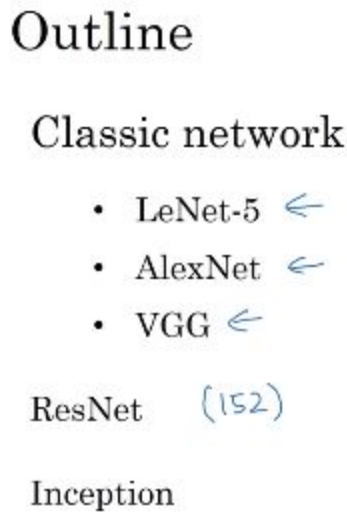
**Andrew Ng's Convolutional Neural Networks**

# Week 2: Deep Convolutional Models, case studies

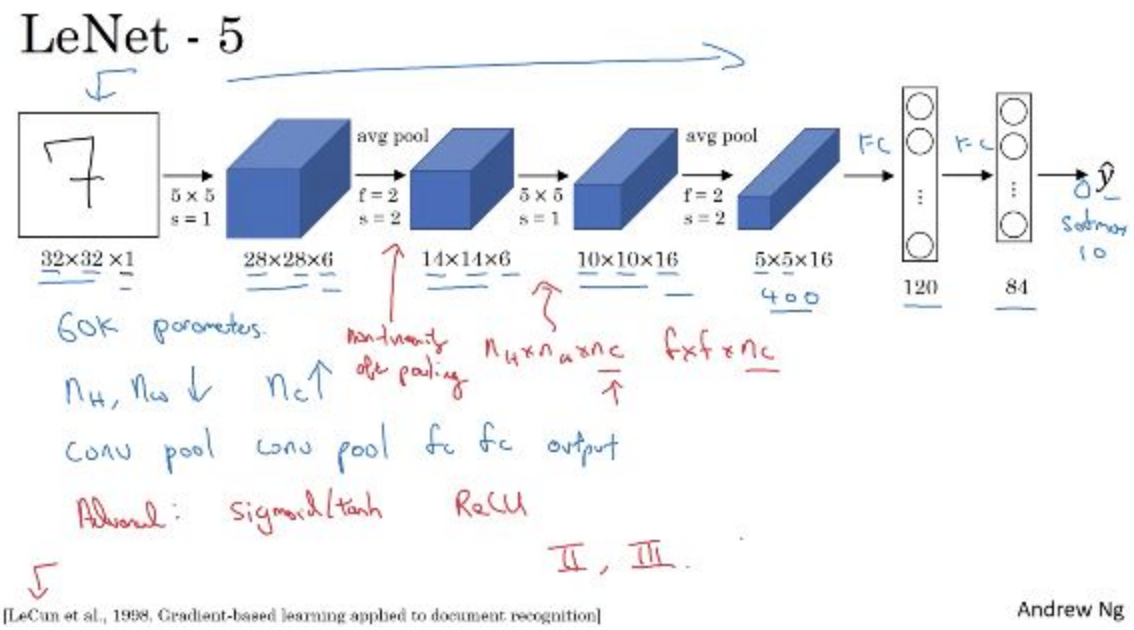
*Why look at case studies*

- This week we are going to look at some case studies, why? Because it turns out that a neural network that works for one computer vision task usually works well on other tasks. We are going to look at classic networks, residual networks and inception networks.



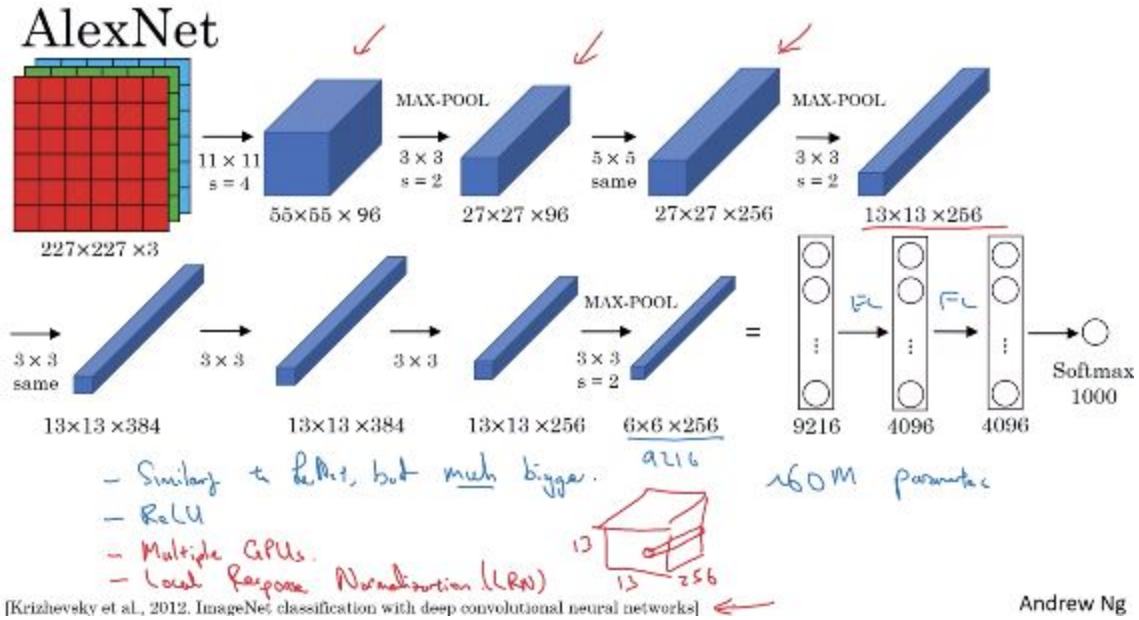

*Classic Networks*

- In this session we will look at the architecture of some classical neural networks starting with LetNet-5 (1998). The architecture is shown below. Notice a few things:
    - The sequence of layers is similar to the ones we have seen before: conv->pool->conv->pool->fc->fc->output. This is a very common architecture that is still used today.
    - Height and width decreases as we go through the layers and the number of channels increases, just like we have seen in our previous examples.



