Course 4: Convolutional Neural Networks

Content

1. Foundations of Convolutional Neural Networks

1.1. Computer Vision

Throughout this course we will be dealing with the following **computer vision** problems:

- Image classification
- Object detection
- Neural Style Transfer

One problem of dealing with images is that if we get high resolution images, these can have a lot of features (i.e.millions). This means that if we use a dense network we will be getting a really high number of weights, which make it difficult not to overfit.

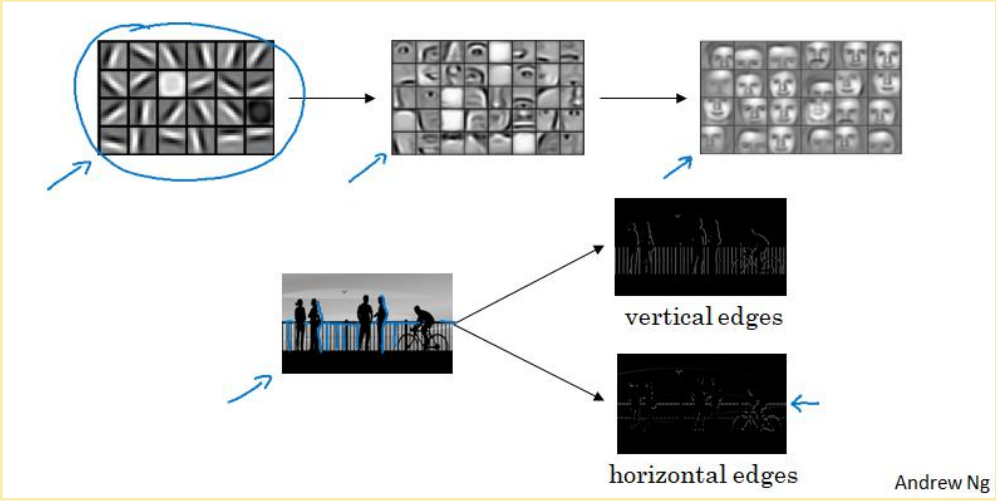
So as to be able to work with such high-res images, we will need the **convolution** operation, which is one of the fundamental building blocks of CNN.

1.2. Convolution operation

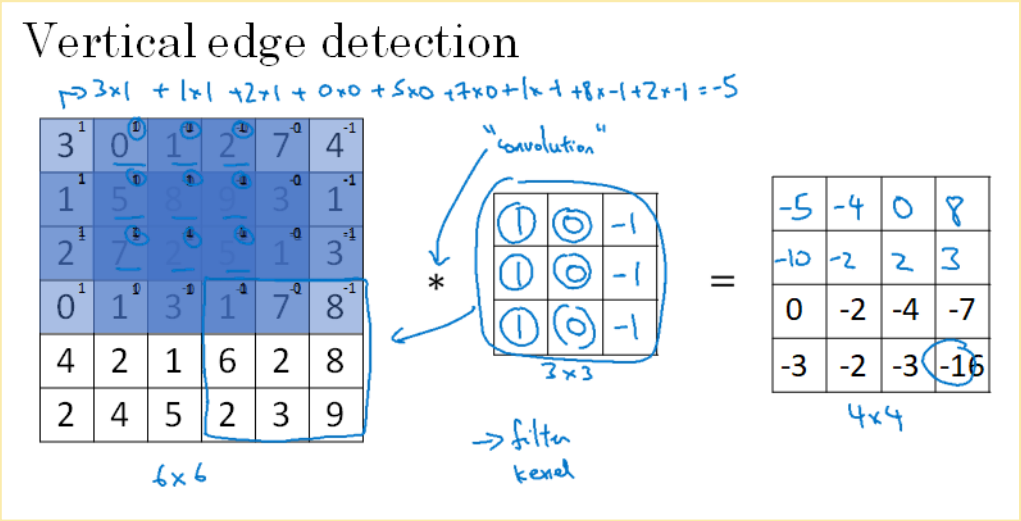
We'll cover the convolution operation through examples:

1.2.1. Edge detection

If we have an image like this, we could detect horizontal or vertical edges:

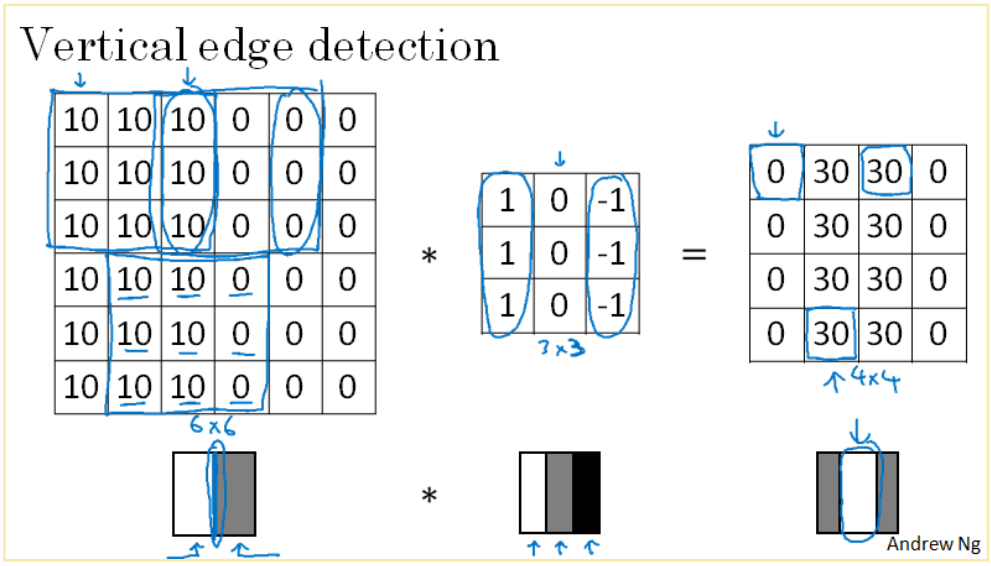


How can we detect them?

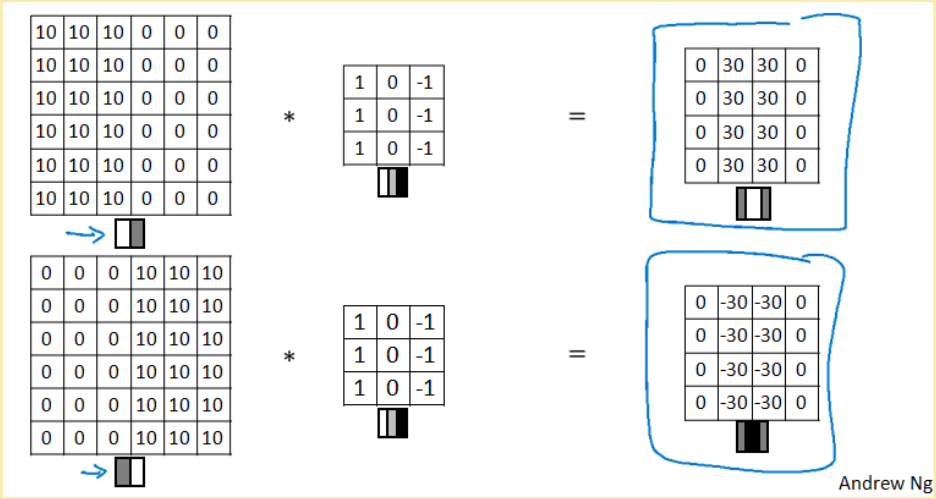


If we have an image represented by the 6x6 matrix on the left, we'll apply a 3x3 filter (or kernel) matrix to it. This is, we'll convolve the left matrix with the filter, and we'll get a 4x4 matrix in which each element is the product of two matrices: the corresponding one on the left matrix and the filter matrix (see image above).

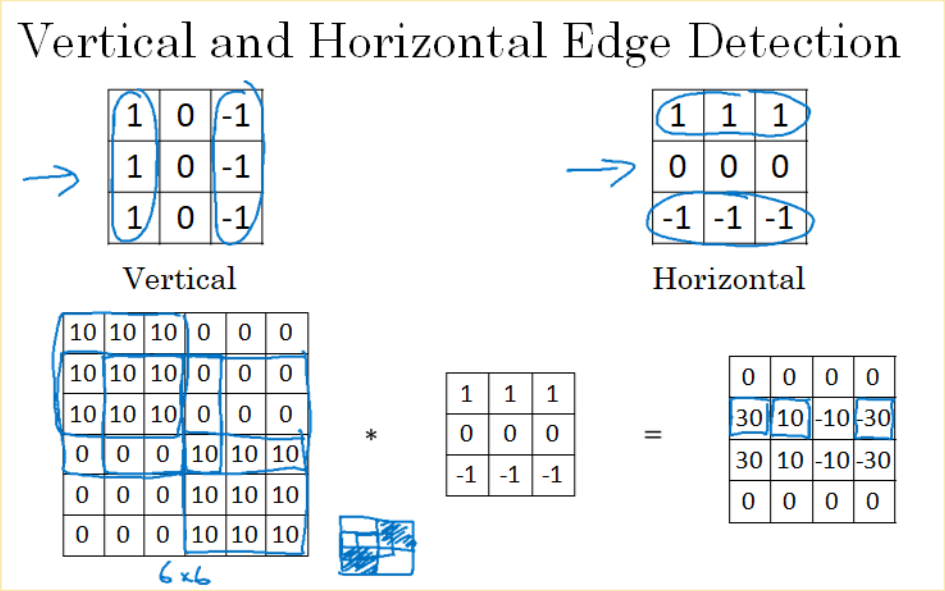
Although these are only matrices, we could think of the left one as an image, the middle one as a filter and the right one as another image. And it turns out that this specific filter turns out to be a vertical edge detector. We'll see how with a simplified example:



In addition, this is useful to differentiate between light-to-dark edges or dark-to-light ones:

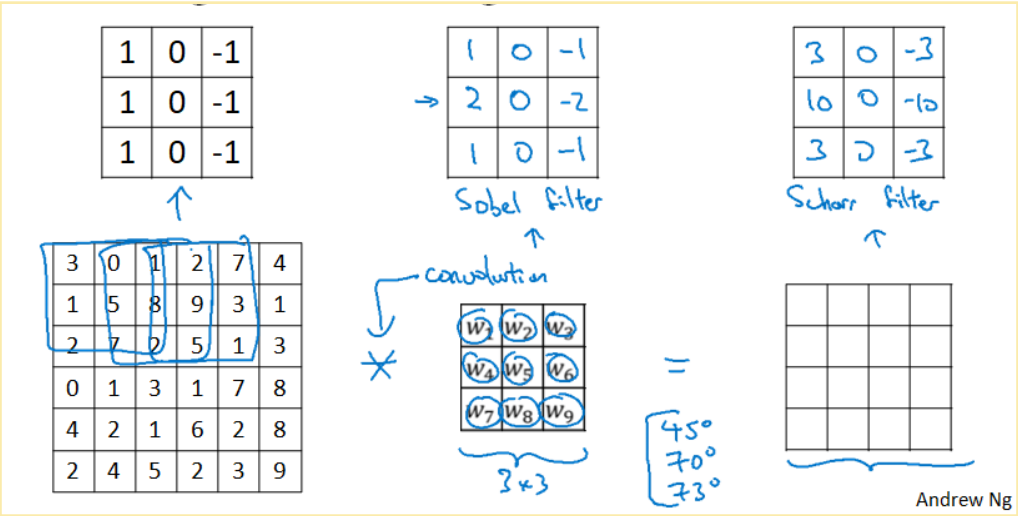


We can also detect horizontal edges:



There can be many possible filters we can use (for example, the Sobel or Scharr filters). But it turns out that we don't have to decide which numbers appear in the filter. In fact, if we treat these numbers as parameters and learn them in the training process, we will have better edge detectors for our data than any of these hand coded filters, which will also be able to learn to detect, for example, 45 degree edges.

So, by doing this, neural networks can learn low level features such as edges more robustly than computer vision researchers that code up things by hand.



1.2.2. Padding

In order to build CNN one modification to the basic convolution operation that we'll need to use is **padding**.

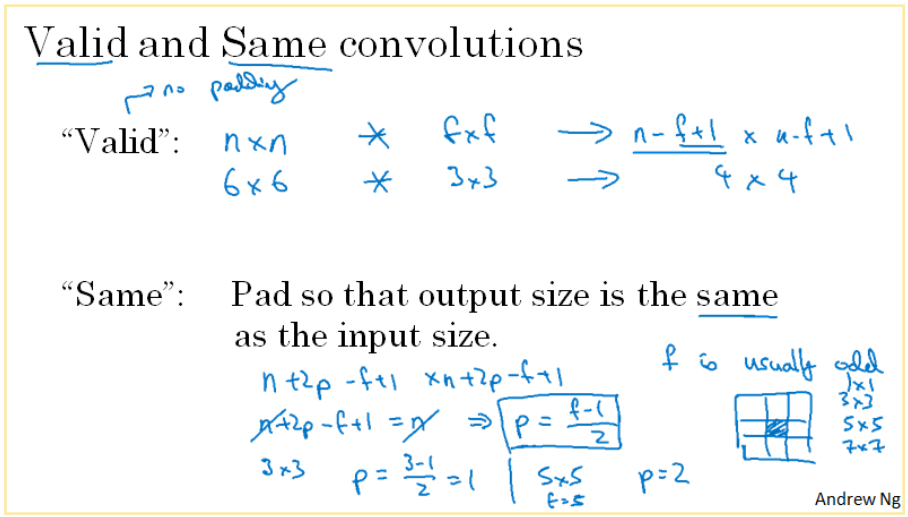
Doing the convolution as we have done has two downsides:

- We have seen that if we convolve a 6x6 matrix with a 3x3 filter, we get a 4x4 matrix. This means the image shrinks when convolving it, so perhaps if we do several convolution operations we can get a really small image.
- The corner pixels are used only once in the calculations, whereas the central ones are used more. This means the output may have "less information" of some pixels than others.

We can solve this two downsides with padding: we'll add a border to the image. We need to determine how thick is this border (one, two, three rows or pixels) and which values they take.

In terms of how much to pad, there are two common choices:

- "Valid" convolutions: no padding.
- "Same" convolutions: the output size is the same as the input size.

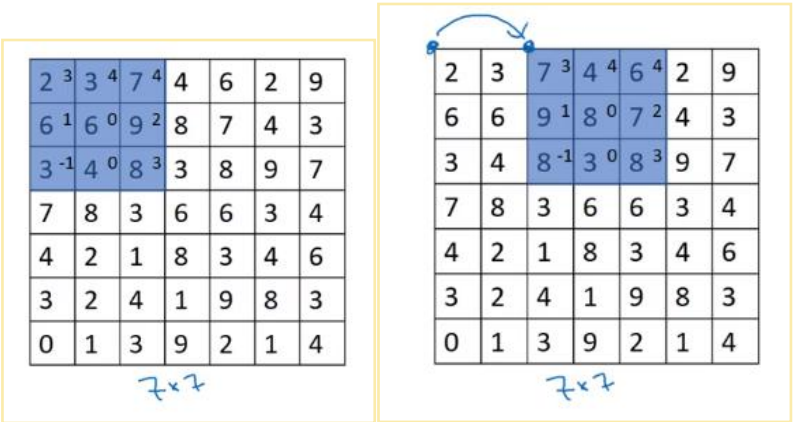


With the squared formula, we can obtain the *p* parameter in terms of the size of the filter *f* (note that *f* is usually odd in computer vision literature).

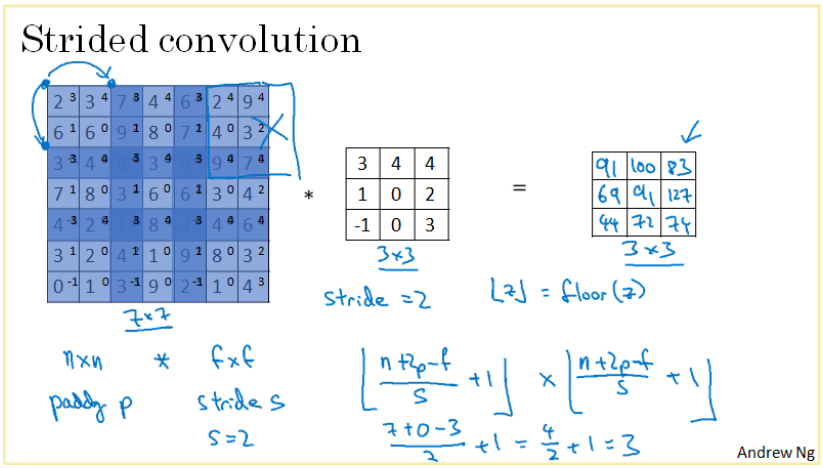
1.2.3. Strided convolutions

Strided convolutions is another piece of the basic building block of convolutions as used in CNN.

