in your training set, you flip it horizontally to get that image on the right and for most computer vision task, if the left picture is a cat then mirroring it is though a cat and if the mirroring operation preserves whatever you're trying to recognize in the picture, this would be a good data augmentation technique to use. Another commonly used technique is **random cropping**. So given this dataset, let's pick a few random crops. So you might pick that, and take that crop or you might take that, to that crop, take this, take that crop and so this gives you different examples to feed in your training sample, sort of different random crops of your datasets. So random cropping isn't a perfect data augmentation. What if you randomly end up taking that crop which will look much like a cat but in practice and worthwhile so long as your random crops are reasonably large subsets of the actual image. So, **mirroring and random cropping are frequently used and in theory, you could also use things like rotation, shearing of the image**, so that's if you do this to the image, distort it that way, introduce various forms of local warping and so on. And there's really no harm with trying all of these things as well, although in practice they seem to be used a bit less, or perhaps because of their complexity.



The second type of data augmentation that is commonly used is **color shifting**. So, given a picture like this,



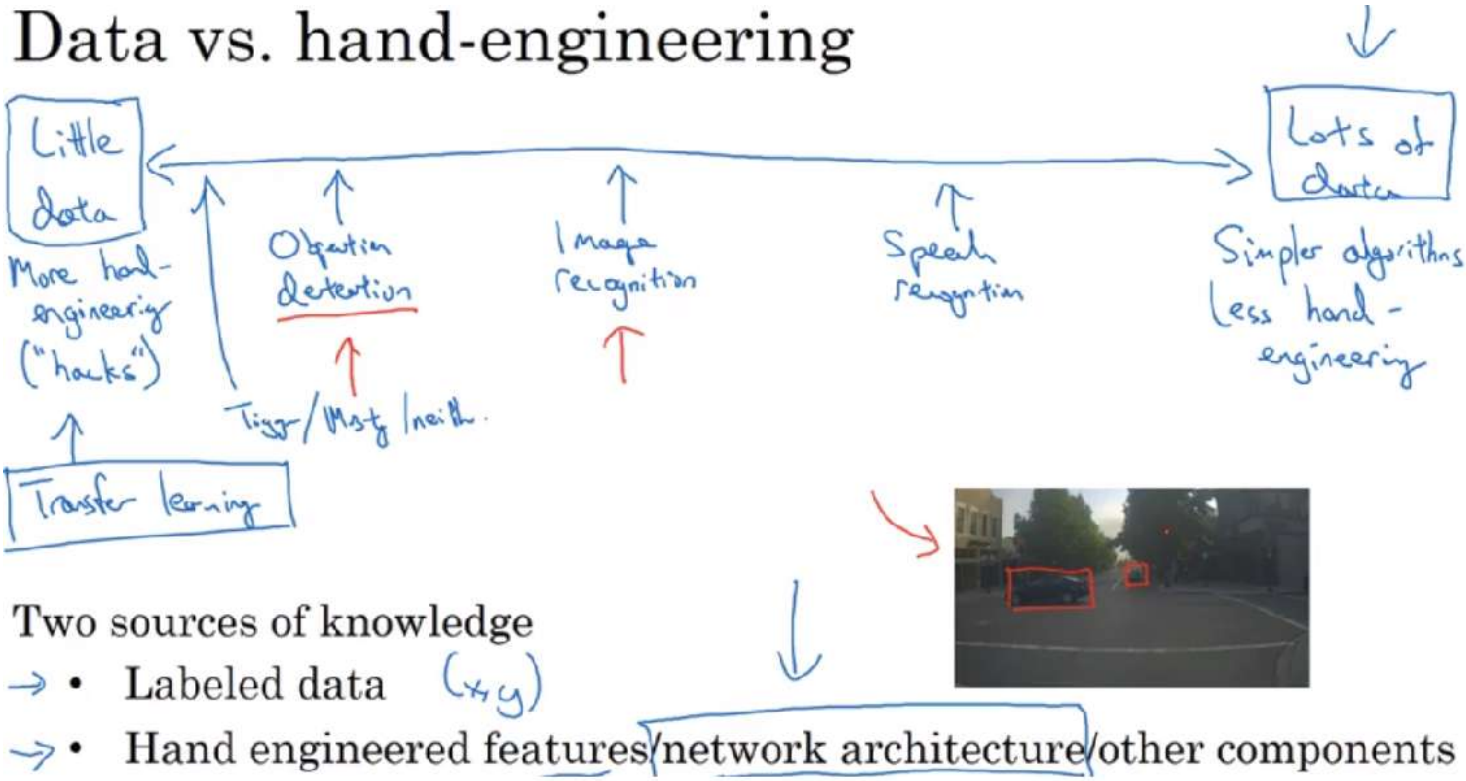
let's say you add to the R, G and B channels different distortions. In this example, we are adding to the red and blue channels and subtracting from the green channel. So, red and blue make purple. So, this makes the whole image a bit more purpley and that creates a distorted image for

training set. For illustration purposes, I'm making somewhat dramatic changes to the colors and practice, you draw R, G and B from some distribution that could be quite small as well. But what you do is take different values of R, G, and B and use them to distort the color channels. So, in the second example, we are making a less red, and more green and more blue, so that turns our image a bit more yellowish. And here, we are making it much more blue, just a tiny little bit longer. But in practice, the values R, G and B, are drawn from some probability distribution. And the motivation for this is that if maybe the sunlight was a bit yellow or maybe the in-goal illumination was a bit more yellow, that could easily change the color of an image, but the identity of the cat or the identity of the content, the label y, just still stay the same. And so introducing these color distortions or by doing color shifting, this makes your learning algorithm more robust to changes in the colors of your images. Just a comment for the advanced learners in this course, that is okay if you don't understand what I'm about to say when using red. There are different ways to sample R, G, and B. One of the ways to implement color distortion uses an algorithm called PCA. This is called **Principles Component Analysis**, which I talked about in the ml-class.org Machine Learning Course on Coursera. But the details of this are actually given in the AlexNet paper, and sometimes called **PCA Color Augmentation**. But the rough idea at the time PCA Color Augmentation is for example, if your image is mainly purple, if it mainly has red and blue tints, and very little green, then PCA Color Augmentation, will add and subtract a lot to red and blue, where it balance all the greens, so kind of keeps the overall color of the tint the same. If you didn't understand any of this, don't worry about it. But if you can search online for that, you can and if you want to read about the details of it in the AlexNet paper, and you can also find some open source implementations of the PCA Color Augmentation, and just use that. So, you might have your training data stored in a hard disk and uses symbol, this round bucket symbol to represent your hard disk. And if you have a small training set, you can do almost anything and you'll be okay. But the very last training set and this is how people will often implement it, which is you might have a CPU thread that is constantly loading images of your hard disk. So, you have this stream of images coming in from your hard disk. And what you can do is use maybe a CPU thread to implement the distortions, yet the random cropping, or the color shifting, or the mirroring, but for each image, you might then end up with some distorted version of it. So, let's see this image, I'm going to mirror it and if you also implement colors distortion and so on. And if this image ends up being color shifted, so you end up with some different colored cat. And so your CPU thread is constantly loading data as well as implementing whether the distortions are needed to form a batch or really many batches of data. And this data is then constantly passed to some other thread or some other process for implementing training and this could be done on the CPU or really increasingly on the GPU if you have a large neural network to train. And so, a pretty common way of implementing data augmentation is to really have one thread, almost four threads, that is responsible for loading the data and implementing distortions, and then passing that to some other thread or some other process that then does the training. And often, this and this, can run in parallel. So, that's it for data augmentation. And similar to other parts of training a deep neural network, the data augmentation process also has a few hyperparameters such as how much color shifting do you implement and exactly what parameters you use for random cropping? So, similar to elsewhere in computer vision, a good place to get started might be to use someone else's open source implementation for how they use data augmentation. But of course, if you want to capture more in variances, then you think someone else's open source implementation isn't, it might be reasonable also to use hyperparameters yourself. So with that, I hope that you're going to use data augmentation, to get your computer vision applications to work better.

State of Computer Vision

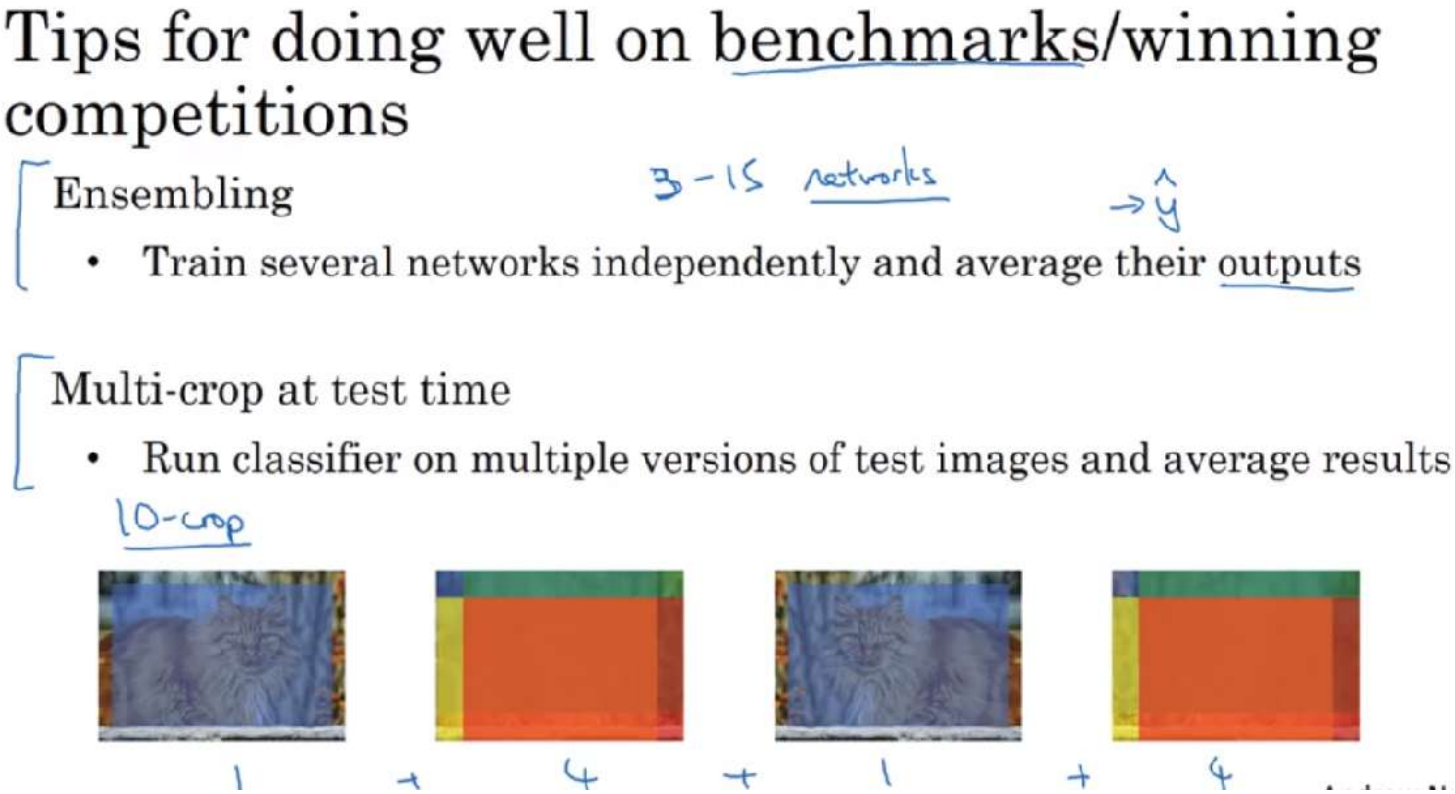
Deep learning has been successfully applied to computer vision, natural language processing, speech recognition, online advertising, logistics, many, many, many problems. There are a few things that are unique about the application of deep learning to computer vision, about the status of computer vision. In this video, I will share with you some of my observations about deep learning for computer vision and I hope that that will help you better navigate the literature, and the set of ideas out there, and how you build these systems yourself for computer vision. So, you

can think of most machine learning problems as falling somewhere on the spectrum between where you have relatively little data to where you have lots of data. So, for example I think that today we have a decent amount of data for speech recognition and it's relative to the complexity of the problem. And even though there are reasonably large data sets today for image recognition or image classification, because image recognition is just a complicated problem to look at all those pixels and figure out what it is. It feels like even though the online data sets are quite big like over a million images, feels like we still wish we had more data. And there are some problems like object detection where we have even less data. So, just as a reminder image recognition was the problem of looking at a picture and telling you is this a cattle or not. Whereas object detection is look in the picture and actually you're putting the bounding boxes are telling you where in the picture the objects such as the car as well. And so because of the cost of getting the bounding boxes is just more expensive to label the objects and the bounding boxes. So, we tend to have less data for object detection than for image recognition. And object detection is something we'll discuss next week. So, if you look across a broad spectrum of machine learning problems, you see on average that when you have a lot of data you tend to find people getting away with using simpler algorithms as well as less hand-engineering. So, there's just less needing to carefully design features for the problem, but instead you can have a giant neural network, even a simpler architecture, and have a neural network. Just learn whether we want to learn we have a lot of data. Whereas, in contrast when you don't have that much data then on average you see people engaging in more hand-engineering. And if you want to be ungenerous you can say there are more hacks. But I think when you don't have much data then hand-engineering is actually the best way to get good performance. So, when I look at machine learning applications I think usually we have the learning algorithm has two sources of knowledge. One source of knowledge is the labeled data, really the (x,y) pairs you use for supervised learning. And the second source of knowledge is the hand-engineering. And there are lots of ways to hand-engineer a system. It can be from carefully hand designing the features, to carefully hand designing the network architectures to maybe other components of your system. And so when you don't have much labeled data you just have to call more on hand-engineering. And so I think computer vision is trying to learn a really complex function. And it often feels like we don't have enough data for computer vision. Even though data sets are getting bigger and bigger, often we just don't have as much data as we need. And this is why this data computer vision historically and even today has relied more on hand-engineering. And I think this is also why that either computer vision has developed rather complex network architectures, is because in the absence of more data the way to get good performance is to spend more time architecting, or fooling around with the network architecture. And in case you think I'm being derogatory of hand-engineering that's not at all my intent. When you don't have enough data hand-engineering is a very difficult, very skillful task that requires a lot of insight. And someone that is insightful with hand-engineering will get better performance, and is a great contribution to a project to do that hand-engineering when you don't have enough data. It's just when you have lots of data then I wouldn't spend time hand-engineering, I would spend time building up the learning system instead. But I think historically the fear the computer vision has used very small data sets, and so historically the computer vision literature has relied on a lot of hand-engineering. And even though in the last few years the amount of data with the right computer vision task has increased dramatically, I think that that has resulted in a significant reduction in the amount of hand-engineering that's being done. But there's still a lot of hand-engineering of network architectures and computer vision. Which is why you see very complicated hyper frantic choices in computer vision, are more complex than you do in a lot of other disciplines. And in fact, because you usually have smaller object detection data sets than image recognition data sets, when we talk about object detection that is task like this next week. You see that the algorithms become even more complex and has even more specialized components. Fortunately, one thing that helps a lot when you have little data is transfer learning. And I would say for the example from the previous slide of the tigger, misty, neither detection problem, you have soluble data that transfer learning will help a lot.



Another set of techniques that's used a lot for when you have relatively little data. If you look at the computer vision literature, and look at the sort of ideas out there, you also find that people are really enthusiastic. They're really into doing well on standardized benchmark data sets and on winning competitions. And for computer vision researchers if you do well and the benchmark is easier to get the paper published. So, there's just a lot of attention on doing well on these benchmarks. And the positive side of this is that, it helps the whole community figure out what are the most effective algorithms. But you also see in the papers people do things that allow you to do well on a benchmark, but that you wouldn't really use in a production or a system that you deploy in an actual application. So, here are a few tips on doing well on benchmarks. These are things that I don't myself pretty much ever use if I'm putting a system to production that is actually to serve customers. But one is ensembling. And what that means is, after you've figured out what neural network you want, train several neural networks independently and average their outputs. So, initialize say 3, or 5, or 7 neural networks randomly and train up all of these neural networks, and then average their outputs. And by the way it is important to average their outputs y hats. Don't average their weights that won't work. Look and you say seven neural networks that have seven different predictions and average that. And this will cause you to do maybe 1% better, or 2% better. So is a little bit better on some benchmark. And this will cause you to do a little bit better. Maybe sometimes as much as 1 or 2% which really help win a competition. But because ensembling means that to test on each image, you might need to run an image through anywhere from say 3 to 15 different networks quite typical. This slows down your running time by a factor of 3 to 15, or sometimes even more. And so ensembling is one of those tips that people use doing well in benchmarks and for winning competitions. But that I think is almost never use in production to serve actual customers. I guess unless you have huge computational budget and don't mind burning a lot more of it per customer image. Another thing you see in papers that really helps on benchmarks, is multi-crop at test time. So, what I mean by that is you've seen how you can do data augmentation. And multi-crop is a form of applying data augmentation to your test image as well. So, for example let's see a cat image and just copy it four times including two more versions. There's a technique called the 10-crop, which basically says let's say you take this central region that crop, and run it through your crossfire. And then take that crop up the left hand corner run through a crossfire, up right hand corner shown in green, lower left shown in yellow, lower right shown in orange, and run that through the crossfire. And then do the same thing with the mirrored image. Right. So I'll take the central crop, then take the four corners crops. So, that's one central crop here and here, there's four corners crop here and here. And if

you add these up that's 10 different crops that you mentioned. So hence the name 10-crop. And so what you do, is you run these 10 images through your crossfire and then average the results. So, if you have the computational budget you could do this. Maybe you don't need as many as 10-crops, you can use a few crops. And this might get you a little bit better performance in a production system. By production I mean a system you're deploying for actual users. But this is another technique that is used much more for doing well on benchmarks than in actual production systems. And one of the big problems of ensembling is that you need to keep all these different networks around. And so that just takes up a lot more computer memory. For multi-crop I guess at least you keep just one network around. So it doesn't suck up as much memory, but it still slows down your run time quite a bit. So, these are tips you see and research papers will refer to these tips as well. But I personally do not tend to use these methods when building production systems even though they are great for doing better on benchmarks and on winning competitions. Because a lot of the computer vision problems are in the small data regime, others have done a lot of hand-engineering of the network architectures. And a neural network that works well on one vision problem often may be surprisingly, but they just often would work on other vision problems as well. So, to build a practical system often you do well starting off with someone else's neural network architecture.



And you can use an open source implementation if possible, because the open source implementation might have figured out all the finicky details like the learning rate, case scheduler, and other hyper parameters. And finally someone else may have spent weeks training a model on half a dozen GP use and on over a million images. And so by using someone else's pre-trained model and fine tuning on your data set, you can often get going much faster on an application. But of course if you have the compute resources and the inclination, don't let me stop you from training your own networks from scratch and in fact if you want to invent your own computer vision algorithm, that's what you might have to do.

Week 3: Object detection

Learn how to apply your knowledge of CNNs to one of the toughest but hottest field of computer vision: Object detection

Learning Objectives

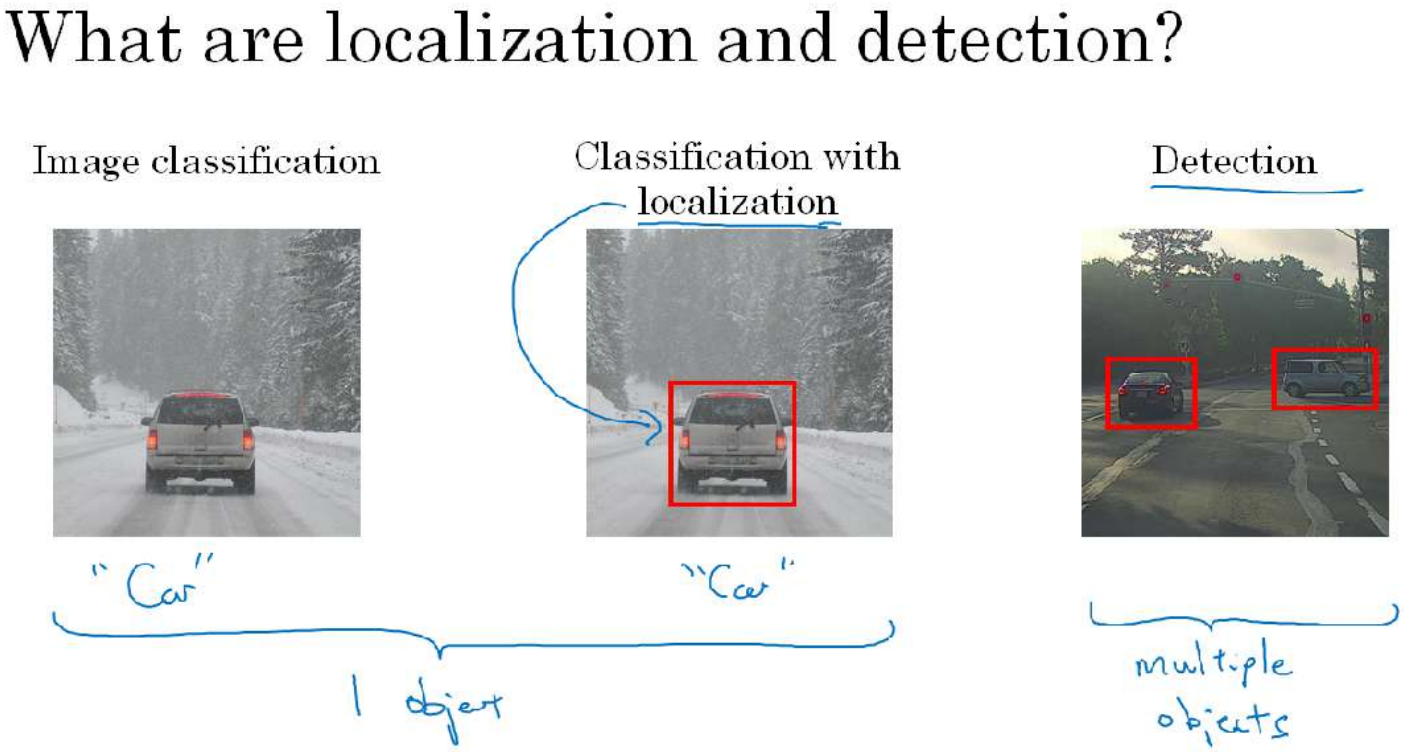
- Understand the challenges of Object Localization, Object Detection and Landmark Finding

- Understand and implement non-max suppression
- Understand and implement intersection over union
- Understand how we label a dataset for an object detection application
- Remember the vocabulary of object detection (landmark, anchor, bounding box, grid, ...)