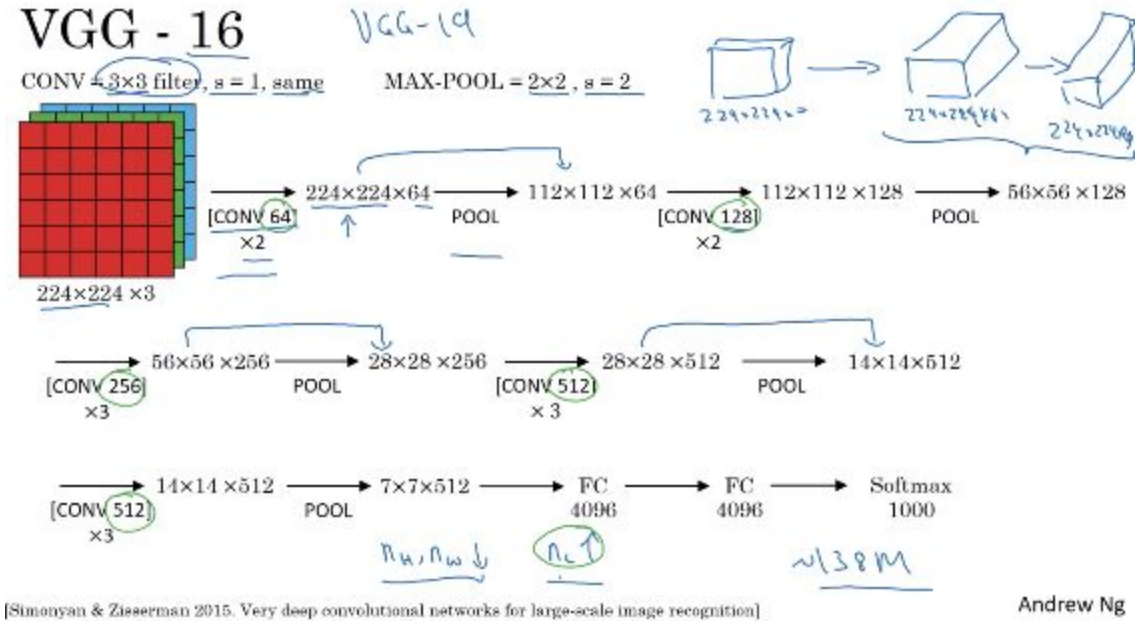

LeNet-5 was trained for grayscale images which is why we only have 1 channel in our input image. Its size is small by current standards, 60K parameters; these days it is common to see neural nets with millions of parameters. Finally, in those days average pooling was used more often, today we do max pooling more. The original research paper is available online if we want to read it.

- The next architecture we are going to see is AlexNet, its architecture is shown below. It is similar to LeNet but much bigger, it had 60 M parameters. It also used ReLu as an activation function (LetNet-5 didn't).



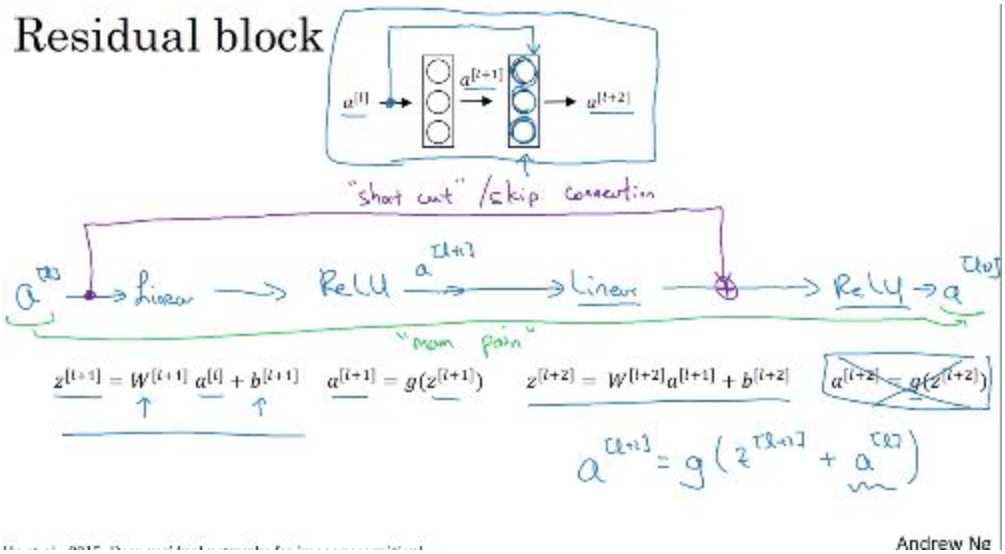

- The last network we will see is the VGG-16, named like that because it has 16 layers with parameters. The architecture below illustrates the advantages of this network: its

simplicity. It only used 'same' convolution and max pooling. In general we see the number of channels doubling every time we apply convolution and we continue to see the pattern of decreasing height and width as we go through all the layers. The downside is that it is a big network (138 M parameters), even for modern standards. The uniformity of the architecture made it appealing to researchers.

