SuperDataScience

Sunday, 6 August 2017 4:29 pm

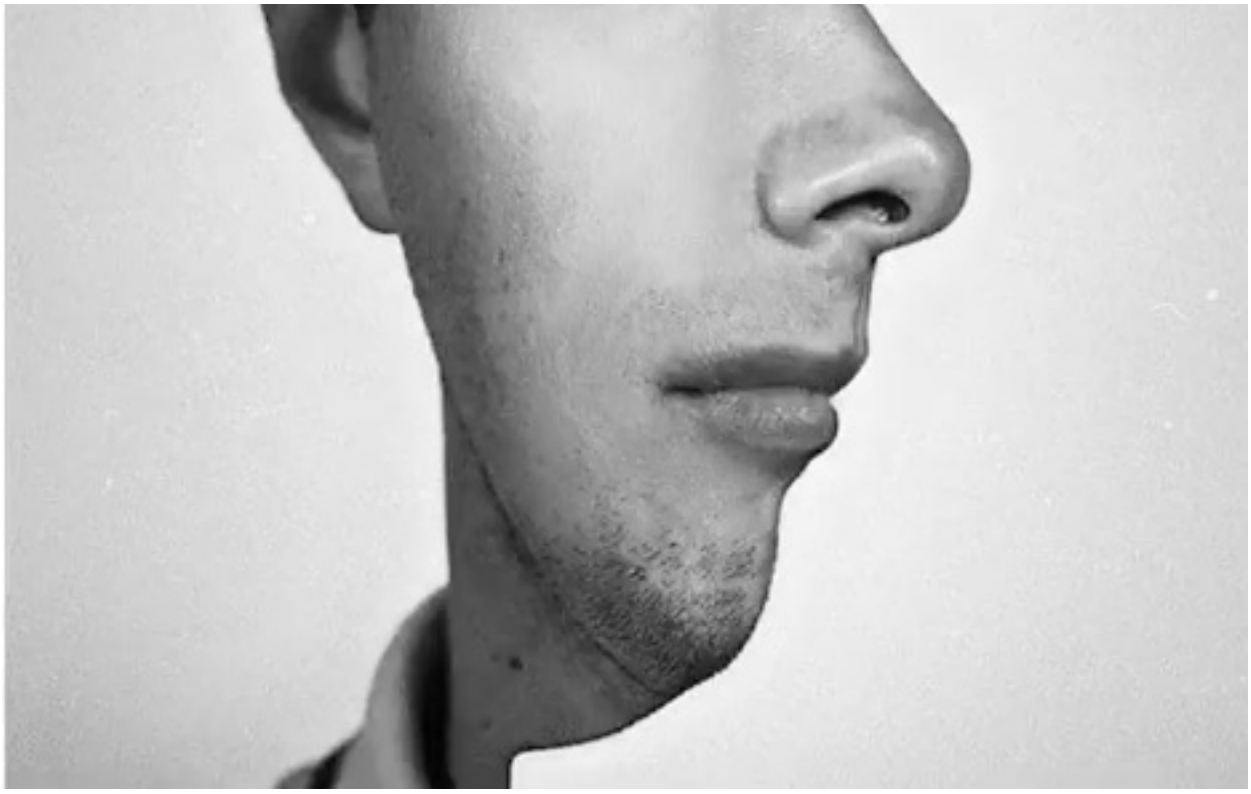
Part 2: Convolutional Neural Networks

Agenda

- What are convolutional neural networks
- Step 1 - Convolutional Operation
  - Feature detectors
  - Feature maps
- Step 1b - ReLU layer
  - Rectified linear unit
  - Why linearity is not good
  - Why we want more non-linearity
- Step 2 - Pooling
  - How pooling works
  - Max Pooling
  - Mean pooling & sub pooling
  - Example - visualisation interactive tool
- Step 3 - Flattening
  - How to go from a pooled layer to a flattened layer
- Step 4 - Full Connection
- Summary
- Extra: Softmax and cross-entropy

Convolutional Neural networks

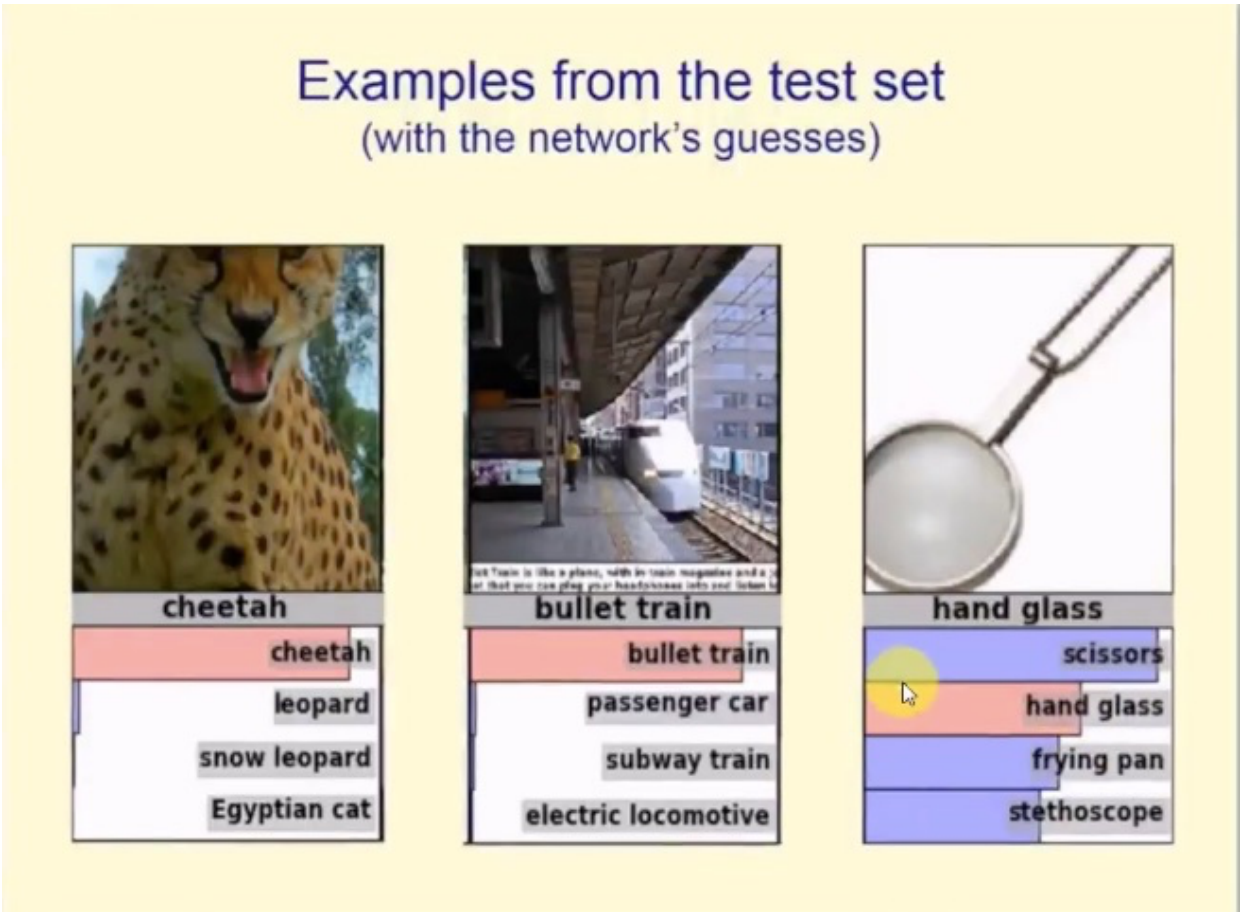
Look at this image



- We can see either someone looking straight ahead or looking to the right. It depends on what you look at.
- It proves that brains look for features. Depending on the features it sees, it categorises it.
- So if you look at certain features of the image, you can see someone looking to the right.
- If you look at the left side of the image, you see more features looking straight ahead, and so your brain classifies it as such. If you look at the right side of the image, you see more features that would classify it as rightward facing.

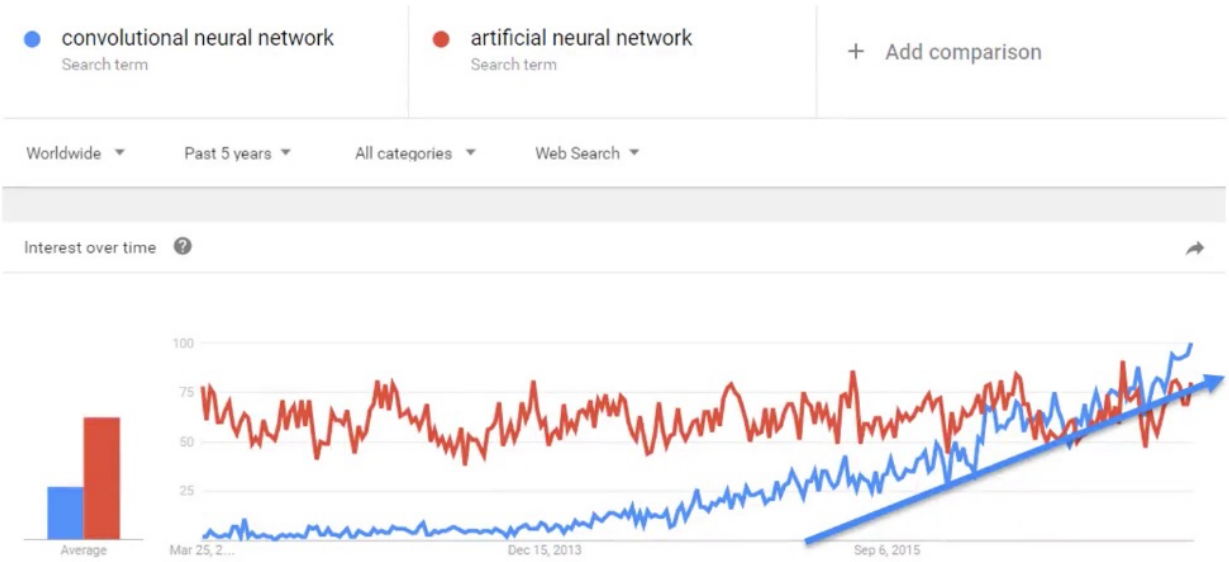
Look at this one

- Same deal. You can either see a young woman looking away, or an old woman looking down.
- Again, it depends on which features of the picture you look at. Once you have a cluster of features, your brain classifies the image one way or another



- This is how an algorithm classifies three images
- The actual label is at the top in grey. The correct answer is in pink. The other guesses are in blue
- We can see that the algorithm is quite sure of itself in the first two images, but less so in the third

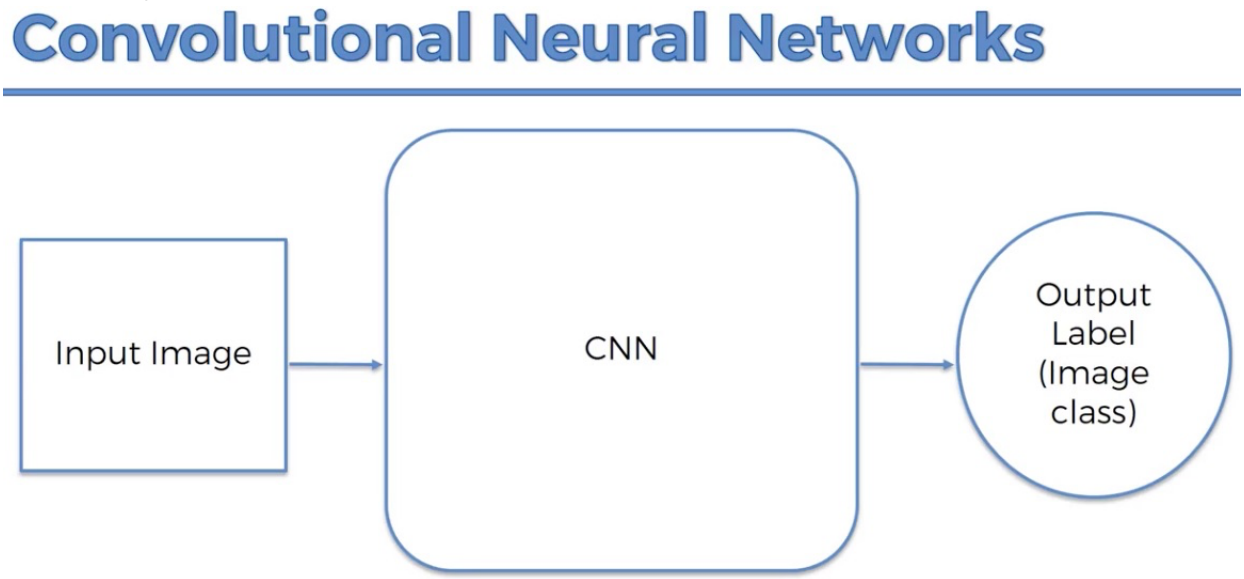
but less so in the third



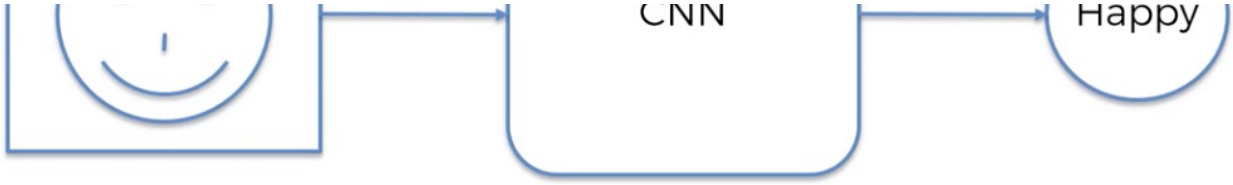
- This chart shows that the search term "convolutional neural network" has overtaken "artificial neural network" over time.
- Convolutional Neural Networks were invented by Yann LeCun. He was a student of Geoffrey Hinton, who invented Deep Learning and Artificial Neural networks. Yann is at Facebook. Geoffrey is at Google.