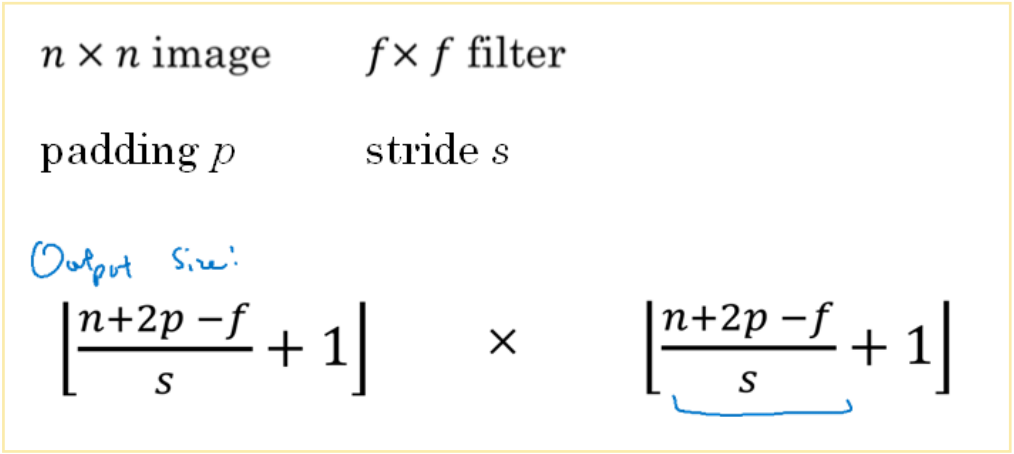
Strided convolutions consists of performing "jumps" when convoluting the input image. If we use (*stride=2*):



The output dimensions depend on the stride:



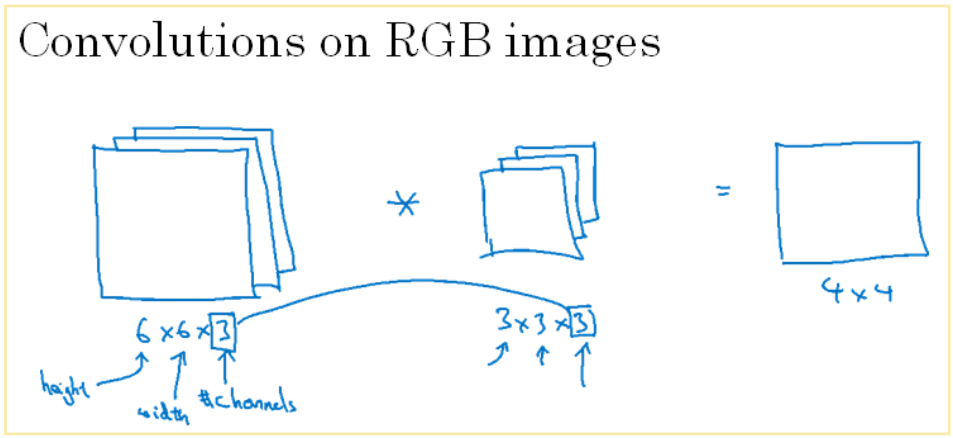
The below slide is a summary of convolutions and dimensions:



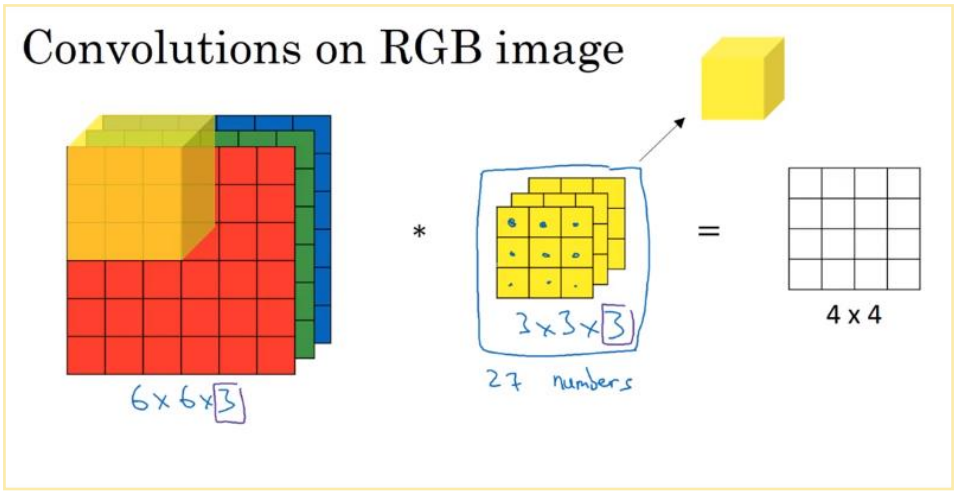
Finally, let's discuss a technical note. If we see convolution in math textbooks, it involves a previous flipping of the filter matrix. We have skipped this step, and doing this is strictly called cross-correlation. So this last thing is what we are actually doing. But in deep learning, it is normally called convolution by convention. That flipping is useful in signal processing applications, but for deep learning it is just not necessary.

1.2.4. Convolution over volume

So far we have applied convolution to 2D matrices. Let's suppose we want to apply convolution to an RGB image:

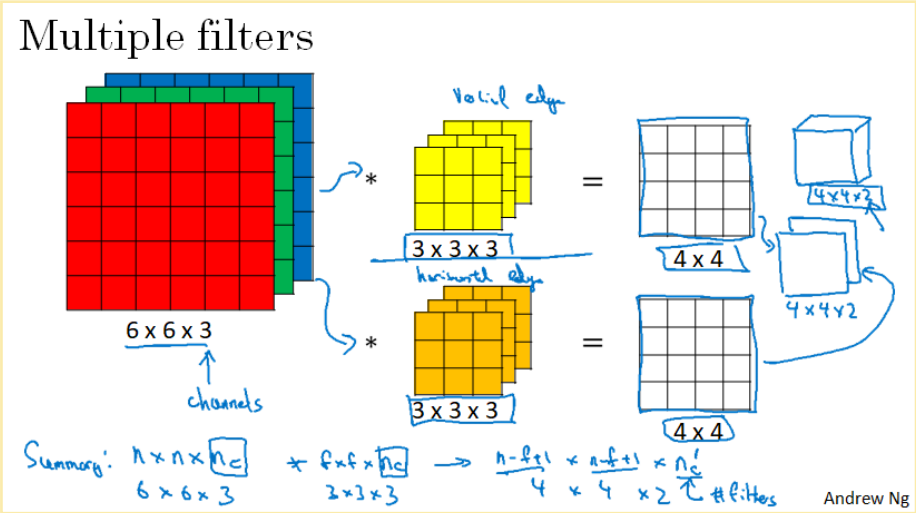


We have the three dimensions in our input image: height, width and channel. We'll use a 3D filter with the same (by convention) channels as the image, and the output will be a 4x4x1 image. Let's see how if works:



It's the same idea but in three dimensions (with cubes). And, just like the 2D case, depending on the values of the filter we will be able to detect different things.

Let's see one last idea that is crucial for CNN. What if we want to detect horizontal edges, or vertical, or 45 degree, or 73 degree at a time? We can use multiple filters:



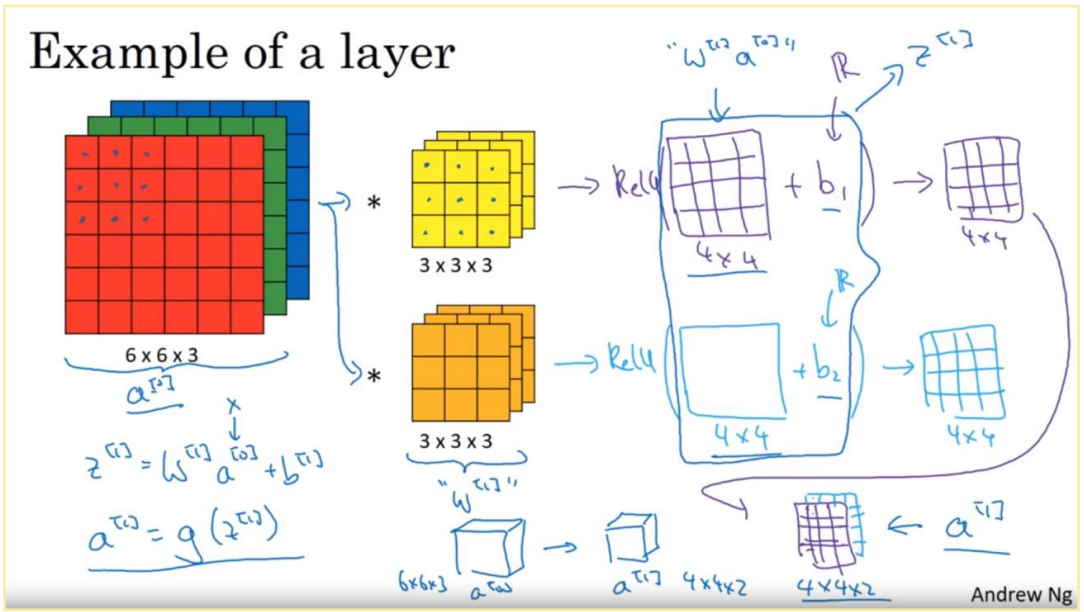
We'll then get multiple output images, which we can stack together. At the bottom of the slide above we can see a summary of the dimensions when we use multiple filters.

This is really powerful because we can operate on RGB images with multiple channels, and detect hundreds of different features. And the output will have a number of channels equal to the number of filters we are using.

Now that we are familiar with convolution, let's see how to implement a layer of a CNN.

1.3. One layer of a Convolutional Network

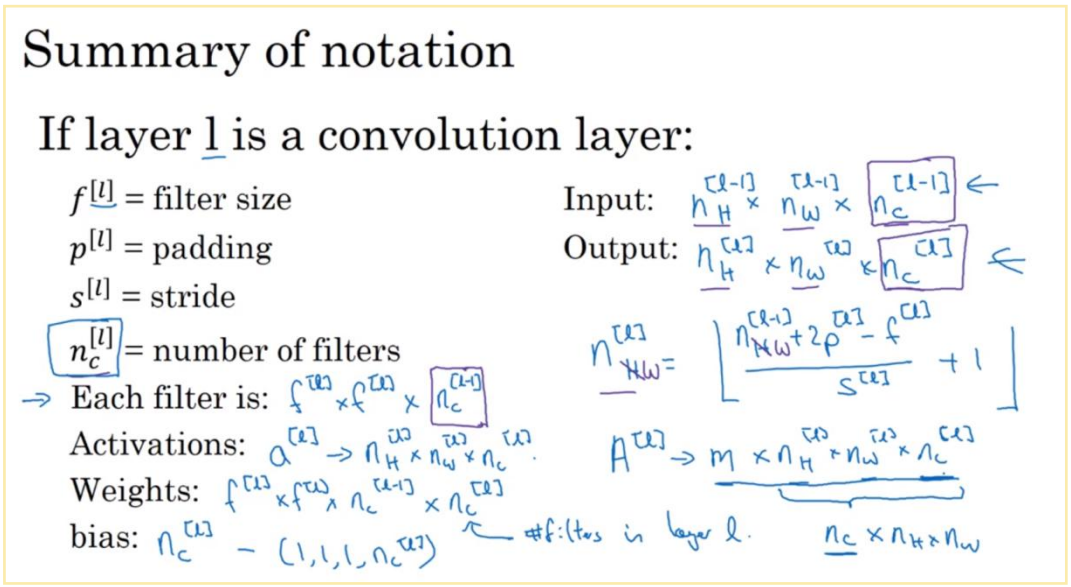
We can see a summary on the slide below:



We get our image, apply our filters to it and get the two 4x4 matrices, add bias to them and then pass them through the non-linear function. Finally we stack them up and we get the output cube. We can see the filter values as the weights of the layer.

Let's calculate how many parameters we have in a single layer: if we have 10 filters that are 3x3x3 in one layer of a NN, this layer will have 280 parameters (27 parameters for each filter plus the bias times 10 filters).

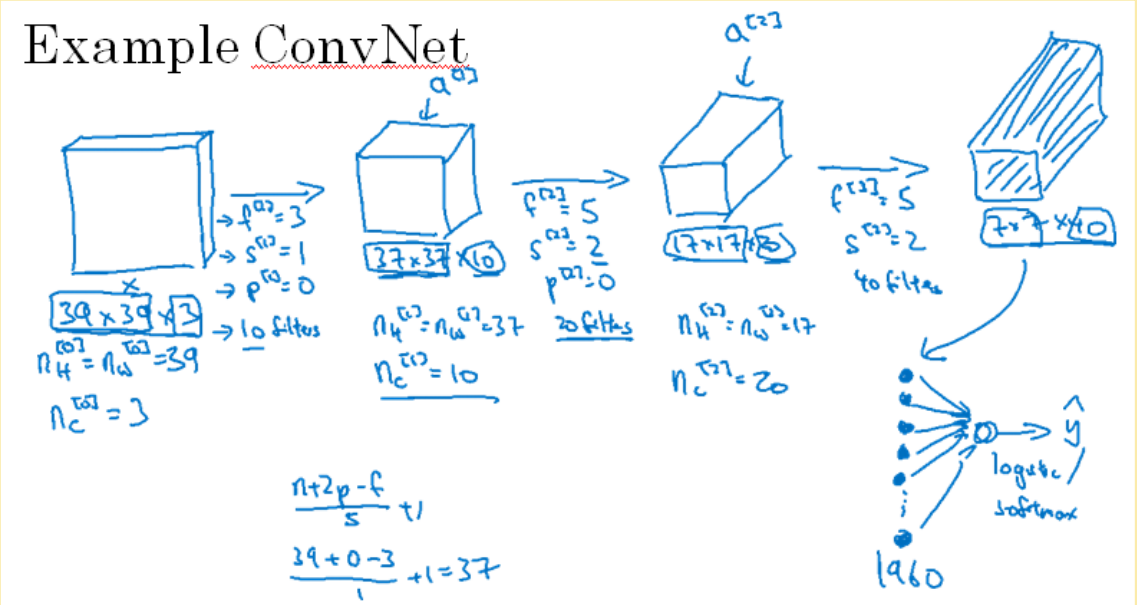
Let's make a summary of the notation on convolution layers:



Let's stack a bunch of these layers together to get to know how a deeper neural network works:

1.4. Simple Convolutional Network example

Let's see a typical example of a CNN:



A lot of the work in designing CNN is selecting the best hyperparameters such as the total size, the stride, the padding, the number of filters… We'll see how to decide later in the course.

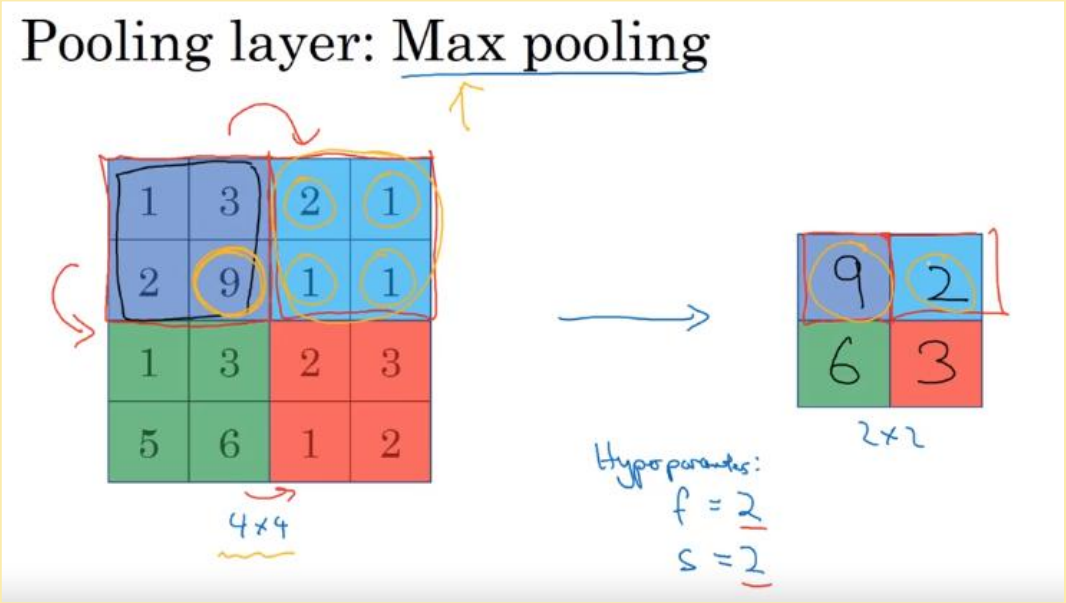
In a typical ConvNet there are three common types of layers:

- Convolution
- Pooling
- Fully connected layers

Let's see what are pooling and fully connected layers.

