Convolutional Neural Networks

Computer Vision

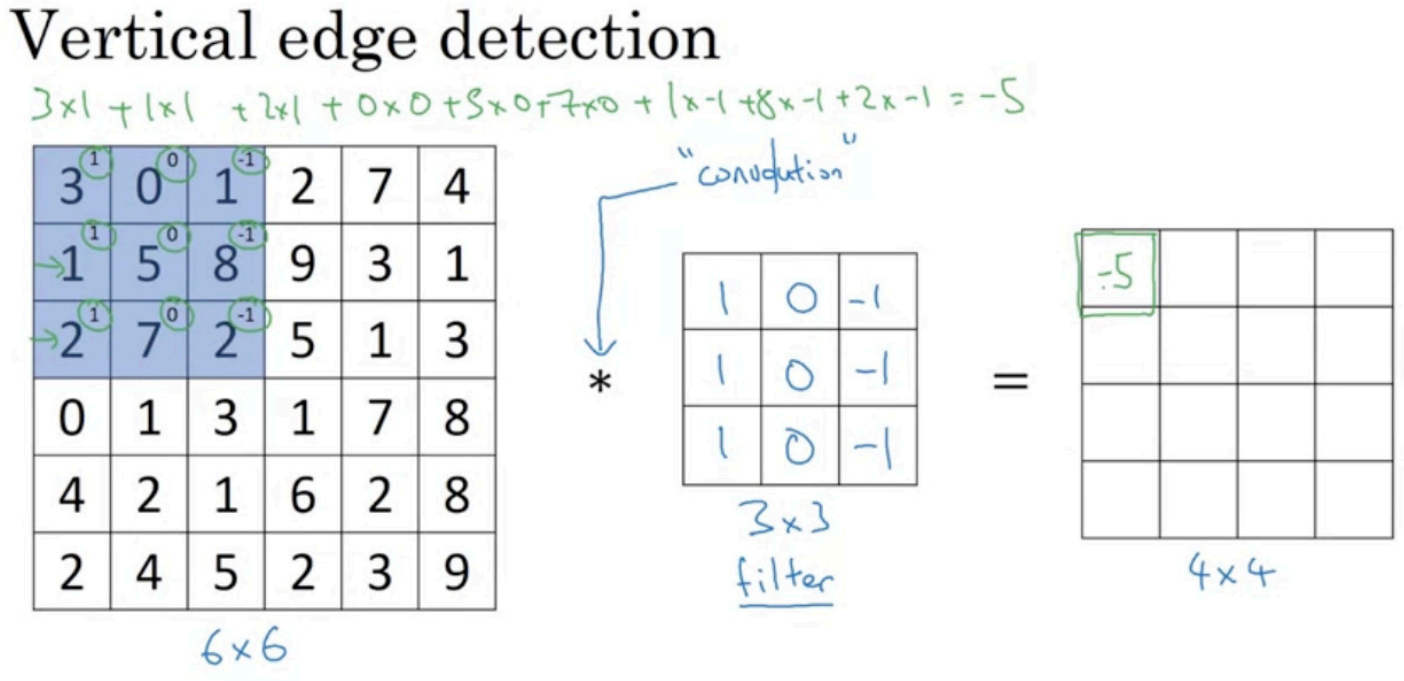
- Image classification
- Object detection
- Neural style transfer -> content image + style image = put them together (art?)

inputs are big: 1000x1000x3(RGB)
let's say first hidden layer has 1k units:
weight W1 will be (1000, 3m) dimensional matrix
: 3 billion parameters

