So Far We have talked about the general structure of neural networks, However when we look at specific applications, much more can be done.

This is because If your data has some structure, and your network doesn’t need to learn that structure from scratch, the performance is much better. Ex. If you want to read the letter in an image, color isn’t relevant so design your network to take an input that is greyscale. It will reduce the complexity

Ex. If you have an image input, and you want to determine wheter the image is a cat. Does it matter wether the cat is in the center, top left/right? It doesn’t matter where the image is. If it doesn’t matter where it is, it lowers the complexity if you tell your network that objects in images are the same wheter they are in the left or the right side. It is called **Translational Invariance**

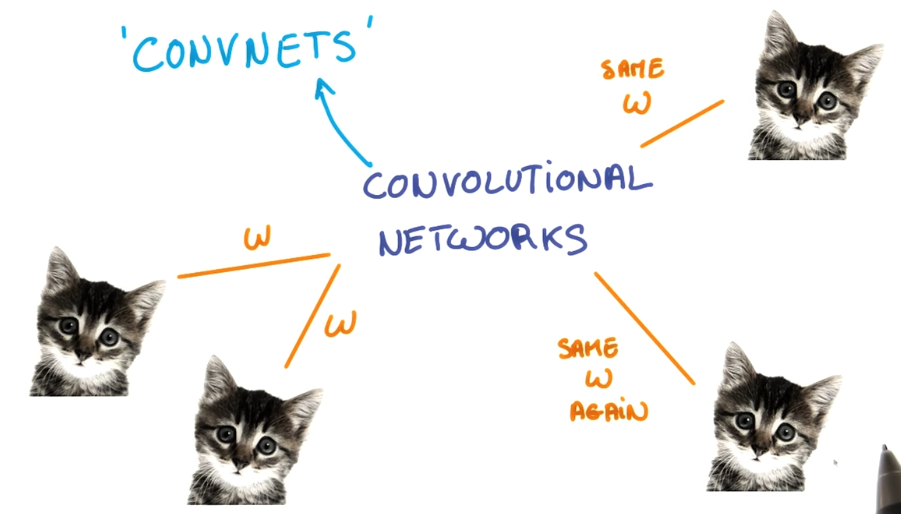
Ex. When gaining meaning from text, does the position of a noun like kitten change the definition? No so the same part of the network that learns what a kitten is can be resused instead of re-learning it everytime.

To achieve this (reusing same part of the network), you use something called **Weight Sharing**. When 2 inputs gain the same kind of info, you share their weights and train them jointly. This is used all the time for things with Statistical Invariance (Things that don’t change with time or space) and are everywhere.

Weight sharing is used for Images with Convolutional Neural Networks, and with text and sequence in general, it leads to embeddings and recurring neural networks.

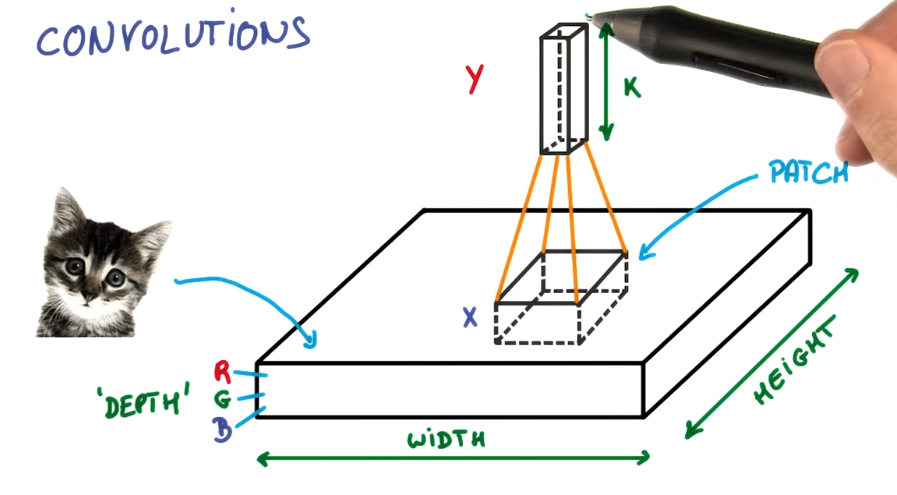
**Convolutional Neural Networks (Covents)**