1.4.1. 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 to make some of the features that detects a bit more robust. Let's take a look at an example, and then we'll discuss why we would be interested in pooling.

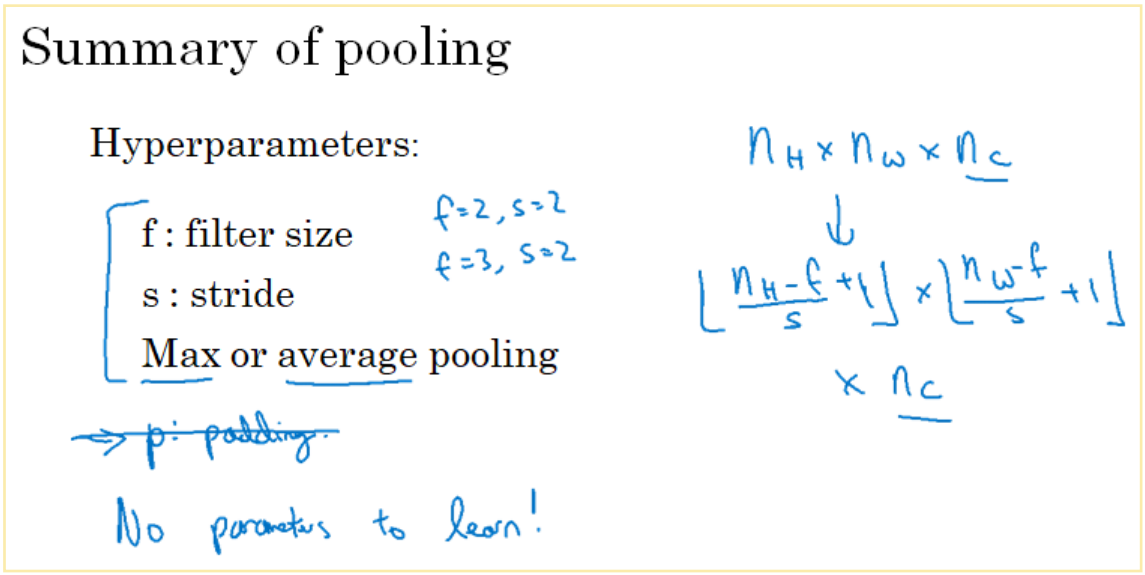


In the case of **max pooling,** it consists of taking the max of each quadrant. This allows to preserve the "important" features and kind of ignore the non-relevant features. This is only an intuition, but it turns out that pooling works really well in deep learning experiments.

Another interesting thing is that a pooling layer requires hyperparameters (such as the filter size or the stride), but it doesn't involve a learning process. There are no weights to be learnt on the gradient descent in these layers.

There is another kind of pooling that is not used very often: **average pooling**. It is only used in very deep nets when we want to collapse our representation.

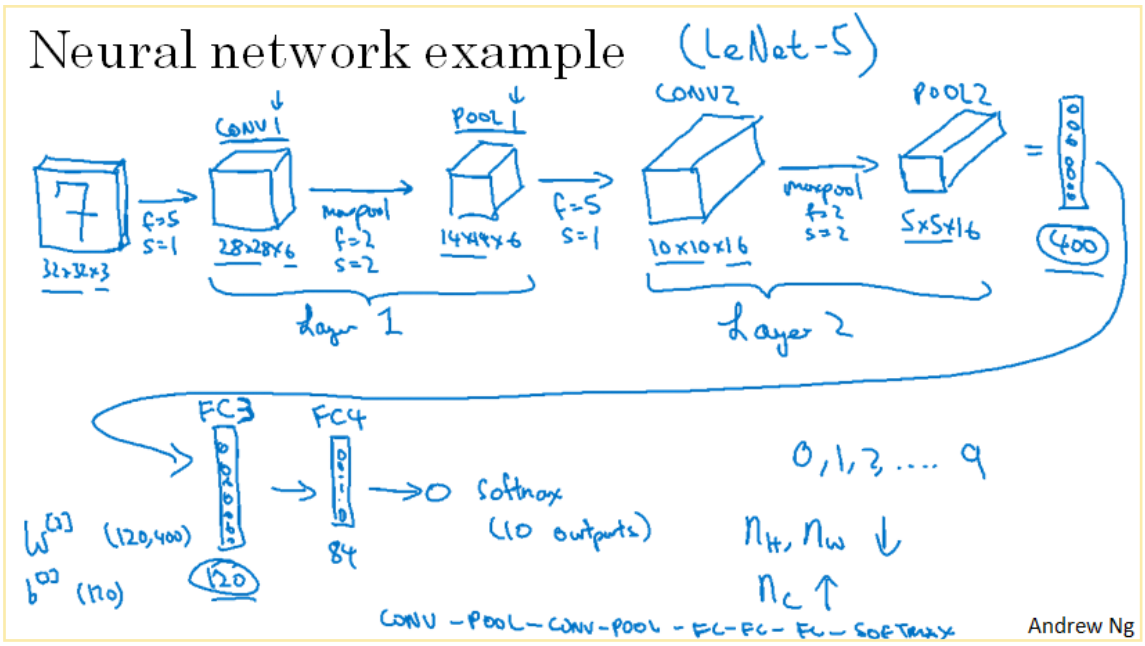
In the next slide we can see a summary of the pooling hyperparameters and some common choices:



We usually don't use padding, although we'll see a particular application on next week's material.

1.4.2. CNN example with conv, pooling and fully connected layers

Let's see a CNN example:



Once we have used conv and pool layers, we can stack the values to a unidimensional vector and then use classic dense layers. These are the fully connected layers.

1.5. Why convolutions?

There are two main advantages of using convolution layers instead of just fully connected layers, which make it possible to have a lot less parameters:

- Parameter sharing: a feature detector that's useful in one part of the image is probably useful in another part of the image. This reduces the number of weights.
- Sparsity of connections: in each layer, each output value depends only on a small number of inputs (instead of depending on the whole image, it depends on the area/volume where the filter is being applied).

2. Deep Convolutional Models: Case studies

2.1. Why look at case studies?

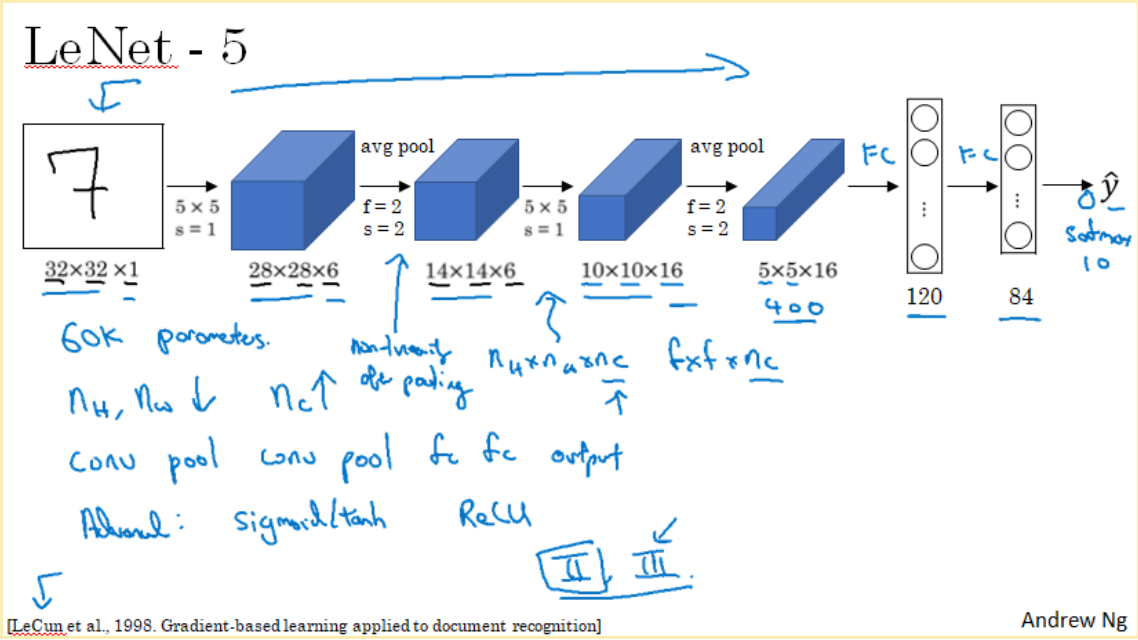
There can be a lot of combinations when creating our ConvNet. In addition, an already coded neural net used for one task may be useful for another task.

We'll see some classic neural networks such as LeNet-5, AlexNet or VGG. We'll then turn into the ResNet and finally we'll cover the Inception neural network.

2.2. Classic Networks

2.2.1. LeNet-5

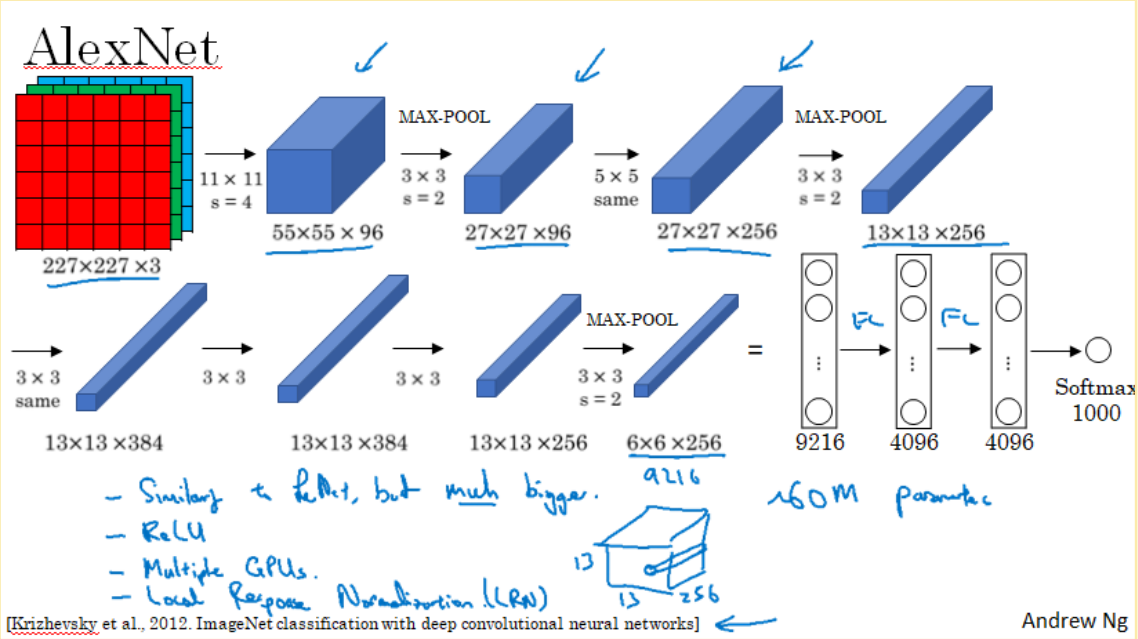
The goal of LeNet-5 was to recognize handwritten digits. It was trained on grayscale images.



It is a small network. It has 60k parameters. This pattern of convolution layer, then pool, then conv, then pool, and then some fully connected layers is quite common. In the original paper (1998) there are also some ideas that were discussed because the computers were much slower then and don't apply nowadays.

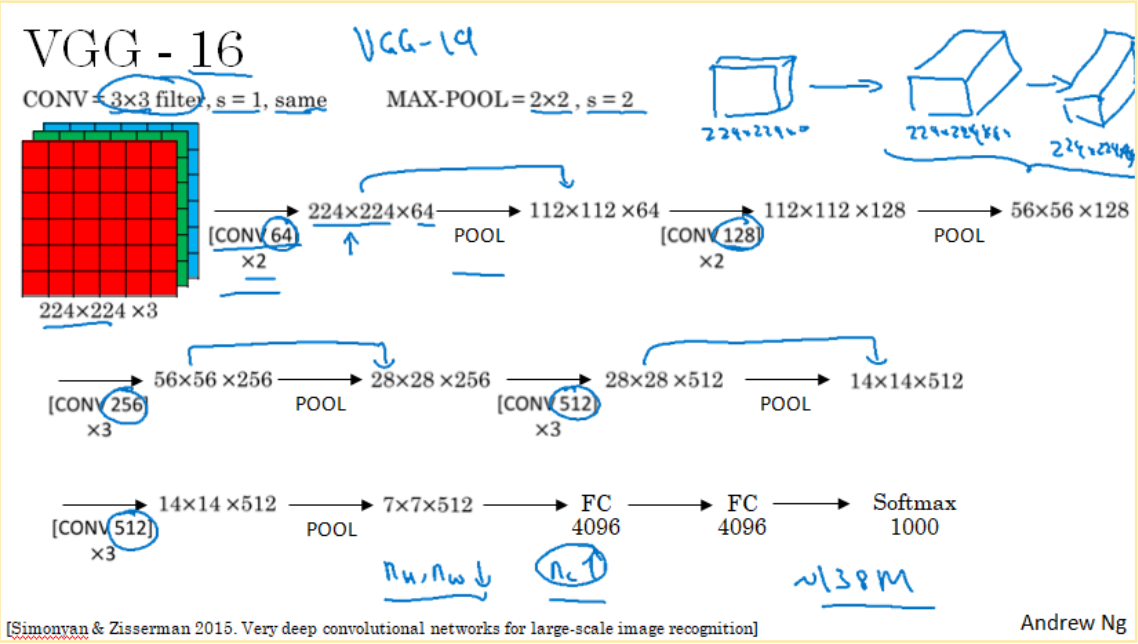
2.2.2. AlexNet

This network was much bigger than LeNet (it has 60 million parameters). It consequently achieves much better performance. One thing that makes its performance better than LeNet is the use of ReLu activation function. It also uses Local Response Normalization, which we won't cover since it doesn't help that much. It is not really used today.



2.2.3. VGG-16

The AlexNet architecture has so many hyperparameters, so the VGG-16 comes with the idea of using a much simpler network where we focus on having conv layers that are simple filters with the same padding.

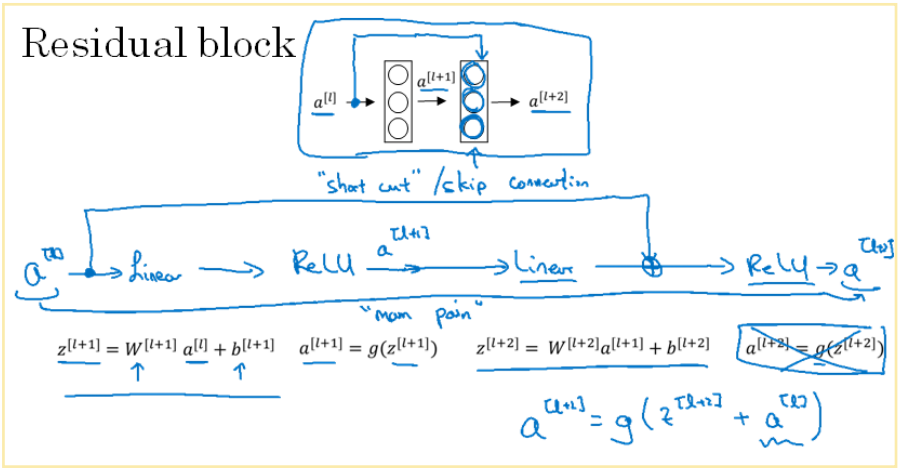


This is a really large network (138 million trainable parameters) with a simple architecture.

