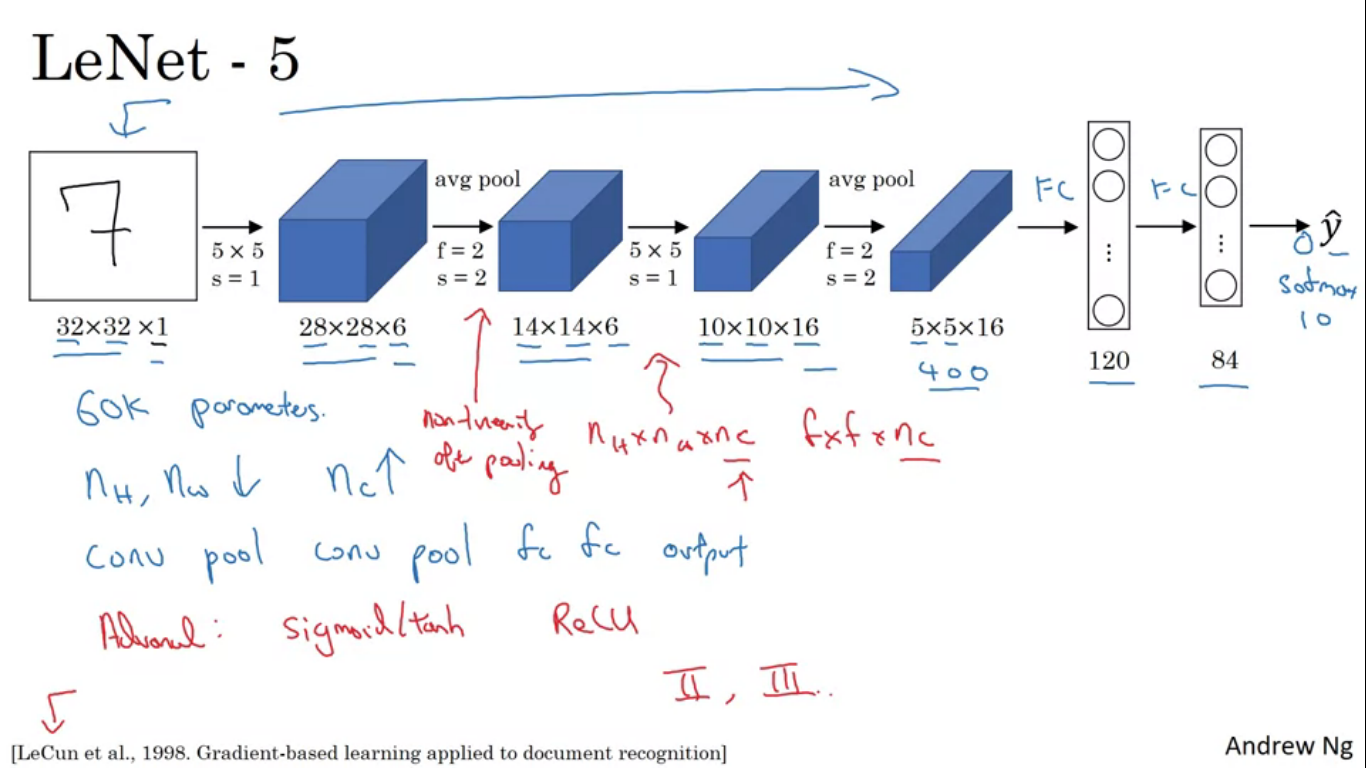
**Why look at case studies of CNN?**

A good way to get an intuition on how to build conv nets is to read and see other examples of conv nets.

We will first dive into a few classic networks

* LeNet-5 from 1980
* AlexNet
* VGG
* ResNet (152 layer NN)
* Inception NN

**LeNet – 5 architecture**

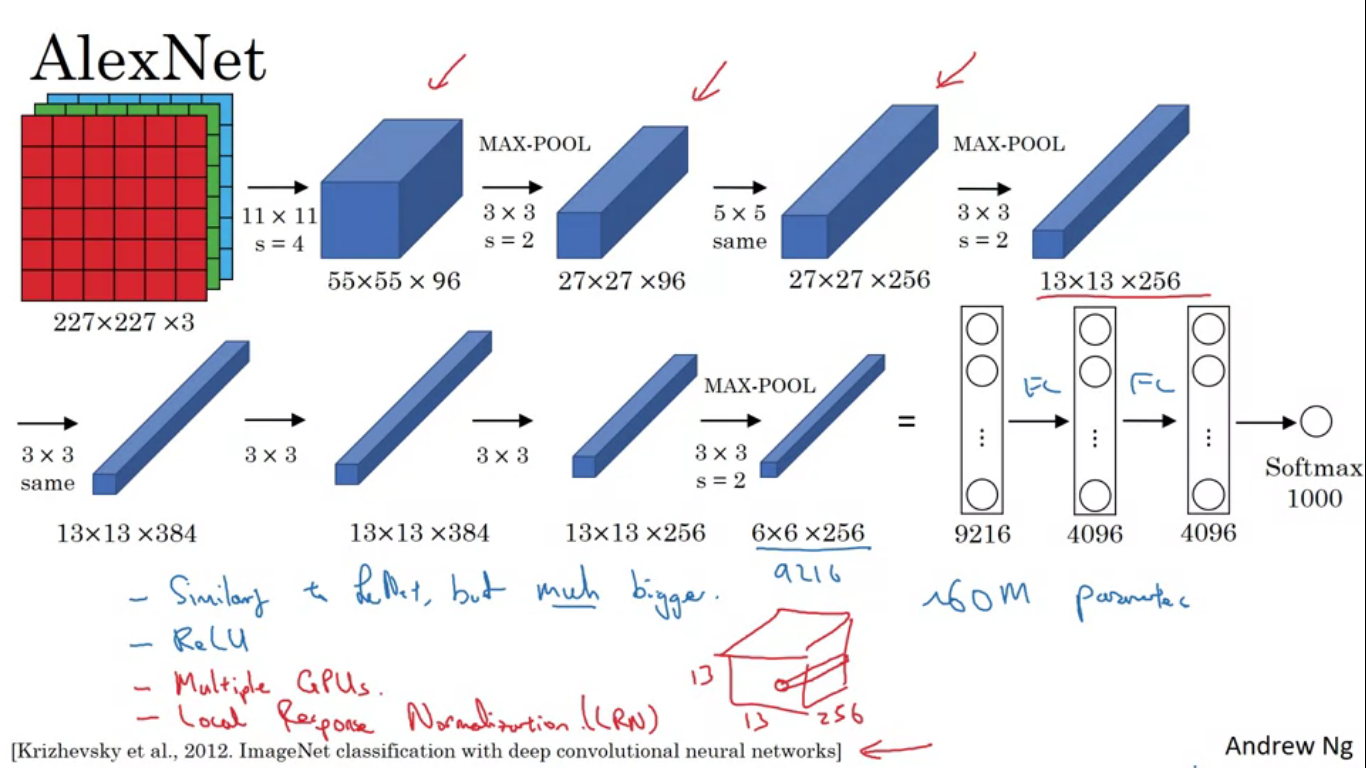


Goal of Lenet 5 was to recognise hand written digits. Lenet 5 was trained on gray scale Images.

