Deep Learning Specialization

4. Convolutional Neural Networks

1. Foundations of Convolutional Neural Networks

- CNN are mainly used to isolate features in images. Image * filter = output.

- Vertical edge detector
  - Idea of a filter is that it convolutes with an image to get an output image.
  - There are different kinds of filters. e.g.: vertical, horizontal. Some filters can detect all kinds of edges, those at 70 degrees for example
    for images: 0s means darker, while 10s lighter, -ve values are even darker.
  - Filter weights can be learnable, through backprop. Such filters are more robust of course.
  - In python: conv-forward. tensorflow: tf.nn.conv2d.

- Padding process of adding (p=1) pixel in each dimension for the input image.
  - It is done mainly for 2 reasons:
    1. avoid shrinking output. when using filters multiple times. the output is shrunk significantly.
    2. information from edge pixels (e.g. upper most left pixel) are less used. Unlike middle pixels which are used more often.
  - Valid convolution: no padding.
    Same convolution: pad, so that the output size is the same as the input size.

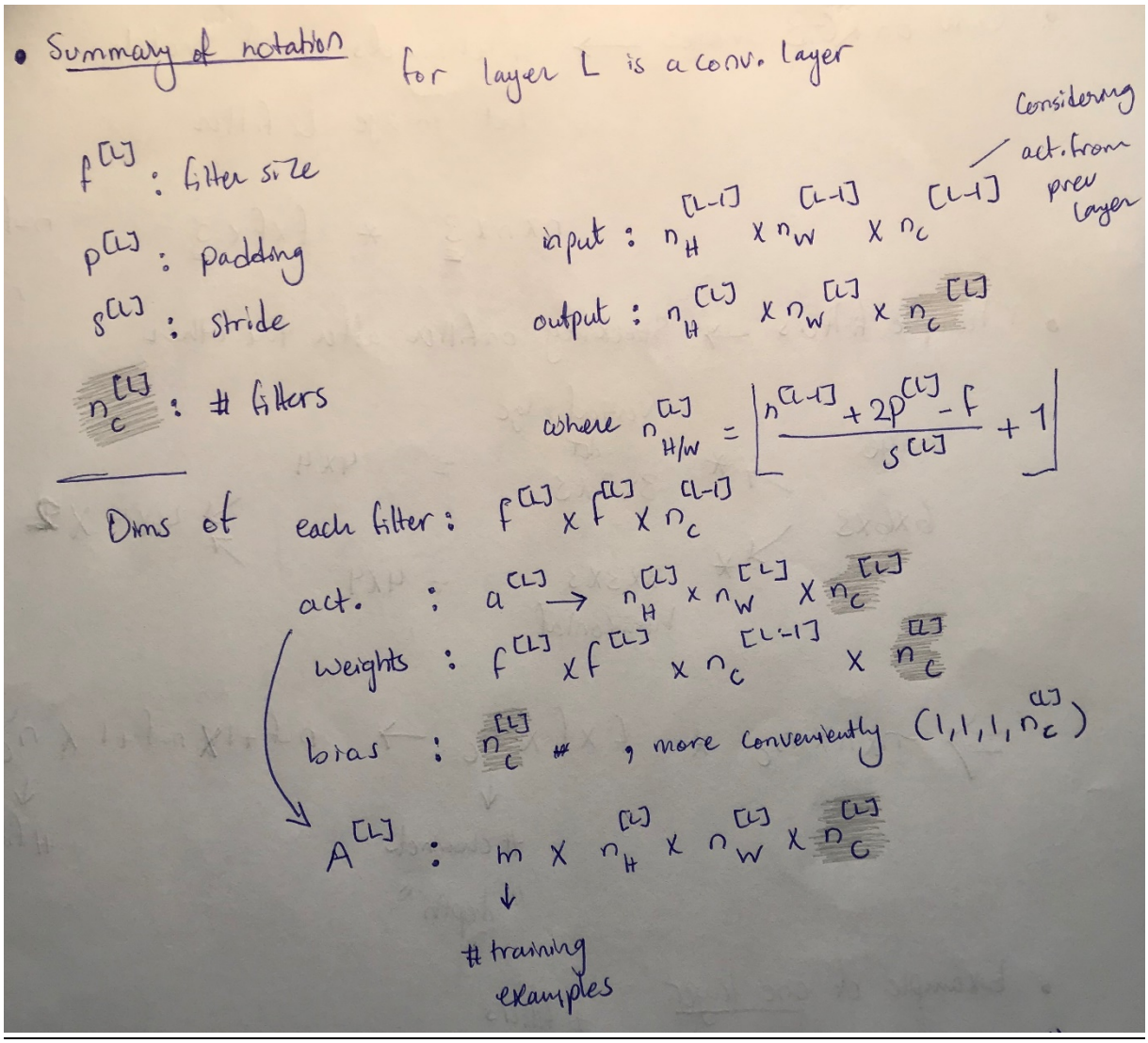
- Striding is the # pixels the filter move during convolution.
  - $output\ size = \frac{n + 2p - f}{s} + 1$
    - n: input size, p: padding, f: filter size, s: stride
    - the output shall be floored: ⌊output⌋ i.e. to round down the output to the nearest integer.
  - Make sure that the filter lies completely within the image dimension. Do not compute outside the padding frame!

- Side Note: technically speaking in math textbooks, *convolution* is done by flipping the filter both vertically and horizontally and then convolute.
  What we actually have been doing is called *cross-correlation*.

- Convolutions over volume
  - **N.B.** # channels (sometimes depth) in the previous input layer (L-1) must match # channels in the current filter layer (L).
  - **N.B.** # filters in the current layer (L) shall be equal # channels in the output layer(L+1).
  - We apply multiple filters; in order to detect multiple edges. for example, horizontal as well as 45 degrees edges.

- Architecture of one layer of convolutional network
  - The architecture goes as follows:
    RELU(input image * filter + bias -----> output * # applied filters
  - Example: for 10 filters of 3*3*3, there is 280 parameters for one layer.
    3*3*3 = 27 + bias (1) = 28 * # filters (10) = 280
  - CNN are less prone to overfitting, due to the large large # of parameters involved.

- Summary of notation



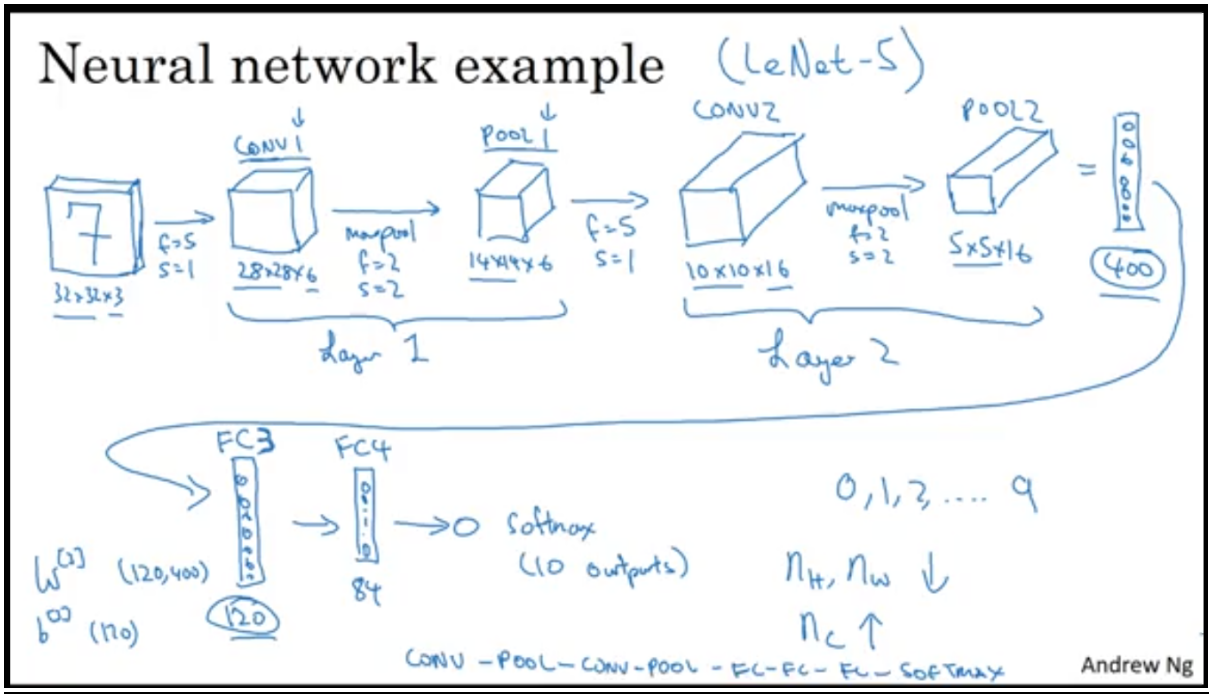
- Some general notes
  - Usually the last output from CNN is flattened and then an activation fn is applied.
  - For now, height and width should decrease over layers, while the depth shall increase.
  - There are 3 types of layers:
    1. Convolution (CONV)
    2. Pooling (POOL)
    3. Fully-connected (FC)

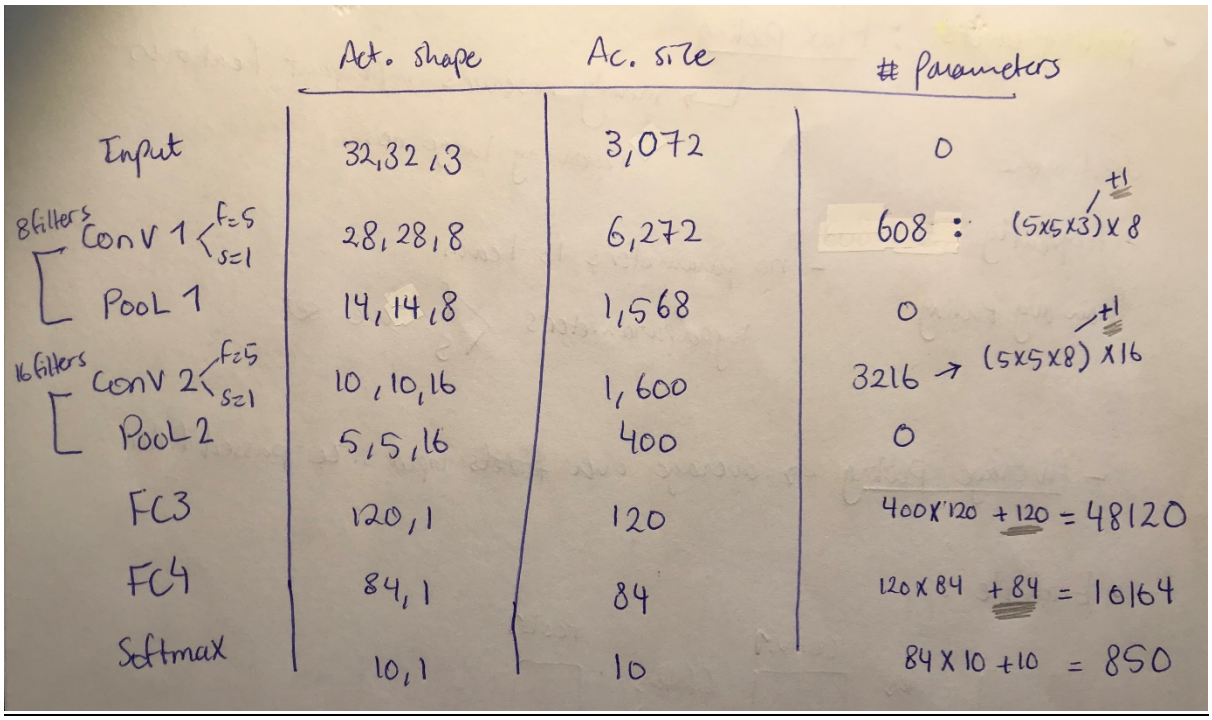
- Pooling Layer
  - It's mainly used to reduce the height and width of the layer.
  - It's a technique used to reduce info in image, while maintaining features.
  - *Max Pooling*: think of it as it detects features of high importance.
  - It reduces the height and width of the input. It helps reduces the computation, as well as helps make feature detector more invariant to its position in the input.

    N.B. It has a set of hyperparameters (mostly, f & s),
    yet no parameters/weights to learn through backprop!
    N.B. f = 2 & s = 2 will reduce the height & width with scale of 2.
  - *Average Pooling*: it takes the average of each filter convoluted with the image, instead of the max.

- Usefulness of CNN
  - 2 main advantages of CNNs over FCs are:
    1. *Parameters Sharing*. A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
    2. *Sparsity of Connections*. In each layer, each output value depends only on a small # inputs.
    3. *Translation Invariance*. A cat shifted couple of pixels is still recognized as a cat.

- Neural Network Example



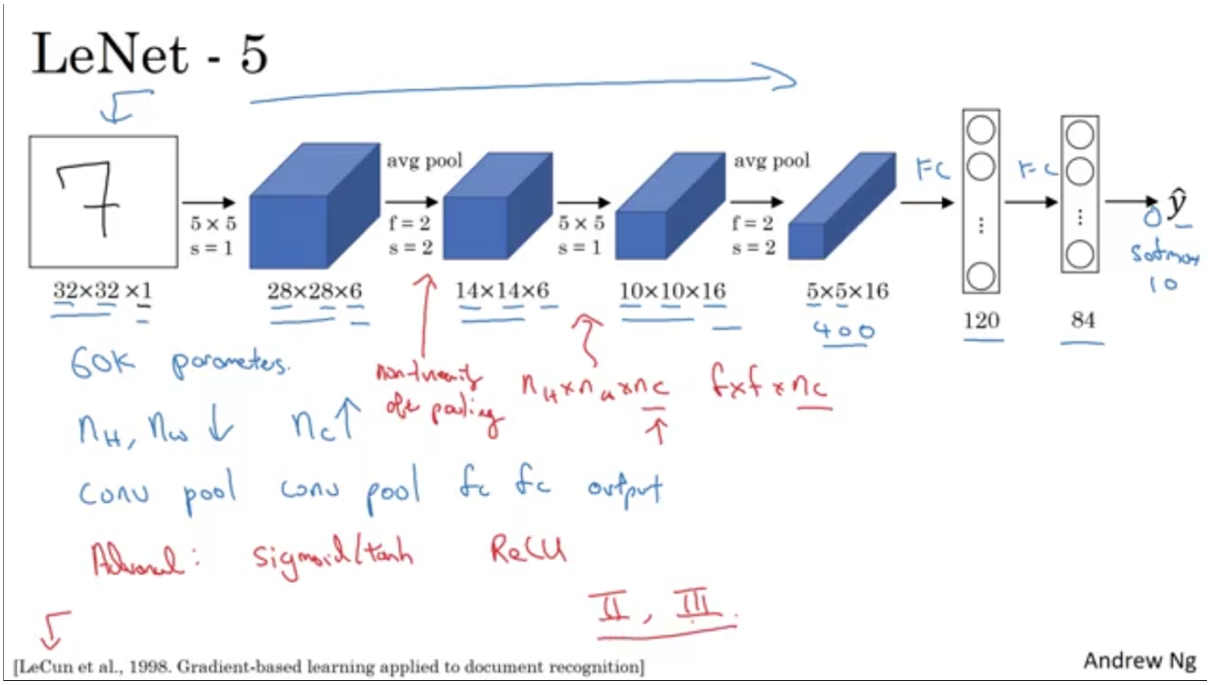


- Remember:
  - the activation size decrease smoothly, if it drops sharply this is usually not good.
  - POOL layers have no parameters to tune.
    N.B. # parameters = (f * f * # channels in (L-1) + bias) * # filters in (L)