A neural network that shares its parameters across space (due to the fact that an object in an image is the same object regardless of where the object is in the image (it is Statistically Invariant))

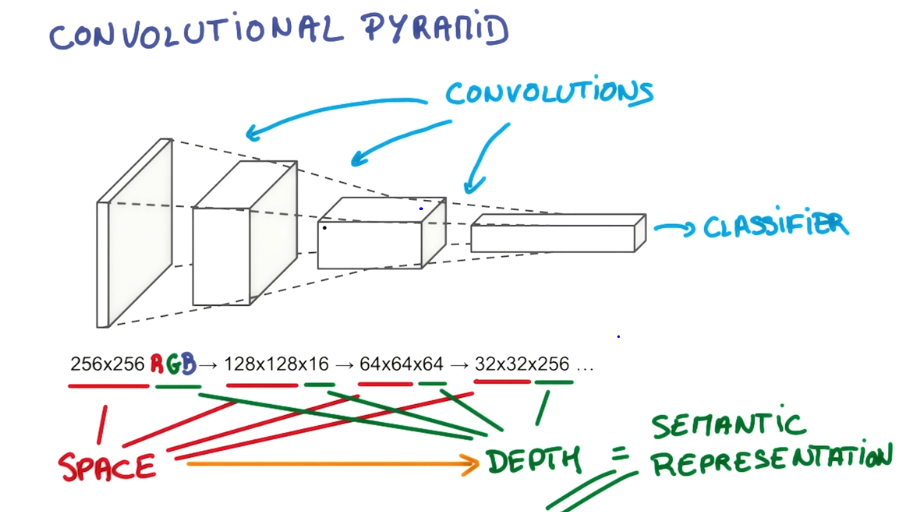


Ex. A color image has height, width and depth (3). Image taking a small patch of the image and running a neural network to get K outputs, then using convolution on that output and the entire image. The final output will be another image with a different width, height, and depth/color channels (k channels).

If the small patch was the whole image, it would be the same and running a neural network on the whole image, so making it a smaller patch lowers the complexity greatly.

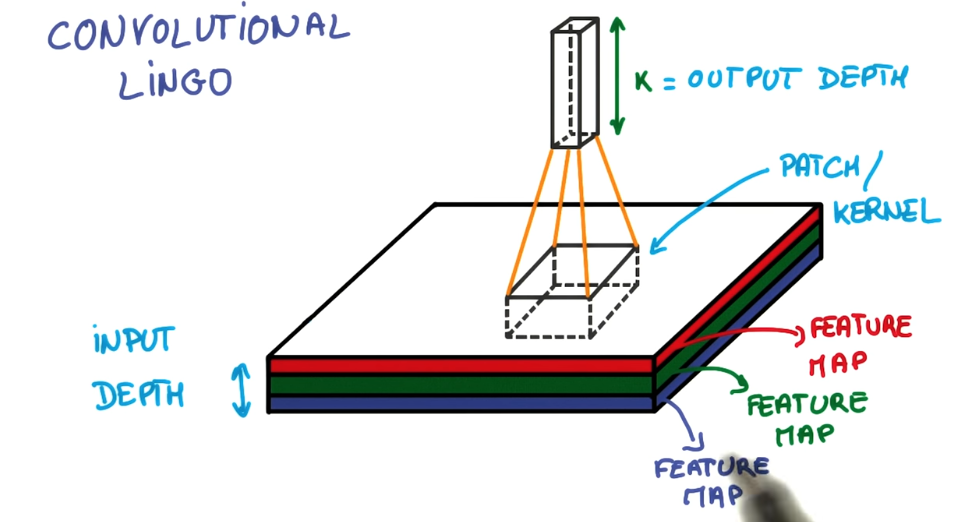


The smaller patch allows for much less weights, and those weights are shared across space. A convolutional neural network is a deep network that instead of having stacks of matrix multiplied layers, we will have stacks of convolutions.



General Idea is the Pyramid Structure. At the bottom you have a large, shallow image (Shallow because of the depth of 3). The convolutions you do will progressively squeeze the space dimensions while increasing the depth, where the depth represents the semantic complexity of your representation.

