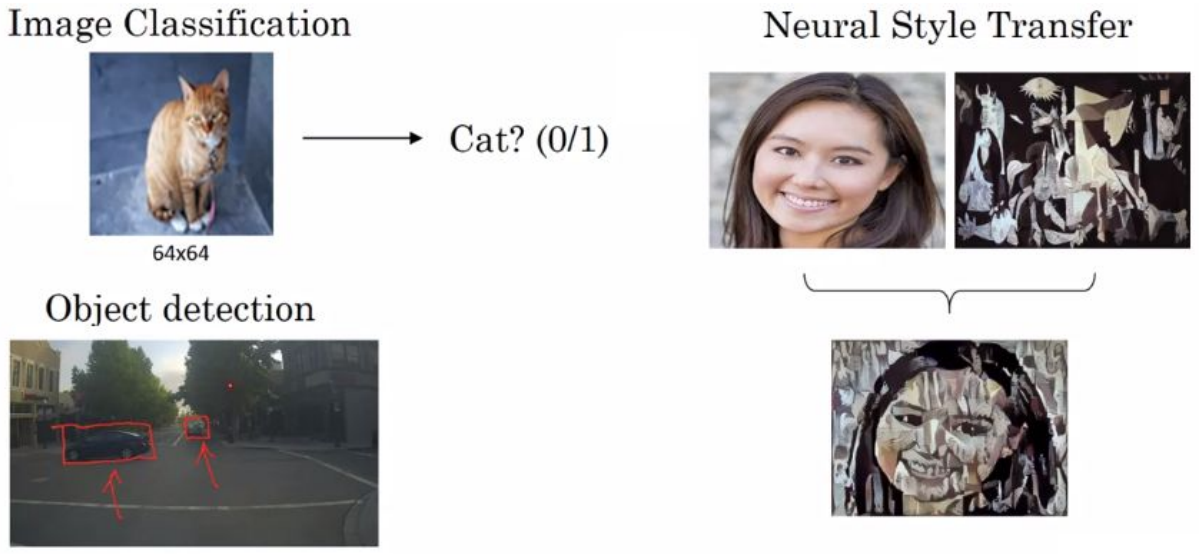
# COURSE 4: CONVOLUTIONAL NEURAL NETWORKS

## Convolutional Neural Networks

### Computer Vision

Computer vision is an interdisciplinary scientific field that deals with how computers can gain high-level understanding from digital images or videos. Computer vision has lots of applications, such as image classification, object detection, neural style transfer (transforming a random image into an image with Van Gogh's style, for example) and so on.

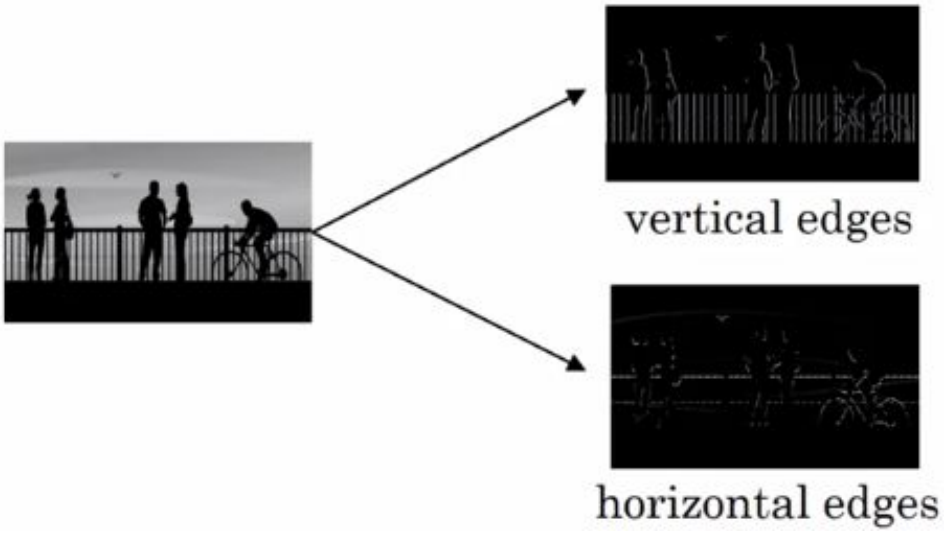



One of the challenges of computer vision is that the inputs can get really big. A single image with 1000x1000 pixels (1MP) will become a vector with 3 million dimensions (1000 x 1000 x 3 (RGB)). If you use a single image with 1MP, a neural network with 1000 hidden units in the first hidden layer and a fully connected network you will end up having a (1000, 3M) dimension matrix as the weight parameter matrix. That is a 3 billion elements matrix, making the computation very costly.

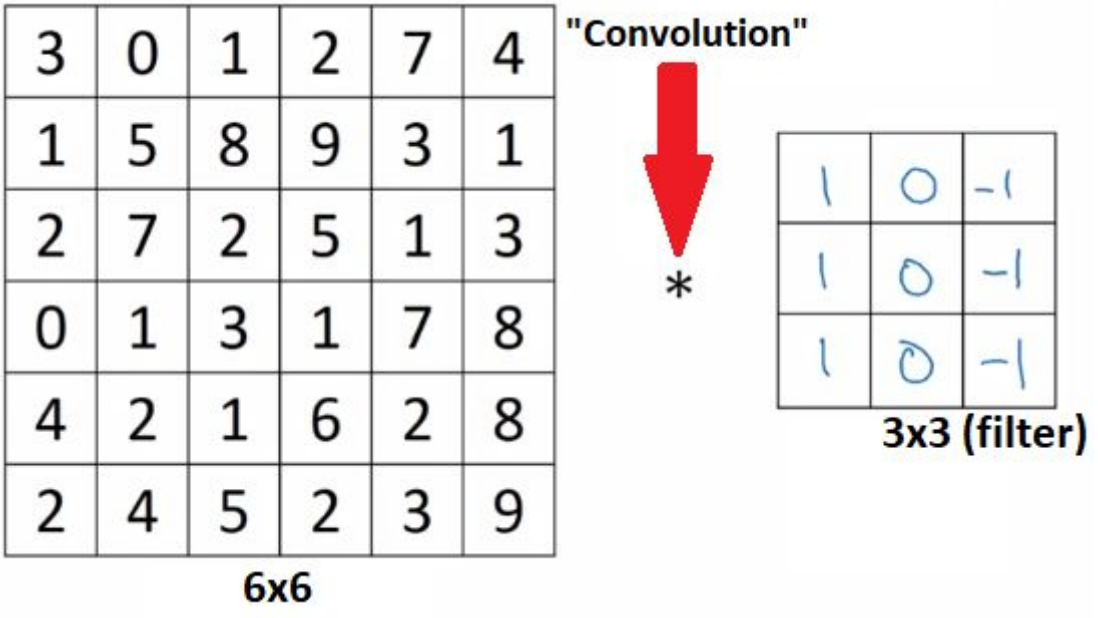
### Edge Detection Example

The convolutional operation is one of the fundamental building blocks of a convolutional neural network. Using edge detection as a motivating example, you'll see how the convolutional operation works. As we saw in the previous courses, one of the techniques to make image recognition is to detect edges in the earlier layers, then detect parts of objects, and in the later layers detect parts of complete objects.

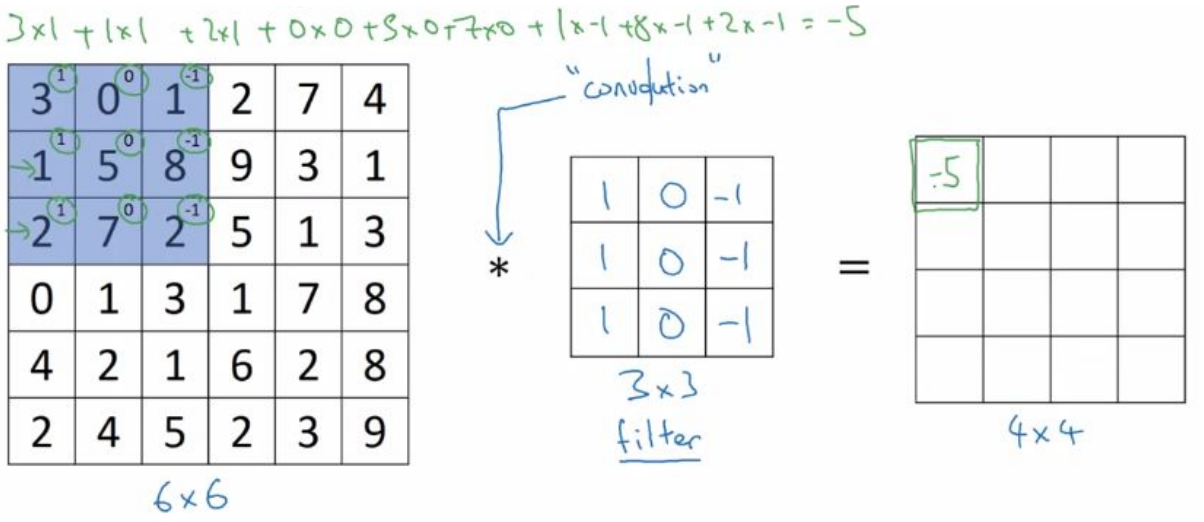
You can use the convolutional operation to detect vertical and horizontal edges in a image, as you can see in the example below:



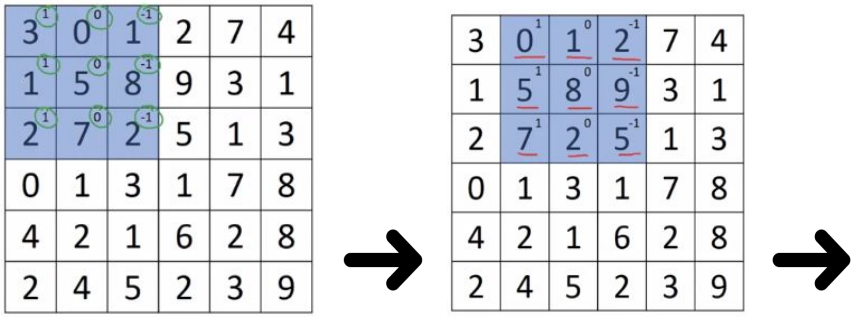

Let's study how vertical edges detection works. We'll use a 6x6 matrix to represent an image as an example. First of all, we need to define the filter matrix (or sometimes also called kernel or feature detector). The filter matrix is the element that is going to perform the convolutional operation through the image matrix, the filter matrix changes depending on the problem you are dealing with. The 6x6 matrix that represents the image and the filter matrix can be seen below:

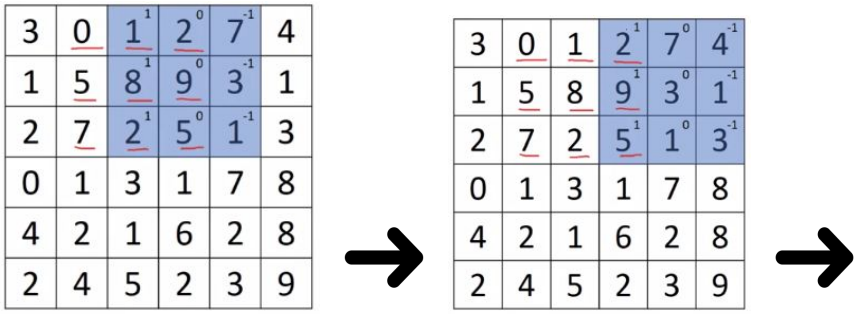


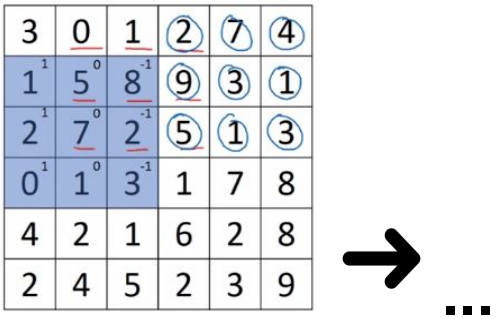

The " * " symbol represents the convolution operator. In this case, the output matrix of this operation is going to be a 4x4 matrix (that can be an image as well). You can see how the convolutional operation works in the image below:



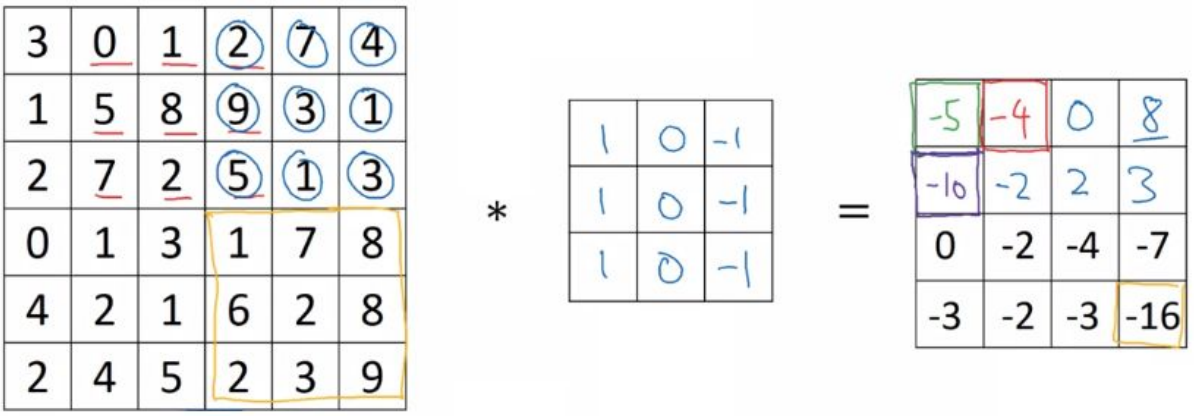

The filter is kind of “projected” in the original matrix, and an element wise multiplication is performed. All the elements of the resultant matrix are added together, and this result is going to be the first element of the output matrix (in this case, it has a value of -5).



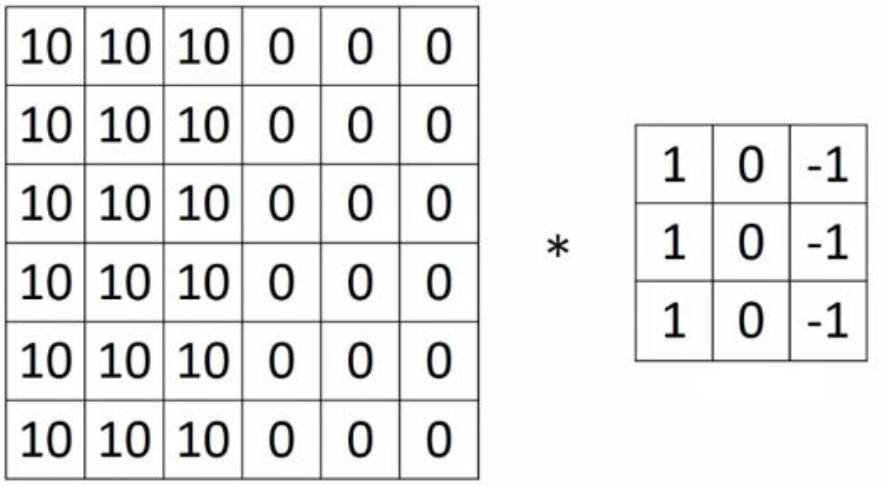




The output matrix is going to be:




Different programming tools and languages will have different notations for making this operation. In Python, the usual function name we use to perform this operation is conv_forward. If you are using Tensorflow the command is tf.nn.conv2d, and if you are using Keras the command is Conv2D. Let's see another example:





In this case, we have another 6x6 image and a 3x3 filter. The difference between the values in each section of the matrices is used to represent vertical edges. The numbers in the filter shown above are wisely chosen to perform the vertical edges detection. The output of the convolutional operation is:

| | | | | | |
|---|---|---|---|---|---|
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |

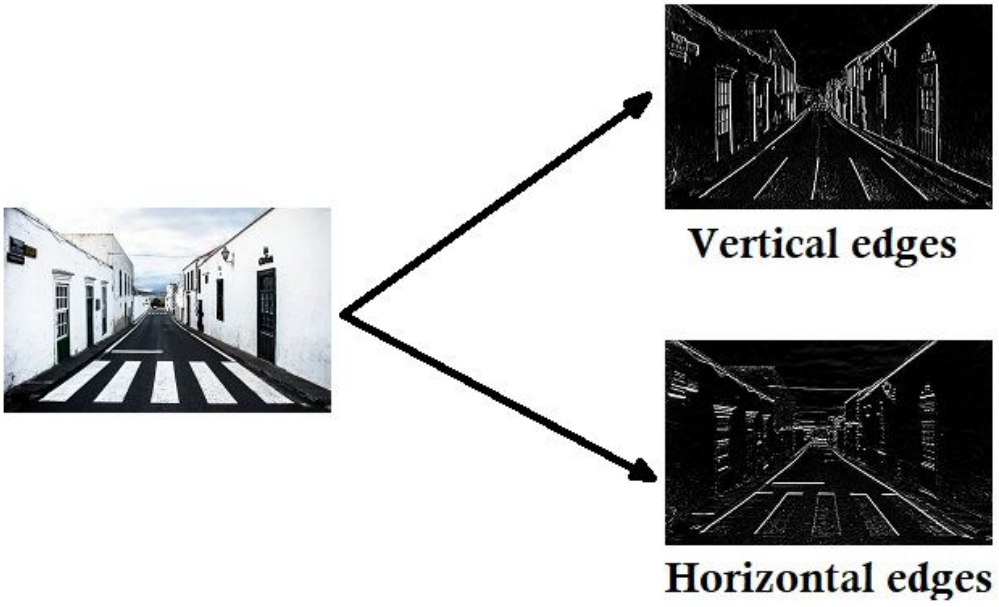
*

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

=

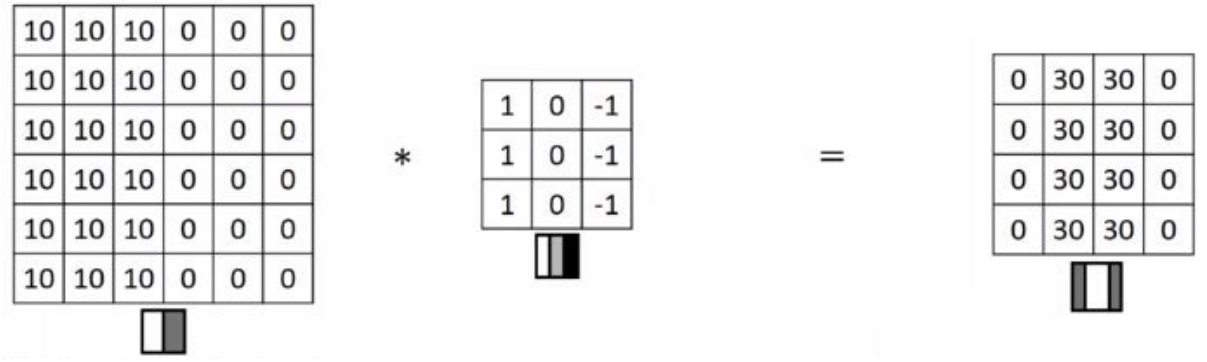
| | | | |
|---|---|---|---|
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |
| 0 | 30 | 30 | 0 |

As you can see, the output matrix has a brighter region (the white region in the middle region of the matrix, represented with the 30's), and that corresponds to finding a vertical edge in the picture.
Of course this is a very simple example, with only 6x6 pixels, but if this operation is applied to a 1000x1000 image, the results are very interesting. You can see an example of a complex image below:

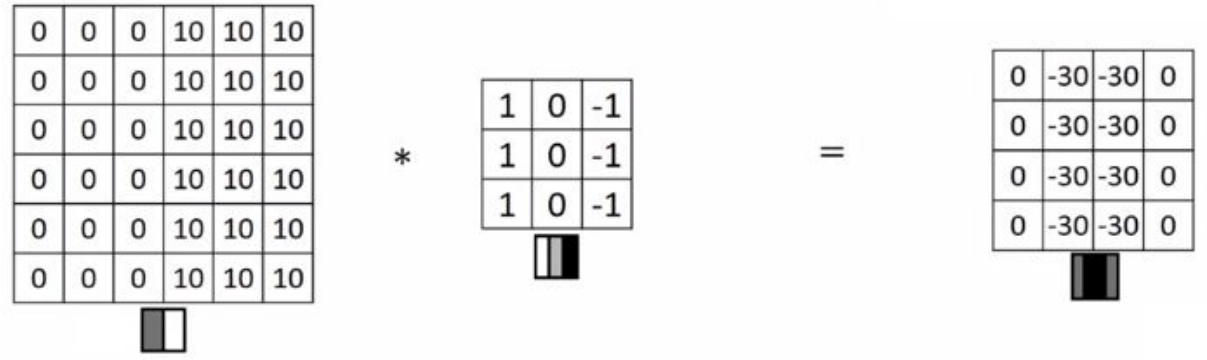



## More Edge Detection

You saw how the convolutional operation works in the previous section. Let's study now the difference between positive and negative edges (that is, the difference between light to dark versus dark to light edge transitions). Let's use the same example seen in the previous section:




As you can see, the output image is a light to dark vertical transition. But what happens if we flip all the values of the matrix? (substitute 10 with 0, and 0 with 10)




As expected, the output is going to have a dark to light vertical transition and all the numbers of the matrix will have an inverted signal. To do a horizontal edge detector, we just need to change the filter matrix, as follows:

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| -1 | -1 | -1 |

Horizontal

Let's see an example using the horizontal edge detector:

| | | | | | |
|---|---|---|---|---|---|
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 10 | 10 | 10 | 0 | 0 | 0 |
| 0 | 0 | 0 | 10 | 10 | 10 |
| 0 | 0 | 0 | 10 | 10 | 10 |
| 0 | 0 | 0 | 10 | 10 | 10 |

*

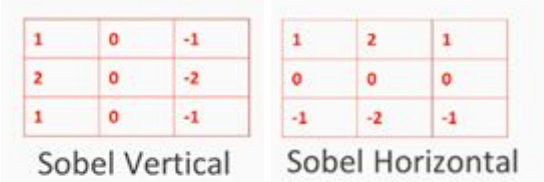
| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| -1 | -1 | -1 |

=

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 30 | 10 | -10 | -30 |
| 30 | 10 | -10 | -30 |
| 0 | 0 | 0 | 0 |

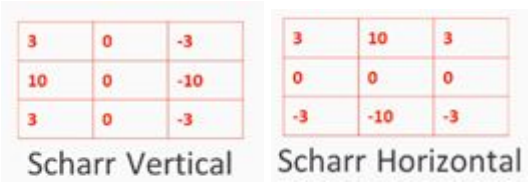
There are different kinds of filters to recognize different edges in images. Historically, in the computer vision literature, there was a fair amount of debate about what is the best set of numbers to use in the filters. You can see some different filters bellow:

Sobel filter:




Advantage: this filter puts more weight to the central row (in the horizontal filter) and the central column (in the vertical filter).