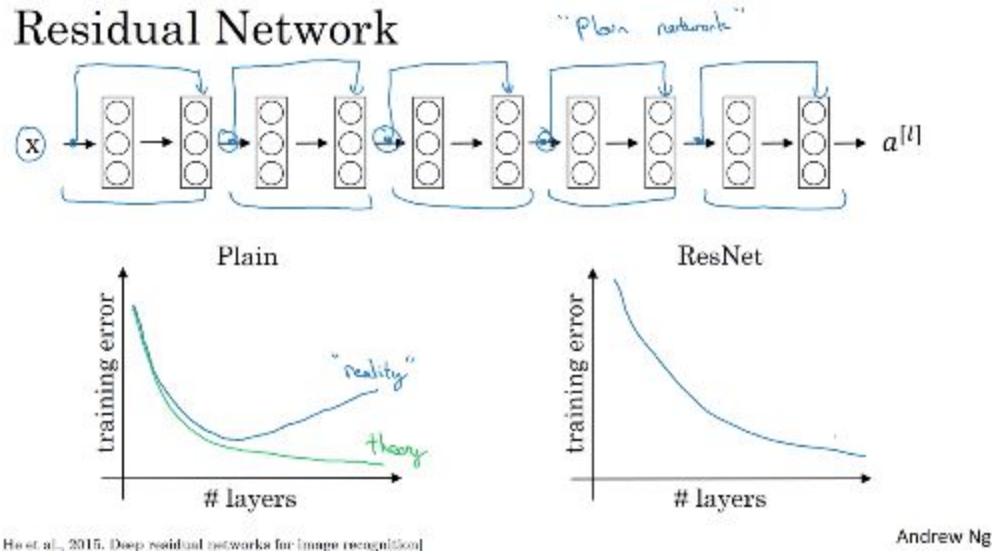



## *Residual Networks (ResNets)*

- Very deep neural networks are hard to train due to the exploding gradient problem. Here we will learn how we can take an activation from one layer, skip a layer and feed that activation deep into the neural network. ResNet are build on a residual block. Let's see what it is with an example. In the neural network below we have 3 activations and the 'traditional' path of processing is also shown below:
$a^{[l]} \rightarrow Linear \rightarrow ReLu\ activation \rightarrow a^{[+1]} \rightarrow Linear \rightarrow ReLU\ activation \rightarrow a^{[l+2]}$ . What we do different in this case is that we take $a^{[l]}$ and create a shortcut (also known as skip connection) and feed it to the $a^{[l+2]}$, our new equation is then $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$. $a^{[l]}$ is known as the residual block. The inventors of this network found that it allowed them to train much deeper neural networks.



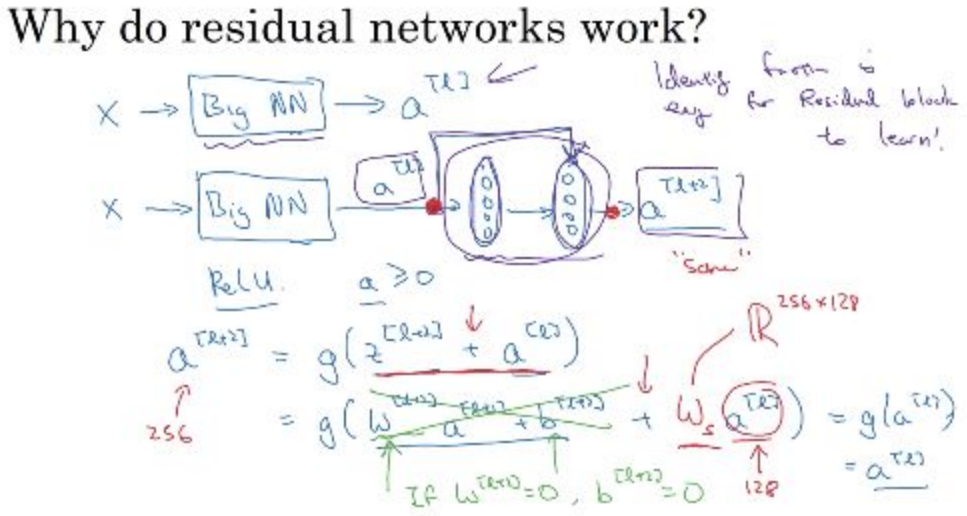

- The way we build a resnet network is to take many of these residual blocks and we stack them together. Let's take the example below. We start with a plain network and we create 5 residual blocks as shown below. If we take a traditional network and we use an optimization algorithm like gradient descent we find out that, in practice, the training error will decrease but eventually will increase again. However, the theory says we should always see a decrease. A ResNet actually matches the theory, the training error continues to decrease as the number of layers increase and this allows us to train deeper neural networks.



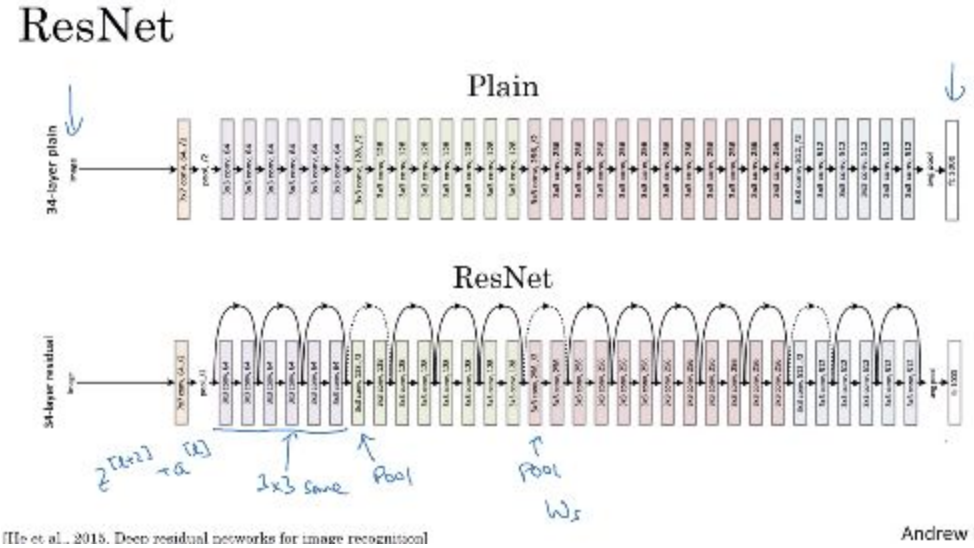

*Why ResNets work*

- Let's use an example to illustrate why resnets work. Below we have a neural network that outputs $a^{[l]}$ and then we have a second network that is a bit deeper. It adds two more