At the end, you have a representation of all the spatial information is squeezed out and only parameters that are associated with the content of the image remain. At the end is where you can use your classifier.



The small patch of the image chosen is sometimes called a kernel and each layer of your image is called a feature map. (So you are mapping your feature maps to your k feature maps)

The **Stride** is the number of pixels you shift each time you move the weights gained from the patch (those weights are a filter applied to the image through convolution).

A Stride of 1 makes the output roughly the same size as the input (depending on boundary conditions used when applying the filters). A Stride of 2 makes an output half of that size.

Boundary Conditions: not going passed the image is called **Valid Padding**. Padding the boundary with zeros to get an output image that is the same size as the input is called **Same Padding**.

Same Padding: Output Width (or Height) = Width (or Height)/Strides

Valid Padding: Out Width (or Height) = (Width (or Height) – Filter Width + 1)/Strides

Example:

Covnets work similar to how humans determine what is in an image. To determine wheter a dog is a golden retriever we try and identify certain parts. The nose, the eyes, the fur and try to break down the image into smaller parts, recognize them, and then combine them to get an idea of the overall dog.

Those parts like the nose, can be broken down even more into the 2 black holes for the nostrils, and an oval for the overall nose.

Broadly speaking, this is what a CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

In our case, the levels in the hierarchy are:

* Simple shapes, like ovals and dark circles
* Complex objects (combinations of simple shapes), like eyes, nose, and fur
* The dog as a whole (a combination of complex objects)

With deep learning, we don't actually program the CNN to recognize these specific features. Rather, the CNN learns on its own to recognize such objects through forward propagation and backpropagation!

A CNN might have several layers, and each layer might capture a different level in the hierarchy of objects. The first layer is the lowest level in the hierarchy, where the CNN generally classifies small parts of the image into simple shapes like horizontal and vertical lines and simple blobs of colors. The subsequent layers tend to be higher levels in the hierarchy and generally classify more complex ideas like shapes (combinations of lines), and eventually full objects like dogs.

Once again, the CNN ***learns all of this on its own***. We don't ever have to tell the CNN to go looking for lines or curves or noses or fur. The CNN just learns from the training set and discovers which characteristics of a Golden Retriever are worth looking for.

**Steps of a Convolutional Neural Network**

First Step is to break up an image into smaller pieces by selecting a width and height that define a filter. Filter looks at small pieces, or patches, of the image that are the same size as the filter. Then we slide the filter to focus on a different place of the image.

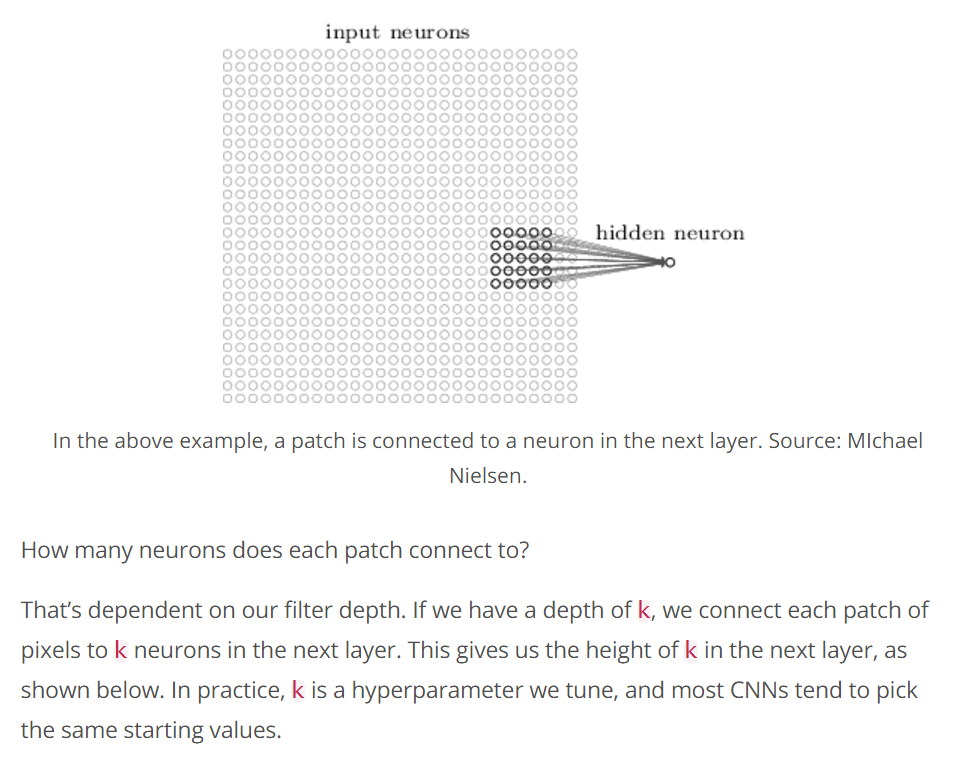
The amount of stride the filter has is a hyper parameter chosen by us. A large size stride reduces the number of total patches