2.3. Residual Networks (ResNets)

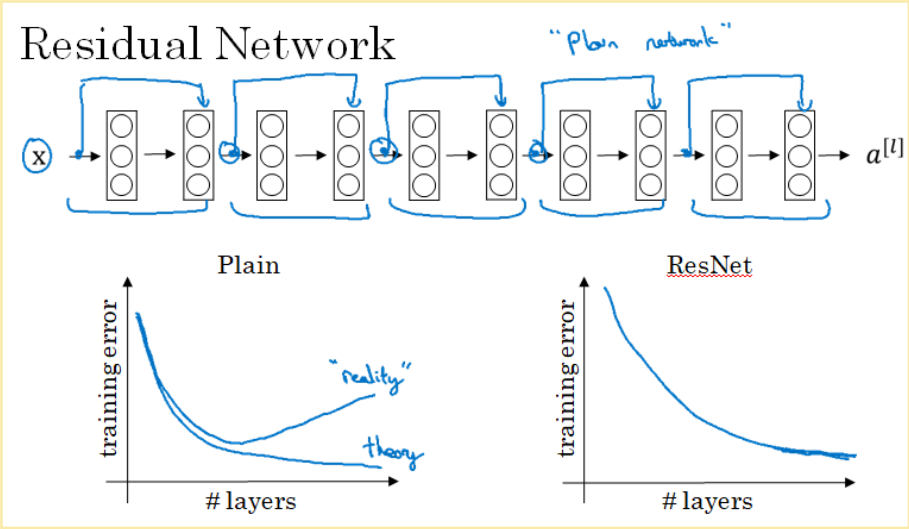
Very deep neural nets are difficult to train because of vanishing and exploding gradients. We'll learn how to skip connections and take one activation from one layer to another one even much deeper in the network. This will help us to train really deep networks.

ResNets are built out of **residual blocks**. A residual block is the following:



On a classic neural net, we have the activation on the layer *l*, compute the linear transformation and pass it through the non-linearity, and then we have the activation on layer *l+1*. We do the linear transformation again and then pass it through the non-linearity. But instead of doing this, we'll get the activation on layer *l* and shortcut it to the second non-linearity, so that $a^{[l+2]}$ is now calculated as the bottom right equation shows. And this allows us to train very deep networks.

So the way we build ResNets is by taking many of these residual blocks and stacking them together:



In theory, if we go on building a deeper network, the training error will reduce. But in reality, a very deep network has a really difficult training algorithm and the training error can increase. We can empirically find that this doesn't happen to us with ResNets.

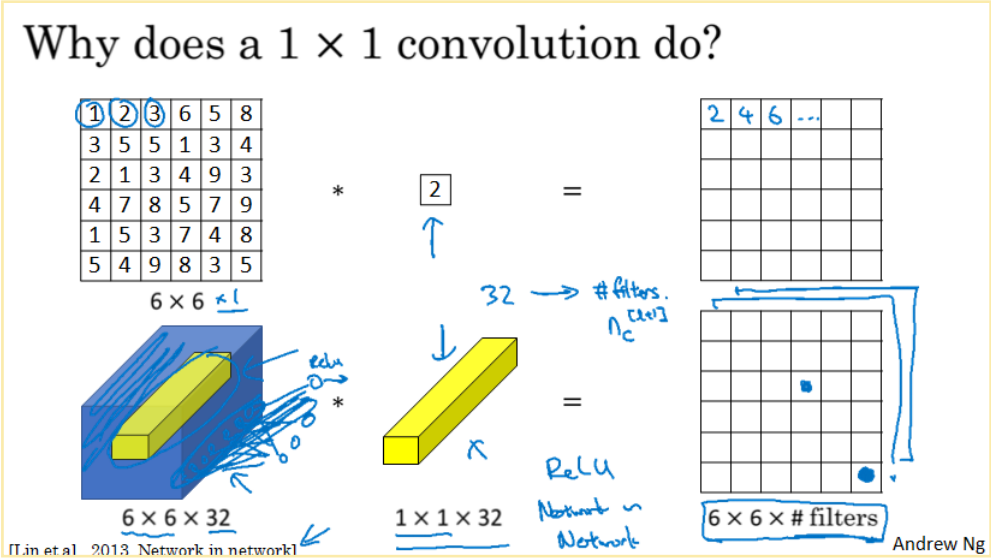
So, if we want to use Residual Networks for images, we need to add these extra connections. It is important to use *same* convolutions, since the addition of terms of different layers needs to make sense. We also need to do a weight adjustment when we perform pooling.

2.4. Inception Network

First of all, we'll discuss what a 1x1 convolution does:

2.4.1. 1x1 Convolutions

If we have only one channel on our images, a 1x1 convolution is not really useful since it just multiplies the image by the value of the filter. But if we have more than one channel, it does something that makes much more sense:

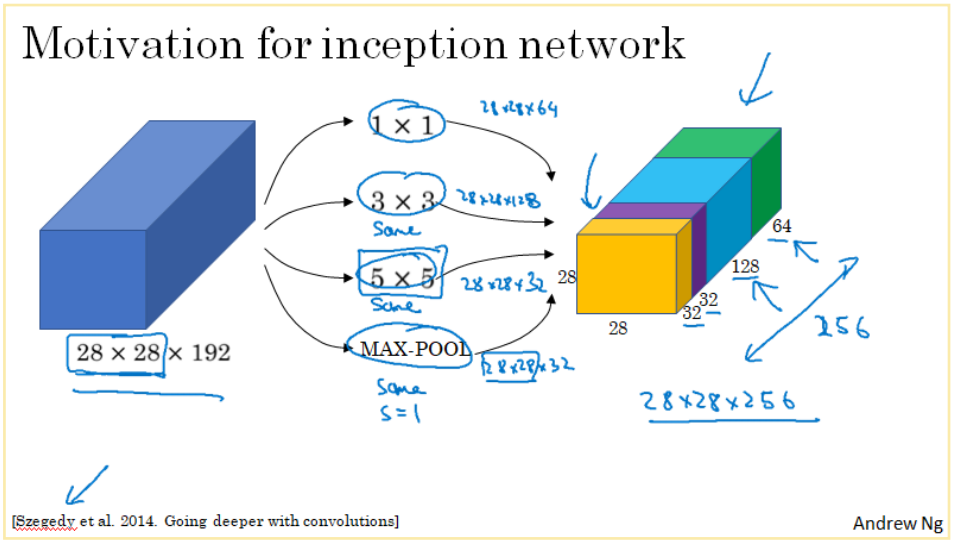


It can help us to reduce the number of channels of the input (and we can maintain width and height). This can help us to save computation. In addition, a non-linearity function is added after the filter (this is sometimes called network on network).

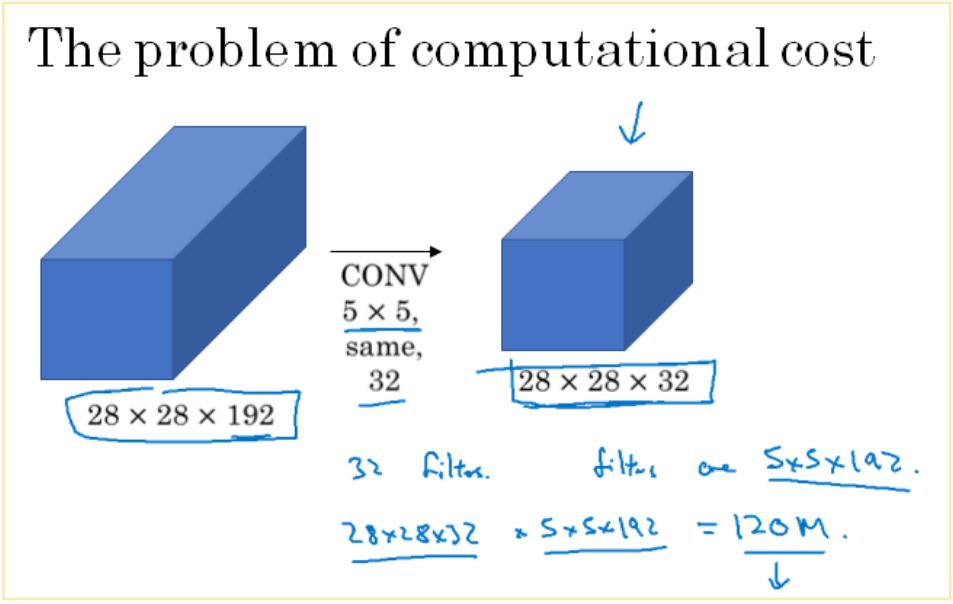
2.4.2. Inception Network Motivation

When we design a layer for a ConvNet we need to choose the dimension of the filter, or putting or not a pooling layer, etc…

Instead of choosing the values of the filters or the existence or not of a pooling layer, the inception network uses a lot of values and combines them all:

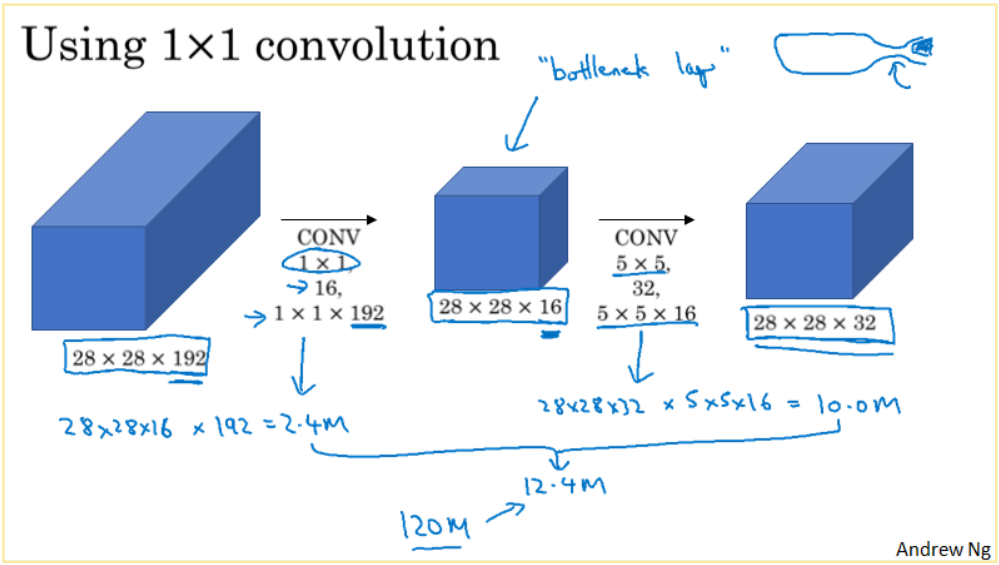


The main problem is the computational cost:



If we get one of the possible combinations (for example, the 5x5 one), we can see that we need to do 120 million multiplications in order to make the computation.

For this reason, it is really useful to use 1x1 convolutions, since it can reduce computation cost by a factor of 10:



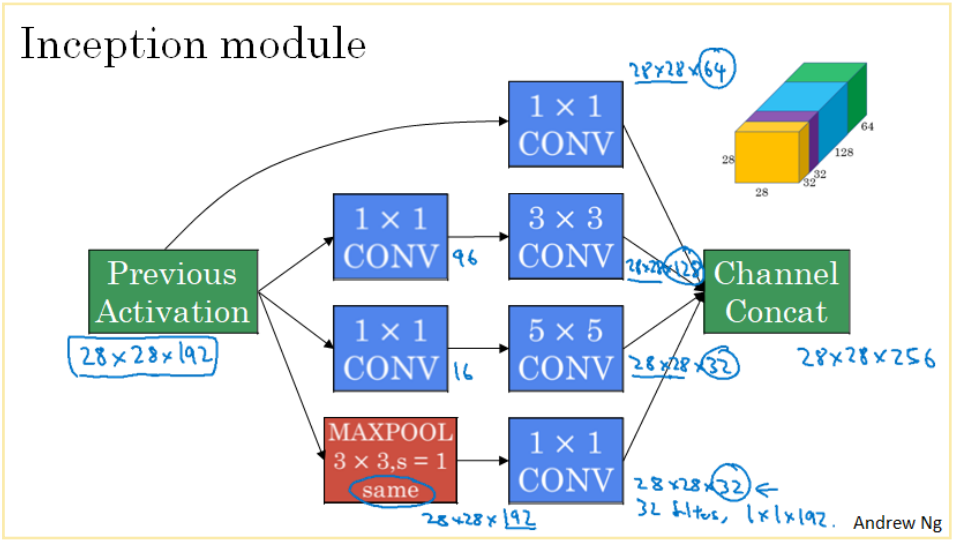
We are reducing the volume we had on the left and then performing the 5x5 conv. In this case, the number of multiplications we have to do is about 12 million.

So, to summarize, the motivation of the inception networks is the following: if we don't want to choose the size of the filters, or pooling, etc… we build a network with all the combinations and concatenate the results. This leads us to a high computational cost, and to avoid it we can use 1x1 convolutions to shrink the volumes.

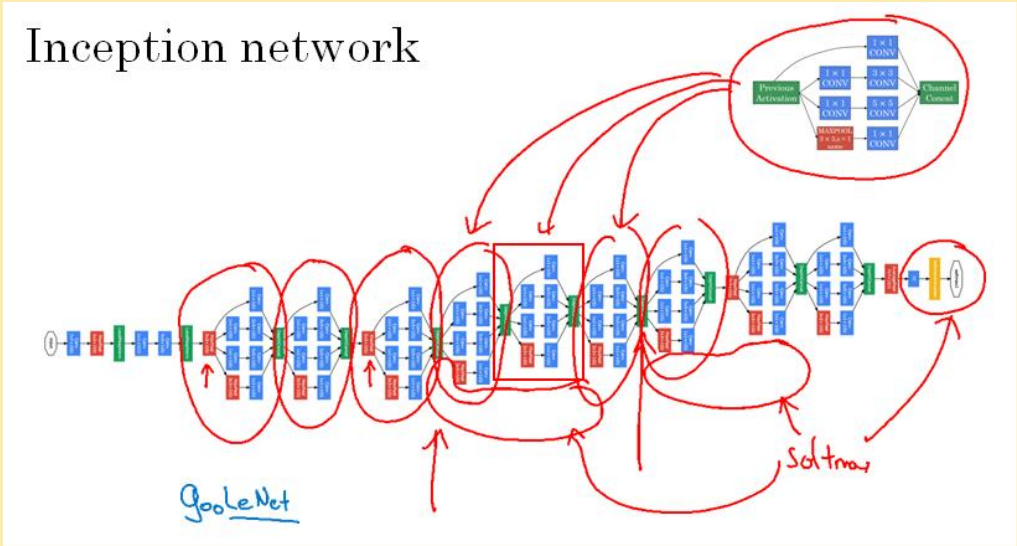
Let's see how to build these networks:

2.4.3. Inception Network Construction

It follows this scheme:



We are using the 1x1 convs before applying the 3x3 and 5x5 convs. We are also adding a 1x1 conv with no layers after it, and we are also adding maxpool. Because we use *same* convolutions for that, we need to later add a 1x1 conv as well. Finally we concatenate the channels. This corresponds to a single inception module, and we can build a net with multiple inception modules.



3. Practical advices for using ConvNets

We have seen some case studies and particular applications. Now let's focus on more practical advice for building ConvNets:

3.1. Using open-source implementation

If we want to use the architecture of a paper or build some results on top of it, we should try to find an open-source implementation (often uploaded by the authors) instead of re-implementing the net from scratch. We should also upload our implementations so that other people can use them.

3.2. Transfer Learning

When we have little data or we want to skip the process of training a network for a long time, we can download neural network architectures with the weights trained and modify the last layers to meet our requirements. Fortunately, the most common deep learning frameworks support transfer learning.

3.3. Data augmentation

Data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. The most common techniques are:

- Mirroring images
- Random cropping
- Rotation
- Shearing
- Local warping
- Color shifting (can also be done with PCA)

3.4. State of computer vision

When we deal with machine learning problems, we may have lots of data or just not enough data. When we have little data, we need to put more effort in hand-engineering (features, or network architecture, etc…), whereas when we have lots of data we just don't need that much hand-engineering and the model can learn itself.