layers and the output is $a^{[l+2]}$. Let's also assume we are using the ReLu activation function so the output a >= 0. For this second network $a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(w^{[l+2]}a^{[l+2]} + b^{[l+2]} + a^{[l]})$ . If we use weight decay on $w^{[l+2]}$ and $b^{[l+2]}$ that whole term goes away and $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$ . This shows the identity function is easy for residual block to learn and this means adding the 2 new layers doesn't hurt the network's ability to learn. What is wrong with plain deeper neural networks is that it is very difficult to choose parameters that learn even the identity function which is why adding more layers makes our result worse. One last thing: the addition below assumes both operands are of the same dimensions. What this means is that resnets use 'same' convolutions to preserve dimensions. If the operands are not of the same dimensions we add a matrix $W_s$ so that we can do the addition; in the example below $W_s$ has dimensions (256,128) so that we end up with an operand of dimensions (256,1)



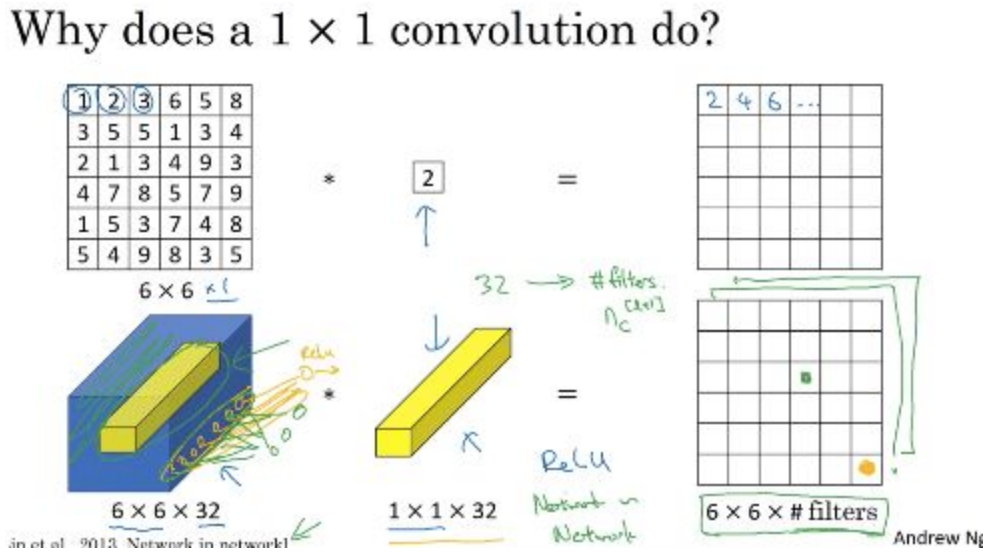

- Finally let's take a look at res nets in images. We have a plain network that we turn into a resnet by adding the residual blocks as shown below.



This means we have a bunch of 'same' convolutions and occasionally a pooling layer where we need to adjust our dimensions by adding matrix $W_s$ so that the addition $z^{[l+2]} + a^{[l]}$ can happen.

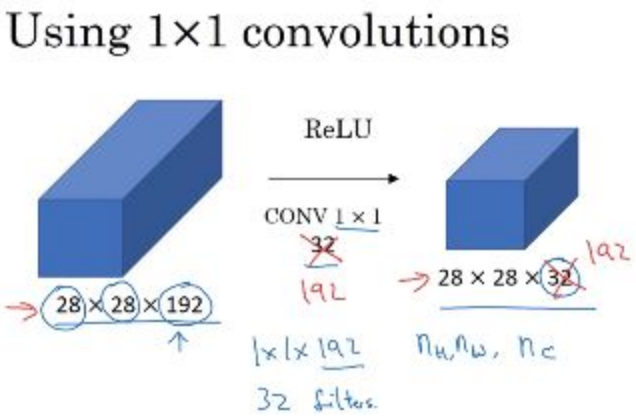
## *Networks in Networks and 1x1 Convolutions*

- What does a 1x1 convolution do? Let's take a look below. In the first example below with just one channel a 1x1 filter doubles the values of each number in the input. Ok but doesn't do much for us. The magic is in the 2nd example. In this second example we have a 6x6x32 input and a 1x1x32 filter. The 1x1 convolution will take 32 numbers on the left and the 32 numbers on the filter and do the element-wise multiplication between them. After that we apply a ReLu activation. Our output will be a single number which is one of the outputs below. One way to think about a 1x1 convolution is that we have a fully connected neural network that applies to each of the different positions so we end up with an output of 6x6x#filters.