Detection algorithms

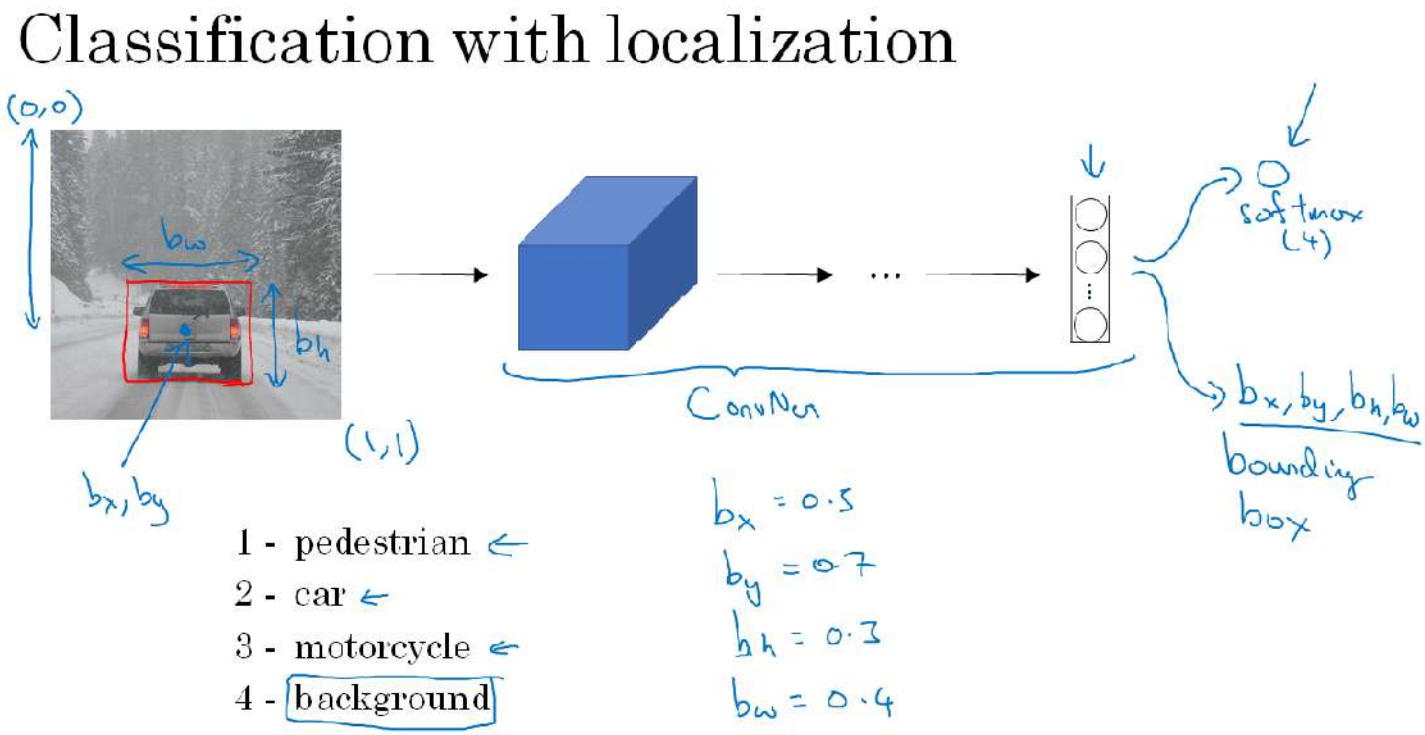
Object Localization

In this section we'll learn about **object detection**. This is one of the areas of computer vision that's just exploding and is working so much better than just a couple of years ago. In order to build up to object detection, you first learn about object localization. Let's start by defining what that means. You're already familiar with the image classification task where an algorithm looks at this picture below and might be responsible for saying this is a car. So that was classification.



So, not only do we have to label this as say a car but the algorithm also is responsible for putting a bounding box, or drawing a red rectangle around the position of the car in the image. So that's called the **classification with localization** problem. Where the term localization refers to figuring out where in the picture is the car you've detective. Later we'll also learn about the detection problem where now there might be multiple objects in the picture and you have to detect them all and and localized them all and if you're doing this for an autonomous driving application, then you might need to detect not just other cars, but maybe other pedestrians and motorcycles and maybe even other objects. So in the terminology we'll use this week, the classification and the classification of localization problems usually have one object. Usually one big object in the middle of the image that you're trying to recognize or recognize and localize. In contrast, in the detection problem there can be multiple objects and in fact, maybe even multiple objects of different categories within a single image. So the ideas you've learned about for image classification will be useful for classification with localization and that the ideas you learn for localization will then turn out to be useful for detection. So let's start by talking about classification with localization. You're already familiar with the image classification problem, in which you might input a picture into a ConfNet with multiple layers so that's our ConfNet and this results in a vector features that is fed to maybe a softmax unit that outputs the predicted output. So if you are building a self driving car, maybe your object categories are the following. Where you might have a pedestrian, or a car, or a motorcycle, or a background. This means none of the above. So if there's no pedestrian, no car, no motorcycle, then you might have an output background. So these are your classes, they have a

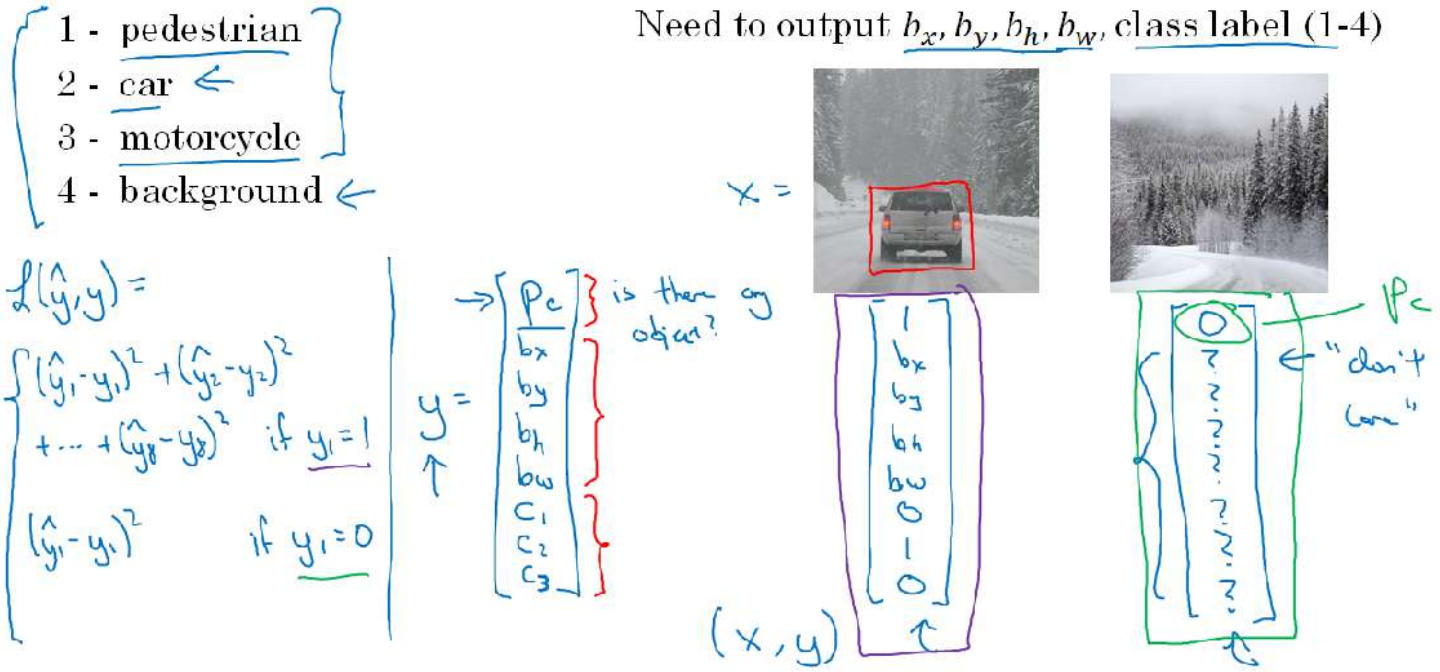
softmax with four possible outputs. So this is the standard classification pipeline. How about if you want to localize the car in the image as well. To do that, you can change your neural network to have a few more output units that output a bounding box. So, in particular, you can have the neural network output four more numbers, and I'm going to call them bx, by, bh, and bw. And these four numbers parameterized the bounding box of the detected object.



So in these sections, we are going to use the notational convention that the upper left of the image, I'm going to denote as the coordinate (0,0), and at the lower right is (1,1). So, specifying the bounding box, the red rectangle requires specifying the midpoint. So that's the point bx, by as well as the height, that would be bh, as well as the width, bw of this bounding box. So now if your training set contains not just the object cross label, which a neural network is trying to predict up here, but it also contains four additional numbers. Giving the bounding box then you can use supervised learning to make your algorithm outputs not just a class label but also the four parameters to tell you where is the bounding box of the object you detected. So in this example the ideal bx might be about 0.5 because this is about halfway to the right to the image. by might be about 0.7 since it's about maybe 70% to the way down to the image. bh might be about 0.3 because the height of this red square is about 30% of the overall height of the image. And bw might be about 0.4 let's say because the width of the red box is about 0.4 of the overall width of the entire image.

So let's formalize this a bit more in terms of how we define the target label y for this as a supervised learning task. So just as a reminder these are our four classes, and the neural network now outputs those four numbers as well as a class label, or maybe probabilities of the class labels. So, let's define the target label y as follows. Is going to be a vector where the first component pc is going to be, is there an object?

Defining the target label y



So, if the object is, classes 1, 2 or 3, pc will be equal to 1 and if it's the background class, so if it's none of the objects you're trying to detect, then pc will be 0. And pc you can think of that as standing for the probability that there's an object. Probability that one of the classes you're trying to detect is there. So something other than the background class. Next if there is an object, then you wanted to output bx, by, bh and bw, the bounding box for the object you detected and finally if there is an object, so if pc is equal to 1, you wanted to also output c1, c2 and c3 which tells us is it the class 1, class 2 or class 3. So is it a pedestrian, a car or a motorcycle and remember in the problem we're addressing we assume that your image has only one object. So at most, one of these objects appears in the picture, in this classification with localization problem. So let's go through a couple of examples. If this is a training set image, so if that is x, then y will be the first component pc will be equal to 1 because there is an object, then bx, by, by, bh and bw will specify the bounding box. So your labeled training set will need bounding boxes in the labels and then finally this is a car, so it's class 2. So c1 will be 0 because it's not a pedestrian, c2 will be 1 because it is car, c3 will be 0 since it is not a motorcycle. So among c1, c2 and c3 at most one of them should be equal to 1. So that's if there's an object in the image. What if there's no object in the image? What if we have a training example where x is equal to that? In this case, pc would be equal to 0, and the rest of the elements of this, will be don't cares, so I'm going to write question marks in all of them. So this is a don't care, because if there is no object in this image, then you don't care what bounding box the neural network outputs as well as which of the three objects, c1, c2, c3 it thinks it is. So given a set of label training examples, this is how you will construct x, the input image as well as y, the cost label both for images where there is an object and for images where there is no object and the set of this will then define your training set. Finally, let's describe the loss function you use to train the neural network. So the ground true label was y and the neural network outputs some yhat. What should be the loss be? Well if you're using squared error then the loss can be (y1 hat- y1) squared + (y2 hat- y2) squared + ...+(y8 hat- y8) squared. Notice that y here has eight components. So that goes from sum of the squares of the difference of the elements and that's the loss if y1=1. So that's the case where there is an object. So y1= pc. So, pc = 1, that if there is an object in the image then the loss can be the sum of squares of all the different elements.

The other case is if y1=0, so that's if this pc = 0. In that case the loss can be just (y1 hat-y1) squared, because in that second case, all of the rest of the components are don't care us and so all you care about is how accurately is the neural network ourputting pc in that case.

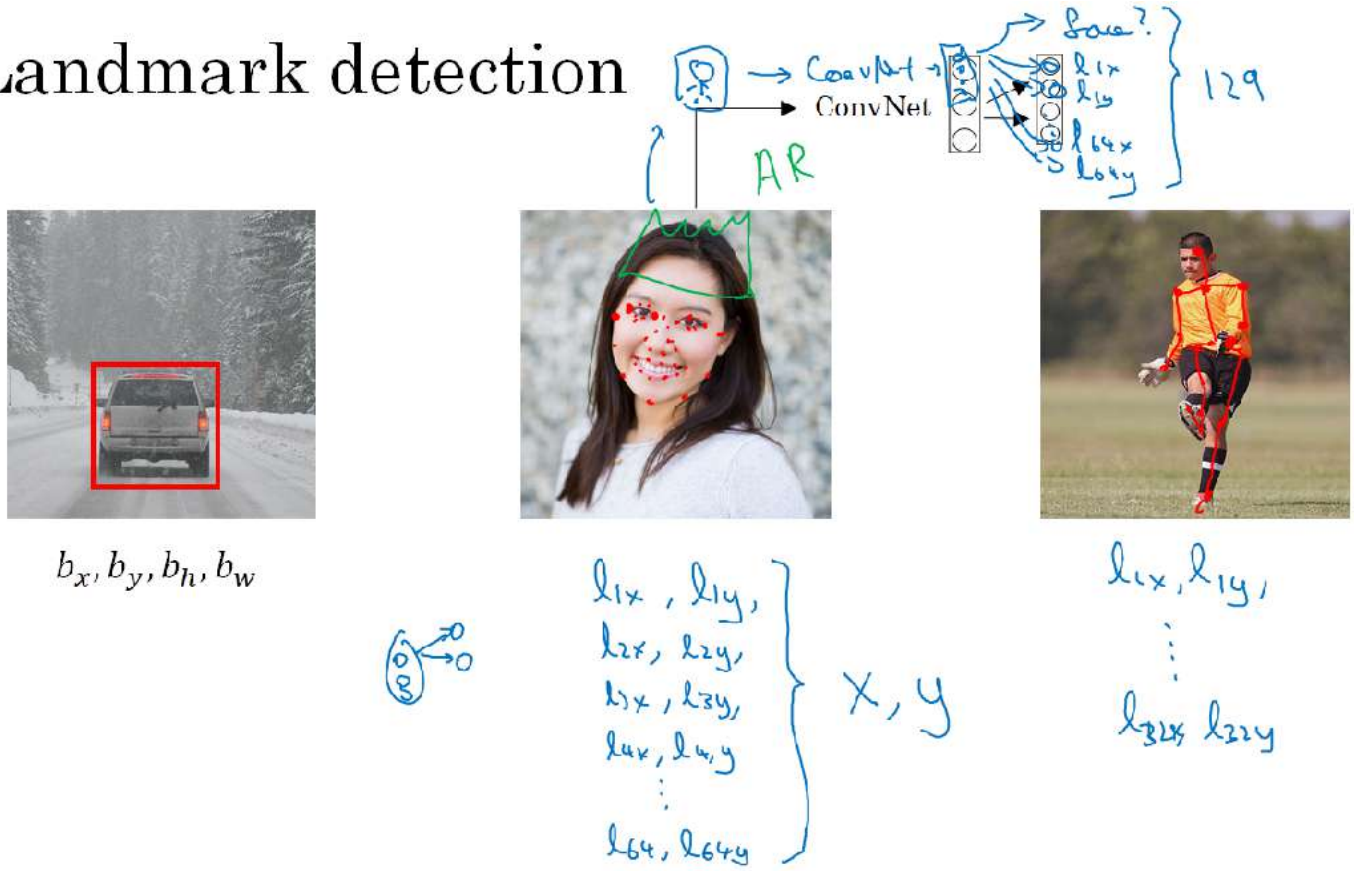
So just a recap, if y1 = 1, that's this case, then you can use squared error to penalize square deviation from the predicted, and the actual output of all eight components. Whereas if y1 = 0, then the second to the eighth components I don't care. So all you care about is how accurately is your neural network estimating y1, which is equal to pc.

Landmark Detection

In the previous section, we saw how you can get a neural network to output four numbers of bx, by, bh, and bw to specify the bounding box of an object you want a neural network to localize. In more general cases, you can have a neural network just output X and Y coordinates of important points and image, sometimes called **landmarks**, that you want the neural networks to recognize. Let's go through few examples.

Let's say you're building a face recognition application and for some reason, you want the algorithm to tell you where is the corner of someone's eye. So that point has an X and Y coordinate, so you can just have a neural network have its final layer and have it just output two more numbers which I'm going to call our lx and ly to just tell you the coordinates of that corner of the person's eye. Now, what if you want it to tell you all four corners of the eye, really of both eyes. So, if we call the points, the first, second, third and fourth points going from left to right, then you could modify the neural network now to output l_{1x}, l_{1y} for the first point and l_{2x}, l_{2y} for the second point and so on, so that the neural network can output the estimated position of all those four points of the person's face. But what if you don't want just those four points? What do you want to output this point, and this point and this point and this point along the eye? Maybe I'll put some key points along the mouth, so you can extract the mouth shape and tell if the person is smiling or frowning, maybe extract a few key points along the edges of the nose but you could define some number, for the sake of argument, let's say 64 points or 64 landmarks on the face. Maybe even some points that help you define the edge of the face, defines the jaw line but by selecting a number of landmarks and generating a label training sets that contains all of these landmarks, you can then have the neural network to tell you where are all the key positions or the key landmarks on a face.

Landmark detection

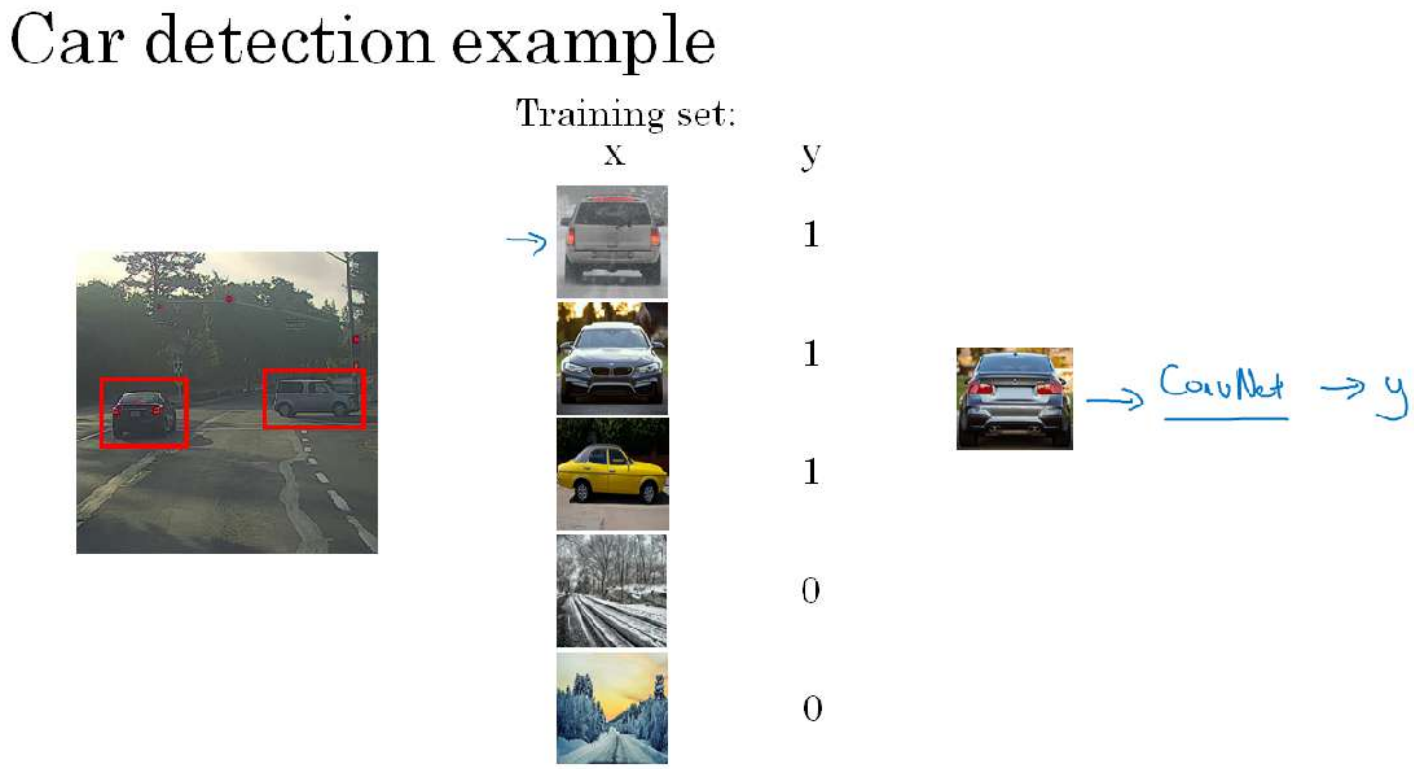


So what you do is you have this image, a person's face as input, have it go through a convnet and then have some set of features, maybe have it output 0 or 1, like zero face changes or not and then have it also output l_{1x}, l_{1y} and so on down to l_{64x}, l_{64y} and here I'm using **l** to stand for a landmark. So this example would have 129 output units, one for is your face or not? and then if you have 64 landmarks, that's sixty-four times two, so 128 plus one output units and this can tell you if there's a face as well as where all the key landmarks on the face. So, this is a basic building block for recognizing emotions from faces and if you played with the Snapchat and the other entertainment, also AR augmented reality filters like the Snapchat photos can draw a crown on the face and have other special effects. Being able to detect these landmarks on the face, there's also a key building block for the computer graphics effects that warp the face or drawing various special effects like putting a crown or a hat on the person. Of course, in order to treat a network like this, you will need a label training set. We have a set of images as well as labels Y where people, where someone will have had to go through and laboriously annotate all of these landmarks. One last example, if you are interested in people pose detection, you could also define a few key positions like the midpoint of the chest, the left shoulder, left elbow, the wrist, and so on, and just have a neural network to annotate key positions in the person's pose as well and by having a neural network output, all of those points I'm annotating, you could also have the neural network output the pose of the person and of course, to do that you also need to specify on these key landmarks like maybe l_{1x} and l_{1y} is the midpoint of the chest down to maybe l_{32x}, l_{32y} if you use 32 coordinates to specify the pose of the person. So, this idea might seem quite simple of just adding a bunch of output units to output the X,Y coordinates of different landmarks you want to recognize. To be clear, the identity of landmark one must be consistent across different images like maybe landmark one is always this corner of the eye, landmark two is always this corner of the eye, landmark three, landmark four, and so on. So, the labels have to be consistent across different images but if you can hire labelers or label yourself a big enough data set to do this, then a neural network can output all of these landmarks which is going to used to carry out other interesting effect such as with the pose of the person, maybe try to recognize someone's emotion from a picture, and so on. So that's it for **landmark detection**. Next, let's take these building blocks and use it to start building up towards object detection.

Object Detection

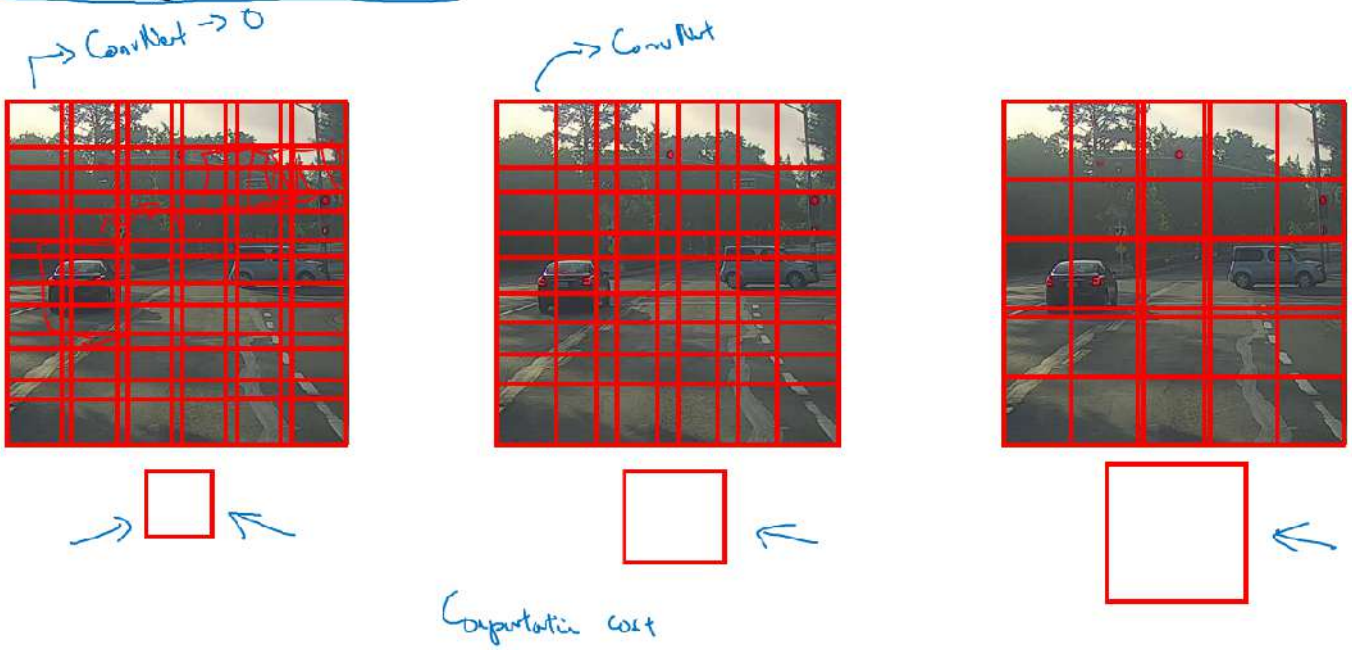
We've learned about Object Localization as well as Landmark Detection. Now, let's build up to other object detection algorithm. In this section, we'll learn how to use a ConvNet to perform object detection using something called the **Sliding Windows Detection Algorithm**.