A variation of the Sobel filter is the Scharr filter, that has slightly different properties.




With the rise of deep learning, one of the things you learned is that when you really want to detect edges in some complicated image, maybe you don't need to have computer vision researchers to find these nine numbers. You can just learn them and treat the nine numbers of the matrix as parameters, which you can learn using back propagation. The goal then is to learn nine parameters so that when you take the image and convolve it with your filter, that gives you a good edge detector.

| | | |
|---|---|---|
| $w_1$ | $w_2$ | $w_3$ |
| $w_4$ | $w_5$ | $w_6$ |
| $w_7$ | $w_8$ | $w_9$ |

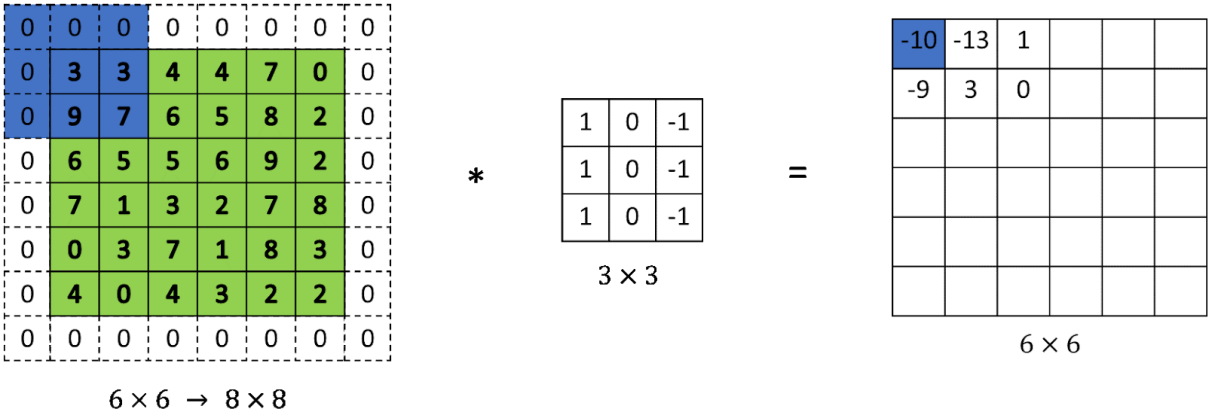
This training process can lead you to develop a filter that detects edges that are at 45º, or 70º, or whatever orientation it chooses.

## Padding

In order to build deep neural networks, one modification to the basic convolutional operation that you need to use is padding. We saw in the previous section that if you take a 6x6 image and convolve it with a 3x3 filter, you'll end up with a 4x4 output. The dimension of the output, considering a nxn image and a fxf filter, is going to be:
$(n-f+1)\,x\,(n-f+1)$

As expected, if we have a 6x6 image and a 3x3 filter, n = 6 and f = 3, then the output will have a (4,4) dimension. One downside to this method is that every time you apply a convolutional operator, your image shrinks, so you come from a bigger

image to a much smaller one. You can do this process only a few times before the image starts getting really small. Another downside to this method is that the pixels in the corners of the image are less used and never centered by the filter, whereas if you take a pixel in the middle of the image there are a lot of regions that overlap so that pixel. To solve these problems you can pad the image, adding a new border with 0 values. In this case if you use a padding of one pixel, when you have a 6x6 image, you'll end up with a 8x8 image, and after the convolutional operation the output is also going to be 6x6, solving the problems of matrix shrinkage and loss of information in the edges.




If p is the number of pixels added in the border, the new dimension of the output is going to be:

$(n + 2p - f + 1)\, x\, (n + 2p - f + 1)$

The methods you've seen above are called "Valid" and "Same" convolutions. Valid convolution refers to the traditional convolutional method, without padding (the output dimension is going to be $(n - f + 1)\, x\, (n - f + 1)$). Same convolution refers to adding a pad so that the output size is the same as the input size (the output dimension is going to be $(n + 2p - f + 1)\, x\, (n + 2p - f + 1)$). In the same convolution, you can find the value of p with respect to f:

$n + 2p - f + 1 = n$ *#you want the output size to be equals to the input size*

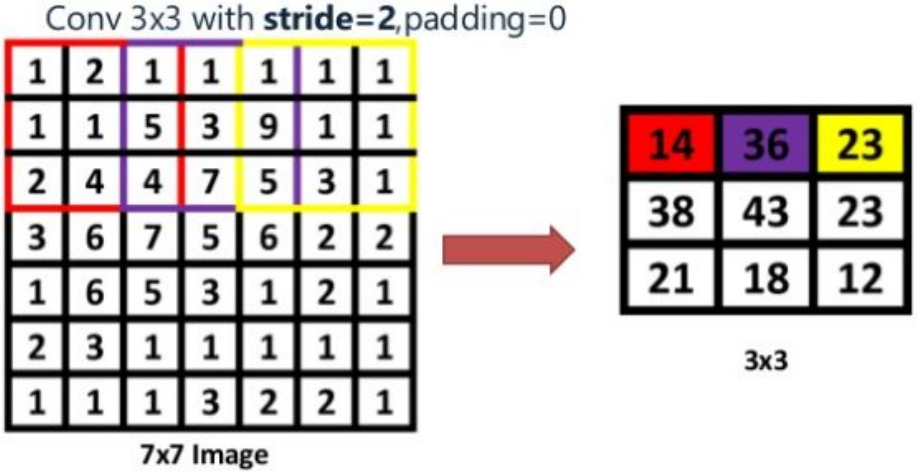
$2p = f - 1$

$p = (f - 1) / 2$

If f = 3, p = 1; if f = 5, p = 2 and so on… . Also by convention, in computer vision f is usually odd. If you use an even f value, your padding will probably be asymmetrical and you'll not have a central pixel in the filter, that's why an odd f number is prefered.

## Strided Convolutions

Strided convolutions is another piece of the basic building block of convolutions used in Convolutional Neural networks. Stride is the distance between spatial locations where the convolution filter is applied. In the default scenario, the distance is 1 in each dimension. If we use a stride-2 convolution, the filter is going to be applied in the following way:




(application of a stride-2 convolution, the input image is on the left and the output is on the right)

As you can see, the distance between one filter application and another is 2 in each direction. We convolve a 7x7 image with a 3x3 filter and get a 3x3 output. The input and output dimensions are governed by the following formula (using a n x n image, f x f filter, padding p and stride s):

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

If the fraction in this formula results in a non integer number, we need to use the floor of the division.

Technical note: cross-correlation vs. convolution
In the math textbooks, you will realize that in the convolution operation the filter is rotated before the element-wise computation. If we have the following filter:

| | | |
|---|---|---|
| 3 | 4 | 5 |
| 1 | 0 | 2 |
| -1 | 9 | 7 |

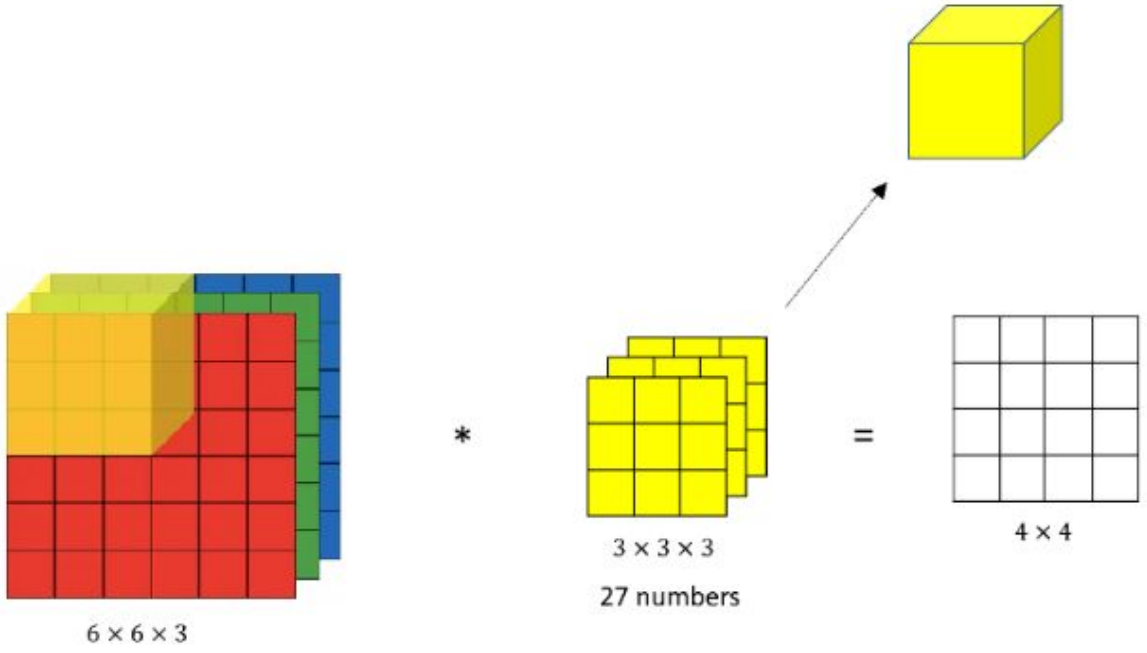
Before applying the convolutional operation with this filter, we have to flip it vertically and horizontally:

| | | |
|---|---|---|
| 7 | 9 | -1 |
| 2 | 0 | 1 |
| 5 | 4 | 3 |

In the cross-correlation, the filter is not rotated. However, the computer vision community adopted the term “convolution” to denote the cross-correlation operation, but they are different from each other.

## Convolutions Over Volume

You’ve seen how convolutions over 2D images work. Now, let’s see how you can implement convolutions over three dimensional volumes, that will allow us to convolute RGB images. A RGB image is composed of 3 matrices (three channels), one for the red color, another for the green color and the last one for the blue color. These matrices have the same height and width. The filter applied in the convolutional operation is also going to have 3 matrices (the number of channels has to be the same for the image and the filter).




Each element of the output matrix is going to be the sum of 27 element-wise multiplication numbers (3x3x3). If you want to detect vertical edges in the red channel, you can set up the filters like this:

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

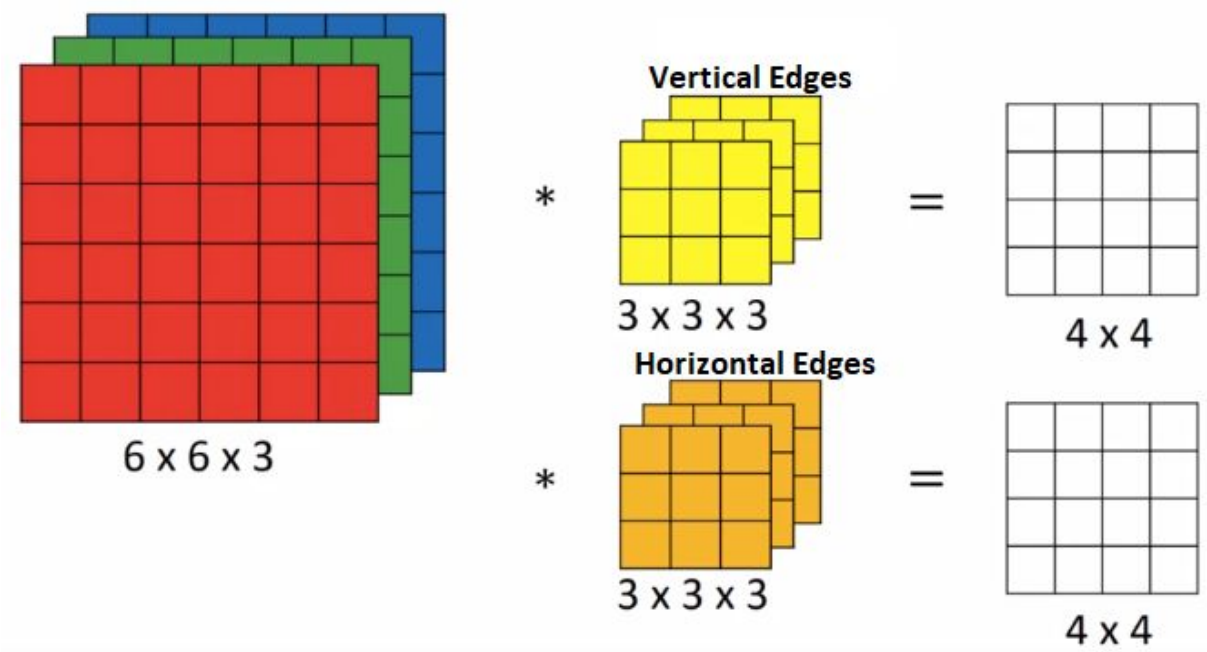
| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 1 | 0 | -1 |

| | | |
|---|---|---|
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 0 | 0 |

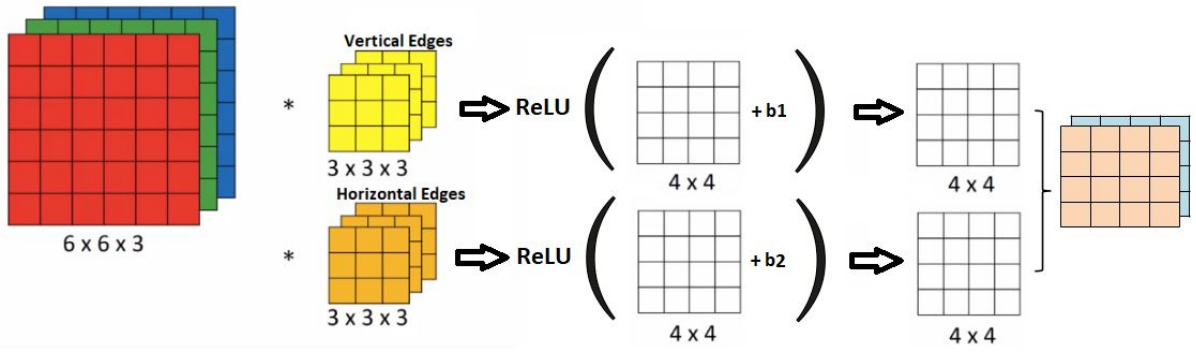
The first matrix is for the red channel, and the other two are for the green and blue channels.
If you want, you can apply multiple filters in the same input image. Imagine you want to detect vertical edges and horizontal edges. All you have to do is use 2 different filters in the image, and you'll end up with 2 output images.




You can also stack up the outputs. In this case, if you stack up the outputs, you'll end up with a 4x4x2 dimension image.

## One Layer of a Convolutional Network

Let's learn how to build an one layer convolutional neural network using the example below:

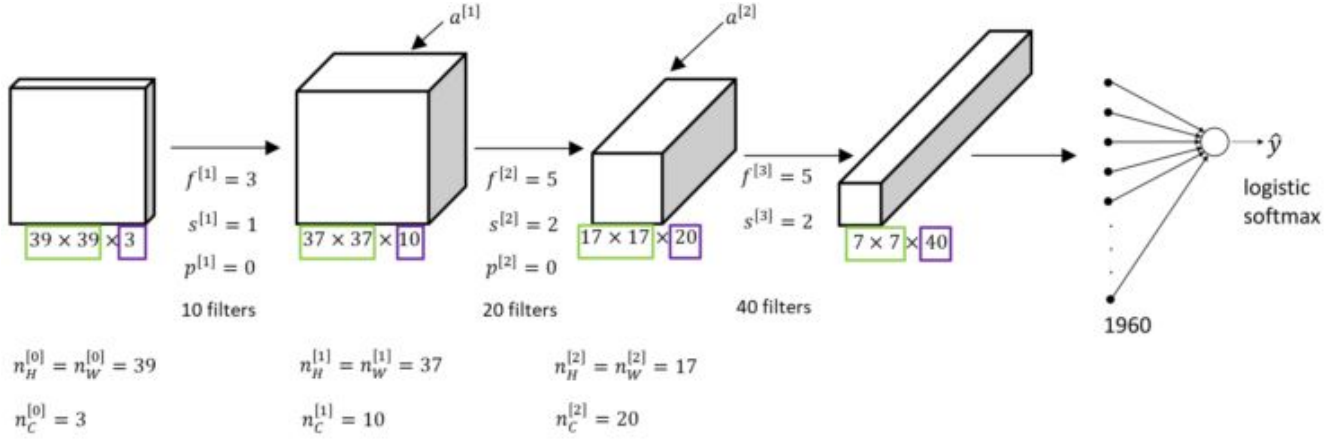



Making an analogy to the already studied neural networks, the input image works as the $a^{[0]}$. The filters work like the weights $w^{[1]}$. Then, adding the bias value allow us

to compute $z^{[1]}$. Using some activation function, such as the ReLU, we can compute the outputs of the convolutional neural network, that is the $a^{[1]}$ value. Notice that we went from a 6x6x3 image to a 4x4x2 image.

Summary of notation - If layer $l$ is a convolution layer:
$f^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
$n_c^{[l]}$ = number of filters
Each filter is: $f^{[l]} \; x \; f^{[l]} \; x \; n_c^{[l-1]}$
Activations: $a^{[l]} \rightarrow \; n_H^{[l]} \; x \; n_W^{[l]} \; x \; n_c^{[l]}$
Activations (vectorized version): $A^{[l]} \rightarrow \; m \; x \; n_H^{[l]} \; x \; n_W^{[l]} \; x \; n_c^{[l]}$
Weights: $f^{[l]} \; x \; f^{[l]} \; x \; n_c^{[l-1]} \; x \; n_c^{[l]}$
bias: $n_c^{[l]}$
Input: $n_H^{[l-1]} \; x \; n_W^{[l-1]} \; x \; n_c^{[l-1]}$
Output: $n_H^{[l]} \; x \; n_W^{[l]} \; x \; n_c^{[l]}$

Let's see an example of a simple convolutional network of recognizing a cat picture:




As you can see, we've started with a 39x39x3 image. Then we applied 10 filters with 3x3x3 dimension, with no padding and stride equals to 1. After applying these filters we end up with a 37x37x10 layer, that corresponds to $a^{[1]}$. Then we applied another filters and conditions, computed $a^{[2]}$ and finally computed $a^{[3]}$, that is a 7x7x40 dimensional layer. We unroll all the 1960 features (7x7x40=1960) of the last conv layer into units and apply a logistic regression/softmax regression to these units, to find out the output.

There are 3 common types of layers in a convolutional network:
-Convolution (Conv)
-Pooling (Pool)
-Fully connected (FC)

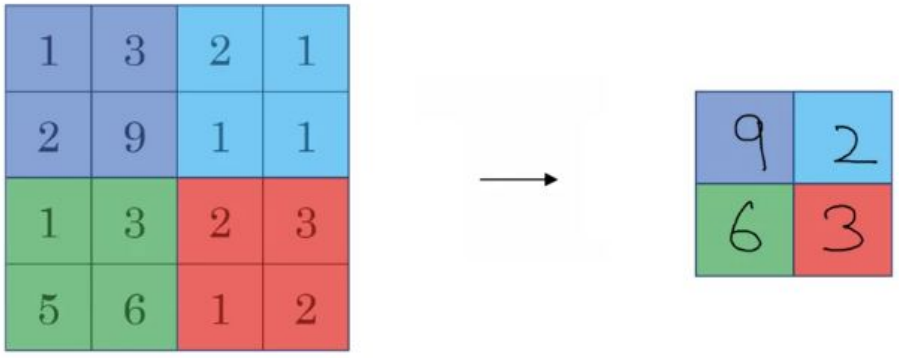
At this moment, we've already studied the convolution layers. We'll study more about the other two layers later in the course.