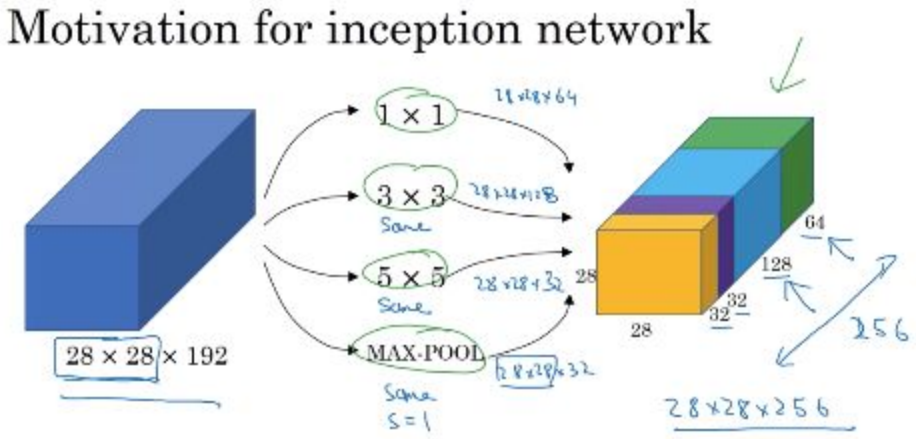

1x1 convolution is also known as network in network.

- How can we use a 1x1 convolution? It allows us to shrink, increase or keep the same number of channels. In the example below we have an input matrix of dimensions 28x28x192 and let's assume we want to decrease our number of channels. We can use 32 filters of 1x1x192 dimensions and the output volume will have dimensions 28x28x32. This is a way to shrink $n_c$. So 1x1 convolution adds non-linearity and allows us to learn a more complex function by adding another layer to our network.

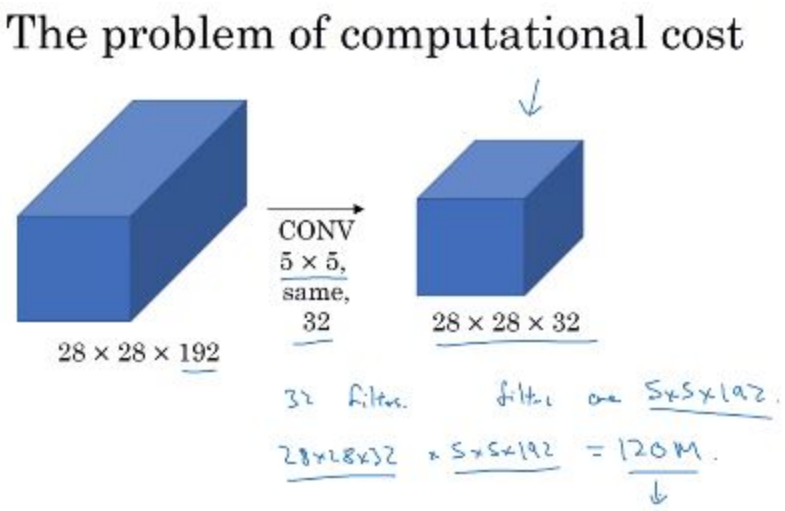



*Inception Network Motivation*

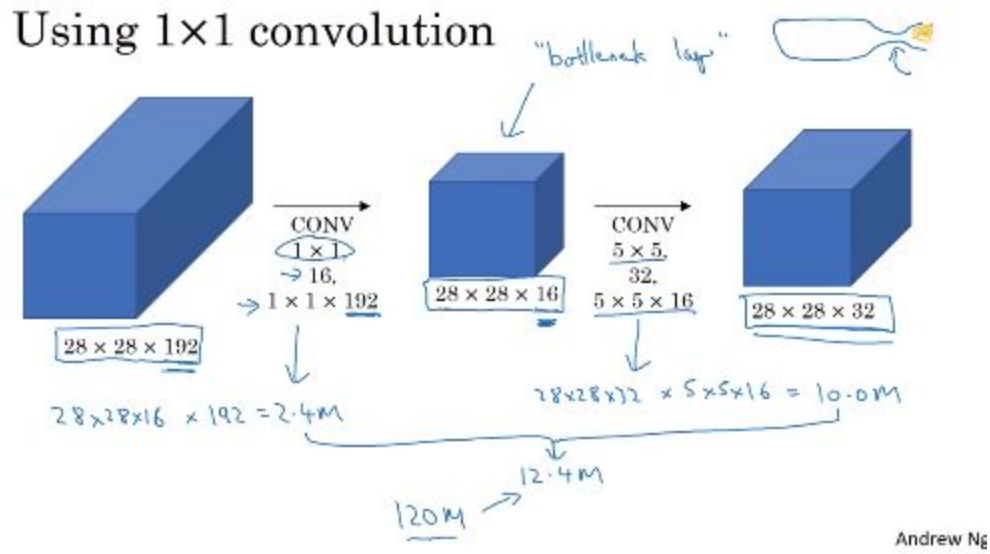
- When we are designing a Convolutional Network we might have to pick a filter size or whether or not to use pooling. An Inception Network basically says: let's do them all. In the example below we use 3 different filter sizes and pooling and what we do is we stack all the outputs so we go from an input of 28x28x192 to an output of 28x28x256.




- There is a problem with inception networks and it is the computational cost. Let's start by looking at the cost of a convolution operation below. Using a 'same' convolution on the input below will require us to compute 120 M numbers, an expensive operation.




- Now compare the cost above with the cost below. If we use an intermediate layer, also known as the bottleneck layer, we are able to shrink the number of operations needed to 12.4 million, about 1/10th of the previous operation's cost.