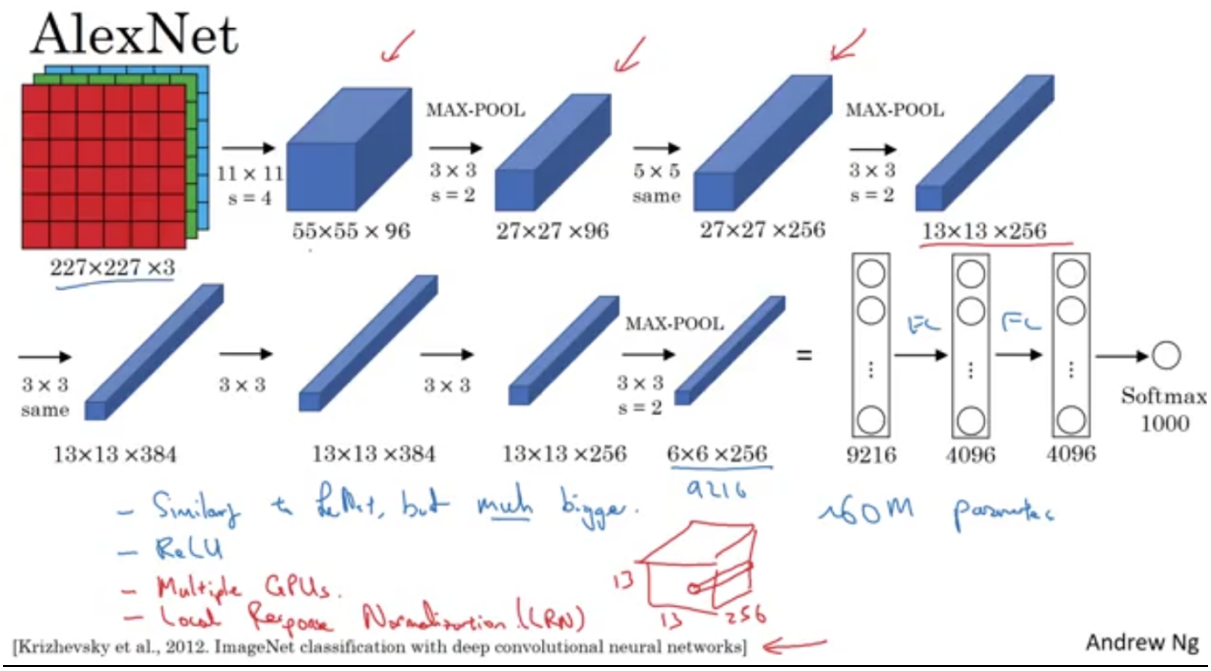
2. Deep Convolutional Models: Case Studies

- Classical examples include: LeNet, AlexNet, VGG, ResNet.

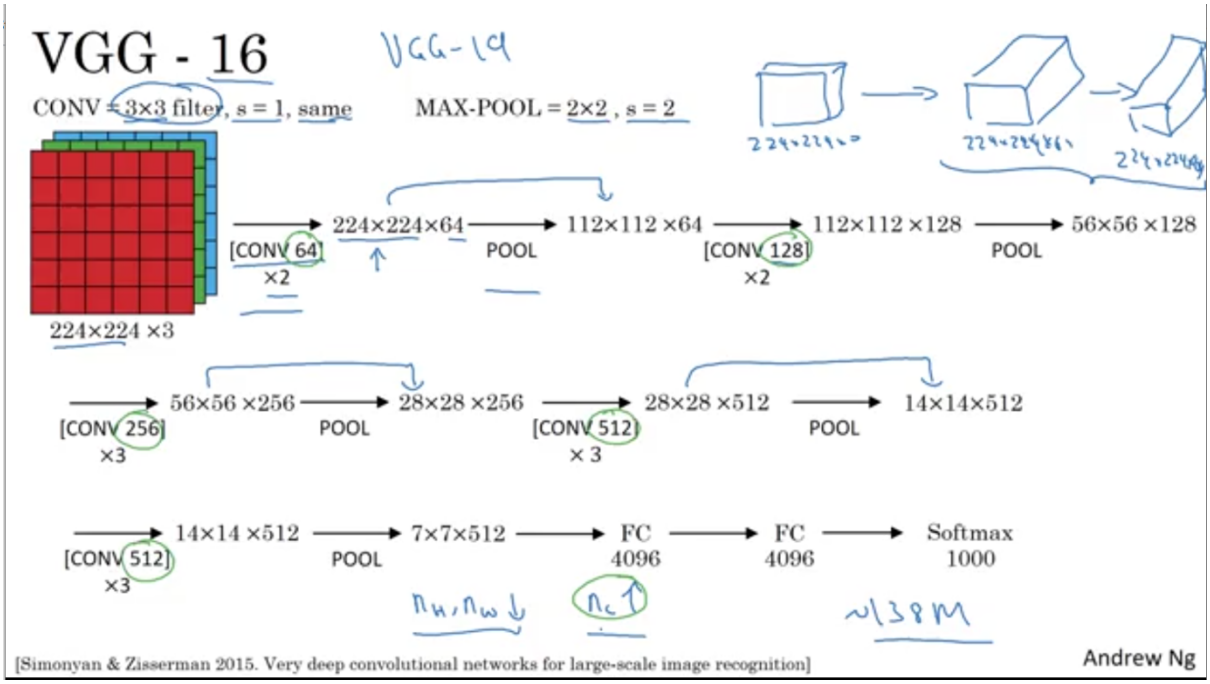
- LeNet-5
  - Mainly to recognize hand-written images of numbers.



- AlexNet



- VGG-16



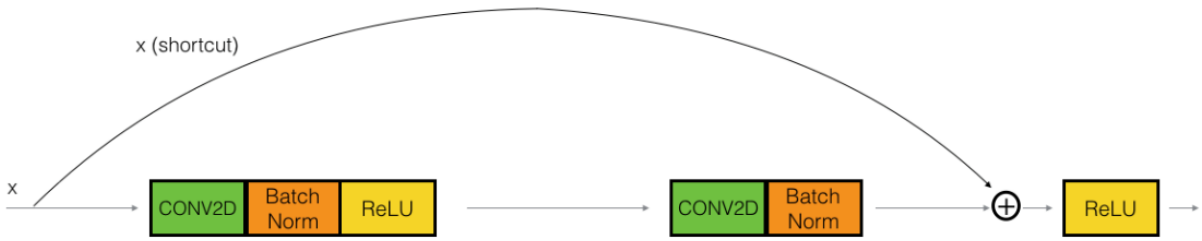
  - Number of layers that have weights are 16.
  - [CONV 64] * 2: means 2 Conv layers, each with 64 filters.

- Very Deep Neural Networks
  - The main benefit of a very deep neural networks is representing complex functions. It can also learn features at many different levels of abstraction, from edges (at the shallower layers, closer to the input) to very complex features (at the deeper layers, closer to the output).
  - In practice however they are harder to train; because of vanishing/exploding (in very rare cases) gradient problems.

- ResNet
  - Skip Connection is based upon residual block, which allows training much deeper networks.
    As the name suggests: residual block mainly skips connection, be reusing the activation from a previous layer until the adjacent weights learn its weights.

- They skips double or triple layers that contain non-linearities (ReLU) and Batch Normalization in between.
- DenseNets: are models with several parallel skips.
- The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers.
- **N.B.** the main reason these residual nets work is their smooth ability to learn the identity function, i.e. when both 'w' & 'b' are both equal to zero in one specific layer.
  This means that you can stack on additional ResNets blocks with little risk harming the training set performance.
- Imagine the hidden layers actually learn something useful,
  then with res nets it's actually even better to learn.

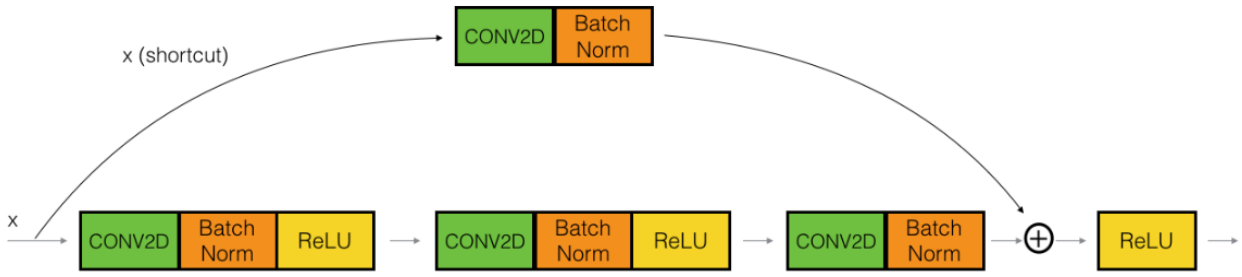
- There are 2 types of ResNets:
  - The identity block is the standard block used in ResNets, and corresponds to the case where the input activation (say $a^{[L]}$) has the same dimension as the output activation (say $a^{[L+2]}$).



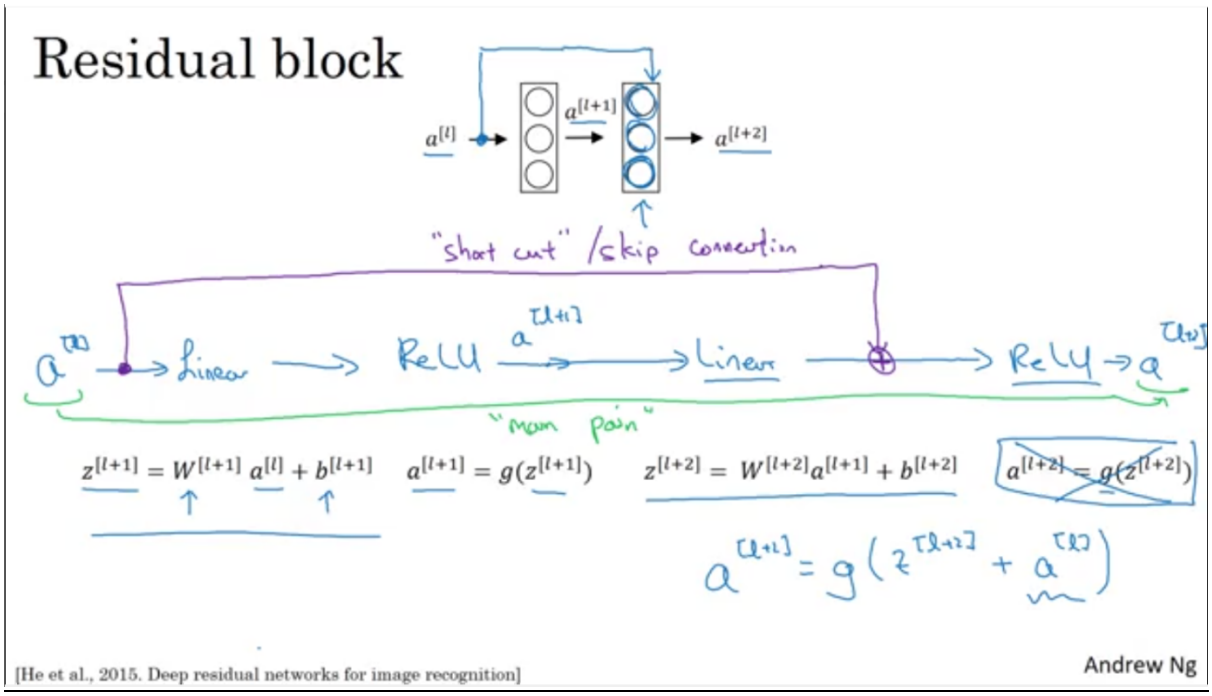
  - The convolutional block can be used when the input and output dimension don't match up. The difference with identity block is there is a Conv2D layer in the shortcut path.
    i. The Conv2D layer in the shortcut path is used to resize the input to a different dimension, so that the dimensions match up in the final addition needed to add the shortcut value back to the main path. (This plays a similar role as the matrix W_s discussed in lecture.

    ii. For example, to reduce the activation dimensions' height and width by a factor of 2, you can use a 1x1 convolution with a stride of 2.

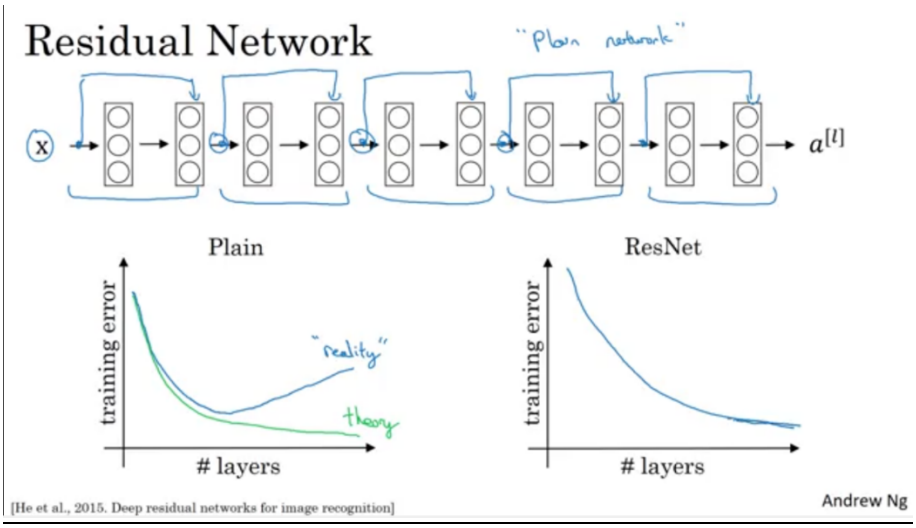
iii. The Conv2D layer on the shortcut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.