Let's say you want to build a car detection algorithm. Here's what you can do. You can first create a label training set, so x and y with closely cropped examples of cars.



So, this is image x has a positive example, there's a car, here's a car, here's a car, and then there's not a car, there's not a car and for our purposes in this training set, you can start off with the one with the car closely cropped images. Meaning that x is pretty much only the car. So, you can take a picture and crop out and just cut out anything else that's not part of a car. So you end up with the car centered in pretty much the entire image. Given this label training set, you can then train a ConvNet that inputs an image, like one of these closely cropped images and then the job of the ConvNet is to output y, zero or one, is there a car or not. Once you've trained up this ConvNet, you can then use it in Sliding Windows Detection. So the way you do that is, if you have a test image, what you do is you start by picking a certain window size, shown down there and then you would input into this ConvNet a small rectangular region. So, take just this below red square, input that into the ConvNet, and have a ConvNet make a prediction and presumably for that little region in the red square, it'll say, no that little red square does not contain a car.

Sliding windows detection

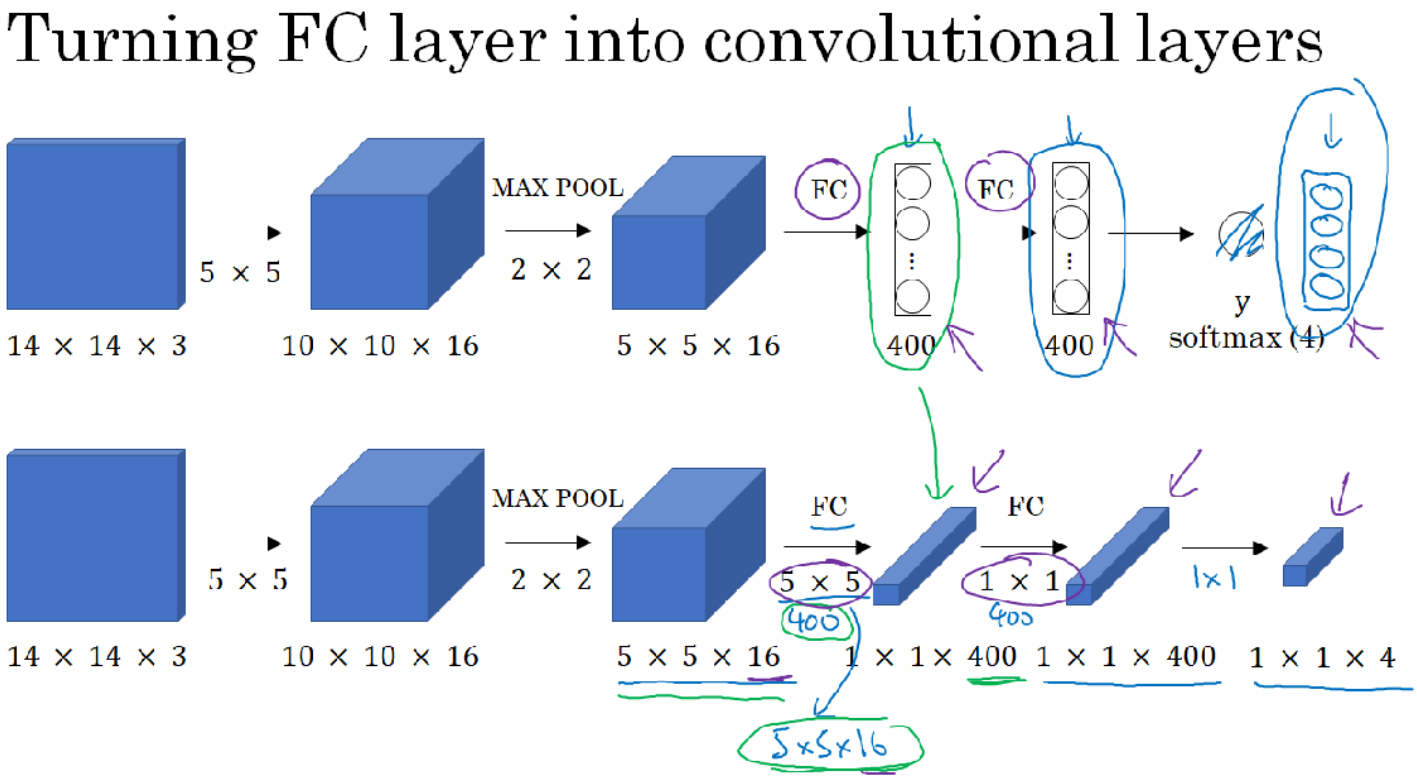


In the Sliding Windows Detection Algorithm, what you do is you then pass as input a second image now bounded by this red square shifted a little bit over and feed that to the ConvNet. So, you're feeding just the region of the image in the red squares of the ConvNet and run the ConvNet again and then you do that with a third image and so on and you keep going until you've slid the window across every position in the image and I'm using a pretty large stride in this example just to make the animation go faster. But the idea is you basically go through every region of this size, and pass lots of little cropped images into the ConvNet and have it classified zero or one for each position as some stride. Now, having done this once with running this was called the sliding window through the image. You then repeat it, but now use a larger window. So, now you take a slightly larger region and run that region. So, resize this region into whatever input size the cofinite is expecting, and feed that to the ConvNet and have it output zero or one and then slide the window over again using some stride and so on and you run that throughout your entire image until you get to the end and then you might do the third time using even larger windows and so on. Right. And the hope is that if you do this, then so long as there's a car somewhere in the image that there will be a window where, for example if you are passing in this window into the ConvNet, hopefully the ConvNet will have outputs one for that input region. So then you detect that there is a car there. So this algorithm is called **Sliding Windows Detection because you take these windows, these square boxes, and slide them across the entire image and classify every square region with some stride as containing a car or not.** Now there's a huge disadvantage of Sliding Windows Detection, which is the computational cost. Because you're cropping out so many different square regions in the image and running each of them independently through a ConvNet and if you use a very coarse stride, a very big stride, a very big step size, then that will reduce the number of windows you need to pass through the cofinite, but that courser granularity may hurt performance. Whereas if you use a very fine granularity or a very small stride, then the huge number of all these little regions you're passing through the ConvNet means that means there is a very high computational cost. So, before the rise of Neural Networks people used to use much simpler classifiers like a simple linear classifier over hand engineer features in order to perform object detection and in that era because each classifier was relatively cheap to compute, it was just a linear function, Sliding Windows Detection ran okay. It was not a bad method, but with ConvNet now running a single classification task is much more expensive and sliding windows this way is infeasibily slow. And unless you use a very fine granularity or a very small stride, you end up not able to localize the objects that accurately within the image as well. Fortunately however, this problem of computational cost has a pretty good

solution. In particular, the Sliding Windows Object Detector can be implemented convolutionally or much more efficiently.

Convolutional Implementation of Sliding Windows

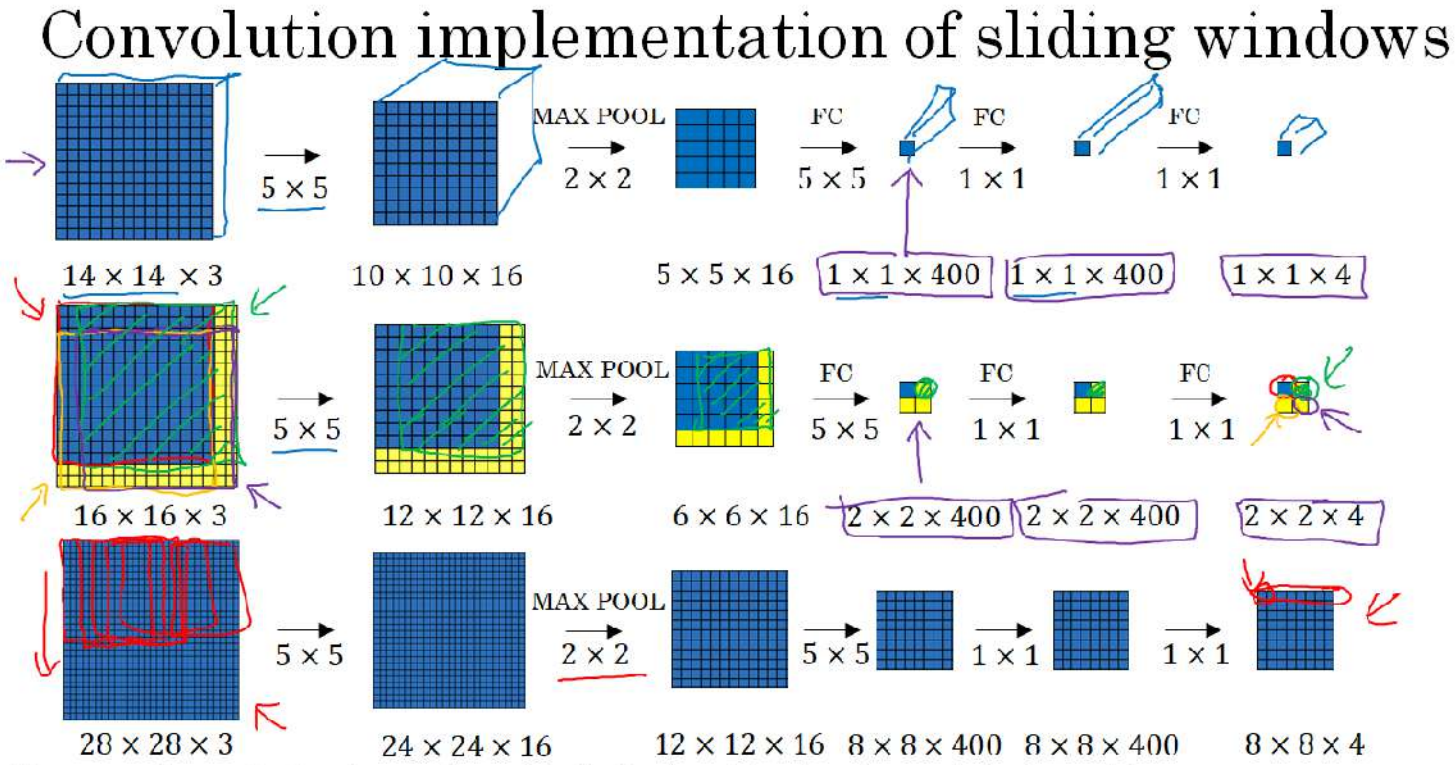
In the last section, we've learned about the sliding windows object detection algorithm using a convnet but we saw that it was too slow. In this section, we'll learn how to implement that algorithm convolutionally. Let's see what this means. To build up towards the convolutional implementation of sliding windows let's first see how you can turn fully connected layers in neural network into convolutional layers. We'll do that first on this slide and then the next slide, we'll use the ideas from this slide to show you the convolutional implementation. So let's say that your object detection algorithm inputs 14x14x3 images. This is quite small but just for illustrative purposes, and let's say it then uses 5x5 filters, and let's say it uses 16 of them to map it from 14x14x3 to 10x10x16. And then does a 2x2 max pooling to reduce it to 5x5x16. Then has a fully connected layer to connect to 400 units. Then now they're fully connected layer and then finally outputs a Y using a softmax unit.



Lets take Y as four numbers, corresponding to the cause probabilities of the four causes that softmax units is classified amongst and the full causes could be pedestrian, car, motorcycle, and background or something else. Now, what we'd like to do is show how these layers can be turned into convolutional layers. So, the convnet will draw same as before for the first few layers. And now, one way of implementing this next layer, this fully connected layer is to implement this as a 5 by 5 filter and let's use 400, 5 by 5 filters. So if you take a 5 by 5 by 16 image and convolve it with a 5 by 5 filter, remember, a 5 by 5 filter is implemented as 5 by 5 by 16 because our convention is that the filter looks across all 16 channels. So the 16 must match with other 16 so the outputs will be 1 by 1 and if you have 400 of these 5 by 5 by 16 filters, then the output dimension is going to be 1x1x400. So rather than viewing these 400 as just a set of nodes, we're going to view this as a 1x1x400 volume. Mathematically, this is the same as a fully connected layer because each of these 400 nodes has a filter of dimension 5 by 5 by 16. So each of those 400 values is some arbitrary linear function of these 5 by 5 by 16 activations from the previous layer.

Next, to implement the next convolutional layer, we're going to implement a 1 by 1 convolution. If you have 400 1 by 1 filters then, with 400 filters the next layer will again be 1 by 1 by 400. So that gives you this next fully connected layer and then finally, we're going to have another 1 by 1

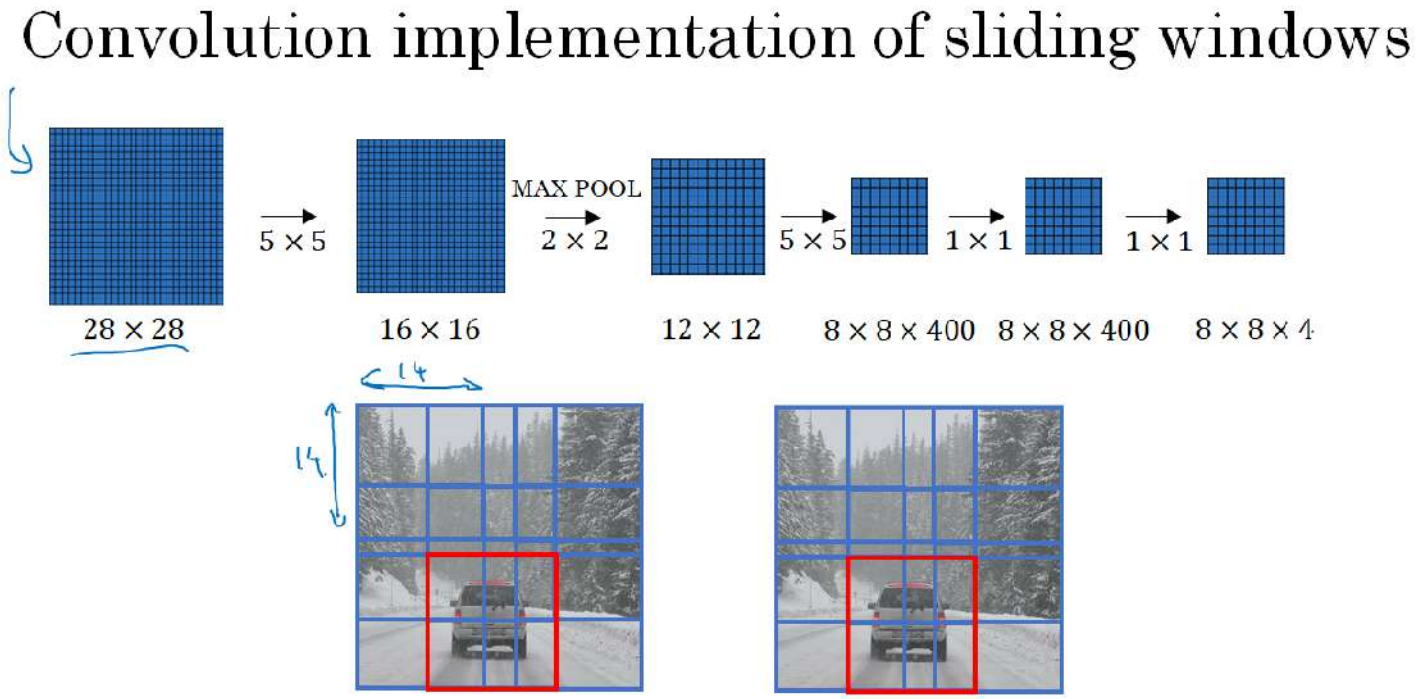
filter, followed by a softmax activation. So as to give a 1 by 1 by 4 volume to take the place of these four numbers that the network was operating. So this shows how you can take these fully connected layers and implement them using convolutional layers so that these sets of units instead are not implemented as 1 by 1 by 400 and 1 by 1 by 4 volumes. After this conversion, let's see how you can have a convolutional implementation of sliding windows object detection.



[Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks]

The presentation on this slide is based on the OverFeat paper, referenced at the bottom, by Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Robert Fergus and Yann Lecun. Let's say that your sliding windows convnet inputs 14 by 14 by 3 images and again, I'm just using small numbers like the 14 by 14 image in this slide mainly to make the numbers and illustrations simpler. So as before, you have a neural network as follows that eventually outputs a 1 by 1 by 4 volume, which is the output of your softmax. Again, to simplify the drawing here, 14 by 14 by 3 is technically a volume 5 by 5 or 10 by 10 by 16, the second clear volume. But to simplify the drawing for this slide, I'm just going to draw the front face of this volume. So instead of drawing 1 by 1 by 400 volume, I'm just going to draw the 1 by 1 cause of all of these. So just dropped the three components of these drawings, just for this slide. So let's say that your convnet inputs 14 by 14 images or 14 by 14 by 3 images and your tested image is 16 by 16 by 3. So now added that yellow stripe to the border of this image. In the original sliding windows algorithm, you might want to input the blue region into a convnet and run that once to generate a consecration 01 and then slightly down a bit, least he uses a stride of two pixels and then you might slide that to the right by two pixels to input this green rectangle into the convnet and we run the whole convnet and get another label, 01. Then you might input this orange region into the convnet and run it one more time to get another label. And then do it the fourth and final time with this lower right purple square. To run sliding windows on this 16 by 16 by 3 image is pretty small image. You run this convnet four times in order to get four labels. But it turns out a lot of this computation done by these four convnets is highly duplicative. So what the convolutional implementation of sliding windows does is it allows these four pauses in the convnet to share a lot of computation. Specifically, here's what you can do. You can take the convnet and just run it same parameters, the same 5 by 5 filters, also 16, 5 by 5 filters and run it. Now, you can have a 12 by 12 by 16 output volume. Then do the max pool, same as before. Now you have a 6 by 6 by 16, runs through your same 400, 5 by 5 filters to get now your 2 by 2 by 40 volume. So now instead of a 1 by 1 by 400 volume, we have instead a 2 by 2 by 400 volume. Run it through a 1 by 1 filter gives

you another 2 by 2 by 400 instead of 1 by 1 like 400. Do that one more time and now you're left with a 2 by 2 by 4 output volume instead of 1 by 1 by 4. It turns out that this blue 1 by 1 by 4 subset gives you the result of running in the upper left hand corner 14 by 14 image. This upper right 1 by 1 by 4 volume gives you the upper right result. The lower left gives you the results of implementing the convnet on the lower left 14 by 14 region. And the lower right 1 by 1 by 4 volume gives you the same result as running the convnet on the lower right 14 by 14 medium. And if you step through all the steps of the calculation, let's look at the green example, if you had cropped out just this region and passed it through the convnet through the convnet on top, then the first layer's activations would have been exactly this region. The next layer's activation after max pooling would have been exactly this region and then the next layer, the next layer would have been as follows. So what this process does, what this convolution implementation does is, instead of forcing you to run four propagation on four subsets of the input image independently, Instead, it combines all four into one form of computation and shares a lot of the computation in the regions of image that are common. So all four of the 14 by 14 patches we saw here.
Now let's just go through a bigger example.

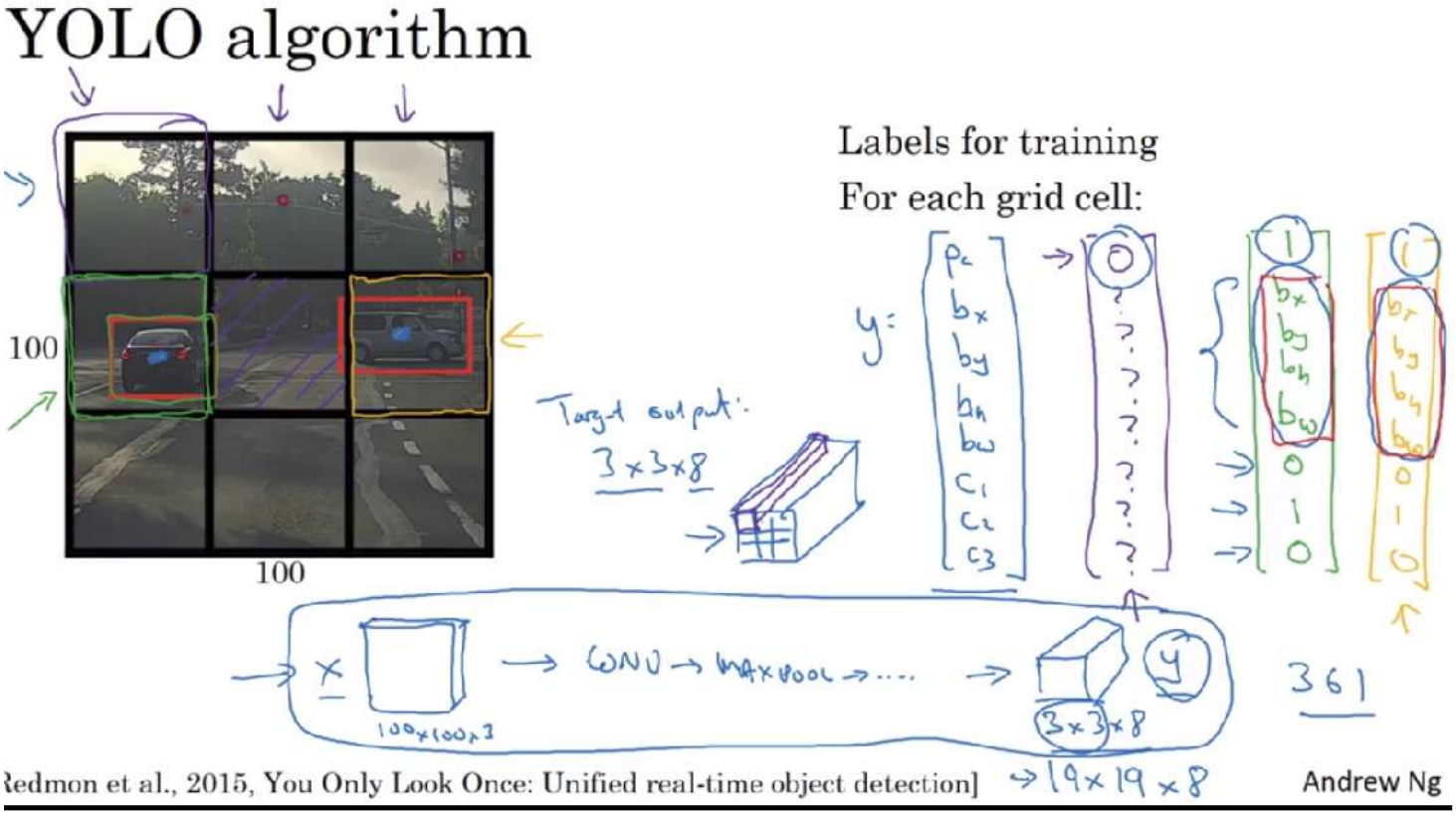


Let's say you now want to run sliding windows on a 28 by 28 by 3 image. It turns out If you run four from the same way then you end up with an 8 by 8 by 4 output. And just go small and surviving sliding windows with that 14 by 14 region. And that corresponds to running a sliding windows first on that region thus, giving you the output corresponding the upper left hand corner. Then using a slider too to shift one window over, one window over, one window over and so on and the eight positions. So that gives you this first row and then as you go down the image as well, that gives you all of these 8 by 8 by 4 outputs. Because of the max pooling up too that this corresponds to running your neural network with a stride of two on the original image. So just to recap, to implement sliding windows, previously, what you do is you crop out a region. Let's say this is 14 by 14 and run that through your convnet and do that for the next region over, then do that for the next 14 by 14 region, then the next one, then the next one, then the next one, then the next one and so on, until hopefully that one recognizes the car. But now, instead of doing it sequentially, with this convolutional implementation that you saw in the previous slide, you can implement the entire image, all maybe 28 by 28 and convolutionally make all the predictions at the same time by one forward pass through this big convnet and hopefully have it recognize the position of the car. So that's how you implement sliding windows convolutionally and it makes the whole thing much more efficient. Now, this still has one weakness, which is the position of the bounding boxes is not going to be too accurate.