Input is 32x32x1, first convolution has six 5x5 filters over stride of 1, output from this image is 28x28x6. Then average pooling was applied with filter width and stride of two to reduce the size of the image by a factor of 2. So output is 14x14x6. Next another convolutional layer, with 16 filters of size 5x5, so output was 10x10x16. Then another pooling layer reduces the image by a factor of 2. The image size we obtained has to be flattened so 5\*5\*16 = 400. Next layer is fully connected which connects each off the 400 nodes with 120 neurons, followed by another FC layer with 84 nodes. Finally it uses the 84 features with a final output. Final layer will be a softmax layer with 10 outputs.

This NN was small by modern standards, and it had 60k parameters, as we go deeper in the network the height and width tend to go down whereas the number of channels increases.

**Alex Net Named after Alex Krizhevsky, author of the paper describing this work.**

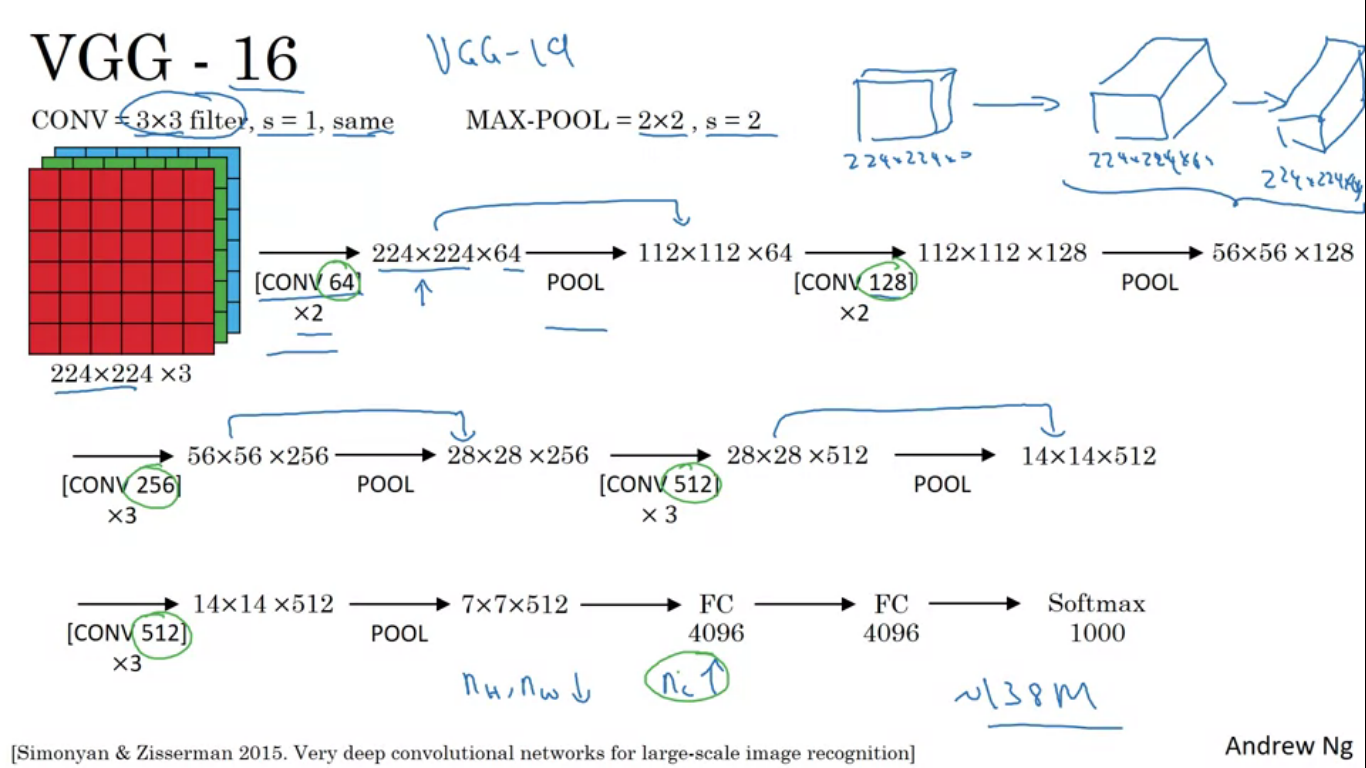


Alex Net starts with 227x227x3 inputs. The first layer applies a set of 96, 11x11 filters with a stride of 4 so output comes as 55x55x96. Next come pooling with a stride of 2 and size of filter as 3x3, reducing the size to 27x27x96. Then apply same padding convolution filter size 5x5 and resulting in 27x27x256 size image. Maxpooling again, reducing the size to 13x13x256. Again same padding convolution so result is 13x13x384 filters. Then 3x3 same convolution gives 13x3x256. Then maxpool reduces the size to 6x6x256. Then comes flatten 6\*6\*256 = 9216 nodes are flattened. Finally it has some fully connected layers and uses a softmax to output out of 1000 classes.

Similar to lenet, although much bigger. This has 60M parameters. It also used ReLu activation in comparison to sigmoid or tanh.

It also used the idea of local response normalization, not much used nowadays though. It is normalizing the image across all channels. Motivation was that we don’t want too many neurons with a very high activation.