How CNNs work

- Here's a picture



- You have an input
- It goes through the CNN
- It comes out with a label, that tells you what the object is
- .
- Here's an example

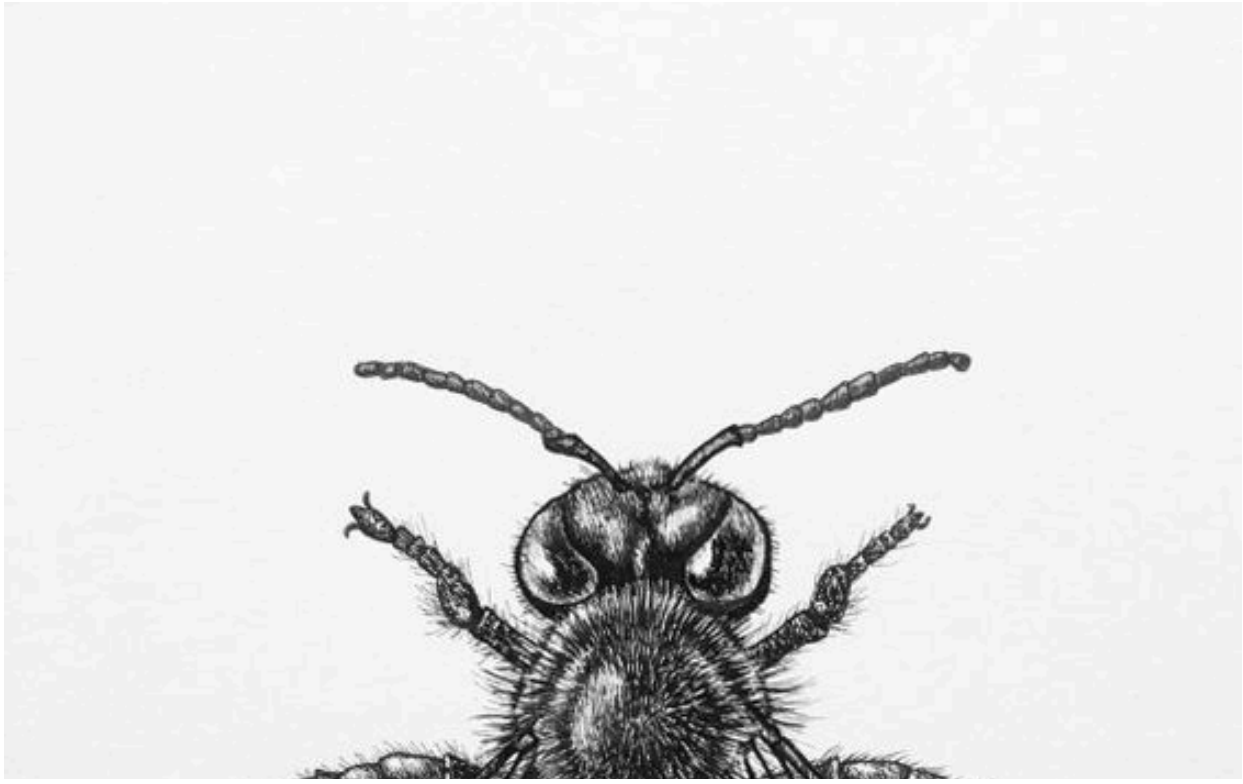


- Give it an actual face, and it'll tell you whether the person is happy, sad etc
- It will also give you a probability

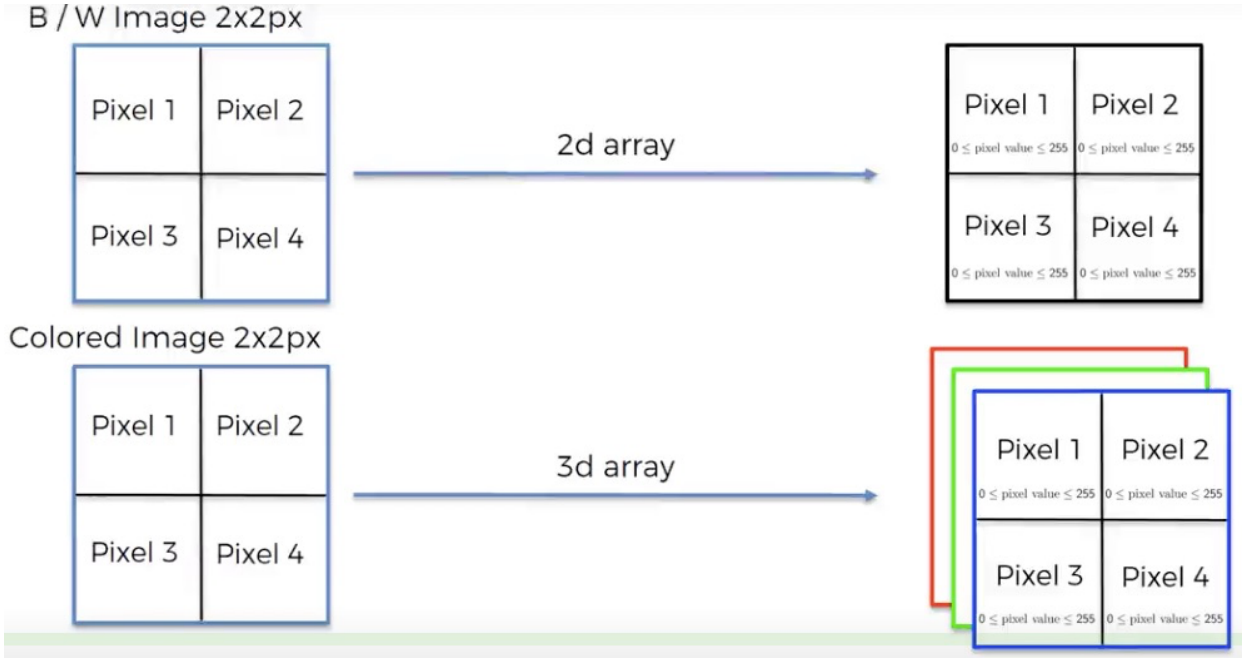
How does the CNN figure out what's in the picture?

- Greyscale vs Black & White vs Colour
- A picture is made up of lots of pixels.
- This is greyscale picture

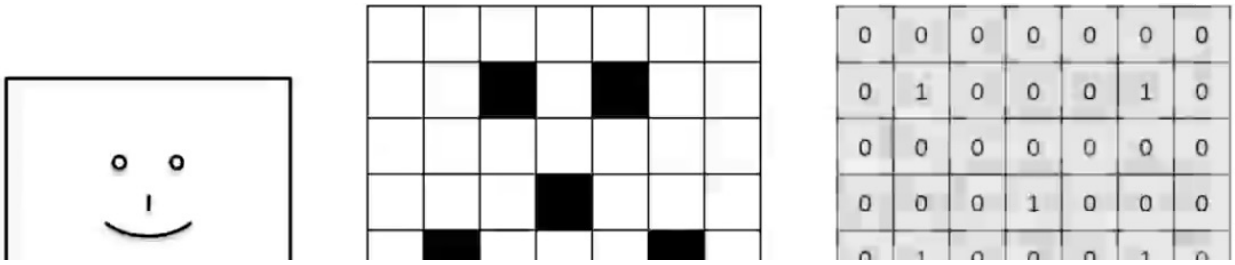
- Each pixel can have up to 256 different colours (labelled 0 to 255)
- That's not the same as a black and white picture, where each pixel is either 100% black or 100% white.



- A colour picture on the other hand, is a mixture of red, green and blue pixels, so has [256 x 256 x 256] possible colours
-



- .
- Let's take a simple example.
- Let's look at a black and white picture and see if we can identify its shape.





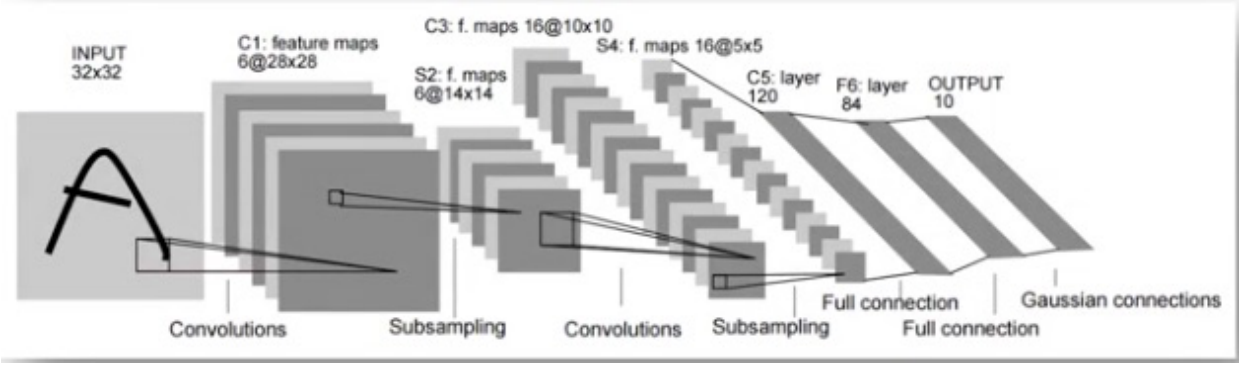
- This smiley face is black and white, so each pixel is either a 1 or 0.
- On the left (above) is the original picture. In the middle, we've zoomed in to what it looks like in terms of pixels. On the right is a picture of the underlying numbers.
- So how do we figure out if the smiley face is smiling? We use four steps to get to Convolutional Neural Networks

The four steps to Convolutional Neural Networks

- STEP 1: CONVOLUTION
- STEP 2: POOLING
- STEP 3: FLATTENING
- STEP 4: FULL CONNECTION
- .

Additional reading

- LeCun, Yann, 1998, "Gradient-Based Learning Applied to Document Recognition", http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf



- This is the famous picture from that paper. We will explain what it means below

CONVOLUTIONS

The Function

- I am legally obliged to tell you that the formula for convolutional neural networks, just in case you already have a maths degree.
- Honestly, you don't need to know it in this tutorial.

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

- The formula shows the combined integration of two functions. It shows how one function affects the other & modifies it
- Additional reading for anyone who wants to understand the maths behind what we're doing. (Don't read it until the end)
    - Introduction to Convolutional Neural Networks, by Jianxin Wu (2017)
    - http://cs.nju.edu.cn/wujx/paper/CNN.pdf

Converting numbers to understanding

- The image on the left is the smiley face, but with the black lines converted to 1s.
- The image on the right is called a "feature detector", but it might also be called a "kernel" or even a "filter". It shows the left hand side of the smile, as well as the nose. What we'll be doing is putting the filter over our main image to see if it contains .. er … the left hand side of a smile and a nose.