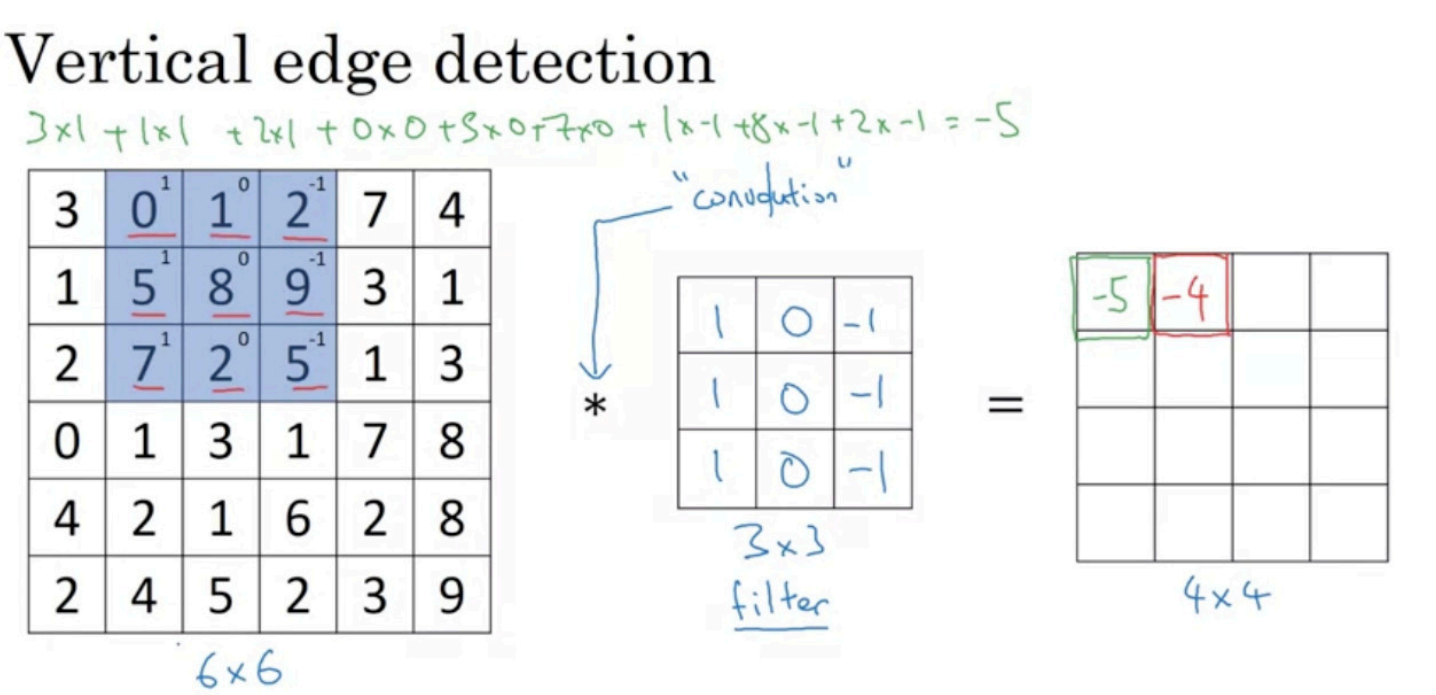
computational/memory requirements

Solution: Convolution

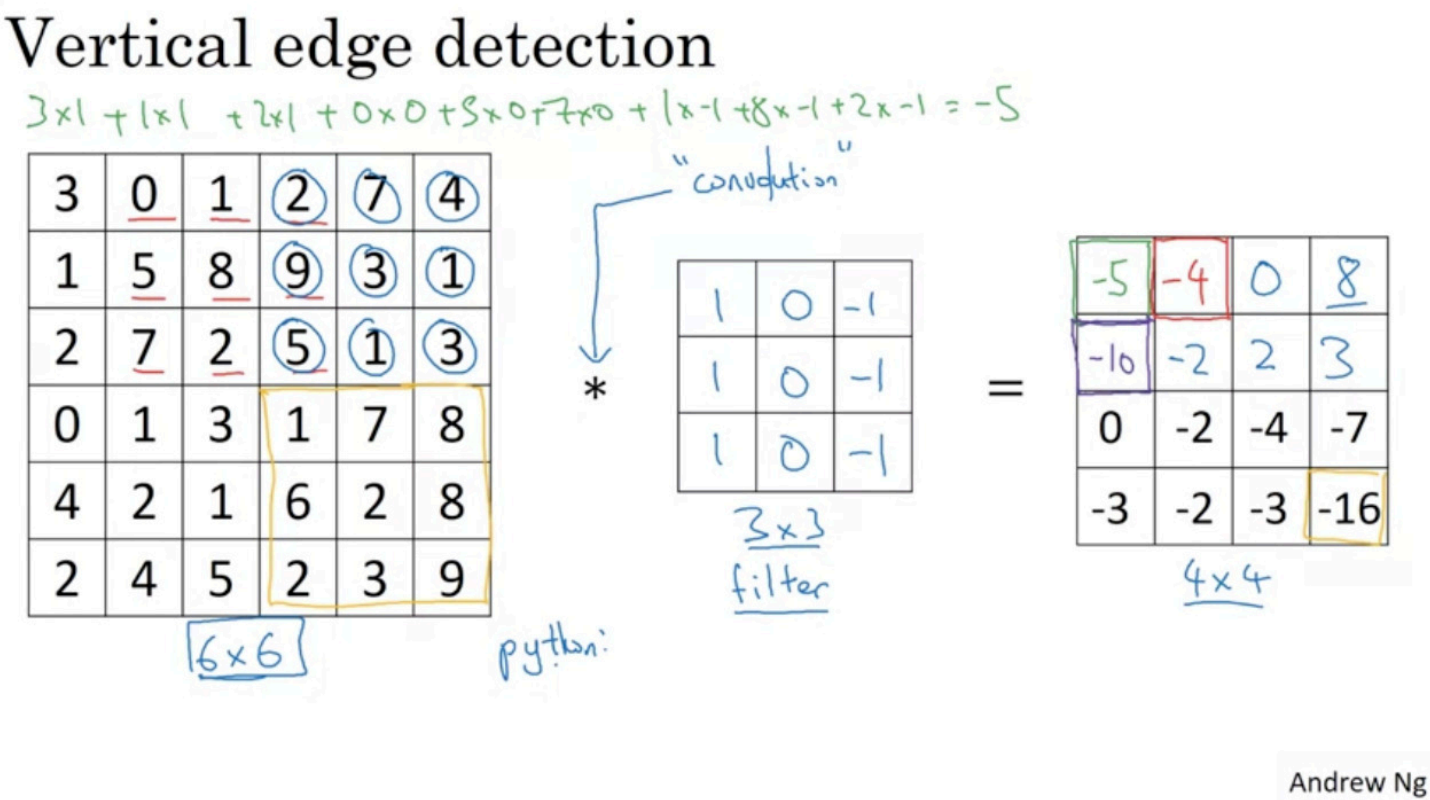
Filter/Kernel



Andrew Ng



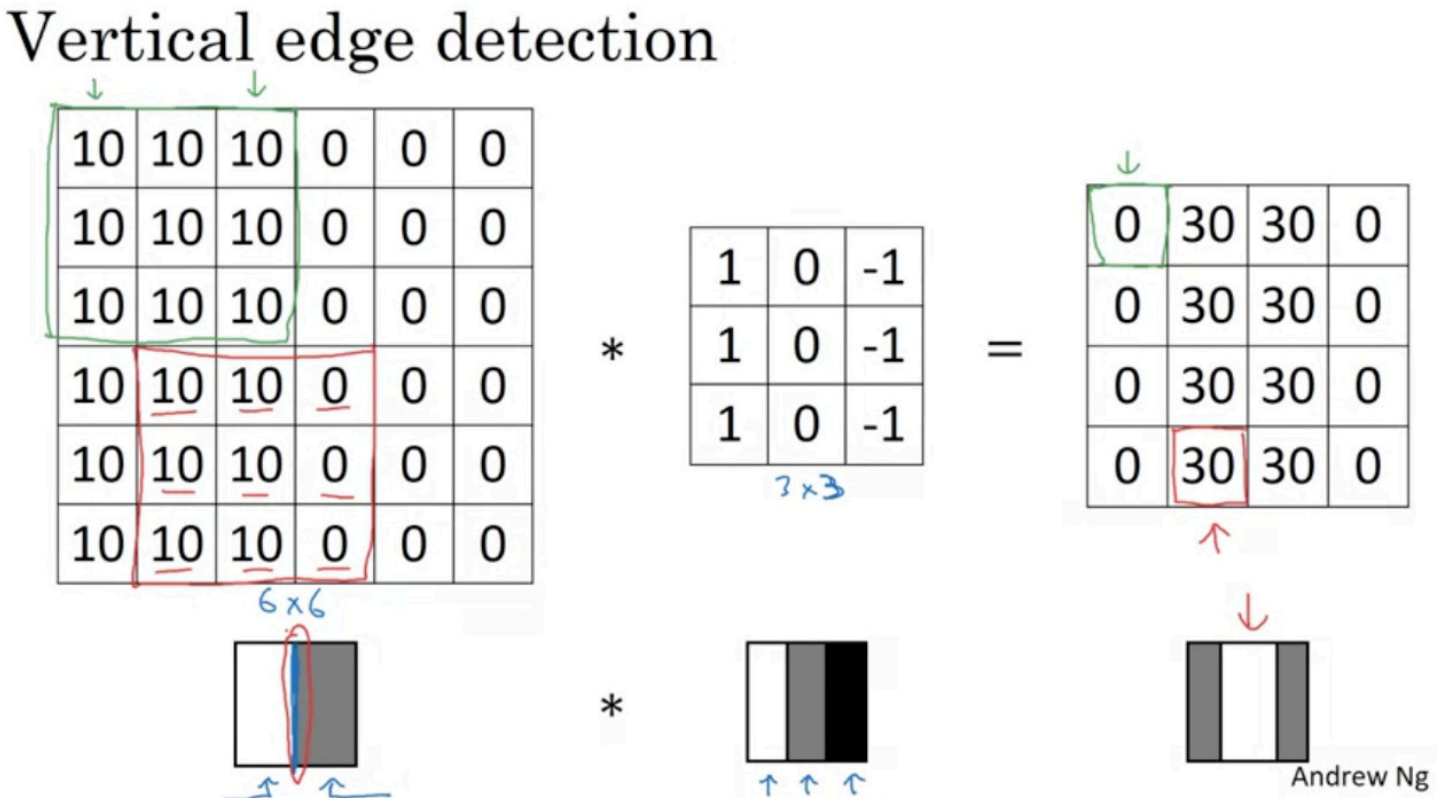
Vertical edge detector:



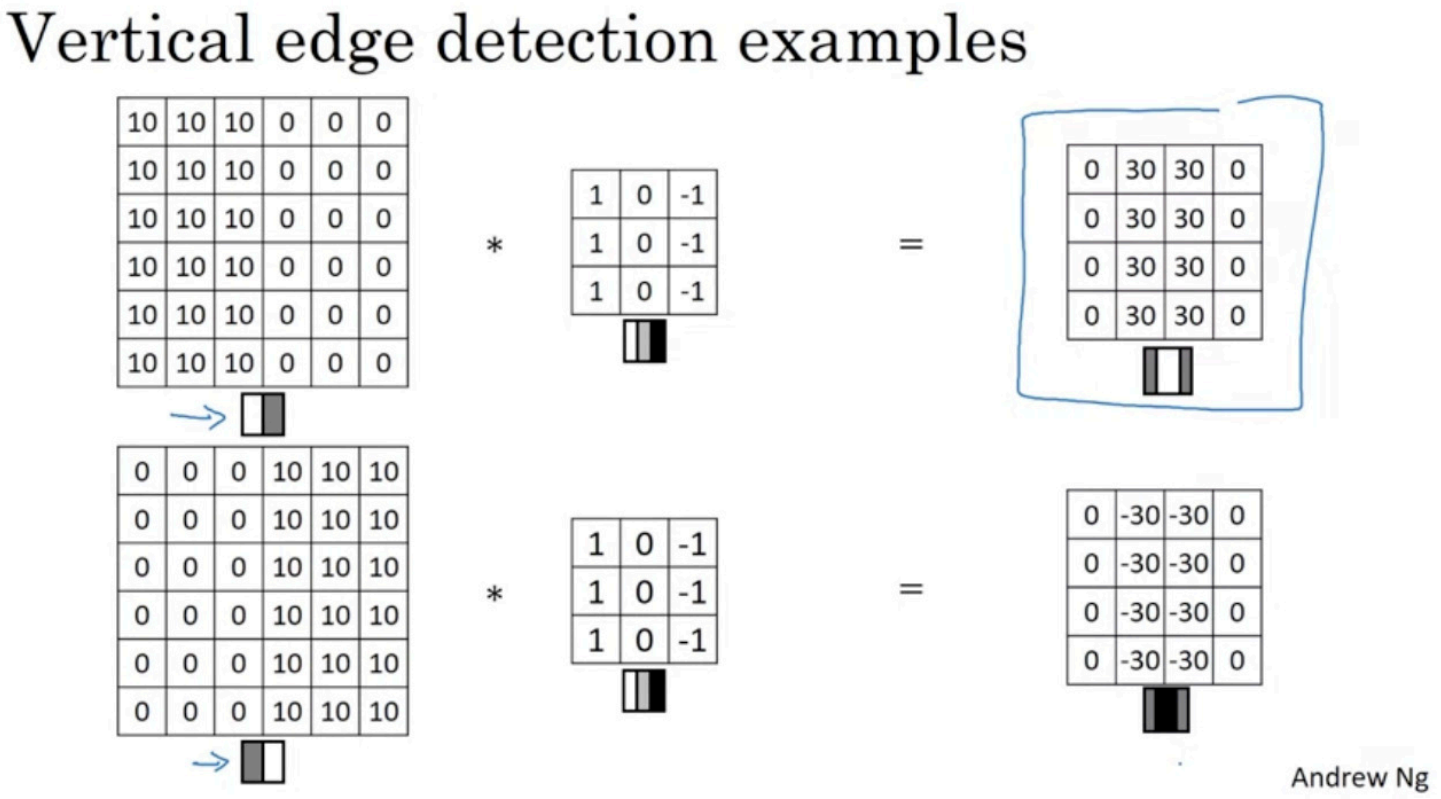
python: conv_forward

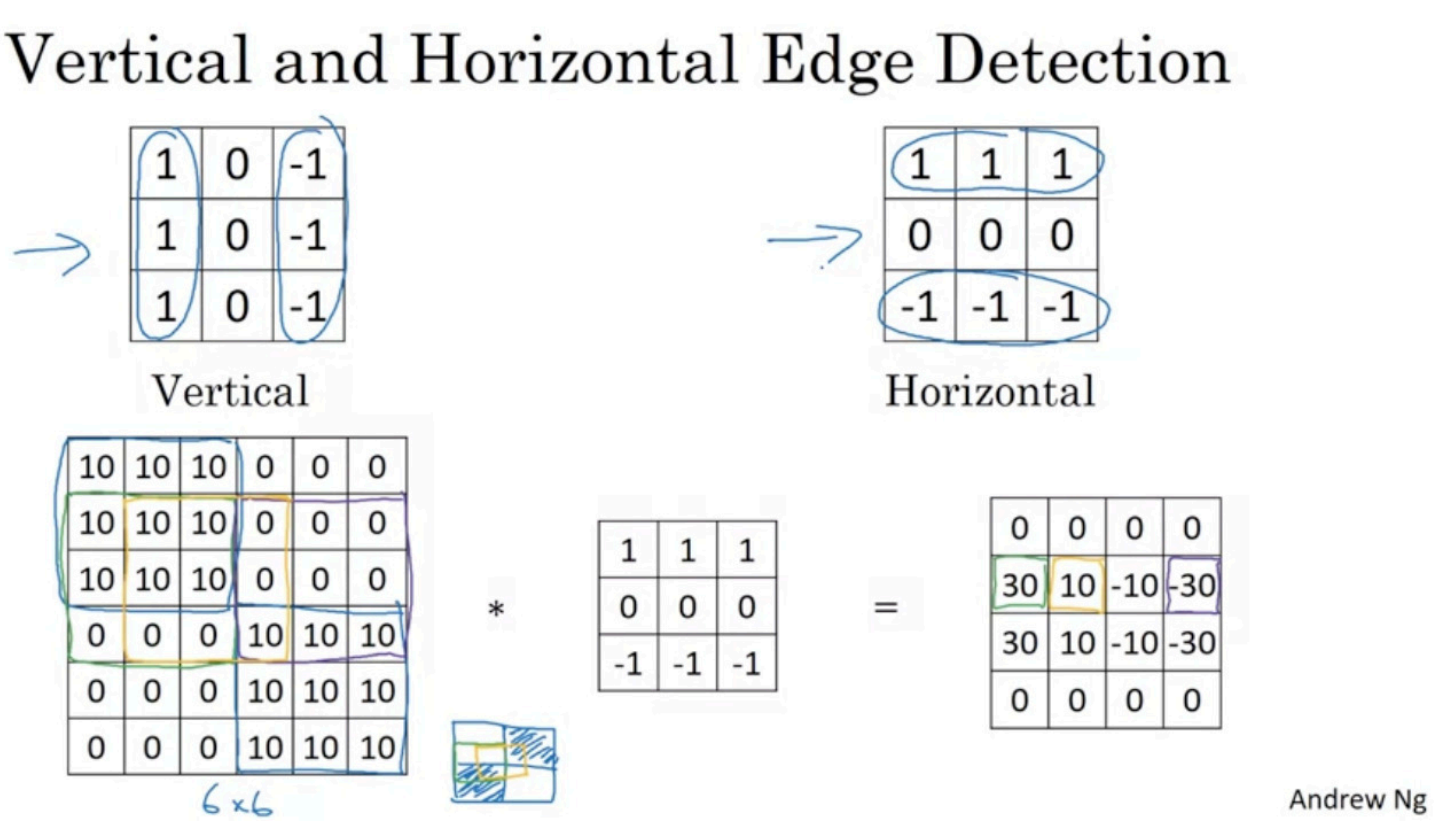
tensorflow: tf.nn.conv2d

...

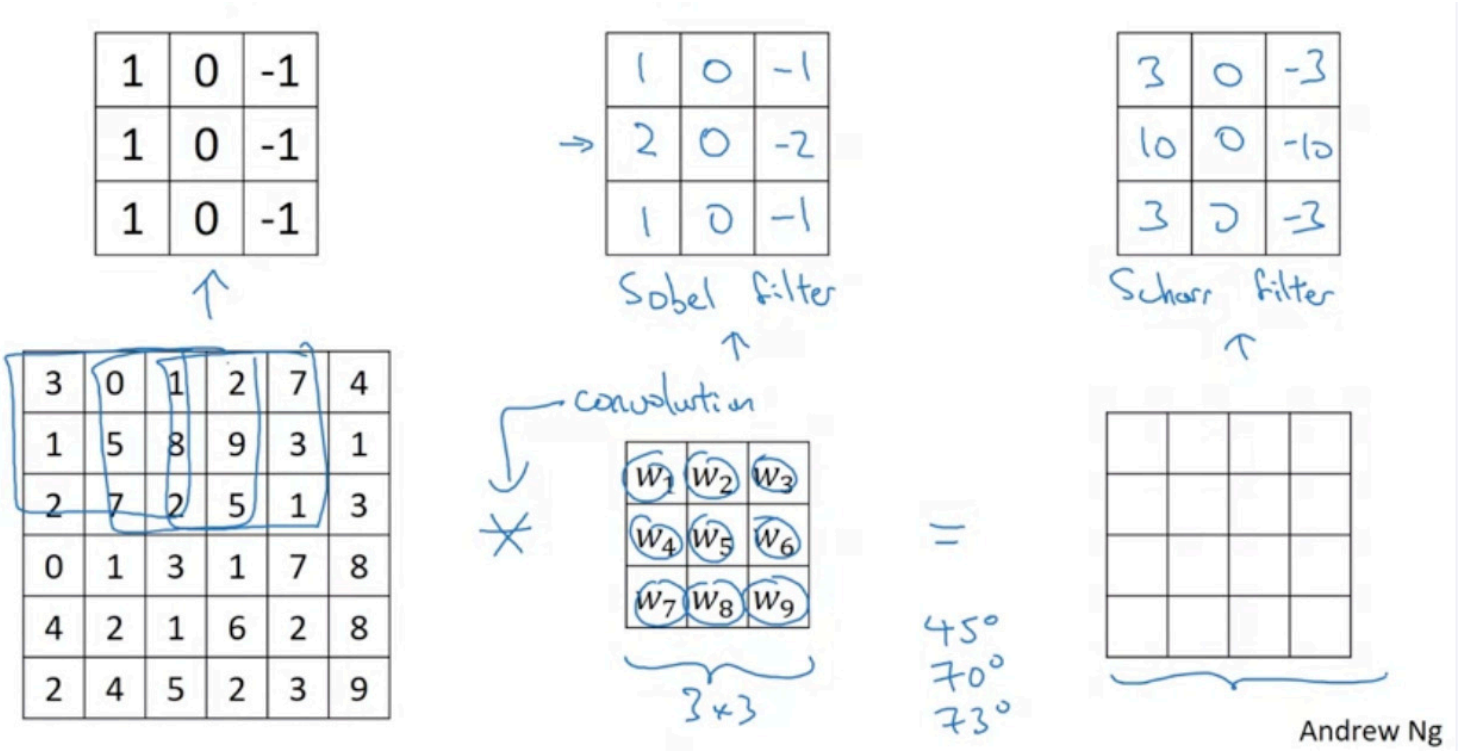


Other edge detection examples:





What is the best set of numbers to use? - for filter.

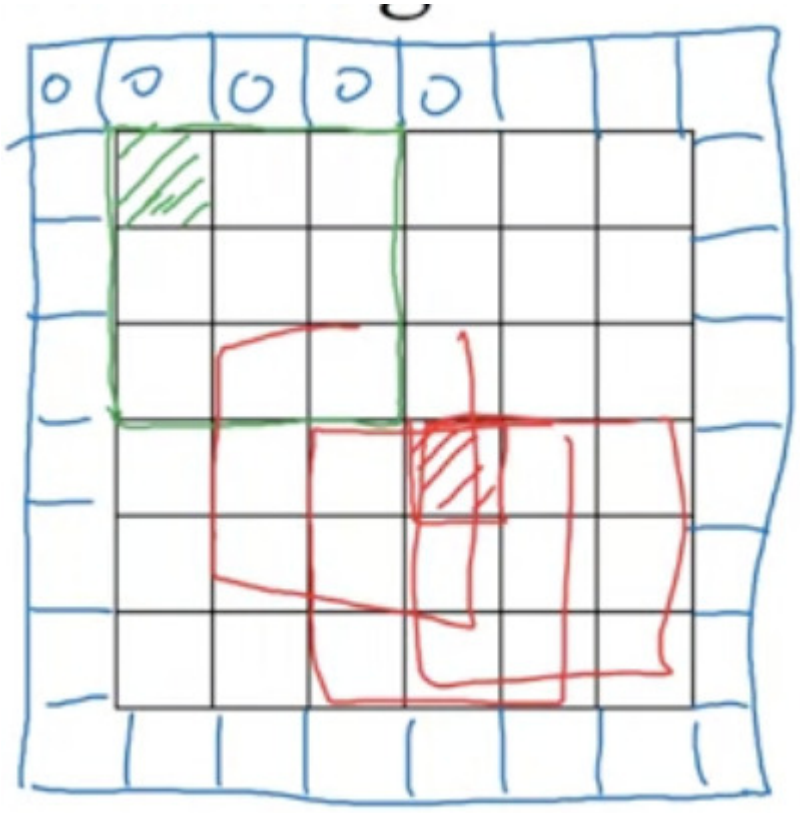


Sometimes you do not need to pick these numbers, but set parameters for each, find using back. propagation

Padding

When applying filter, throwing away too much information on the edges of the image.
(a pixel on the middle gets x10 (salladım) more times processed compared to a pixel in the border)

- apply padding to fix this problem.