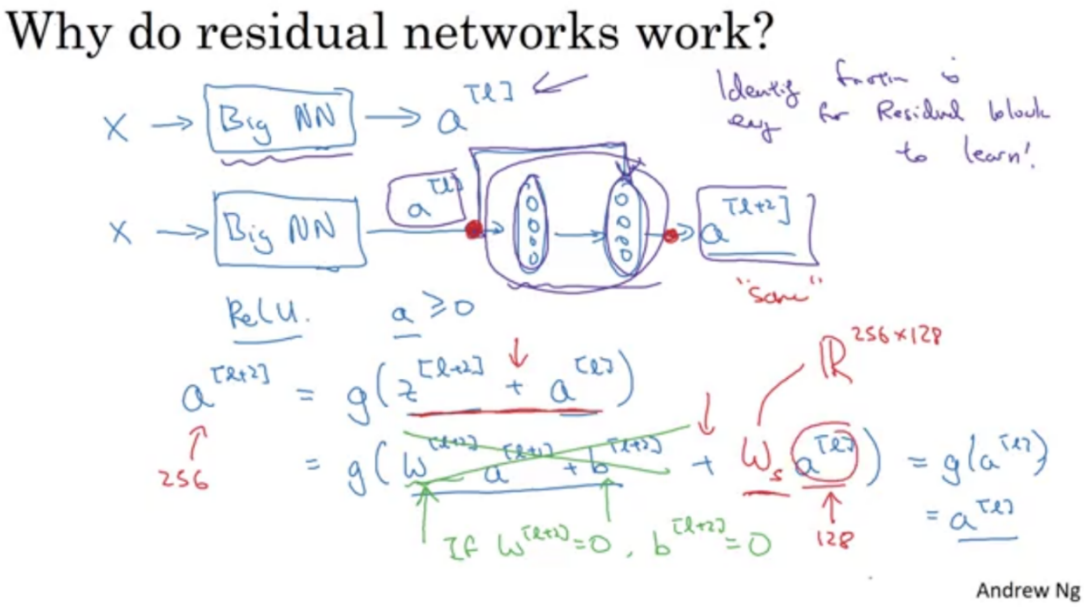
- Residual Block



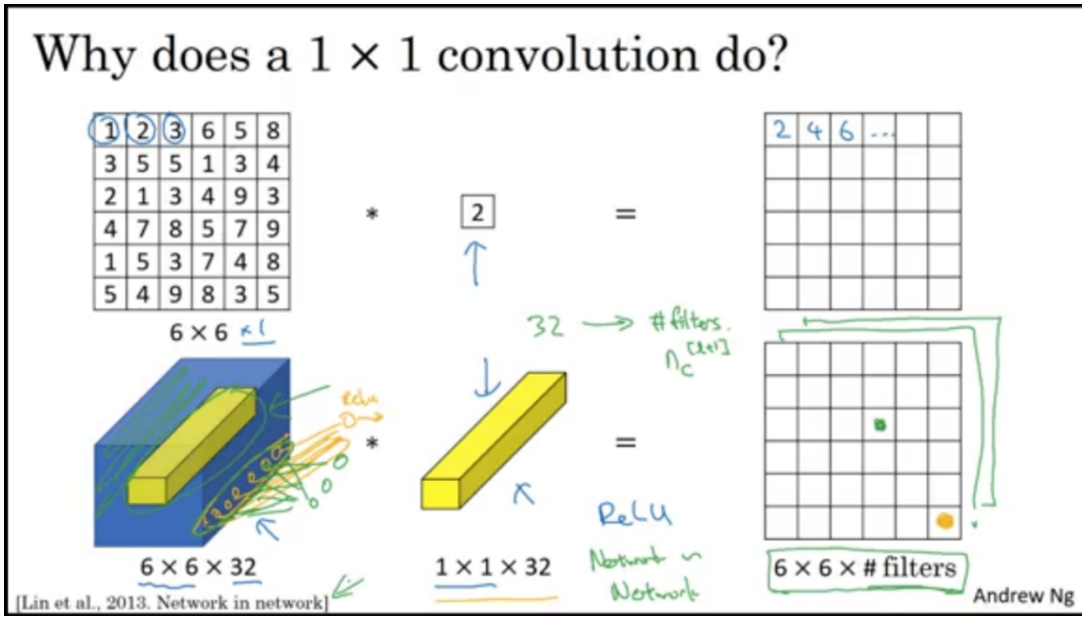
- Plain networks vs. Residual networks



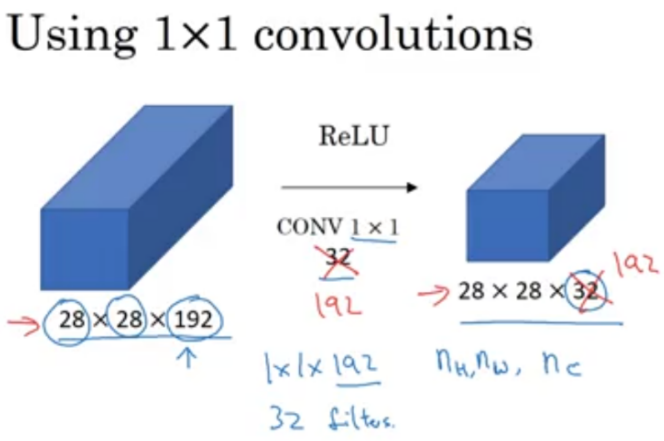
- Comments on ResNets
  - One assumption is done here is that both $z^{[L-2]}$ & $a^{[L]}$ have the same dimensions. Convolutions used are lots of 'same conv'.
    If the dimensions don't match, then another matrix 'w_s' shall be added, as seen in the picture.
    'w_s' could be a matrix of parameters we learnt, or it could be a fixed matrix that applies zero padding to $a^{[L]}$.
  - There is also some evidence that the ease of learning an identity function accounts for ResNets' remarkable performance even more than skip connections helping with vanishing gradients.



- 1*1 CONV
  - It scales the input, if # channels is 1.
    However if # channels is 32 for example for both the 1*1 CONV and the input, the output would be n * n * # filters ($n_c^{[L+1]}$).
  - think of it as one neurons that's taking its input from a previous 32 neurons at a position -for example (1*1), and multiply them by 32 weights and then applying ReLU to them.
  - the green drawing below represent doing the same with multiple filters.

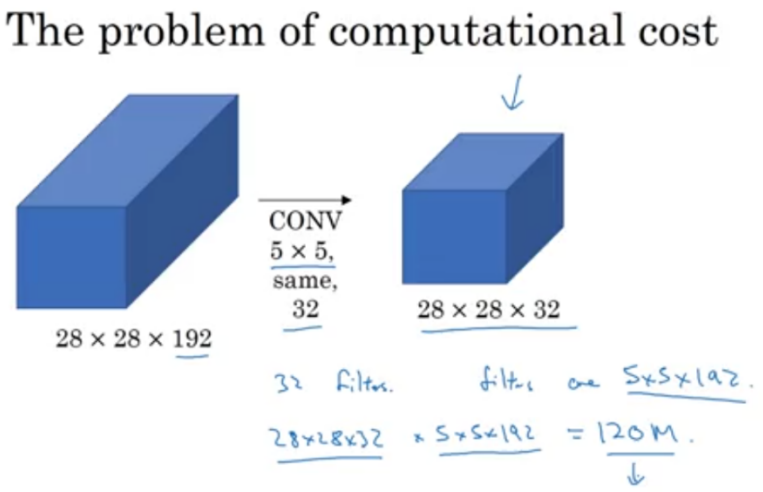


  - 1*1 Conv is mainly used to <u>shrink (or increase</u> if you want) <u># channels</u>.
    # 1*1 Conv filters applied is desired # channels for the output.
    technically speaking the filter would be 1*1* # channels of previous layer
    N.B. Pooling layer was the one used to shrink the height and width.
  - The effect of using 1*1 Conv with the same # channels for the previous layer, would be as <u>adding non-linearity</u>. i.e. learning more features.
    In this case, the input dim will be the same as the output. (red comments below)

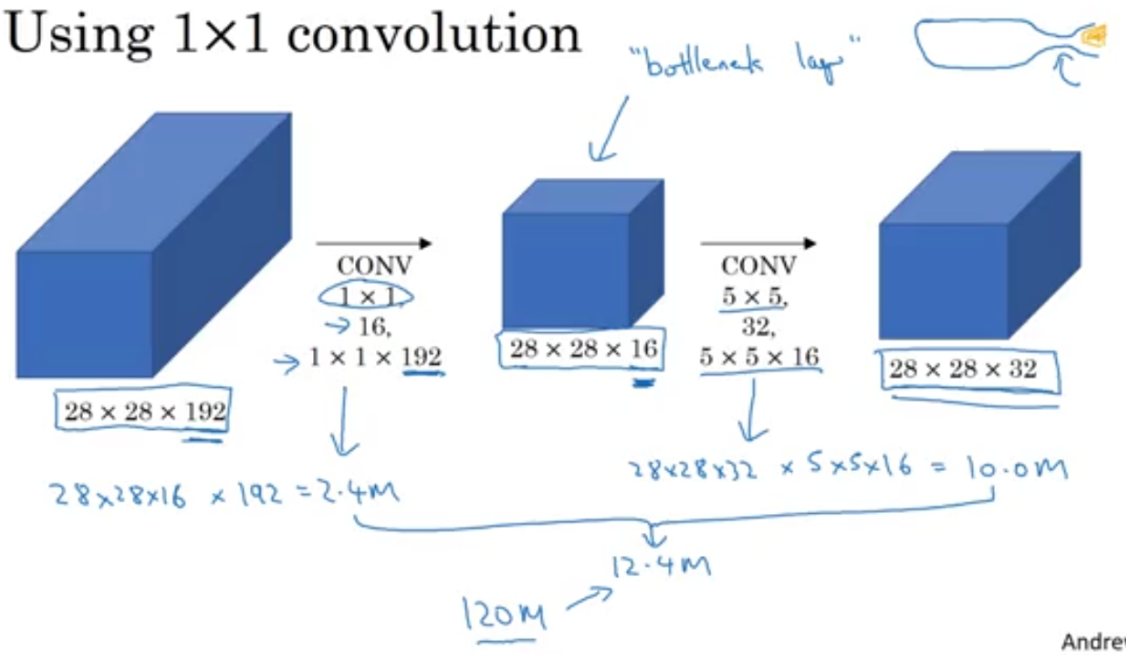


- Inception Networks
    - the main idea is to combine the benefits of Conv and Pool layers. Instead of using one conv/pool one at a time and check for which filter sizes to use, we are going to stack them all and let the network learn whatever parameters are.
    - One Con: the computational cost!
    - Figure below:

32 filters, each of dim: 5*5*192
output dim: 28*28*32
Hence, total parameters to be calculated will be 28*28*32 * 5*5*192 = 120 *10^6!

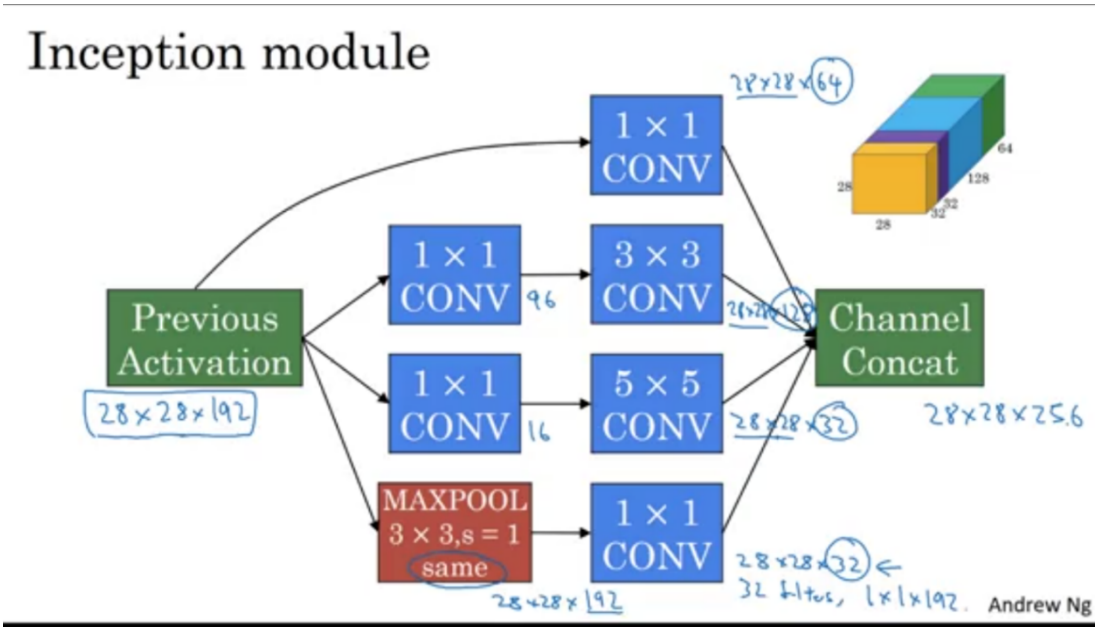


N.B. Using 1*1 Conv will significantly decrease the computational cost.



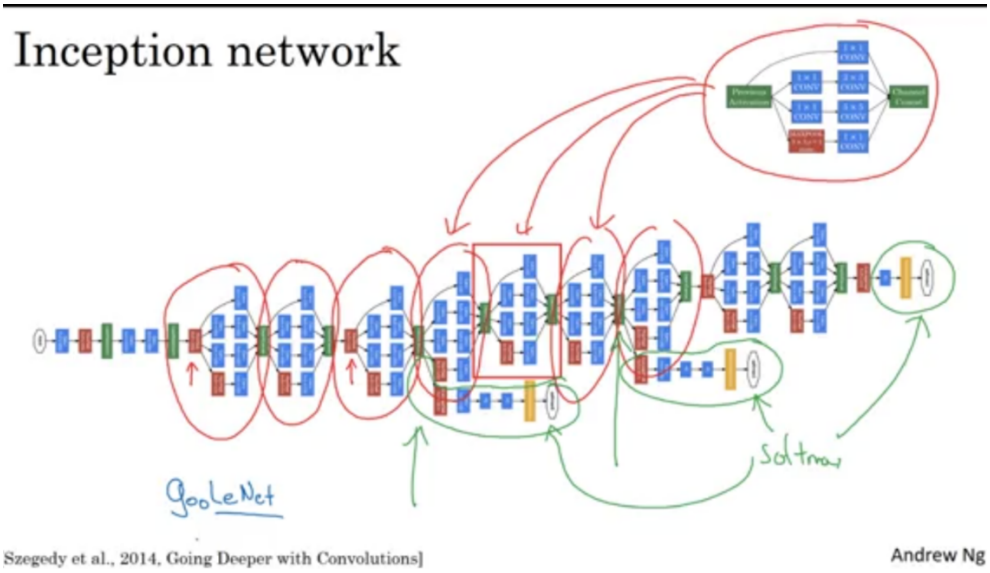
N.B. shrinking down the representation size significantly does not hurt performance as long as you implement the bottle neck (1*1 conv) layer.

- The architecture of Inception Module would go as follows:
    - the final # of channels is the concatenation of the previous 4 channels:
      64 + 128 + 32 + 32 = 256
    - 1*1 conv is done after the MaxPool layer;
      in order to shrink # channels.

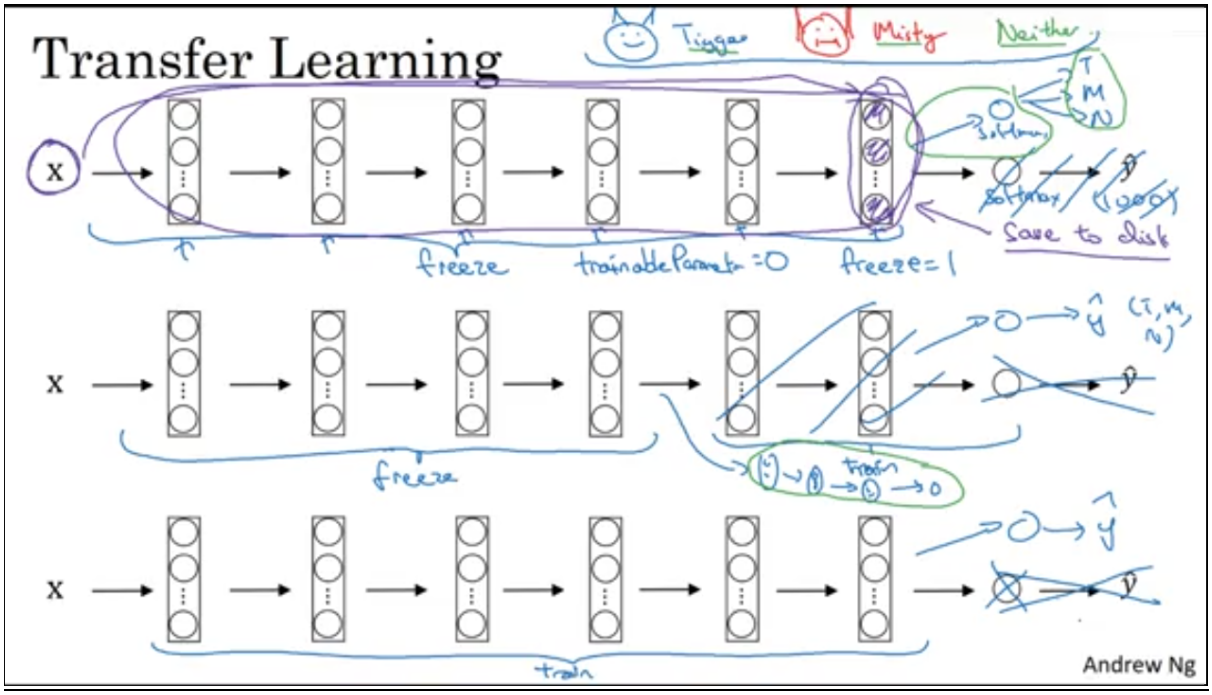


- Inception Network
    - the side branches in the network (figure below), takes some hidden layers and try to use that to make a prediction. In other words, it ensures that even within hidden layers, the features computed are not too bad to predict the output class.

  this appears to have a regularizing effect on the inception network and helps prevent this network from overfitting.