**VGG-16 Network**



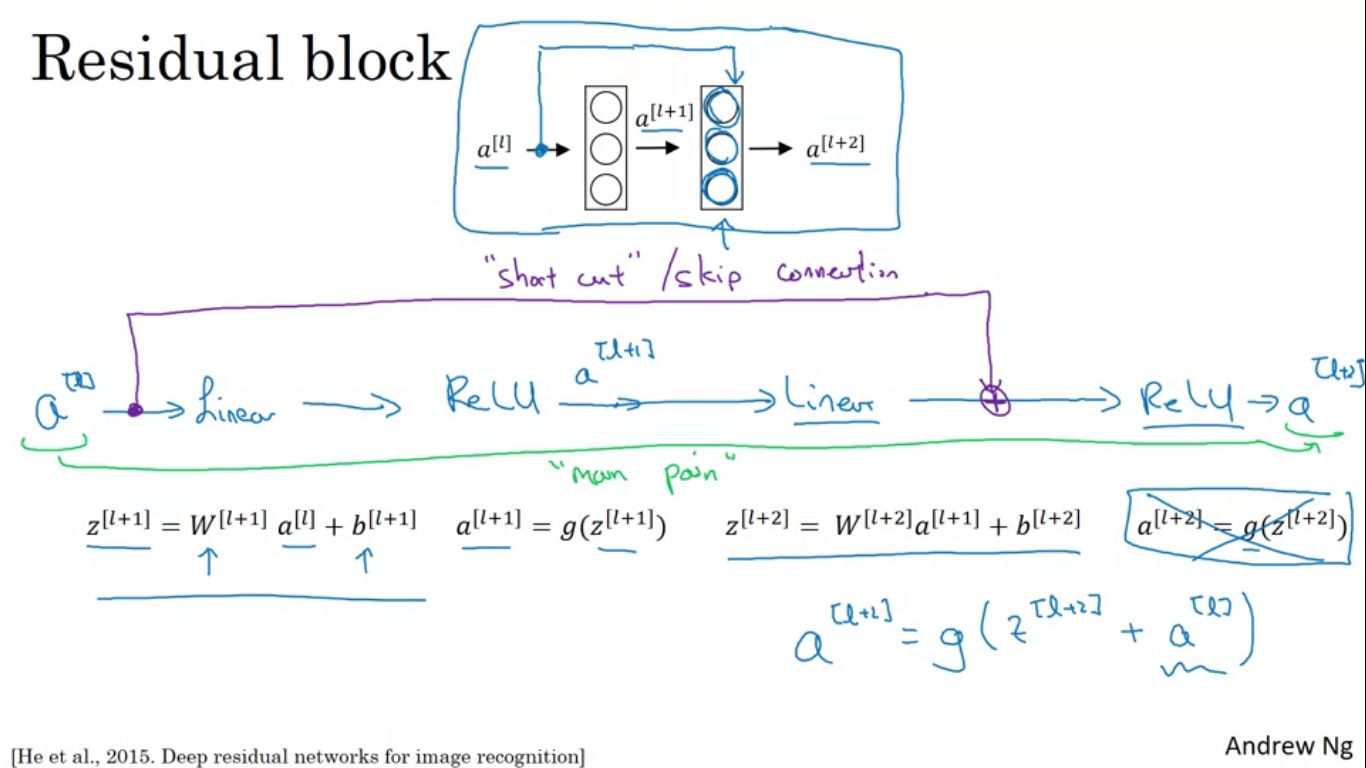
Instead of having so many hyper parameters let’s use a simpler network, where the conv layers are just 3x3 filters with a stride of 1and always use the same padding and pooling layers with a stride of 2 and filter size of 2x2.VGG network really simplified this NN architectures.

Input is 224x224x3, then comes two conv layers with 64 filters each so we get 224x224x64. Then comes the pooling layer reducing the image to 112x112x64. Then a 2 more conv layers 128 filters each, resulting in 112x112x128. Then comes pooling reduces the size to 56x56x128. Next comes 3 conv layers with 256 filters each, the image size is 56x56x256. Then pooling layer reduces the size to 28x28x256. 3 more conv layer with 512 filters, image size is 28x28x512 followed by pooling layers reducing the size to 14x14x512.. the above process repeats and the final output image is 7x7x512. The final 7\*7\*512 = 4096 feeds into FC layer again the FC layer with 4096 units and then a softmax output with 1000 classes.

“16” in VGG-16 refers to the fact that this has 16 layers with weights. Has a total of 138M parameters. VGG-19 is an even bigger version of this network.

Nc in VGG goes up by a factor of 2 .

**Res Nets**

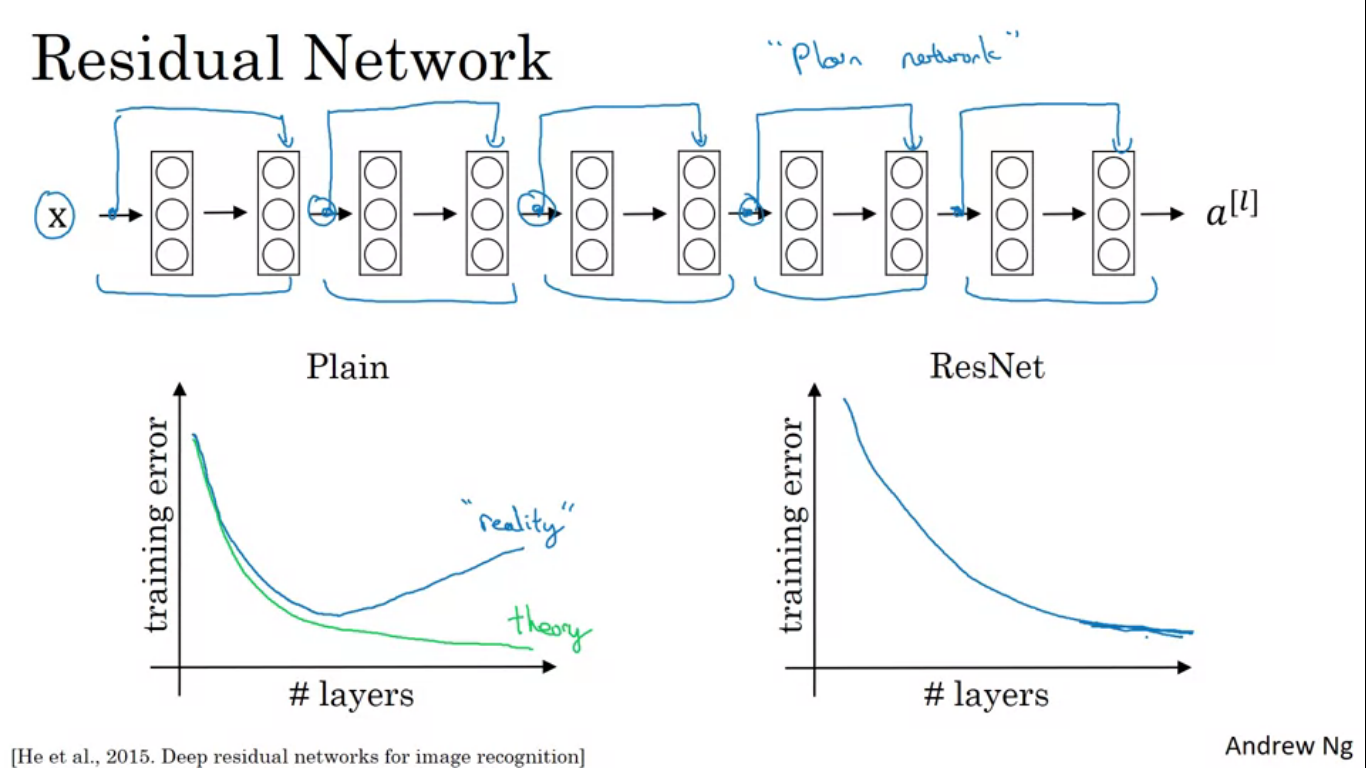


Very deep NN are difficult to train because of vanishing and exploding gradients problems. It uses skip connections which allows us to take the activation from one layer and suddenly feed it to another layer even much deeper in the NN. Using skip connections we build res nets which enables us to train very deep networks.

Res nets are build out of something called residual blocks. To information to flow from a[l] to flow to a[l+2], it needs to go through all of the main path as depicted. For residual network, we follow the shortcut path as shown. So

A[l+2] = g(Z[l+2] + a[l])

Shortcut mentioned above is also known as skip connections.