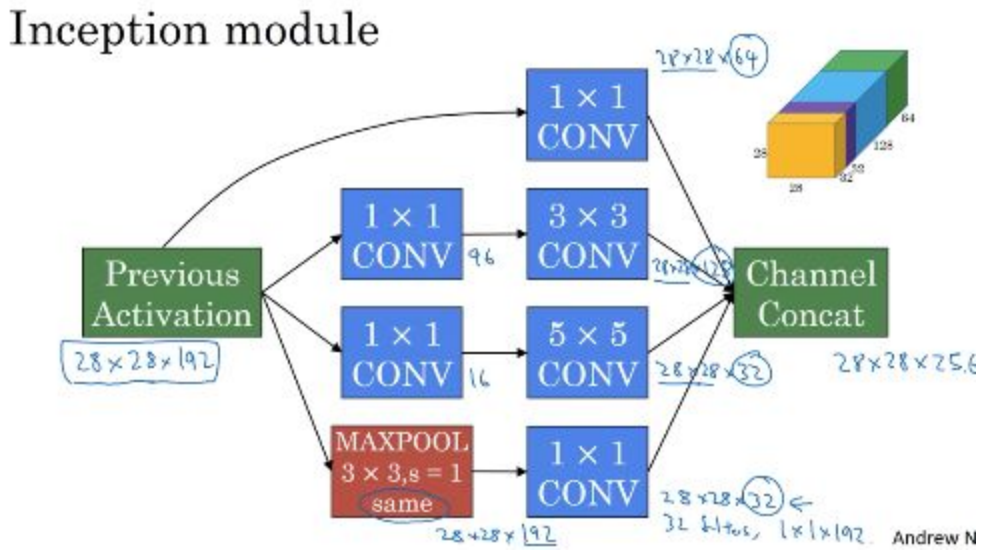



Summarizing, if we are building a layer of a neural network and we don't want to decide on the filter size or whether or not to do pooling, the inception module allows you to do all of them. We have also shown how we can minimize the computational cost problem by using a 1x1 convolution.

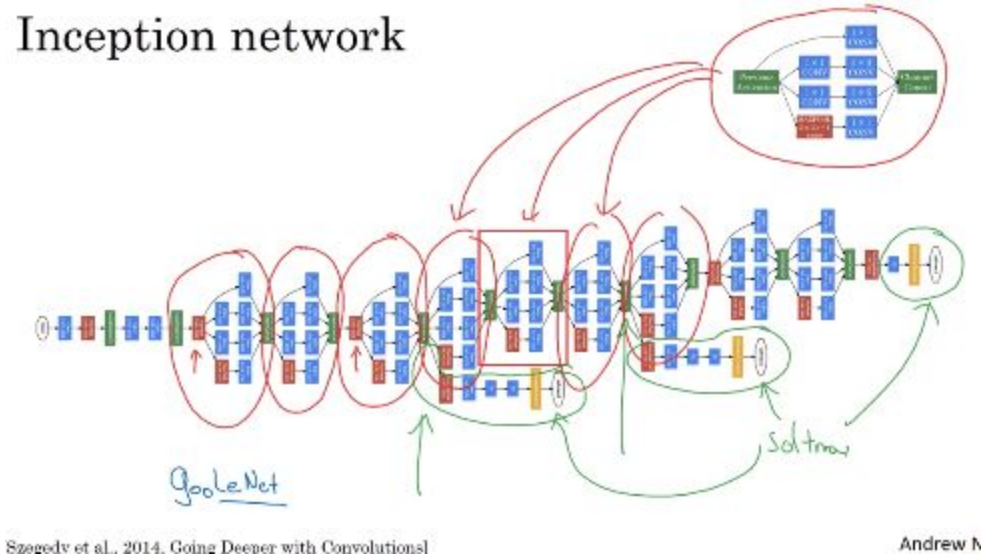
## *Inception Network*

- Let's see how we can put together an inception network using what we learned last time around. Below is an inception module that takes the previous activation as input and

uses different filters and pooling to output a concatenation of the 4 outputs. In this case it is a 28x28x256 output.




- An inception network contains a number of inception modules as shown below. Notice that we have some branches that also predict labels just like the last branch. They all end with a softmax activation. These side branches takes a hidden layer and try to make a prediction. These layers have a regularization effect on the network and prevent it from overfitting.



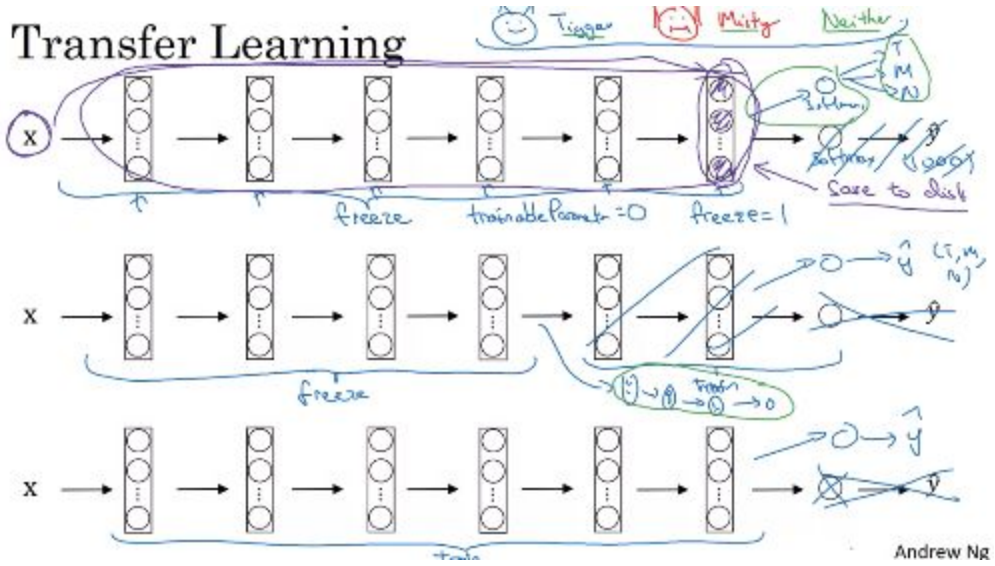

*Practical Advice for using Conv Nets: Using Open Source Implementations*

- Professor Ng recommend we go to github to find open source implementations of the types of networks we learned about. Most likely there will be an open source version of

what we just learned and we shouldn't be surprised if it is from the authors of the original research paper