Bounding Box Predictions

In the last section we've learned how to use a convolutional implementation of sliding windows. That's more computationally efficient, but it still has a problem of not quite outputting the most accurate bounding boxes. In this section, let's see how you can get your bounding box predictions to be more accurate. With sliding windows, you take this three sets of locations and run the crossfire through it and in this case, none of the boxes really match up perfectly with the position of the car. So, maybe that box is the best match and also, it looks like in drawn through, the perfect bounding box isn't even quite square, it's actually has a slightly wider rectangle or slightly horizontal aspect ratio. So, is there a way to get this algorithm to outputs more accurate bounding boxes? A good way to get this output more accurate bounding boxes is with the **YOLO algorithm**. YOLO stands for, You Only Look Once. And is an algorithm due to Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi. Here's what you do.

Let's say you have an input image at 100 by 100, you're going to place down a grid on this image and for the purposes of illustration, we're going to use a 3 by 3 grid. Although in an actual implementation, you use a finer one, like maybe a 19 by 19 grid and the basic idea is you're going to take the image classification and localization algorithm that you saw a few sections back, and apply it to each of the nine grids and the basic idea is you're going to take the image classification and localization algorithm that you saw in the first section of this week and apply that to each of the nine grid cells of this image. So the more concrete, here's how you define the labels you use for training. So for each of the nine grid cells, you specify a label Y, where the label Y is this eight dimensional vector, same as you saw previously. Your first output PC 01 depending on whether or not there's an image in that grid cell and then BX, BY, BH, BW to specify the bounding box if there is an image, if there is an object associated with that grid cell. And then say, C1, C2, C3, if you try and recognize three classes not counting the background class. So you try to recognize pedestrian's class, motorcycles and the background class. Then C1 C2 C3 can be the pedestrian, car and motorcycle classes.



So in this image, we have nine grid cells, so you have a vector like this for each of the grid cells. So let's start with the upper left grid cell, this one up here. For that one, there is no object. So, the label vector Y for the upper left grid cell would be zero, and then don't cares for the rest of these. The output label Y would be the same for this grid cell, and this grid cell, and all the grid cells with nothing, with no interesting object in them. Now, how about this grid cell? To give a bit

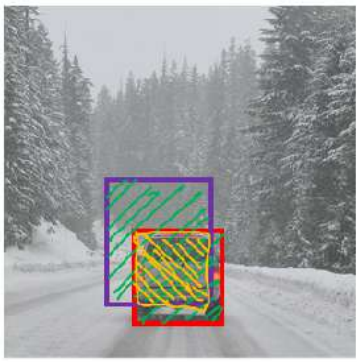
more detail, this image has two objects. And what the **YOLO algorithm does is it takes the midpoint of reach of the two objects and then assigns the object to the grid cell containing the midpoint. So the left car is assigned to this grid cell, and the car on the right, which is this midpoint, is assigned to this grid cell**. And so even though the central grid cell has some parts of both cars, we'll pretend the central grid cell has no interesting object so that the central grid cell the class label Y also looks like this vector with no object, and so the first component PC, and then the rest are don't cares. Whereas for this cell, this cell that I have circled in green on the left, the target label Y would be as follows. There is an object, and then you write BX, BY, BH, BW, to specify the position of this bounding box. And then you have, let's see, if class one was a pedestrian, then that was zero. Class two is a car, that's one. Class three was a motorcycle, that's zero. And then similarly, for the grid cell on their right because that does have an object in it, it will also have some vector like this as the target label corresponding to the grid cell on the right. So, for each of these nine grid cells, you end up with a eight dimensional output vector. And because you have 3 by 3 grid cells, you have nine grid cells, the total volume of the output is going to be 3 by 3 by 8. So the target output is going to be 3 by 3 by 8 because you have 3 by 3 grid cells. And for each of the 3 by 3 grid cells, you have a eight dimensional Y vector. So the target output volume is 3 by 3 by 8. Where for example, this 1 by 1 by 8 volume in the upper left corresponds to the target output vector for the upper left of the nine grid cells. And so for each of the 3 by 3 positions, for each of these nine grid cells, does it correspond in eight dimensional target vector Y that you want to the output. Some of which could be don't cares, if there's no object there. And that's why the total target outputs, the output label for this image is now itself a 3 by 3 by 8 volume. So now, to train your neural network, the input is 100 by 100 by 3, that's the input image. And then you have a usual convnet with conv, layers of max pool layers, and so on. So that in the end, you have this, should choose the conv layers and the max pool layers, and so on, so that this eventually maps to a 3 by 3 by 8 output volume. And so what you do is you have an input X which is the input image like that, and you have these target labels Y which are 3 by 3 by 8, and you use map propagation to train the neural network to map from any input X to this type of output volume Y. So the advantage of this algorithm is that the neural network outputs precise bounding boxes as follows. So at test time, what you do is you feed an input image X and run forward prop until you get this output Y. And then for each of the nine outputs of each of the 3 by 3 positions in which of the output, you can then just read off 1 or 0. Is there an object associated with that one of the nine positions? And that there is an object, what object it is, and where is the bounding box for the object in that grid cell? And so long as you don't have more than one object in each grid cell, this algorithm should work okay. And the problem of having multiple objects within the grid cell is something we'll address later. Of use a relatively small 3 by 3 grid, in practice, you might use a much finer, grid maybe 19 by 19. So you end up with 19 by 19 by 8, and that also makes your grid much finer. It reduces the chance that there are multiple objects assigned to the same grid cell. And just as a reminder, the way you assign an object to grid cell as you look at the midpoint of an object and then you assign that object to whichever one grid cell contains the midpoint of the object. So each object, even if the objects spends multiple grid cells, that object is assigned only to one of the nine grid cells, or one of the 3 by 3, or one of the 19 by 19 grid cells. Algorithm of a 19 by 19 grid, the chance of an object of two midpoints of objects appearing in the same grid cell is just a bit smaller. So notice two things, first, this is a lot like the image classification and localization algorithm that we talked about in the first video of this week. And that it outputs the bounding balls coordinates explicitly. And so this allows in your network to output bounding boxes of any aspect ratio, as well as, output much more precise coordinates that aren't just dictated by the stripe size of your sliding windows classifier. And second, this is a convolutional implementation and you're not implementing this algorithm nine times on the 3 by 3 grid or if you're using a 19 by 19 grid.19 squared is 361. So, you're not running the same algorithm 361 times or 19 squared times. Instead, this is one single convolutional implantation, where you use one consonant with a lot of shared computation between all the computations needed for all of your 3 by 3 or all of your 19 by 19 grid cells. So, this is a pretty efficient algorithm. And in fact, one nice thing about the YOLO algorithm, which is constant popularity is because this is a convolutional implementation, it actually runs very fast. So

this works even for real time object detection. Now, before wrapping up, there's one more detail I want to share with you, which is, how do you encode these bounding boxes bx, by, BH, BW? Let's discuss that on the next slide. So, given these two cars, remember, we have the 3 by 3 grid. Let's take the example of the car on the right. So, in this grid cell there is an object and so the target label y will be one, that was PC is equal to one. And then bx, by, BH, BW, and then 0 1 0. So, how do you specify the bounding box? In the YOLO algorithm, relative to this square, when I take the convention that the upper left point here is 0 0 and this lower right point is 1 1. So to specify the position of that midpoint, that orange dot, bx might be, let's say x looks like is about 0.4. Maybe its about 0.4 of the way to their right. And then y, looks I guess maybe 0.3. And then the height of the bounding box is specified as a fraction of the overall width of this box. So, the width of this red box is maybe 90% of that blue line. And so BH is 0.9 and the height of this is maybe one half of the overall height of the grid cell. So in that case, BW would be, let's say 0.5. So, in other words, this bx, by, BH, BW as specified relative to the grid cell. And so bx and by, this has to be between 0 and 1, right? Because pretty much by definition that orange dot is within the bounds of that grid cell is assigned to. If it wasn't between 0 and 1 it was outside the square, then we'll have been assigned to a different grid cell. But these could be greater than one. In particular if you have a car where the bounding box was that, then the height and width of the bounding box, this could be greater than one. So, there are multiple ways of specifying the bounding boxes, but this would be one convention that's quite reasonable. Although, if you read the YOLO research papers, the YOLO research line there were other parameterizations that work even a little bit better, but I hope this gives one reasonable condition that should work okay. Although, there are some more complicated parameterizations involving sigmoid functions to make sure this is between 0 and 1. And using an explanation parameterization to make sure that these are non-negative, since 0.9, 0.5, this has to be greater or equal to zero. There are some other more advanced parameterizations that work things a little bit better, but the one you saw here should work okay. So, that's it for the **YOLO** or the **You Only Look Once** algorithm. And in the next few videos I'll show you a few other ideas that will help make this algorithm even better. In the meantime, if you want, you can take a look at YOLO paper reference at the bottom of these past couple slides I use. Although, just one warning, if you take a look at these papers which is the YOLO paper is one of the harder papers to read. I remember, when I was reading this paper for the first time, I had a really hard time figuring out what was going on. And I wound up asking a couple of my friends, very good researchers to help me figure it out, and even they had a hard time understanding some of the details of the paper. So, if you look at the paper, it's okay if you have a hard time figuring it out. I wish it was more uncommon, but it's not that uncommon, sadly, for even senior researchers, that review research papers and have a hard time figuring out the details. And have to look at open source code, or contact the authors, or something else to figure out the details of these outcomes. But don't let me stop you from taking a look at the paper yourself though if you wish, but this is one of the harder ones. So, that though, you now understand the basics of the YOLO algorithm. Let's go on to some additional pieces that will make this algorithm work even better.

Intersection Over Union

So how do you tell if your object detection algorithm is working well? In this section, we'll learn about a function called, "**Intersection Over Union**". and as we use both for evaluating your object detection algorithm, as well as in the next section, using it to add another component to your object detection algorithm, to make it work even better. Let's get started. In the object detection task, you expected to localize the object as well.

Evaluating object localization



Intersection over Union (IoU)

$$= \frac{\text{Size of}}{\text{Size of}}$$

"Correct" if IoU $\geq$ 0.5 ←

0.6 ←

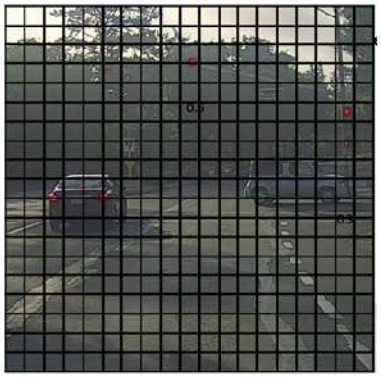
More generally, IoU is a measure of the overlap between two bounding boxes.

So if that's the ground-truth bounding box, and if your algorithm outputs this bounding box in purple, is this a good outcome or a bad one? So what the intersection over union function does, or IoU does, is it computes the intersection over union of these two bounding boxes. So, the union of these two bounding boxes is this area, is really the area that is contained in either bounding boxes, whereas the intersection is this smaller region here. So what the intersection of a union does is it computes the size of the intersection. So that orange shaded area, and divided by the size of the union, which is that green shaded area. And by convention, the low compute division task will judge that your answer is correct if the IoU is greater than 0.5. And if the predicted and the ground-truth bounding boxes overlapped perfectly, the IoU would be one, because the intersection would equal to the union. But in general, so long as the IoU is greater than or equal to 0.5, then the answer will look okay, look pretty decent. And by convention, very often 0.5 is used as a threshold to judge as whether the predicted bounding box is correct or not. This is just a convention. If you want to be more stringent, you can judge an answer as correct, only if the IoU is greater than equal to 0.6 or some other number. But the higher the IoUs, the more accurate the bounding the box. And so, this is one way to map localization, to accuracy where you just count up the number of times an algorithm correctly detects and localizes an object where you could use a definition like this, of whether or not the object is correctly localized. And again 0.5 is just a human chosen convention. There's no particularly deep theoretical reason for it. You can also choose some other threshold like 0.6 if you want to be more stringent. I sometimes see people use more stringent criteria like 0.6 or maybe 0.7. I rarely see people drop the threshold below 0.5. Now, what motivates the definition of IoU, as a way to evaluate whether or not your object localization algorithm is accurate or not. But more generally, IoU is a measure of the overlap between two bounding boxes. Where if you have two boxes, you can compute the intersection, compute the union, and take the ratio of the two areas. And so this is also a way of measuring how similar two boxes are to each other. And we'll see this use again this way in the next video when we talk about non-max suppression. So that's it for IoU or Intersection over Union. Not to be confused with the promissory note concept in IoU, where if you lend someone money they write you a note that says, " Oh I owe you this much money," so that's also called an IoU. It's totally a different concept, that maybe it's cool that these two things have a similar name. So now, onto this definition of IoU, Intersection of Union. In the next section, we'll discuss non-max suppression, which is a tool you can use to make the outputs of YOLO work even better.

Non-max Suppression

One of the problems of Object Detection as you've learned about this so far, is that your algorithm may find multiple detections of the same objects. Rather than detecting an object just once, it might detect it multiple times. Non-max suppression is a way for you to make sure that your algorithm detects each object only once.

Non-max suppression algorithm



19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3

Discard all boxes with $p_c \leq 0.6$

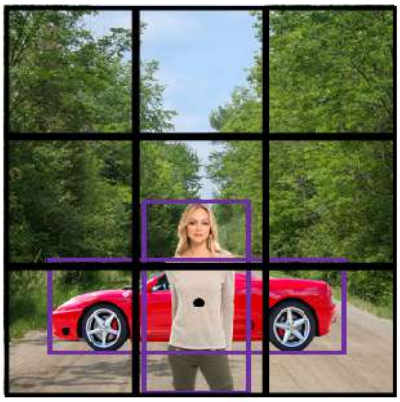
While there are any remaining boxes:

- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

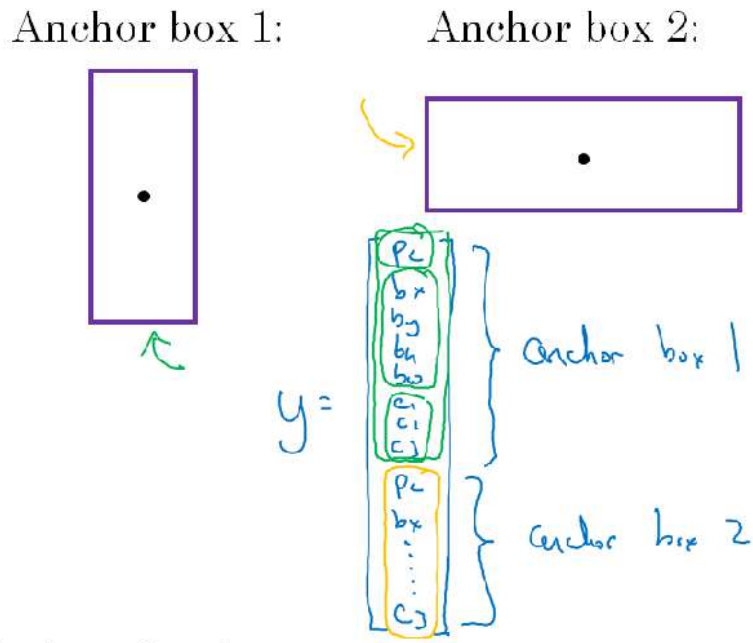
Anchor Boxes

One of the problems with object detection as you have seen it so far is that each of the grid cells can detect only one object. What if a grid cell wants to detect multiple objects? Here is what you can do. You can use the idea of anchor boxes. Let's start with an example. Let's say you have an image (see below).

Overlapping objects:



$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$



[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

And for this example, I am going to continue to use a 3 by 3 grid. Notice that the midpoint of the pedestrian and the midpoint of the car are in almost the same place and both of them fall into the same grid cell. So, for that grid cell, if Y outputs this vector where you are detecting three causes, pedestrians, cars and motorcycles, it won't be able to output two detections. So I have to pick one of the two detections to output. With the idea of anchor boxes, what you are going to do, is pre-define two different shapes called, anchor boxes or anchor box shapes. And what you are going to do is now, be able to associate two predictions with the two anchor boxes. And in general, you might use more anchor boxes, maybe five or even more. But for this video, I am just going to use two anchor boxes just to make the description easier. So what you do is you define the cross label to be, instead of this vector on the left, you basically repeat this twice. S, you will have PC, PX, PY, PH, PW, C1, C2, C3, and these are the eight outputs associated with anchor box 1. And then you repeat that PC, PX and so on down to C1, C2, C3, and other eight outputs associated with anchor box 2. So, because the shape of the pedestrian is more similar to the shape of anchor box 1 and anchor box 2, you can use these eight numbers to encode that PC as one, yes there is a pedestrian. Use this to encode the bounding box around the pedestrian, and then use this to encode that that object is a pedestrian. And then because the box around the car is more similar to the shape of anchor box 2 than anchor box 1, you can then use this to encode that the second object here is the car, and have the bounding box and so on be all the parameters associated with the detected car.

So to summarize, previously, before you are using anchor boxes, you did the following, which is for each object in the training set and the training set image, it was assigned to the grid cell that corresponds to that object's midpoint. And so the output Y was 3 by 3 by 8 because you have a 3 by 3 grid. And for each grid position, we had that output vector which is PC, then the bounding box, and C1, C2, C3. With the anchor box, you now do that following. Now, each object is assigned to the same grid cell as before, assigned to the grid cell that contains the object's midpoint, but it is assigned to a grid cell and anchor box with the highest IoU with the object's shape.

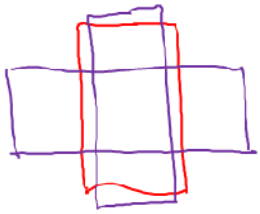
Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y:

$3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box)

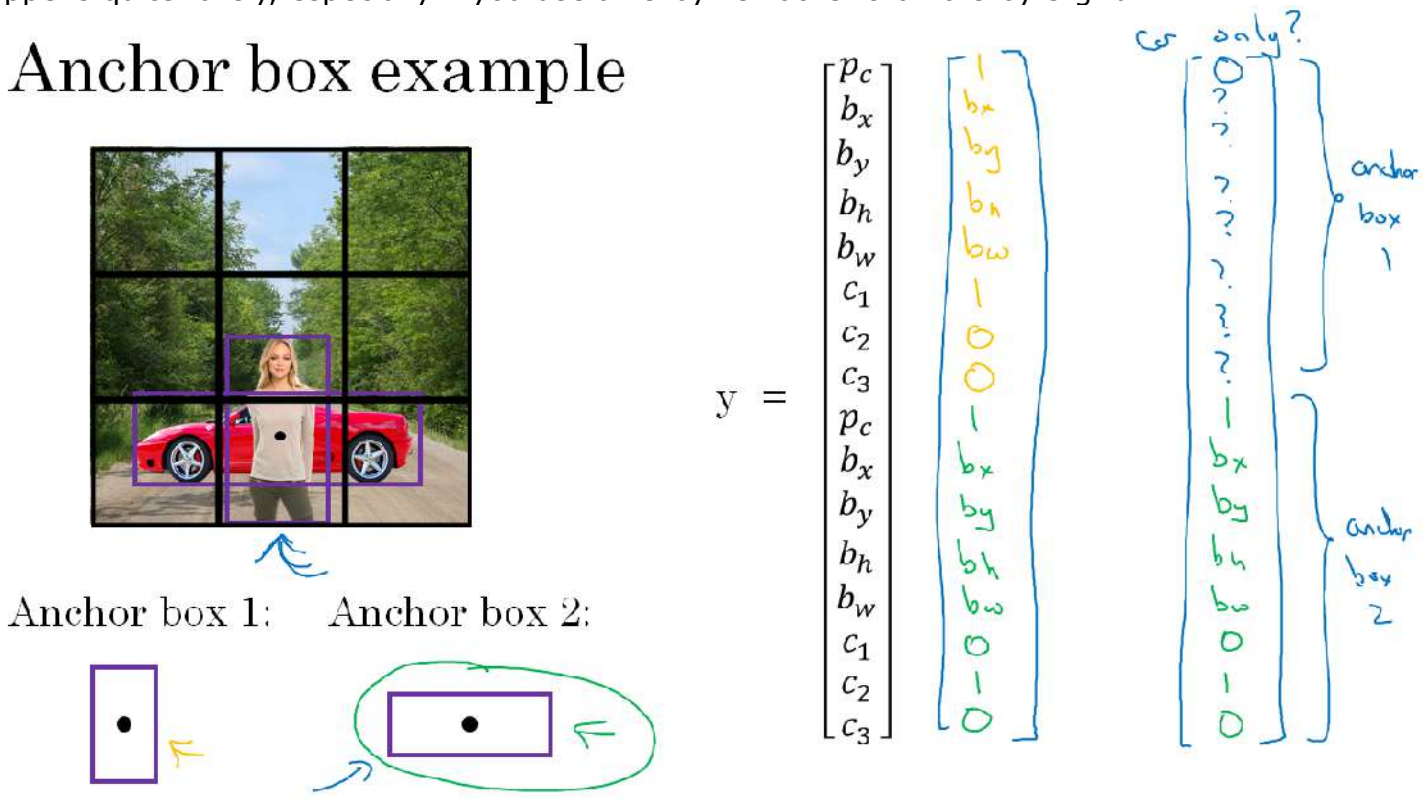
Output y:

$3 \times 3 \times 16$

$3 \times 3 \times 2 \times 8$

So, you have two anchor boxes, you will take an object and see. So if you have an object with this shape, what you do is take your two anchor boxes. Maybe one anchor box is this this shape that's anchor box 1, maybe anchor box 2 is this shape, and then you see which of the two anchor boxes has a higher IoU, will be drawn through bounding box. And whichever it is, that object then gets assigned not just to a grid cell but to a pair. It gets assigned to grid cell comma anchor box pair. And that's how that object gets encoded in the target label. And so now, the output Y is going to be 3 by 3 by 16. Because as you saw on the previous slide, Y is now 16 dimensional. Or if you want, you can also view this as 3 by 3 by 2 by 8 because there are now two anchor boxes and Y is eight dimensional. And dimension of Y being eight was because we have three objects causes if you have more objects than the dimension of Y would be even higher. So let's go through a complete example. For this grid cell, let's specify what is Y. So the pedestrian is more similar to the shape of anchor box 1. So for the pedestrian, we're going to assign it to the top half of this vector. So yes, there is an object, there will be some bounding box associated at the pedestrian. And I guess if a pedestrian is cos one, then we see one as one, and then zero, zero. And then the shape of the car is more similar to anchor box 2. And so the rest of this vector will be one and then the bounding box associated with the car, and then the car is C2, so there's zero, one, zero. And so that's the label Y for that lower middle grid cell that this arrow was pointing to. Now, what if this grid cell only had a car and had no pedestrian? If it only had a car, then assuming that the shape of the bounding box around the car is still more similar to anchor box 2, then the target label Y, if there was just a car there and the pedestrian had gone away, it will still be the same for the anchor box 2 component. Remember that this is a part of the vector corresponding to anchor box 2. And for the part of the vector corresponding to anchor box 1, what you do is you just say there is no object there. So PC is zero, and then the rest of these will be don't cares. Now, just some additional details. What if you have two anchor boxes but three objects in the same grid cell? That's one case that this algorithm doesn't handle well. Hopefully, it won't happen. But if it does, this algorithm doesn't have a great way of handling it. I will just influence some default tiebreaker for that case. Or what if you have two objects associated with the same grid cell, but both of them have the same anchor box shape? Again, that's another case that this algorithm doesn't handle well. If you influence some default way of tiebreaking if that happens, hopefully this won't happen with your data set, it won't happen much at all. And so, it shouldn't affect performance as much. So, that's it for anchor boxes. And even though I'd motivated anchor boxes

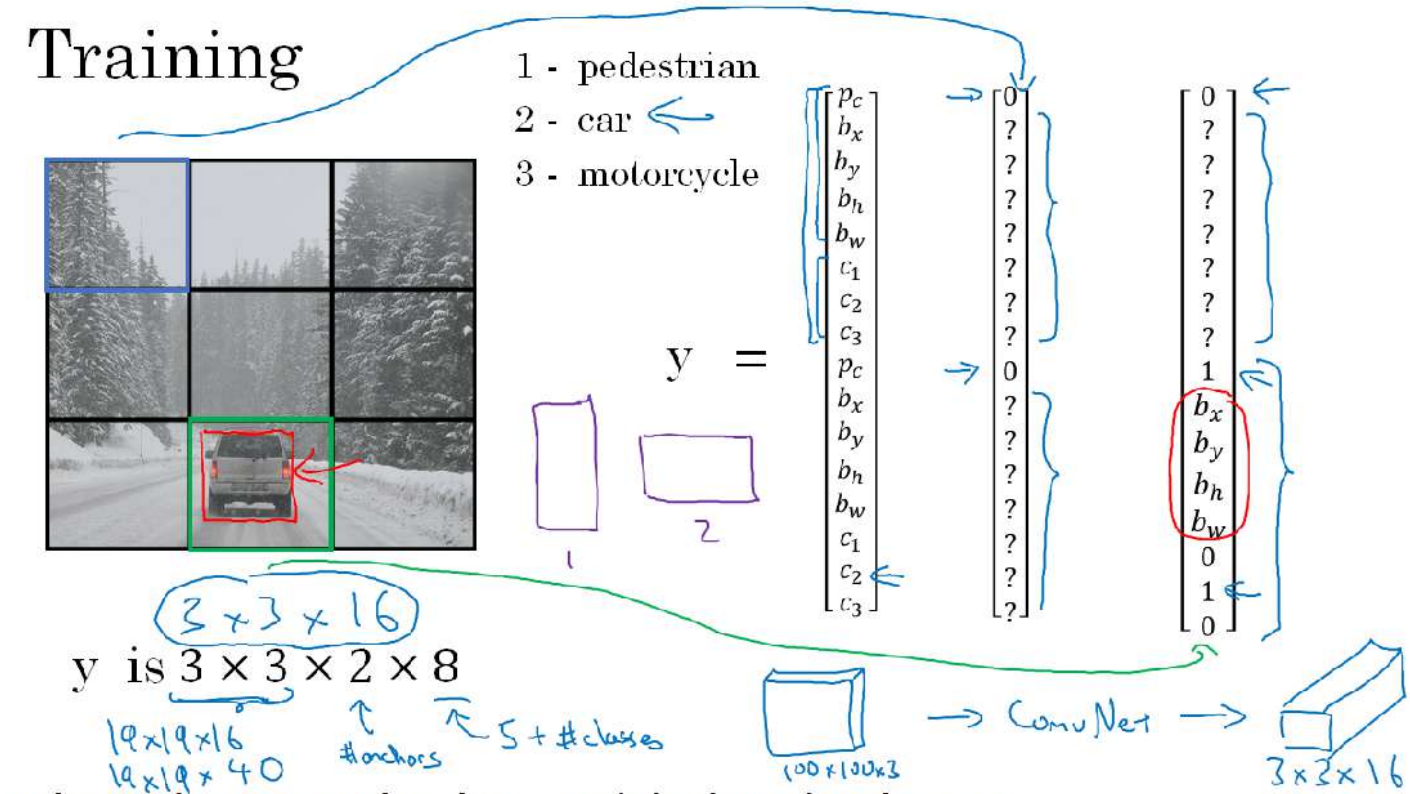
as a way to deal with what happens if two objects appear in the same grid cell, in practice, that happens quite rarely, especially if you use a 19 by 19 rather than a 3 by 3 grid.