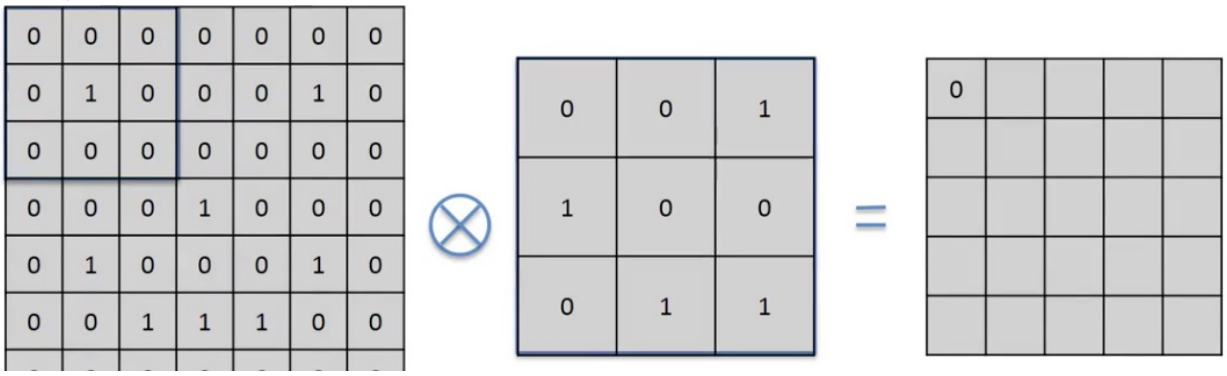
0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

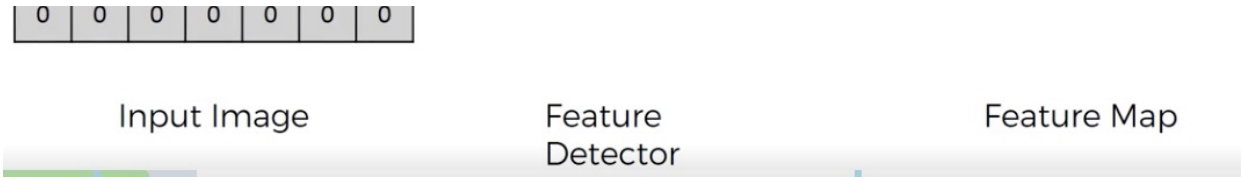
0	0	1
1	0	0
0	1	1

Input Image

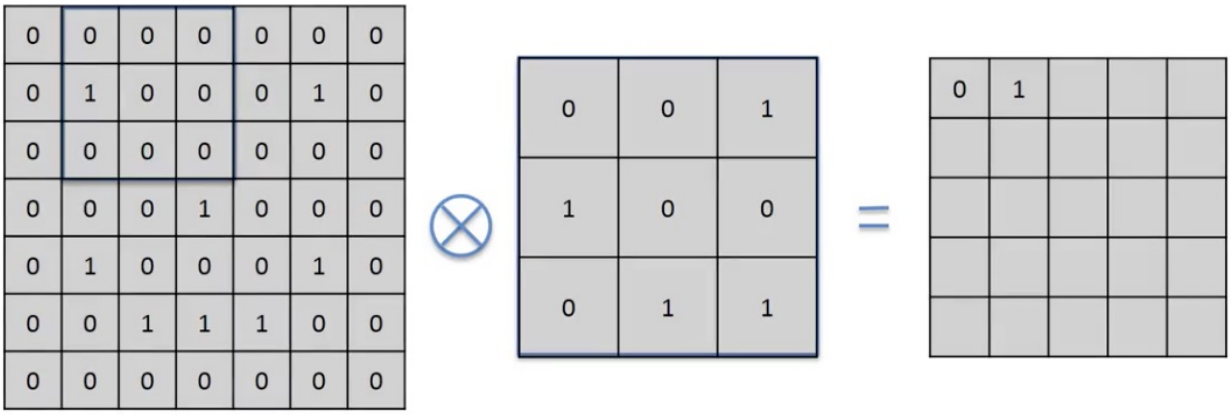
Feature Detector

- A feature detector is a smaller matrix, often a 3 x 3 matrix, but not necessarily
- You pull out the feature like this:

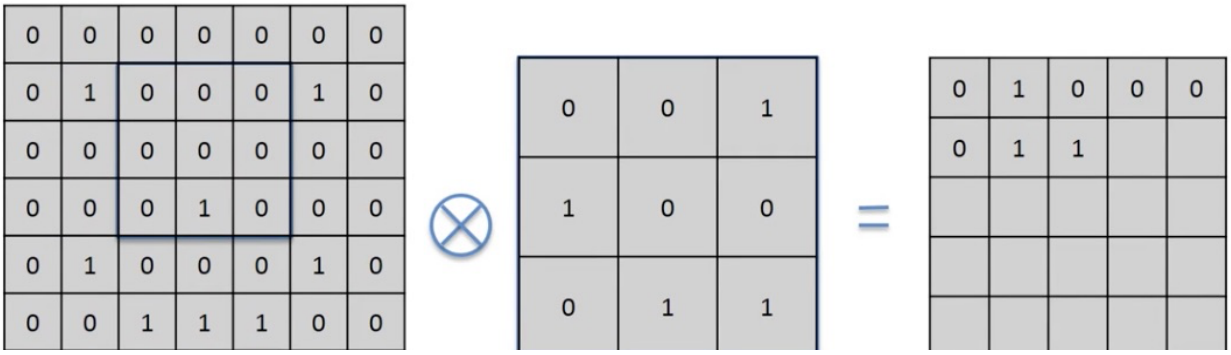




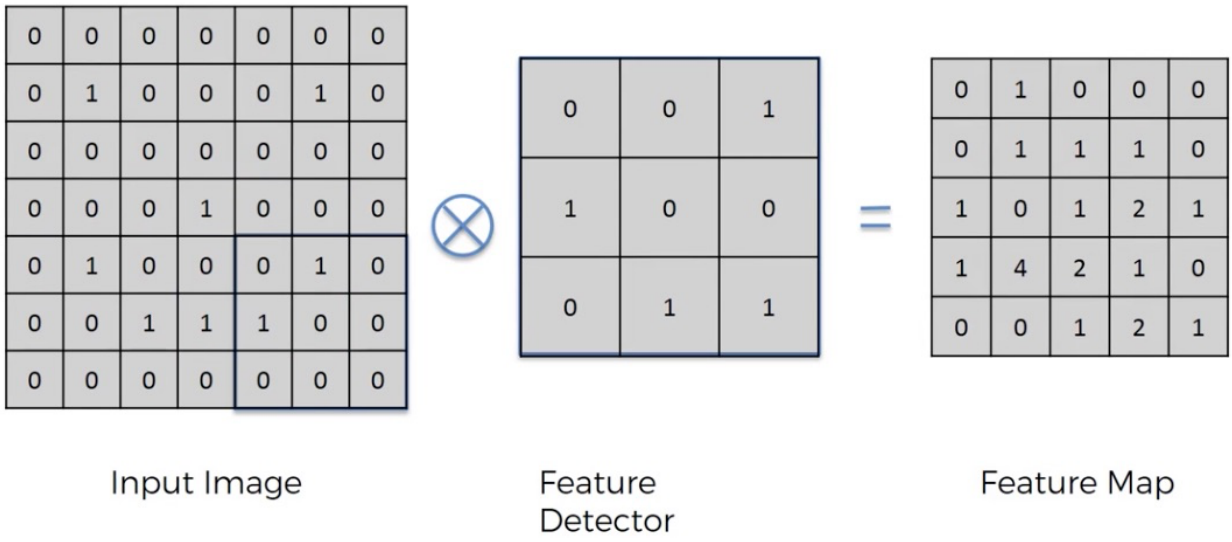
- So you have a section of the input image (the stuff in the blue square at the top left) and a template of what it's supposed to look like (middle). You multiply them together.
- The way you multiply two squares together is by multiplying each position of the 3x3 square on the left with the corresponding position of the 3x3 square in the middle, then add them up. So Input(1,1) x Feature(1,1) + Input(1,2) x Feature(1,2) … etc.
- In the example above, none of the elements match, so every multiplication gives you an answer of zero. So the sum of all nine multiplications is zero. Hence, 0 in the top left corner of the "feature map" (which is the result)
- .
- Now that we've done that, we need to move the apply the filter to some other area. We might move the filter to the fourth column, so that we don't repeat anything. But in this case, we'll move it along only one column. This means we have a "stride" of one.



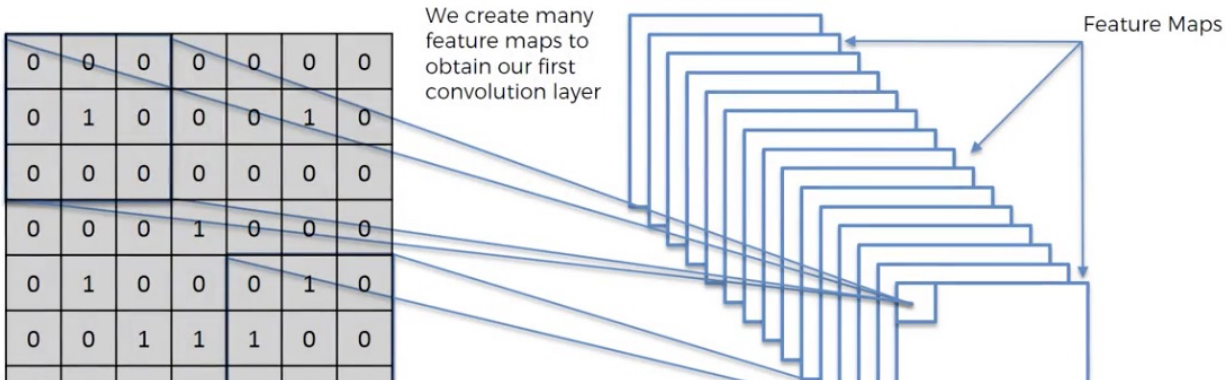
- In this second case, when we move it along one column, the 2nd row of the first column matches. It's the only cell that matches, so the sum of all multiplications is 1.
- Here's where we're at after several strides



- (Just a quick word on stride. In this example, we're only striding one column at a time, but typically, you stride two at a time)
- .
- By the time we've finished, the chart looks like this:



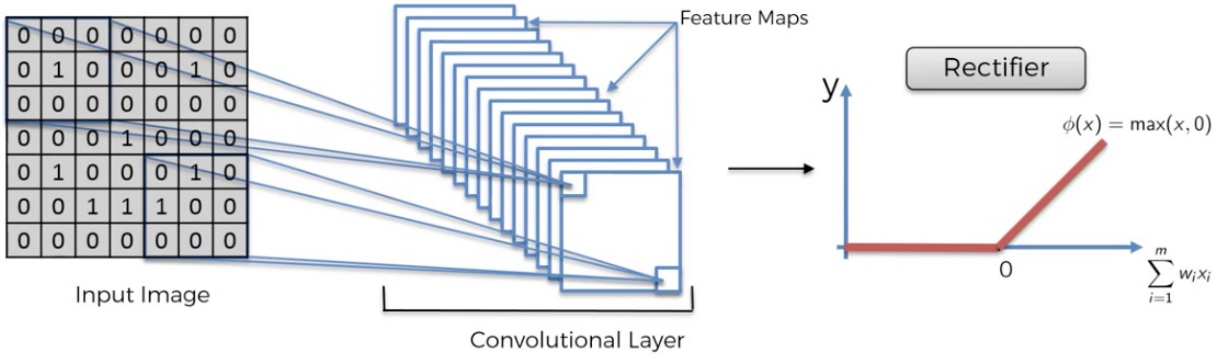
- The image on the right is called a "feature map", also called a "convolved feature", or an "activation map"
- We've reduced the size of the image. The greater the stride, the smaller the size of the feature map. The smaller the map, the faster it will be to process it.
- Do we lose information? Yes, sort of. But we don't care. What we're trying to do with a feature detector is detect the feature only. We don't care about where the feature _doesn't_ appear.
- For instance, in the feature map above, the closest match to the feature is near the bottom left hand corner, where there are four matches. That means that the feature we're looking for is most likely in that spot
- .
- But usually, we have lots of features we want to map. This will give us the following diagram:



- As you can see by the blue squares on the right, we have a STACK of features we're looking for. Each feature map is one feature we looked for in the same input image. After all, if we want multiple convolutions, we need to test lots of features.
- (Note: The computer will figure out which features to look for, so we don't have to!)