Valid & Same Convolutions

Valid: nxn -> no padding (shitty naming)
nxn ***fxf -> (n-f+1)*** n-f+1
6x6 *** 3x3 -> (4x4)

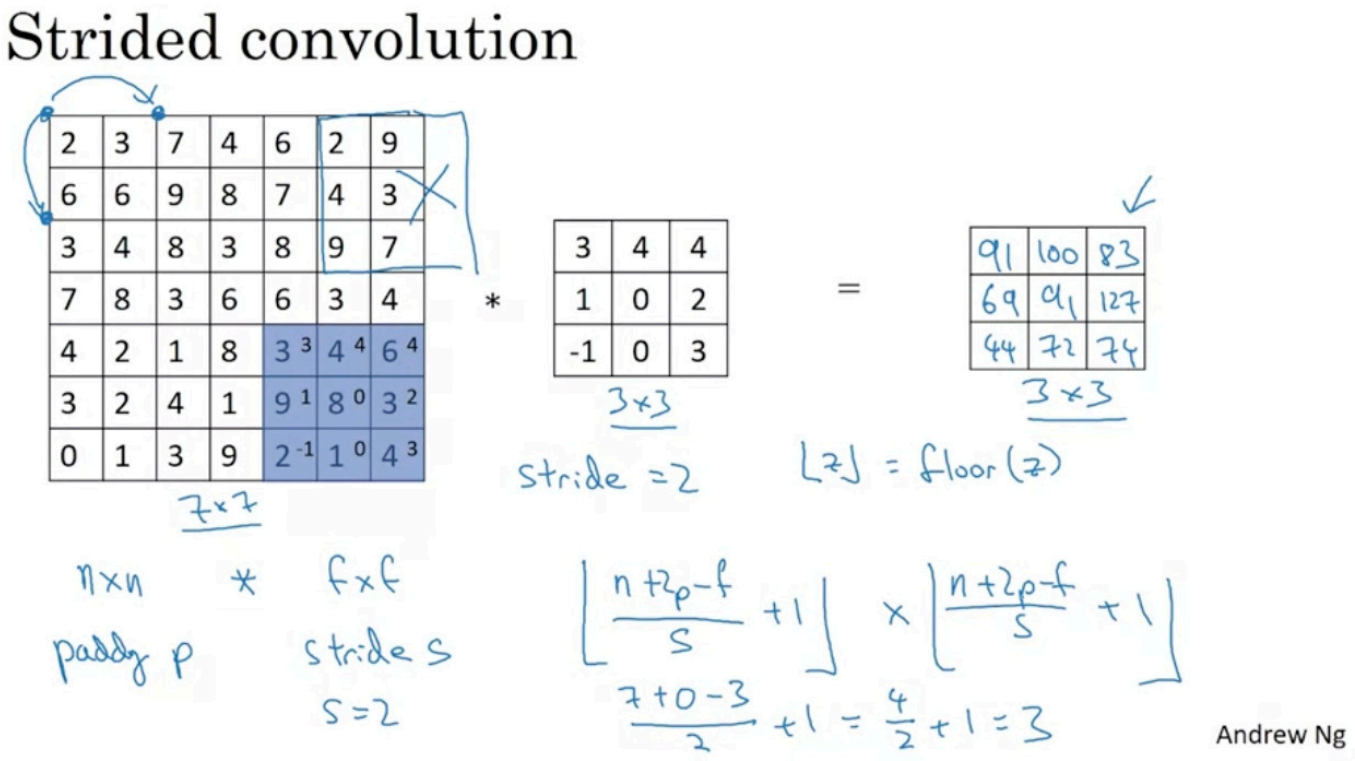
Same: Pad so that output size is the same as the input size.
n+2p-f+1 *** n+2p-f+1
n+2p-f+1 = n => p = (f-1) / 2
3x3 p = (3-1)/2 = 1 | 5x5 f=5, p=2

f is usually odd
when odd: has central pixel. *sometimes nice to have?*

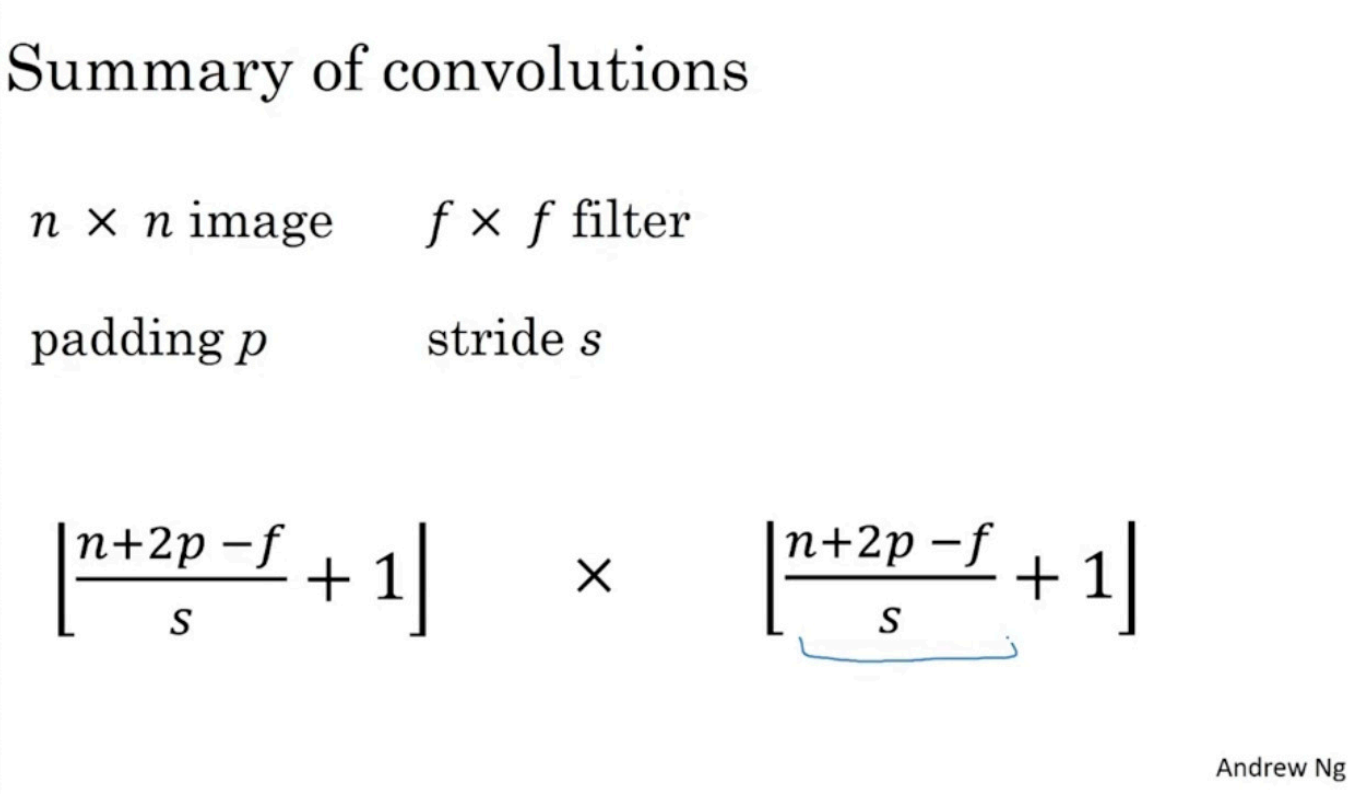
Strided Convolutions

another basic building block of CNNs.

- Same thing, but when multiplying image with filter, there is a step which is different from 1.



formulas for output size:

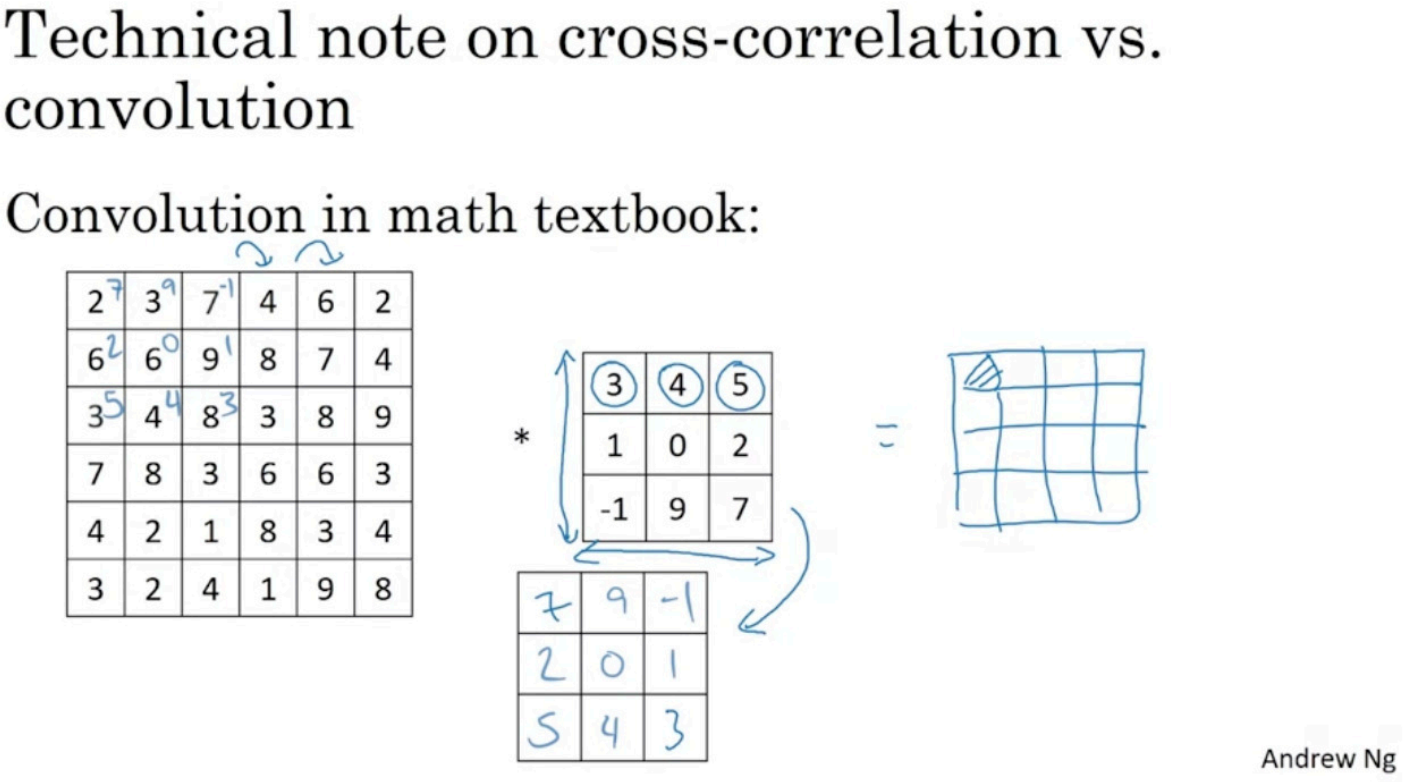


Technical note on cross-correlation vs. convolution

we skipped the 'mirroring operation', that is the difference between DL convolution and real - mathematical convolution:

The non-mirrored operations real name is cross-correlation, but in DL, it is said to be convolution

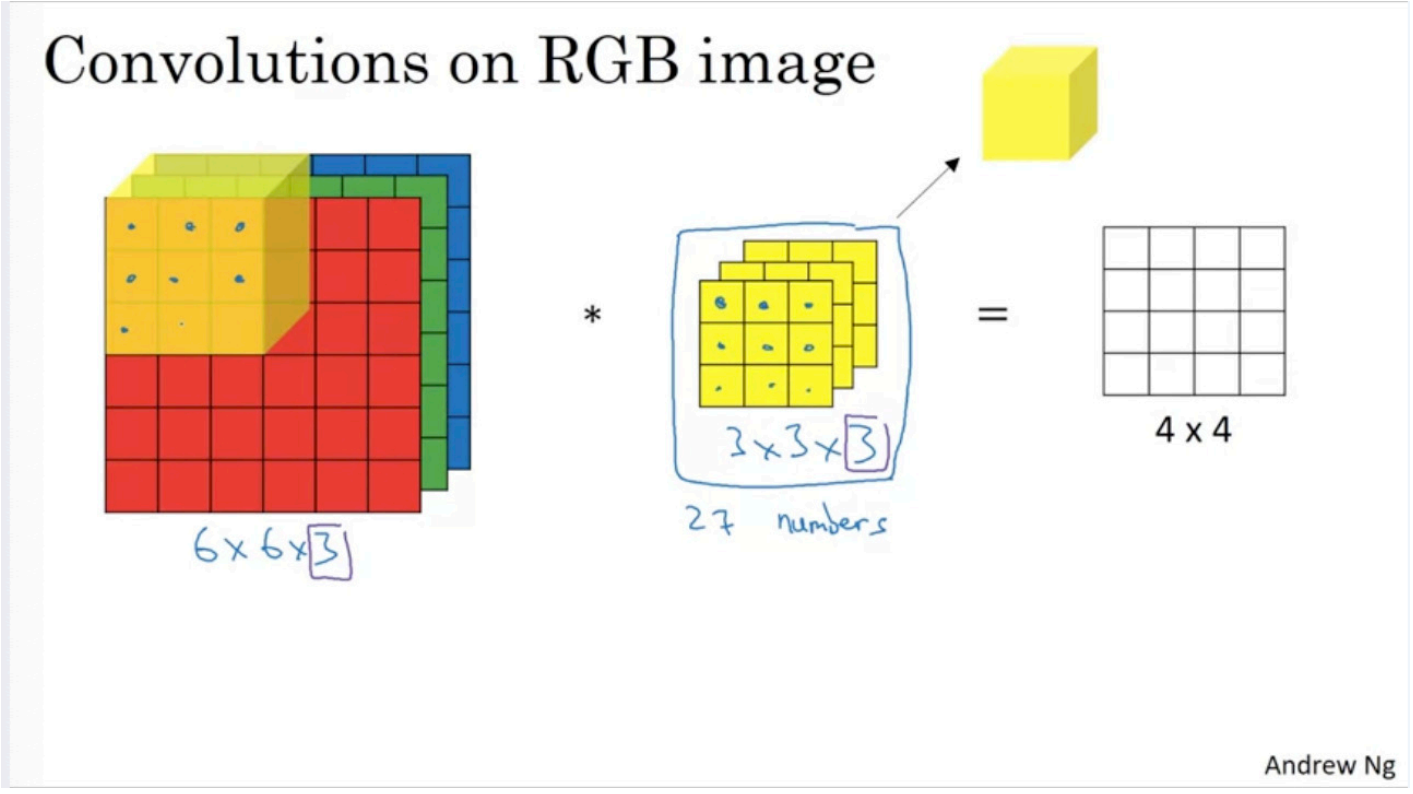
anyway.



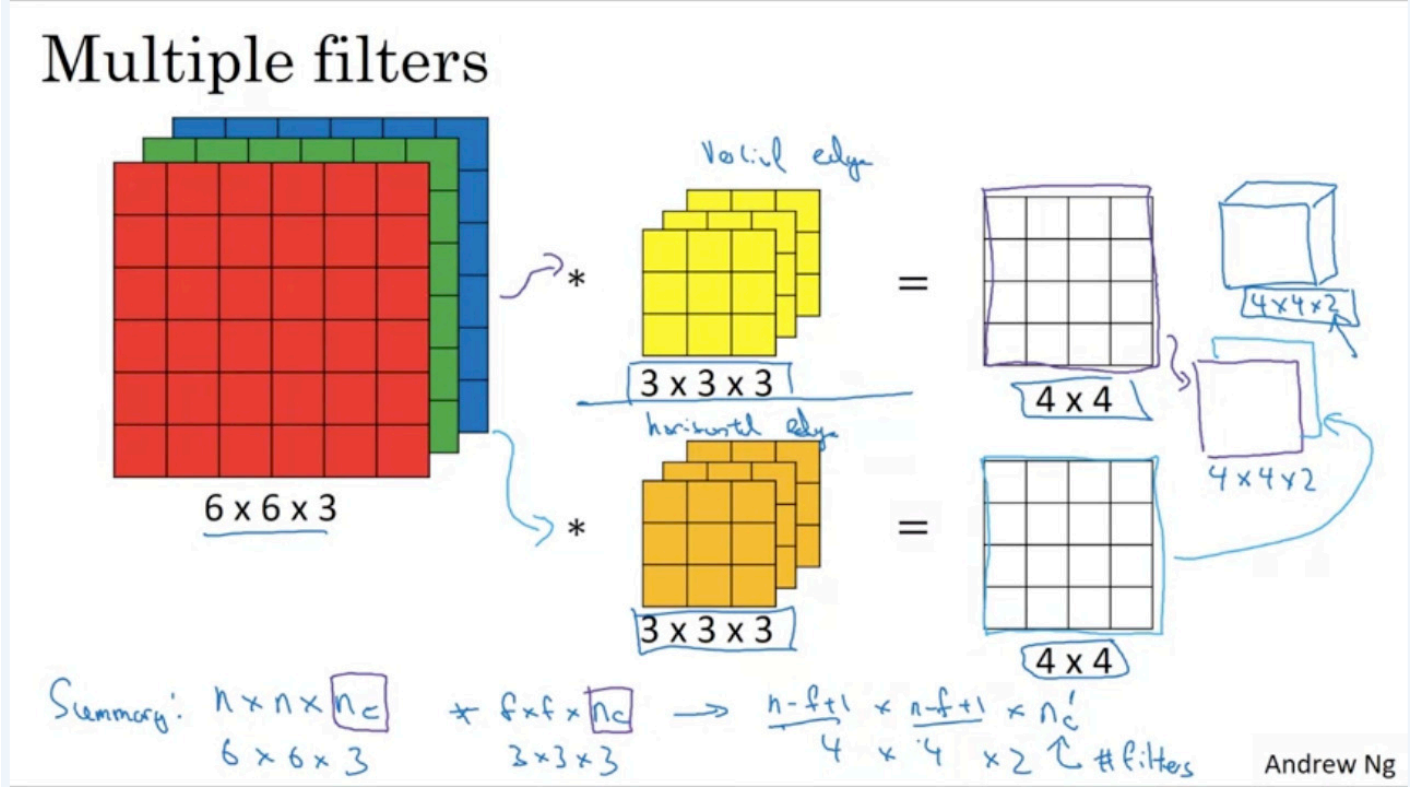
Convolutions Over Volume

Convolutions on RGB images

- 6 x 6 x 3 (image) (height x width x channels)
- 3 x 3 x 3 (filter) (")
  '#' of channels must match
  = 4 x 4 image output