The chance of two objects having the same midpoint rather these 361 cells, it does happen, but it doesn't happen that often. Maybe even better motivation or even better results that anchor boxes gives you is it allows your learning algorithm to specialize better. In particular, if your data set has some tall, skinny objects like pedestrians, and some white objects like cars, then this allows your learning algorithm to specialize so that some of the outputs can specialize in detecting white, fat objects like cars, and some of the output units can specialize in detecting tall, skinny objects like pedestrians. So finally, how do you choose the anchor boxes? And people used to just choose them by hand or choose maybe five or 10 anchor box shapes that spans a variety of shapes that seems to cover the types of objects you seem to detect. As a much more advanced version, just in the advance common for those of who have other knowledge in machine learning, and even better way to do this in one of the later YOLO research papers, is to use a K-means algorithm, to group together two types of objects shapes you tend to get. And then to use that to select a set of anchor boxes that this most stereotypically representative of the maybe multiple, of the maybe dozens of object causes you're trying to detect. But that's a more advanced way to automatically choose the anchor boxes. And if you just choose by hand a variety of shapes that reasonably expands the set of object shapes, you expect to detect some tall, skinny ones, some fat, white ones. That should work with these as well. So that's it for anchor boxes. In the next video, let's take everything we've seen and tie it back together into the YOLO algorithm.

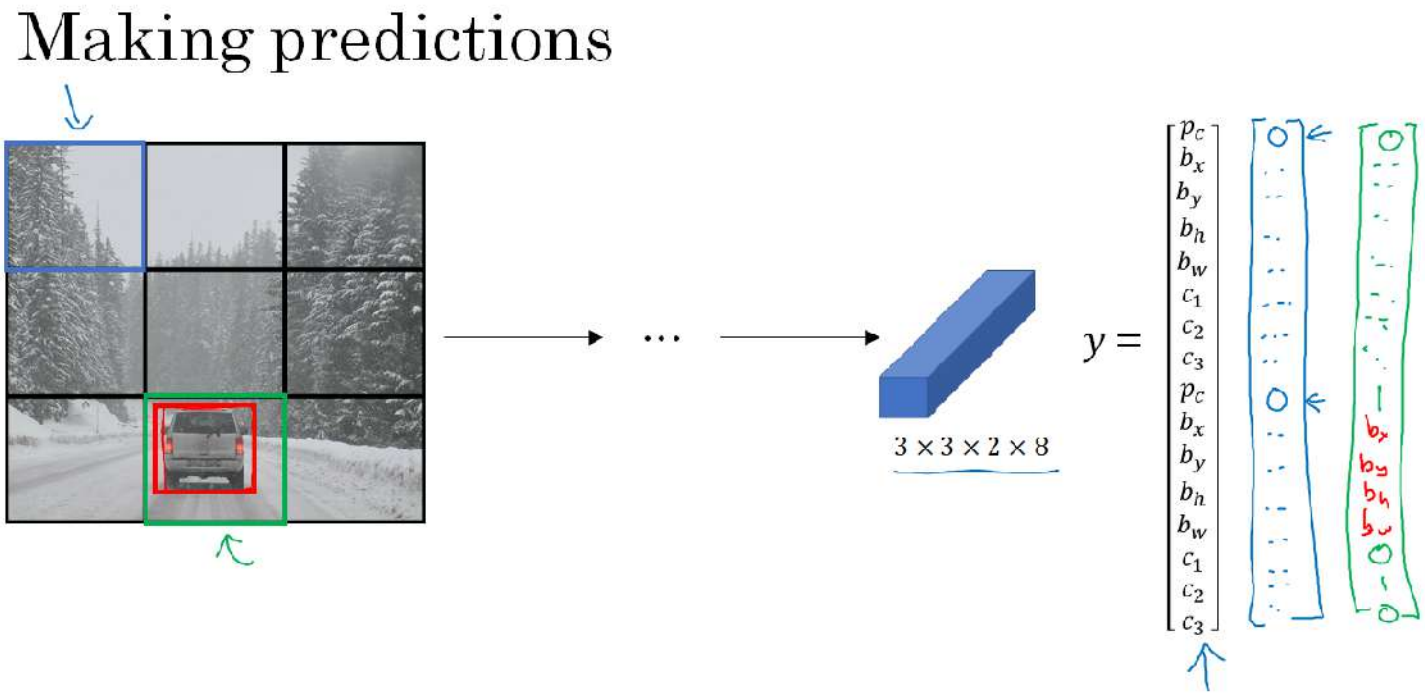
YOLO Algorithm

We've already seen most of the components of object detection. In this section, let's put all the components together to form the **YOLO object detection algorithm**. First, let's see how you construct your training set. Suppose you're trying to train an algorithm to detect three objects: pedestrians, cars, and motorcycles.



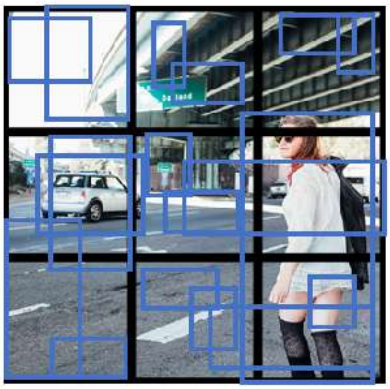
[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

And you will need to explicitly have the full background class, so just the class labels here. If you're using two anchor boxes, then the outputs y will be three by three because you are using three by three grid cell, by two, this is the number of anchors, by eight because that's the dimension of this. Eight is actually five which is plus the number of classes. So five because you have Pc and then the bounding boxes, that's five, and then C1, C2, C3. That dimension is equal to the number of classes. And you can either view this as three by three by two by eight, or by three by three by sixteen. So to construct the training set, you go through each of these nine grid cells and form the appropriate target vector y. So take this first grid cell, there's nothing worth detecting in that grid cell. None of the three classes pedestrian, car and motocycle, appear in the upper left grid cell and so, the target y corresponding to that grid cell would be equal to this. Where Pc for the first anchor box is zero because there's nothing associated for the first anchor box, and is also zero for the second anchor box and so on all of these other values are don't cares. Now, most of the grid cells have nothing in them, but for that box over there, you would have this target vector y. So assuming that your training set has a bounding box like this for the car, it's just a little bit wider than it is tall. And so if your anchor boxes are that, this is a anchor box one, this is anchor box two, then the red box has just slightly higher IoU with anchor box two. And so the car gets associated with this lower portion of the vector. So notice then that Pc associate anchor box one is zero. So you have don't cares all these components. Then you have this Pc is equal to one, then you should use these to specify the position of the red bounding box, and then specify that the correct object is class two. Right that it is a car. So you go through this and for each of your nine grid positions each of your three by three grid positions, you would come up with a vector like this. Come up with a 16 dimensional vector. And so that's why the final output volume is going to be 3 by 3 by 16. Oh and as usual for simplicity on the slide I've used a 3 by 3 the grid. In practice it might be more like a 19 by 19 by 16. Or in fact if you use more anchor boxes, maybe 19 by 19 by 5 x 8 because five times eight is 40. So it will be 19 by 19 by 40. That's if you use five anchor boxes. So that's training and you train ConvNet that inputs an image, maybe 100 by 100 by 3, and your ConvNet would then finally output this output volume in our example, 3 by 3 by 16 or 3 by 3 by 2 by 8. Next, let's look at how your algorithm can make predictions. Given an image, your neural network will output this by 3 by 3 by 2 by 8 volume, where for each of the nine grid cells you get a vector like that.



So for the grid cell here on the upper left, if there's no object there, hopefully, your neural network will output zero here, and zero here, and it will output some other values. Your neural network can't output a question mark, can't output a don't care. So I'll put some numbers for the rest. But these numbers will basically be ignored because the neural network is telling you that there's no object there. So it doesn't really matter whether the output is a bounding box or there's is a car. So basically just be some set of numbers, more or less noise. In contrast, for this box over here hopefully, the value of y to the output for that box at the bottom left, hopefully would be something like zero for bounding box one. And then just open a bunch of numbers, just noise. Hopefully, you'll also output a set of numbers that corresponds to specifying a pretty accurate bounding box for the car. So that's how the neural network will make predictions. Finally, you run this through non-max suppression. So just to make it interesting. Let's look at the new test set image. Here's how you would run non-max suppression. If you're using two anchor boxes, then for each of the non-grid cells, you get two predicted bounding boxes. Some of them will have very low probability, very low Pc, but you still get two predicted bounding boxes for each of the nine grid cells. So let's say, those are the bounding boxes you get. And notice that some of the bounding boxes can go outside the height and width of the grid cell that they came from. Next, you then get rid of the low probability predictions. So get rid of the ones that even the neural network says, gee this object probably isn't there. So get rid of those. And then finally if you have three classes you're trying to detect, you're trying to detect pedestrians, cars and motorcycles. What you do is, for each of the three classes, independently run non-max suppression for the objects that were predicted to come from that class. But use non-max suppression for the predictions of the pedestrians class, run non-max suppression for the car class, and non-max suppression for the motorcycle class.

Outputting the non-max supressed outputs



- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

But run that basically three times to generate the final predictions. And so the output of this is hopefully that you will have detected all the cars and all the pedestrians in this image. So that's it for the YOLO object detection algorithm. Which is really one of the most effective object detection algorithms, that also encompasses many of the best ideas across the entire computer vision literature that relate to object detection.

Week 4: Special applications: Face recognition & Neural style transfer

Discover how CNNs can be applied to multiple fields, including art generation and face recognition. Implement your own algorithm to generate art and recognize faces!

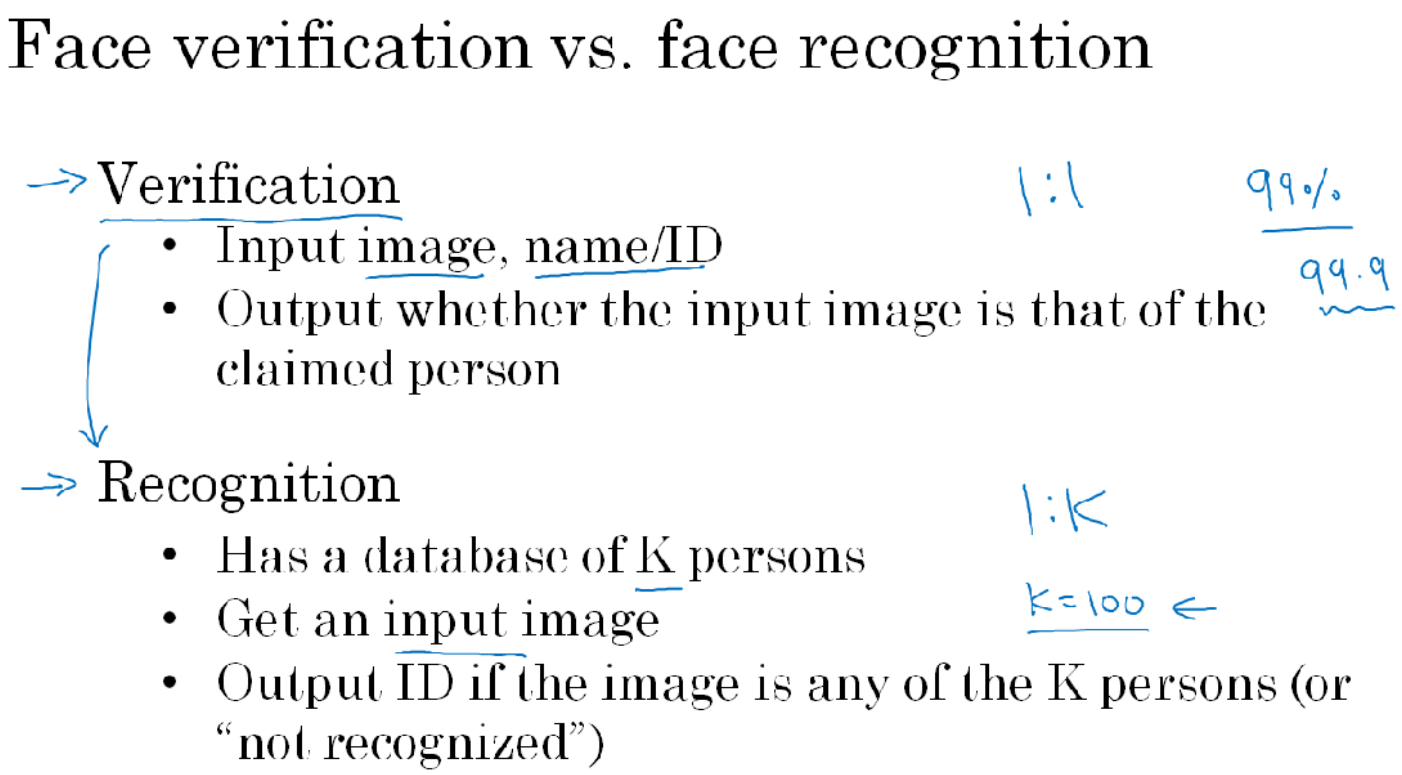
Face Recognition

What is face recognition?

We've already learned a lot about convnet. I this week we'll see a couple of importatnt applications of convnets. We'll start with the **face recognition**, and then go on later this week to neural style transfer. But first, let's start the face recognition and just for fun, I want to show you a demo. When I was leading by those AI group, one of the teams I worked with led by Yuanqing Lin had built a face recognition system that I thought is really cool. Let's take a look. So, I'm going to play this video here, but I can also get whoever is editing this raw video configure out to this better to splice in the raw video or take the one I'm playing here. I want to show you a face recognition demo. I'm in Baidu's headquarters in China. Most companies require that to get inside, you swipe an ID card like this one but here we don't need that. Using face recognition, check what I can do. When I walk up, it recognizes my face, it says, "Welcome Andrew," and I just walk right through without ever having to use my ID card. Let me show you something else. I'm actually here with Lin Yuanqing, the director of IDL which developed all of this face recognition technology. I'm gonna hand him my ID card, which has my face printed on it, and he's going to use it to try to sneak in using my picture instead of a live human. I'm gonna use Andrew's card and try to sneak in and see what happens. So the system is not recognizing it, it refuses to recognize. Okay. Now, I'm going to use my own face. So face recognition technology like this is taking off very rapidly in China and I hope that this type of technology soon makes it way to other countries.. So, pretty cool, right? The video you just saw demoed both face recognition as well as liveness detection. The latter meaning making sure that you are a live human. It turns out liveness detection can be implemented using supervised learning as well to predict live human versus not live human but I want to spend less time on that. Instead, I want to focus our time on talking about how to build the face recognition portion of the system.

First, let's start by going over some of the terminology used in **face recognition**. In the face recognition literature, people often talk about **face verification** and **face recognition**. This is the

face verification problem which is if you're given an input image as well as a name or ID of a person and the job of the system is to verify whether or not the input image is that of the claimed person. So, sometimes this is also called a one to one problem where you just want to know if the person is the person they claim to be. So, the **recognition problem is much harder than the verification problem**.

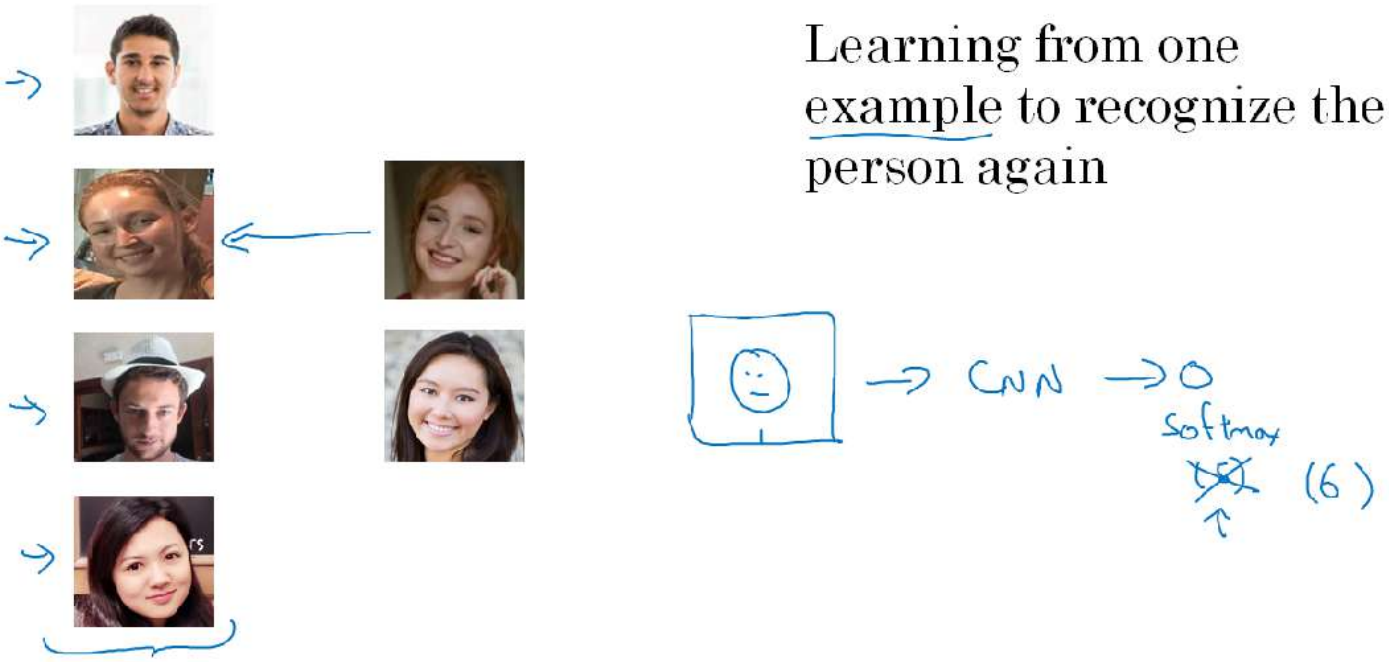


To see why, let's say, you have a verification system that's 99 percent accurate. So, 99 percent might not be too bad but now suppose that K is equal to 100 in a recognition system. If you apply this system to a recognition task with a 100 people in your database, you now have a hundred times of chance of making a mistake and if the chance of making mistakes on each person is just one percent. So, if you have a database of a 100 persons and if you want an acceptable recognition error, you might actually need a verification system with maybe 99.9 or even higher accuracy before you can run it on a database of 100 persons that have a high chance and still have a high chance of getting incorrect. In fact, if you have a database of 100 persons currently just be even quite a bit higher than 99 percent for that to work well. But what we do in the next few chapters is focus on building a face verification system as a building block and then if the accuracy is high enough, then you probably use that in a recognition system as well.

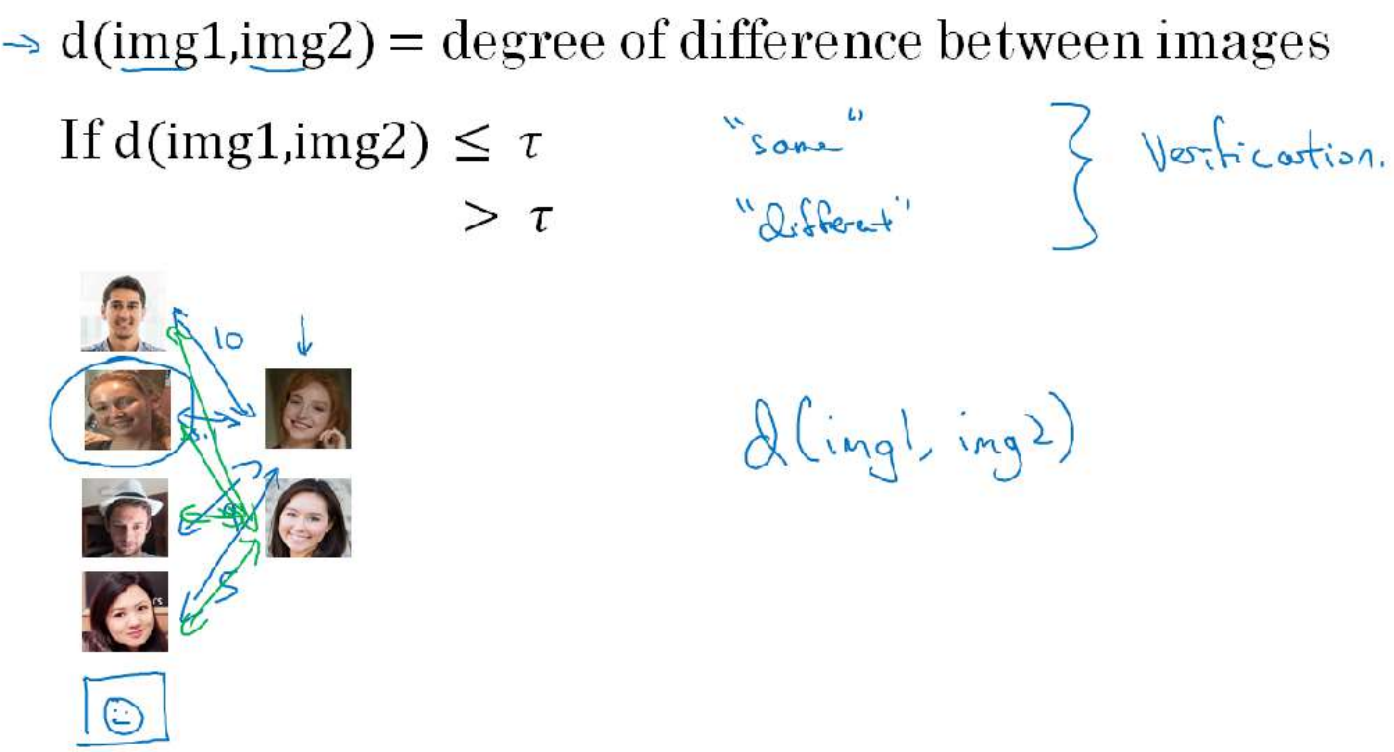
One Shot Learning

One of the challenges of face recognition is that you need to solve the **one-shot** learning problem. What that means is that for most face recognition applications you need to be able to recognize a person given just one single image, or given just one example of that person's face and, historically, deep learning algorithms don't work well if you have only one training example. Let's see an example of what this means and talk about how to address this problem. Let's say you have a database of four pictures of employees in you're organization.