The ReLU Layer

- Rectified Linear Units
- This is where we add a rectifier function to our convolutional layer



- We do that in order to increase non-linearity in our network. Rectifiers do that
- Images are highly non-linear. e.g., the transitions between pixels is non-linear (there's borders, different colours etc)
- .
- Here's what happens if you try looking for features in a greyscale image, as opposed to a black and white image.
- This is the original image.

Image Source: http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf

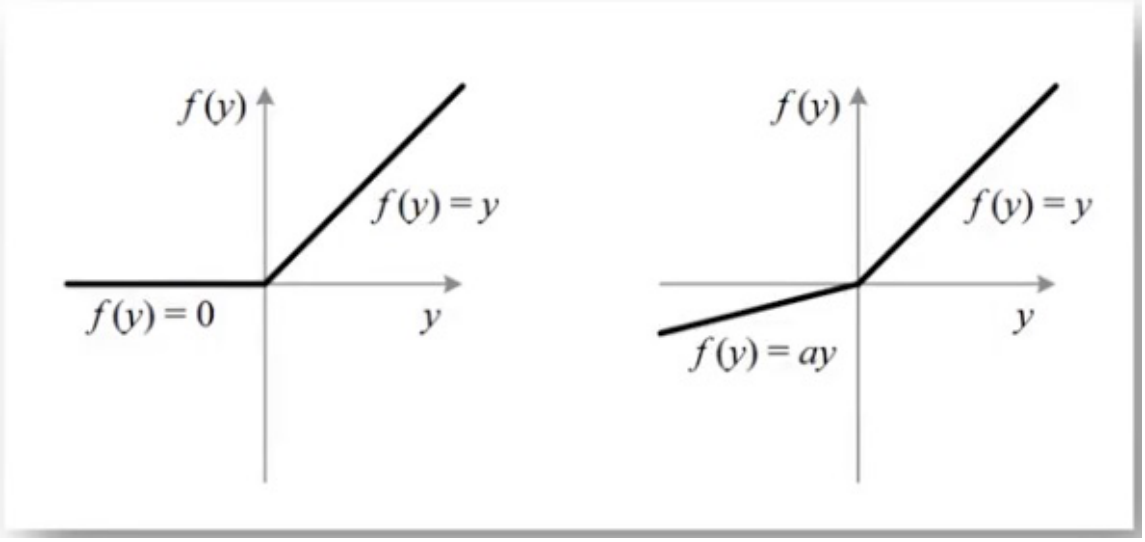
- If you add a feature detector, you get this, where black is a negative value, and white is a positive value. (You get this when you have lots of values for each pixel, e.g. 255 shades of grey instead of just 1 for white & 0 for black. This is because you can have negative values for particular shades. Hence, after running one of those through the filter, you can end up with this:

Image Source: http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf

- It's a bit blurry, right? This is because of all the shades of black where the value is below zero.
- What we really need to do is to make anything below zero one colour, and then have shades of white for positive values. That makes the building shapes easier to see.

Only non-negative values

- like this for instance
- The advantage of putting in a rectifier is that it allows you to concentrate on the important details - the positive values in this case which we've coloured white.
- Additional reading: "Understanding Convolutional Neural Networks with a Mathematical Model" by CC Jay Kuo (2016) https://arxiv.org/pdf/1609.04112.pdf
- He answers two questions. We only have to look at one.
- Additional reading: "Delving Deep into Rectifiers: Surpassing Human Level Performance on ImageNet Classification" by Kaiming He (2015). https://arxiv.org/pdf/1502.01852.pdf
    - The propose a different type of rectified linear unit function, called a "parametric rectified linear unit function"



    - They say it gives better results. Hurrah.

STEP 2: MAX POOLING

What is pooling and why do we need it?

- If you have lots of pictures of (what we know is) the same thing, we need to let the computer know that it is the same thing.



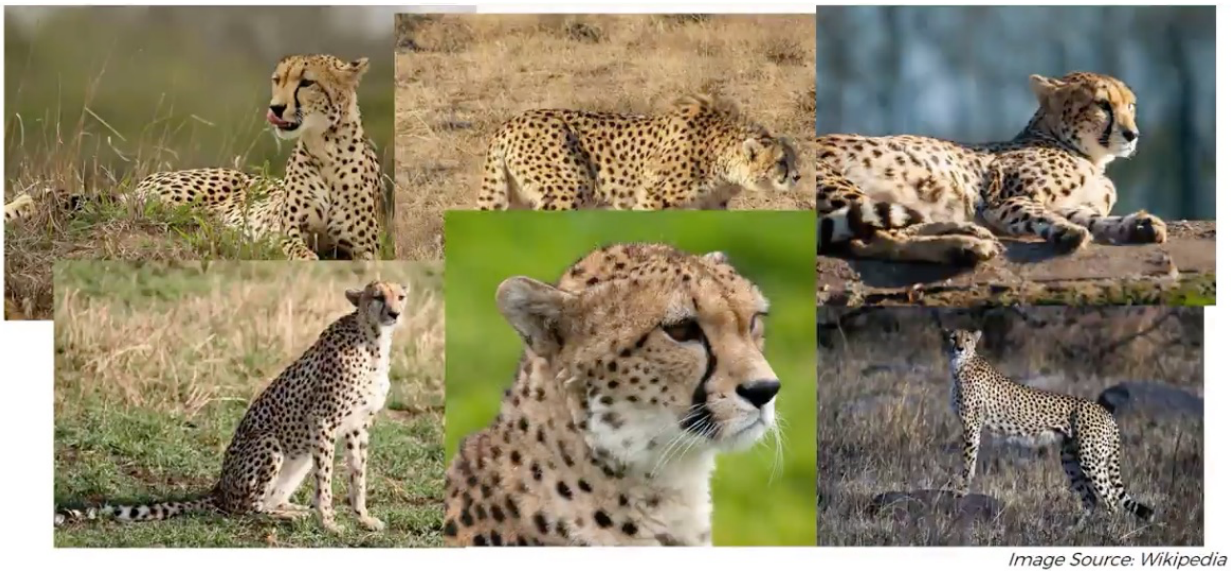
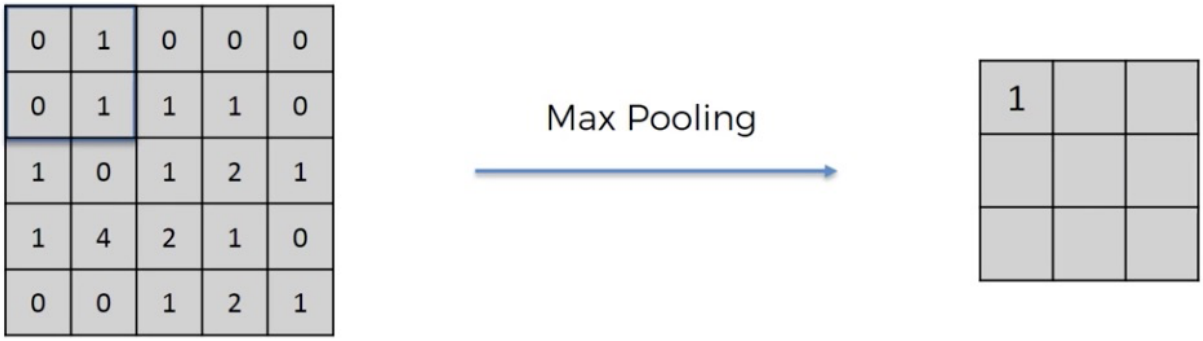


Image Source: Wikipedia

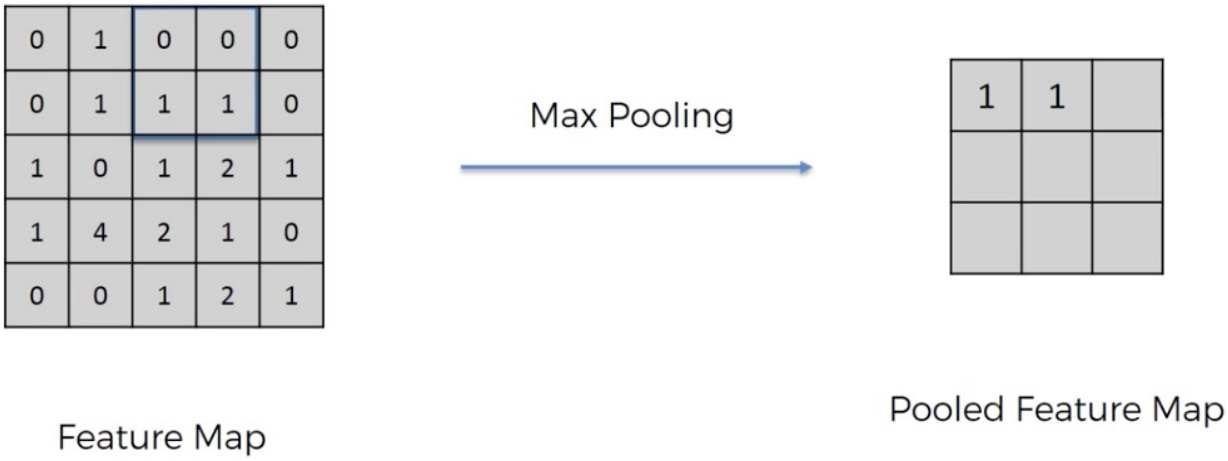
- All of these are pictures of cheetahs. But if you look at the underlying pixels, they all look different.
- We want the neural network to realise they're all the same sort of picture. But we don't want to have to sit there labelling everything, like a putz.
- We need … a pool
- What we need is for the neural network to recognise "spatial invariance". That is, they need to understand that an object is an object, regardless of whether it's tilted to the side, or closer or further away, or sitting against a different background etc.

How we do max pooling

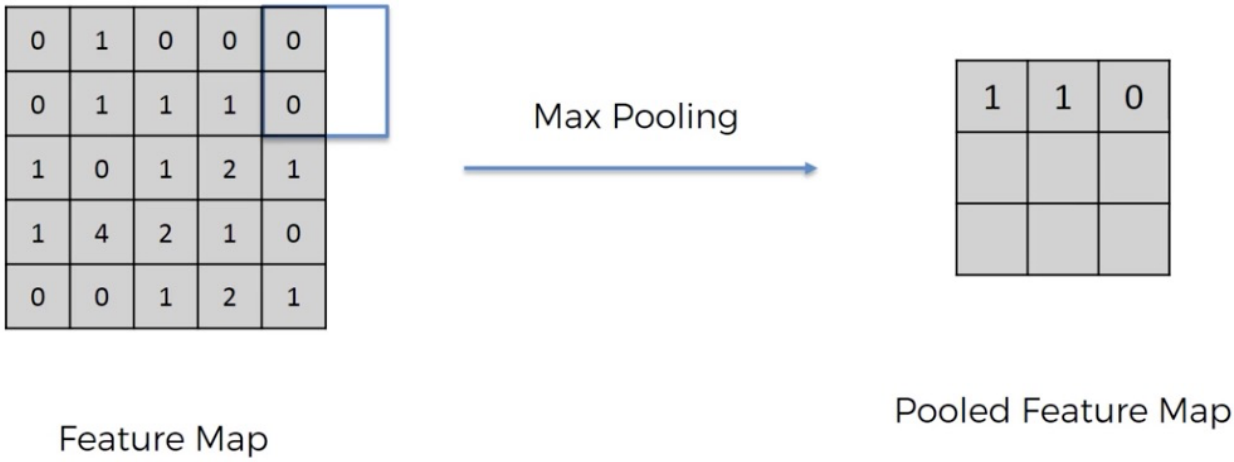


- Take our feature map, and draw a box around, say, 2 x 2.
- Find the maximum value in that box, and transfer it to the "pooled feature map" on the right.
- (Note, choosing the maximum value gives you max pooling. Choosing the average value would give you mean pooling. There are several types of pooling you can do.)
-