## Pooling Layers

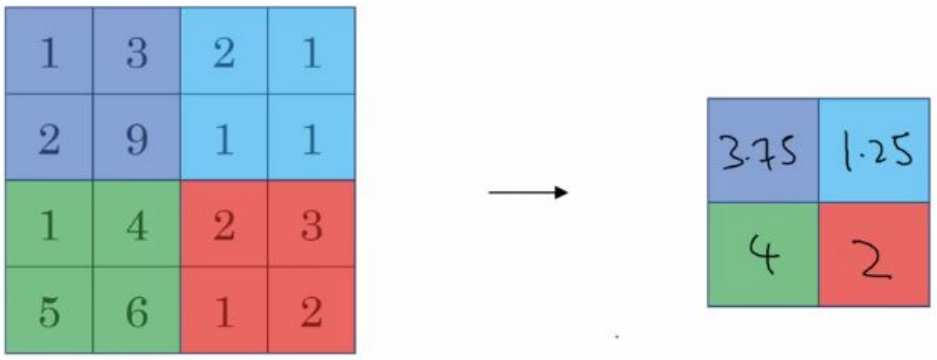
Convolutional Networks often use pooling layers to reduce the size of the representation, to speed the computation as well as make some of the features that detect a bit more robust. Let's start with an example of a Max pooling applied to a 4x4 matrix:

| | | | |
|---|---|---|---|
| 1 | 3 | 2 | 1 |
| 2 | 9 | 1 | 1 |
| 1 | 3 | 2 | 3 |
| 5 | 6 | 1 | 2 |

We'll use a filter with 2x2 dimension (f=2), and a stride s=2. What the Max pooling technique does is that it takes the maximum value of the current region and copies that value in the output matrix. In this case, we are dealing with a 4x4 input and a 2x2 filter, so the output is also going to be a 2x2 matrix. This process can be seen in the picture below:



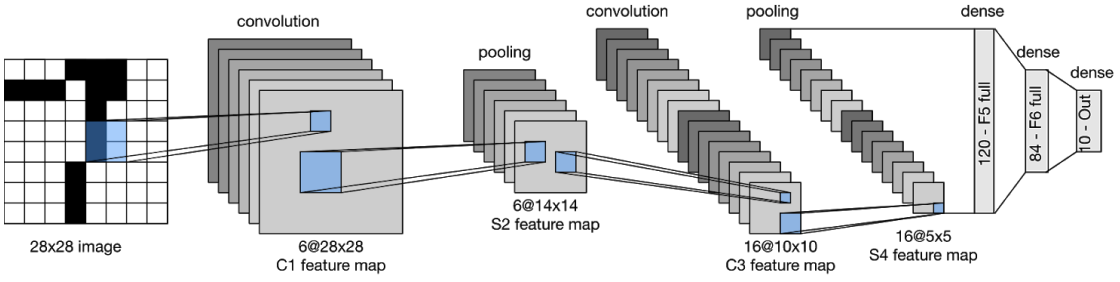

One interesting thing about max pooling is that it has a set of hyperparameters (f and s), but doesn't have parameters. This means that there is nothing to learn in the back propagation, making the computations much quicker.
There is another type of pooling that is not used that often, which is average pooling. As the name suggests, it takes the average of the numbers in the current region that the filter is located. Let's use the same matrix and filter hyperparameters of the previous example to show how it works:




You can also use the p (padding) hyperparameter, but this is almost never used in pooling.

### CNN Example

Let's do an example of number recognition, the LeNet-5 model. This model was introduced by (and named for) Yann LeCun, in 1998. This work represented the culmination of a decade of research developing the technology. In 1989, LeCun published the first study to successfully train CNNs via backpropagation. Let's take a look:




The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s. Each convolutional layer uses a 5x5 filter and a sigmoid activation function. Notice that the output has 10 elements, one for each digit from 0 to 9. Let's take a look at the activation shape, the activation size and the number of parameters in each component of the LeCun model:

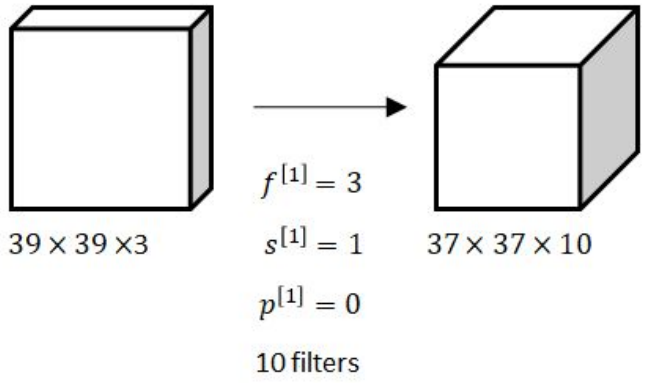
| | Activation Shape | Activation Size | # parameters |
|---|---|---|---|
| Input | (28,28,1) | 784 (28*28*1) | 0 |
| CONV1 | (28,28,6) | 4704 (28*28*6) | 156 ((5*5*1+1)*6) |
| POOL1 | (14,14,6) | 1176 (14*14*6) | 0 |

| CONV2 | (10,10,16) | 1600 (10*10*16) | 2416 ((5*5*6+1)*16) |
|---|---|---|---|
| POOL2 | (5,5,16) | 400 (5*5*16) | 0 |
| FC3 (fully connected) | (120,1) | 120 (120*1) | 48120 (400*120+120) |
| FC4 | (84,1) | 84 (84*1) | 10164 (120*84+84) |
| Output | (10,1) | 10 (10*1) | 850 |

Note that the activation size starts relatively big in the CONV1 step and ends up being quite small in the end. Also, the number of the parameters is much smaller in the convolutional layers compared to the dense layers and the number of parameters in the pooling steps is equal to zero (these steps only have hyperparameters).

## Why Convolutions?

There are two main advantages of convolutional layers over fully connected layers: parameter sharing and sparsity of connections. Let's see an example:

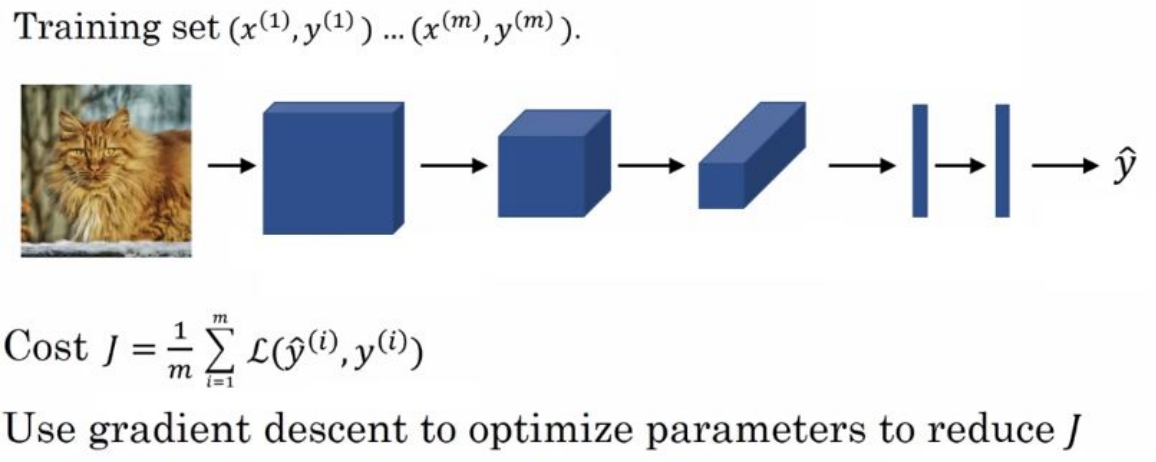



The dimension of the input is 39x39x3 and the dimension of the first Conv layer is 37x37x10. If we used fully connected layers in the example above, we would have 39*39*3*37*37*10 different parameters to train, this number is approximately 62 million. Notice that we are dealing with a very low resolution input image, with only 39x39 resolution, now imagine working with a 1000x1000 image, the number of parameters could easily reach billions and billions. Now if we look at the number of parameters using a convolutional layer, it is going to be (3x3x3+1)*10 = 280. Compared to 62 million, 280 parameters is a huge improvement in the number of parameters needed.
Obs: (3x3x3+1)*10 is based on the equation -> $(f^{[l]} \, x \, f^{[l]} \, x \, n_c^{[l-1]} + 1) \, x \, n_c^{[l]}$ . It is the sum of the numbers of weights and biases.

The reason for this small amount of parameters, is parameter sharing. A feature detector (such as a vertical edge detector, for example) that is helpful in one part of

the image, is probably useful in another part of the image (there are usually lots of vertical edges in an image).
The other reason is the sparsity of connections. In each layer, each output value depends only on a small number of inputs. If you use a 6x6 image and a 3x3 filter for example, each output depends only on 9 numbers of the input.

Putting all the content together with a cat recognition model:




## Case Studies

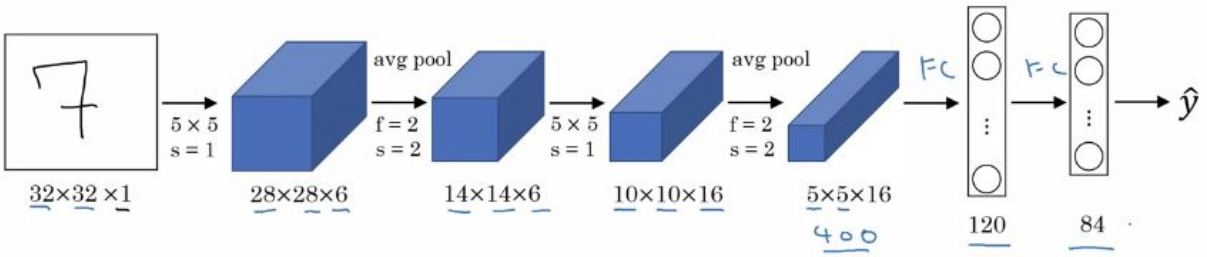
### Why look at case studies?

One of the best ways to get intuition on how to construct good neural networks is to see how different neural networks architectures work. We'll study some classic networks and some with particular characteristics:
Classic Networks:
-LeNet-5
-AlexNet
-VGG
Other networks:
-ResNet
-Inception

### Classic Networks

You'll learn more about the LeNet-5, AlexNet and VGG architectures. Let's start with the LeNet-5.
LeNet-5:
The LeNet-5 was already introduced in the "CNN Example" section. It was first designed to recognize digits. Back in the year the paper was written, people didn't

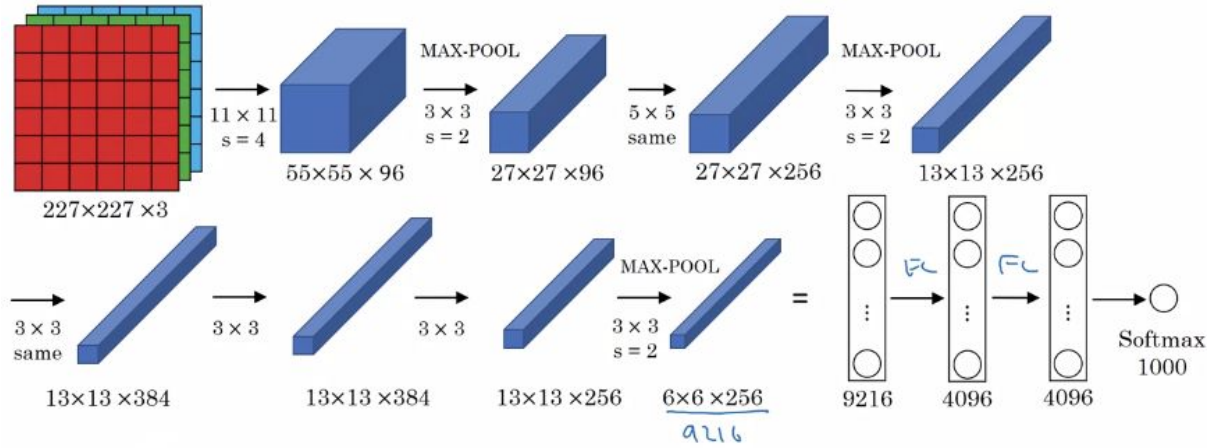
really use padding or Max Pooling, so as you go deeper in the network, the image sizes shrink $(n_H,\ n_w\ \downarrow)$ and the number of channels increases $(n_c\ \uparrow)$. You can see one example below:




This architecture has about 60k parameters, that is a quite small number. This architecture originally used the sigmoid and tanh functions as activation functions of the network.

AlexNet:

The name AlexNet comes from the author of the architecture, Alex Krizhevsky. It is very similar to the LeNet-5 architecture, but for a much bigger input image (while the LeNet-5 has about 60k parameters, the AlexNet has about 60M parameters).

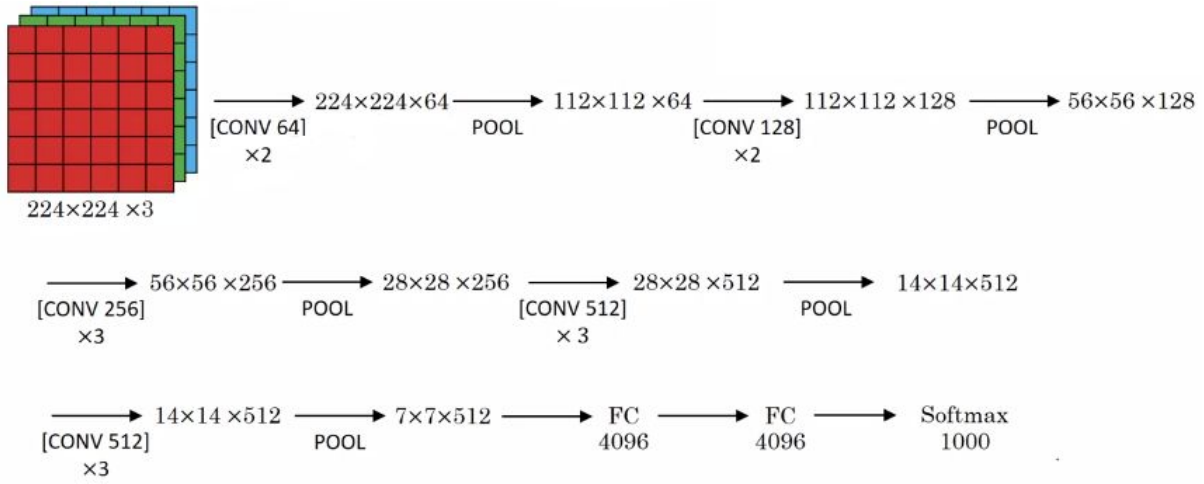



This architecture uses a ReLU activation function.

VGG-16:

Instead of having lots of hyperparameters, the VGG network focuses on having conv layers with only 3x3 filters, stride = 1 and same convolutions. The pooling layers only use 2x2 MAX-Pooling with stride = 2. The 16 in the name of the architecture, refers to the fact that the neural network has 16 layers with weights and VGG is the name of the group in Oxford University that created the architecture (Visual Geometry Group). The notation [CONV 64] x2, means that 2 convolutional layers with 64 filters each has been applied. This architecture has about 138M parameters.

CONV = 3×3 filter, s = 1, same  MAX-POOL = 2×2 , s = 2

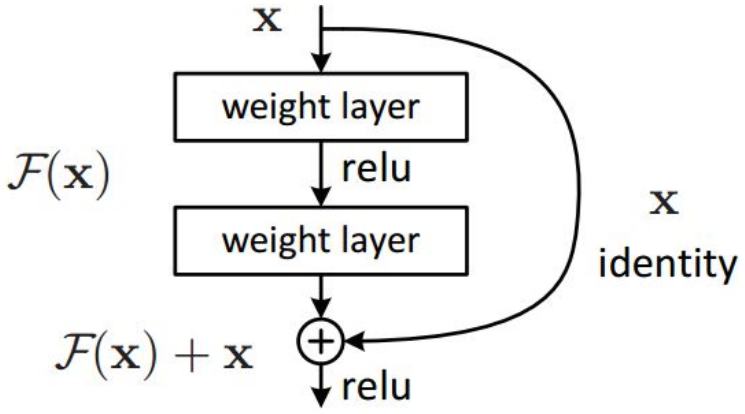



As you can see, the values of height and width decrease as you go deeper in the net, and the number of channels increases.

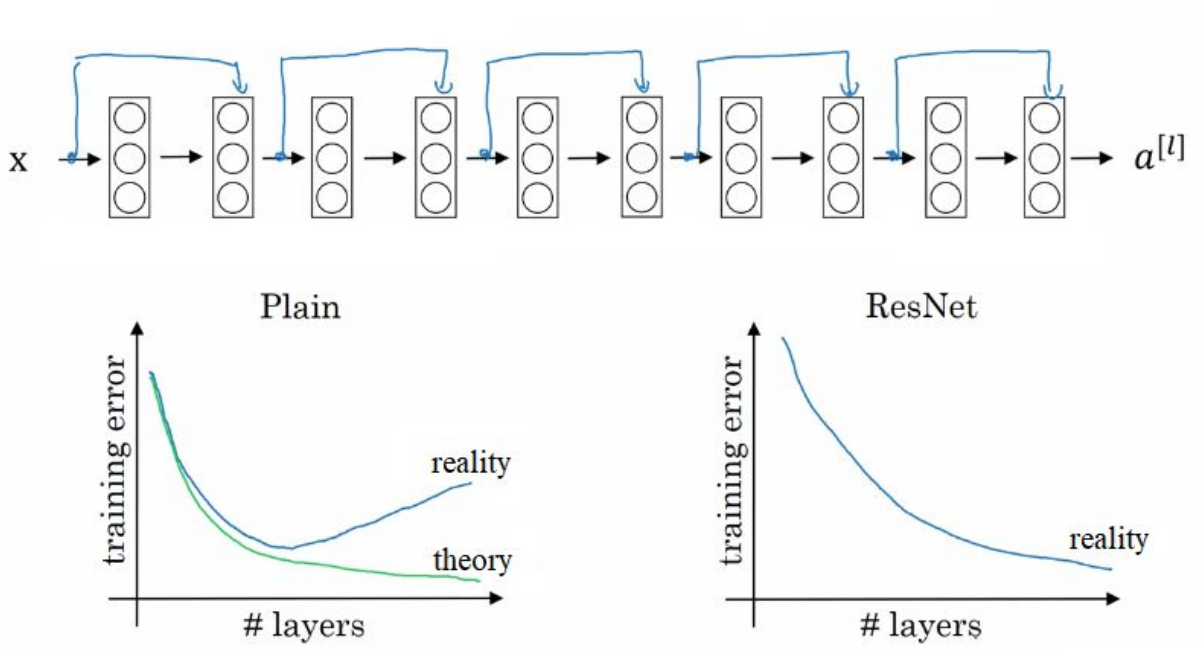
### Residual Networks (ResNets)

Deep neural networks are usually difficult to train because of vanishing and exploding gradients. To solve this problem, you'll learn about skip connections, which allows you to take the activation from one layer and feed it to another layer even deeper in the neural network. Using this technique, you'll learn about the ResNet, which enables you to train very deep networks (sometimes with over 100 layers).

ResNets are built of residual blocks. The idea behind this architecture is that when you allow an activation value to "short cut" over lots of layers, you can make your neural network much deeper. You can see how it works in the image below:




As you can see, the value x is also used in future layers. This is called a residual block, and stacking lots of residual blocks allows you to make a residual network (the networks without residual blocks are called "plain networks").

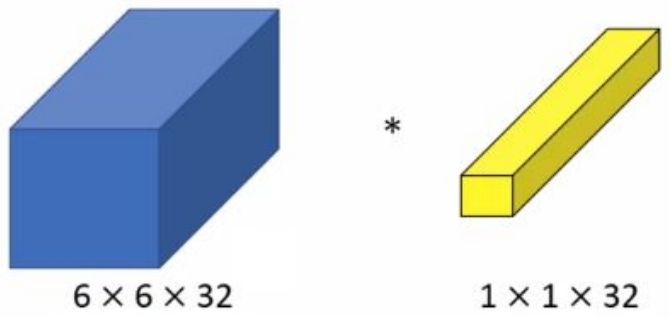



In theory, if we train a much deeper plain neural network, the performance would improve, but what happens is that at some point the error starts growing and the performance doesn't improve. Using ResNet networks solves this problem, and the performance usually improves if you use more deeper layers.