For this reason, we usually have two sources of knowledge: hand-engineering or labeled data.

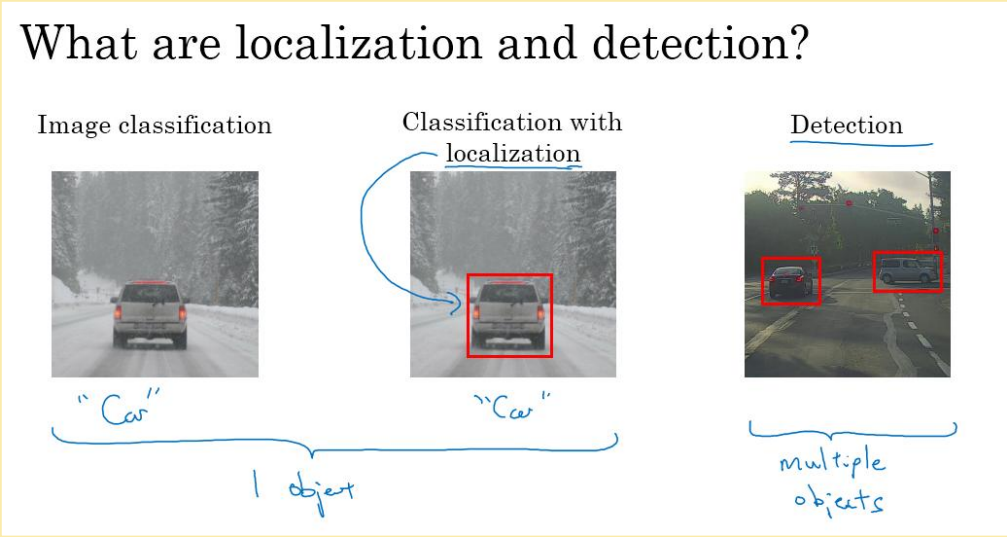
Nowadays, computer vision is in a stage where we feel like we need more data. We don't have enough data yet, so we have needed to develop complex architectures.

4. Object detection

Object detection is one of the areas of computer vision that is just exploding and is working so much better than just a couple of years ago. In order to build up to object detection, we'll first learn about object localization.

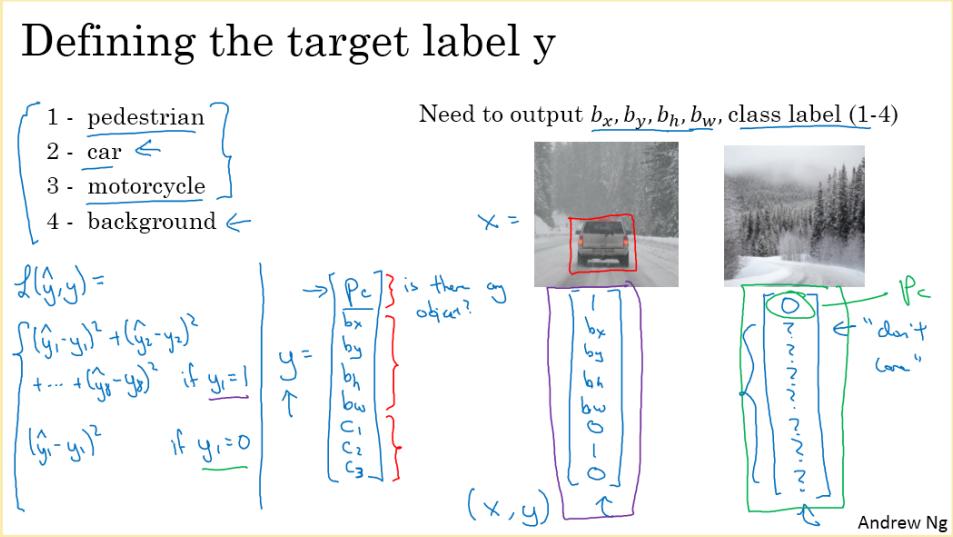
4.1. Object localization

We have learned how to solve the classification problem. The detection problem is composed of a classification with localization problem. (this means we not only have to classify the items in the images, but also say where they are). Finally, object detection also involves detecting multiple objects or classes in a single image.



In order to perform the detection, our ConvNet can output the coordinates of the bounding box that has the object in its inside.

To do this, we need to define the target label array as follows (as of now, we are assuming a single object is present in the image):

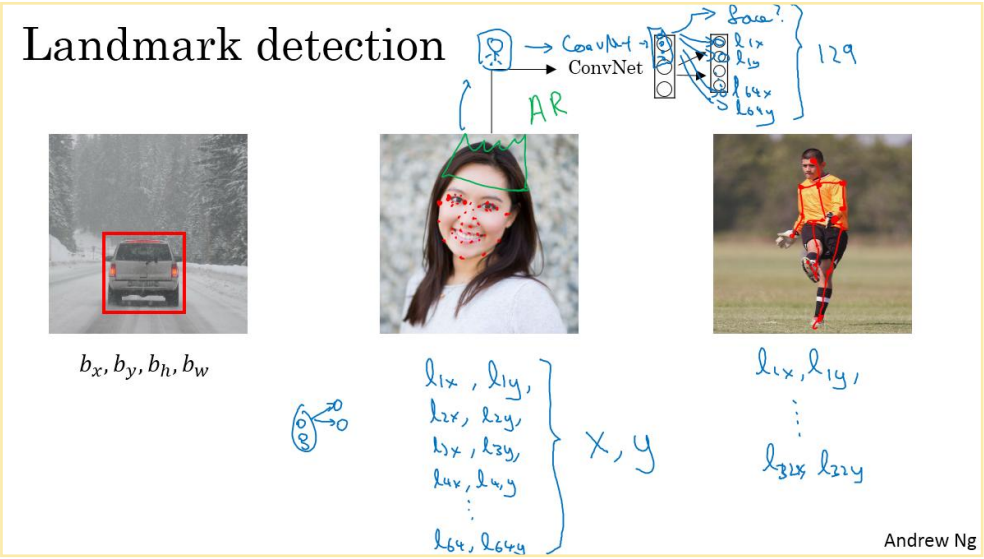


The first element is 0 or 1 depending on whether an object is present or not. The four following elements are the coordinates, and the last three the predicted category.

The loss function will be defined in two segments, depending on whether there is an object in the image. We could use a log loss, or a squared error one.

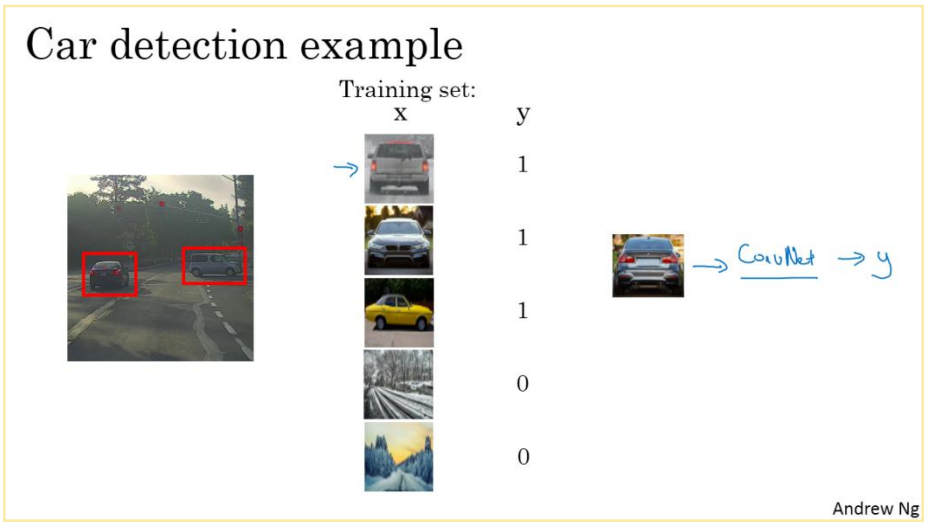
4.2. Landmark detection

We can train a neural network that can detect whatever point we want. For example, in a face, we could train the neural network so that it can detect any point we want. We would train the ConvNet with images with the landmarks annotated to make this possible.



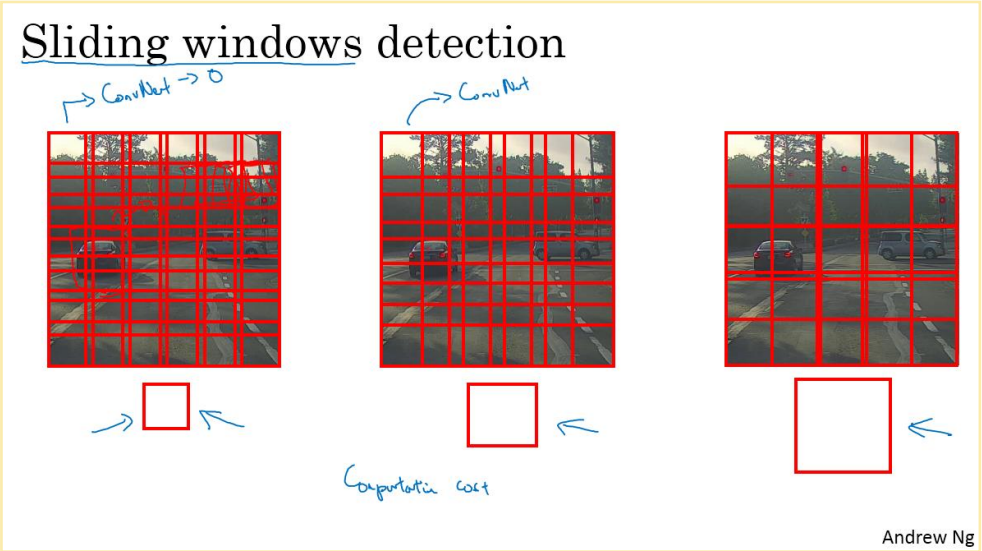
4.3. Object detection

We will begin to cover object detection with the sliding windows algorithm. Let's see an example of car detection:



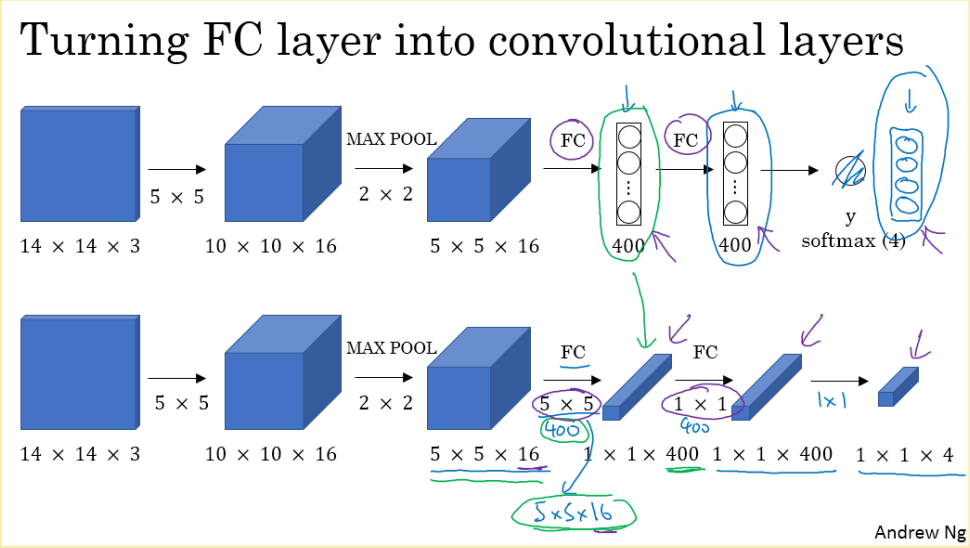
The first step would be to train a ConvNet that can detect if an image has a car or not. It is important that the training set has its images as cropped as possible, meaning that they only contain the car if there is a car in it.

Then we divide the image in a grid and feed each part to the network. On each part of the grid, the net will predict whether a car is in it or not. The main disadvantage of this method is the computational cost, but we will see how to reduce this cost with a **convolutional implementation**.

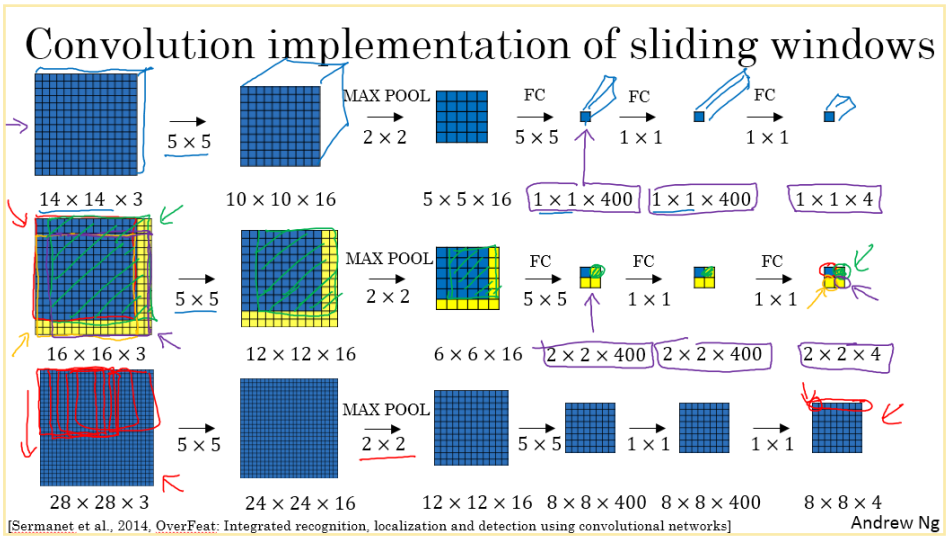


4.4. Convolutional implementation of sliding windows

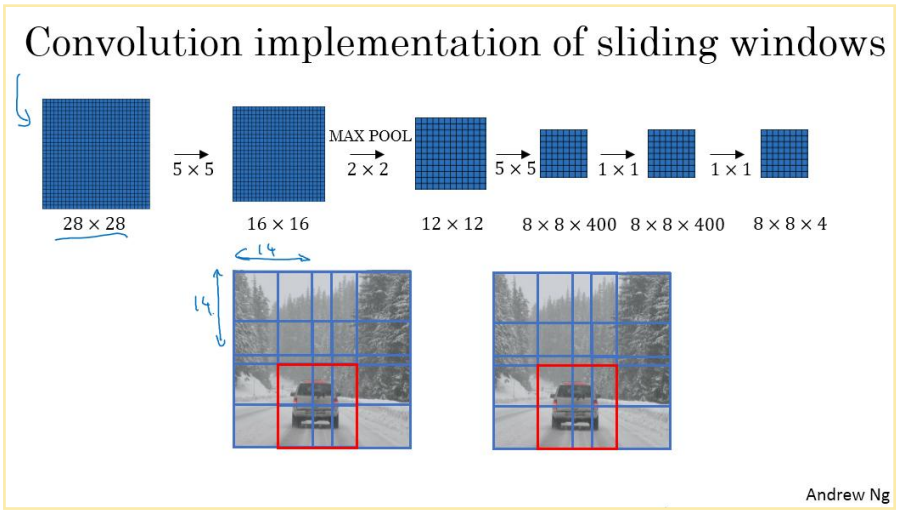
First, we will see how to turn fully connected layers into convolutional layers (which will be computationally cheaper).



If we replace the 400 nodes fully connected layer by 400 5x5 filters, we will mathematically get the same result. Once we know this, we can see a convolutional implemention of the sliding windows algorithm:



The idea is that, instead of feeding every region of the grid to the net and get a prediction at a time, if we feed the whole image, each element of the output will be the result of the prediction of a grid (see the yellow squares for intuition). If we do this, with a single pass through the net we can get the output we want.