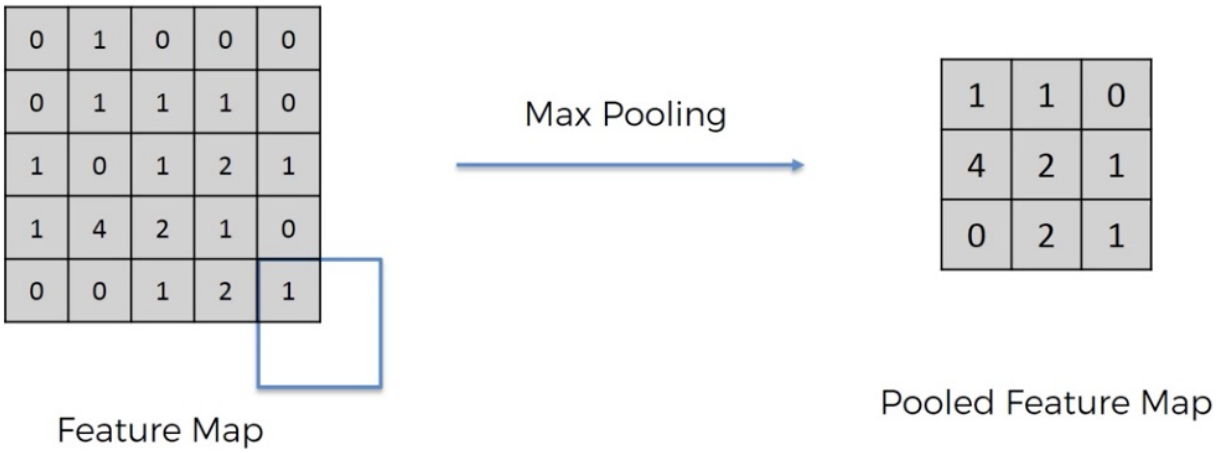
- .
- Let's set a stride of 2 units for reasons that will become apparent later.



- Again, the maximum in the feature map sub box (above) is 1, so we put a 1 in the Pooled Feature Map on the right.
- .
- You can do your stride of 2 again, even if you run out of numbers



- Now we can repeat all the way along until we finish our feature map:



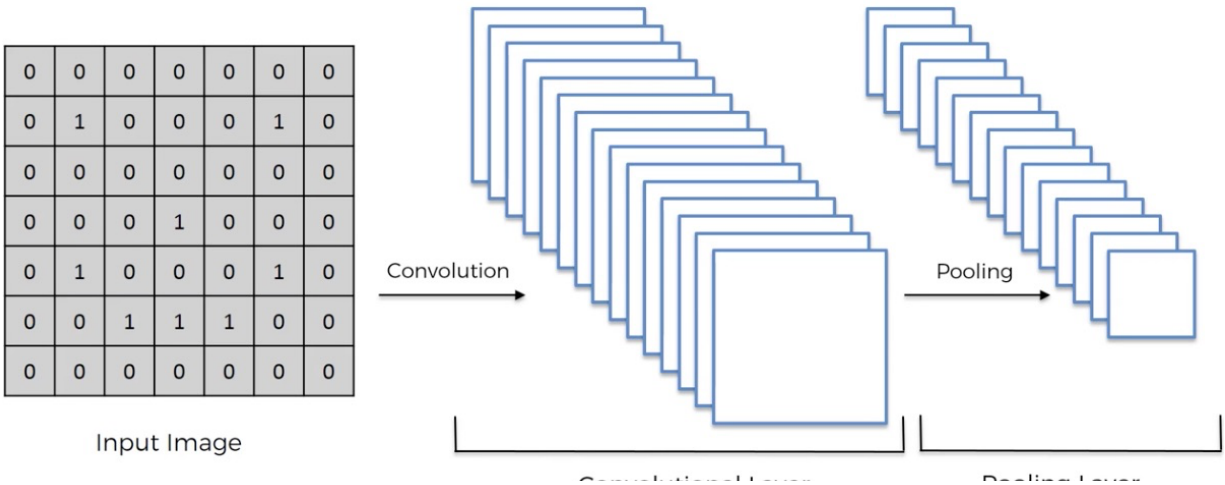
What's the result?

- The large numbers are preserved. The large number represents the place where our feature matched the best.
- But by pooling the features, we get rid of the unnecessary values (not a feature).

- The result is that the features are still in the pooled feature map, but we answer the question “is the feature in this general sort of area”, rather than “is the feature in a particular small area.



- For instance, if one feature we looked for was the “tears” next to the eyes of the cheetah (those black lines on either side of its nose), then we’d find them, regardless of whether they were tilted or not.
- We preserve whether the features are there or not.
- We reduce the amount of data we have by 75%.
- We also account for distortions - the cheetah that’s turned around on its axis will be in the same spot on the small grid.
- We also reduce our parameters that we have to test, which prevents overfitting!
- .
- The next questions are: Why a stride of 2? Why max pooling? Why a size of 2x2 pixels?
- Additional Reading “Evaluation of Pooling Operations in Convolutional Architectures for Object recognition” by Dominik Scherer et al (2010)
- http://ais.uni-bonn.de/papers/icann2010_maxpool.pdf
- They talk about “sub-sampling”, which is taking the average of the pool.
- .
- Summary



- .

Example

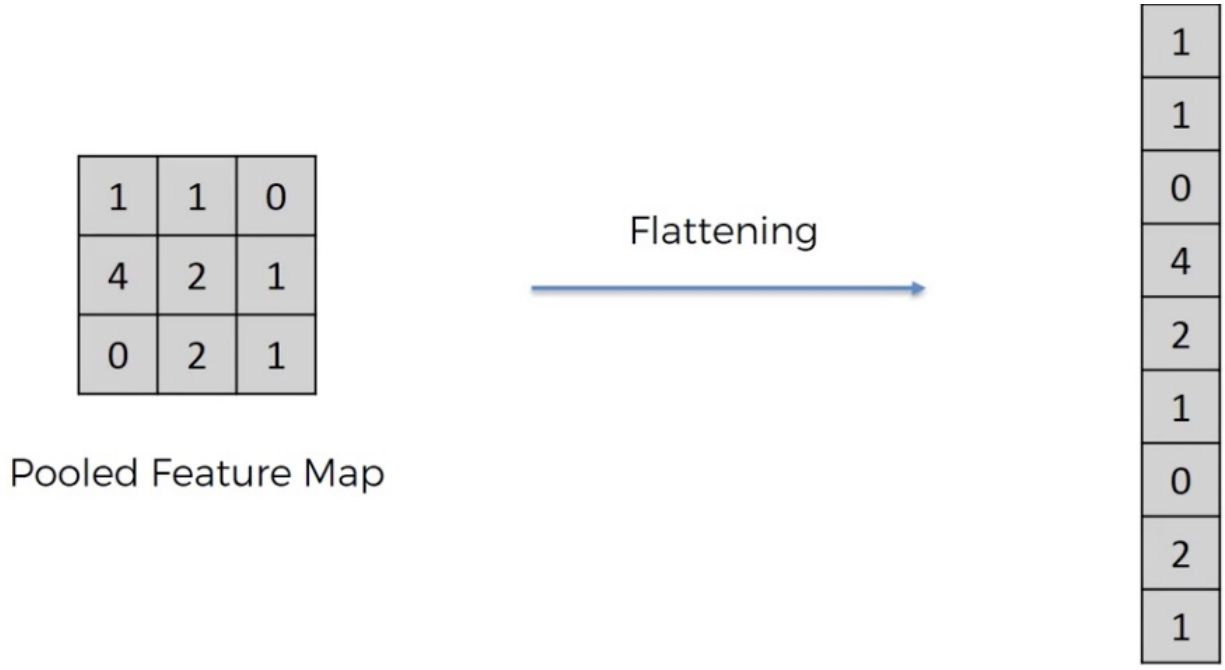
Image Source: scs.ryerson.ca/~aharley/vis/conv/flat.html

- If you look on the left of the screen, it tells you what it's doing at each step
- Input layer - You draw a number (5 in this case)
- Convolution layer - It applies a feature detector, to see if the features that are supposed to be in a number are indeed in it.
    - Here's picture showing how you can hover over the convolution layer and see what pixels were used to map it

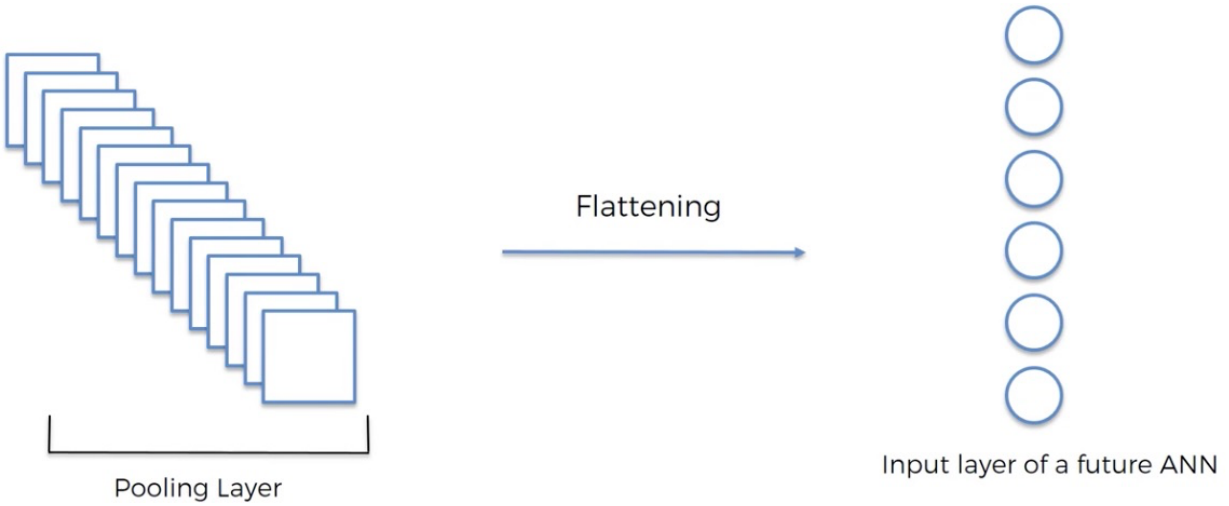
- Max Pooling layer - It makes the convolutional layer smaller, to save data, reduce noise, preserve features, and account for distortions
- Several other layers we haven't talked about

STEP 3: FLATTENING

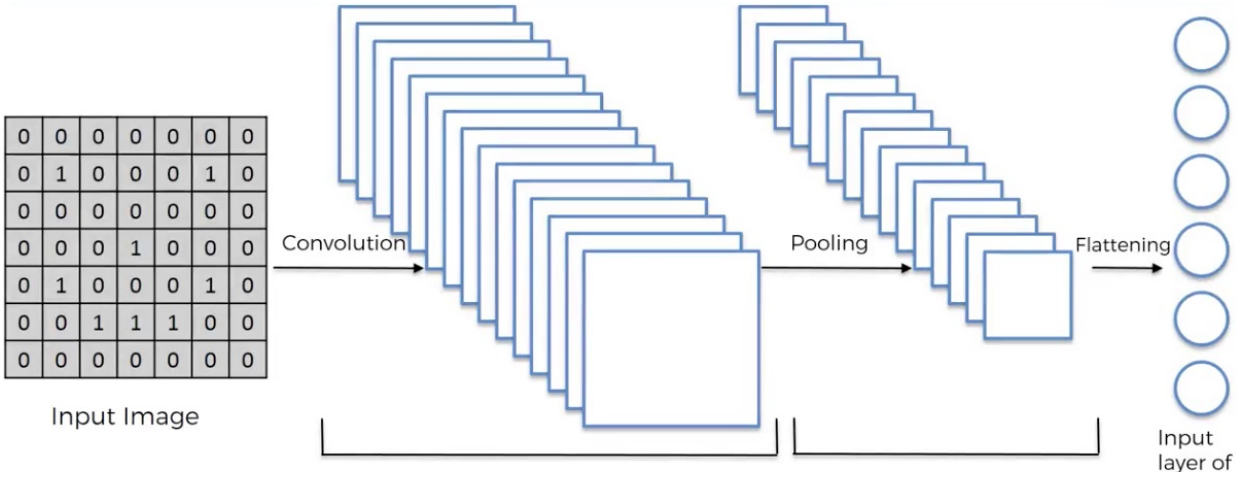
- This is where we take your Pooled feature map and turn it into one long column