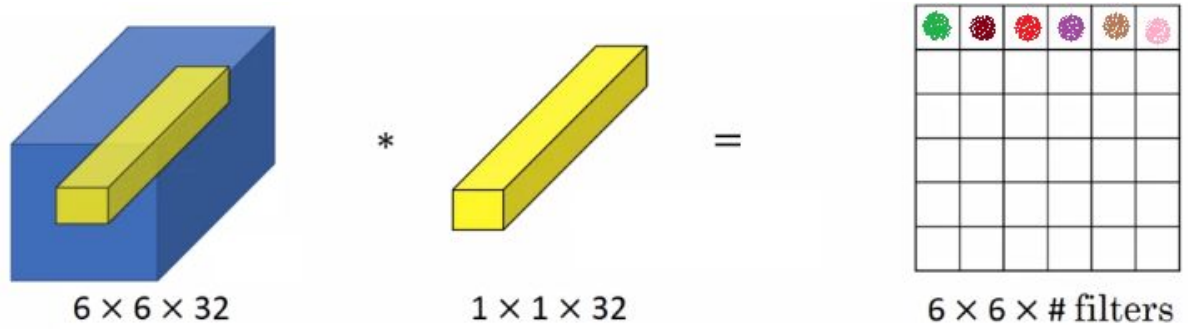
### Networks in Networks and 1x1 Convolutions

A 1x1 convolution, when applied to a matrix with a single channel results in another matrix multiplied by the value of the filter:


However, if you apply a 1x1 filter to an image with lots of channels, the convolutional operation results in an element-wise multiplication:



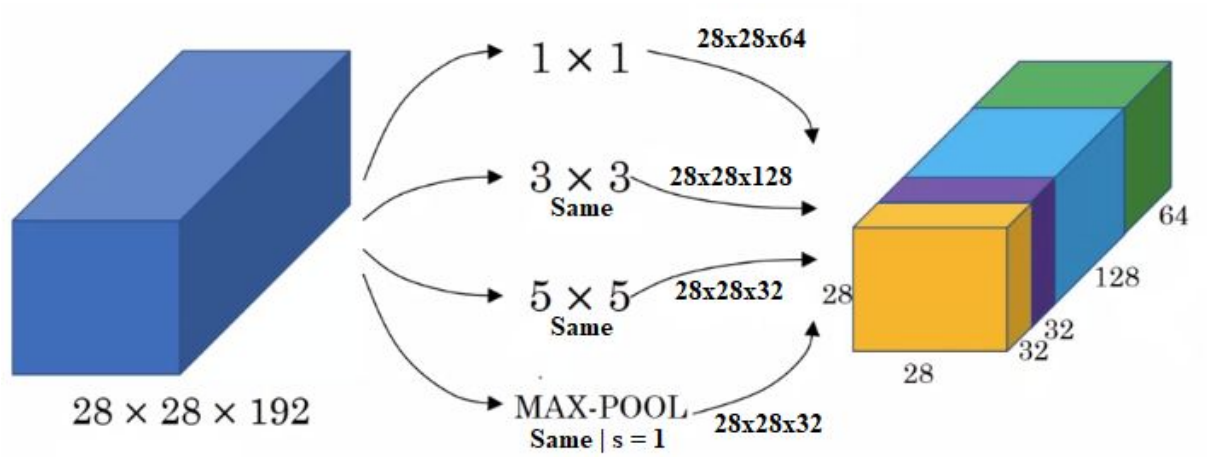

You can then apply a ReLU function after that operation, and the result will be a single matrix with 6x6 dimension.



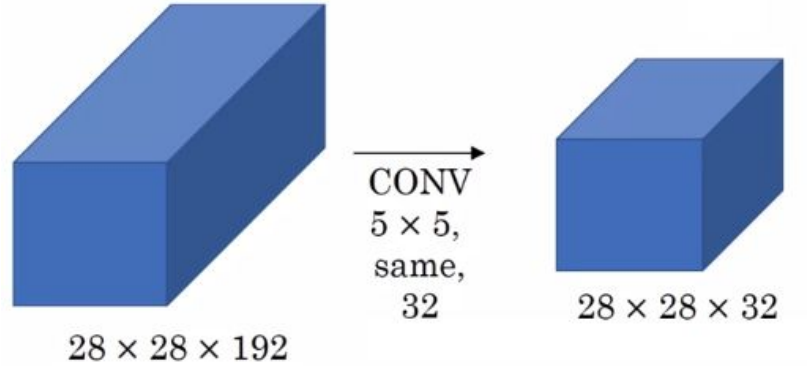

If you have multiple filters the output of the convolutional operation is going to have more than one channel. This operation is also called Network in Network. The 1x1 convolutions are really helpful when you need to shrink the number of channels in the neural network. You have now an arsenal of tools to reduce sizes: if you want to shrink the height and width you can use a pooling layer and if you want to shrink the number of channels use a 1x1 convolution. If you want to keep the number of channels, the effect of a 1x1 convolution is that it adds non-linearity to the layer, allowing the learning of more complex functions.
The 1x1 convolutions are very useful for building the inception networks that you are going to see in the next section.

## Inception Network

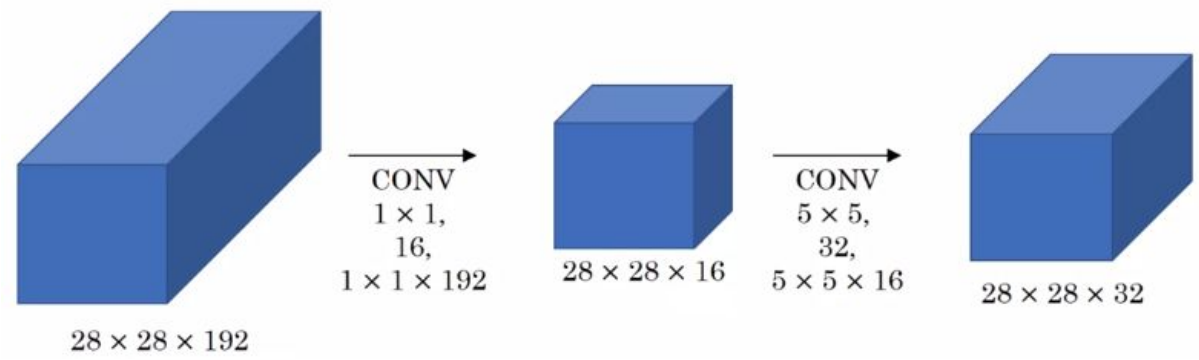
When designing a layer for a ConvNet, you might have to choose between different sizes of filters, different numbers of channels, if you want to use a pooling layer or not… . The idea  behind the inception network is to do all you want at the same time. You can see an example below:



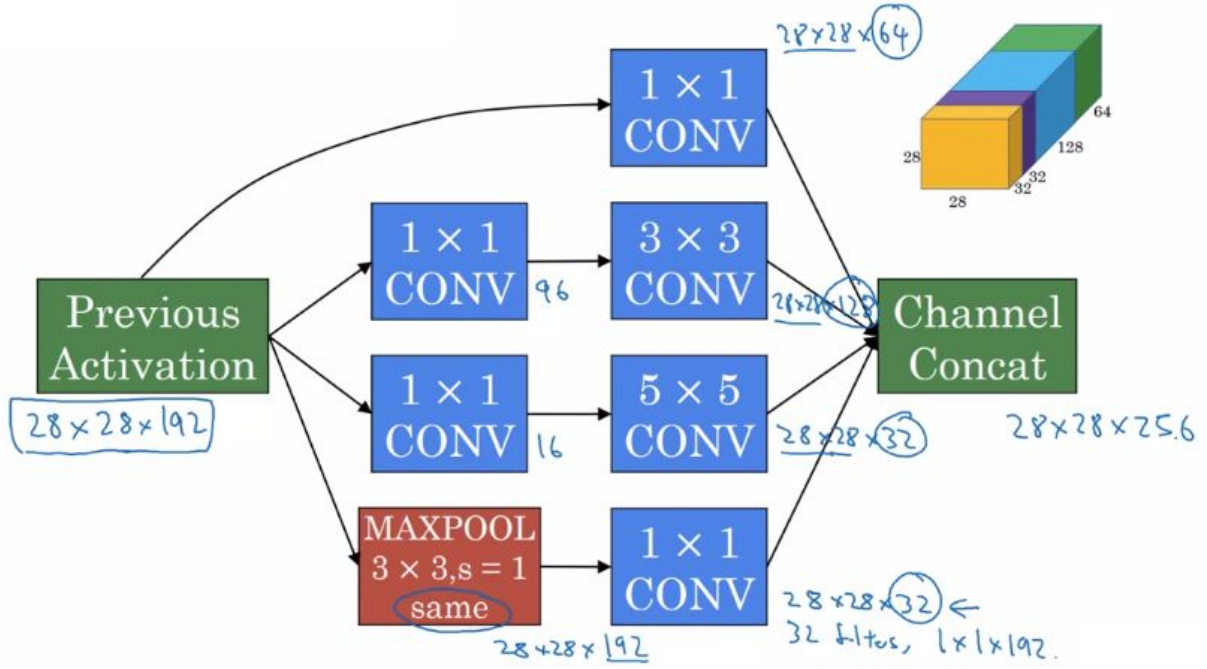

Using this technique allows you to make the number of channels in the output even bigger than the number of channels in the input (in this case, number of channels of the input = 192, and the number of channels of the output = 256).
This technique sounds pretty good, however there is a big problem: computational cost. Let's show an example:



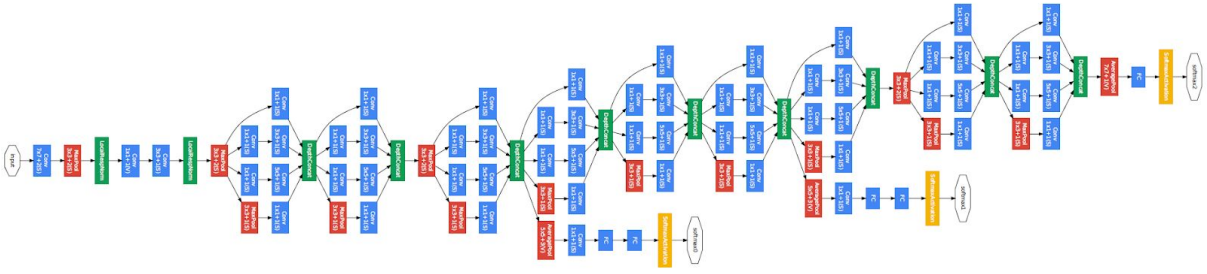

In this case, we have 32 filters with 5x5x192 dimensions. The output is 28x28x32, so we need to perform (5x5x192)x(28x28x32) multiplications, that is equal to more than 120 million operations. A modern computer can compute all these operations, but notice that we are only dealing with a single small example, so this value can easily surpass billions and billions of operations for more complex applications. However, there is a technique that allow us to cut the number of operations by a factor of 10, that is to use the 1x1 convolution:



As you can see, the input and output dimensions are still the same. We've taken the huge volume of data in the input, shrank it into a much smaller intermediate volume, and then used our filter to compute the output. Sometimes the middle layer is called the "bottleneck layer". The cost of the 1x1 convolution operation is only (1x1x192)x(28x28x28)=2.4M. The cost of the second convolutional layer is (5x5x16)x(28x28x32)=10.0M. The total number of multiplications is about 12.4M. As you can see, using the 1x1 convolutional layer, we reduced the numbers of operations from 120M to only 12.4M.
In the image below you can see an example of a inception module:




An inception network is composed by many of these inception modules. You can see a full representation of an inception network below:

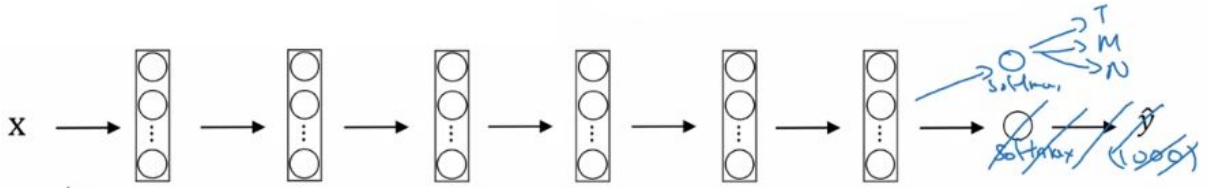


This is called the GoogLeNet network (the name of the team that participated in the ILSVRC14 competition, composed by Google employees and some university researchers). As you can notice, there are lots of inception modules and even some side branches that try to predict the output based on the values of a hidden layer (using softmax activation).
In the table below, you can see how some details about this network:

| type | patch size/ stride | output size | depth | #1×1 | #3×3 reduce | #3×3 | #5×5 reduce | #5×5 | pool proj | params | ops |
|---|---|---|---|---|---|---|---|---|---|---|---|
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K | 34M |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | | 64 | 192 | | | | 112K | 360M |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K | 128M |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K | 304M |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K | 73M |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K | 88M |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K | 100M |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K | 119M |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K | 170M |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K | 54M |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | | |

## Transfer Learning

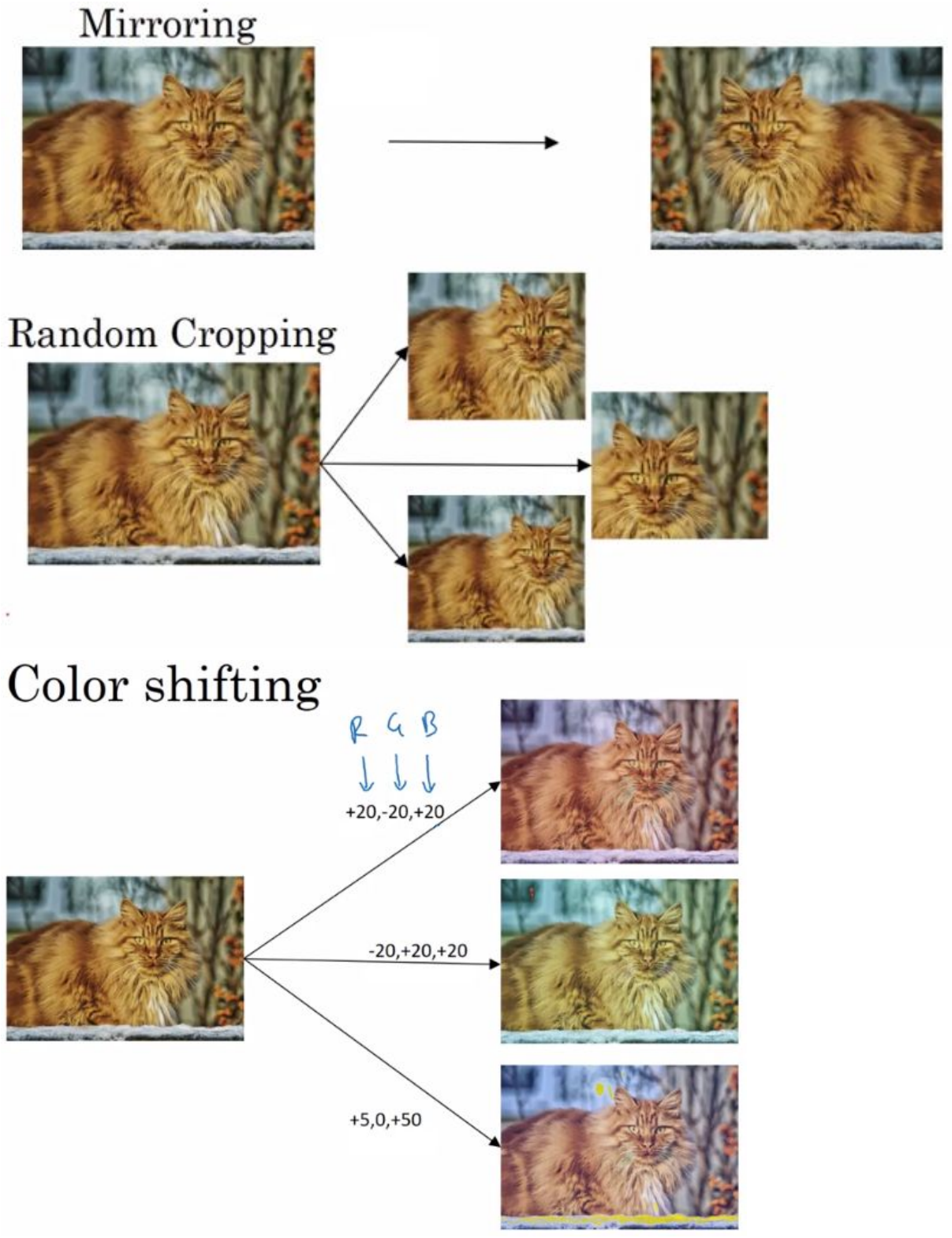
If you're building a computer vision application, rather than training the weights from scratch using random initialization, you often make much faster progress if you download weights that someone else has already trained, and use that as pre-training and transfer that to a new task that you might be interested in. For example, imagine you want to build a cat recognition system to detect only two species of cats. In this case, you might look at the internet for open source projects (in Github, for example) in the cat recognition application, and use the already existing weights to build your specific model. Let's look at an example:




In this model, it's used a softmax classifier to identify the species of the cat among 1000 different species. If you want to identify only two species, you should "freeze" the learning in the previous layers and only train the parameters of the final layers to recognize specific characteristics of cats (the first hidden layers are already trained to recognize different characteristics, and you don't want to lose that information).
If you have a lot of data, you can also use the already existing weights to avoid initializing your network with random weights, accelerating the learning process.

## Data Augmentation

Data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. Let's see some methods:




You can also use rotation, shearing, local warping, and so on… .

**Important tips in computer vision: Use open source code**

1) Use architectures of networks published in the literature
2) Use open source implementations if possible
3) Use pretrained models and fine-tune on your dataset

## Object Detection

### Object Localization

You've studied about image classification in the previous sections. You will learn now how to localize an object in an image.

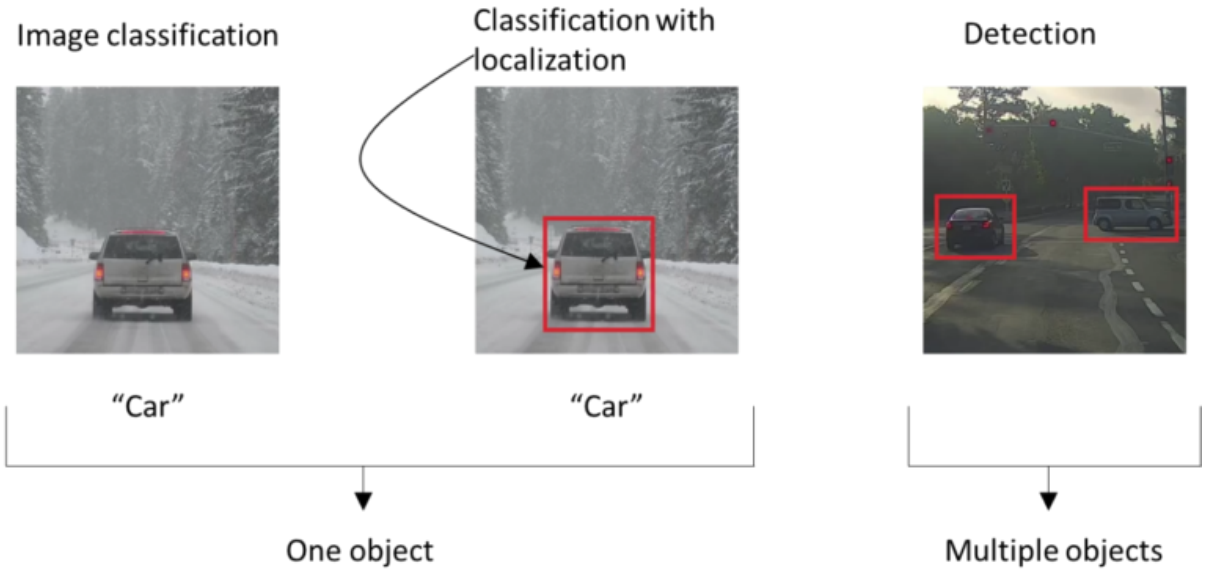
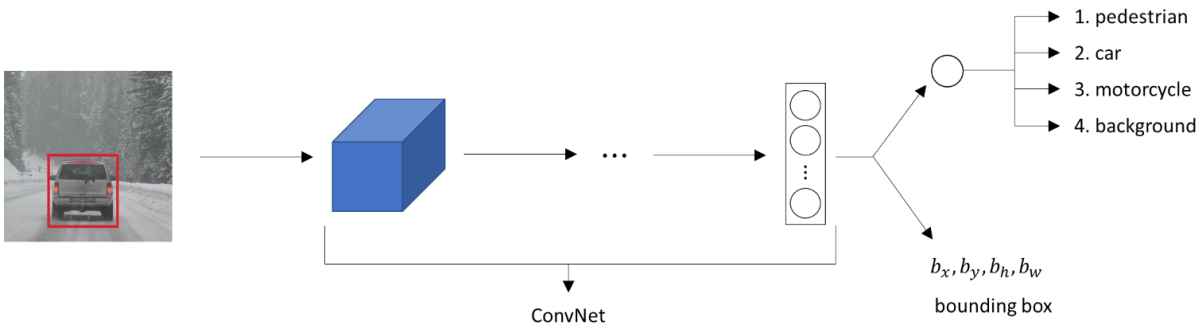
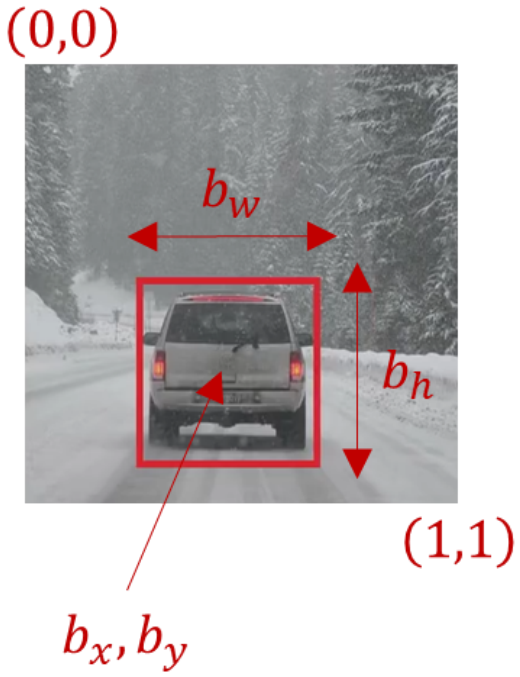



Image classification only detects if the object is in the picture, while image classification with localization detects if the object is in the picture + returns its location. If you want to detect and know the localization of multiple objects in the same image, you should use a detection system.