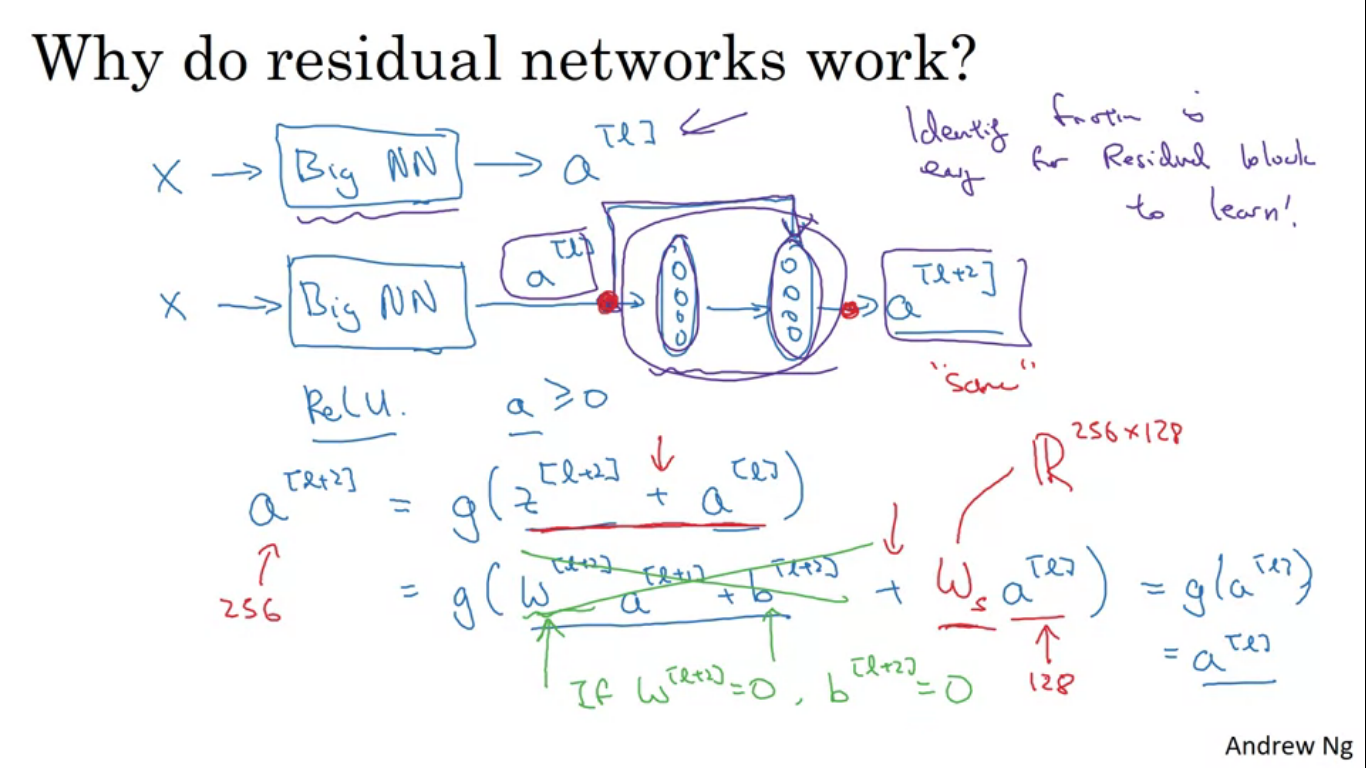


To build a res net we take many of these residual blocks and stack them together to form a deep network.

To build a plain network to residual network we do as shown. The picture shows five residual blocks stacked together.

In a plain network as we increase the number of layers the training error will increase for a while but then it will tend to go back up. In theory having a deeper network helps but in practice having a very deep plain network, our optimization algorithm will have a much harder time training and our training error may get worse. But in resnets even as number of layers increase we can have the error going down.

**Why do res nets work so well?**



If you make your network deeper it can hurt the ability to train the network to do well on the training set.

X feeding in to some big network and output as a[l]. Modify the NN to make it a little bit deeper. We add the two layers with that res net block. We are using ReLu activation fn so all activations are going to be >=0.

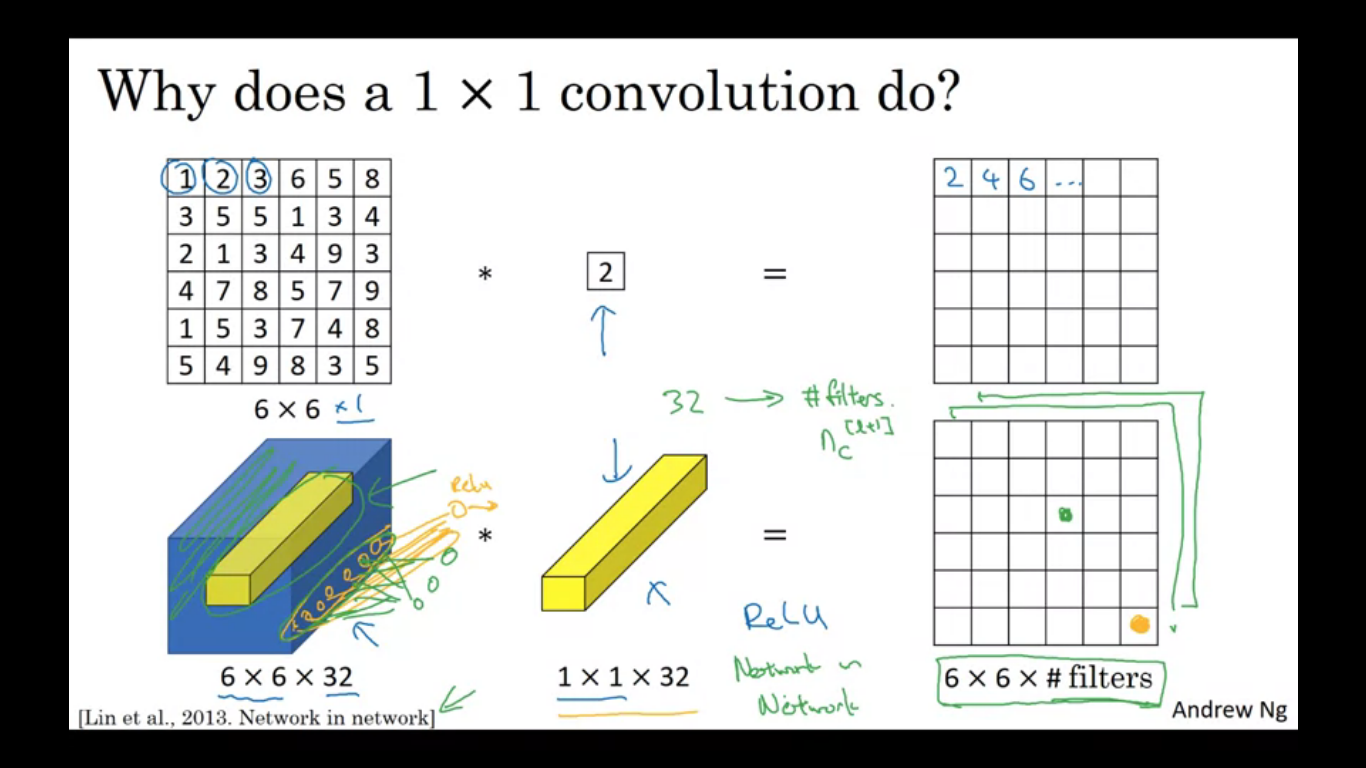
A[l+2] = g(Z[l+2] + a[l])

So if we are using L2 regularisation or weight decay that will tend to reduce the value of w[l+2]. So if w[l+2] = 0, and also b[l+2] = 0, then these terms go away as they are equal to zero. And g(a[l]) = a[l] as we are using the Relu activation function.