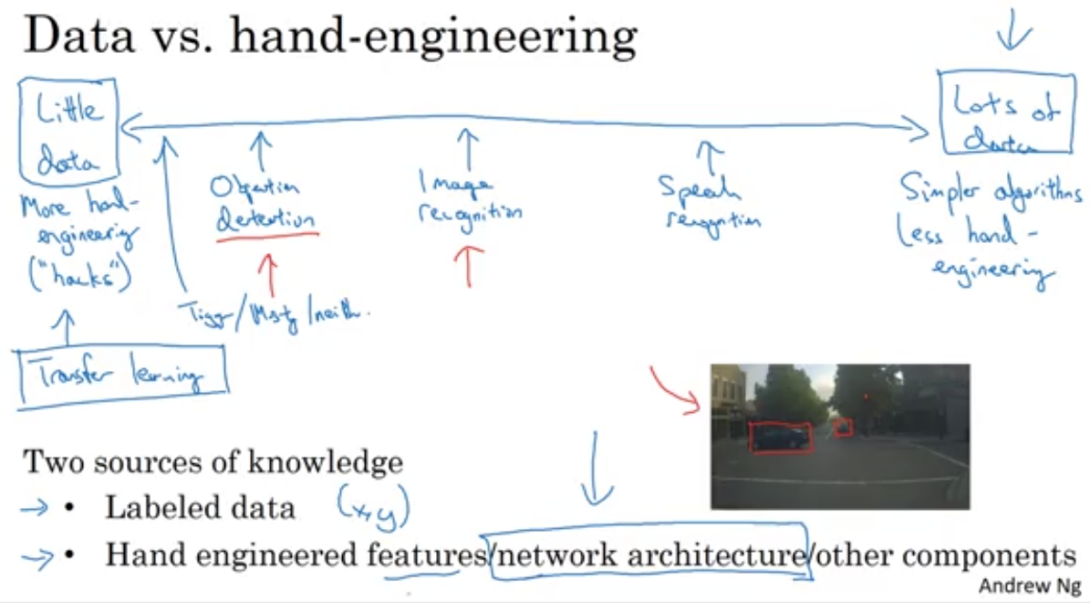
- Transfer Learning



- Data Augmentation
    - common methods: mirroring, random cropping, rotation, shearing, local wrapping ..etc.
    - Colour Shifting, adding some distortions to RGB channels.

  "PCA Colour Augmentation"

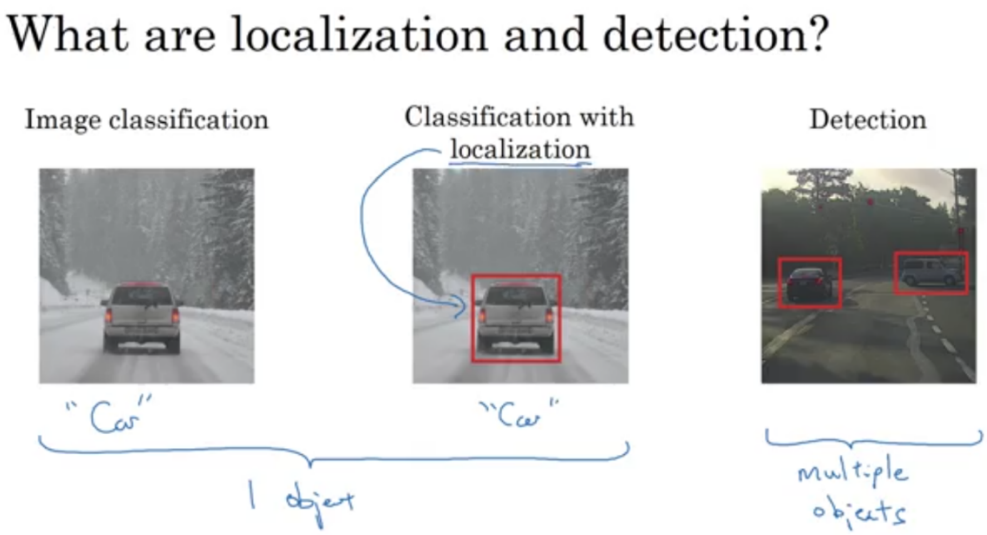
- Data vs Hand-Engineering



- Tips for doing well on benchmarks/winning competitions
    - The techniques do not do well on actual system, more on benchmarks.
    - *Ensembling*:
        - training several networks (3-15 on average) independently and average their outputs, not their weights. Yet it is computationally costly.
    - *Multi-crop at test time*:
        - run classifier on multiple versions of test sets and average their results.
    - *Open-source code*:
        - use architecture of nets published
        - open source implementation, if possible
        - pretrained models and fine-tune your data

3. Object Detection

- Localization vs Detection



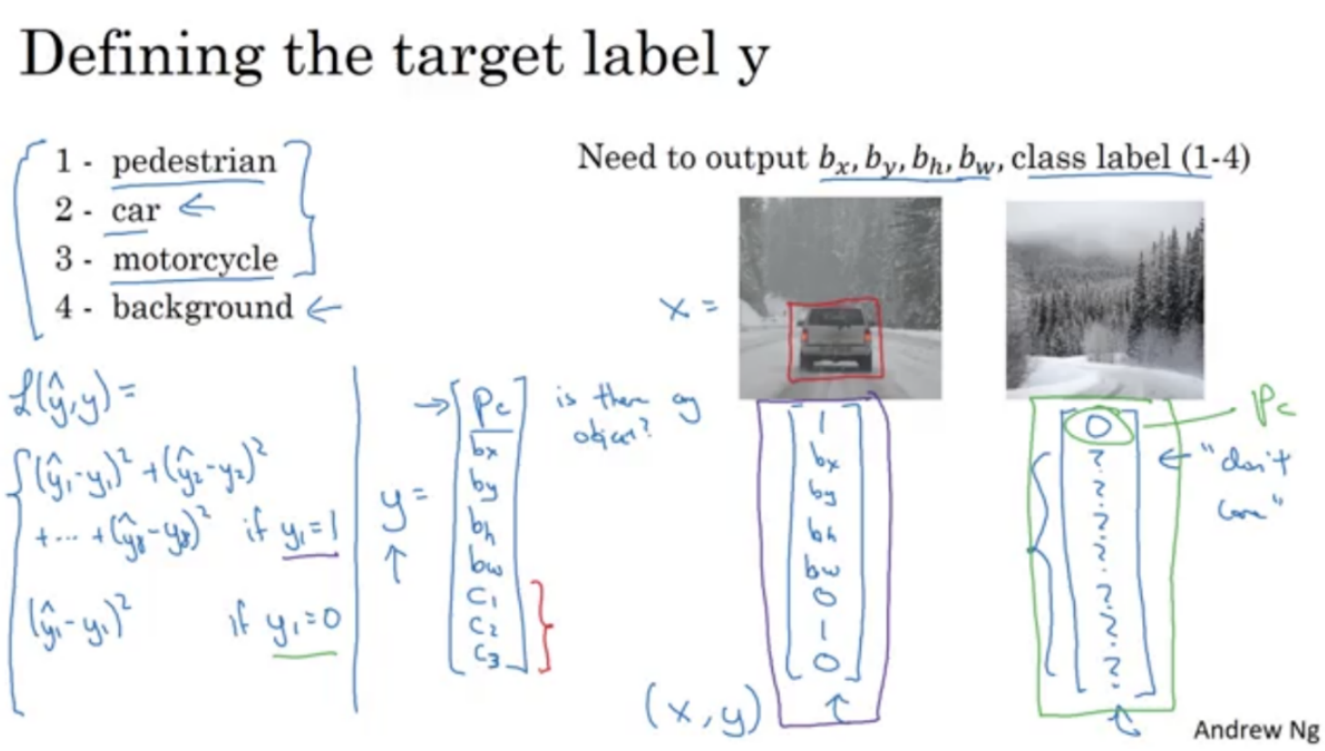
- Object Localization

  - In classification with localization, the output of the CNN is not just a class label, another output is the boundary box with parameters: $\mathbf{b_x}$, $\mathbf{b_y}$, $\mathbf{b_h}$, $\mathbf{b_w}$

  $\mathbf{b_x}$, $\mathbf{b_y}$ represent the origin of the box
  $\mathbf{b_h}$, $\mathbf{b_w}$ represent the height & width
  by convention, the upper left of the box is (0,0), while the lower right is (1,1)

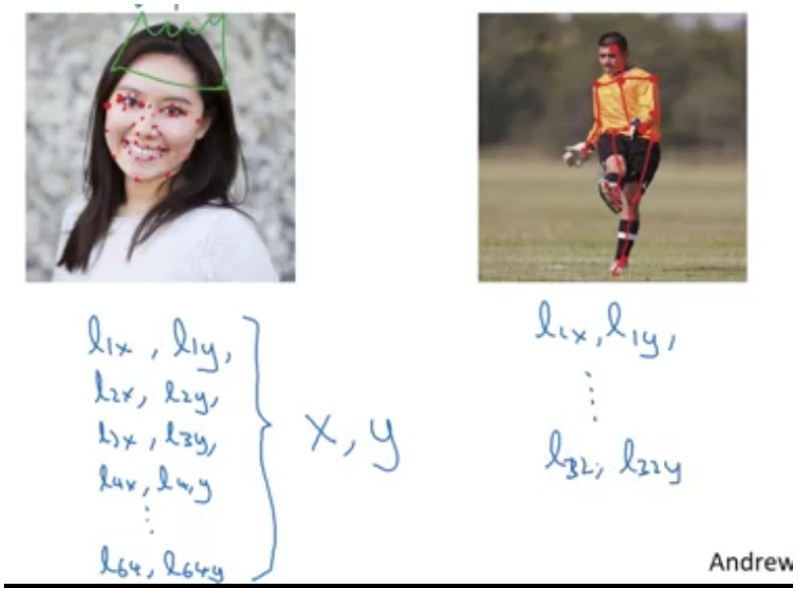
  - You can specify more classes, than those 3, if you want.

  - Assumption: we are assuming only one object per image.
  - For error functions:
    - squared error was used for simplicity reasons.
    - $\mathbf{c_1}$, $\mathbf{c_2}$, $\mathbf{c_3}$, we use log likelihood loss.
    - $\mathbf{b_x}$, $\mathbf{b_y}$, $\mathbf{b_h}$, $\mathbf{b_w}$, squared error or alike.
    - $\mathbf{P_c}$, logistic regression



- Landmark Detection

  - is also known as object localization.
  - sometimes you want to detect landmarks in an image as well.
    e.g. recognizing emotions in a face or pose positions.
  - landmarks should be consistent across images. For example, L1 shall be the left edge of the left eye in a face, in all images.



- Sliding Window Detection

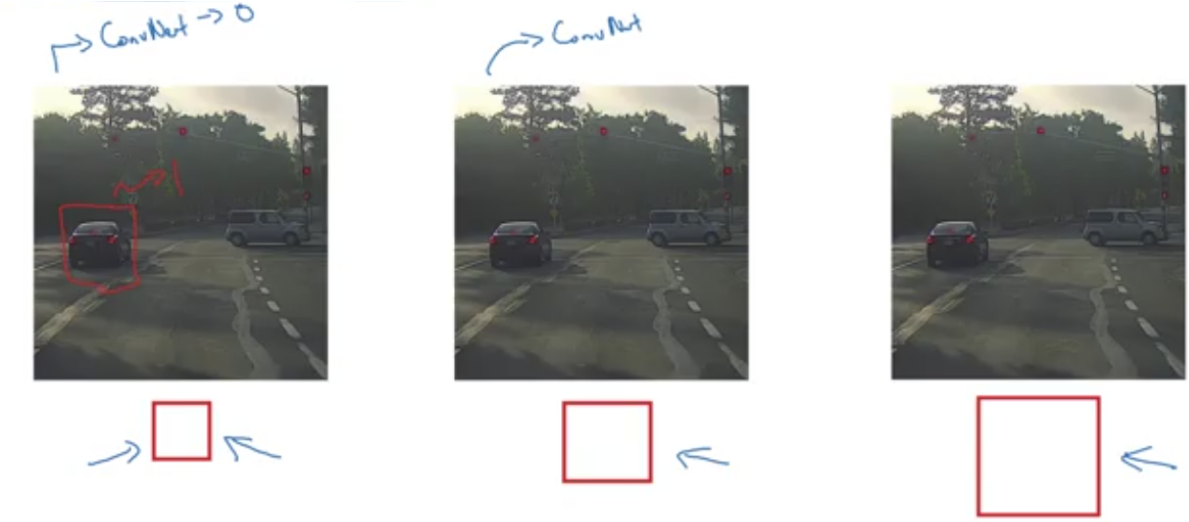
    - Main building block in object detection.
    - Object Detection is detecting multiple objects in the image using 2 main concepts: classification and localization.
    - The training set is built up from closely-cropped images of a specific object that we target to be detected. We then train a conv net to accurately predict such objects.

    for example, we target to detect cars. So we train cropped images of cars though conv net

    - we choose some window size, and pass it through the conv net we had already built. we start sliding this window over the entire image.

    N.B. we run conv net each time a different sliding window is passed through.
    i.e. for each window slide, we would normally apply a conv net; to determine if the window has the object we are interested in.

    - the sliding window is resized multiple times; hoping that the object will be included one time.