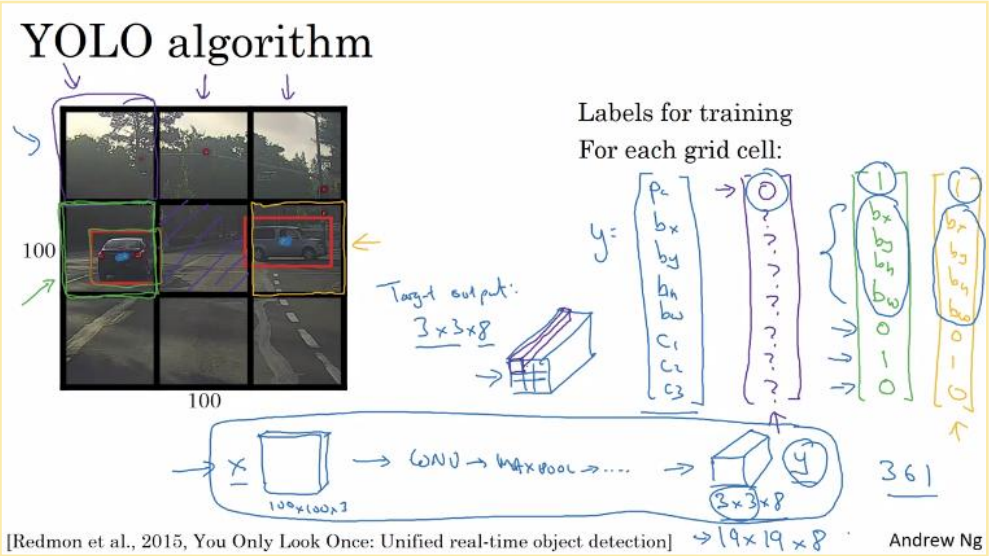


This is a much more computationally efficient implementation. However, it still has a drawback: it does not accurately print the bounding boxes, because when dividing the image with the grid, maybe none of the boxes really match up perfectly with the position of the car. In addition, the perfect bounding box may not even be a square: instead, it can be a wider rectangle.

One way to solve this is the YOLO (You Only Look Once) algorithm.

4.5. Bounding Box Predictions

What we are going to do is apply the algorithm we saw at 4.1. We will divide the image in cells and apply it to each one.

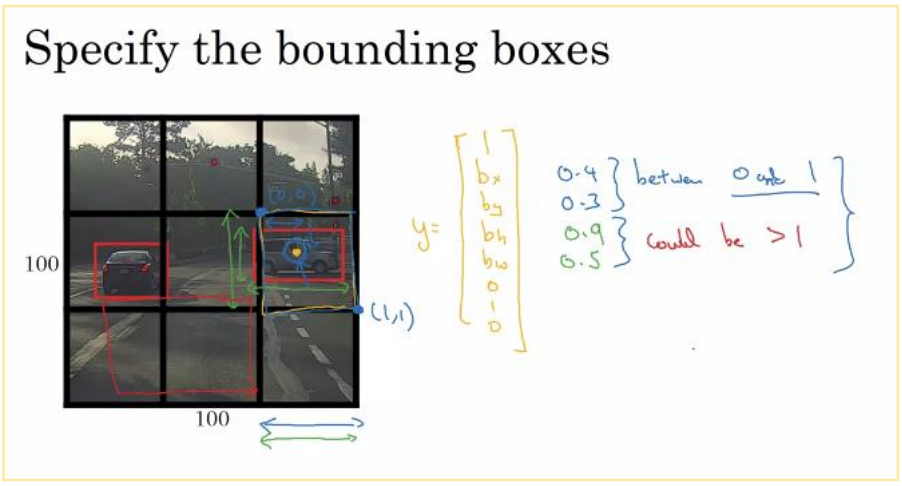


If we have a 3x3 grid, then the output vector will be 3x3x8 (for each cell, we'll have a vector). So we will take our input image and apply convolutional layers, maxpooling, etc… to get to a 3x3x8 volume.

In addition, it's important to note that each object will be assigned to a single cell (the one who contains its midpoint). This will work fine unless we have multiple objects in the same cell, but this is an issue we'll address later. But one thing we could try is to use a finer grid.

Note that this algorithm **outputs the bounding boxes coordinates**, and is carried out with a **convolutional implementation**. This is a fast and efficient algorithm that even works for real time object detection.

One last thing to note is the way bounding boxes are specified (see next slide):

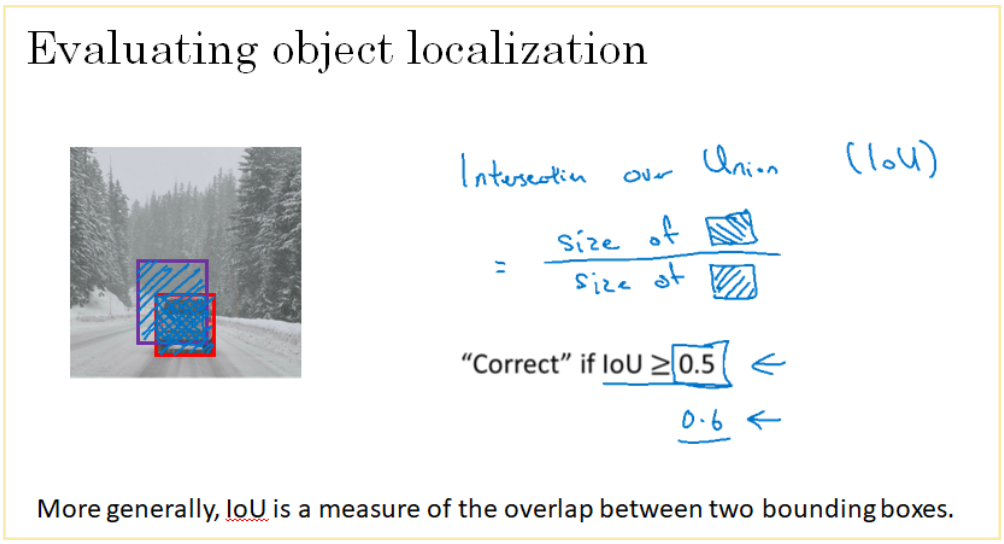


Let's see some ways of making this algorithm even better.

4.6. Intersection over union

So, how can we know if our detection algorithm is working fine? We'll learn a function called intersection over union, which will be used both for evaluating our detection algorithm and adding another component to our algorithm.

If we have the real bounding box and the predicted one for an object, we can measure the intersection and the union area. The more similar they are, the better.



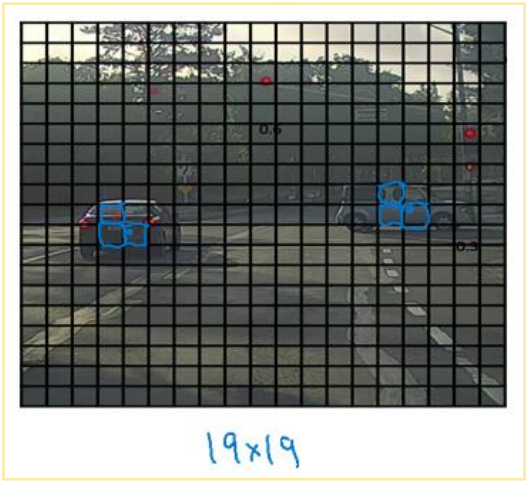
The *correct* threshold is specified by convention.

4.7. Non-max suppression

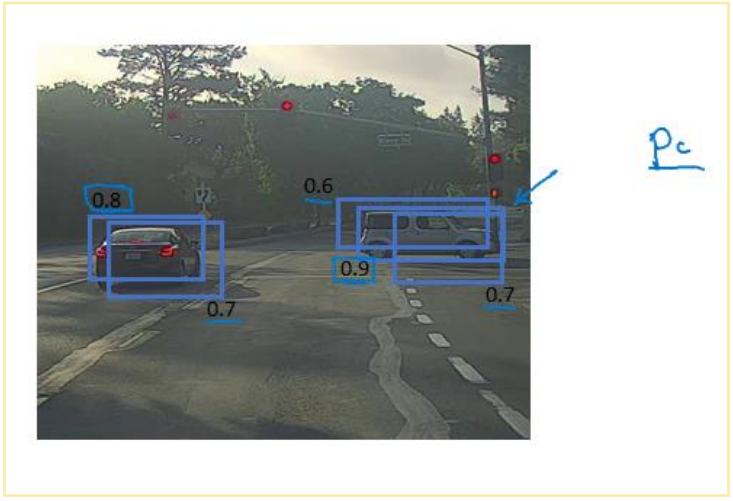
One of the problems of object detection as we've learned so far is that our algorithm may find multiple detections of the same objects. **Non-max suppression** is a way for us to make sure that our algorithm detects each object only once.

Let's say we want to detect pedestrians, cars, and motorcycles in this image:

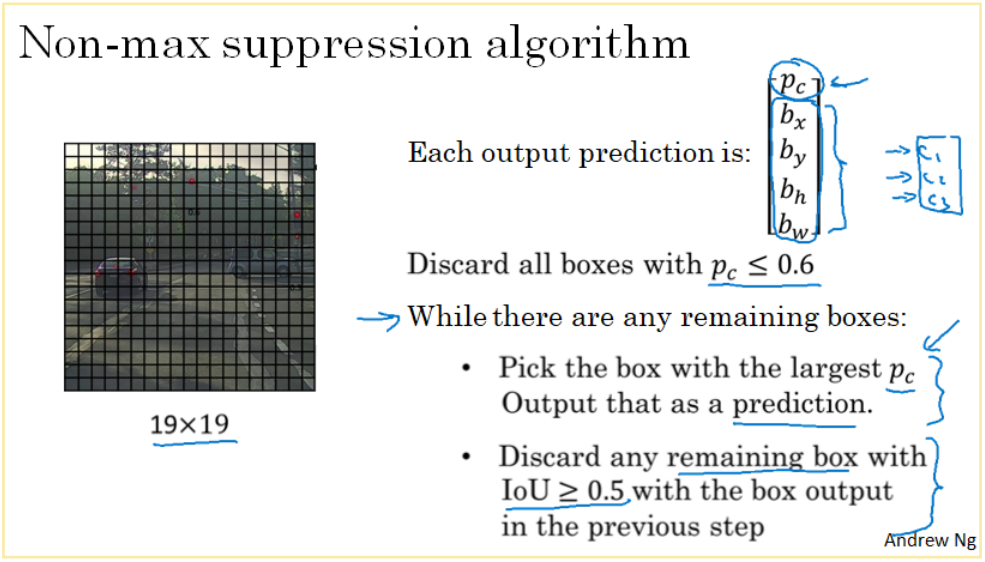
And we create a 19x19 grid. It turns out that the object may be detected in several different cells.



So we might end up with this:



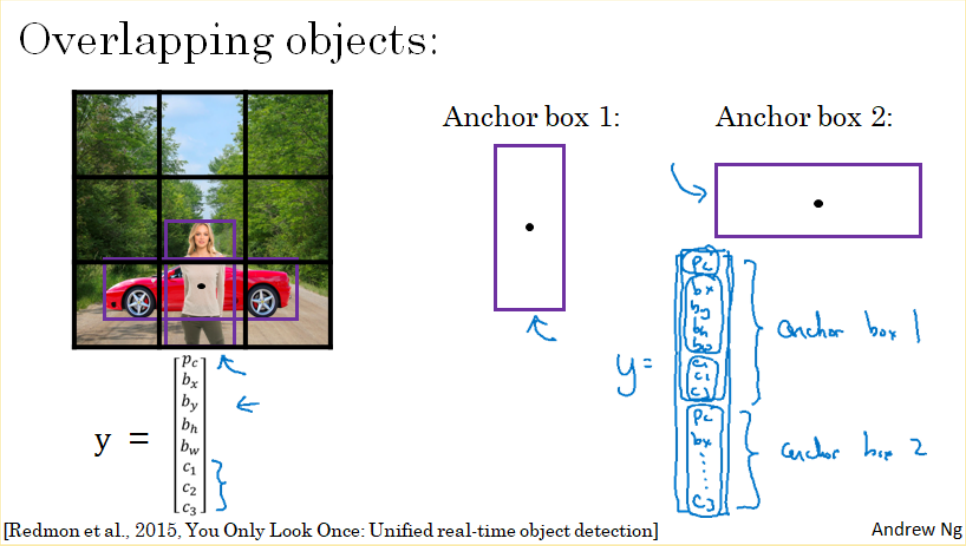
What non-max suppression does is clean this up and find a unique box for every object, doing the following:



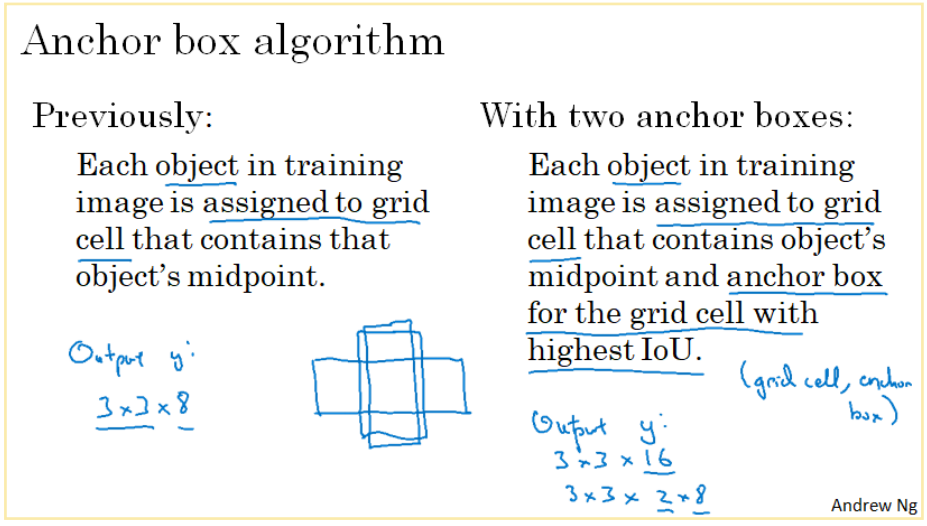
Finally, there is one last idea that makes the YOLO algorithm work much better:

4.8. Anchor Boxes

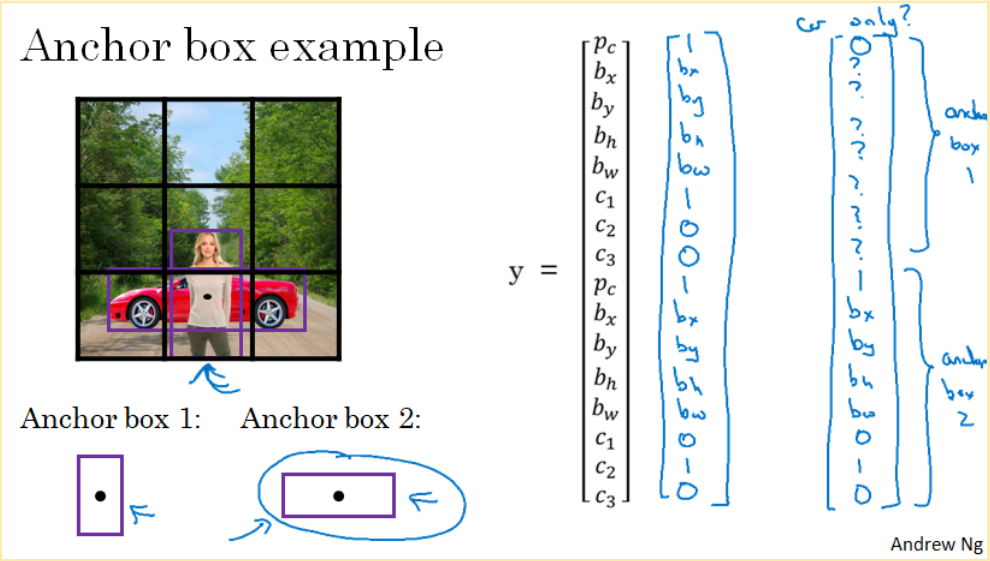
Another problem we have seen is that each of the grid cells can only detect one object. If we want to detect multiple objects, here's what we can do:



We'll basically repeat the *y* vector twice, and we'll *compare* it with the anchor boxes. SO, the algorithm looks like this:



Let's see an example:



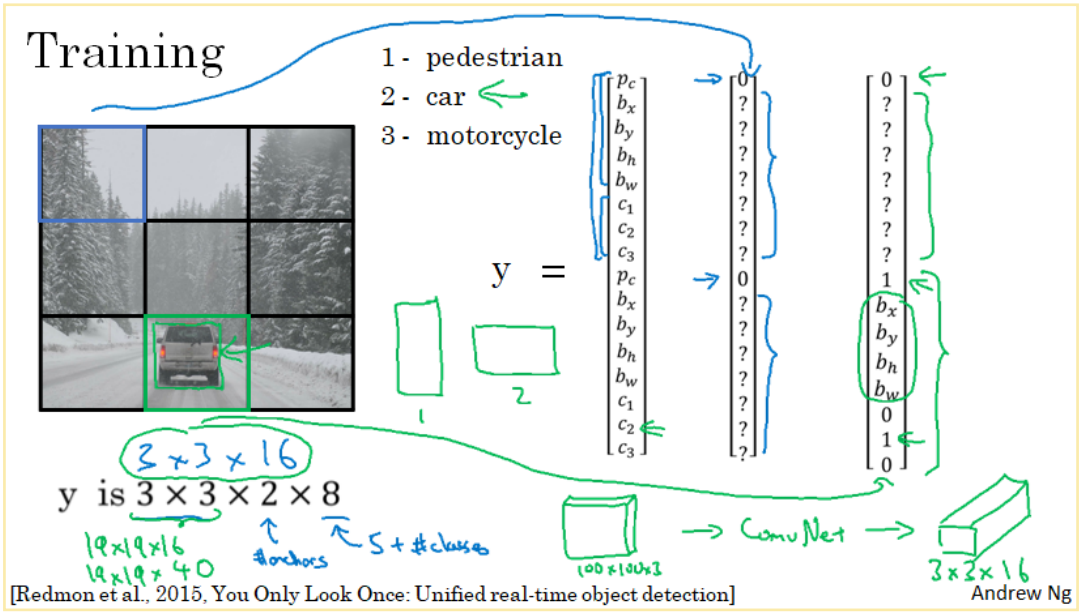
So, how do we choose anchor boxes? We can choose them by hand or choose a number of shapes to cover the types of objects we are expecting to detect.