This shows that identity function is easy for residual block to learn. And because of this skip connection we get a[l+2] = a[l]. So we just copied the value of a[l] to a[l+2] despite the addition of the two layers. As the network gets deeper it becomes very difficult to choose the parameters to learn the identity function. The main reason the residual nets work is because it is so easy for these extra layers to learn the identity function sometimes even helping performance.

Through the addition term applied to Z[l+2] and a[l] we assume both of them have he same dimension. So in resnets we see a use of lot of use of same convolutions. Same convolution preserves dimensions so it is easy to carry out this addition of two equal dimension vectors. In case if they are not same dimensional we add an extra matrix named Ws it would be a 256x128 dimensional matrix. Ws can be zero padding to make the a]l] 256 dimensional. Generally we need to do this if there are pooling or pooling like layers.

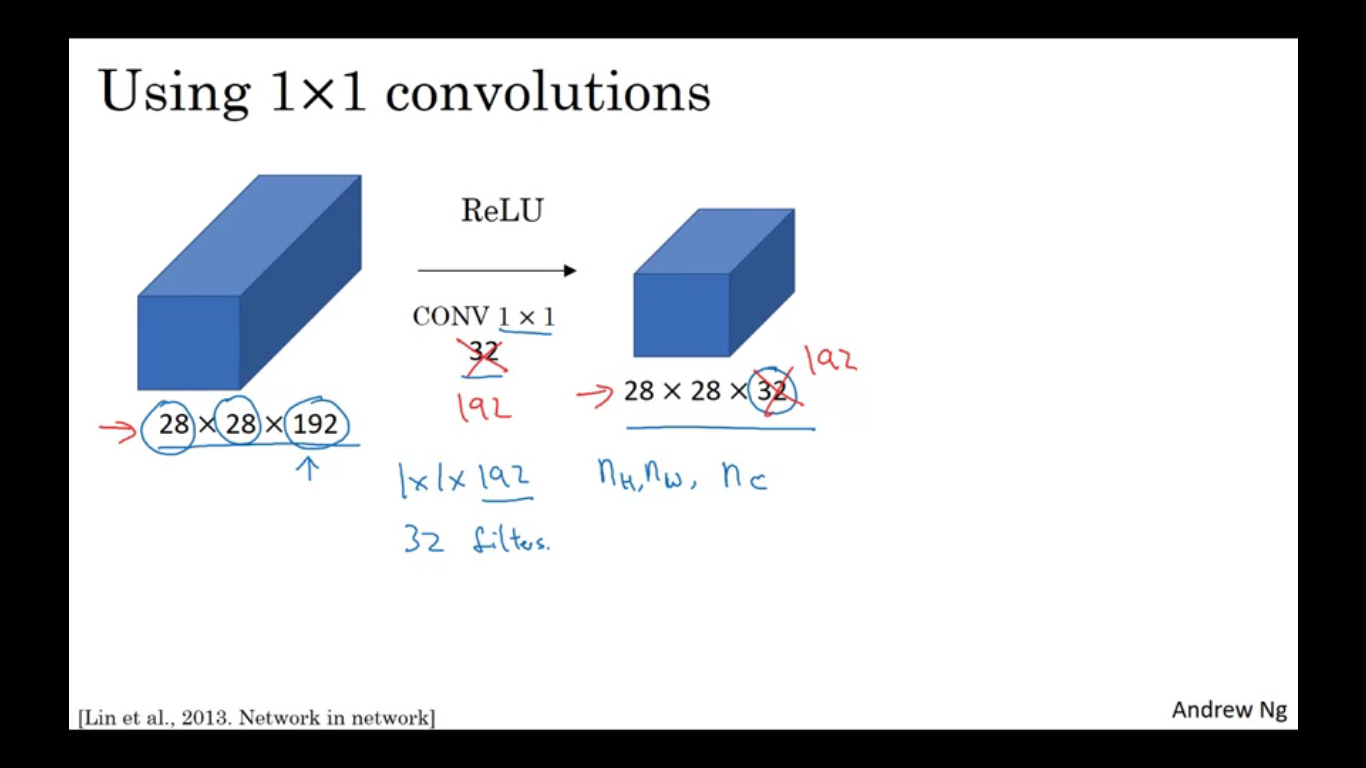
**What does a 1x1 filter do?**



A convolution by 1x1 filter just multiplies the value of pixel by the number present as the filter value. So it doesn’t seem particularly useful. That is the case with 6x6x1 channel images. So 6x6x32 image when convolved with 1x1x32 filter actually makes much more sense. So the filter will look at 32 number in the image and in the filter and apply element wise multiplication and apply relu non linearity to that. So we multiply and end up with a single real number which then gets plotted in one of the outputs. So we get the output size as 6x6xnumber of filters.

So actually 1x1 filter have a FC NN that applies to each of the 62 different positions .