### *Practical Advice for using Conv Nets: Transfer Learning*

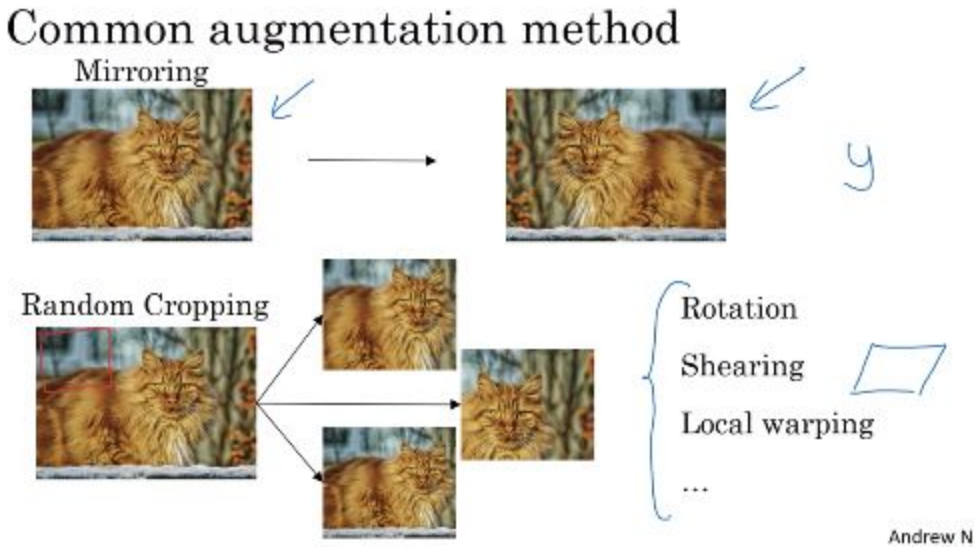
- If we are building a computer vision application one way to go about it and make a lot of progress is to download an architecture that someone has already trained and use that as initialization for your problem. Below we have a case where we have to classify if an image is one of 3 cats (tigger, misty or neither). We consider 3 cases:
    - If we don't have many pictures of tigger and misty we could use the whole network we downloaded and just sub the last layer with our own layer, perhaps a softmax layer that has 3 predictions (tigger, misty and/or neither). In this case we 'freeze' all the layers and just add our at the end. We can also store the result of the original network (minus the last layer) and then add ours.
    - If we have more data what we can do is freeze a smaller amount of layers of the downloaded architecture and train the other layers.
    - Finally, if we have a lot of data we might decide to train all the layers and we still use our own last layer.



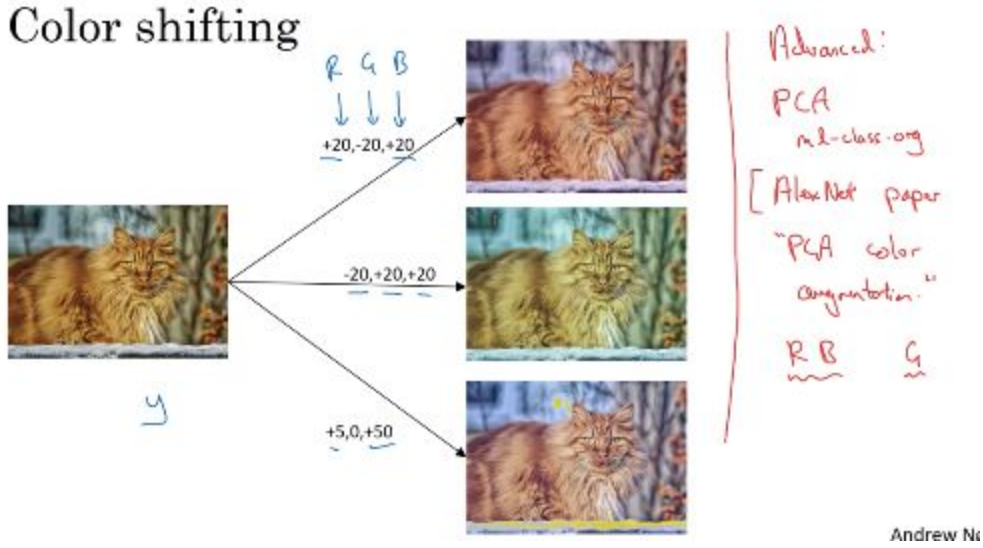

So the rule of thumb is that the number of layers to train decreases as we have more data available. In practice, computer vision is an area where we should always use transfer learning unless we have a huge data set for training everything from scratch.

### *Practical Advice for using Conv Nets: Data Augmentation*

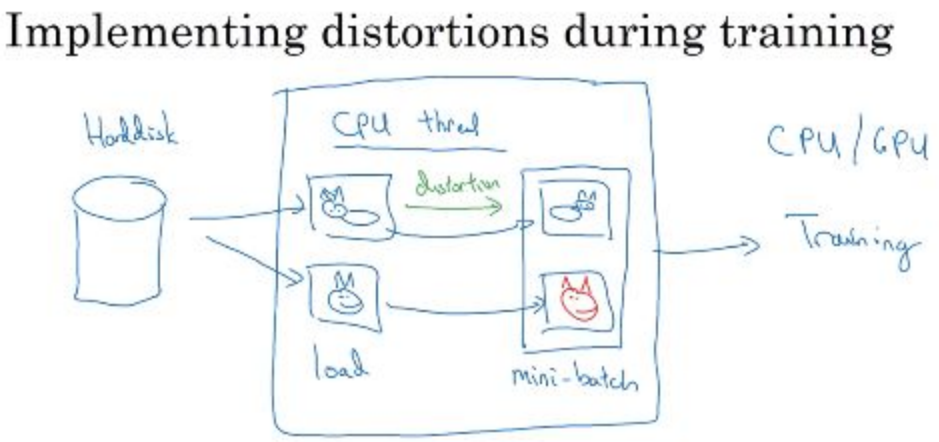
- Most computer tasks could be improved using more data so data augmentation is one technique that is often used to improve computer vision systems. Let's take a look at some common data augmentation methods. Two commonly used methods are mirroring and random cropping, both shown below. Others are rotation, shearing and local wrapping.