An important point is that this filter is that it is grouping together adjacent pixels and treating them as collective.

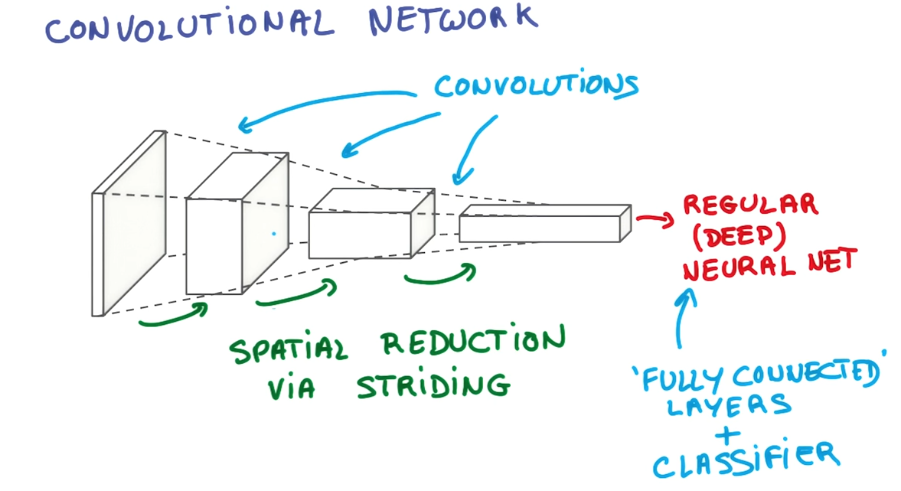
In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have special meaning.

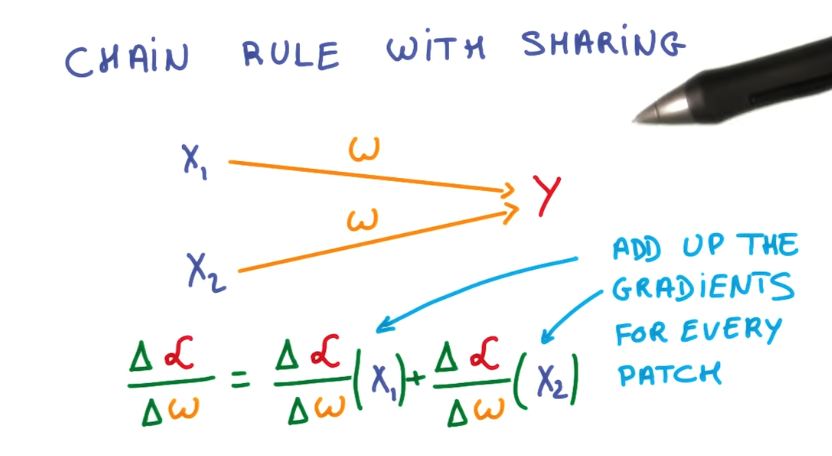
By taking advantage of this local structure, our CNN learns to classify local patterns, like shapes and objects, in an image.

It is common to have more than one filter. Different filter pick up different qualities of a patch (Ex. Color and Object shape). The number of filters in a convolution layer is called the **filter depth**.



The reason one patch can have multiple (k) neurons in the next layer is because a patch can have multiple interesting characteristics we want to capture, Like color and shape.