If you are building a self-driving car, maybe your object categories are a pedestrian, a car, a motorcycle and the background (this means none of the other options). There are four classes, so you need a softmax with 4 possible outputs. To localize the object in the image, you need 4 more output numbers, that are $b_x$, $b_y$, $b_h$, $b_w$. These four numbers parameterize the bounding box of the detected object.




Here is an used notation for object classification:




We consider the upper left point of the image as (0,0) and the bottom right point as (1,1). The $b_x$ and $b_y$ numbers specify the midpoint (the middle point of the red rectangle) while $b_h$ and $b_w$ specify the height and width of the box. In this case, we'll probably have $b_x = 0.5$, $b_y = 0.7$, $b_h = 0.3$ and $b_w = 0.4$.
The target label y is going to be a vector composed by:

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

- $p_c$ tells if the picture is an object in the picture. In the example above, if there is a car, a pedestrian or a motorcycle in the image, $p_c = 1$. If there is not, $p_c = 0$ (in this case, this is a background image);
- $b_x$, $b_y$, $b_h$ and $b_w$ represent the bounding box for the detected object;
- $c_1$, $c_2$, $c_3$ represent what class has been recognized (c1 = pedestrian, c2 = car, c3 = motorcycle).

If $p_c = 1$, the target label y is going to have all the 8 components, if $p_c = 0$, all the other components are not important anymore, because there is nothing relevant to detect in the image.

The loss function will be like this:

$$L(\hat{y}, y) = (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \ldots + (\hat{y}_8 - y_8)^2, y_1 = 1$$

$$L(\hat{y}, y) = (\hat{y}_1 - y_1)^2, y_1 = 0$$

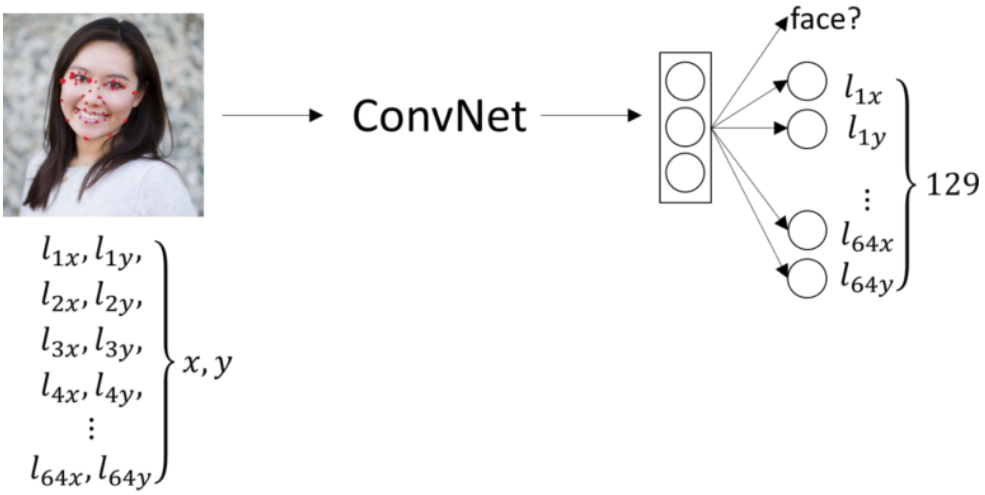
Obs1: $y_1 = p_c$
Notice that when $y_1$ =1, the loss function will have 8 components (all the components of y), and when $y_1$ = 0 y will have only one component.
Obs2: we've used the squared error to simplify the loss function, but in practice we could use a log loss function for the c1, c2 and c3, keep the square error for the bounding box coordinates and use a logistic regression loss for the $p_c$ value.

## Landmark Detection

In general cases, you can have a neural network just output X and Y coordinates of important points in an image. These important points are called landmarks. Let's see an example of a face recognition application.

$$\begin{matrix} l_{1x}, l_{1y}, \\ l_{2x}, l_{2y}, \\ l_{3x}, l_{3y}, \\ l_{4x}, l_{4y}, \\ \vdots \\ l_{64x}, l_{64y} \end{matrix}$$

In this case, we've selected 64 landmarks in the woman's face (every landmark is composed of two coordinates - x and y). These landmark points can determine the shape of her face, if she is smiling or not, the shape of her eyes, etc.



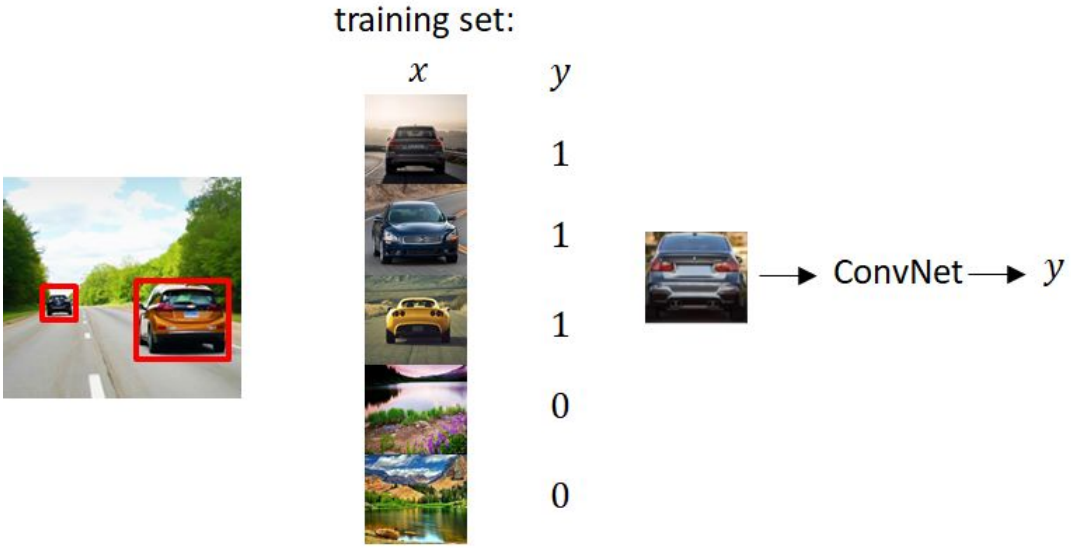

In this case, the label of the training set is going to have 129 output units. 1 to determine if it is a face or not, and 128 representing the landmarks (64 x and 64 y). The detection of landmarks in a face is a very used application, mostly because of social media camera filters (putting a crown or a hat in the person, for example).
If you are interested in pose detection, you could also define a few key positions like the midpoint of the chest, left shoulder, left elbow, wrist and so on. One example can be seen in the picture below:

$$l_{1x}, l_{1y},$$
$$\vdots$$
$$l_{32x}, l_{32y}$$

In this case, we've used 32 landmark coordinates to specify the pose of the person.

### Object Detection

We've learned about object localization as well as landmark detection. Let's study now how to build up an object detection algorithm.




We can first create a labeled training set with closely cropped examples of cars and some other pictures that are not pictures of cars. Given this labeled training set, we can then train a convnet that uses an input image and outputs whether there is a car (y=1) or not (y=0). To make the detection, we can use the sliding windows detection technique.

The sliding windows technique consists of picking a certain window size, using it to crop a certain region of the input image and use the convnet to detect if there is a car in the picture or not. You can see how it works in the image below:

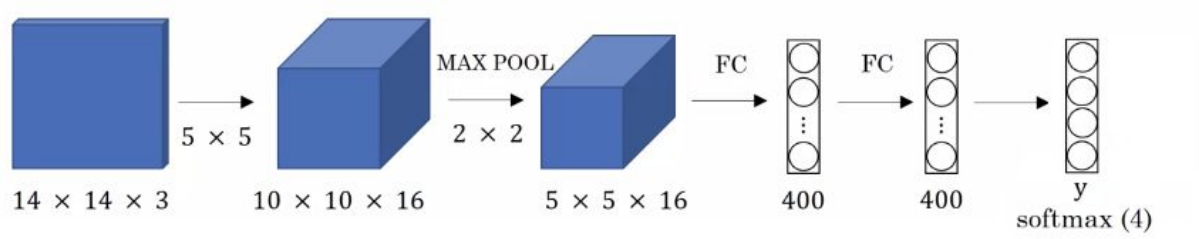
In this case, the size of the window is in the right and the selected part of the input is in the left. The convnet output is going to be y=0, because there is no car in the selected region. The sliding window technique then slides to the next region of the image, repeating the process of verifying if there is a car or not.

The process ends when the window slides throughout the entire image. Notice that different strides can result in different detections. To make sure that we detect correctly all the cars in the image, we use different three different windows sizes and repeat the detection process:

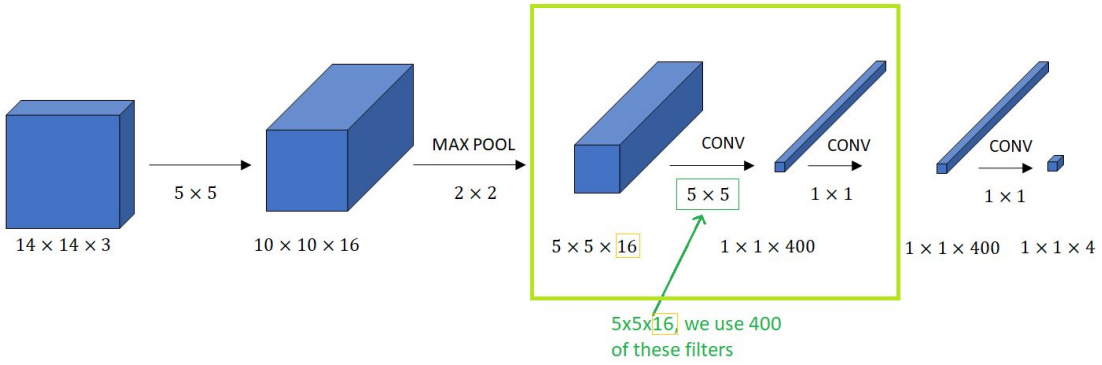
One of the problems of this technique is that it has a high computational cost, because you're cropping out so many different square regions in the image and running each of them independently through a convnet. Fortunately however, this problem of computational cost has a pretty good solution, that is to use a convolutionally implementation of the sliding windows technique, you'll see that in the next section.

## Convolutional implementation of Sliding Windows

The convolutional implementation of sliding windows solves the problem of high computational cost of this method. To understand better how it works, let's see how to turn a fully connected layer into convolutional layers.

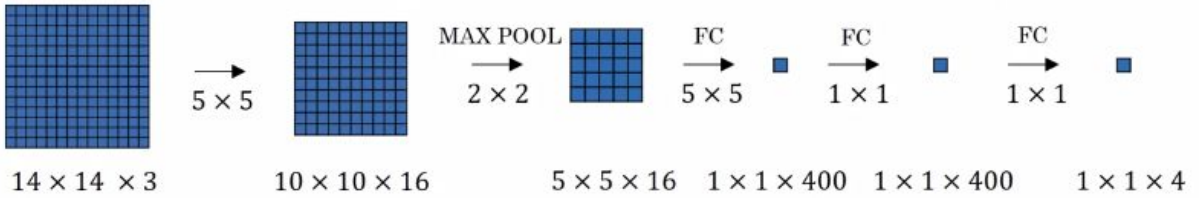



In this case, we want to turn the two fully connected layers into convolutional layers. If we perform a convolution operation in the third layer with a 5x5x16 filter, the result is going to be an 1x1 dimension output. So in order to turn all the 400 units in the fully connected layer, we need to use 400 filters with 5x5x16 dimension. Then, to turn the last fully connected layer to a convolutional layer, we have to use 400 filters with 1x1 dimension. Finally, we turn the output layer to a convolutional layer as well, using a softmax function to give a 1x1x4 dimension layer, taking the place of the four numbers the network was outputting.

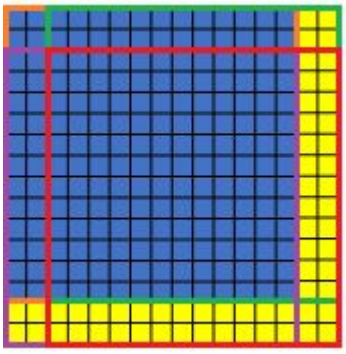



Mathematically, the new convolutional layers are the same as the fully connected | output layers. Let's see how the convolution implementation of sliding windows works.

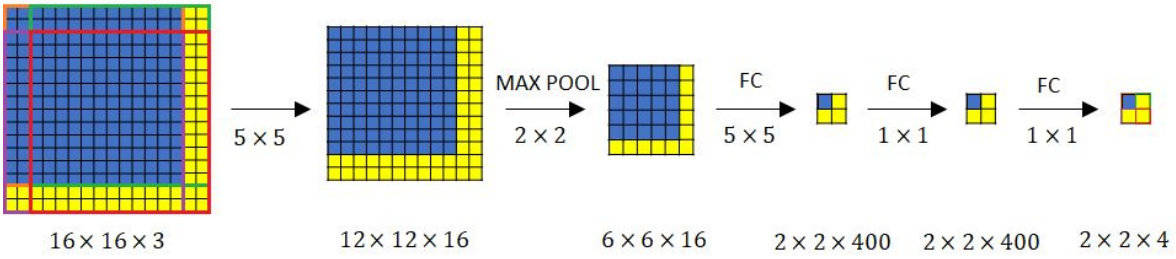
Let's say that the input dimension of our convnet is 14x14x3. The full convnet can be seen below:




Our test set image is 16x16x3. We could crop the image into 4 14x14x3 smaller images and pass them through the convnet, but there is an optimization we can do. In order to run the sliding windows technique, we can divide the image into smaller images using stride and run the image through a convnet. In this case, the stride is equals to 2, making the smaller images' dimension 14x14x3. Each one of the 4 smaller images can be seen below (the yellow squares represents the additional pixels compared to the original convnet input layer):

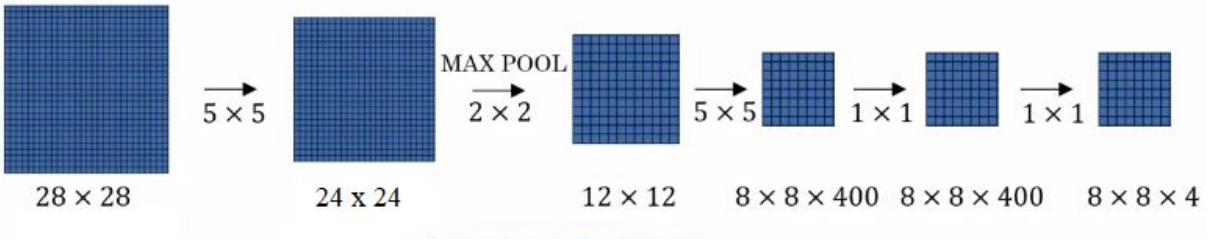


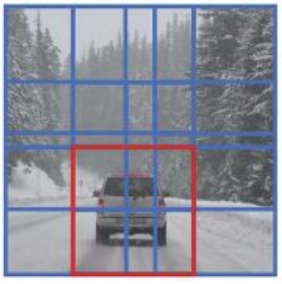
This is the convnet we are using with this image:




Notice that the output is the same compared to the output if we use the o convnet with 4 separate images.

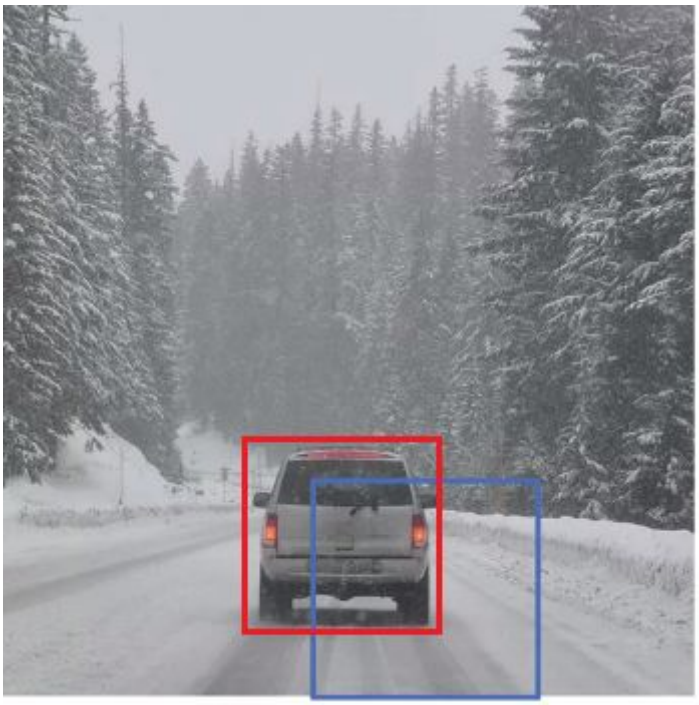
This way of performing the sliding windows technique allows an optimization in the computational cost, because lots of values are shared between one smaller image and the other. So instead of forcing you to run four propagations on four subsets of the output image independently, it combines all four into one form of computation and shares a lot of the computation in the regions of image that are common. Here is another example:



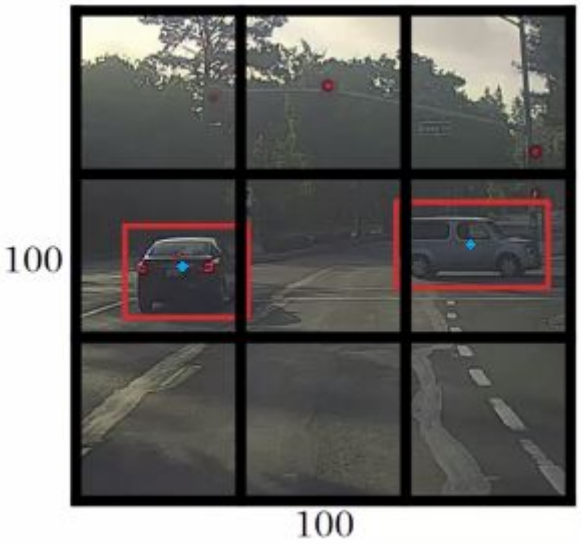
## Bounding Box Predictions

In the last section, you've learned how to use a convolutional implementation of sliding windows. That's very efficient implementation but still has a problem of not outputting the most accurate bounding boxes. In this section you'll learn how to get more accurate bounding boxes. As you can see in the picture below, the sliding windows implementation sometimes doesn't identify the proper bounding box (the red rectangle).



A good way to get more accurate bounding boxes is with the YOLO ("You Only Look Once") algorithm. To learn how this algorithm works, let's take an example of a 100x100 image.

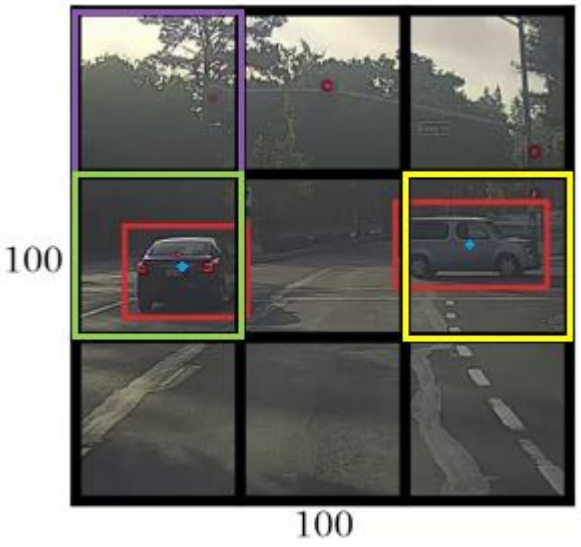
First, we place down a grid on this image (for learning purposes we'll use a 3x3 grid, but in reality we could use a much finer grid, like 19x19).