- The reason you do this is so that you can put the flat vector into an artificial neural network.
- Each pooled feature map / vector represents one convolution, remember, so we'll start with a whole stack of pools

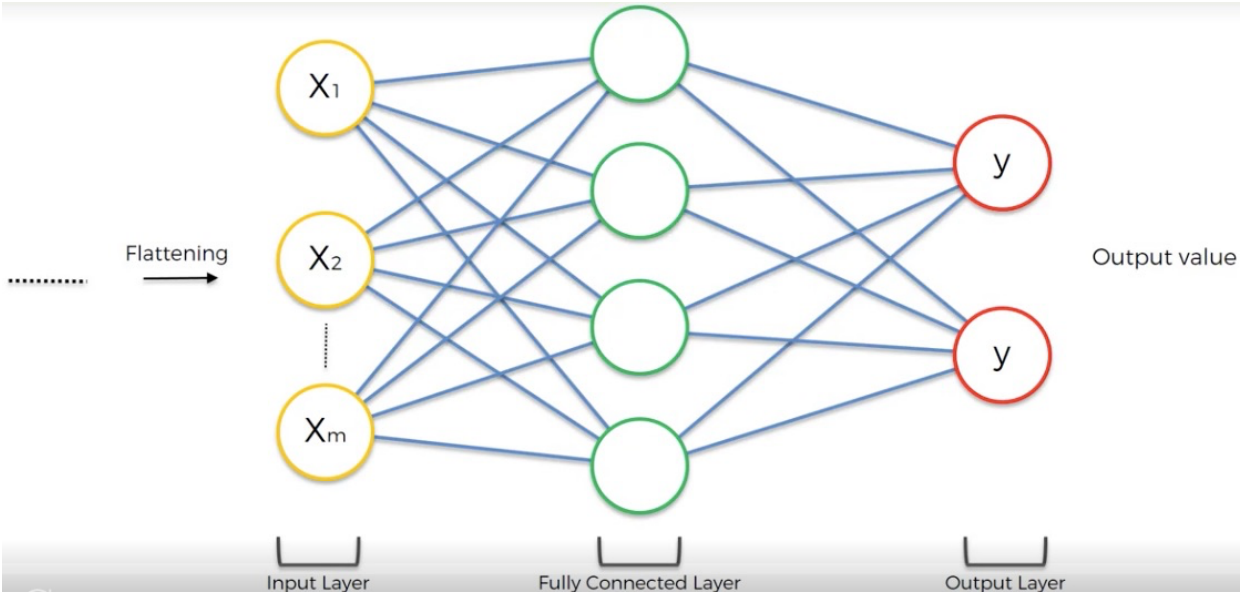


- Summary: We now have four steps

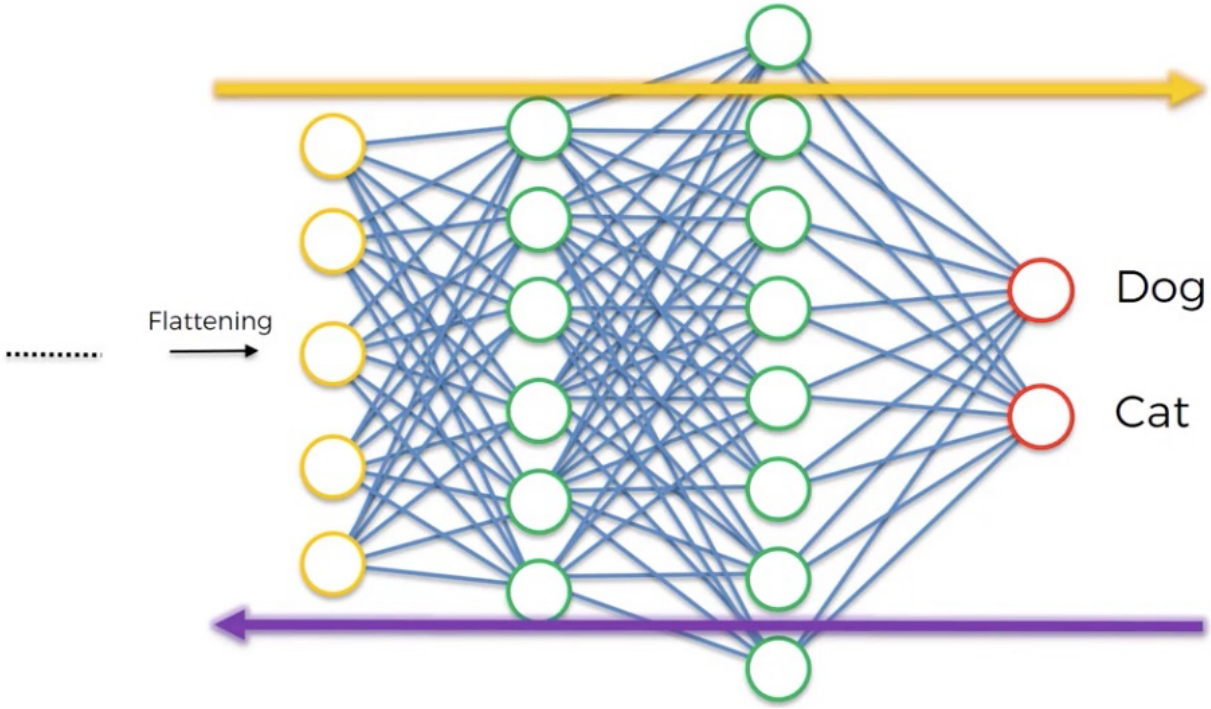


STEP 4: FULL CONNECTION

- This is where we feed the flattened layer into the front of an artificial neural network

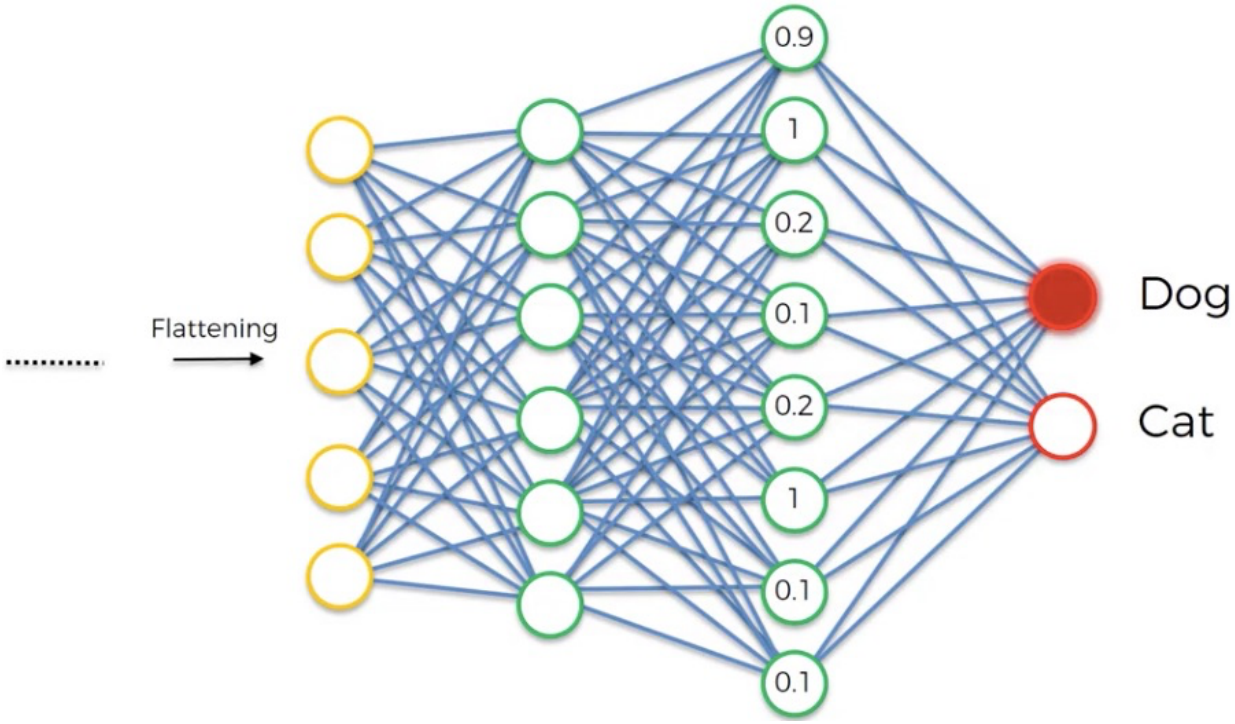


- Note: The hidden layers here are called "Fully Connected Layers"
- .
- For example, let's take an example where we're predicting whether something is a dog or a cat

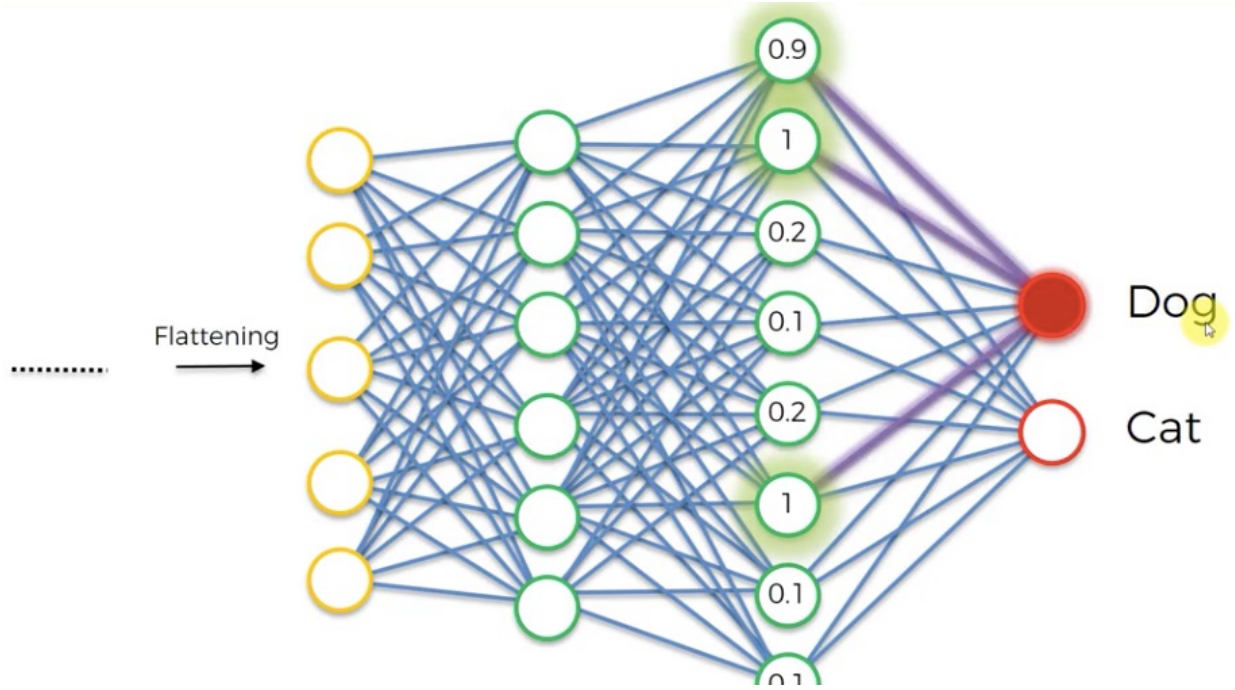


- Why do we have two outputs? Usually we have only one
- The answer is that this is a classification problem. We could say "Cat" vs "Not a cat", but for the purposes of education, let's make it two different types of animal, because that will allow us to add a third type of animal later.