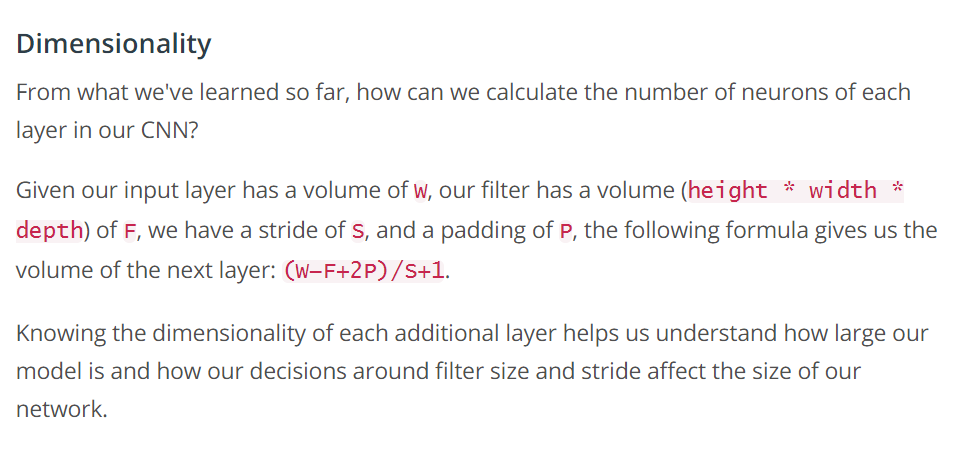
**Parameter Sharing:**

In order to find an object in an image we need to not care where in the image the object is. We want our Convolutional Neural Network to also possess this ability known as **translation invariance**.

How? Well a classification of a given patch in an image is determined by the weights and biases corresponding to that patch. So in order for an object in the top left patch to be classified in the same way as the same object in the bottom right patch, we need the weights and biases corresponding to those patches to be the same.

Therefore the weights and biases we learn for a given output layer are shared across all patches in a given output layer. As the depth of our filter increases, the number of weights and biases we have to learn still increases as the weights aren’t shared across different output channels.

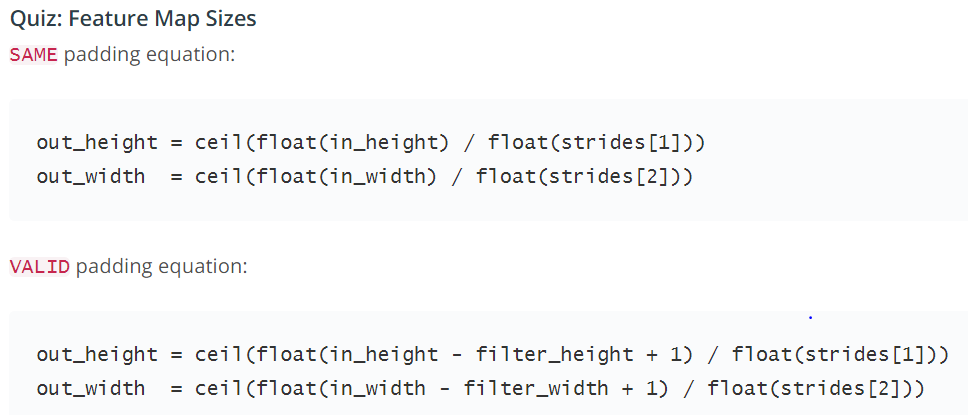
Another benefit to sharing parameters is we don’t need to learn new parameters for every single patch. Sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.



The depth of the output is equal to the number of filter you choose. Each layer of the filter looks for somewhat different things, like color, shape, etc. We choose the filter depth we wish.

Sizes:

Feature Map is: 1 Layer of Output if Convolution of Filter and input Image.



The weights and biases learned from each layer is shared with the other patches of the same layer.

How much Neurons are there?

Without parameter sharing, each neuron in the output layer must connect to each neuron in the filter. In addition, each neuron in the output layer must also connect to a single bias neuron.

Therefore: If our output Layer is 14x14x20, and our Filter is 8x8x3,

The Number of Parameters is: (8x8x3 + 1)x(14x14x20) = 756,560

With parameter Sharing however:

Every Neuron in the Output Layer shares its parameters with every other neuron in its same channel.

With parameter sharing, each neuron in an output channel shares its weights with every other neuron in that channel. So the number of parameters is equal to the number of neurons in the filter, plus a bias neuron, all multiplied by the number of channels in the output layer.