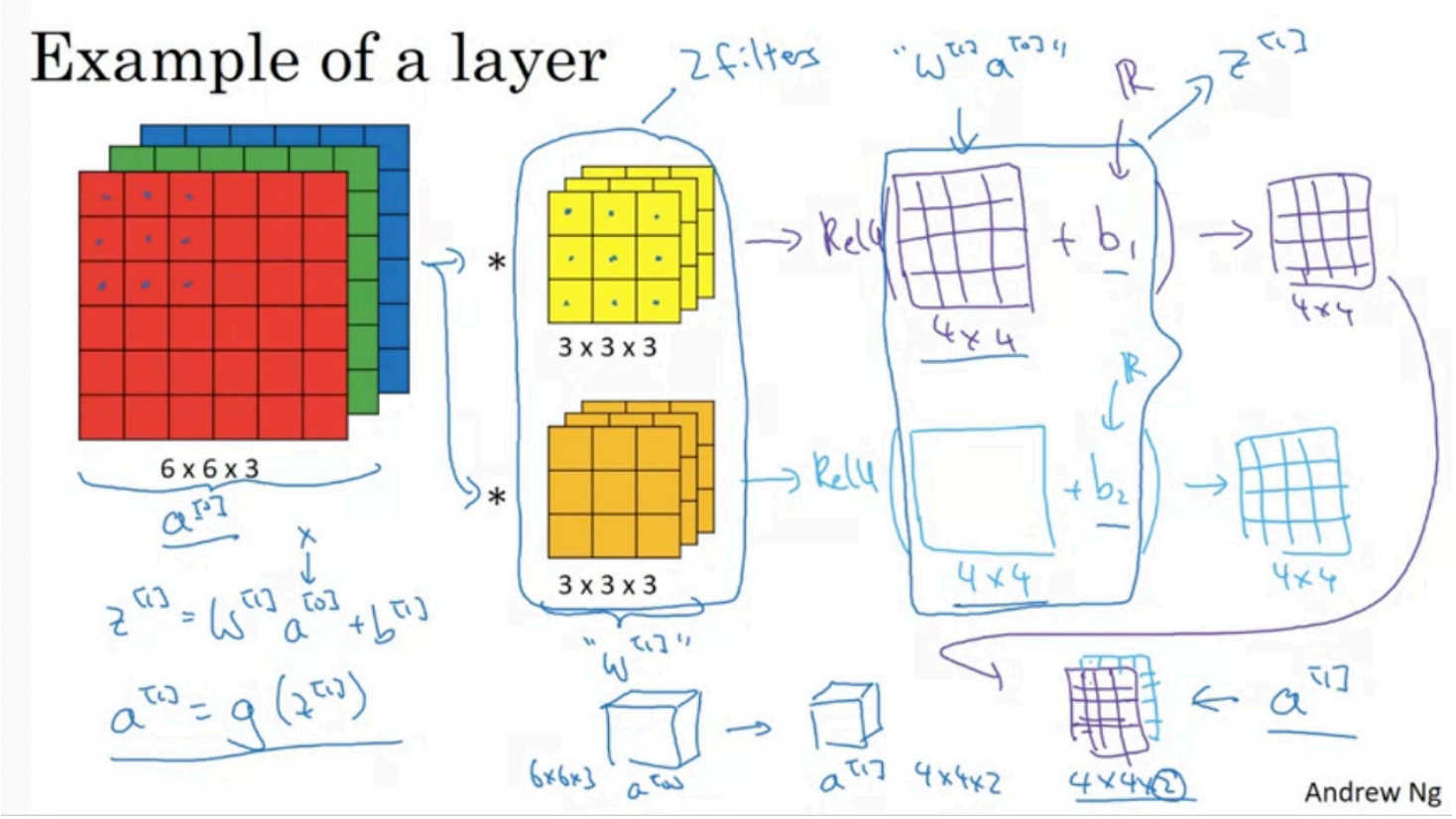
also can apply multiple filters



formulas based on stride:1, no padding assumed

channel - depth : size of 3rd dimension

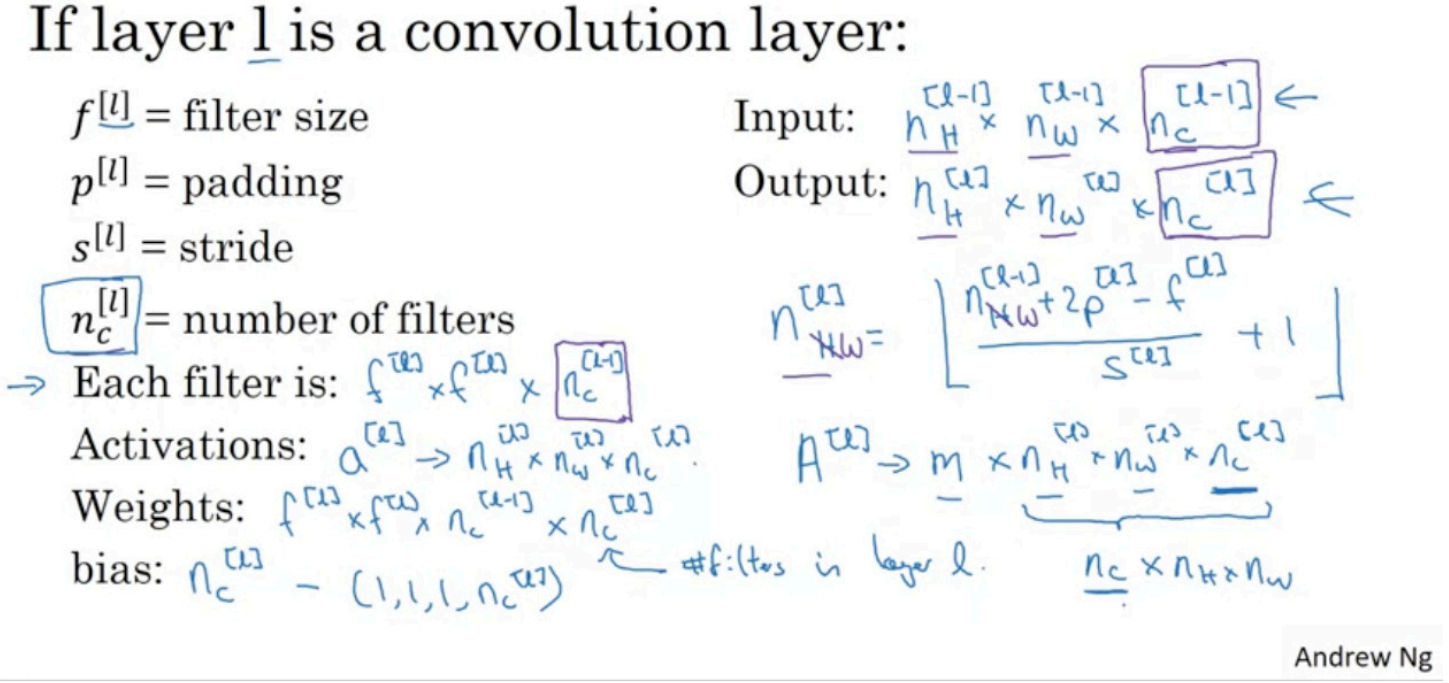
One Layer of a CN



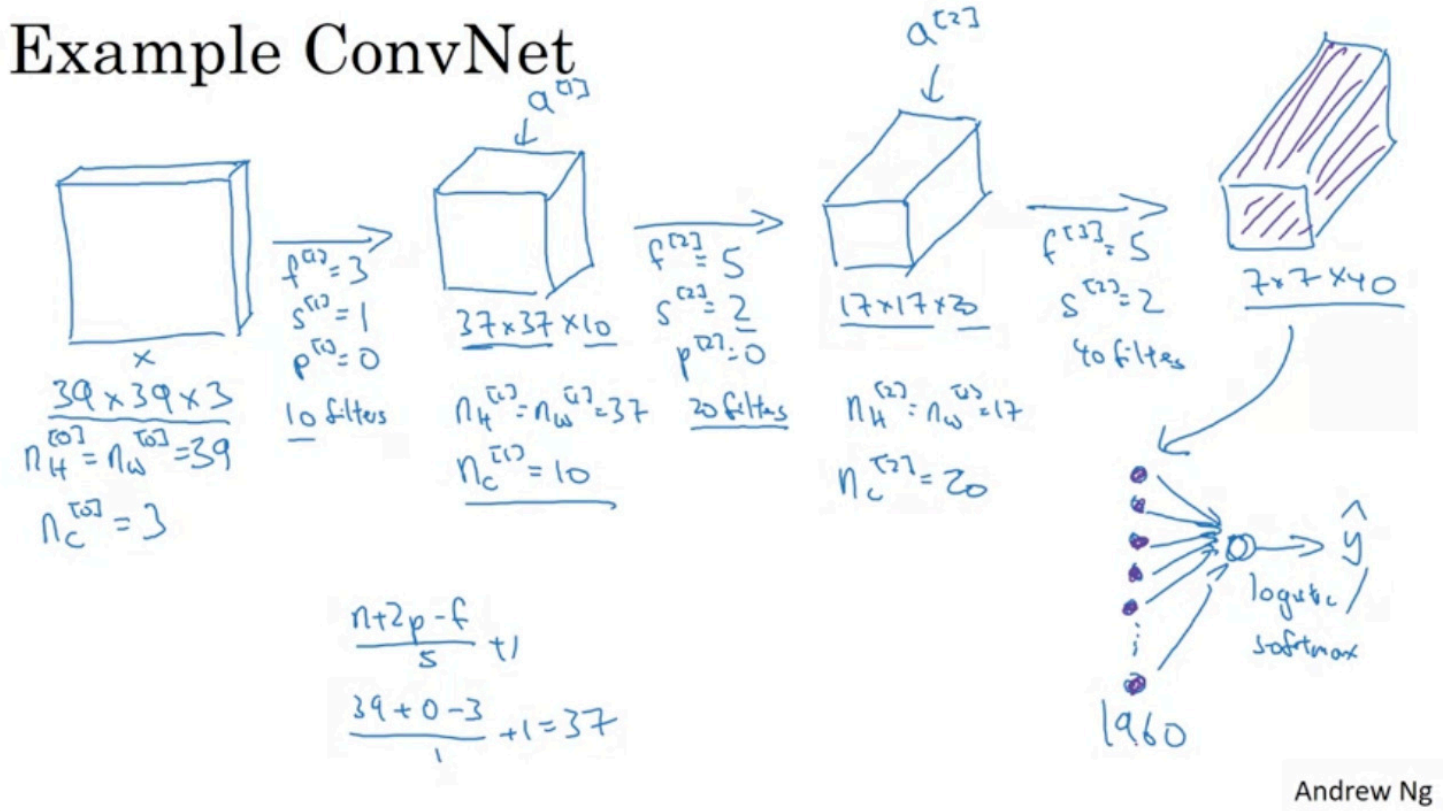
2 filters=2 features, if it was 10, 4x4x10 output volume instead of 4x4x2

If you have 10 3x3x3 filters in one layer, how many parameters?