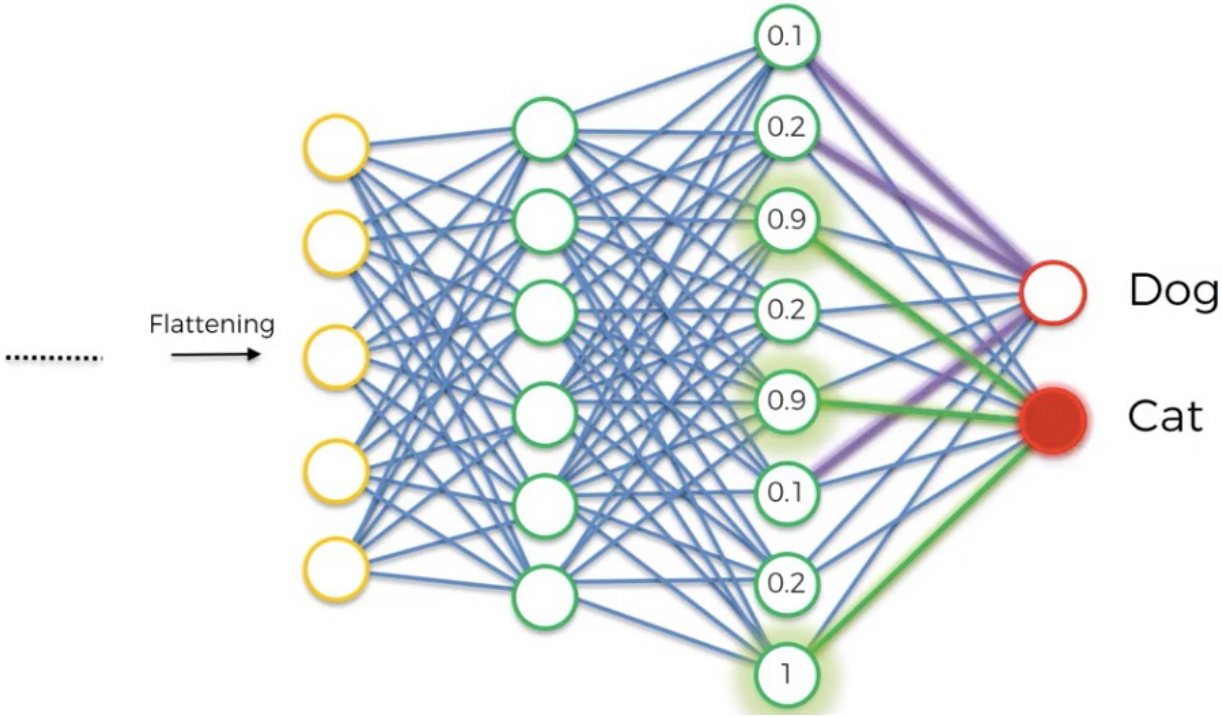
- The clever thing about Convolutional neural networks is that when you backpropogate, you backpropogate all the way back to the start when you broke the original image up into features, and pooled them with 3x3 grids. The gradient descent changes it so you can find the optimal number of grid sizes, optimal features to look for, optimal stride size etc.
- .
- Let's give an example of how important each neuron is to the outcome. Let's take the "dog" output. Different neurons would give greater confidence about whether it's a dog. Eg, the sound it makes. If it's a bark, it's much more likely to be a dog, so sound is an important predictor. Let's show how important all the neurons are to an outcome:



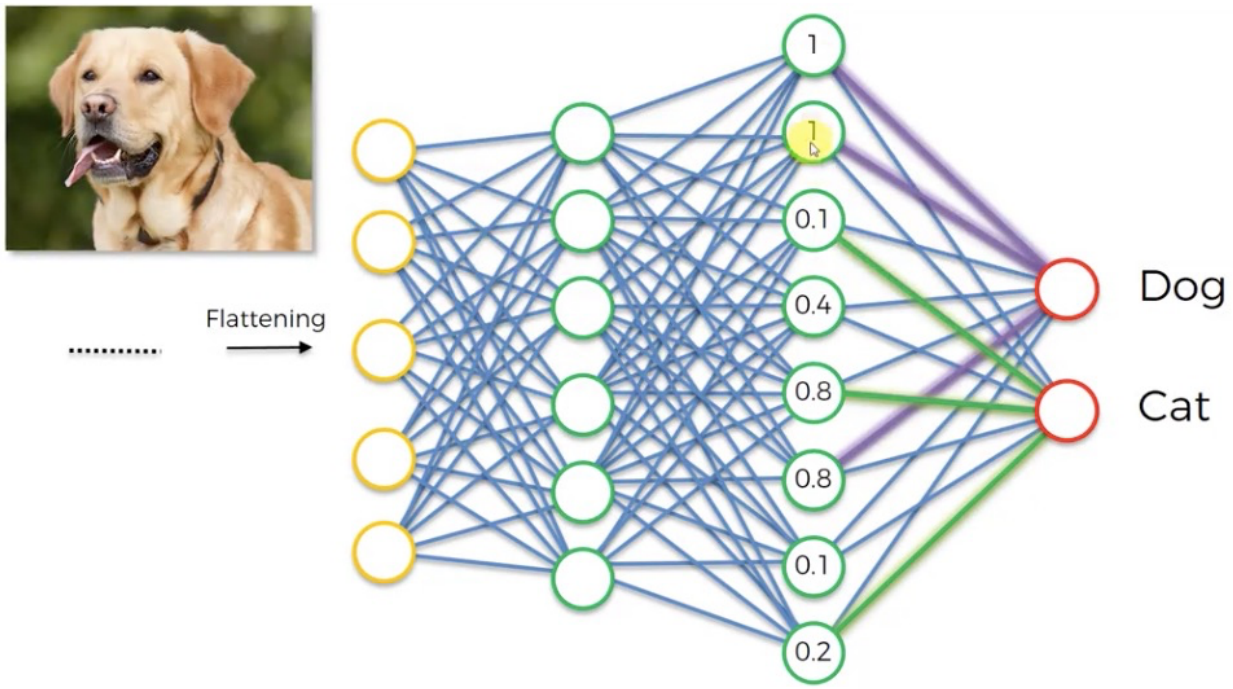
- It's lit up that it's a dog. But why?



- It's because the Dog neuron knows that when the previous layer's neurons 1, 2 & 6 light up, that's an indication for the dog neuron that it's a dog. The Dog neuron is ignoring the other neurons in the previous layer
- Bear in mind that by the time it gets to the 2nd last layer, the data is unrecognisable from the original picture.
- .
- Now let's look at the cat neuron



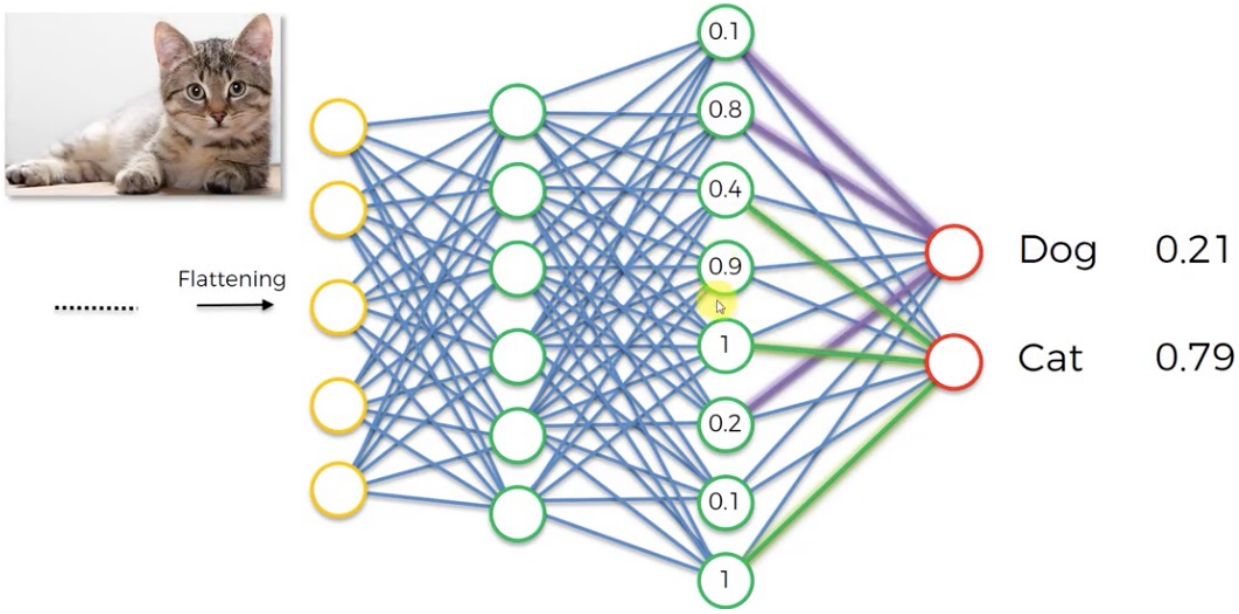
- These other neurons are particularly important to identify cats.
- .
- So let's actually send a picture through to the last layer



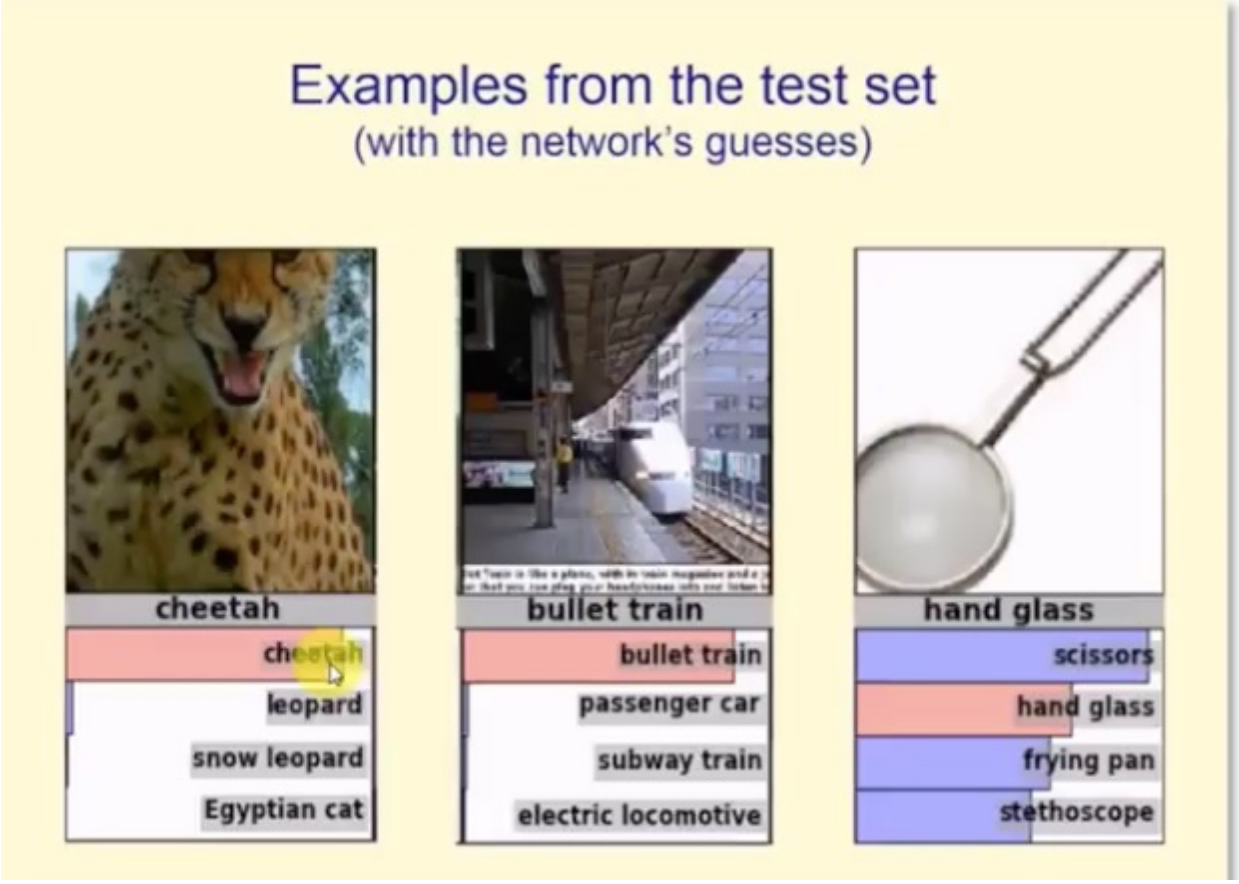
- Now we can see that the dog's neurons in purple are all very high, while only

one of the cat's important neurons are high.