- Another commonly used method is color shifting where you change the value of the R, G or B channels as shown below.

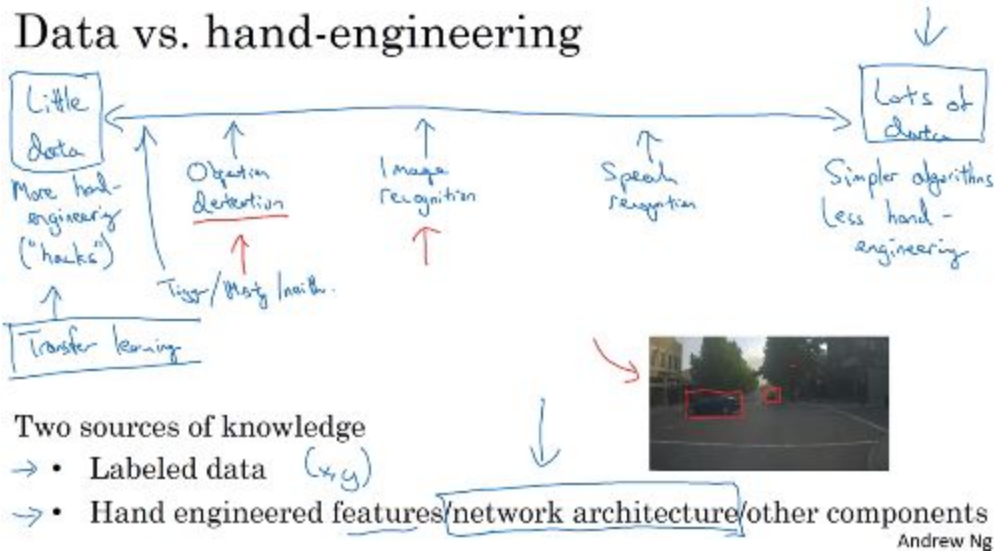



- Implementing distortions during training often follows the process below: our training data is on disk and we have one CPU thread applying the distortions we want to each image. Once the distortion is applied then the resulting image is given to another CPU (or GPU) thread for actual training. Often these 2 threads run in parallel.



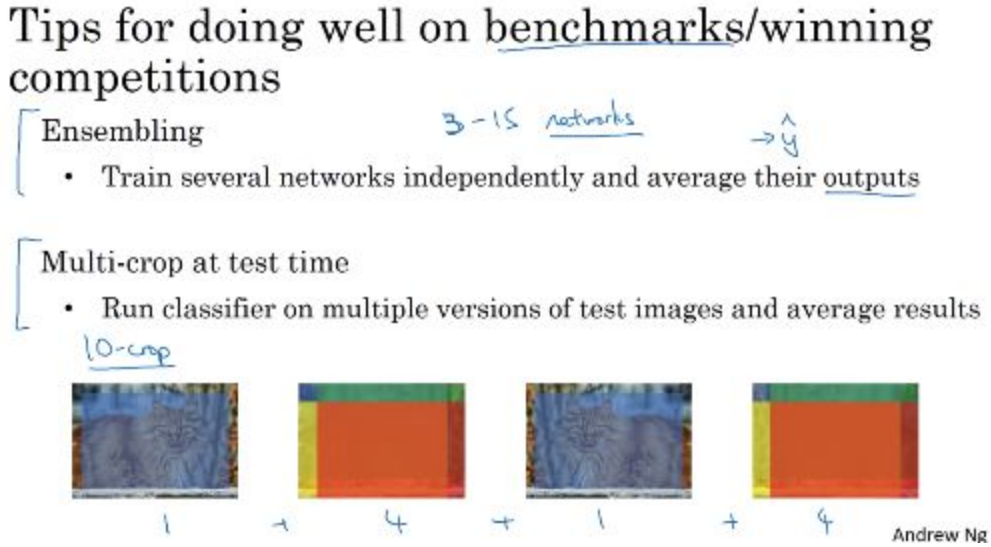

*State of Computer Vision*

- Most ML problems fall somewhere over the spectrum of having little data and a lot of data. In this continuum, speech recognition, in general, has a lot of data while image recognition and object detection have, in general, a little amount of data. If we have a lot of data we have simpler algorithms and less hand-engineering. If we don't have a lot of data we then do more hand-engineering/hacks. Two sources of knowledge for ML problems are labeled data and hand engineering features/network architecture and other components.




- We also have some tips for doing well on benchmarks. This is relevant because if you do well on benchmarks you have a better chance of your paper being selected for publishing. A couple of common techniques are ensembling (where we train several networks independently and average the outputs) and multi-crop at test time (where we run a classifier on multiple versions of the same image and average the results). For multi-crop there is a commonly used technique called 10-crop where we run the

classifier in 5 different versions of one image and on 5 different versions of the mirrored image.




- Finally, to build a practical system we can use architectures of networks already published, use open source implementations and use pretrained modules and fine-tune our dataset.