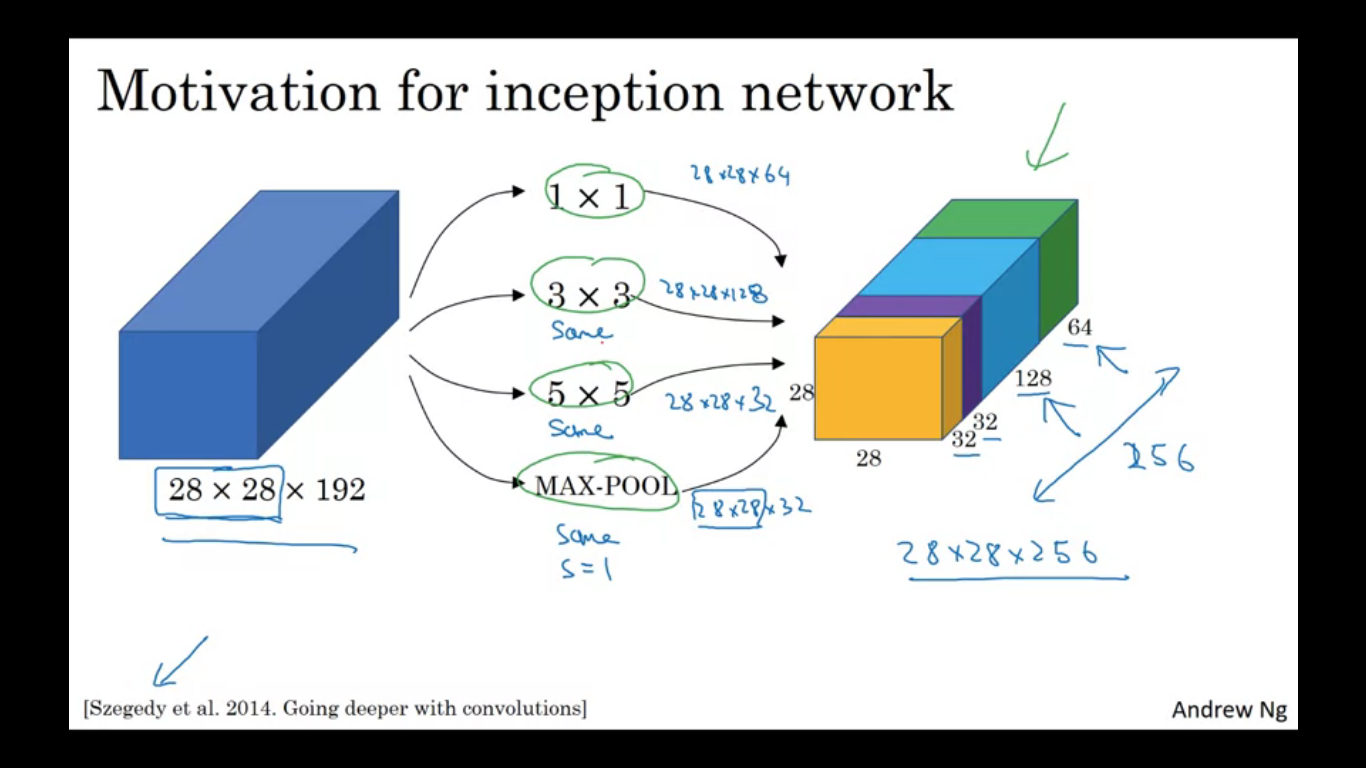
1x1 convolutions are also called network in a network.



Let’s say you have a 28x28x192 sized image. If we want to shrink the height and width, we can use the pooling layers but what if the number of channels have gotten too big and we want to reduce that. To reduce it to 28x28x32 image what we can use 32 1x1 filters and technically each filter would be 1x1x192 as number of channels in filter has to match number of channels in the input volume. So output of this process will be a 28x28x32 volume.

Effect of 1x1 convolution is that it just adds non-linearity

**Motivation for inception Network**



Input = 28x28x192

Instead of choosing what filter size you want in a conv layer, why don’t we do them all. So we use a 1x1 convolution so it gives 28x28x64 output. But also convolve with a 3x3 filter giving us 28x28x128 output. Stack up this second volume next to the first volume. Next we also do a convolution by a 5x5 filter and we output a 28x28x32. Next maybe if you didn’t want convolution so we do the pooling and that resulted in 28x28x32 and all the results are stacked together. To make the dimensions match we need to use same padding and stride of 1 for pooling.

So we have 1 inception module input as 28x28x192 and output 28x28x256 (obtained from adding all). So instead of choosing the filter sizes and pooling, we can do them all and concatenate the outputs.

Problem of inception layer as we described it here is the computational cost. Finding the computation cost of 5x5 convolution. So each filter is 5x5x192 and we have 32 filters. So we need to compute 28x28x32 numbers and for each of them we need to do 5x5x192 multiplications. Total computation = number of multiples to compute each output value \* number of output values we need to compute. When we multiply the above

28x28x32 \* 5x5x192 = 120M

This is a pretty expensive operation. Using 1x1 convolution operation we will be able to reduce the computation cost by about a factor of 10.

So using 1x1 convolution we first convert 28x28x192 to 28x28x16, then run 5x5 convolution to give ou final output. We have taken the huge volume and converted into intermediate volume which has 16 channels instead of 192 channels. To apply this 1x1 convolution we have 16 filters , each filter is of the dimension 1x1x192, cost of computing this 28x28x16 will be

We need = 28x28x16 and for each of them we need to do 192 multiplications.