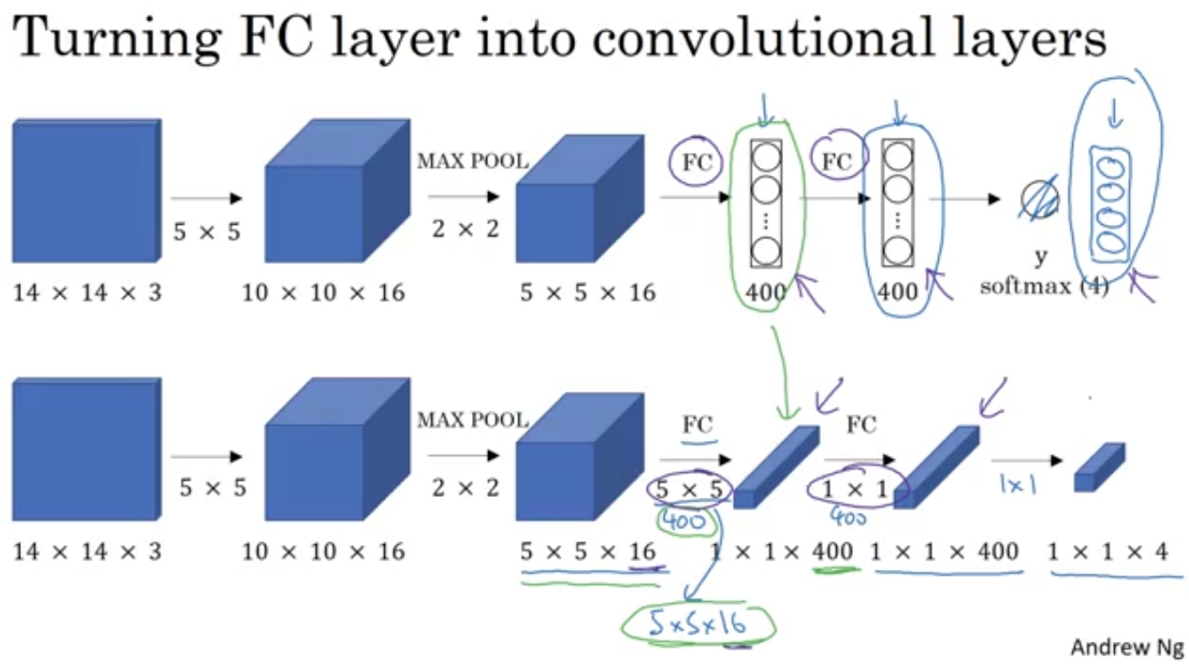
    - One con here is the computational cost of such a process. Using more fine granularities (smaller strides) would allow for more precision and higher probability of object detection, yet it would be expensive.

    On the other hand, with larger strides you might miss the object.

    - One solution to this, is to use conv layers.

- Convolutional Implementation of Sliding Window

    - Turning FC layer into conv layers



- The problem arises when your input in the conv net is 14*14*3, while the test set image is of input size 16*16*3, as in the figure below. Usually you'd need to start sliding the 14*14 over the 16*16 for each and every layer till the end.
  Meaning you will have to run the conv net 4 times; to get 4 labels.

- However it turns out that a lot of this computation is highly duplicative. So what the convolutional implementation of sliding window does, is that it allows these 4 pauses (in our case) to share a lot of computation.

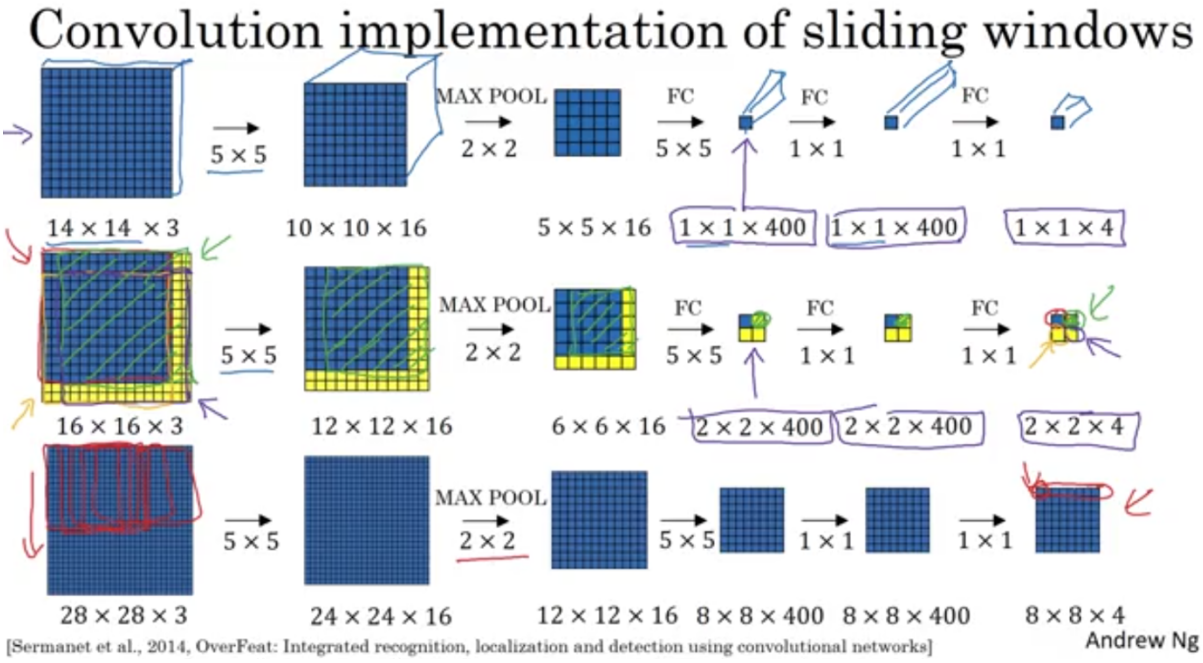
- What can be done to save computation is just to let the conv net run with the same parameters.
  It turns out that this will yield the same result as if you started with the

original 14*14 and kept sliding over all the network layers. What's different here is saving a lot of cost with this method.

- More illustration: the final 2*2*4 represent 4 grid cells, the blue one (upper left) actually is the same as the blue sliding window if we were to let the 4 runs of conv net being executed.
  And so on, the green cell (upper right) was found to yield the same activations across different layers, as if we took the original 14*14*3 and pass it through the earlier conv net.

- in the figure below, the last example of 28*28*3 is for more illustration.

- Conclusion: instead of running forward prop sequentially on 4 subsets of the input image independently, you can just combine all 4 in one computation and share it across common regions.
- One Con here is that the position of the boundary boxes will not be too accurate.
  the object might not be fully included in the boundary box for example.



- YOLO Algorithm (introduction)

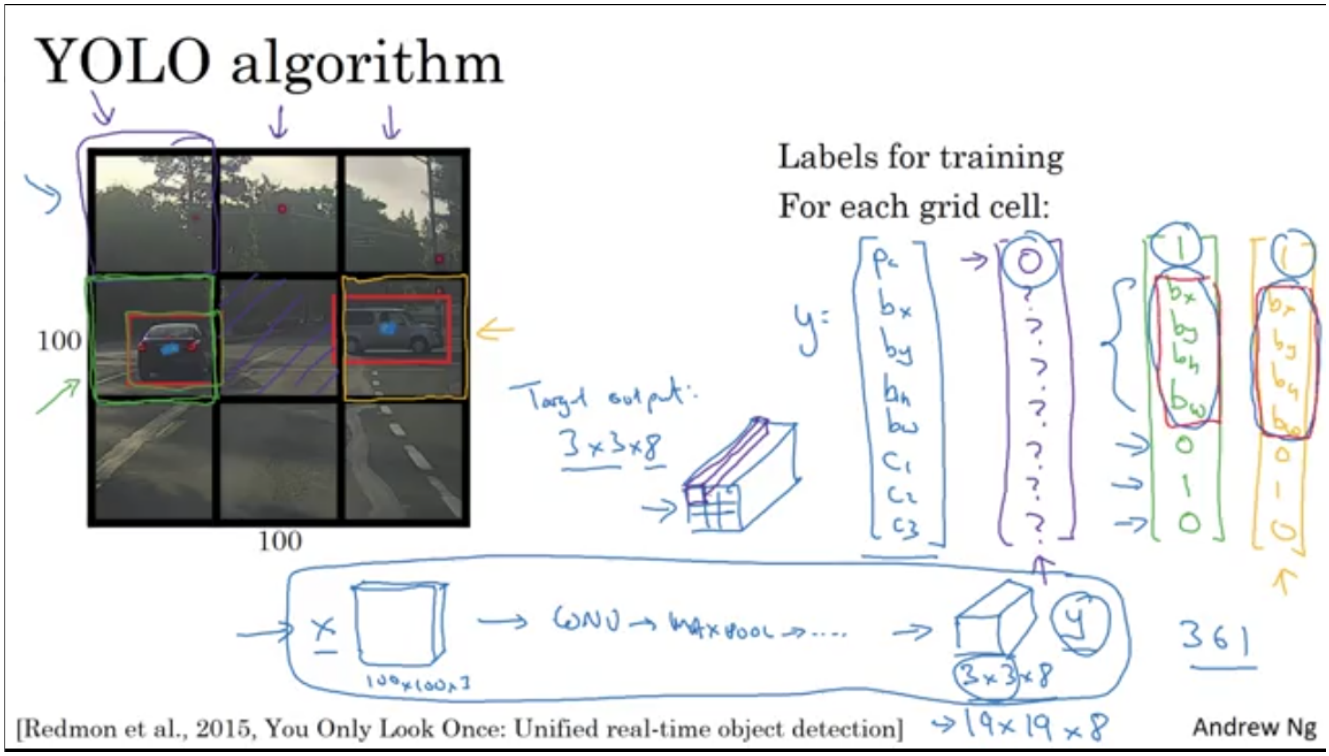
    - YOLO: You Only Look Once.
    - it is used to accurately output boundary boxes.

    - How it works: for one image as shown below, we are going to divide it into 3*3 grid cells, usually 19*19 would work better.
      For each grid cell, we are going to specify label **y**, which would include 8 parameters as previously-mentioned through object classification and detection techniques.
    - It requires only one forward propagation through the network to make predictions.

    - The target output shall be 3*3*8.
      3*3 is # grid cells, for each cell there is 8 dimensional vector

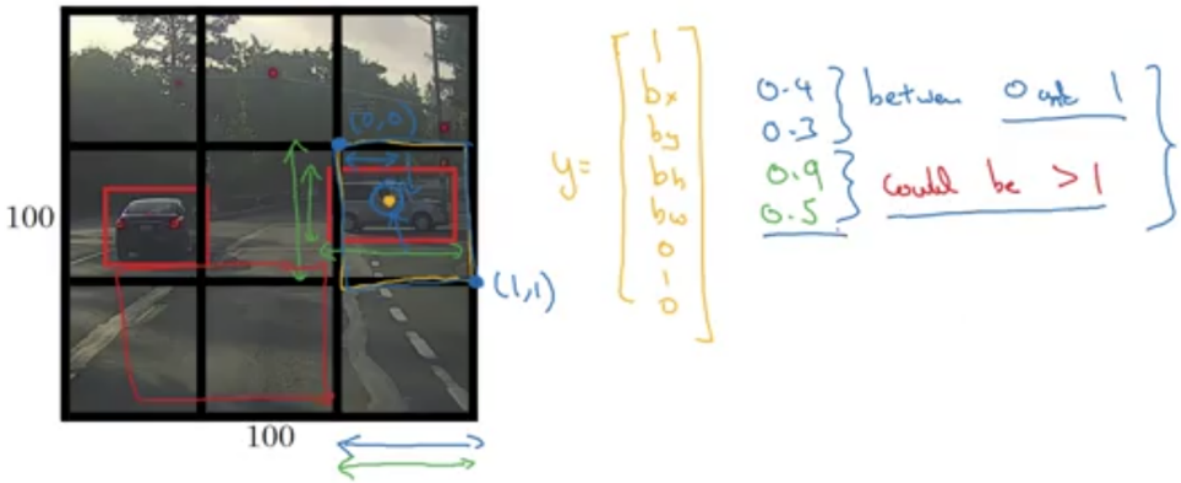
    - The object is detected according to its midpoint, even if it spans across multiple cells, we only consider the cell with its midpoint.
    - This method allows for more accurate estimate for box.

    - One Pro is that the conv net prediction of boundary box is accurate.
      As long as you have only one object in each grid cell, this alg shall work fine.
    - Another Pro is that it again shares computation across the only one conv net. you're not implementing the alg. 9 times on the 3*3 grid. It's very efficient and used in real-time object detection.



- specify the box boundary
  - $\mathbf{b_x}$, $\mathbf{b_y}$ are between 0 & 1; since this is the box boundary we considered before.
  - $\mathbf{b_h}$, $\mathbf{b_w}$ could be more than 1; it could extend outside the cell.

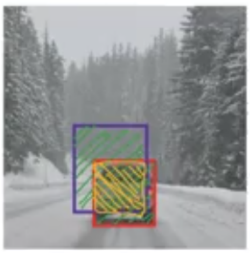
Specify the bounding boxes



- Intersection Over Union (IOU)
    - It is mainly used to <u>evaluate object detection</u>
      For example, if the blue box below is the box predicted and the red one is ground truth, how can we evaluate such an output.
    - On the other hand, it adds another component (“Non-max Suppression”) to object detection to make it work better.

    - It’s the size of the intersection over the size of the union.

Evaluating object localization



Intersection over Union (IoU)

$$= \frac{\text{size of}}{\text{size of}}$$

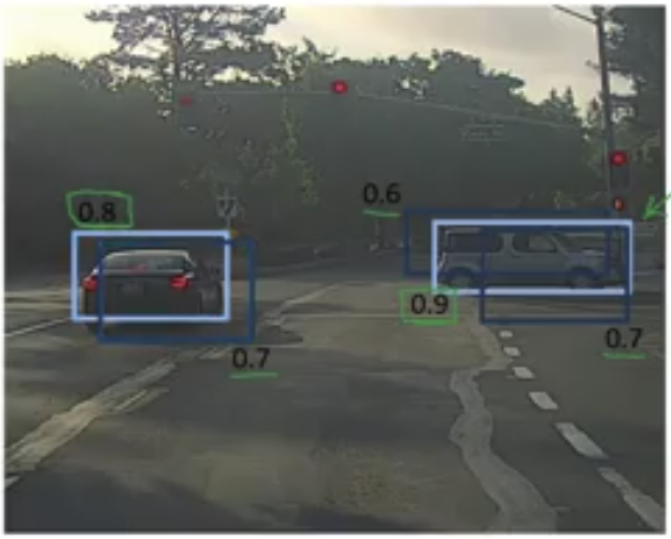
“Correct” if IoU ≥ 0.5 ←

0.6 ←

More generally, IoU is a measure of the overlap between two bounding boxes.

- Non-max Suppresion (NMS)

    - It mainly solves the problem of <u>multiple detections of an object</u>.
      Possibly, many grid cells would detect the same object, forming many boundary boxes on just one object.
      This is what the output from YOLO might look like before using NMS.
    - The alg actually suppresses the windows with lower probabilities and maximize those with higher probs. $\mathbf{P_c}$ is the prob of detecting an object.
    - It shall be run independently on each class you have



- Here's how it works: normally the output shall be 19*19*8, yet we would neglect # classes for now.
- Neglect all the predictions with a specific threshold.
- Pick the highest prediction output.
- For the remaining boundary boxes, we are going to calculate IOU with the pre-selected one (the one with the highest prediction output). Discard those with above a specific threshold.

- The alg works well on just a single object.

If there are more than one object in an image (e.g car, pedestrain, motorcycle), the right thing to do is to repeatedly carry out NMS 3 times; one on each output classes.