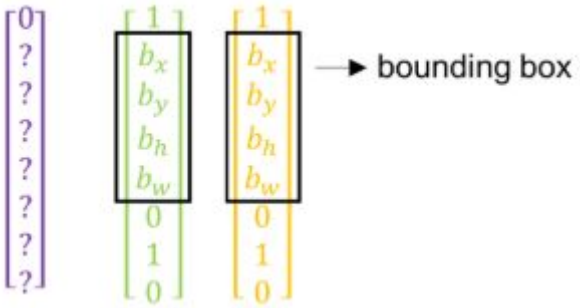

The idea is, we're gonna take the image classification and localization algorithm and apply it to each of the nine grids. The output labels for training, for each grid cell is going to be:

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

Let's choose some grids to make an analysis:

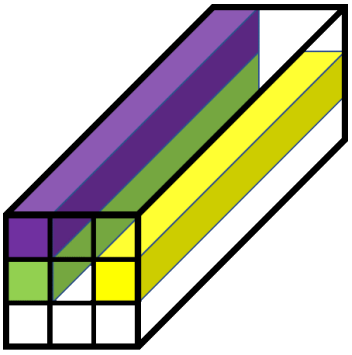


In this case, the first grid (the purple one) doesn't have a pedestrian/car/motorcycle, so the output is going to be the purple vector below. The 4th (green) and 6th (yellow) grids are pictures of cars, so the output is going to be the green and yellow vectors.




What the YOLO algorithm does is it takes the midpoint of the objects (the blue points in the image above) and then assigns the object to the grid cell containing the midpoint. In our example, even though the central grid cell has some parts of both cars, we'll pretend that it doesn't have any object (because the midpoint of both vehicles is in another grid cell).

For each grid cell in the image, we'll have an eighth dimensional vector, and because we have 3x3 grids, the total volume of the output is going to be 3x3x8.

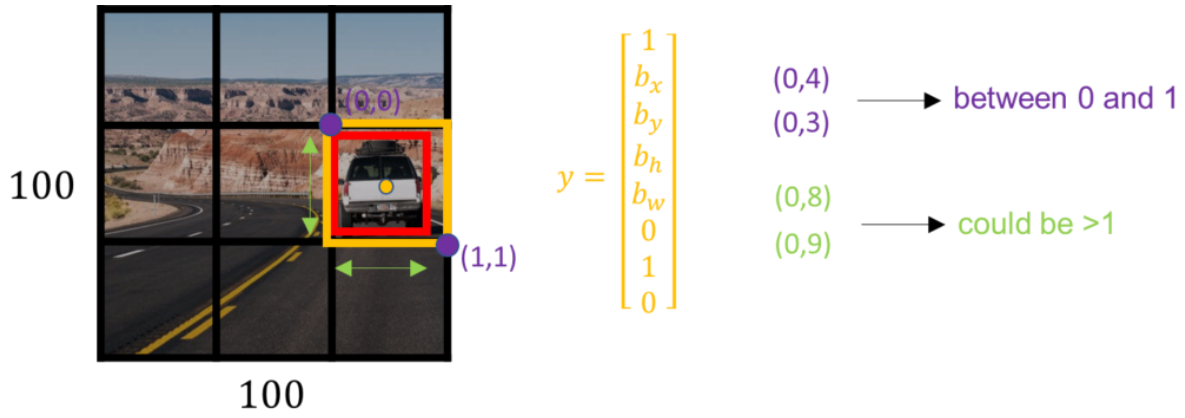


To train this neural network, we'll use an 100x100x3 dimensional input, then we have a usual convolutional neural network with convolutional layers, max pool layers, and so on. This neural network maps from an input image to a 3x3x8 output volume. We use the input x image to feed the neural network, run forward propagation and make a prediction, then run back propagation to tune the parameters.

As long as we don't have more than one object in each grid cell, this algorithm should work properly. When multiple objects are in the same grid cell the algorithm doesn't have the best performance, but we'll talk about this problem later.

Notice that this is a convolutional implementation because we're not assessing this algorithm nine times on the 3x3 grid or 361 times if we are using the 19x19 grid. Instead this is one single convolutional evaluation, that's why this algorithm is so efficient.

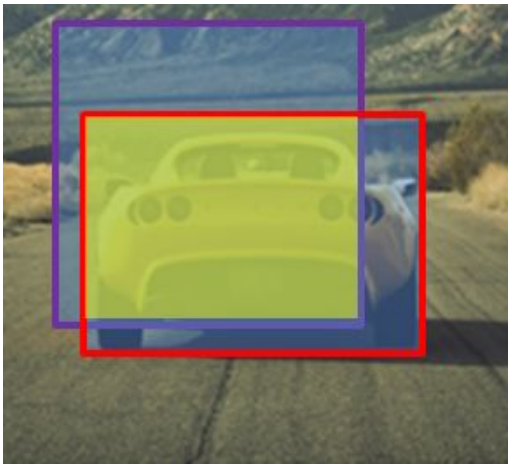
Let's now learn how to specify the bounding boxes using the example below:



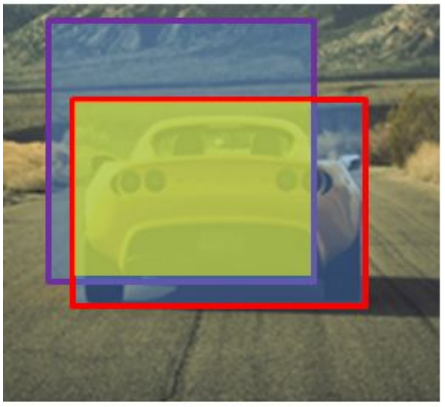

In this case, the upper left corner of the grid cell is the coordinate (0,0) and the bottom right corner is (1,1). We'll probably have $b_x = 0.4$, $b_y = 0.3$, $b_h = 0.9$ and $b_w = 0.8$. If the bounding box is bigger than the grid, the values of $b_h$ and $b_w$ could be bigger than 1. The values of $b_x$ and $b_y$ on the other hand are always less than 1.

## Intersection Over Union

The "Intersection Over Union" is a function that it's used to evaluate your detection algorithm. Let's start with an example:



Imagine that the red rectangle in the picture above is the ground-truth bounding box and your algorithm outputs the box in purple. To evaluate if your detection is a good outcome, the IOU computes the intersection over union of these two boxes.
The intersection of these boxes is the yellow region and the union of these boxes is the whole blue region including the yellow interception. The operation done can be seen below:



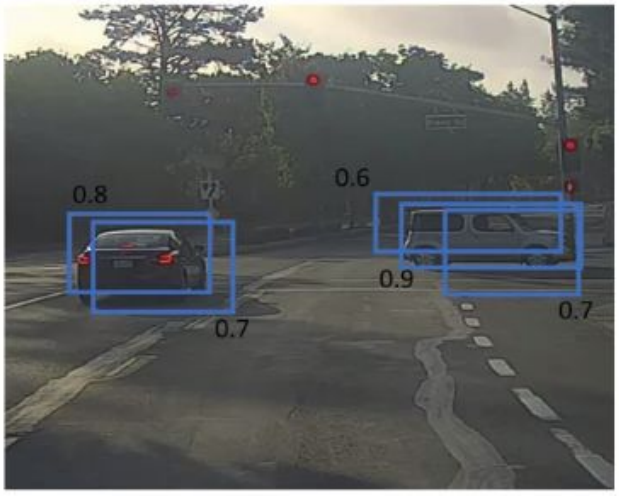
Intersection over union (IoU)

$$= \frac{\text{size of } \square}{\text{size of } \square}$$

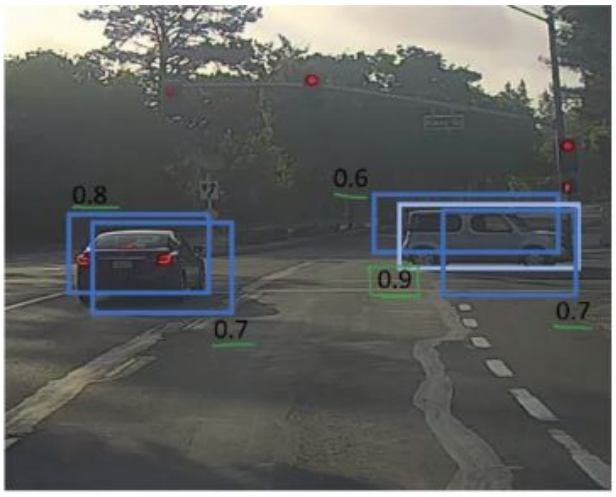
By convention, the answer will be correct if the IoU is greater $\geq 0.5$. This is just a convention, so if you want to be more stringent, you can use a higher threshold.

### Non-max Suppression

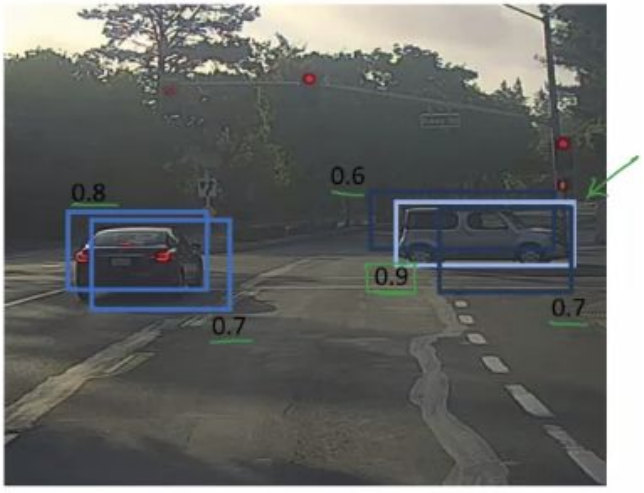
One of the problems of object detection is that your algorithm may find multiple detections of the same object, rather than detecting it just once. Non-max suppression is a way for you to make sure that your algorithm detects each object only once.



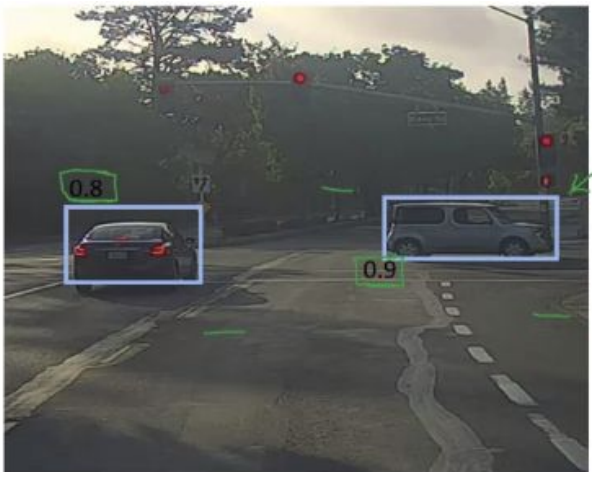

The picture above shows the problem of multiple detection. The non-max suppression algorithm first looks at the probabilities associated with each of these detections, takes the larger one and highlights that detection.




Having done that, the non-max suppression then looks at all of the remaining rectangles and all the ones with a high overlap (a high value of IoU) will get suppressed. So the rectangles with the 0.6 and 0.7 probabilities overlap a lot with the highlighted rectangle, so they are going to be suppressed.




Next, the non-max suppression algorithm will go to the remaining rectangles and will repeat the process. The rectangle with 0.8 probability is going to be highlighted and the rectangle with 0.7 probability is going to be suppressed, due the fact that it has a lot of overlap with the highlighted rectangle. We'll end up with only two rectangles that is the correct number of objects in the image.




Let's see how the algorithm is implemented. We'll use the same image above, with a 19x19 grid. To simplify the algorithm and make the learning easier, let's assume that we only want to detect cars, so we get rid of the c1, c2 and c3 classes in the output vector. The algorithm is going to be like this:

19× 19

Each output prediction is: $\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \end{bmatrix}$

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 $\geq 0.5$ with the box output in the previous step

If your object detection algorithm detects more than one class of object, the correct thing to do is to run the non-max suppression algorithm independently for each class. In this case, if we want to detect cars, pedestrians and motorcycles, we would need to run the non-max three times.

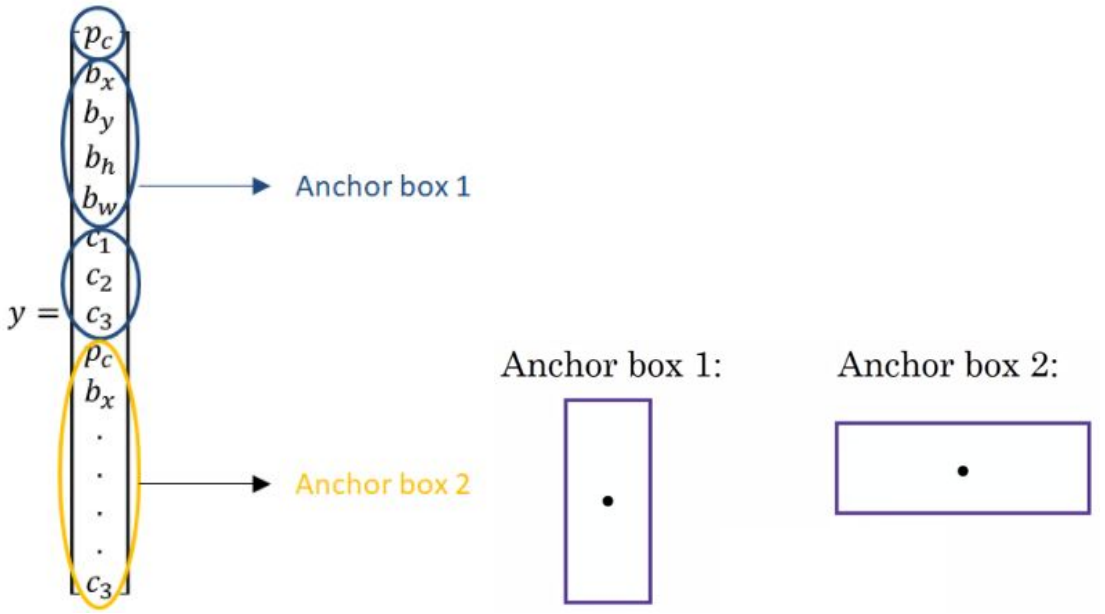
## Anchor Boxes

Another problem with object detection is that each of the grid cells can detect only one object. You can use the idea of anchor boxes to solve this problem. Let's start with an example:

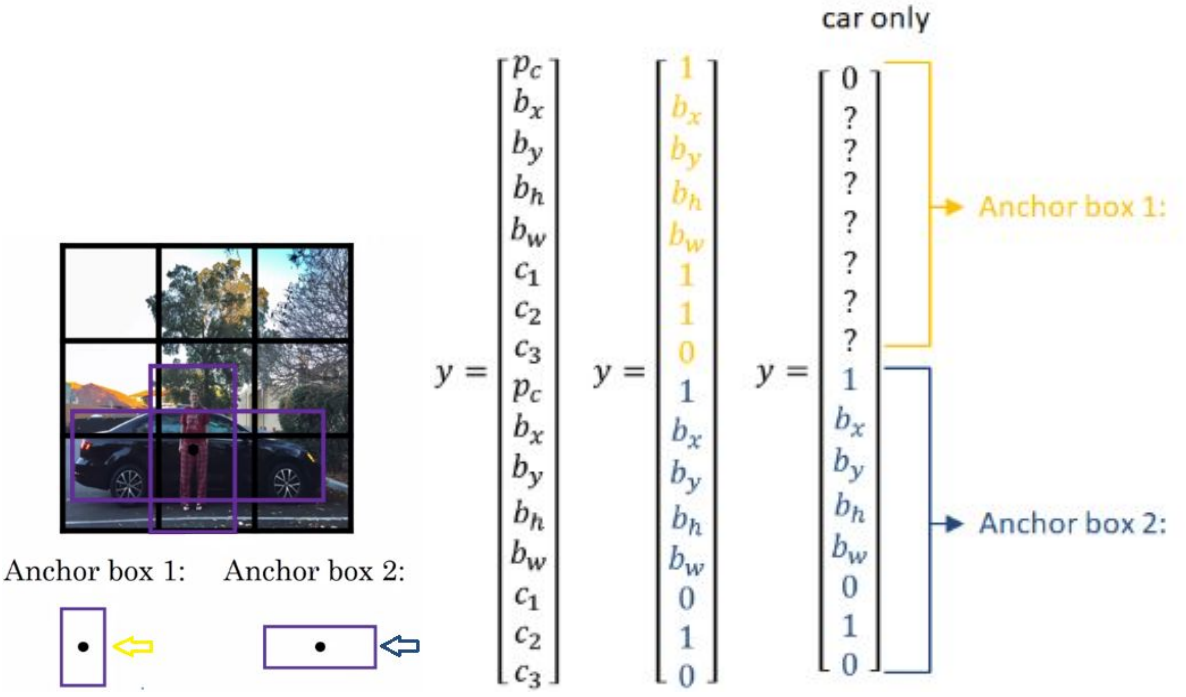
Notice that the midpoint of the car and the midpoint of the pedestrian are almost in the same place and both of them are in the same grid cell. If Y outputs a vector where you are detecting three classes, it won't be able to output two detections. So instead of using the traditional Y output vector:

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

We'll use the idea of anchor boxes, and the output is going to be:




In this case the anchor box 1 represents the pedestrian and the anchor box 2 represents the car. So previously each object in the training image was assigned to the grid cell that contains that object's midpoint (and the output dimension in our example was 3x3x8). With the anchor boxes algorithm, considering two anchor boxes, each object in the training image is assigned to the grid cell that contains the object's midpoint and anchor box for the grid cell with highest IoU (the output dimension is going to be 3x3x16).
Let's see an anchor box example:




A simple approach to choose the number of anchor boxes is to choose a number that spans the objects we wish to detect. For example, if our model needs to detect 3

classes, one reasonable number of anchor boxes would be 3 as well. A more advanced technique is to apply k-means clustering algorithm.
Notice that the algorithm won't work well if there are 2 objects in the same grid cell, and both of them have the same anchor box. However these cases are very rare, mainly if you use a 19x19 grid (these two objects need to share the same grid cell - 2 objects in the same cell on a total of 361 possible cells, that's usually a very low probability).