Once we've seen all the components of the YOLO algorithm, let's wrap up and put all the components together.

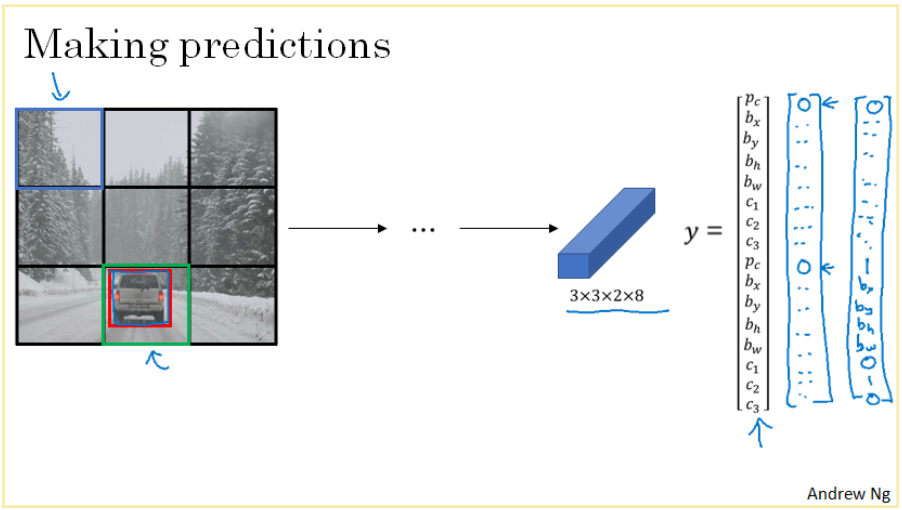
4.9. YOLO algorithm

Let's suppose we are constructing the training set of a model to detect pedestrians, cars and motorcycles using two anchor boxes and 3x3 grids.

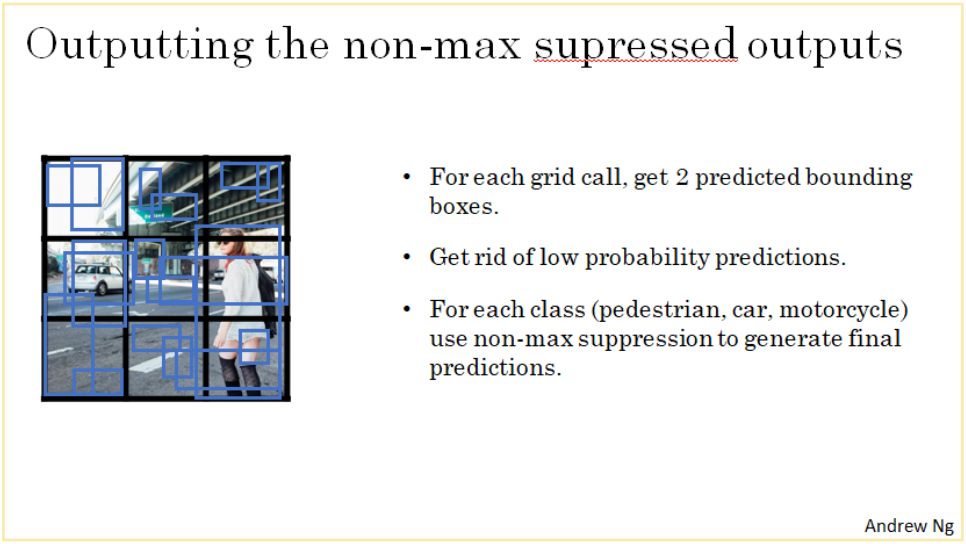
To construct the **training set**, we go through all the cells and construct the *y* vector, taking into account which anchor box each object is associated to.



To **make predictions**, we'll feed each image into the model with the convolutional implementation.



Finally we need to get the **non-max suppressed outputs**:



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.

5. Face Recognition

We have two distinguish between two concepts: face verification and face recognition:

- **Face verification**: we want to recognize whether a given input image or name/ID is that of the claimed person.
- **Face recognition**: we have a database of K people and we output if the image is any of the K people.

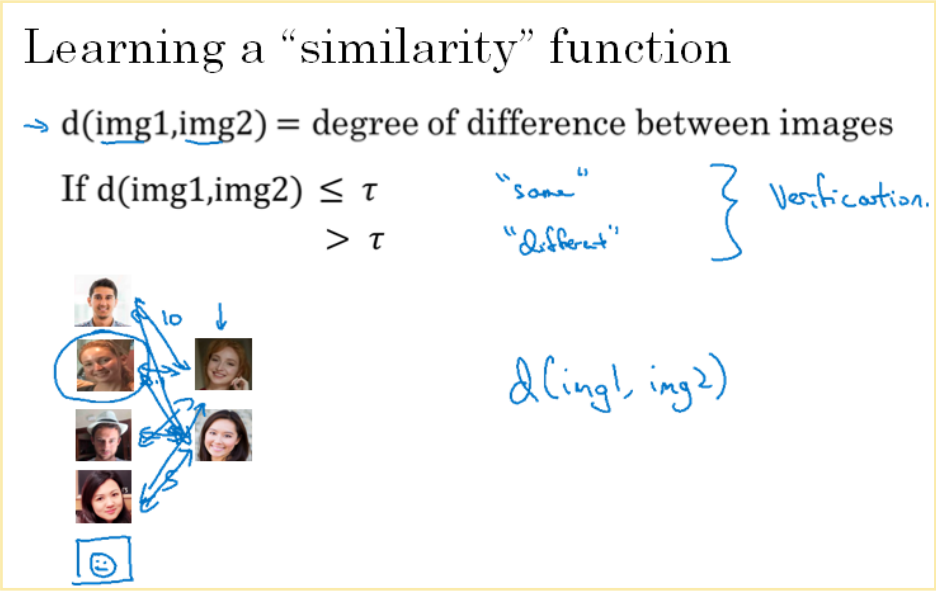
The second problem is much more difficult because we have a lot more chances to get wrong.

We'll start covering the face verification problem. One of its difficulties is that we need to solve a one shot learning problem. Let's see what's that:

5.1. One Shot Learning

In face recognition, we only have a single training image for a face. For example, we might have a single image of an employee that we want to recognize on out database. This is called as **one shot learning**.

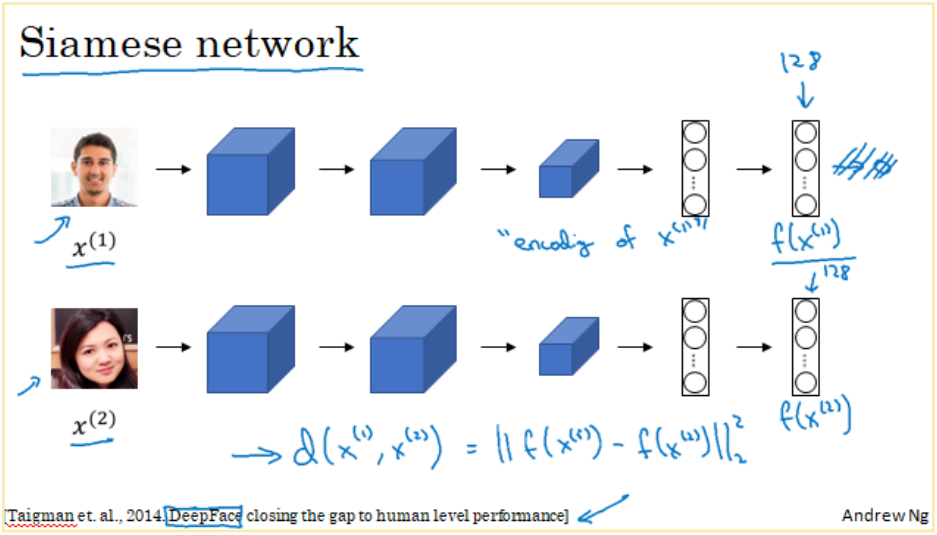
We could train a convnet with a database with a single image for every person, but it would not return a robust model. In addition, if a new person joined, we would have to re-train the model. So what we're going to do is use **similarity**:



If we learn a function that, given two inputs, computes a similarity value between them and we fix a threshold, we'll be able to address the problem.

5.2. Siamese Network

A good way of learning this function is the **siamese network**. The idea is to compare the outputs obtained when feeding each image to the same neural network (which is in fact some kind of "encoder" of the images):



So we can calculate d as the norm of the difference between the two vectors that represent the two images.

So, how do we train this network? It's important to note that these two nets have the same parameters. They are just like encoders.

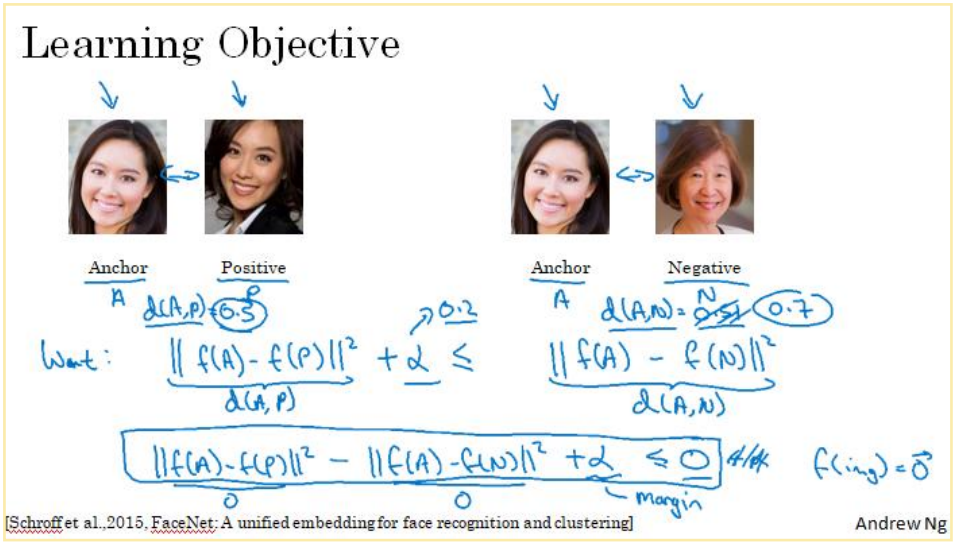
So what we want to do is to learn the parameters so that if the images are of the same person, then the distance between the output vectors is small.

To do this, we'll use the **triplet loss function**.

5.3. Triplet Loss Function

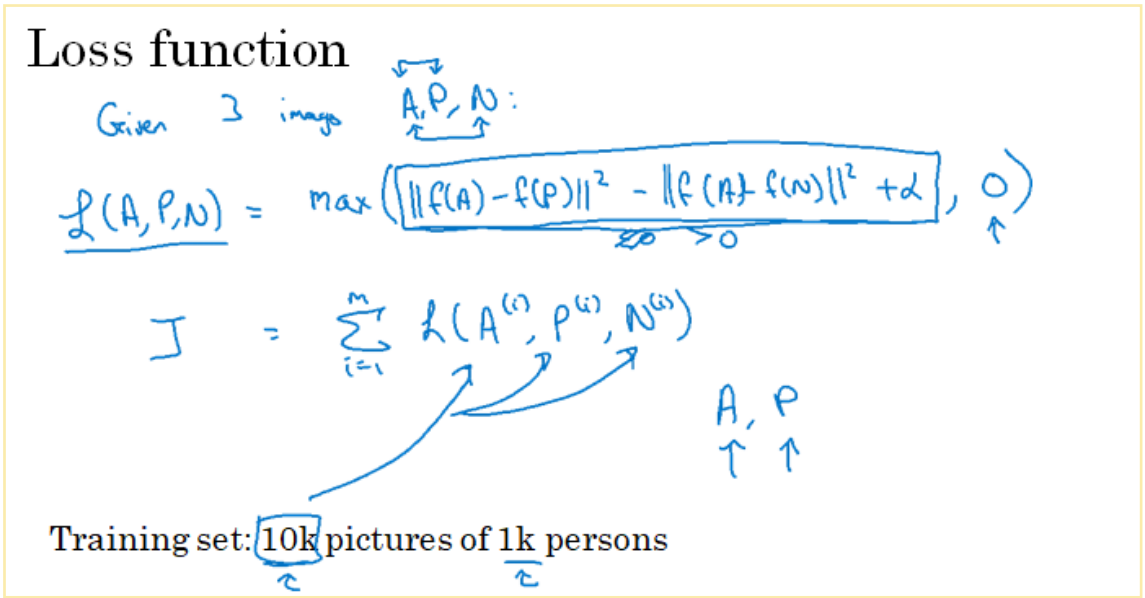
We'll use an anchor image and a positive and negative example. So we want the distance between anchor and positive image to be less than the distance between the anchor and negative image.

So we want the parameters of our network to fulfill the formula at the bottom of the next image:



We introduce the *alpha* hyperparameter, which is near zero, to make sure the equation gets solved with the trivial solution of all terms equaling zero.

Let's formalize the equation and define the loss function:

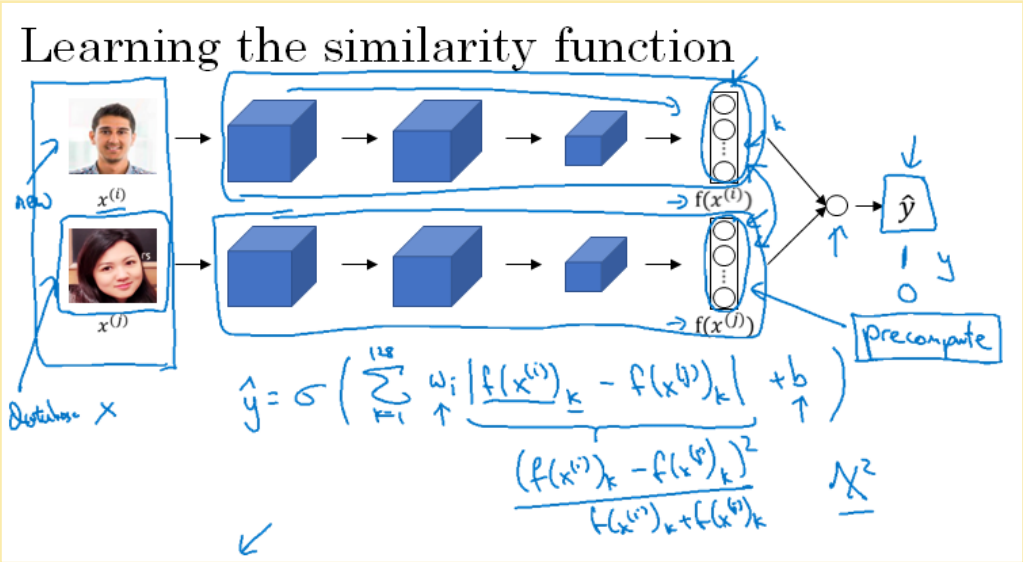


We take the *max* so that if the first term is less than zero then the loss is 0. It is important to know that in the training set we need more than one image of each person (for example 10) so that we can make pairs of anchor and positive images of the same person.