Non-max suppression algorithm

19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3

Discard all boxes with $p_c \leq 0.6$

While there are any remaining boxes:

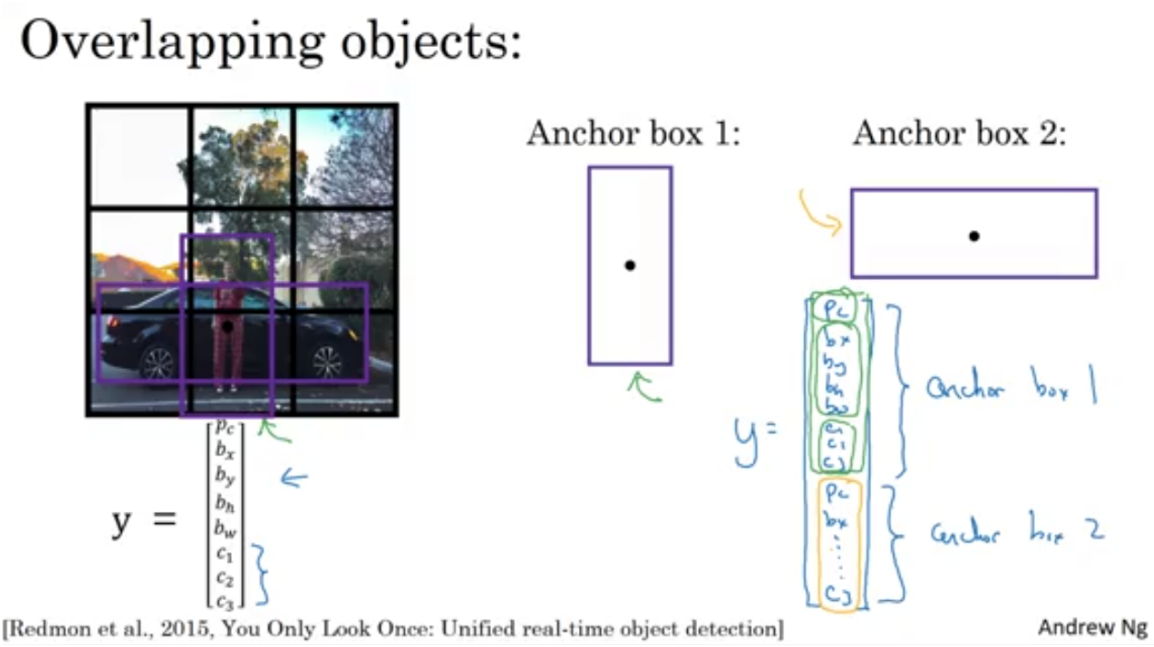
- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

- Anchor Boxes

    - The problem previously was that each grid cell can detect only one object.
    - Anchor boxes mainly solves the problem of detecting overlapping objects; multiple objects in the same grid cell.

    - In the picture below, both objects got the same midpoint.
      We are going to specify 2 or more -according to # objects- anchor boxes for each of the 2 objects

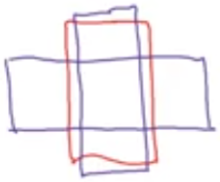


Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y: $3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box)

Output y: $3 \times 3 \times 16$

$3 \times 3 \times 2 \times 8$

Andrew Ng

- Notice: each object is assigned to both; a grid cell and anchor box. This is how the object gets encoded in the output.
  In the picture above, the output would be 3*3*16, since there are 2 anchor boxes each with 8-dimensional vector.

- The alg do not perform well in the following cases:
  - if there are 2 anchor boxes and 3 objects in the same grid cell.
  - if there are 2 objects associated with the same anchor box shape.

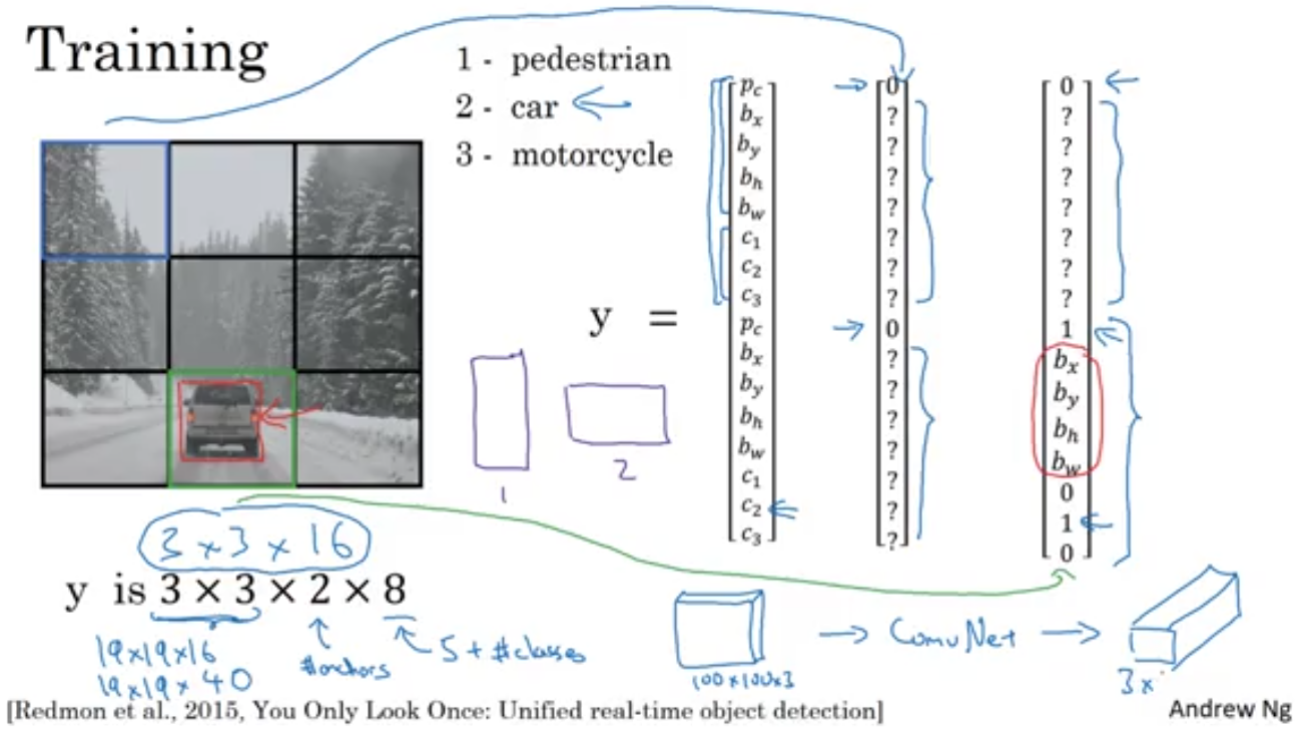
- One pro for the alg is that it allows for better specialization; for example, having thinner-shaped boxes for pedestrians in contrast to wider boxes for cars.

- Anchor boxes are usually chosen by hand, representing common variety of shapes covering types of object you want.

- A more advanced version is to use k-means alg; to group together 2 types of object shapes you tend to get. and then use that to select a set of anchor boxes. the basic approach is to do it by hand.

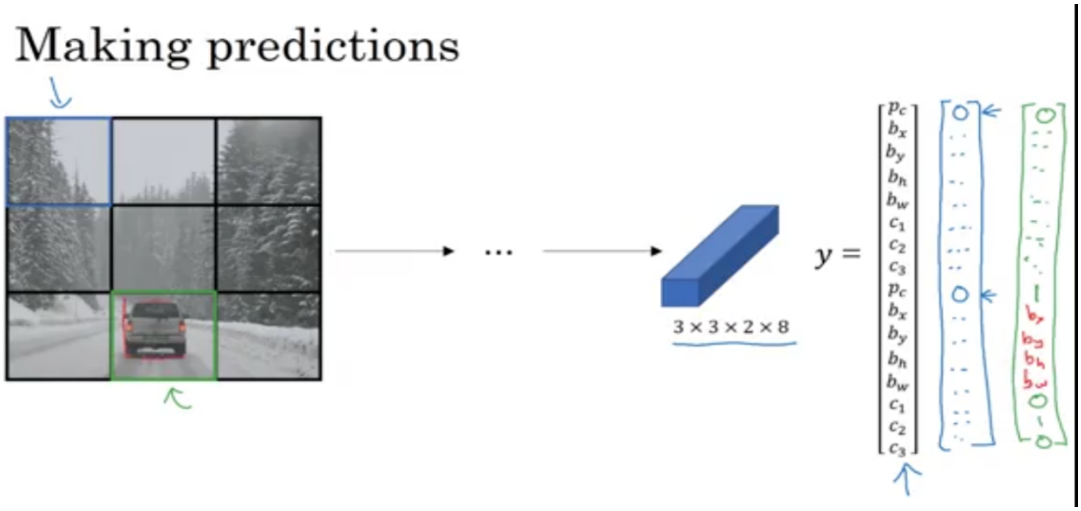
- How YOLO Algorithm works

1. Training



2. Making predictions:

the blue output vector is associated with the blue grid cell (most upper left), the green output vector is similarly associated with the green grid cell, where the car shall be detected.



3. Non-max suppressed outputs:

For each grid cell, you will get 2 anchor boxes. Such anchor boxes can be outside the grid cell.

Outputting the non-max supressed outputs

- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

- Some notes about YOLO alg

  - If the input of YOLO alg is batch of images, the output shall be a list of boundary boxes along with the recognized classes.
  - In the YOLO algorithm, at training time, only one cell ---the one containing the centre/midpoint of an object--- is responsible for detecting this object.
  - Suppose you are using YOLO on a 19x19 grid, on a detection problem with 20 classes, and with 5 anchor boxes.
    Output dimension will be: 19*19*(5*25)
    5: anchors, 25 = 5 + # classes.

    N.B. It's always important to provide the boundary boxes in the training set. No alg can detect such boxes by itself.

  - The YOLO architecture is:

Image (m, 608, 608, 3) ---> Deep CNN ---> Encoding (m, 19, 19, 5, 85).

Each grid cell in the encoding contains information about anchor boxes. Think of it as a matrix of # boxes and # class probabilities.

- More on: Autonomous_driving_application_Car_detection_v3a. (Important).

- Region Proposals

    - One downside for the sliding window idea is that it classifies a lot of regions where there is no object.
    - choosing fewer windows to run your conv net on, it is called R-CNN.

Faster algorithms

R-CNN:	Propose regions. Classify proposed regions one at a time. Output label + bounding box.
Fast R-CNN:	Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.
Faster R-CNN:	Use convolutional network to propose regions.

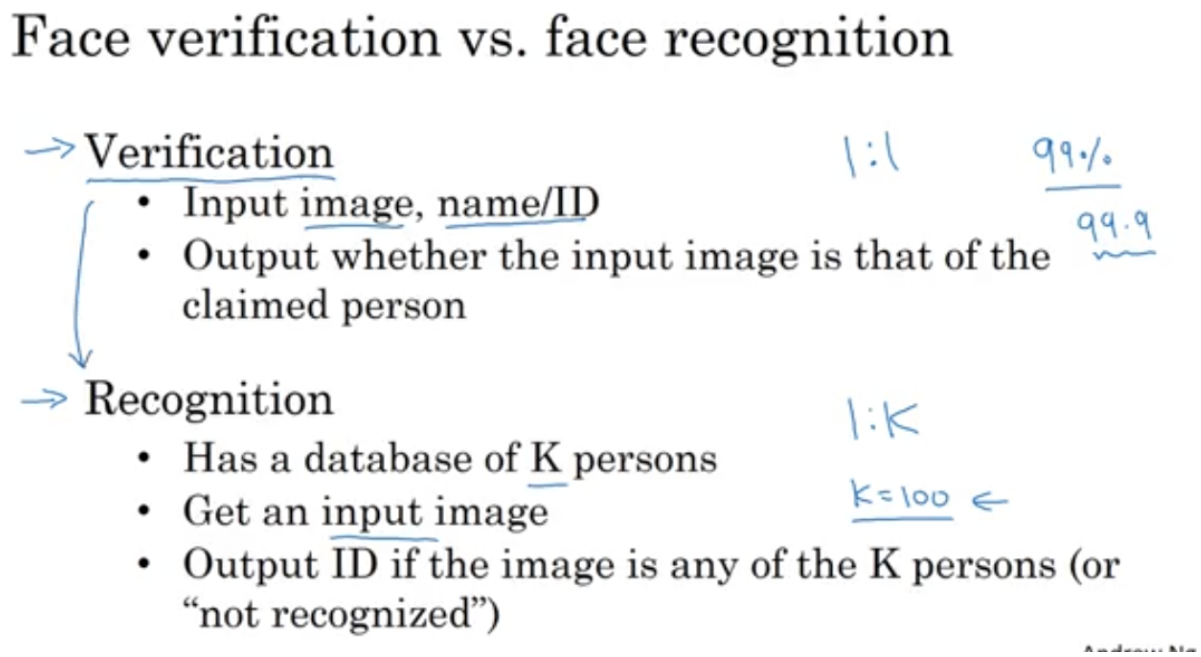
- some general notes

    - When training one of the object detection systems described in lecture, you need a training set that contains many pictures of the object(s) you wish to detect. However, bounding boxes do not need to be provided in the training set, since the algorithm can learn to detect the objects by itself. False. No alg can detect such boxes by itself.
    - Suppose you are applying a sliding windows classifier (non-convolutional implementation). Increasing the stride would tend to increase accuracy, but decrease computational cost. False.

4. Special Applications: Face Recognition & Neural Style Transfer

- Face Recognition

    - Face Verification: “Is this the claimed person?”
      Face Recognition: “Who is this person?”



    - We are going to focus on Verification, as the building block for Recognition.

- One-Shot Learning

    - Recognizing the person’s face using only one image for his face. Unlike most DL techniques, it aims to learn information from one or only few training samples.

    - One classical approach is to pass the image through a CNN ending with SoftMax function for # of persons you want to recognize. Yet this approach is not practical for couple of reasons: first deep learning is not efficient with such small datasets, second what if you want to add a new person.

- One solution would be to learn a similarity function where it measures the degree of difference between 2 images.
  This approach solves the problem of one-shot learning of face verification.
  If the difference is below a certain small number, then the 2 images are similar.

- We learn d(img1, img2); to learn to recognize a new person given just a single image of that person. And to solve the one-shot learning problem.

Learning a "similarity" function

→ d(img1,img2) = degree of difference between images

If d(img1,img2) $\leq \tau$ "same"

$> \tau$ "different"

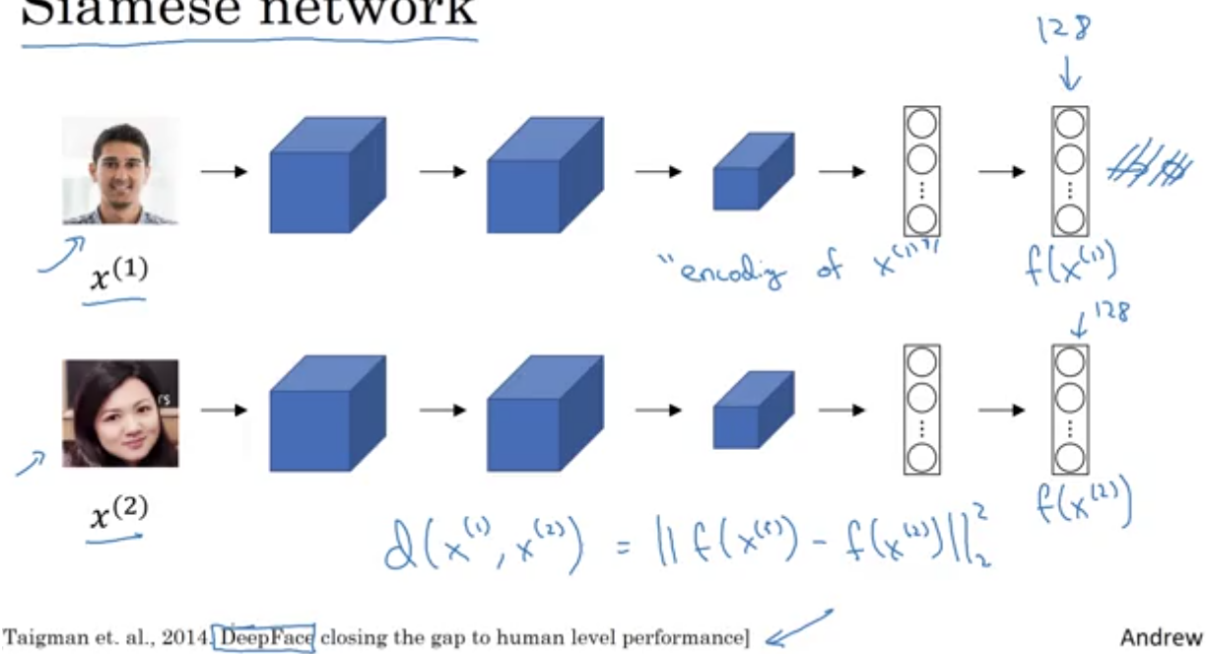
} Verification.