One-shot learning



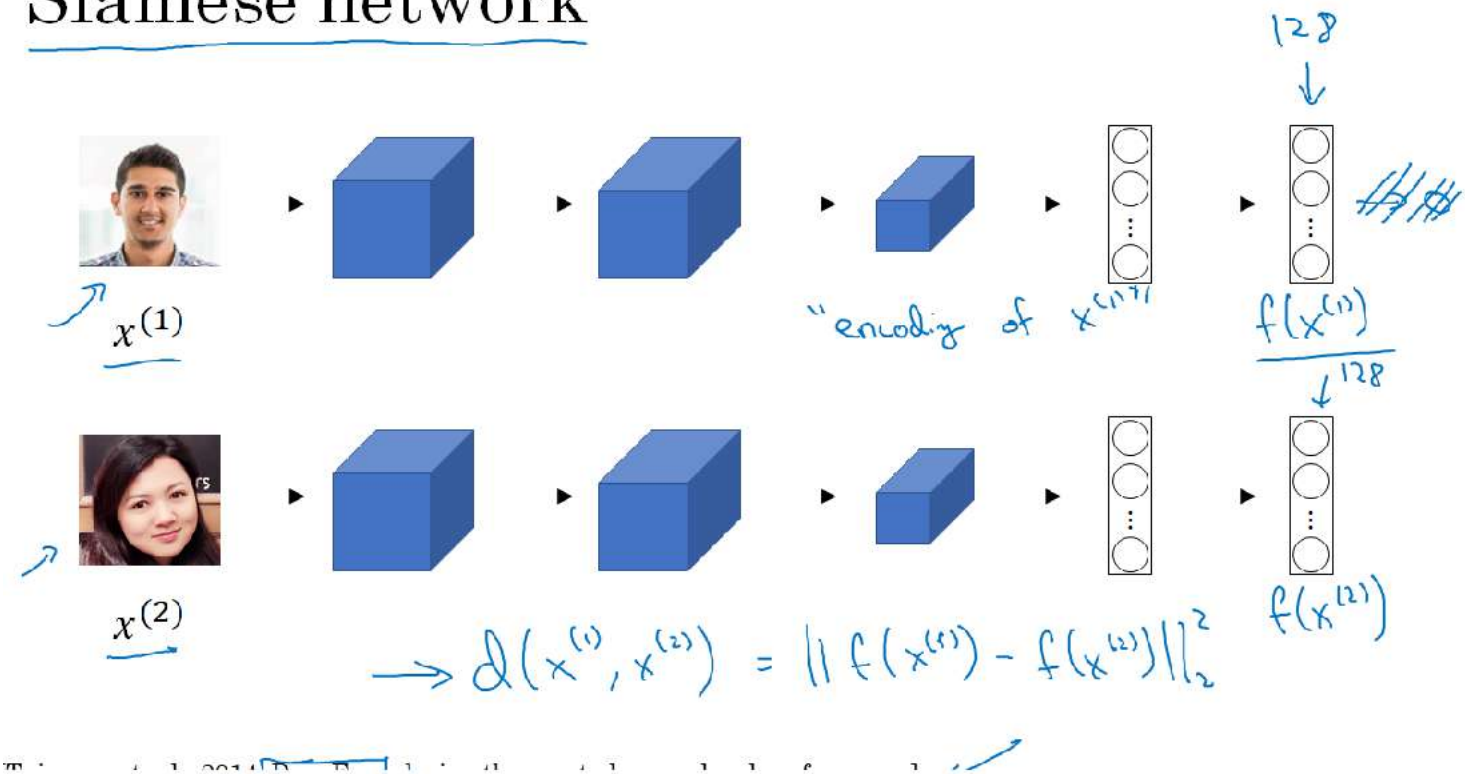
These are actually some of my colleagues at Deeplearning.AI, Khan, Danielle, Younes and Thian. Now let's say someone shows up at the office and they want to be let through the turnstile. What the system has to do is, despite ever having seen only one image of Danielle, to recognize that this is actually the same person and, in contrast, if it sees someone that's not in this database, then it should recognize that this is not any of the four persons in the database. So in the **one shot learning problem, you have to learn from just one example to recognize the person again** and you need this for most face recognition systems because you might have only one picture of each of your employees or of your team members in your employee database. So one approach you could try is to input the image of the person, feed it to a ConvNet and having a output label, y, using a softmax unit with four outputs or maybe five outputs corresponding to each of these four persons or none of the above. So that would be 5 outputs in the softmax. But this really doesn't work well. Because if you have such a small training set it is really not enough to train a robust neural network for this task and also what if a new person joins your team? So now you have 5 persons you need to recognize, so there should now be six outputs. Do you have to retrain the ConvNet every time? That just doesn't seem like a good approach. So to carry out face recognition, to carry out one-shot learning. So instead, to make this work, what you're going to do instead is learn a similarity function. In particular, you want a neural network to learn a function which going to denote d, which inputs two images and outputs the degree of difference between the two images. So if the two images are of the same person, you want this to output a small number and if the two images are of two very different people you want it to output a large number. So during recognition time, if the degree of difference between them is less than some threshold called tau, which is a hyperparameter. Then you would predict that these two pictures are the same person and if it is greater than tau, you would predict that these are different persons and so this is how you address the face verification problem.



To use this for a recognition task, what you do is, given this new picture, you will use this function d to compare these two images and maybe I'll output a very large number, let's say 10, for this example. And then you compare this with the second image in your database. And because these two are the same person, hopefully you output a very small number. You do this for the other images in your database and so on and based on this, you would figure out that this is actually that person, which is Danielle and in contrast, if someone not in your database shows up, as you use the function d to make all of these pairwise comparisons, hopefully d will output have a very large number for all four pairwise comparisons and then you say that this is not any one of the four persons in the database. Notice how this allows you to solve the one-shot learning problem. So long as you can learn this function d, which inputs a pair of images and tells you, basically, if they're the same person or different persons. Then if you have someone new join your team, you can add a fifth person to your database, and it just works fine. So you've seen how learning this function d, which inputs two images, allows you to address the one-shot learning problem. In the next section, let's take a look at how you can actually train the neural network to learn dysfunction d.

Siamese Network

The job of the function d, which you learned about in the last section, is to input two faces and tell you how similar or how different they are. A good way to do this is to use a **Siamese network**. Let's take a look. You're used to seeing pictures of convnet like these where you input an image, let's say x1. And through a sequence of convolutional and pulling and fully connected layers, end up with a feature vector.



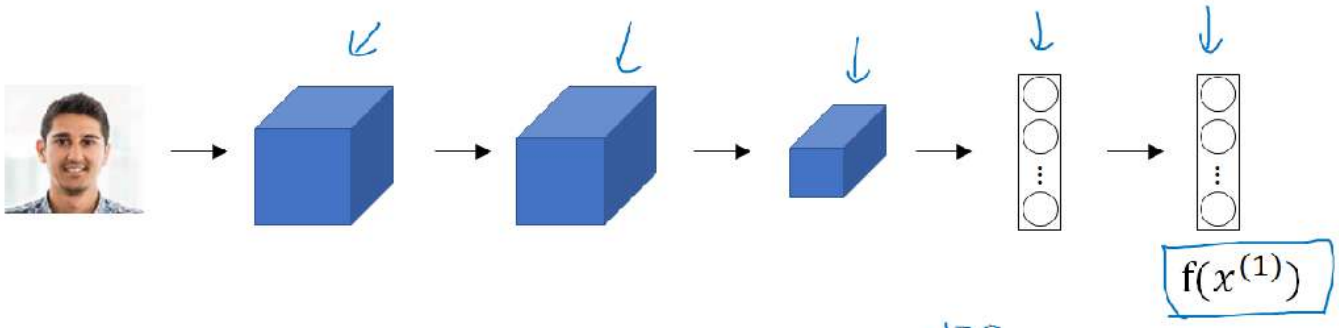
And sometimes this is fed to a softmax unit to make a classification. We're going to focus on this vector of let's say 128 numbers computed by some fully connected layer that is deeper in the network.

And I'm going to give this list of 128 numbers a name. I'm going to call this $f(x^{(1)})$, and you should think of $x^{(1)}$ as an encoding of the input image $x^{(1)}$. So it's taken the input image, here this picture of Kian, and is re-representing it as a vector of 128 numbers. The way you can build a face recognition system is then that if you want to compare two pictures, let's say this first picture with this second picture here. What you can do is feed this second picture to the same neural network with the same parameters and get a different vector of 128 numbers, which encodes this second picture. So we will this second picture.

So we're going to call this encoding of this second picture f of $x^{(2)}$, and here we're using $x^{(1)}$ and $x^{(2)}$ just to denote two input images. They don't necessarily have to be the first and second examples in your training sets. It can be any two pictures. Finally, if you believe that these encodings are a good representation of these two images, what you can do is then define the image d of distance between $x^{(1)}$ and $x^{(2)}$ as the norm of the difference between the encodings of these two images. So this idea of running two identical, convolutional neural networks on two different inputs and then comparing them, sometimes that's called a **Siamese neural network architecture** and a lot of the ideas presented here came from this paper due to Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf in the research system that they developed called **DeepFace**.

So how do you train this Siamese neural network? Remember that these two neural networks have the same parameters. So what you want to do is really train the neural network so that the encoding that it computes results in a function d that tells you when two pictures are of the same person. So more formally, the parameters of the neural network define an encoding $f(x^{(i)})$.

Goal of learning



Parameters of NN define an encoding $f(x^{(i)})$ — 128

Learn parameters so that:

If $x^{(i)}, x^{(j)}$ are the same person, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is small.

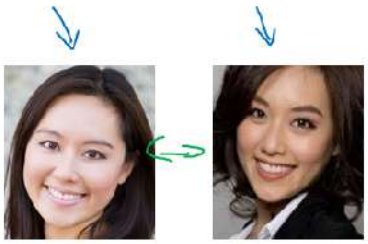
If $x^{(i)}, x^{(j)}$ are different persons, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is large.

So given any input image $x^{(i)}$, the neural network outputs this 128 dimensional encoding $f(x^{(i)})$ So more formally, what you want to do is learn parameters so that if two pictures, $x^{(i)}$ and $x^{(j)}$, are of the same person, then you want that **distance between their encodings to be small** and in the previous section, we were using $x^{(1)}$ and $x^{(2)}$, but it's really any pair $x^{(i)}$and $x^{(j)}$ from your training set and in contrast, if $x^{(i)}$ and $x^{(j)}$ are of different persons, then you want that **distance between their encodings to be large**. So as you vary the parameters in all of these layers of the neural network, you end up with different encodings and what you can do is use back propagation and vary all those parameters in order to make sure these conditions are satisfied. So in this section we've learned about the Siamese network architecture and have a sense of what you want the neural network to output for you in terms of what would make a good encoding. But how do you actually define an objective function to make a neural network learn to do what we just discussed here? Let's see how you can do that in the next section using the triplet loss function.

Triplet Loss

One way to learn the parameters of the neural network so that it gives you a good encoding for your pictures of faces is to define an applied gradient descent on the triplet loss function. Let's see what that means. To apply the triplet loss, you need to compare pairs of images. For example, given this picture, to learn the parameters of the neural network, you have to look at several pictures at the same time. For example, given this pair of images, you want their encodings to be similar because these are the same person. Whereas, given this pair of images, you want their encodings to be quite different because these are different persons. In the terminology of the **triplet loss**, what you're going do is always look at one anchor image and then you want to distance between the anchor and the positive image, really a positive example, meaning as the same person to be similar. Whereas, you want the anchor when pairs are compared to the negative example for their distances to be much further apart. So, this is what gives rise to the term triplet loss, which is that **you'll always be looking at three images at a time**. **You'll be looking at an anchor image, a positive image, as well as a negative image**.
Check below diagrams to show an example of face recognition using triplet loss.

Learning Objective




Anchor (A) Positive (P) Anchor (A) Negative (N)

$d(A,P) \leq 0.5$ $d(A,N) = 0.7$

Want: $\underbrace{\|f(A)-f(P)\|^2}_{d(A,P)} + \alpha \leq \underbrace{\|f(A)-f(N)\|^2}_{d(A,N)}$ ($\alpha \nearrow 0.2$)

$$\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha \leq 0$$

0 0 margin

$f(\text{img}) = \vec{0}$

[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering]

Loss function

Given 3 images A, P, N:

$$\mathcal{L}(A,P,N) = \max\left(\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha, 0\right)$$

≥ 0 > 0

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

A, P

Training set: 10k pictures of 1k persons

[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering]

Choosing the triplets A,P,N

During training, if A,P,N are chosen randomly, $d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.

$\|f(A) - f(P)\|^2 + \alpha \leq \|f(A) - f(N)\|^2$

Choose triplets that're "hard" to train on.

$d(A,P) + \alpha \leq d(A,N)$

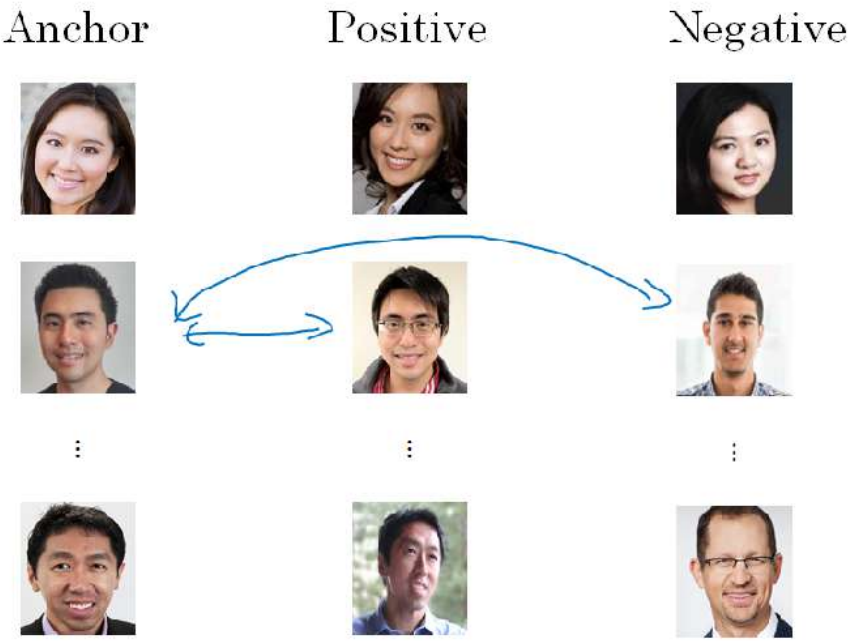
$d(A,P) \approx d(A,N)$

Face Net

Deep Face

[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering]

Training set using triplet loss



J

$d(x^{(i)}, x^{(j)})$

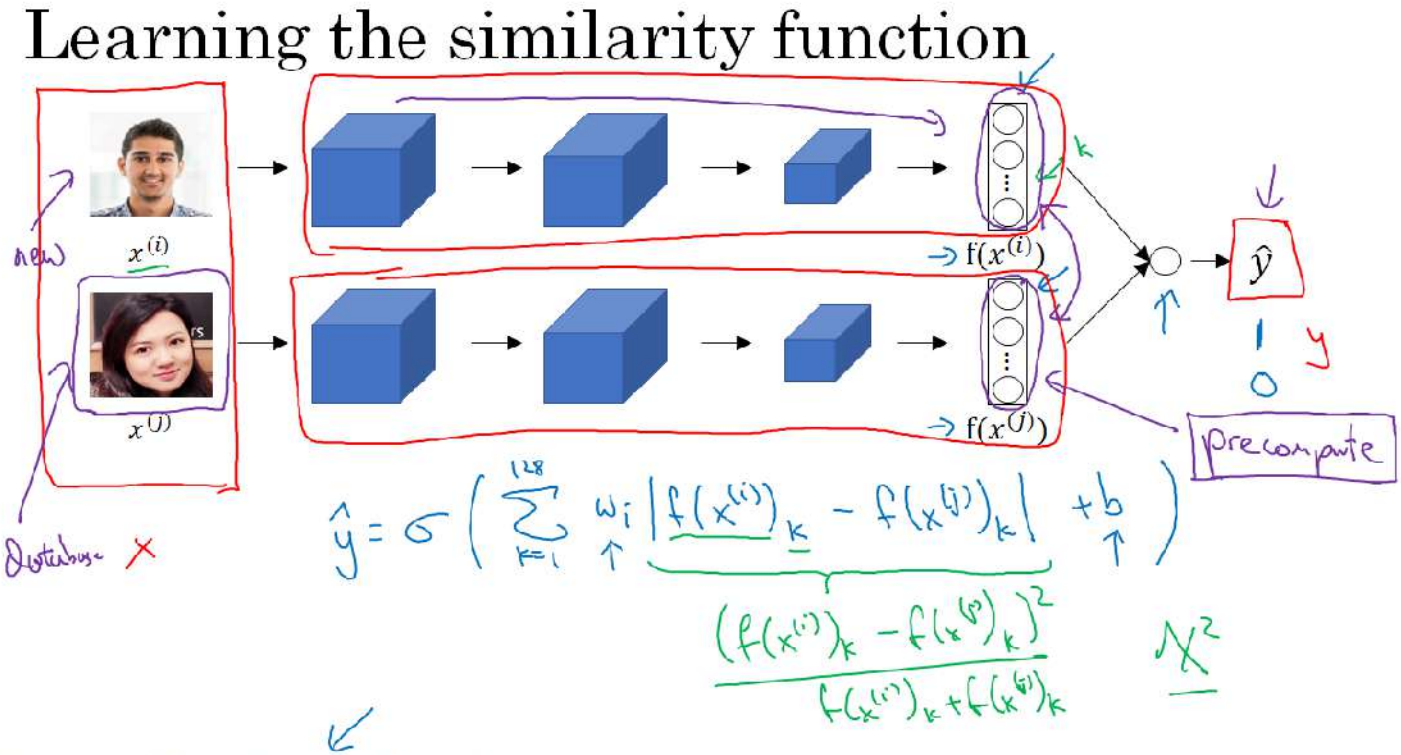
All the details are presented in this paper by Florian Schroff, Dmitry Kalinichenko, and James Philbin, where they have a system called **FaceNet**.

So, just to wrap up, **to train on triplet loss, you need to take your training set and map it to a lot of triples**. So, here (check last diagram) is our triple with an anchor and a positive, both for the same person and the negative of a different person. Here's another one where the

anchor and positive are of the same person but the anchor and negative are of different persons and so on and what you do having defined this training sets of anchor positive and negative triples is **use gradient descent to try to minimize the cost function J we defined on an earlier section and that will have the effect of that propagating to all of the parameters of the neural network in order to learn an encoding so that $d(x^{(i)},x^{(j)})$ of two images will be small when these two images are of the same person, and they'll be large when these are two images of different persons**. So, that's it for the triplet loss and how you can train a neural network for learning and an encoding for face recognition. Now, it turns out that commercial face recognition systems are trained on fairly large datasets at this point. Often, million images and there are some commercial companies talking about using over 100 million images. So these are very large datasets even by modern standards, these dataset assets are not easy to acquire. Fortunately, some of these companies have trained these large networks and posted parameters online. So, rather than trying to train one of these networks from scratch, this is one domain where because of the share data volume sizes, this is one domain where often it might be useful for you to download someone else's pre-train model, rather than do everything from scratch yourself. But even if you do download someone else's pre-train model, I think it's still useful to know how these algorithms were trained or in case you need to apply these ideas from scratch yourself for some application. So that's it for the triplet loss. In the next section, we'll see some other variations on siamese networks and how to train these systems.

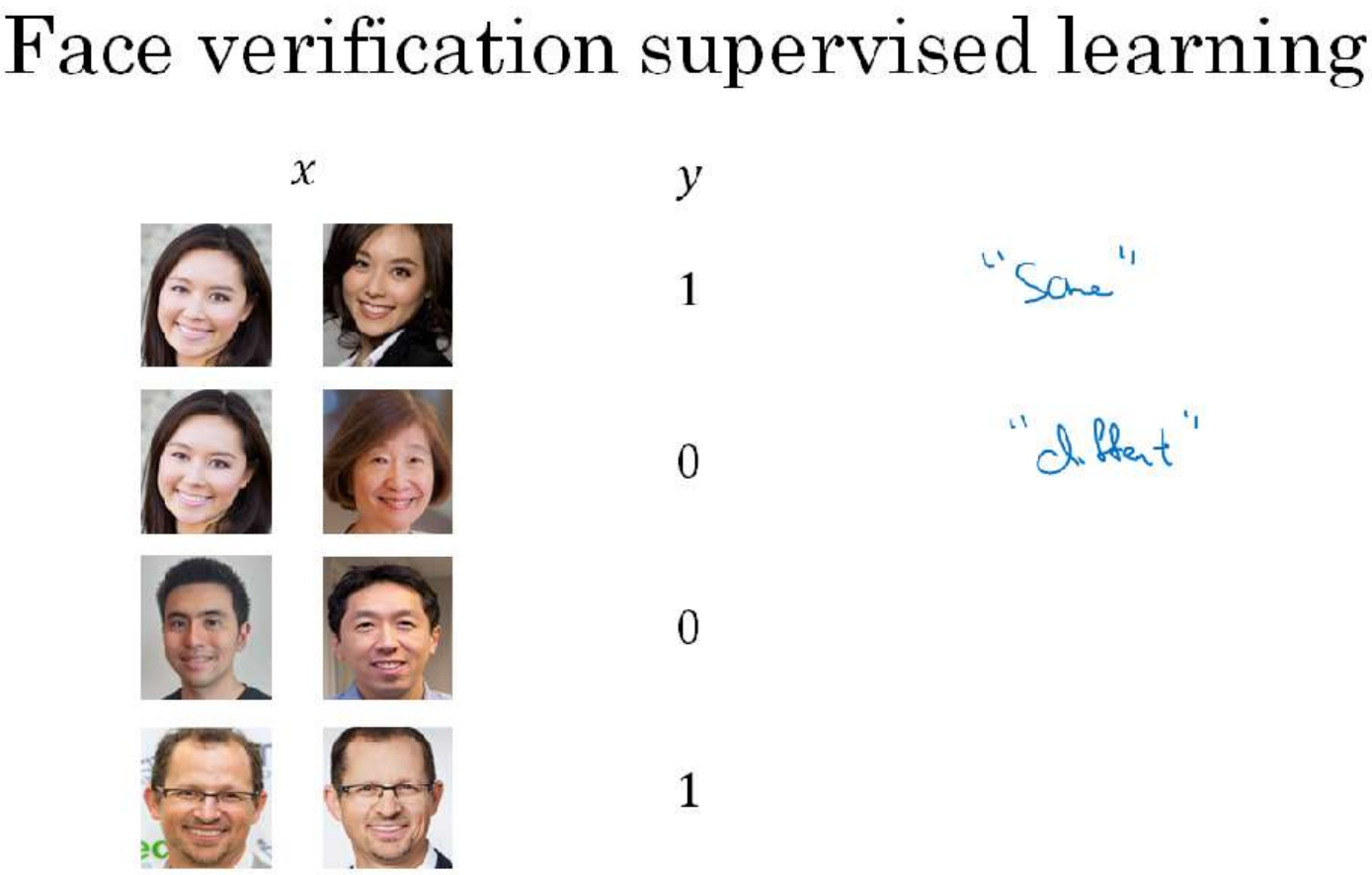
Face Verification and Binary Classification

The Triplet Loss is one good way to learn the parameters of a continent for face recognition. There's another way to learn these parameters. Let me show you how face recognition can also be posed as a straight **binary classification problem**. Another way to train a neural network, is to take this pair of neural networks to take this Siamese Network and have them both compute these embeddings, maybe 128 dimensional embeddings, maybe even higher dimensional, and then have these be input to a **logistic regression** unit to then just make a prediction. Where the target output will be one if both of these are the same persons, and zero if both of these are of different persons. So, this is a way to treat face recognition just as a binary classification problem and this is an alternative to the triplet loss for training a system like this. Now, what does this final logistic regression unit actually do?