## YOLO Algorithm

In this section, we'll put all the components we've seen in the previous sections together in the YOLO algorithm. Let's start with an example in the training set:

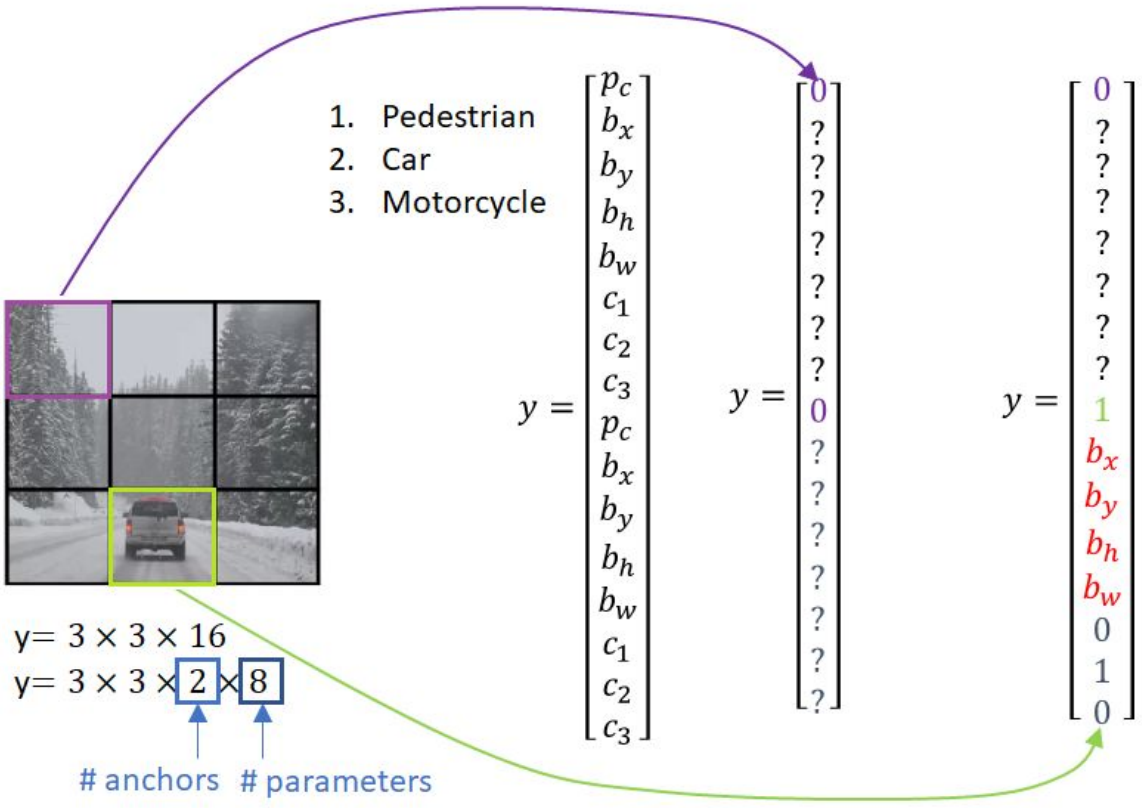
1 - pedestrian
2 - car
3 - motorcycle

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

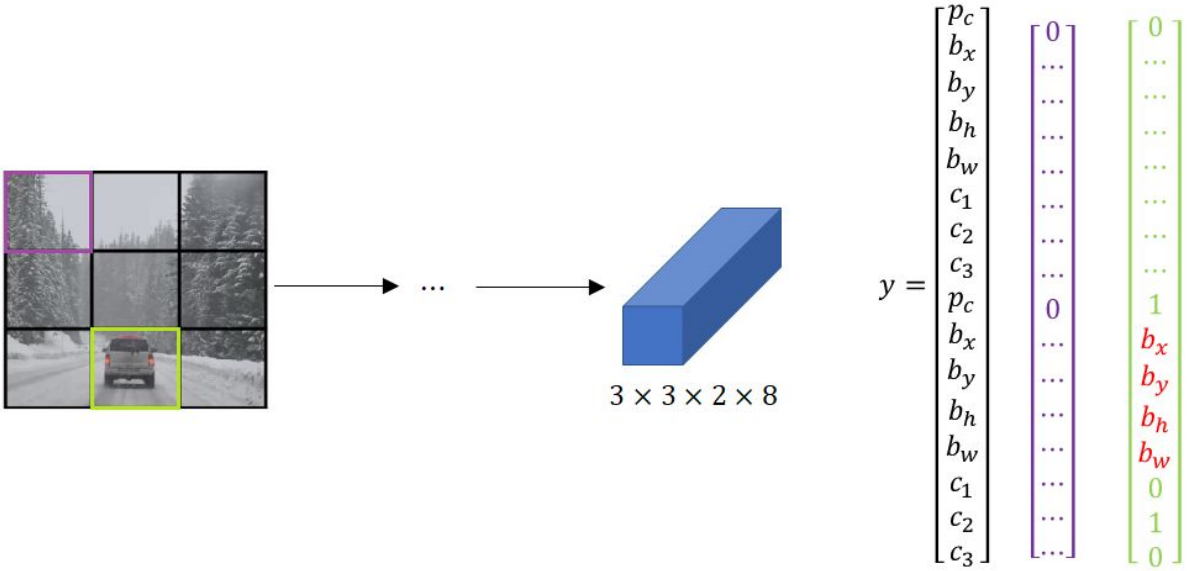
$y$ is $3 \times 3 \times 2 \times 8$

3x3 -> grid size
2 -> number of Anchors
8 -> number of output values in each anchor

As the training process begins, we analyse each one of the grid cells individually. The analysis of the 1st and the 8th cell, for example, is going to be:




This is how the YOLO algorithm makes predictions:




Finally, we need to run the outputs through non-max suppression. We'll use a new image as an example:

For each grid cell, get 2 predicted bounding boxes:

Then, get rid of low probability predictions:

For each class (pedestrian, car, motorcycle), use non-max suppression to generate final predictions:

With all these steps, your algorithm will probably detect all the cars, pedestrians and motorcycles in the image!

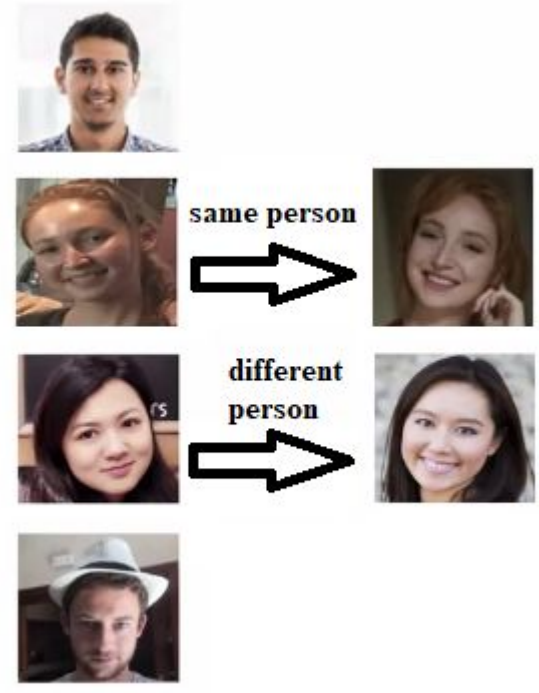
## Face Recognition

### What is face recognition?

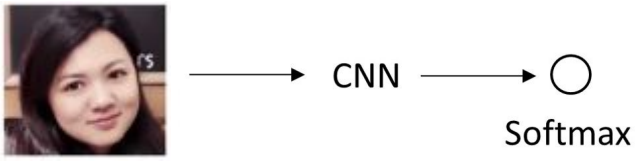
Let’s start telling the difference between face verification and facial recognition.
-Face verification: analysis of an input image or name/ID and output whether the input image is that of the claimed person;
-Face recognition: Has a database of K persons, you get an input image and then output an ID if the image is any of the K persons (or “not recognized”).

### 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, and historically deep learning algorithms don’t work well if you have only one training example. Let’s see an example and learn about how to address this problem.




So this system with only one image, has to determine whether this person is in the database. One approach we could try is to input the image of the person, feed it to a convnet, and have it output a label by using a softmax unit.




However, this approach doesn't really work well, because with such a small training set, the neural network can't learn properly and will perform badly. Instead, we use a "similarity" function. Firstly we compute the degree of difference between images. If this degree is lower than some defined threshold, we assume that the person in the input image is the same person saved on the dataset. If the difference degree is bigger than the threshold, we assume that they are different persons.

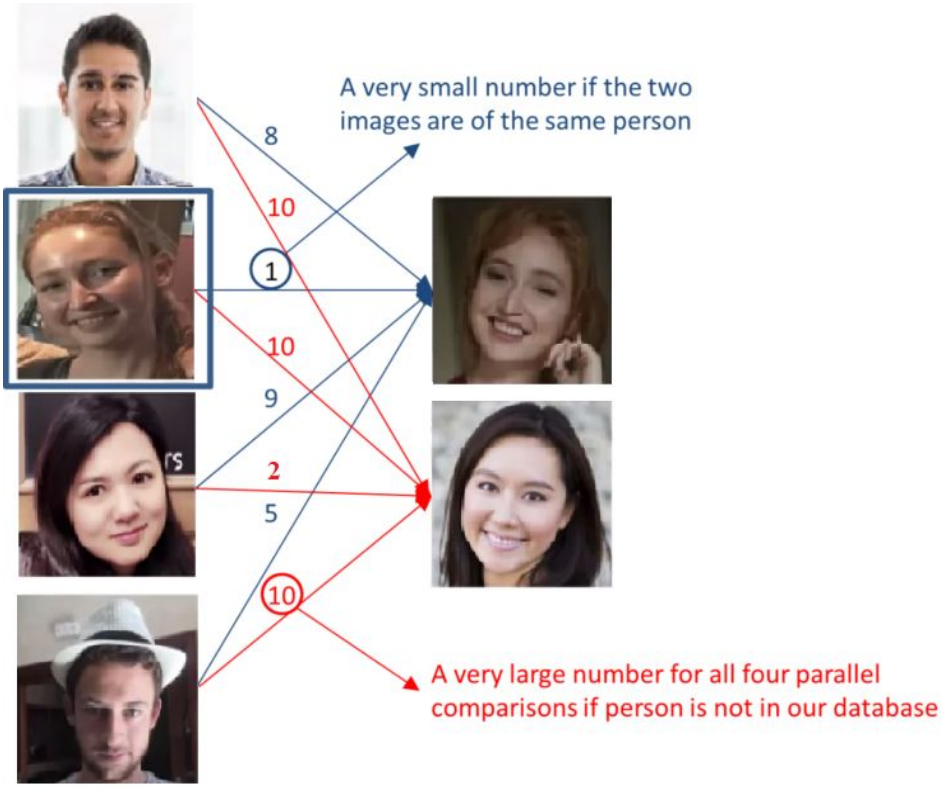
$$d(\text{img1}, \text{img2}) = \text{degree of difference between images}$$

$$\text{If } d(\text{img1}, \text{img2}) \leq \tau \quad \text{"same"}$$

$$> \tau \quad \text{"different"}$$

→ Verification

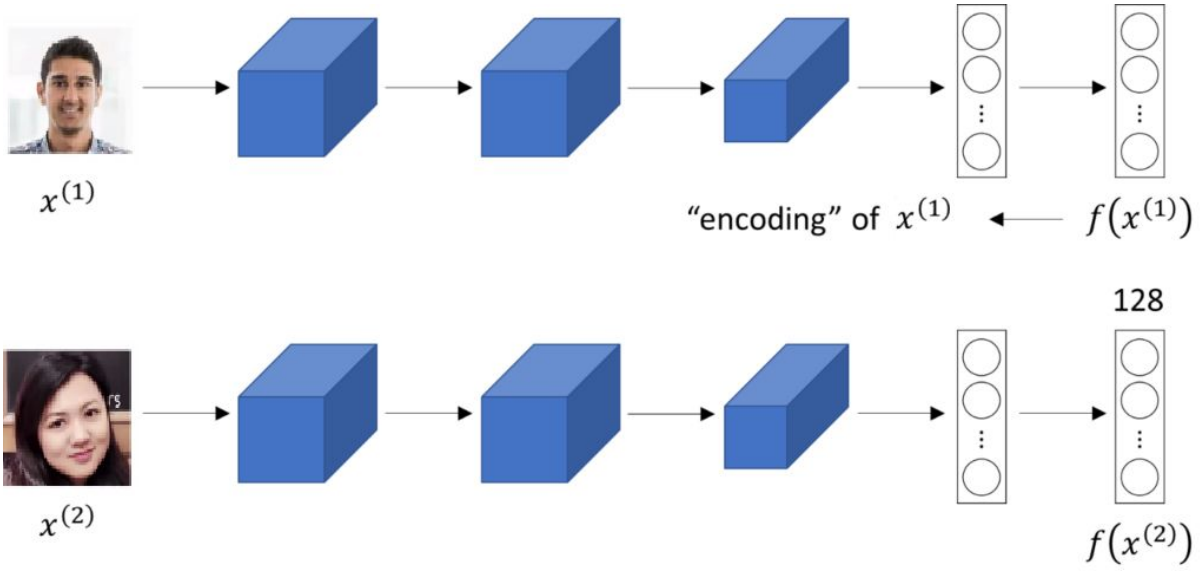
One example of how this similarity function works can be seen in the image below:




One great advantage of a d function is that a new person may join the database easily, without the need to retrain the neural network. You'll see how to implement this function in the next section.

### Siamese Network

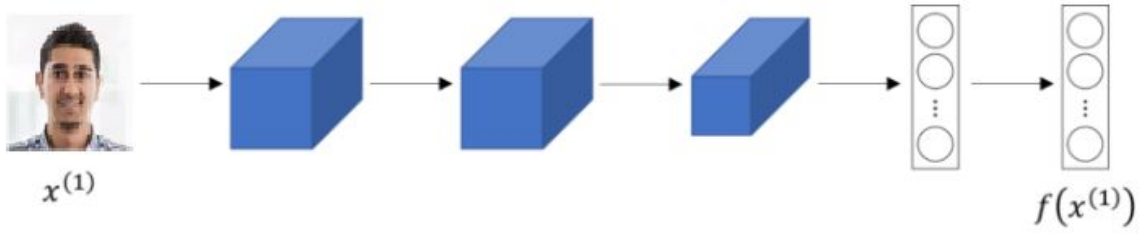
A good way to tell how similar two faces are, is to use the function d, saw in the last section. A good way to implement this function, is to use a siamese network. We get used to seeing pictures of convnets, like these two networks in the picture below. We have two input images, denoted with $x^{(1)}$ and $x^{(2)}$, and through a sequence of convolutional, pooling and fully connected layers we end up with feature vectors.



In this case, we are not going to use a final output layer, using a softmax classification for example. We are going to focus on the vector of 128 numbers computed by some fully connected layer, and call it $f(x^{(1)})$ and $f(x^{(2)})$. To find out the difference between these two images, we can find the distance "d" between $x^{(1)}$ and $x^{(2)}$. A common way is to use a norm of the difference between the encoding of these two images.

$$d\left(x^{(1)}, x^{(2)}\right) = \left\| f\left(x^{(1)}\right) - f\left(x^{(2)}\right) \right\|_2^2$$

The idea of running two identical convolutional neural networks on two different inputs and then comparing them is called a siamese neural network architecture.
To train this siamese neural network, we need to keep in mind that both neural networks have the same parameters, then we need to compute the value of "d" and finally tell when two pictures are of the same person.
Putting it more formally, the parameters of the neural network define an encoding $f(x^{(i)})$. So, given any input image $x^{(1)}$ the neural network outputs a 128 dimensional encoding $f(x^{(1)})$. What we want to do is to learn parameters so that if two pictures $x^{(i)}$ and $x^{(j)}$ are of the same person, then the distance between their encodings should be small.




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

Learn parameters so that:

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

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

We need now to learn the parameters of the NN. To do this, we'll use the Triplet Loss technique, that you'll see in the next section.

## 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. To apply the triplet loss, you need to compare pairs of images.

To learn the parameters of the neural network, you have to look at several pictures at the same time. For example, given these two pairs of pictures:


Anchor


Positive


Anchor


Negative

You want that the encondings of the first pair of images to be similar, because these are the same person, and the encodings of the second pair of images to be quite different, because these are from different persons.
In the terminology of the triplet loss, we always look at one anchor image and then we want the distance between the anchor (A) and a positive image (P) to be very small. Whereas, we want the anchor when pairs are compared to the negative images (N) for their distances to be much further apart. So this gives rise to the term triplet loss, which is that you'll always be looking at three images at a time.
To formalize, we want the parameters of our neural network to have the following property:

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

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

Obs: you can think of "d" as a distance function.
Making a slight change to this expression:

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

However, if $f$ always outputs 0, these two norms are $0 - 0 = 0$, and we can trivially satisfy this equation. We need to make sure that the neural network doesn't just output 0 for all the encoding (that it doesn't set all the encoding equal to each other). To prevent our network from doing that, we are going to modify the equation so that it doesn't need to be just less than or equal to 0, it needs to be a bit smaller than 0. In particular we say this needs to be less than $-\alpha$, where $\alpha$ is another hyperparameter (it is also called a margin).

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

This prevents the neural network from outputting the trivial solutions. By convention, we write $+\alpha$ on the left side of the equation.

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

Formalizing the triplet loss function, considering three images A,P and N:

$$L(A,P,N) = max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha\ ,0)$$

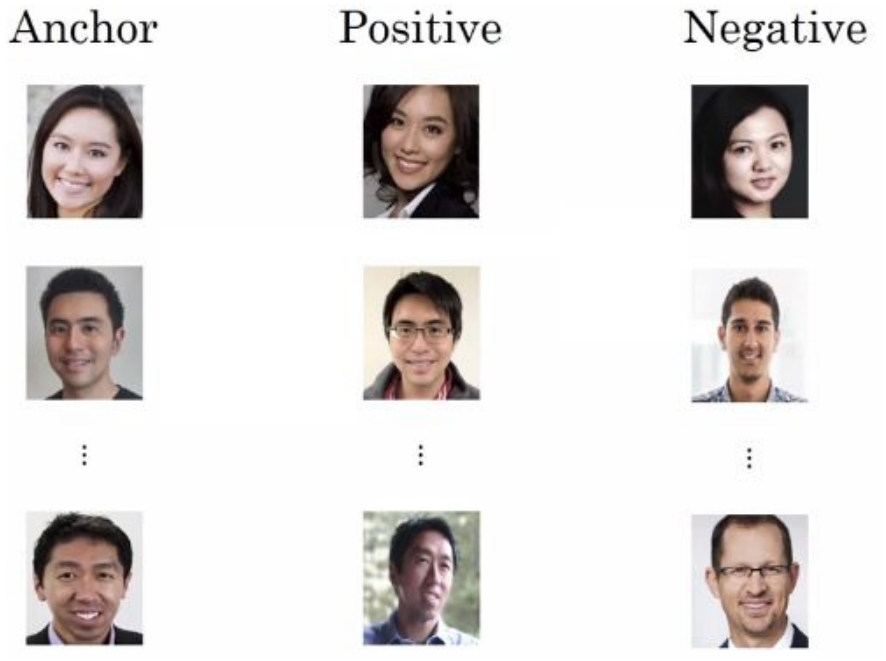
As long as we achieve the goal of making $\|f(A)\ -\ f(P)\|^2 - \|f(A)\ -\ f(N)\|^2 + \alpha\ \leq\ 0$, then the loss is equal to zero. On the other hand, if this is greater than zero we take the max so we get a positive loss. As we want to minimize the loss, the neural network will try to push down the value of $\|f(A)\ -\ f(P)\|^2 - \|f(A)\ -\ f(N)\|^2 + \alpha$. This is how we set the lost function of a single triplet. The overall cost function for our neural network can be a sum over a training set of these individual losses:

$$J = \sum_{i=1}^{M} h\left(A^{(i)}, P^{(i)}, N^{(i)}\right)$$

We need to make sure that our training set has some pairs of different images of the same person.

During training, if A, P and N are chosen randomly, the condition $d(A,\ P)\ +\ \alpha\ \leq\ d(A,\ N)$ is easily satisfied. So we need to choose triplets that are "hard" to train on. To make sure the triplets will be good for training, we need to choose A, P and N so that $d(A,\ P)\ \approx\ d(A,\ N)$. In this case, the learning algorithm has to try extra hard to take $d(A,\ N)$ and push it up, or take $d(A,\ P)$ and push it down, to make sure they are further from each other (making the gradient descent work more, improving the efficiency of the model).

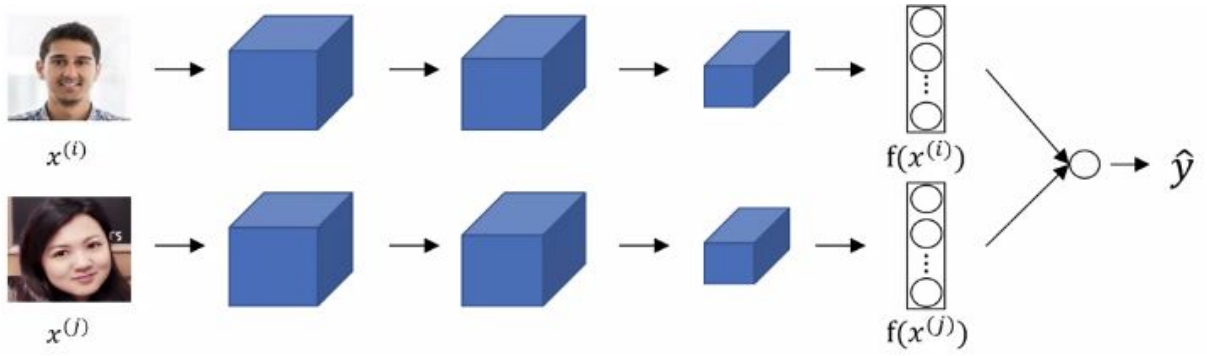
Here is an example of a training set:




Today's face recognition systems, especially the large scale commercial face recognition systems are trained on very large data sets, with more than a million images sometimes. So this is one domain where often it might be useful to download someone else's pretrained model, rather than doing everything from scratch.