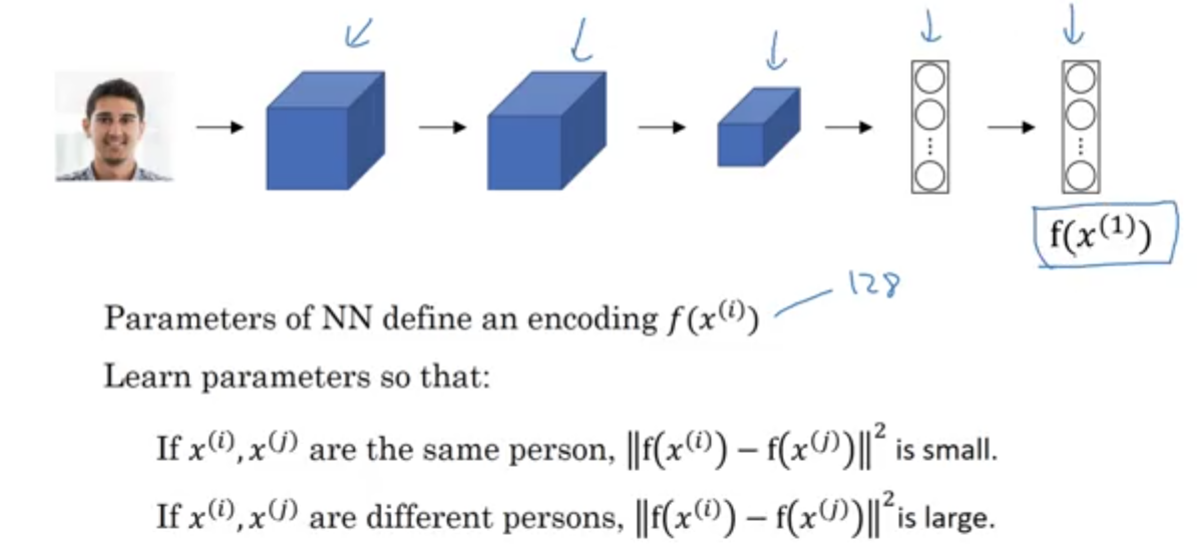
d(img1, img2)

- Siamese Network

  - It uses the same weights while working in tandem on two different input vectors to compute comparable output vectors. This is similar to comparing fingerprints but can be described more technically as a distance function for locality-sensitive hashing

  - It mainly depends on encoding of the input image in the final layer of a CNN.
    The input image is passed through a CNN and instead of outputting a SoftMax fn for example, we are going to output parameters; defining an encoding of that input image.
  - Then, we are going to pass the second image through the same CNN previously used. If the encoding is good representation of the images. We can finally define the distance function as shown in the image below.

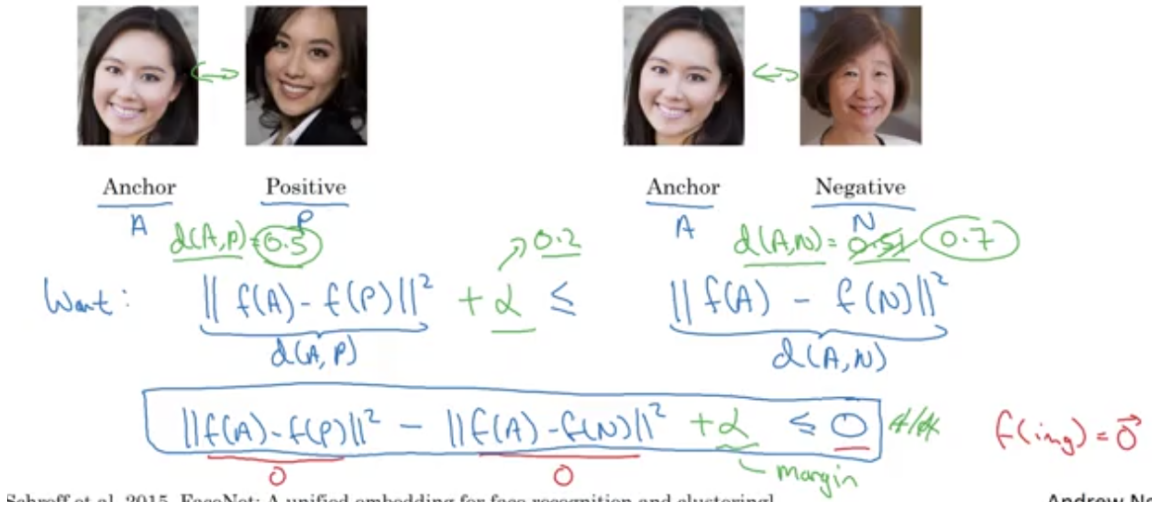




- Triplet Loss

  - It's used to learn the parameters; so that a good encoding of the input images can be produced.
  - Encoding parameters shall vary significantly between different images, while they should match if the images are for the same person.
  - "Triplet" because we are looking at 3 images: Anchor (Baseline), Positive and Negative at a time.

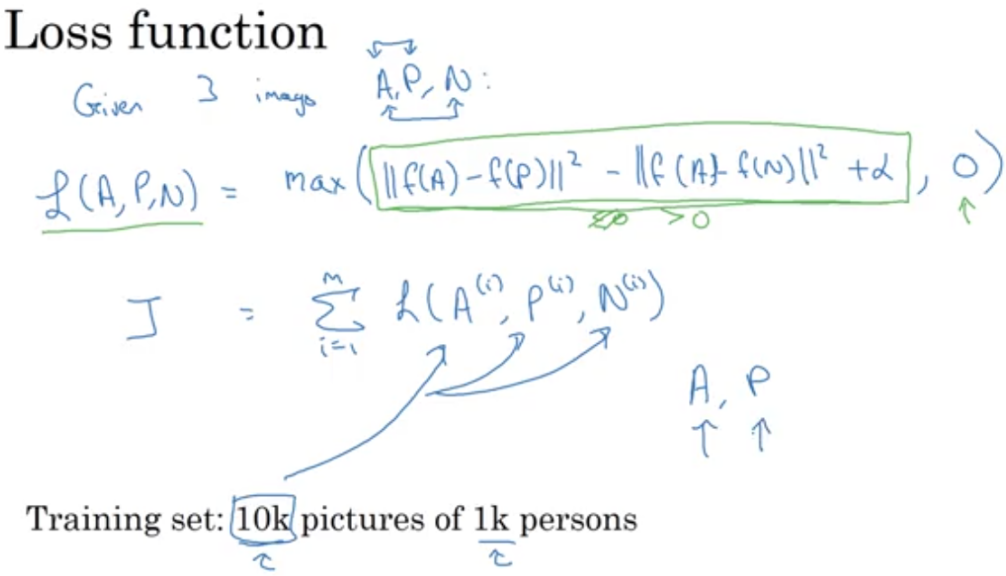
- We add a margin in the final equation -as illustrated below; as we want to make sure that the difference between **d(A,P)** & **d(A,N)** is just large enough.
  In other words, it <u>pushes them further apart</u>.



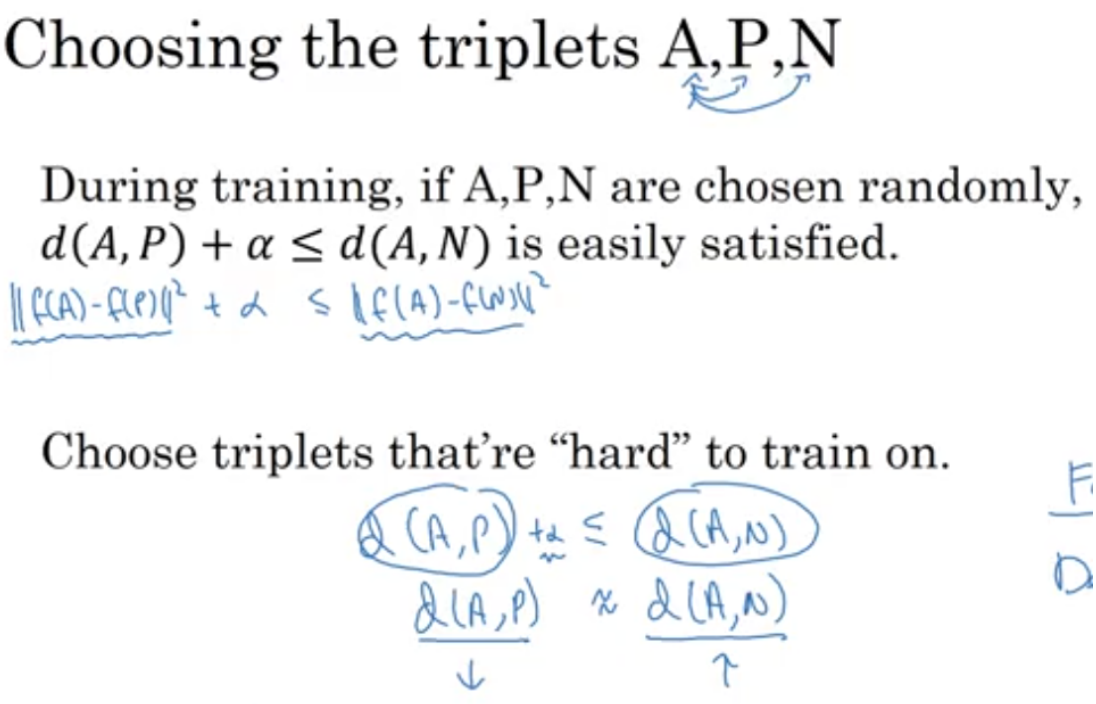
- Loss Function

  - GD can be used to minimize such loss. That would have an effect of backpropagating through parameters to better learn accurate encoding.

  - Training sets shall be large enough; in order to result a good alg.
    For example, the 10k pictures shall be for different 1k persons, meaning 10 pictures on average for each person.
    After training, you can use one-shot learning to verify images, but again with few pictures it won't work.



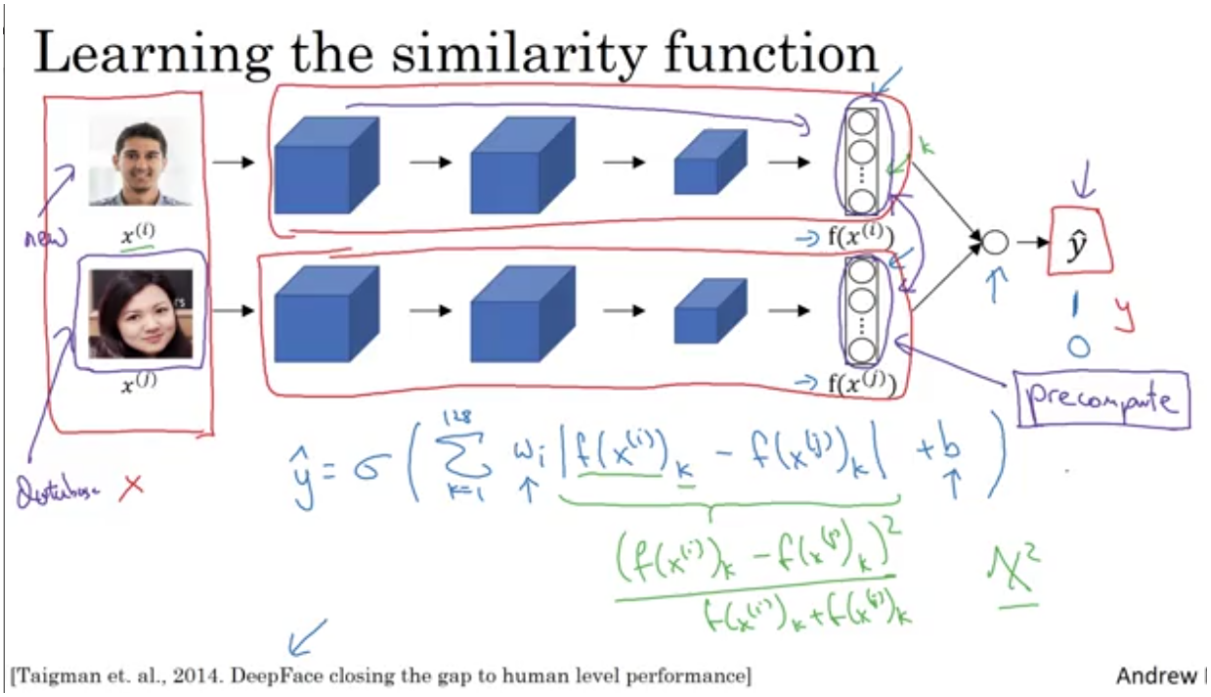
- To choose the triplets A,P, N:
    - choosing triplets that are <u>hard to train</u>.



- Binary Classification

    - The Triplet Loss is one good way to learn the parameters for face recognition. Another way is to apply 2 Siamese nets (same parameters) and output logistic regression, such that 1 would represent same images and 0 different images.

- One workaround to be more computationally efficient is to first, pre-compute the encoding for the available database images. Then, apply the Siamese net on the new set that you want to examine. Finally compare the encoding for both; to compute the final prediction.
  This is helpful for couple of reasons; one you don't have to save all the database, that would be too large. Also computationally more efficient; in order to avoid running the Siamese net on all the database.