So we get 28x28x16x192 = 2.4M

Cost of second conv layer

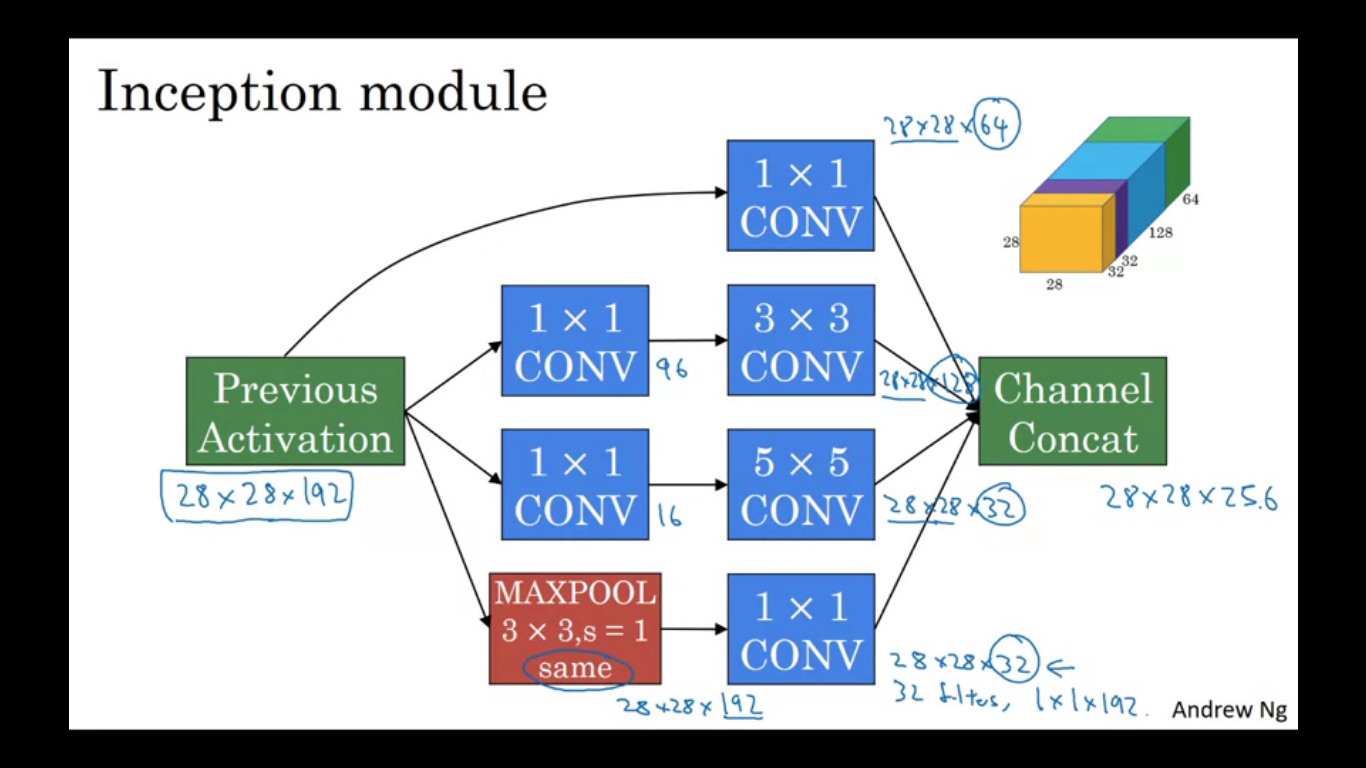
28x28x32 x 5x5x16 = 10.0352M

Total number of multiplications we need to do is sum of the above ie 12.4M

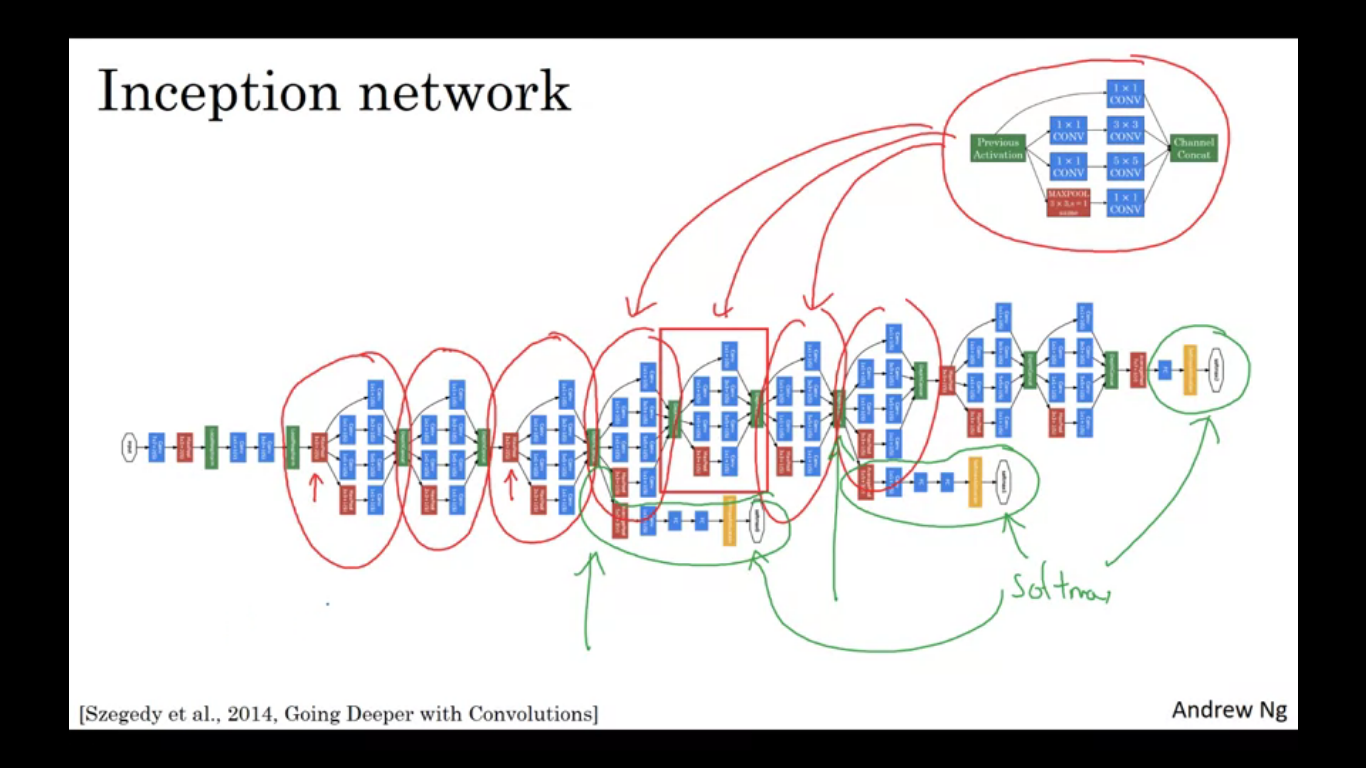
**Inception network**

Input as the activation from some previous layer

**Inception Module**



**Inception Network**



Puts a lot of inception module together. There are some pooling layers also to change the dimension. In the side there are some softmax layers too to do some predictions.

Inception was also called as googLeNet

Using open source Impementations

**Transfer learning for our task**

Let’s say building a cat detector to recognise your cat. We have cat classes as tigger, misty and none. We don’t have a lot of pictures of tigger/misty so our training set will be small.

To do our task download some open source implementation of a NN, download not only the code but also the weights. Then you canget rid of the softmax layer and create our own softmax units that outputs tigger/misty/none. In terms of the network, all of the previous layers are frozen so you freeze the parameters in the layers of the network and just train the parameters with the softmax layers. By using someone else’s pre-training weights we might get good performance on this very small amount of dataset. We might set some of the layers as trainable parameters to zero. So here train only the softmax layer weights but freeze all the earlier layer weights.

Another neat trick is to pre compute the function from the last layer before softmax and save them to disk. We use this fixed function in this first part of NN. So we just pre compute the layer’s activation , advantage of this is that we don’t need to compute the activation of the layers in every epoch.

If we have a good amount of data, we can then maybe freeze fewer layers in the start and train the later layers. Output layer have different softmax so we need to add softmax. The number we can trin on top could be greater.