So, how do we define the triplets? It turns out that if we choose the triplets *A, P, N* randomly the condition is easily satisfied. So we want to find triplets that are "hard" to train on. We are not to see this in a detailed way.

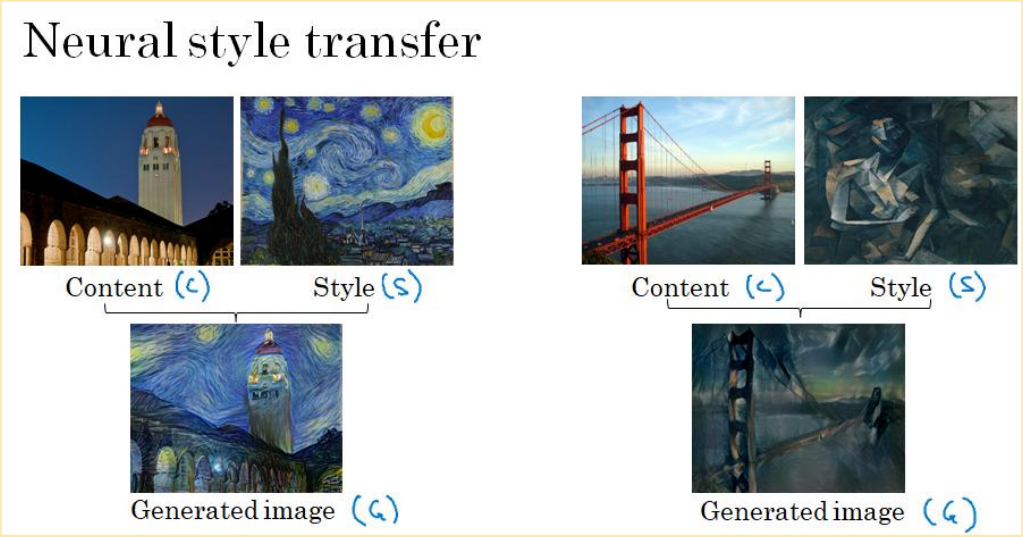
Nowadays, enterprises use datasets of millions of images (1, 10, 100 millions). Luckily, some already trained models are posted so we can use them.

To finish, let's note that there are other alternatives to this method. For example, we could use binary classification in a supervised learning approach:



6. Neural Style Transfer

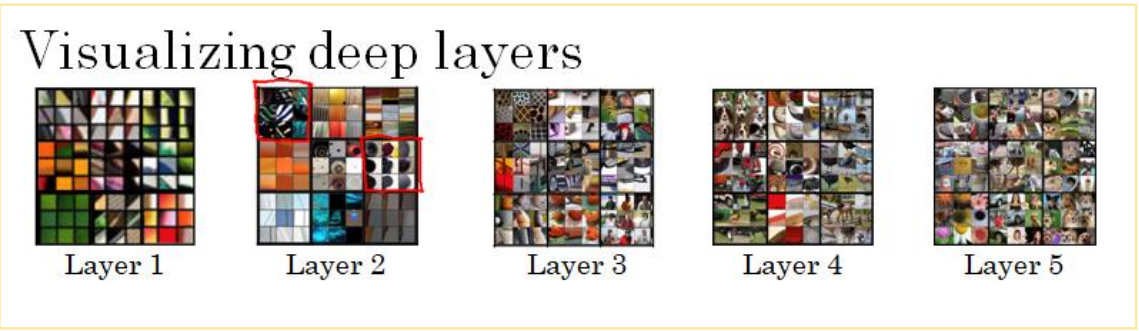
Neural Style is transforming an image (*C*) using a certain style (*S*) to generate a new image (*G*).



Let's first understand what is happening in the deep layers of a convnet.

6.1. What are deep ConvNets learning?

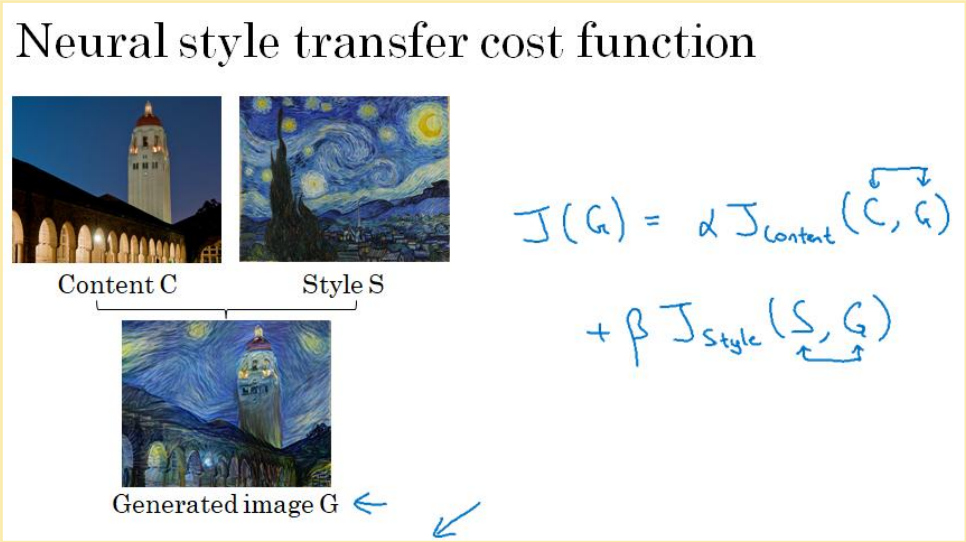
If we take a single unit of the first layer and find the 9 input images that maximize the activation of the unit, we will be able to see what shapes "estimulate" that neuron. If we do it for 9 units in each layer, we can see something like this:



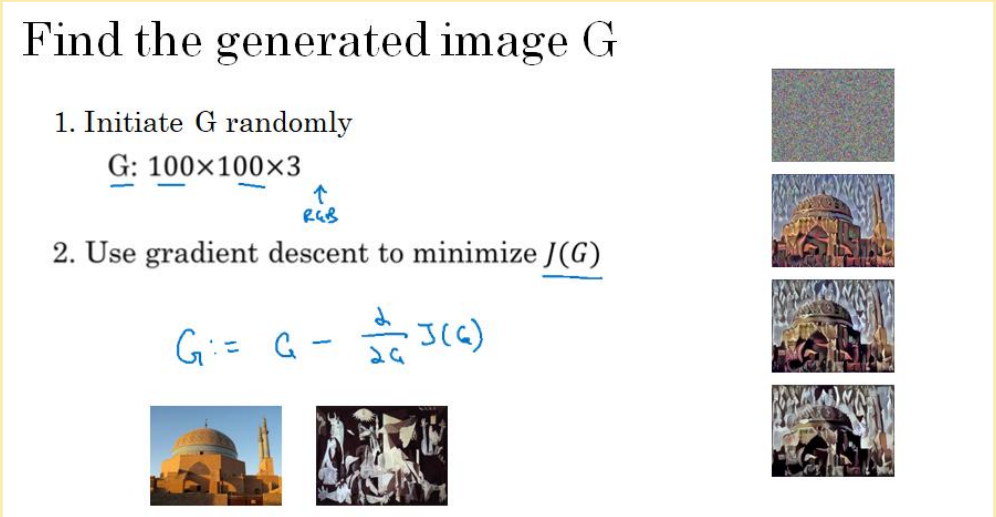
We can see that as we go deeper into the net, more complex patterns are learned.

6.2. What are deep ConvNets learning?

So as to build a neural style transfer system, let's define a cost function for the generated image. By minimizing it, we will be able to generate the image we want. The cost function looks like this:



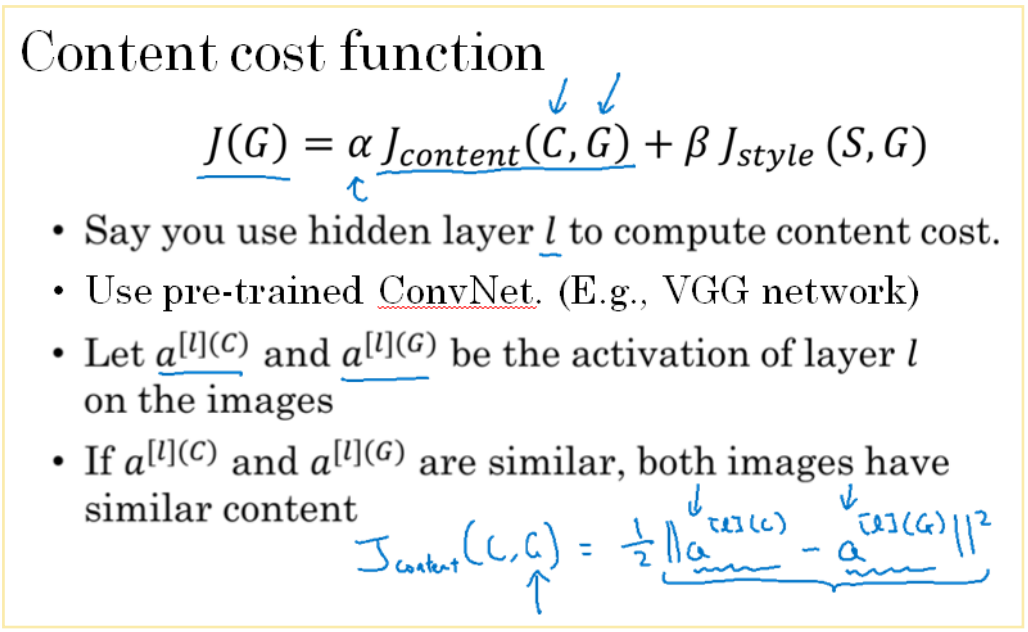
Then, by using gradient descent, we will be able to minimize the cost function and get the image we want:



Let's see how to define the **content** cost function and the **style** cost function:

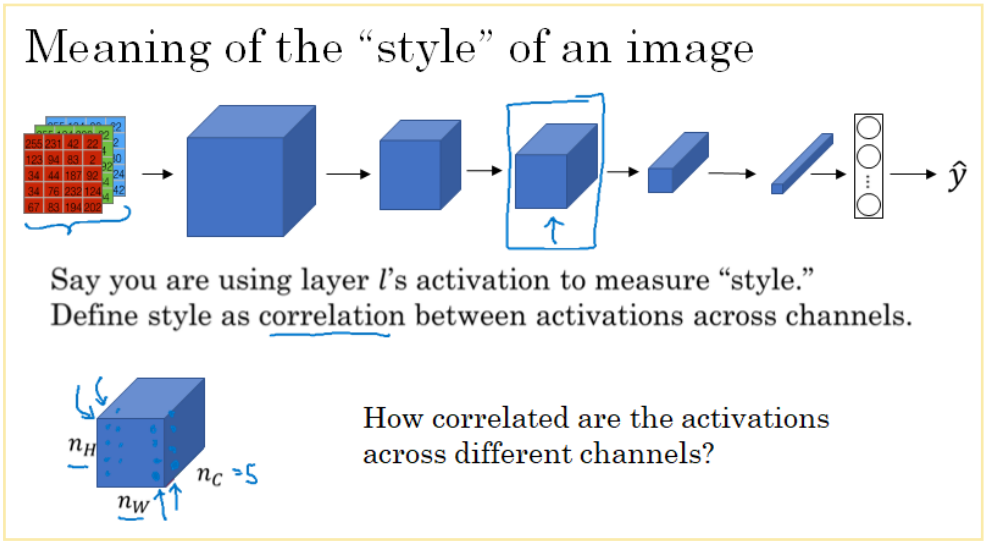
6.3. Content cost function

The content cost function is defined as follows:



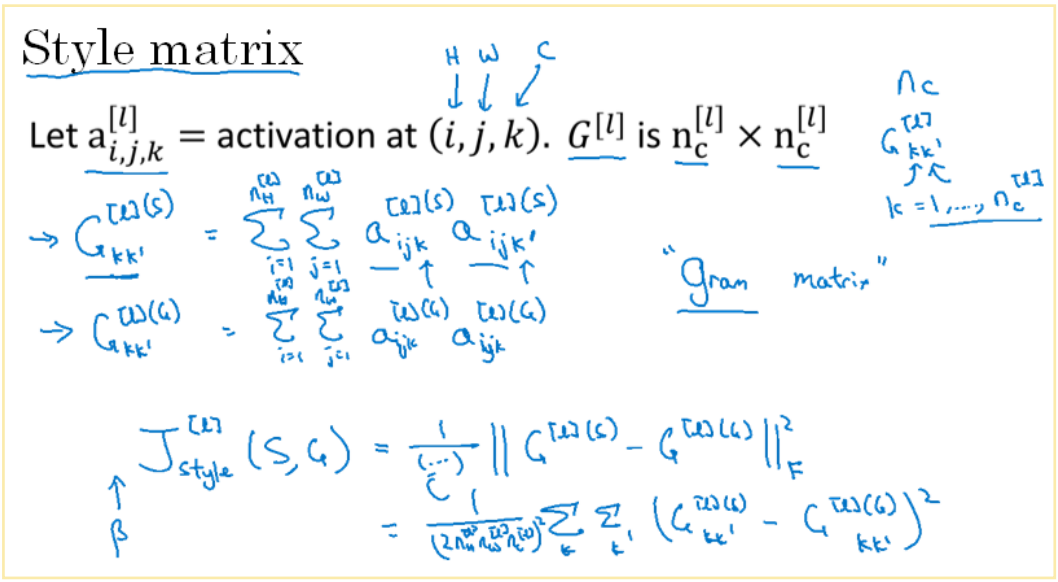
6.4. Style cost function

To capture style, we will look at the correlation between activations across channels:

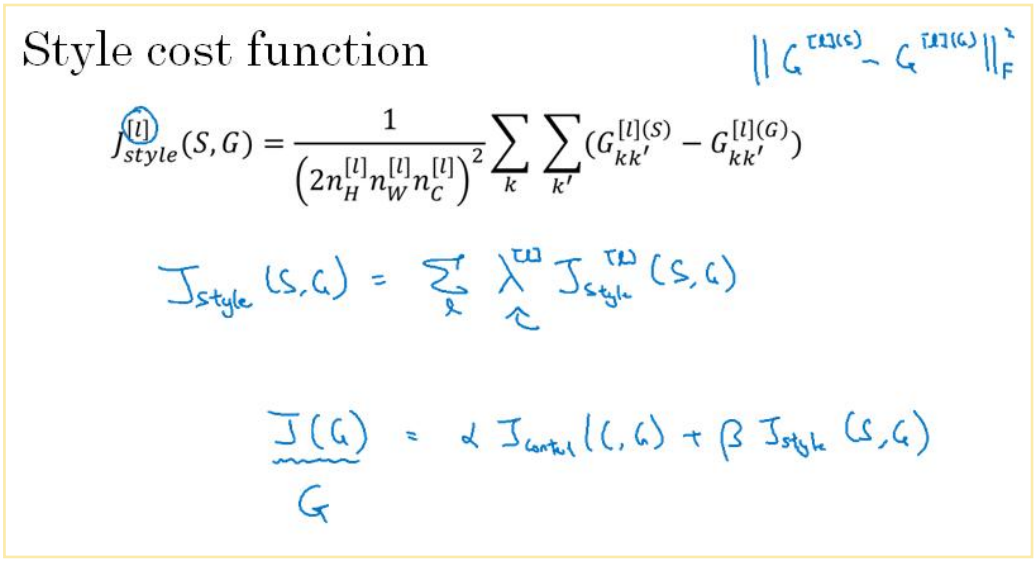


Why does this capture style? The intuition is that if some activations are correlated across channels, the high level features or texture components will tend to occur or not occur together.

So, for a given image, we can compute the style matrix:



Let's see the style cost function formula in a tidier way:



Finally, to wrap up this course, we'll cover how these implementations generalize on 1D and 3D images.

6.5. 1D and 3D generalizations

With 1D data we could use the same ideas of convolutional neural networks. However, it is more common to use recurrent NN, which we'll see in the next course.

With 3D data we can also use the same ideas.