[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

The output yhat will be a sigmoid function, applied to some set of features but rather than just feeding in, these encodings, what you can do is take the differences between the encodings.



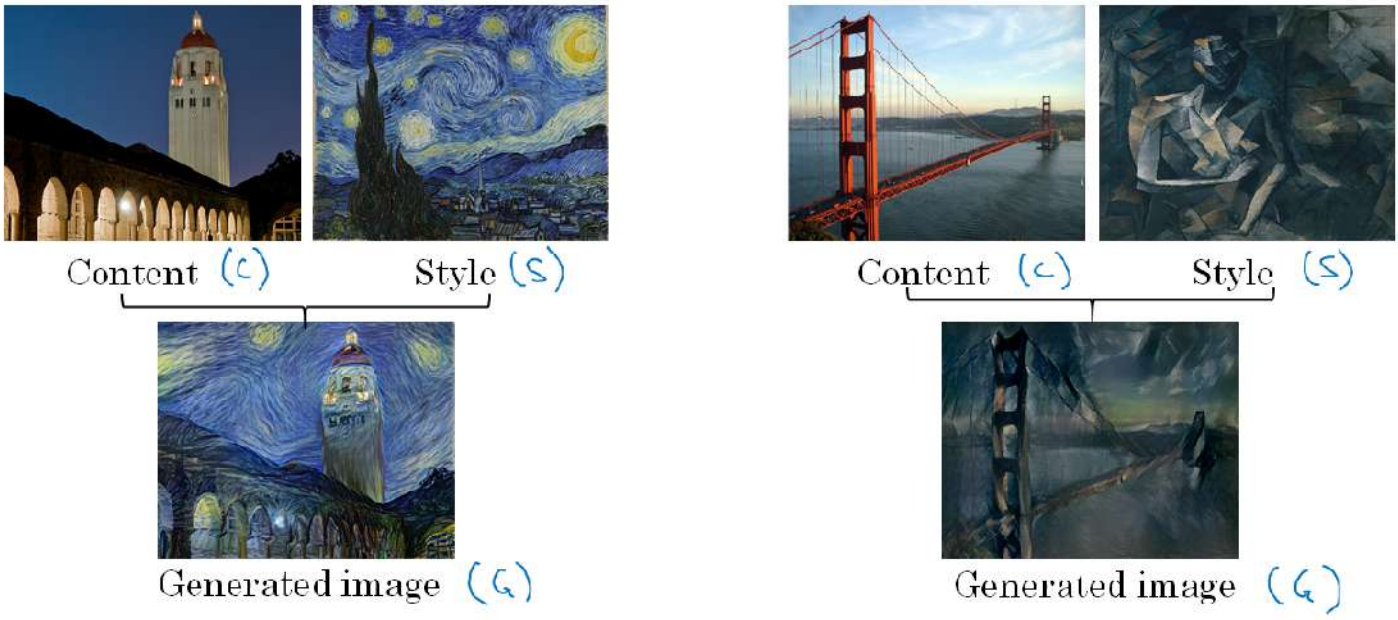
[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

So just to wrap up, to **treat face verification supervised learning, you create a training set of just pairs of images now is of triplets of pairs of images where the target label is one. When these are a pair of pictures of the same person and where the tag label is zero, when these are pictures of different persons and you use different pairs to train the neural network to train the scientists that were using back propagation**. So, this version that you just saw of treating face verification and by extension face recognition as a binary classification problem, this works quite well as well.

Neural Style Transfer

What is neural style transfer?

One of the most fun and exciting applications of ConvNet recently has been **Neural Style Transfer**. What is Neural Style Transfer? Let me take an examples. Let's say you take this image (image 1 in diagram below)



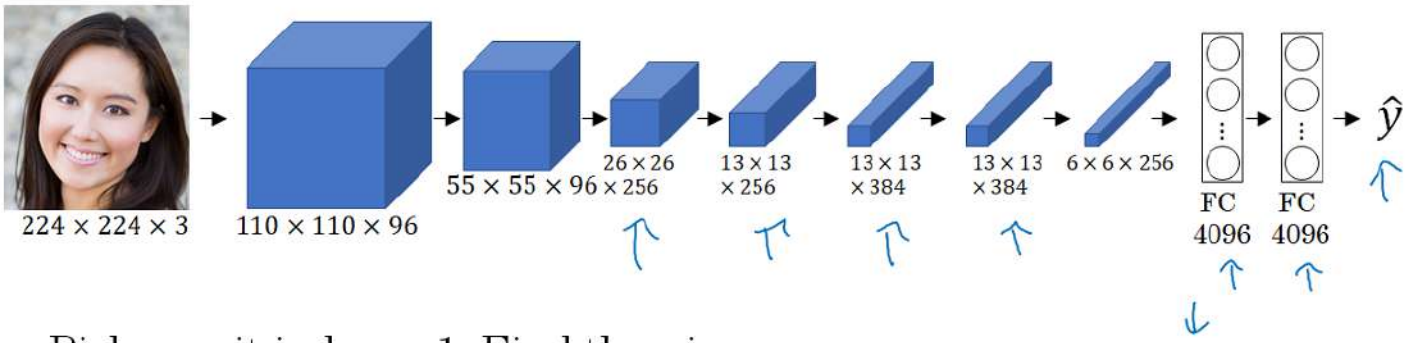
[Images generated by Justin Johnson]

this is actually taken from the Stanford University not far from my Stanford office and you want this picture recreated in the style of this image on the right. This is actually Van Gogh's, Starry Night painting. What Neural Style Transfer allows you to do is generated new image like the one below which is a picture of the Stanford University Campus that painted but drawn in the style of the image on the right. In order to describe how you can implement this yourself, I'm going to use C to denote the content image, S to denote the style image, and G to denote the image you will generate. Here's another example, let's say you have this content image so let's see this is of the Golden Gate Bridge in San Francisco and you have this style image, this is actually Pablo Picasso image. You can then combine these to generate this image G which is the Golden Gate painted in the style of that Picasso shown on the right. What we'll learn in the next few sections is how you can generate these images yourself. In order to implement Neural Style Transfer, you need to look at the features extracted by ConvNet at various layers, the shallow and the deeper layers of a ConvNet.

What are deep ConvNets learning

What are deep ConvNets really learning? In this section, we'll see some visualizations that will help you hold your intuition about what the deeper layers of a ConvNet really are doing and this will help us think through how you can implement neural style transfer as well. Let's start with an example. Lets say you've trained a ConvNet, this is an alex net like network, and you want to visualize what the hidden units in different layers are computing. Here's what you can do.

Visualizing what a deep network is learning



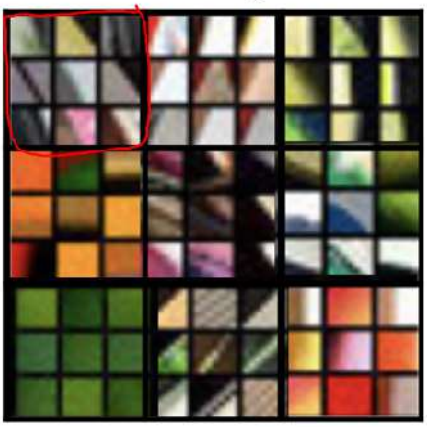
Pick a unit in layer 1. Find the nine image patches that maximize the unit's activation.

Repeat for other units.

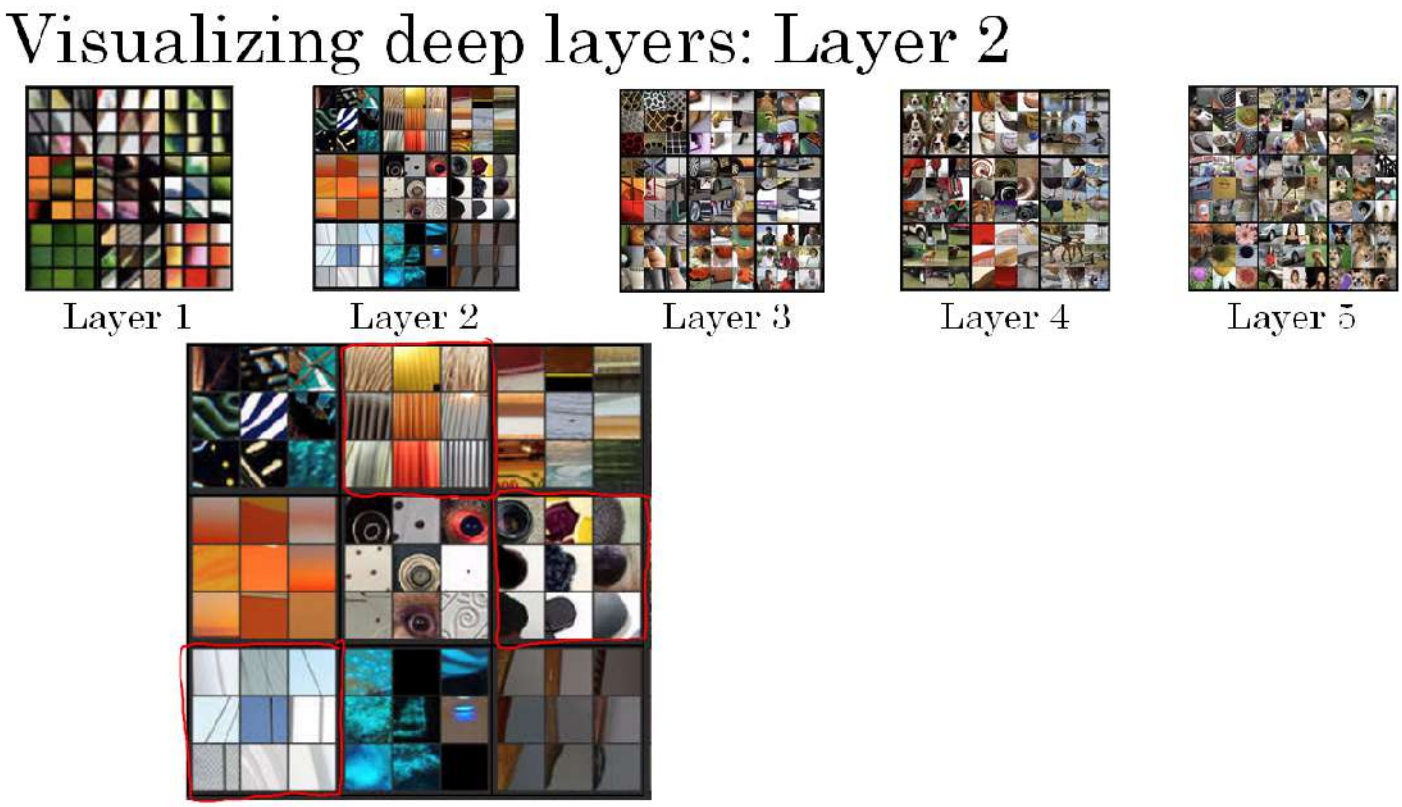
[Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks]

Let's start with a hidden unit in layer 1 and suppose you scan through your training sets and find out what are the images or what are the image patches that maximize that unit's activation. So in other words pause your training set through your neural network, and figure out what is the image that maximizes that particular unit's activation. Now, notice that a hidden unit in layer 1, will see only a relatively small portion of the neural network and so if you visualize, if you plot what activated unit's activation, it makes makes sense to plot just a small image patches, because all of the image that that particular unit sees. So if you pick one hidden unit and find the nine input images that maximizes that unit's activation, you might find nine image patches like shown in diagram below.

Visualizing deep layers: Layer 1

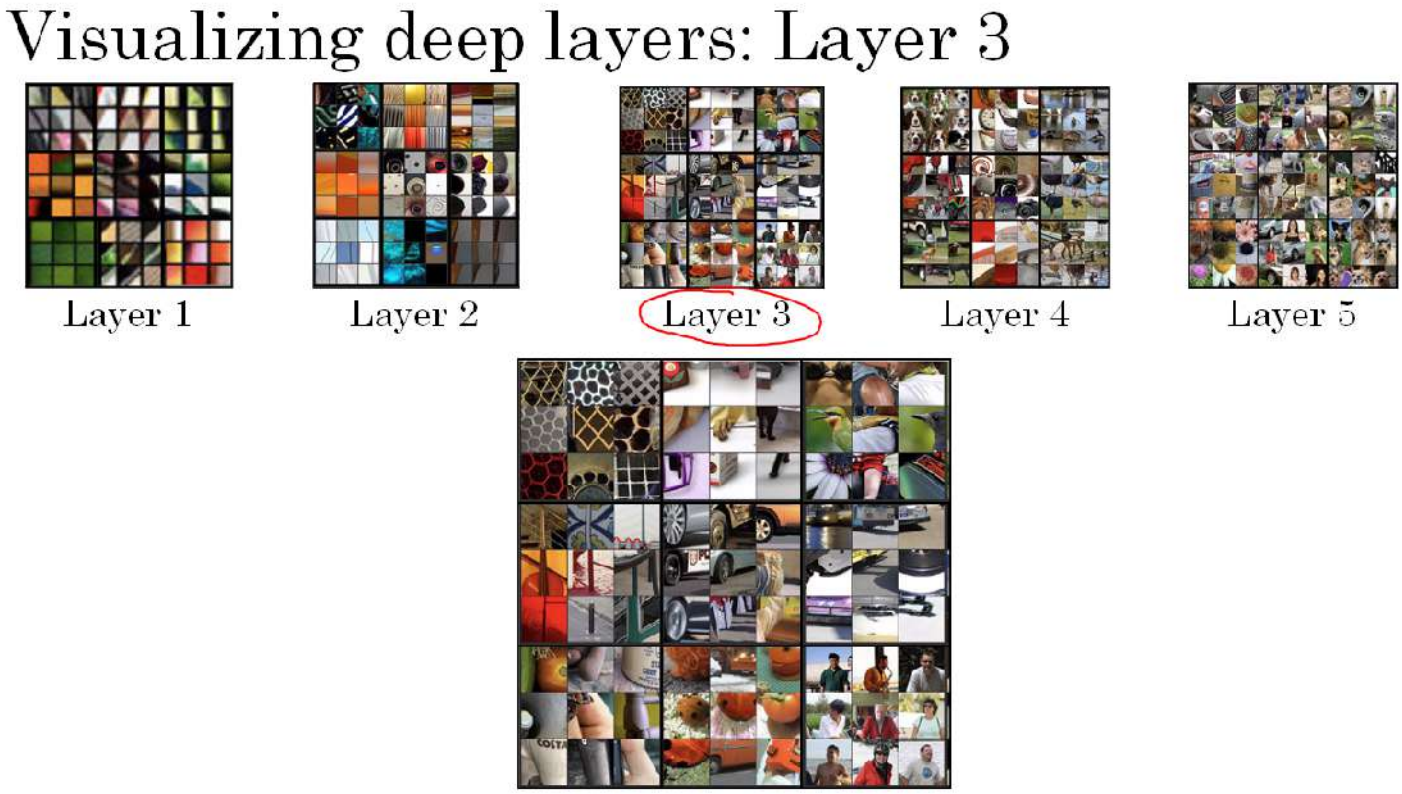


So looks like that in the lower region of an image that this particular hidden unit sees, it's looking for an egde or a line that looks like that. So those are the nine image patches that maximally activate one hidden unit's activation. Now, you can then pick a different hidden unit in layer 1 and do the same thing. So that's a different hidden unit, and looks like this second one, represented by these 9 image patches. Looks like this hidden unit is looking for a line sort of in that portion of its input region, we'll also call this receptive field. And if you do this for other hidden units, you'll find other hidden units, tend to activate in image patches that look like that. This one seems to have a preference for a vertical light edge, but with a preference that the left side of it be green. This one really prefers orange colors, and this is an interesting image patch. This red and green together will make a brownish or a brownish-orangish color, but the neuron is still happy to activate with that, and so on. So this is nine different representative neurons and for each of them the nine image patches that they maximally activate on. So this gives you a sense that, units, train hidden units in layer 1, they're often looking for **relatively simple features such as edge or a particular shade of color** and all of the examples we're using in this section come from this paper by Mathew Zeiler and Rob Fergus, titled **visualizing and understanding convolutional networks**. and we're just going to use one of the simpler ways to visualize what a hidden unit in a neural network is computing. If you read their paper, they have some other more sophisticated ways of visualizing when the ConvNet is running as well.
But now you have repeated this procedure several times for nine hidden units in layer 1. What if you do this for some of the hidden units in the deeper layers of the neuron network. And what does the neural network then learning at a deeper layers. So in the deeper layers, a hidden unit will see a larger region of the image. Where at the extreme end each pixel could hypothetically affect the output of these later layers of the neural network. So l**ater units are actually seen larger image patches**, I'm still going to plot the image patches as the same size on these slides. But if we repeat this procedure, this is what you had previously for layer 1, and this is a visualization of what maximally activates nine different hidden units in layer 2.

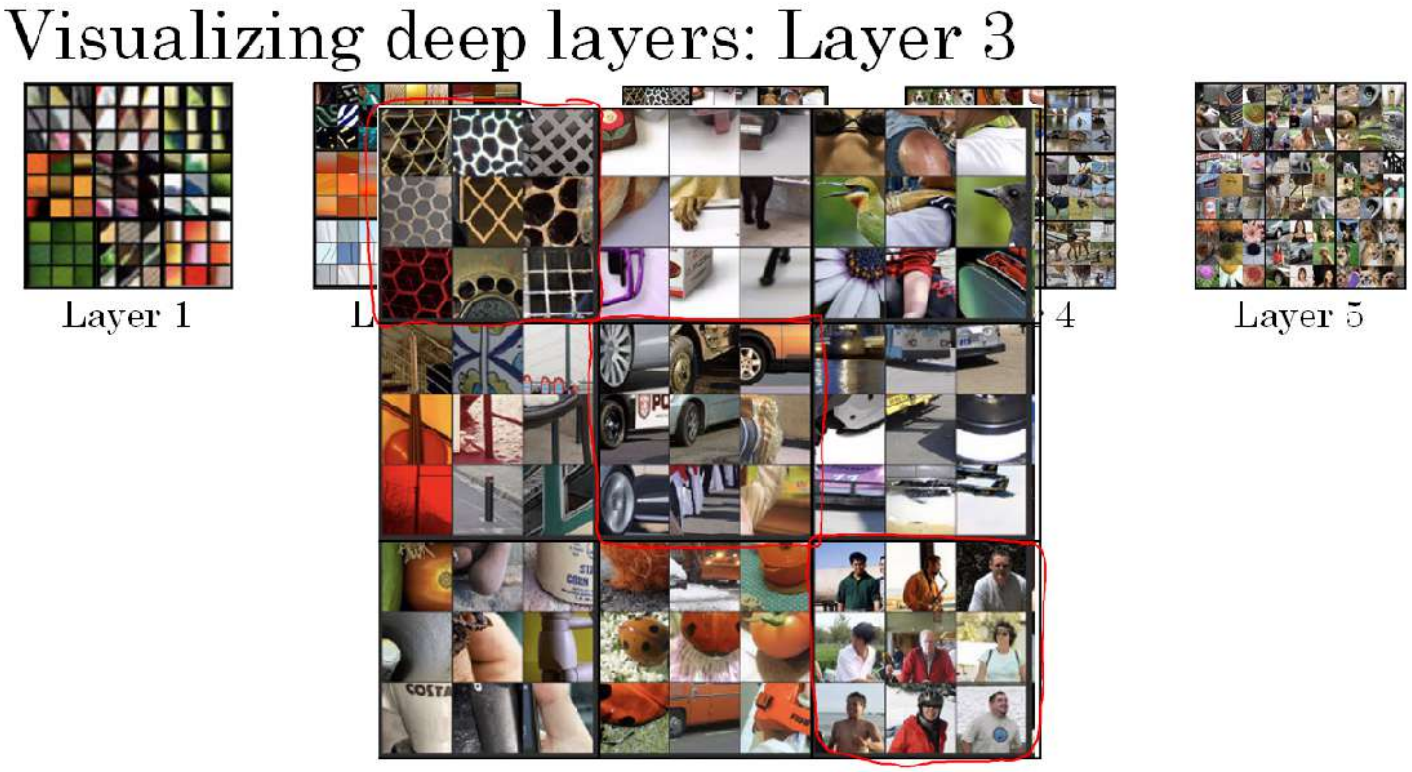


So I want to be clear about what this visualization is. These are the nine patches that cause one hidden unit to be highly activated. And then each grouping, this is a different set of nine image patches that cause one hidden unit to be activated. So this visualization shows nine hidden units in layer 2, and for each of them shows nine image patches that causes that hidden unit to have a very large output, a very large activation. And you can repeat these for deeper layers as well.

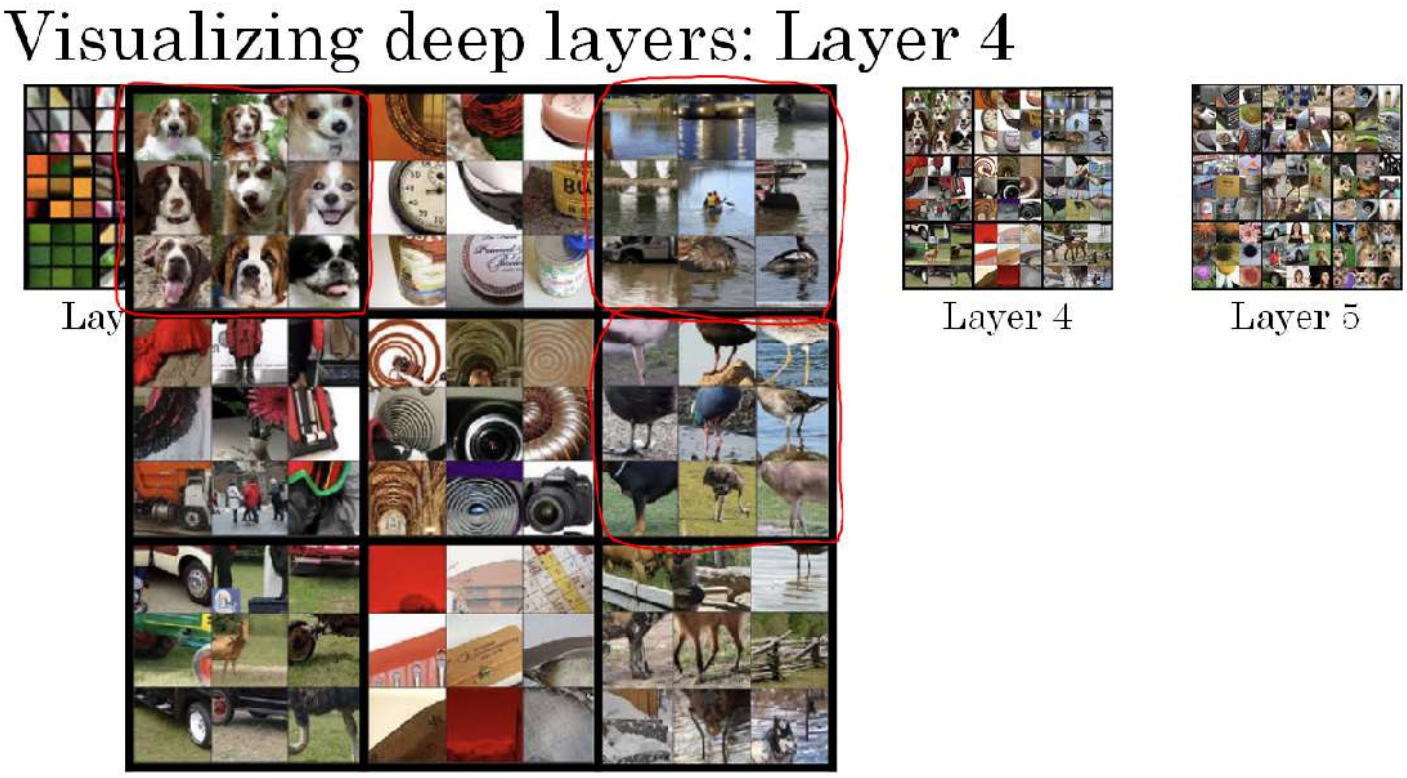
Now, on this section, I know it's kind of hard to see these tiny little image patches, so let me zoom in for some of them. For layer 1, this is what you saw. So for example, this is that first unit we saw which was highly activated, if in the region of the input image, you can see there's an edge maybe at that angle. Now let's zoom in for layer 2 as well, to that visualization. So this is interesting, layer 2 looks it's detecting more complex shapes and patterns. So for example, this hidden unit looks like it's looking for a vertical texture with lots of vertical lines. This hidden unit looks like its highly activated when there's a rounder shape to the left part of the image. Here's one that is looking for very thin vertical lines and so on. And so the features the second layer is detecting are getting more complicated. How about layer 3?