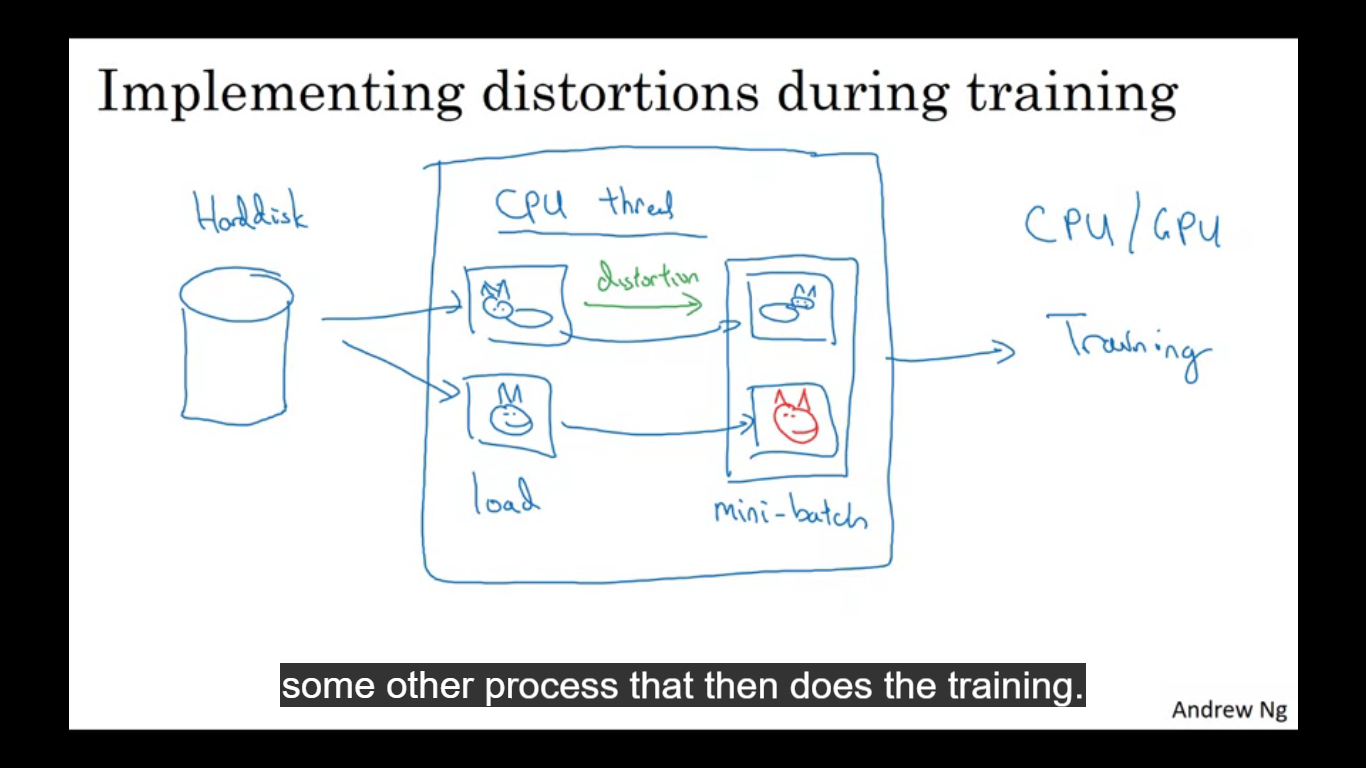
Finally if we have a lot of data, we take this open source networks and weights use the whole thing just as an initialisation and train the whole network. We could use the weights we download just as initialisation, and then do GD, training and updating all the weights of the NN

**Data Augmentation**

Technique used to improve the performance of computer vision systems. Having more data always helps in computer vision.

Simplest data augmentation is mirroring on the vertical axis. Another technique is random cropping. We could also use rotation and shearing of the image, local warping etc.

Another type of data augmentation commonly used is colour shifting. Add in different RGB channel distortions to shift the colours. This makes the learning algorithm more robust to changes in the colours of your image. One of the ways to implement colour distortion uses an algorithm called PCA or Principle Component Analysis, sometimes called PCA colour augmentation. If your image has mainly red and Blue tints and very less green tints then PCA colour augmentation will subtract a lot of red and blue but very less green so it keeps overall colour of the tint the same.



We have a CPU thread that is constantly loading images from the hard disk. We can use the CPU threads to implement the distortions. Then the CPU is really loading the data as well as implementing distortions to form a mini batch. This data is then constantly passed to the thread implementing threading.

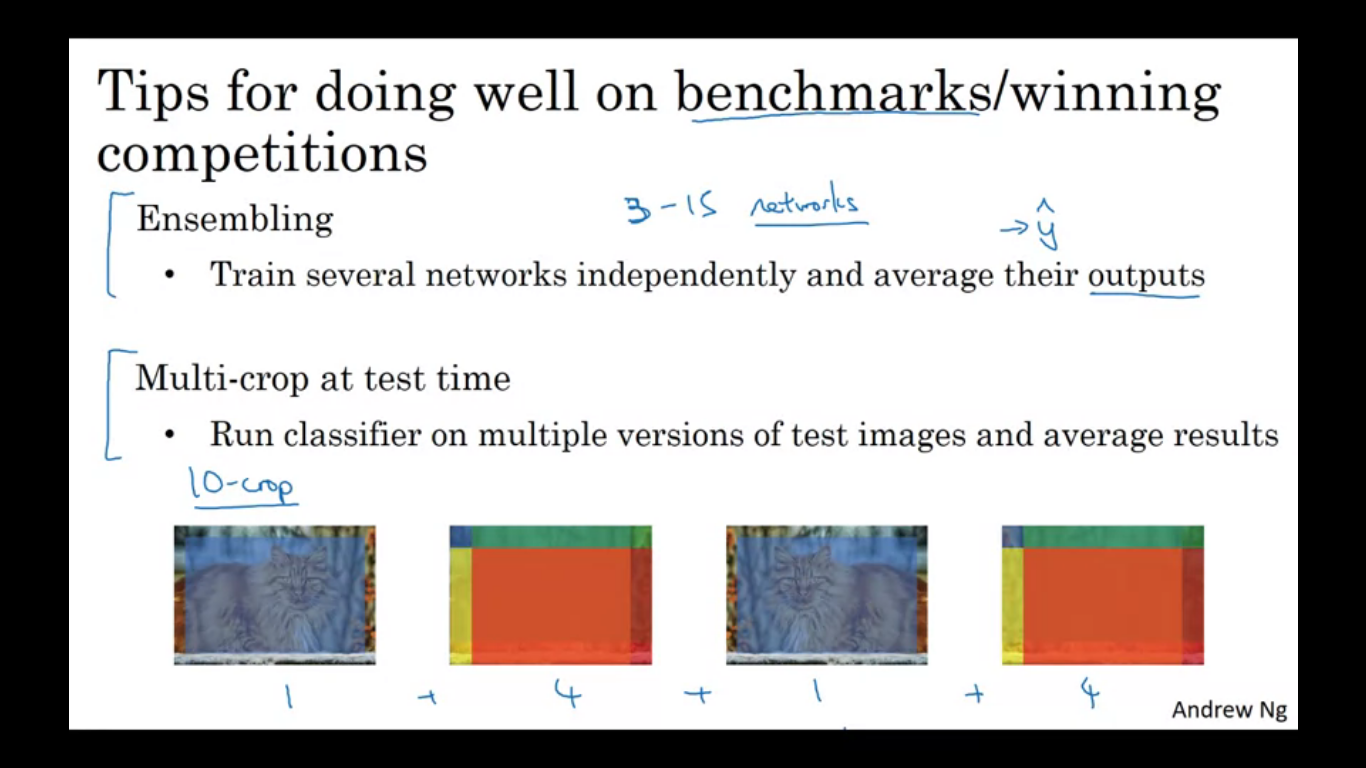
Deep learning for computer vision

Deep learning lies between little data and having excess of data. Speech recognition lies on the side of having enough data, but even though image recognition datasets are huge we still feel we are on the lesser side of data. For object detection we have even less data as in the object detection we need to put bounding boxes.

When we have lot of data we get away with simpler algorithms as well as less hand engineering.

When we have lesser data we have more hand engineering/hacks

Two sources of data



Labelled data

Hand engineering the features and network architecture

Tips for doing well on the benchmarks/winning competitions

Ensembling – train several networks and averaging their output.

Multi crop at test time – apply data augmentation and run classifier on multiple versions of test images and average the results.