The Number of Parameters is: (8x8x3 + 1) \* (20) = 3,840 + 20 = 3,860

Which is much less. (3,840 weights and 20 biases)

**Improving a Covnett:**

**Pooling**.

Reducing the special extent of your feature maps.

This method reduces the stride, which increases the size of the output layer, and then combines local points in the feature map.

**Max pooling** takes the maximum of the neighbourhood and simplifies it to that max.

Good: Parameter-Free, and Often More Accurate

Bad: More Expensive, More Hyper Parameters (Pooling Size, Pooling Stride.)

**Average Pooling** takes the average and has that represent the neighborhood.

A typical Covnet Architecture.



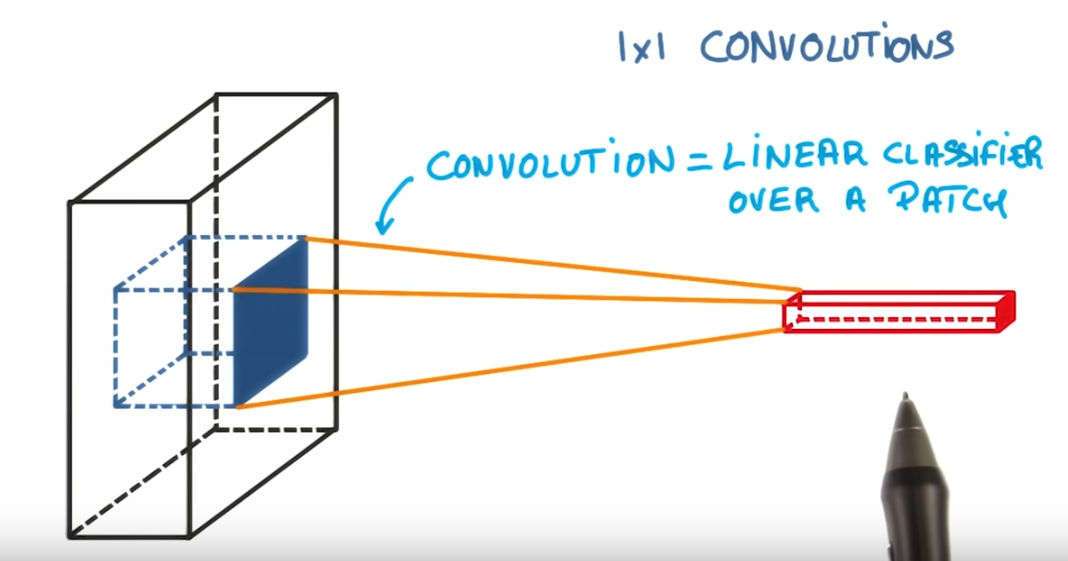
A Pooling Layer Generally is used to Decrease the Size of the Output, and to Prevent Overfitting.

Recently having a pooling layer has fallen out of favor due to Dropout being a much better regularizer, the loss of information from just keeping a Max, and Recent datasets are big and complex enough that we are more concerned with underfitting.

**1x1 Convolutions**

Why would we want 1x1 convolutions? It doesn’t look at a patch of pixels it only looks at one.

If we Look at a normal convolution with a patch:

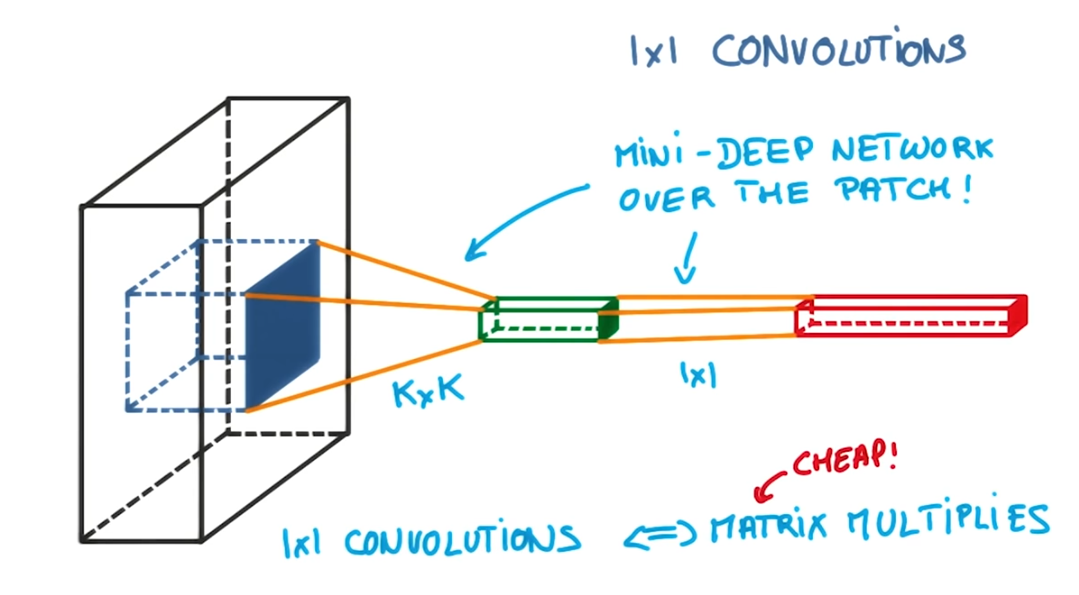


Convolution really is only a linear classifier.

If we add a 1x1 convolution in between, we have a mini-neural network running over the patch instead of a linear classifier.

It is a very inexpensive way to make our model deeper and have more parameters without changing the overall structure.

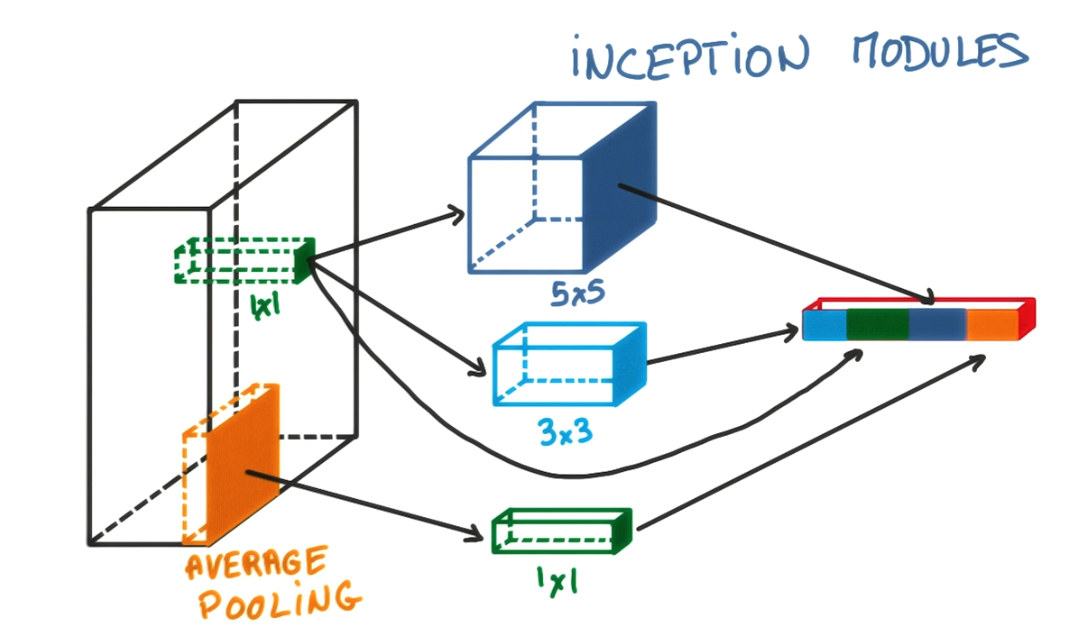
Easy to do as a 1x1 convolution is really just a matrix multiplication.



Pooling and 1x1 Convolutions are used in a general strategy to create Convolutional Neural Networks that are both smaller and better than covnets that simply use a series of convolutions. It is called Inception Modules.

**Inception Modules:**

The idea is that at each layer of your covnet, you make a choice, a pooling operation, a convolution (1x1 or 3x3 or 5x5). In reality why choose if they all benefit. Let us use them all and concatenate the outputs.



Looks complicated but you can choose these parameters so that the total number of parameters in your model is very small, yet the model performs better than if you had a simple convolution layer.