Let's zoom into that, in fact let me zoom in even bigger, so you can see this better, these are the things that maximally activate layer 3. But let's zoom in even bigger, and so this is pretty interesting again. It looks like there is a hidden unit that seems to respond highly to a rounder shape in the lower left hand portion of the image, maybe. So that ends up detecting a lot of cars, dogs and wonders is even starting to detect people.

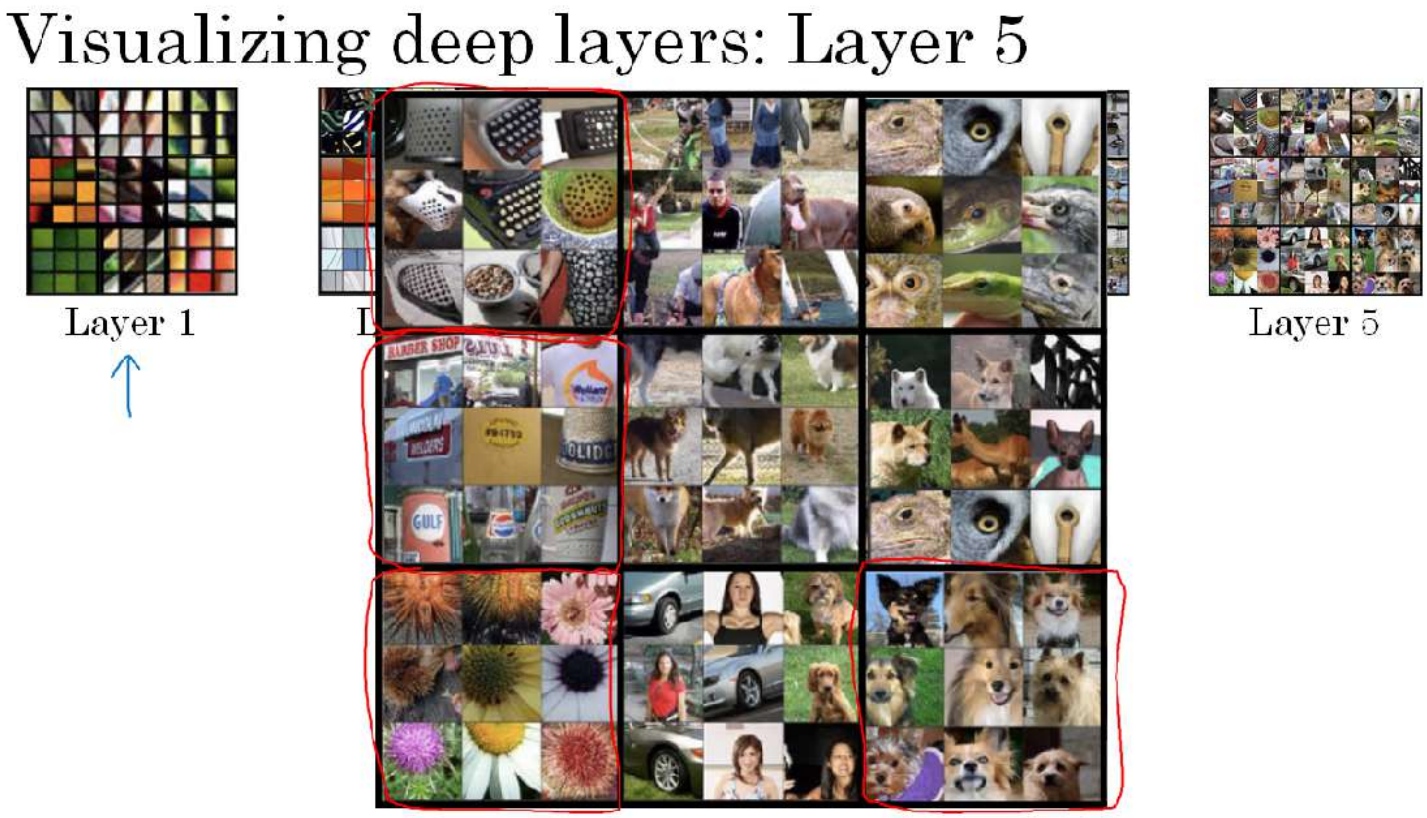


And this one look like it is detecting certain textures like honeycomb shapes, or square shapes, this irregular texture. And some of these it's difficult to look at and manually figure out what is it detecting, but it is clearly starting to detect more complex patterns. How about the next layer? Well, here is layer 4,



and you'll see that the features or the patterns is detecting or even more complex. It looks like this has learned almost a dog detector, but all these dogs likewise similar, right? Is this, I don't know what dog species or dog breed this is. But now all those are dogs, but they look relatively similar as dogs go. Looks like this hidden unit and therefore it is detecting water. This looks like it

is actually detecting the legs of a bird and so on. And then layer 5 is detecting even more sophisticated things.



Cost function

To build a Neural Style Transfer system, let's define a cost function for the generated image. What you see later is that by minimizing this cost function, you can generate the image that you want. Remember what the problem formulation is. You're given a content image C, given a style image S and you goal is to generate a new image G. In order to implement neural style transfer, what you're going to do is define a cost function J of G that measures how good is a particular generated image and we'll use gradient to descent to minimize J of G in order to generate this image. How good is a particular image?

Neural style transfer cost function

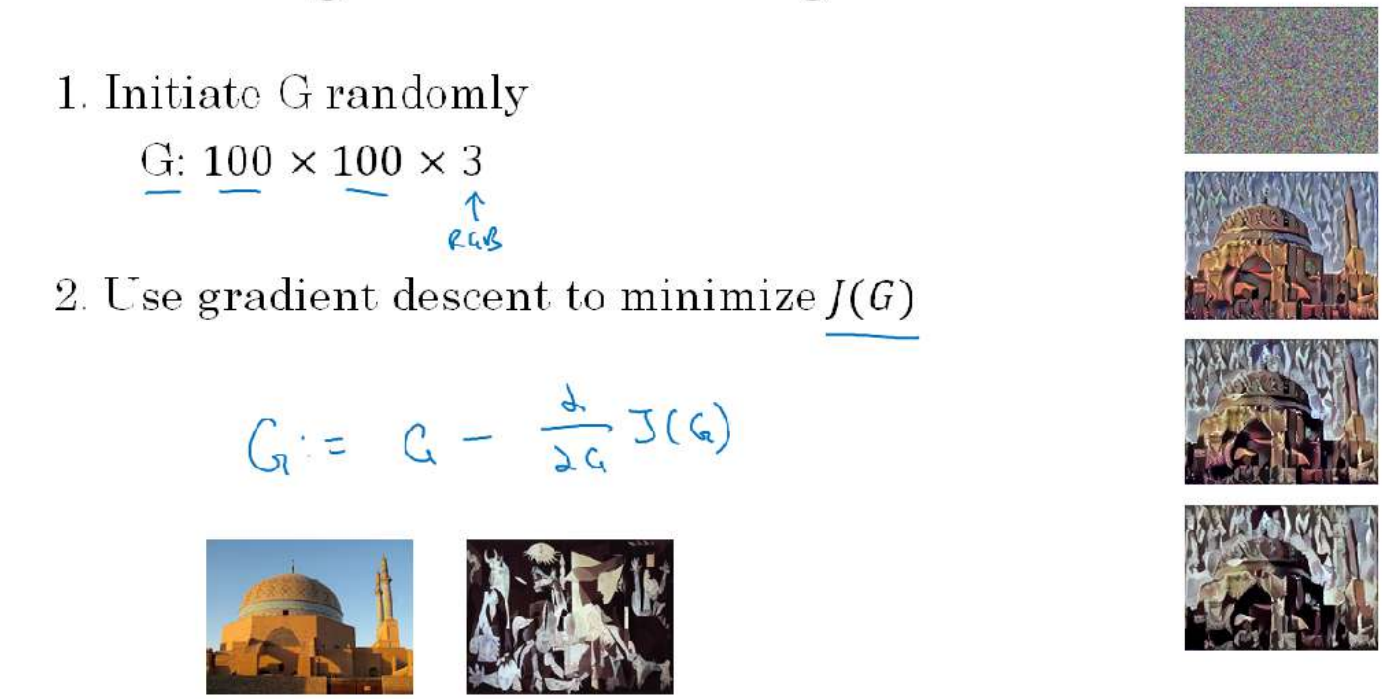
Content C Style S

Generated image G

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

[Gatys et al., 2015. A neural algorithm of artistic style. Images on slide generated by Justin Johnson]

Well, we're going to define two parts to this cost function. The first part is called the content cost. This is a function of the content image and of the generated image and what it does is it measures how similar is the contents of the generated image to the content of the content image C. And then going to add that to a style cost function which is now a function of S,G and what this does is it measures how similar is the style of the image G to the style of the image S. Finally, we'll weight these with two hyper parameters alpha and beta to specify the relative weighting between the content costs and the style cost. It seems redundant to use two different hyper parameters to specify the relative cost of the weighting. One hyper parameter seems like it would be enough but the original authors of the Neural Style Transfer Algorithm, use two different hyper parameters. I'm just going to follow their convention here. The Neural Style Transfer Algorithm is going to present in the next few sections is due to Leon Gatys, Alexander Ecker and Matthias. Their papers is not too hard to read so after watching these few sections if you wish, I certainly encourage you to take a look at their paper as well if you want. The way the algorithm would run is as follows, having to find the cost function J(G) in order to actually generate a new image what you do is the following. You would initialize the generated image G randomly so it might be 100 by 100 by 3 or 500 by 500 by 3 or whatever dimension you want it to be. Then we'll define the cost function J(G) on the previous section. What you can do is use gradient descent to minimize this so you can update G as G minus the derivative respect to the cost function of J(G). In this process, you're actually updating the pixel values of this image G which is a 100 by 100 by 3 maybe RGB channel image. Here's an example, let's say you start with this content image and this style image.



This is a another probably Picasso image. Then when you initialize G randomly, you're initial randomly generated image is just this white noise image with each pixel value chosen at random. As you run gradient descent, you minimize the cost function J(G) slowly through the pixel value so then you get slowly an image that looks more and more like your content image rendered in the style of your style image. In this section, we saw the overall outline of the Neural Style Transfer Algorithm where you define a cost function for the generated image G and minimize it.

Content Cost Function

The cost function of the neural style transfer algorithm had a content cost component and a style cost component. Let's start by defining the content cost component. Remember that this is the overall cost function of the neural style transfer algorithm. So, let's figure out what should the content cost function be. Let's say that you use hidden layer l to compute the content cost. If l is a very small number, if you use hidden layer one, then it will really force your generated image to pixel values very similar to your content image. Whereas, if you use a very deep layer, then it's just asking, "Well, if there is a dog in your content image, then make sure there is a dog somewhere in your generated image. " So in practice, layer l chosen somewhere in between. It's neither too shallow nor too deep in the neural network. Check below diagram for summary.

Content cost function

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

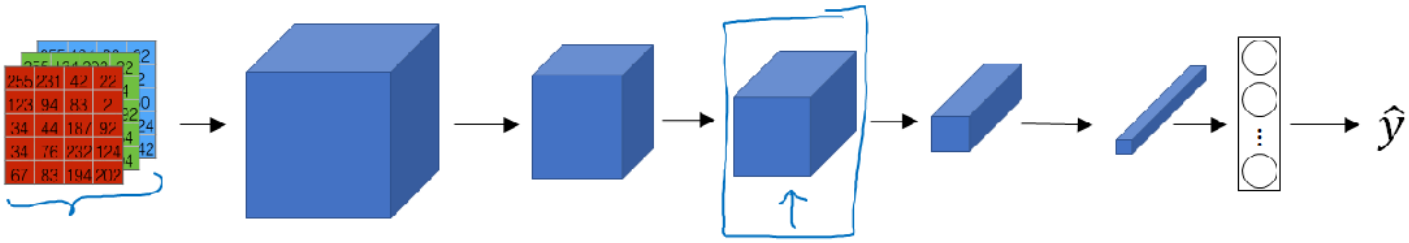
$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

[Gatys et al., 2015. A neural algorithm of artistic style]

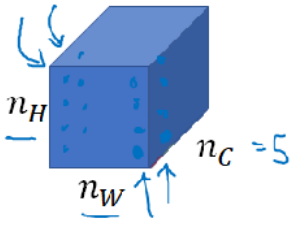
Style Cost Function

In the last section, we saw how to define the content cost function for the neural style transfer. Next, let's take a look at the **style cost function**. So, what is the style of an image mean? Please check the diagram below to understand the style cost function:

Meaning of the "style" of an image



Say you are using layer l's activation to measure "style."
Define style as correlation between activations across channels.

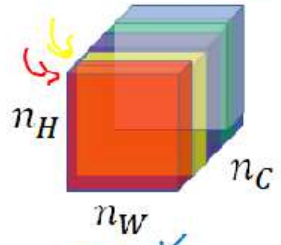


How correlated are the activations across different channels?

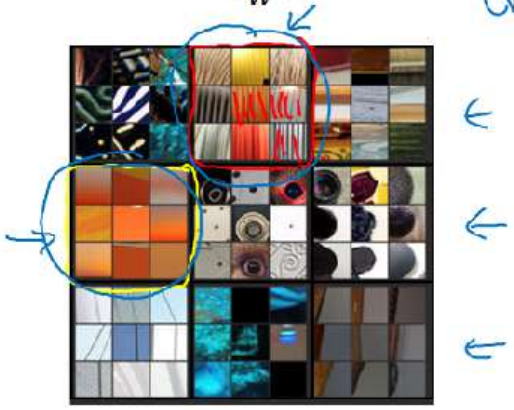
[Gatys et al., 2015. A neural algorithm of artistic style]

Intuition about style of an image

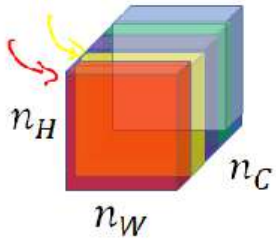
Style image



Correlated?
Uncorrelated



Generated Image



[Gatys et al., 2015. A neural algorithm of artistic style]

Style matrix

H W C

Let $a^{[l]}_{i,j,k}$ = activation at (i,j,k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

n_c

$G^{[l]}_{kk'}$

$k = 1, \ldots, n_c^{[l]}$

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk}$$

"Gram matrix"

$$J^{[l]}_{style}(S,G) = \frac{1}{(\cdots)} \left\| G^{[l](S)} - G^{[l](G)} \right\|_F^2$$

$$= \frac{1}{(2 n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2$$

β

[Gatys et al., 2015. A neural algorithm of artistic style]

Style cost function

$$J^{[l]}_{style}(S,G) = \frac{1}{\left(2n_H^{[l]}n_W^{[l]}n_C^{[l]}\right)^2}\sum_k\sum_{k'}(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})$$

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S,G)$$

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

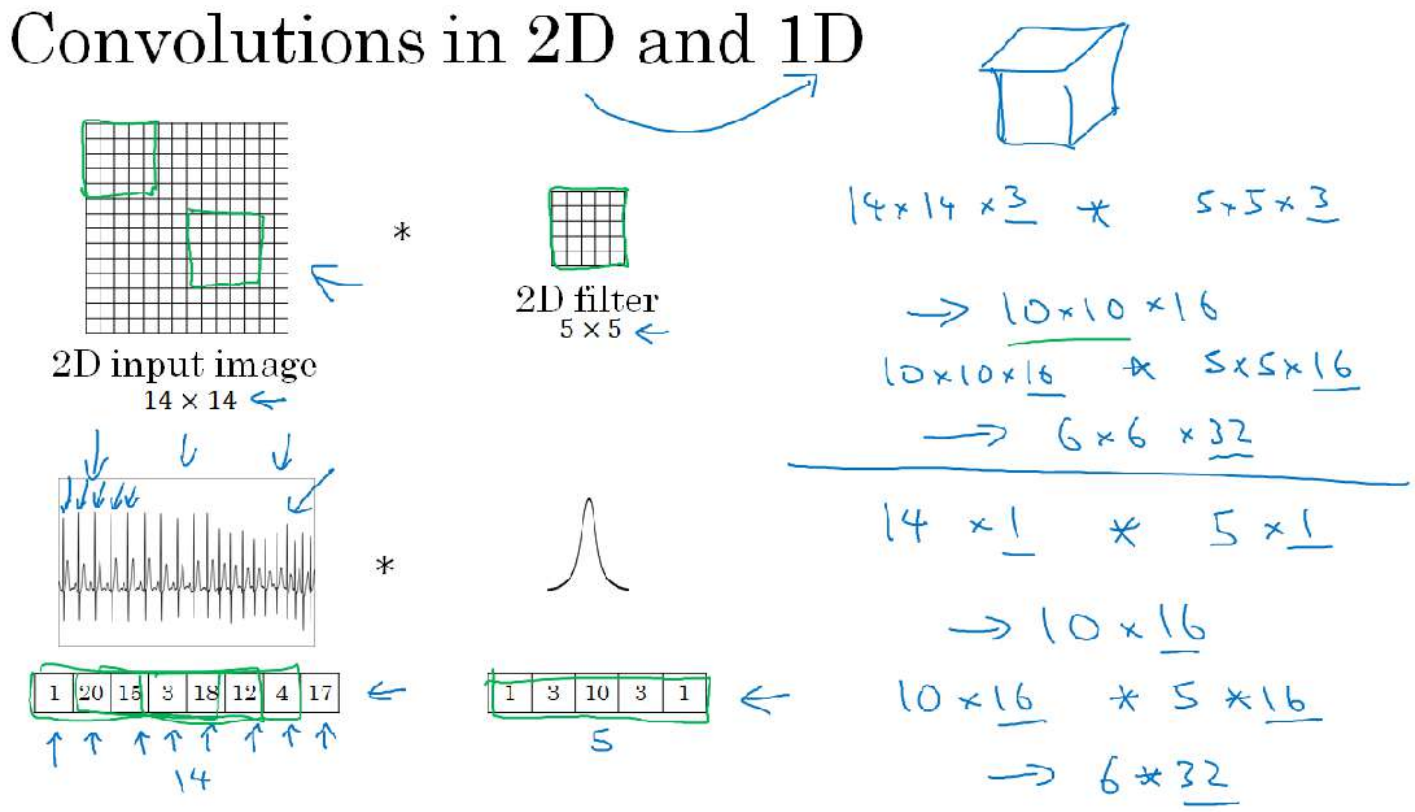
[Gatys et al., 2015. A neural algorithm of artistic style]

So, the overall style cost function, you can define as sum over all the different layers of the style cost function for that layer. We should define the book weighted by some set of parameters, by some set of additional hyperparameters, which we'll denote as lambda l here. So what it does is allows you to use different layers in a neural network. Well of the early ones, which measure relatively simpler low level features like edges as well as some later layers, which measure high level features and cause a neural network to take both low level and high level correlations into account when computing style.

1D and 3D Generalizations

We've learned a lot about ConvNets, everything ranging from the architecture of the ConvNet to how to use it for image recognition, to object detection, to face recognition and neural-style transfer. And even though most of the discussion has focused on images, on sort of 2D data, because images are so pervasive.

It turns out that many of the ideas you've learned about also apply, not just to 2D images but also to 1D data as well as to 3D data. Let's take a look.



In the first week of this course, you learned about the 2D convolution, where you might input a 14 x 14 image and convolve that with a 5 x 5 filter. And you saw how 14 x 14 convolved with 5 x 5, this gives you a 10 x 10 output and if you have multiple channels, maybe those 14 x 14 x 3, then it would be 5 x 5 that matches the same 3. And then if you have multiple filters, say 16 filters, you end up with 10 x 10 x 16. It turns out that a similar idea can be applied to 1D data as well. For example, on the left is an EKG signal, also called an electrocardioagram. Basically if you place an electrode over your chest, this measures the little voltages that vary across your chest as your heart beats. Because the little electric waves generated by your heart's beating can be measured with a pair of electrodes. And so this is an EKG of someone's heart beating. And so each of these peaks corresponds to one heartbeat. So if you want to use EKG signals to make medical diagnoses, for example, then you would have 1D data because what EKG data is, is it's a time series showing the voltage at each instant in time. So rather than a 14 x 14 dimensional input, maybe you just have a 14 dimensional input. And in that case, you might want to convolve this with a 1 dimensional filter. So rather than the 5 by 5, you just have 5 dimensional filter. So with 2D data what a convolution will allow you to do was to take the same 5 x 5 feature detector and apply it across at different positions throughout the image. And that's how you wound up with your 10 x 10 output. What a 1D filter allows you to do is take your 5 dimensional filter and similarly apply that in lots of different positions throughout this 1D signal. And so if you apply this convolution, what you find is that a 14 dimensional thing convolved with this 5 dimensional thing, this would give you a 10 dimensional output. And again, if you have multiple channels, you might have in this case you can use just 1 channel, if you have 1 lead or 1 electrode for EKG, so times 5 x 1. And if you have 16 filters, maybe end up with 10 x 16 over there, and this could be one layer of your ConvNet. And then for the next layer of your ConvNet, if you input a 10 x 16 dimensional input and you might convolve that with a 5 dimensional filter again. Then these have 16 channels, so that has a match. And we have 32 filters, then the output of another layer would be 6 x 32, if you have 32 filters, right? And the analogy to the the 2D data, this is similar to all of the 10 x 10 x 16 data and convolve it with a 5 x 5 x 16, and that has to match. That will give you a 6 by 6 dimensional output, and you have 32 filters, that's where the 32 comes from. So all of these ideas apply also to 1D data, where you can have the same feature detector, such as this, apply to a variety of positions. For example, to detect the different heartbeats in an EKG signal. But to use the same set of features to detect the heartbeats even at different positions along these time

series, and so ConvNet can be used even on 1D data. For along with 1D data applications, you actually use a recurrent neural network, which you learn about in the next course. But some people can also try using ConvNets in these problems. And in the next course on sequence models, which we will talk about recurring neural networks and LCM and other models like that. We'll talk about the pros and cons of using 1D ConvNets versus some of those other models that are explicitly designed to sequenced data. So that's the generalization from 2D to 1D.
How about 3D data? Instead of having a 1D list of numbers or a 2D matrix of numbers, you now have a 3D block, a three dimensional input volume of numbers. CAT scans, medical scans as one example of 3D volumes. But another example of data, you could treat as a 3D volume would be movie data, where the different slices could be different slices in time through a movie and you could use this to detect motion or people taking actions in movies. So that's it on generalization of ConvNets from 2D data to also 1D as well as 3D data. Image data is so pervasive that the vast majority of ConvNets are on 2D data, on image data, but I hope that these other models will be helpful to you as well.

**********END OF COURSE**********