## Face Verification and Binary Classification

There's another good way of learning parameters of a convnet for face recognition. This new way combines the siamese neural network with a binary classification problem. The idea is to use the siamese network and instead of calculating the cost function with the fully connected layers (like the triplet loss function), feed these fully connected layers to a logistic regression unit, then make a prediction. You can see how it works in the image below:




The output $\hat{y}$ will be a sigmoid function, applied to some set of features but rather than just feeding in these encodings (the fully connected layers), we'll take the difference between each of them. Let's assume that each fully connected layer is a 128 dimensional vector. The output $\hat{y}$ is going to be:

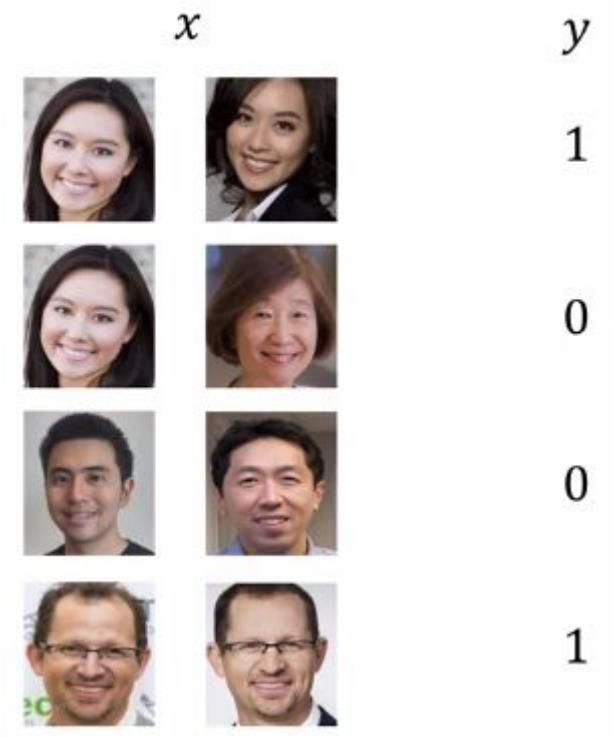
$$y_{hat} = \sigma\left(\sum_{k=1}^{128} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b\right)$$

There are other variants to this output equation. If you want, instead of using $|f(x^{(i)})_k - f(x^{(j)})_k|$, you could use:

$$\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

This formula is sometimes called the $\chi^2$ formula. Another variation can be seen in the paper: "Taigman et. al., 2014. DeepFace closing the gap to human level performance".

To train the neural network, the training dataset needs to be like this:

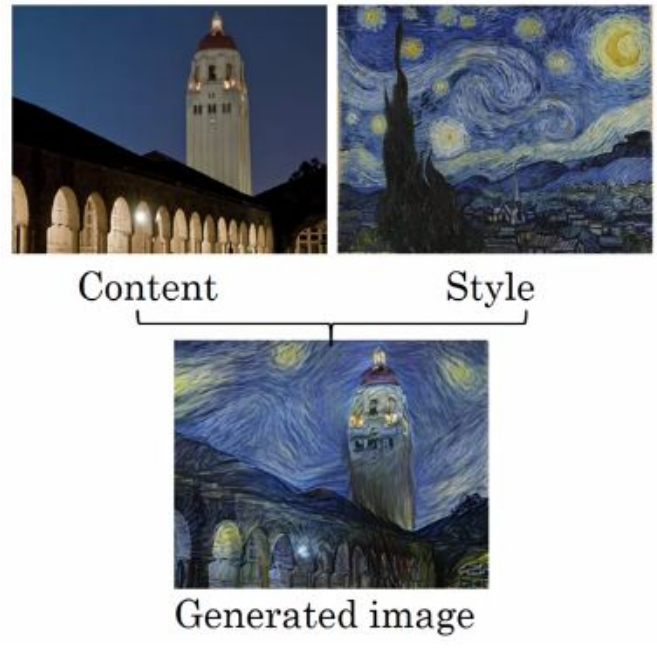



Where y=0 means that the input pair is composed by different persons, and y=1 when the input pair is from the same person.

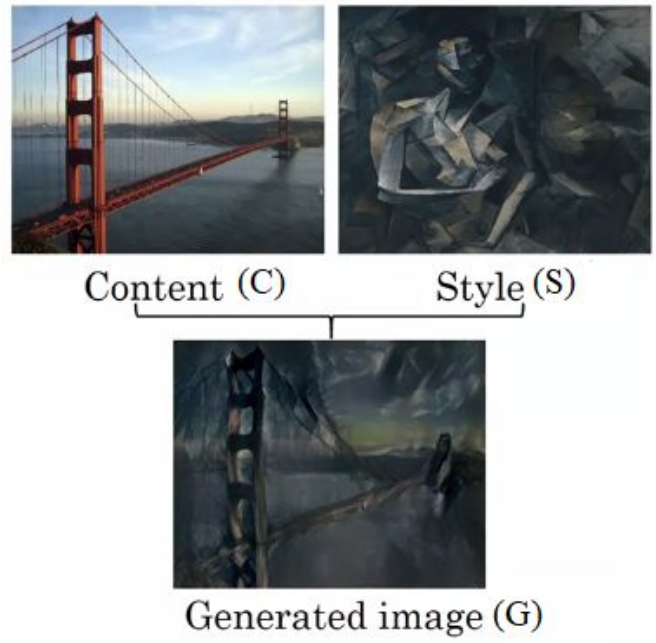
## Neural Style Transfer

### What is neural style transfer?

Neural style transfer is a very fun application of convnets. It allows you to implement and generate your own artwork. Let's see an example using Van Gogh's style:

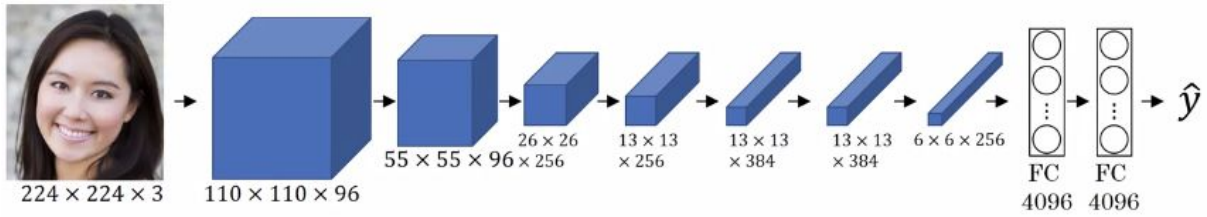


In order to describe how you can implement neural style transfer, let's use C to denote the content image, S to denote the style image and G to denote the generated image. Here is another example, using a Picasso's style:




### What are deep ConvNets really learning?

Understanding what the deep convolutional networks are actually learning helps us think through how to implement neural style transfer (and of course, learn more about the convnets). Let's use an example of a convnet:



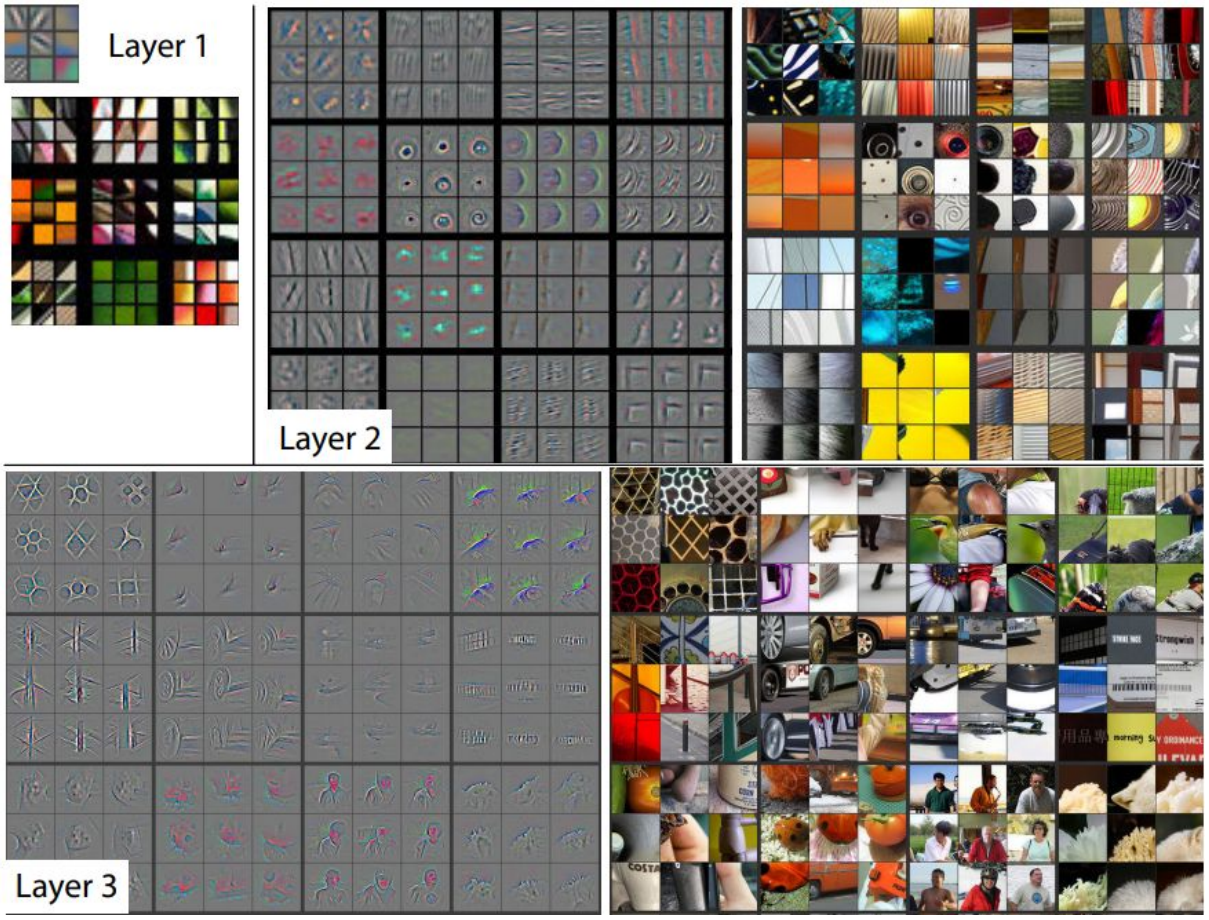

First, let's pick a unit in layer 1 and find the nine image patches that maximize the unit's activation.

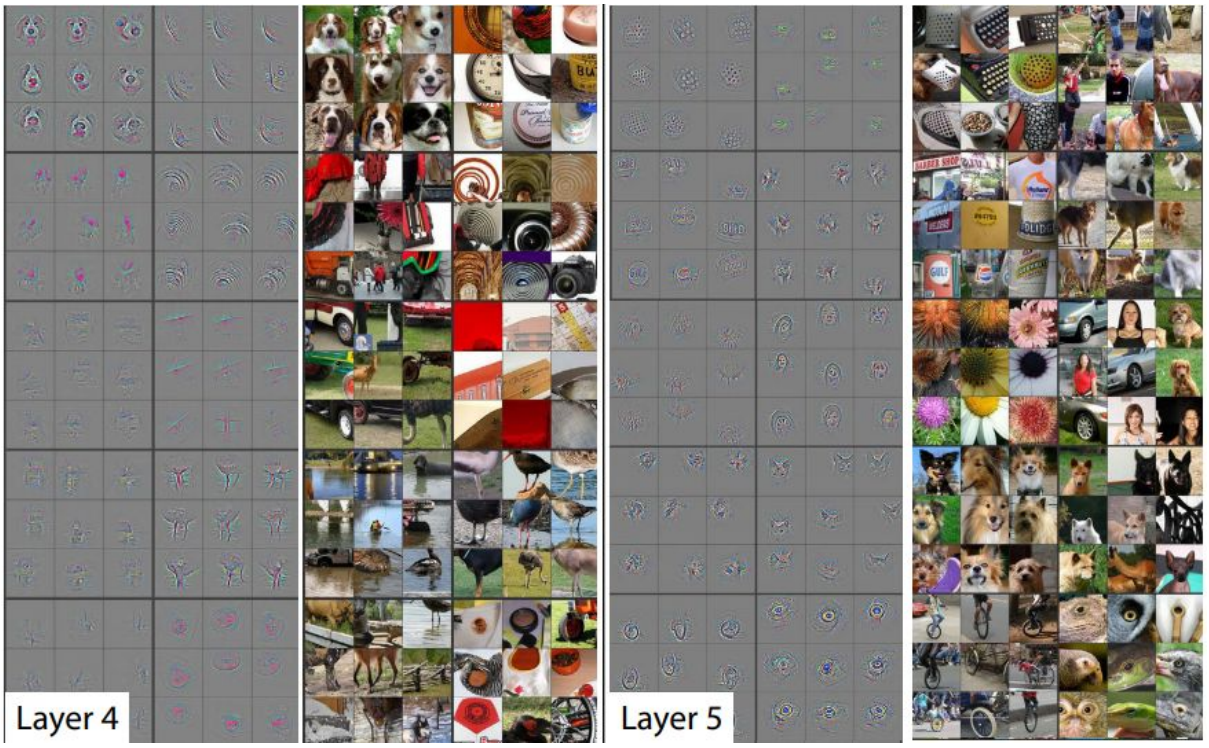
It looks like this unit is taking care of diagonals or edges. Then, we'll repeat the process for other units.

It looks like this unit is looking for images with a line similar to the line on the right square.

This one is probably looking for vertical edges, but with a preference that the left side of it to be green.
And so on…

This is a visualization of what the units in the first layer are recognizing. Let's take a look at all the layers together:






The left side of the images represents what the convnet is really seeing and the right sides are the real images. As you can see, as we go deeper in the convnet, the layers detect more complex shapes and objects in the image.

## Cost Function

In this section, we'll take a look at the neural style transfer cost function. Our cost function will be composed of two other cost functions: the content cost function and the style cost function.

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

The content cost function uses a generated image (G) as the input, then calculates the cost comparing how similar the generated image is with the content image (C).
The style cost function uses a generated image (G) as the input, then calculates the cost comparing how similar the style of the generated image is with the style of the style image (S).
The $\alpha$ and $\beta$ constants are hyperparameters that specify the relative weighting between the content costs and the style cost.
More details about the cost function can be found at the paper: "Gatys et al., 2015. A neural algorithm of artistic style".
Having defined the cost function J(G), in order to generate a new image what you do is the following:

1. Initiate G randomly
   For example: G: 100x100x3

2. Use gradient descent to minimize J(G)
   $G := G - \frac{d}{dG}J(G)$
   This step will actually update the pixels values of the image G.

Imagine you want to apply a *Guernica* style on the following image for example:


The steps taken by the neural network would be:

The first image is a randomly generated G image and using the cost function+gradient descent the image would slowly approach the content image with the desired style.

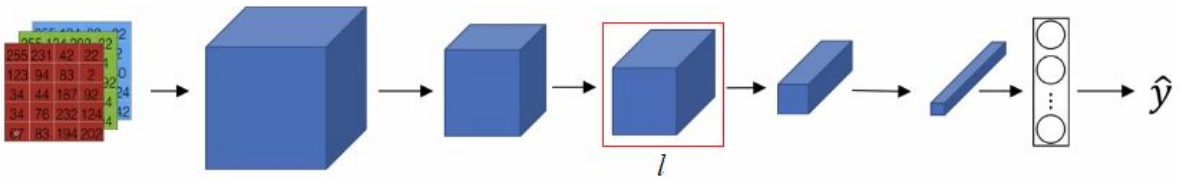
## Content Cost Function

The cost function of the neural style transfer algorithm had a content cost component and a style cost component. Let's start defining the content cost component.
We'll choose a hidden layer $l$ to compute content cost. The chosen hidden layer needs to be somewhere in between the neural network, that is not too deep, nor too shallow. Then, we need to use a pre-trained ConvNet (E.g., VGG network) and measure how similar the generated image and the content image really are.
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. To evaluate this similarity, we use the content cost function:

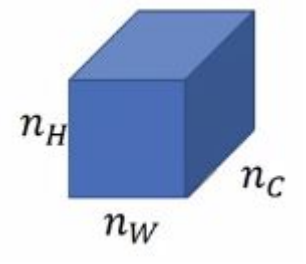
$$J_{content}(C,G) = ||a^{[l](C)} - a^{[l](G)}||^2$$

## Style Cost Function

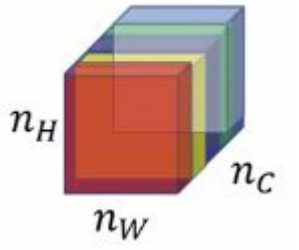
In this section, we'll define the style cost function. Let's use again some layer $l$'s activation to measure "style". We can define style as correlation between activations across different channels.



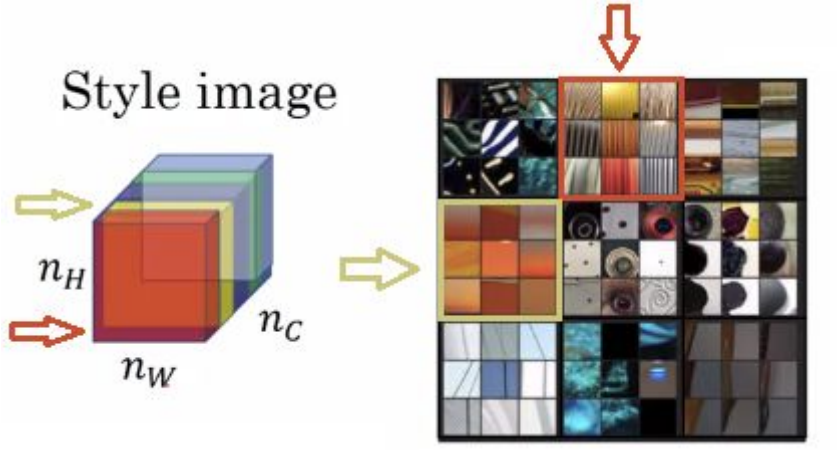

Let's take a look at the layer $l$ block of activations.




We'll analyse now how correlated are the activations across different channels. Let's shade the different channels by different colors.




In this case, the layer $l$ has 5 different channels. Let's take two channels and look at the two neurons that these channels take care off:




These two channels are highly correlated if for whatever part of the image that has this type of vertical texture (in the red channel), that part will probably have an orange-ish tint (represented by the yellow channel). If for every vertical texture the parts don't have an orange-ish tint, these channels are uncorrelated.
The correlation tells you which textures/colors/shapes tend to occur or not occur together in some part of an image. The degree of correlation gives you one way of measuring how often these different high level features occur together (or not) in different parts of the image.
Then we need to measure the degree of correlation between the generated image's style and the input image's style. To compute this, we need to create a style matrix that measures all the correlations we've learned about.
Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ .
i = height
j = width

k = # channels
$G^{[l]}$ = style matrix

The formula of $G^{[l]}$ should be, for the style matrix from the style input (S) and for the style matrix from the generated output (G):

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

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

So all this formula is doing is summing over the different positions of the image over the height and width and just multiplying the activations together of the channels k and k' and that's the definition of $G^{[l]}_{kk'}$.
Obs: in linear algebra, the style matrix is called "Gram matrix".
The cost function will be:

$$J^{[l]}_{Style}(S, G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \|G^{[l](S)} - G^{[l](G)}\|^2_F$$

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

If you want to compute the cost of multiple layers, the equation is the following:

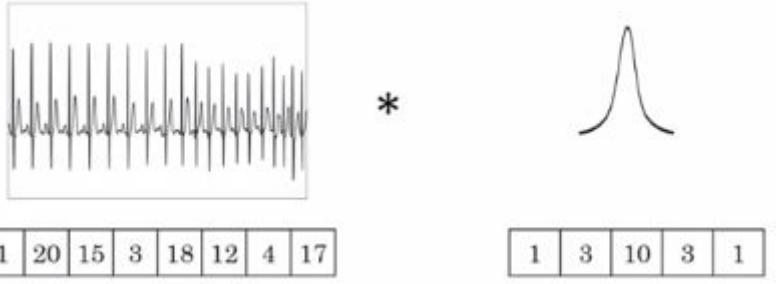
$$J_{Style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{Style}(S, G)$$

where $\lambda^{[l]}$ is a set of additional hyperparameters to weight the individuals costs of each layer. With both cost functions defined, we can now use the overall cost function of the neural style transfer (using the content loss and the style loss):

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

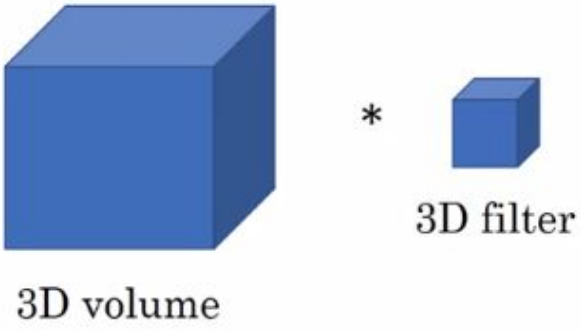
## 1D and 3D Generalizations

Most of the discussions we've had in the previous sections can be used to generalize the convolution operation to 1D and 3D data. Let's take a look at an example of an 1D convolution:



This image on the left shows the measures of an electrocardiogram, representing a time series with the voltage at each instant of time. The image on the right shows a 1 dimensional filter. To apply the convolution operation, we just need to do a similar process used for the 2D convolution operation:


Convolving a 14 dimensional 1D input with a 5 dimensional filter, results in a 10 dimensional output (if we have more than 1 filter, the output would be 10x # filters). The result is similar to the 2D convolution operation - if we convolve a 14x14 2D input image with a 5x5 filter, the output would be 10x10 (if we have more than 1 filter, the output would be 10x10x # filters).
Let's do an example of a 3D convolution:




Considering the input has a dimension of 14x14x14 and the filter has a dimension of 5x5x5, then this would be a 10x10x10 dimension volume (if we have more than 1 channel, the operation would be: 14x14x14x $n_c$ * 5x5x5x $n_c$ = 10x10x10x#filters).