- .
- Similarly, here's a cat

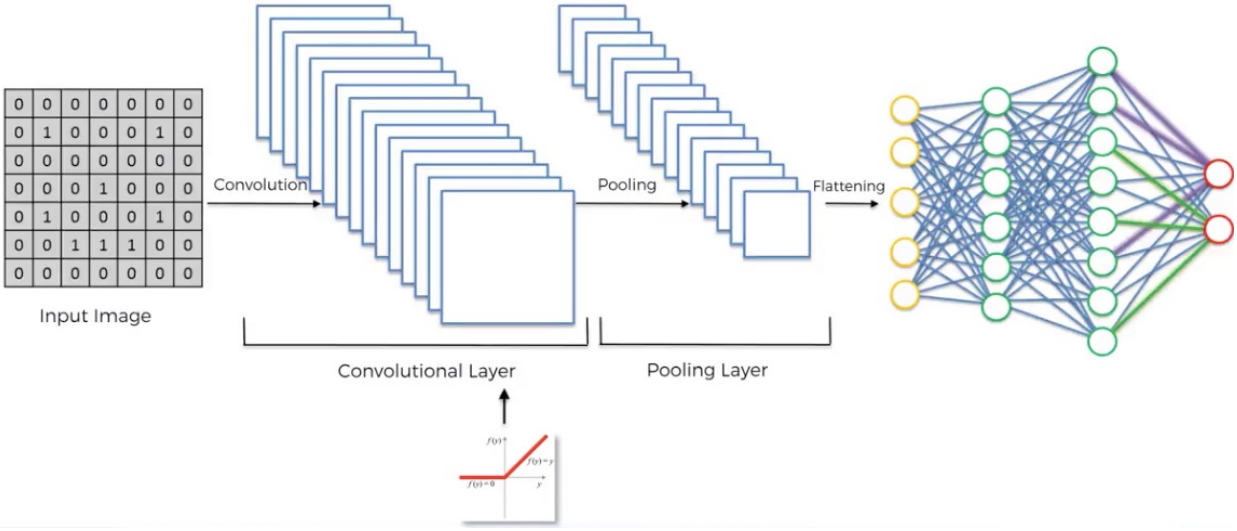


- In this case, the cat's neurons are high, totalling a probability of 0.79, whereas the dog's are low, with 0.21.
- The fully connected layer gets to vote with a probability. The dog and cat neurons assign weights to those neurons to weight their vote, which means the whole neural network comes to a decision about whether it's a cat or a dog.
- That's how you get an image like this:



SUMMARY

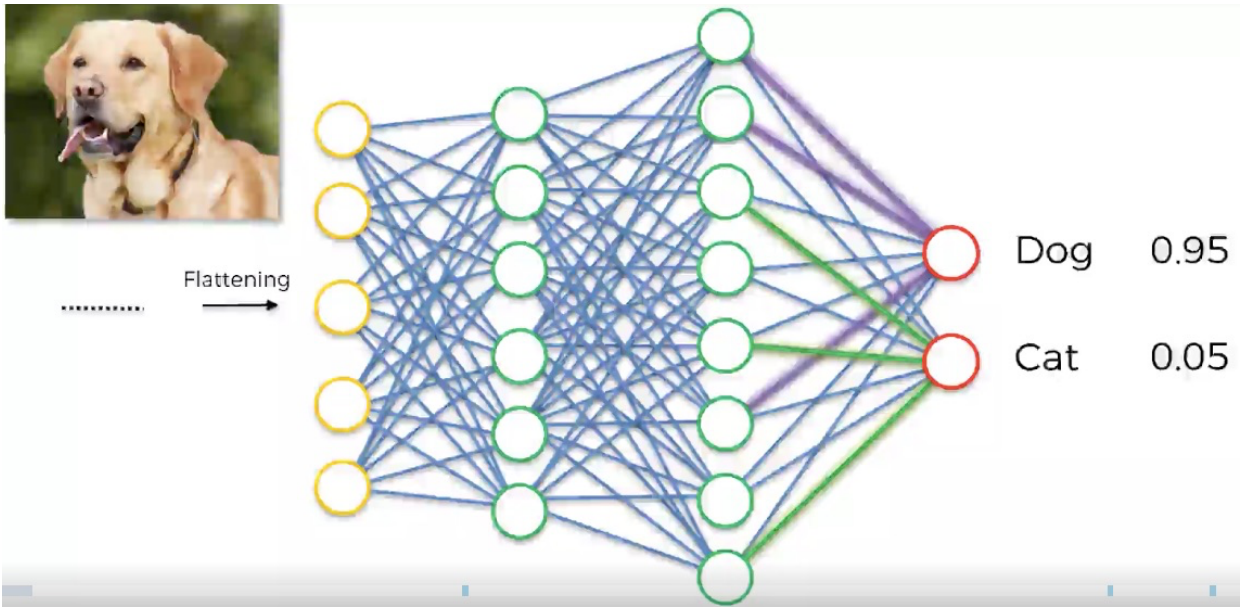
- This is everything all connected together



- Start off with a raw image, which is made up of numbers
- Then we apply a series of feature detectors, which takes a group of pixels and sees to what extent that feature is apparent in a small area. Each feature detector represents one convolutional layer
- Then, for each convolutional layer, we pool our results. That means, we break each convolutional layer into small squares, and look for the maximum value in each. This gives us a pooling layer where we just detect where the feature is, at a coarser level that prevents overfitting.
- Finally, we flatten the resulting pool, and pass it in as inputs into the ANN.
- The second to last layer then passes a series of probabilities between 0 and 1 to the final neuron, which has learned which penultimate neurons are most reliable in detecting the outcome it wants.
- The output layer uses these to estimate how sure it is that the picture is of itself. The neural network compares the output layers and decides which one is more likely.
- Additional reading: "The 9 Deep Learning Papers You Need To Know About (Understanding Deep Learning Part 3)" Adit Deshpande (2016) https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html
- Gives an overview of the papers. Lots of new stuff. Good for after the practical tutorials

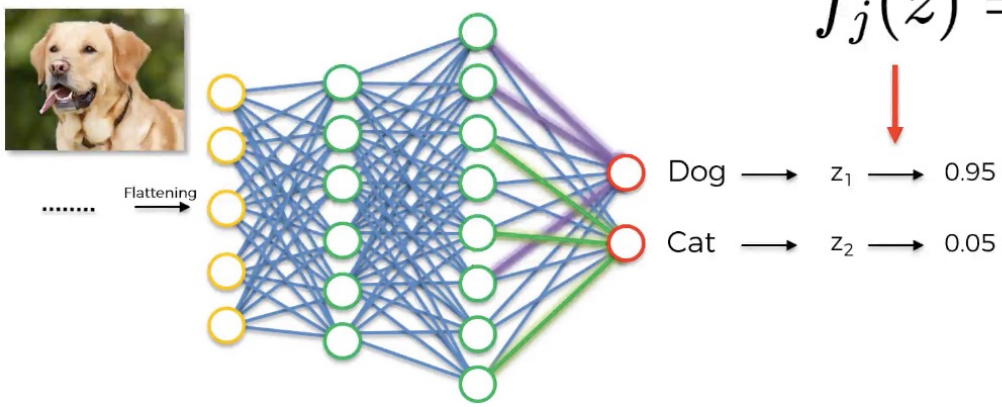
SoftMax and Cross Entropy

- Here's the neural network we built



- It says that the probability of the picture being a dog is 0.95, and the probability of it being a cat is 0.05.
- The question is: How come the probabilities of the output layer add up to 1?
- Eg, the Dog neuron calculates the probability of it being a dog independently of the cat neuron. Conceivably, there are neurons in the penultimate layer that would give a good indication of _either_ dog or cat (e.g., "HasTail == TRUE"). So why does Dog + Cat = 1, and not some other number?
- .
- The answer is, it doesn't normally add up to 1, unless we apply a softmax function first.

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$



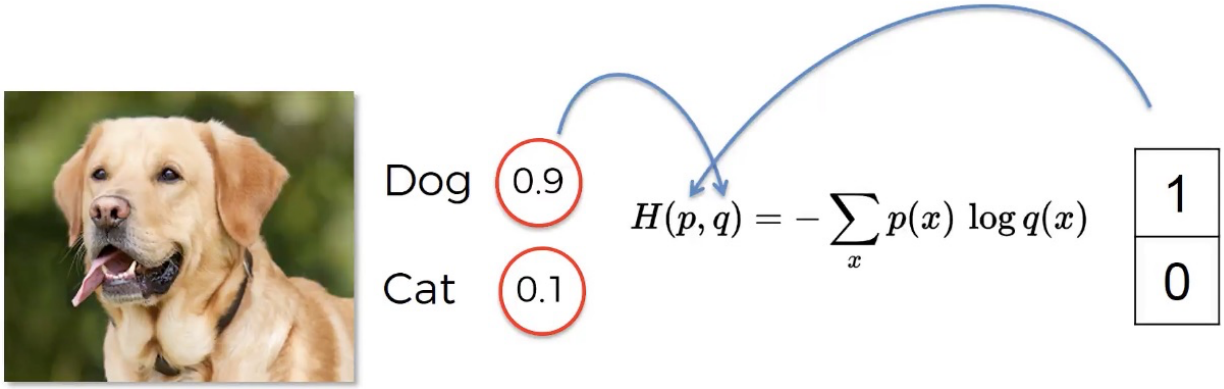
- The SoftMax function is a generalised form of the logistic regression.
- It "squashes" the results of Dog or Cat, and turns them into a probability function that add up to 1
- .
- It goes hand in hand with the "cross entropy function"

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\quad}\right)$$

$$\left(\sum_j e^{f_j}\right)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$

- (The function at the top is the standard function. The one at the bottom is the one we'll use, but they give the same results)
- A cross entropy function is a type of error function, kind of like the mean squared error function in linear regression, except it's logistic
- Here's how it works in practice



- So we put a dog into our network, during training. We know that it's a dog (hence the 1 on the right hand side). We calculate that the training set predicts a 90%
- Let's get a step by step example

- Here we have two neural networks. We feed three pictures into each. The top picture is obviously a dog, the second a cat, and the third looks kind of weird, but is technically a dog. We can tell this from the labels in black and white on the left.
- Each neural network tries to predict what animal it is. NN1 is more sure of itself every single time which gives it a better result. Both NN1 and NN2 predict 2 out of 3 animals correctly.
- Here's the results again, but in a table. (The hat columns show us what the NN predicted. The non-hat columns are the actuals

NN1

Row	Dog^	Cat^	Dog	Cat
#1	0.9	0.1	1	0
#2	0.1	0.9	0	1
#3	0.4	0.6	1	0

NN2

Row	Dog^	Cat^	Dog	Cat
#1	0.6	0.4	1	0
#2	0.3	0.7	0	1
#3	0.1	0.9	1	0

- There are three ways to measure the performance of the neural networks
  - Classification error - What percentage of correct labels were applied. By this logic, both NNs are the same, as they both got 2/3 right
  - Mean squared error - NN1 has a 0.25 Mean Squared error, while NN2 has 0.71. So NN1 is better
  - Cross entropy - NN1 has 0.38, while NN2 has 1.06

Classification Error

1/3 = 0.33 1/3 = 0.33

Mean Squared Error

0.25 0.71

Cross-Entropy

0.38 1.06

- So why use cross-entropy rather than mean squared error? (Both are simple to do)
- The advantages of cross entropy are not obvious
  - If you're at the start of backpropogation, if your output is very tiny, the gradient will be very low, and it will be hard for the neural network based on a mean squared error to move it around. On the other hand, it's easier to do if you're using cross entropy.

- Additional reading : Video: Geoffrey Hinton "Softmax Output Function" https://www.youtube.com/watch?v=mlaLLQofmR8
- Additional reading: "A friendly introduction to cross-entropy loss" by Rob DiPietro (2016) https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/
- Additional reading: "How to implement a neural network intermezzo 2" by Peter Roelants (2016) http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/