3x3x3 + 1 (bias) = 28 x 10 = 280 parameters



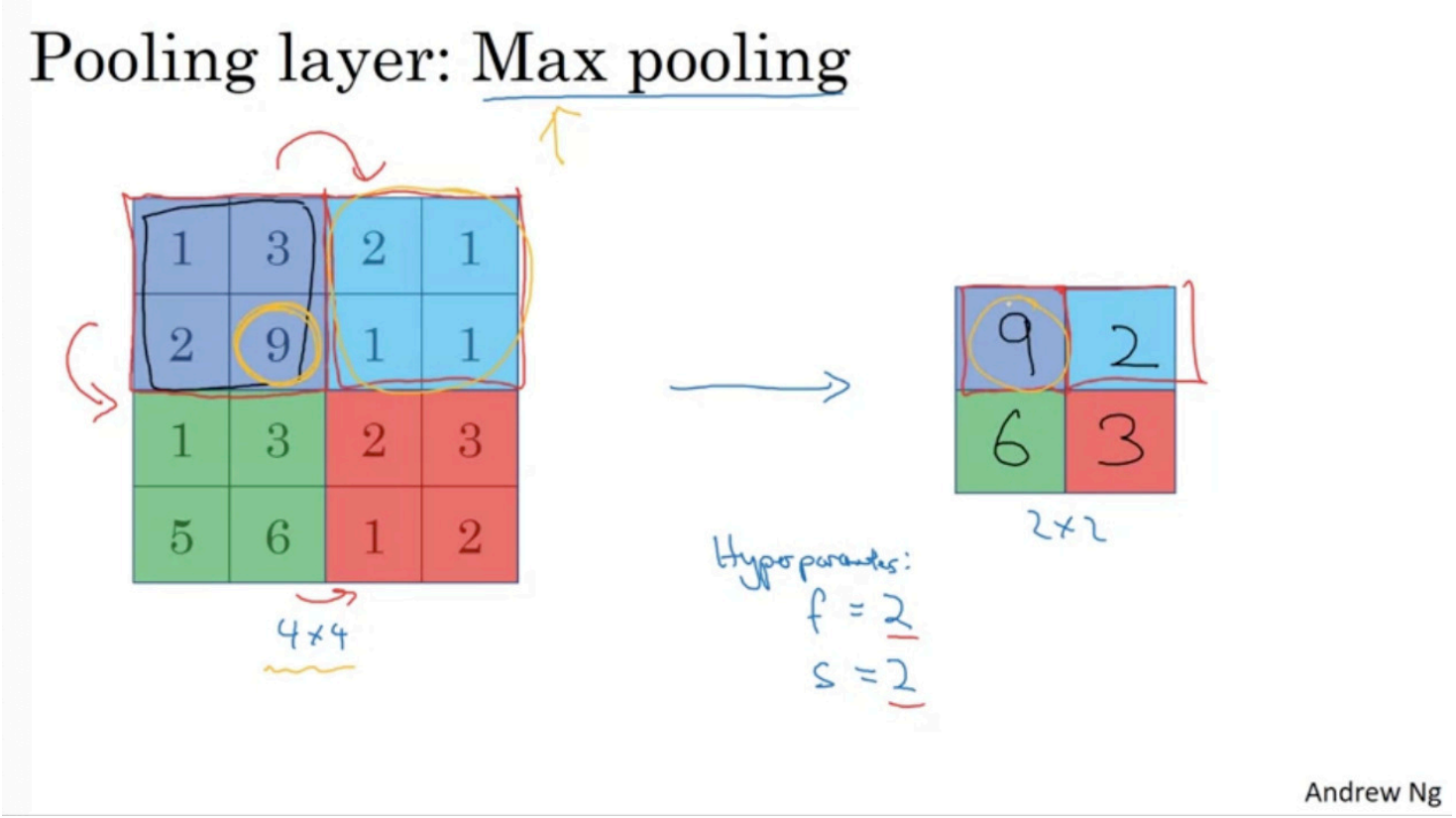
Simple Convolutional Network Example



last step, takes the volume of 7x7x40 and creates a long vector before feeding it to a final logistic regression/softmax unit - which is a common application

Types of layer in a convolutional network:

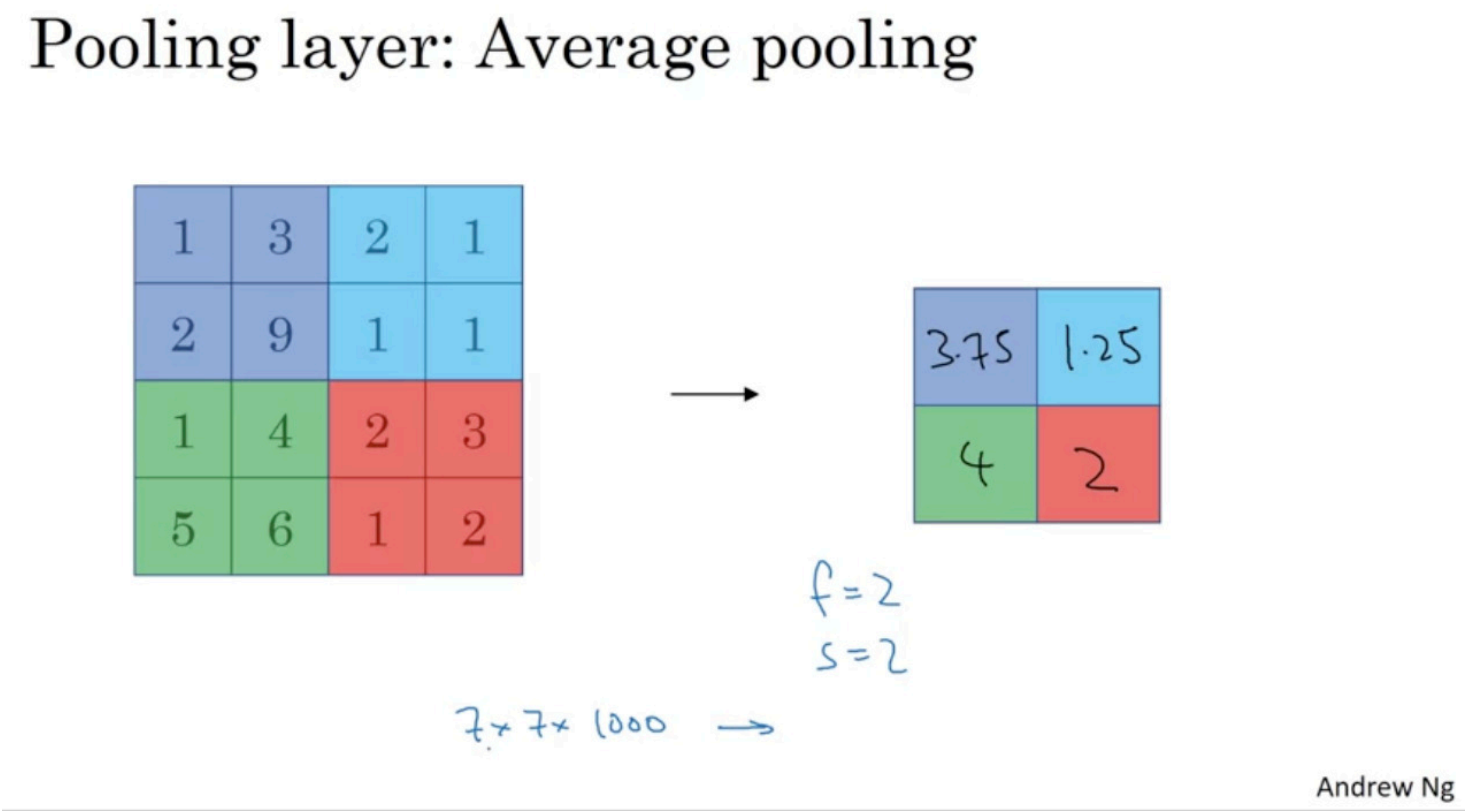
- Convolution CONV
- Pooling POOL
- Fully Connected FC
  other two are simpler