So the formula should be:

$$y_{hat} = \sigma\left(\sum_{k=1}^{128} w_k \left|f(x^{(i)})_k - f(x^{(j)})_k\right| + b\right)$$

- Neural Style Transfer (NST)

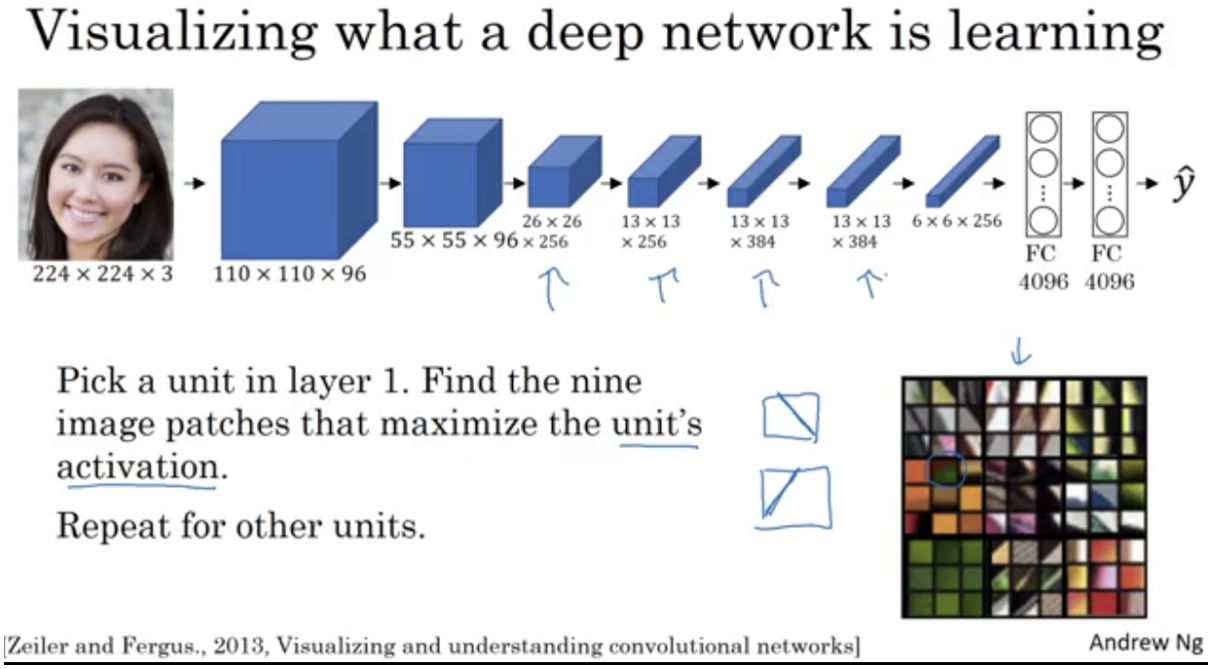
  - It is an optimization technique that takes 2 images – a Content image and a Style reference image- and blend them together so that the output image looks like the content image, but "painted" in the style of the style reference image.

- NST employs a pretrained CNN to transfer styles from a given image to another. This is done by defining a loss fn that tries to minimise the differences between C, S & G.
- Most of the alg you've studied optimize a cost function to get a set of parameter values. In NST, you'll optimize a cost function to get pixel values!

- Intuition behind deep CNN

    - If you pick one hidden unit in the early layers of an AlexNet-like network as shown below, and tries to maximize its activation, it turns out that the info contained in such unit is not very representative.

    In the screenshot below, it obvious that the trained hidden unit of early layers are more interested in simple features like detecting edges or a particle shade of colour.

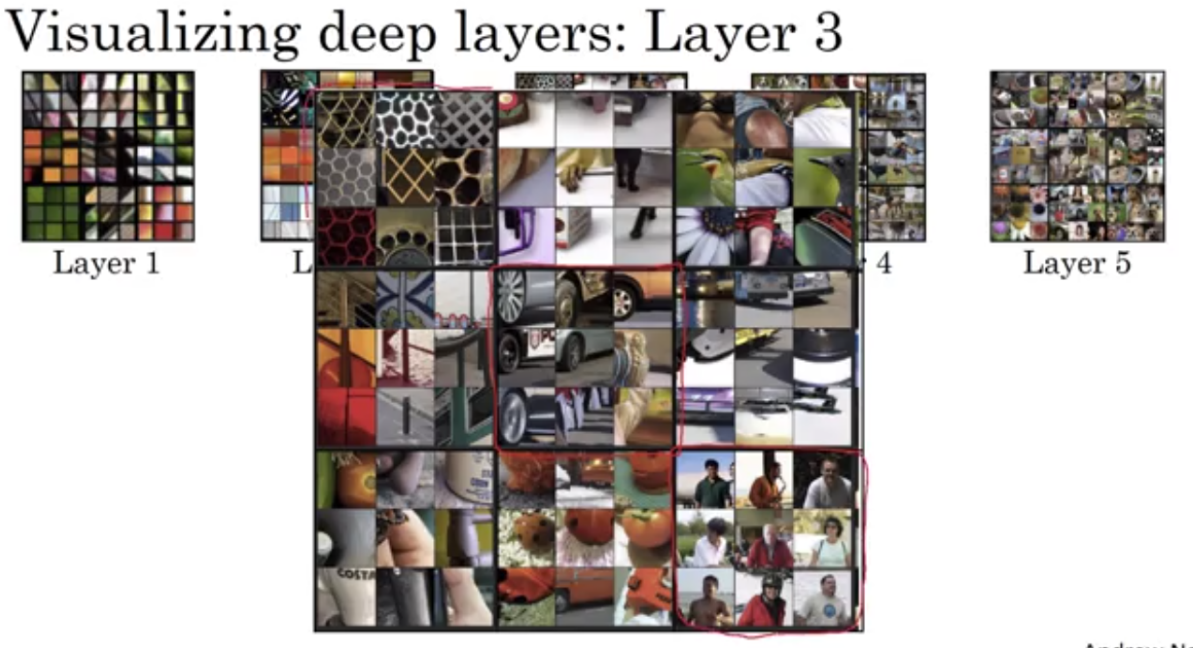


    - On the other hand, in the deeper layers, the hidden units will see a larger region of the image. On the extreme end, each pixel could hypothetically affect the output of the CNN.

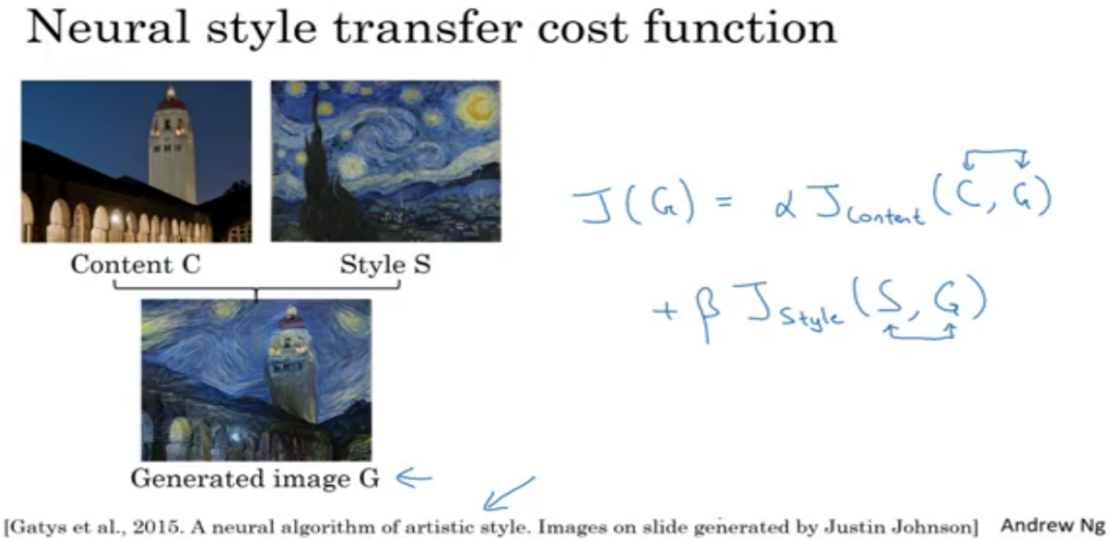
- In the visualization shown below, for example in layer 2, this is what maximally activates 9 different hidden units in layer 2.

On the upper left most cell grid in layer 2 -for example, represents 9 patches that cause one hidden unit to be highly-activated. With different grid cell, this is a different set of 9 image patches that again cause one hidden unit to be highly activated.

- The important takeout from this, is that when visualizing a NN, layer by layer -for example, one grid cell shall reflect how one hidden unit in that layer is being activated.



- Cost Function of NST



- Generated Image (G)

Find the generated image G

1. Initiate G randomly

   G: 100 × 100 × 3 (RGB)

2. Use gradient descent to minimize $J(G)$

$$G := G - \frac{\partial}{\partial G} J(G)$$

❖ Content Cost Fn

- Usually when choosing the layer **L** upon which content cost shall be computed, **L** shall be chosen to be not too deep and too shallow. The too shallow layers are focused on individual pixel values (low-level features), while deeper layers recognizes high level features.
- Again, the content cost fn ensures that the activations of higher layers are similar between the C & G.

- The intuition is that if 2 images have the same content, they should have similar activations in deeper layers.

- The process is shown below

Content cost function

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

$$J_{content}(C,G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

❖ Style Cost Fn

- Using activations of mid-layer **L** -for example, we are trying to measure style, which is basically the correlation between activations across different channels.
- Meaning: we are going to select one unit in channel 1 and compare it with the same unit in channel 2.
  N.B. we are currently working within the same layer!

- It's defined as the difference of correlation present between the feature maps computed by S & G.

- Style Matrix is gram matrix (dot product of entries), where (i,j)th element is computed by computing the element wise multiplication of the ith and jth feature maps and summing across both width and height.

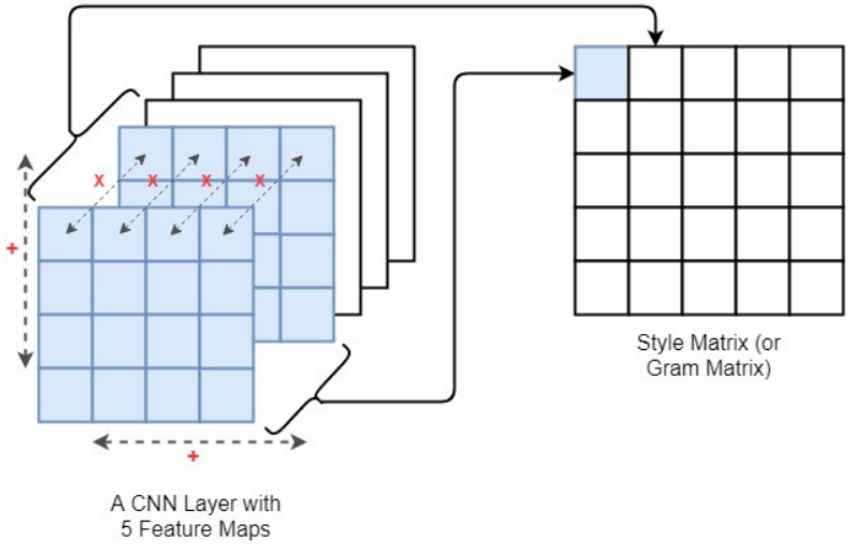
- An important application is to compute linear independence: a set of vectors are linearly independent if and only if the Gram determinant is non-zero.

    - **$G_{(gram)\ i,j}$: correlation**
        i. its dimension is (n_c , n_c) where n_c is # of filters (channels). $G_{(gram)\ i,j}$ measures how similar the activations of filter **i** are to the activation of filter **j**.

    - **$G_{(gram)\ i,i}$: prevalence of patterns or textures**
        i. diagonal elements show how active a filter **i** is.

ii. for example, suppose filter **i** is detecting vertical textures in the image. Then $\mathbf{G}_{(gram)\ i,i}$ measures how common vertical textures are in the image as whole. If it is large, then the image has a lot of vertical texture.

By capturing the prevalence of different types of features ($\mathbf{G}_{(gram)\ i,i}$) , as well as how much different features occur together ($\mathbf{G}_{(gram)\ i,j}$), the Style matrix $\mathbf{G}_{(gram)}$ measures the style of an image.



N.B. by correlation here we mean the unnormalized cross-covariance.

- Style matrix is calculated for both S & G.
- More visually-pleasing results can be obtained by using the style cost fn from multiple different layers.

- **N.B.** In the deeper layers of a ConvNet, each channel corresponds to a different feature detector. The style matrix $\mathbf{G}^{[l]}$ measures the degree to which the activations of different feature detectors in layer **L** vary (or correlate) together with each other.

- the cost is computed using the hidden layer activations for a particular hidden layer $\mathbf{a}^{[l]}$.

N.B. Unlike the content representation, where using just a single layer is sufficient, for style representations, usually using multiple layers gives better results.

- Remember
    - NST uses representations (hidden layers activations) based on a pretrained ConvNet.

- The content cost fn is computed using one hidden layer's activations.
- The style cost fn for one layer is computed using the Gram matrix of that layer's activations.
- The overall cost fn is calculated using several hidden layers.
- Optimizing the total cost fn results in synthesizing new images.

Style Cost Function :

$$J_{style}^{[l]}(S,G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_K \sum_{K'} \left(G_{KK'}^{[l](S)} - G_{KK'}^{[l](G)} \right)^2$$

$$G_{KK'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijK}^{[l](S)} \, a_{ijK'}^{[l](S)}$$

$$G_{KK'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijK}^{[l](G)} \, a_{ijK}^{[l](G)}$$

$a_{ijK}^{[l]}$ = activation at (i,j,K)

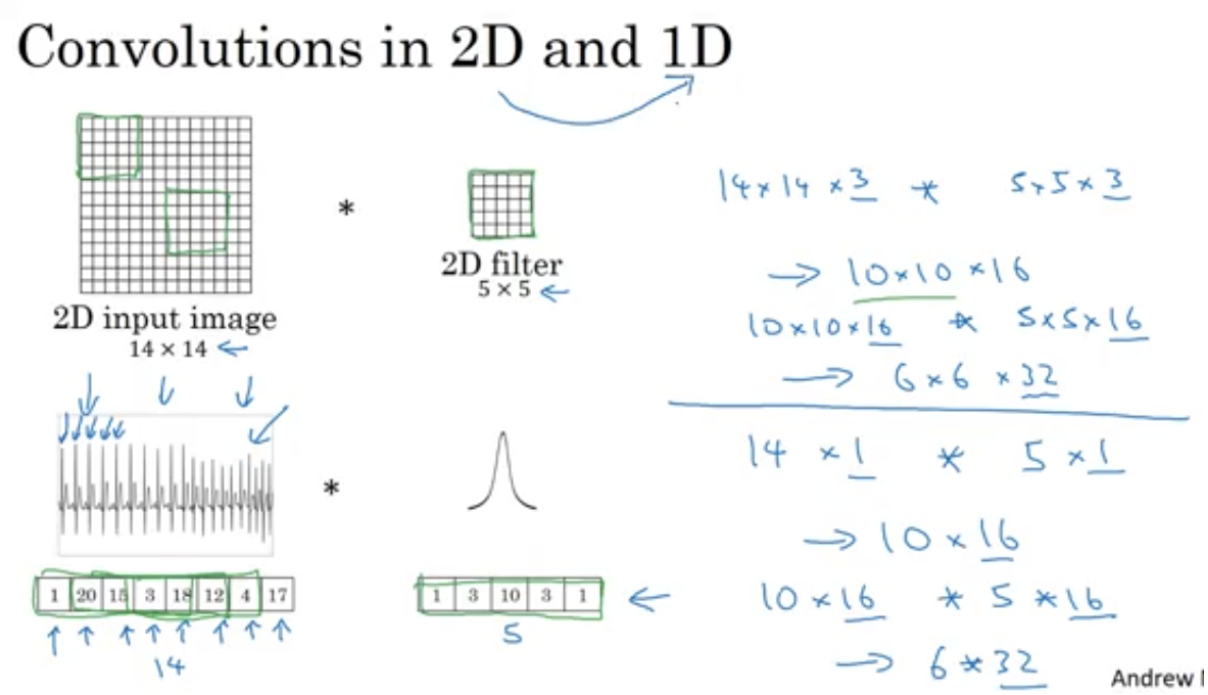
$G^{[l]}$ is $n_C^{[l]} \times n_C^{[l]}$

Overall, using multiple layers

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S,G)$$

- 1D & 2D Generalization

    - Our work so far was with 2D input images, we are aiming now to generalize the concept of Convolution to other dimensions.
    - Usually with 1D data, RNNs are used, yet CNN could be used as well.



    - For 3D, CNN works as well
    - One example for 3D data, is CT Scans. Another example, is movie data; considering time is the 3rd dimension.