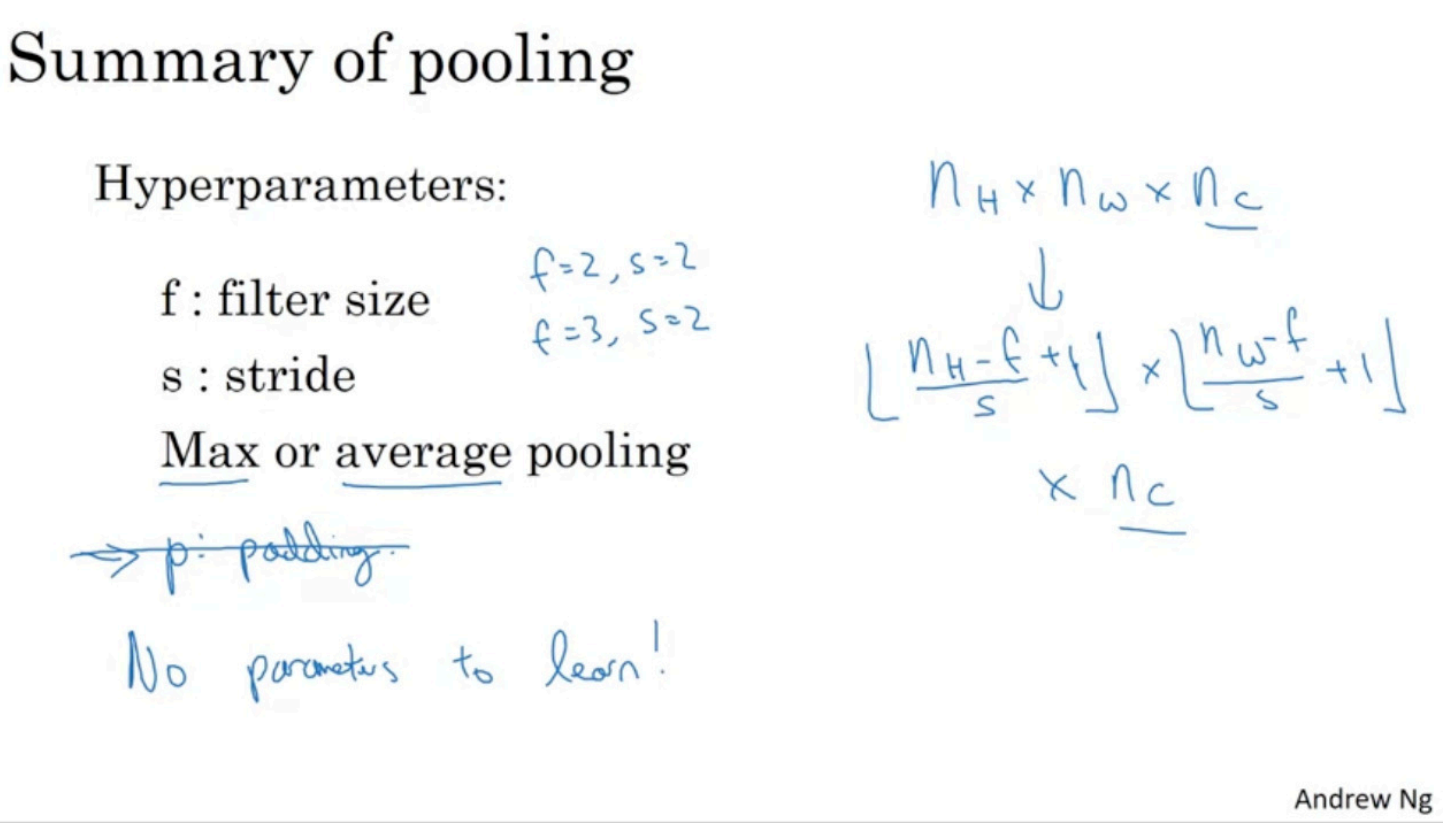
works well, no one really knows why max pool works well. gradient descent doesn't change anything when f and s are fixed.

Max pooling is done independently for each of the channels.

Average Pooling: Not used that frequently



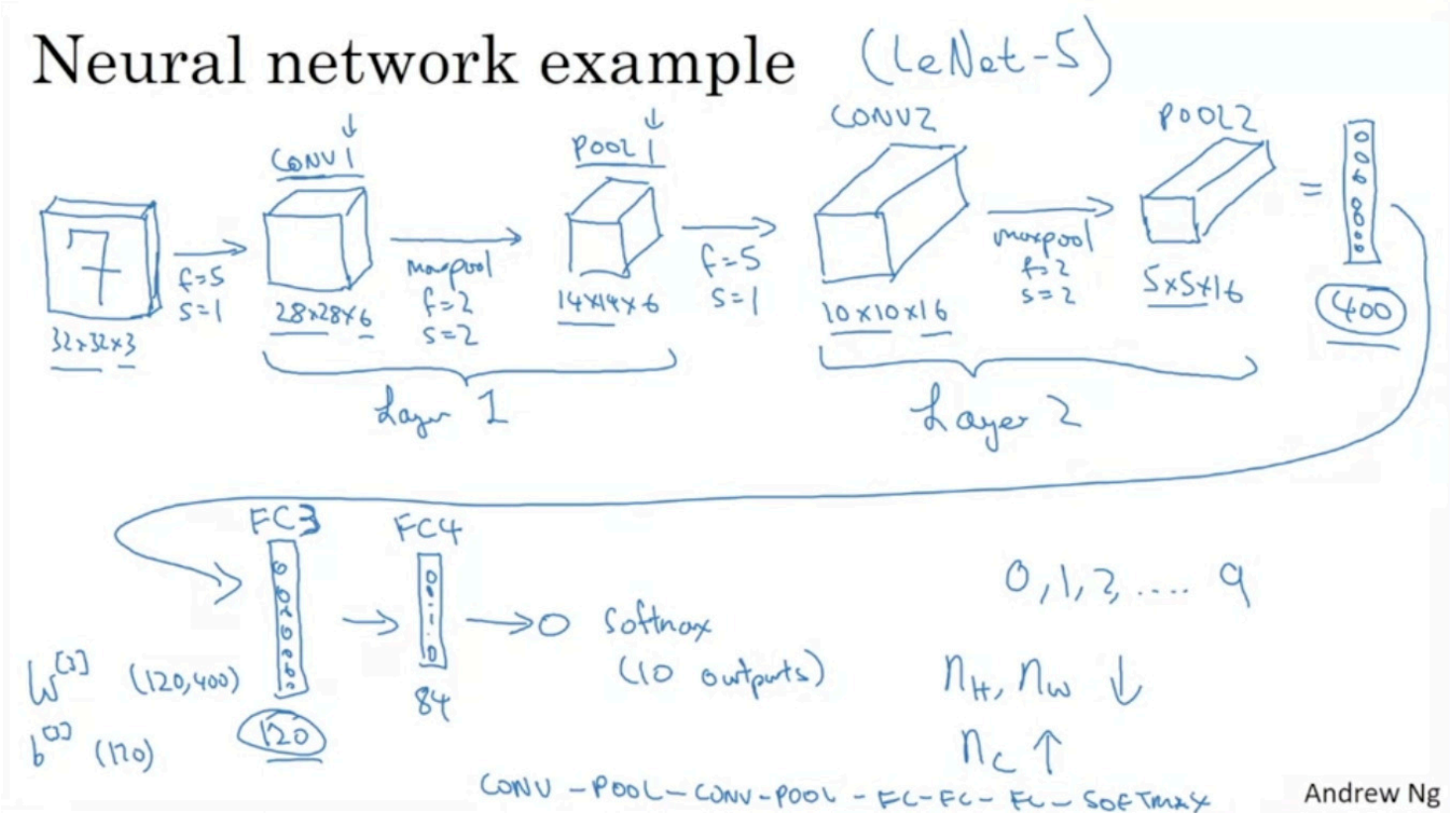
duh



padding: not implemented generally (misses the point)
no parameters to learn, seen in backward propagation

CNN Example

sometimes conv layer and pool layer are considered as separate layers, but general approach is to name them together as one layer. (because pool layer does not have it's own weights)



below is the common pattern for creating CNNs

Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,6)	4,704	456
POOL1	(14,14,6)	1,176	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	2,416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,120
FC4	(84,1)	84	10,164
Softmax	(10,1)	10	850

Andrew Ng

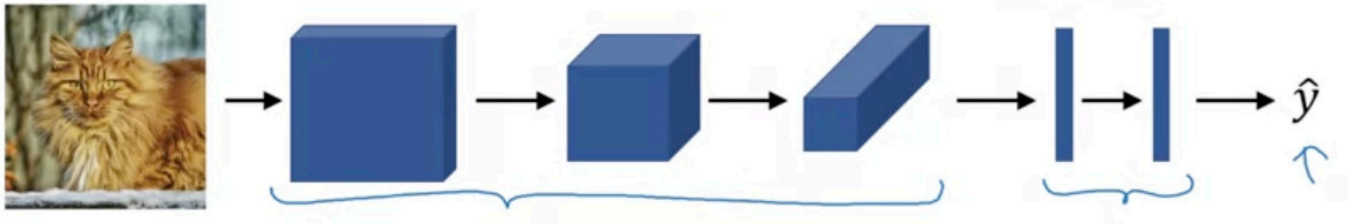
conv layers have relatively few parameters, if the parameter size drops quickly - generally not a good architecture for a CNN

Why Convolutions?

- if we were to connect every neuron with each other - in between layers, even a 34x34 photo generates =>10m parameters.
- each of the parameters can use the same set of parameters, (a filter for edge detection - vertical or horizontal - can be applied and used over and over again) **PARAMETER SHARING**
- **SPARSITY OF CONNECTIONS** - in each layer, each output value depends only on a small number of inputs. - a 3x3 filter, a parameter is only calculated from 3x3=9 pixels.

Putting it together

Training set $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$.



Cost $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Use gradient descent to optimize parameters to reduce J

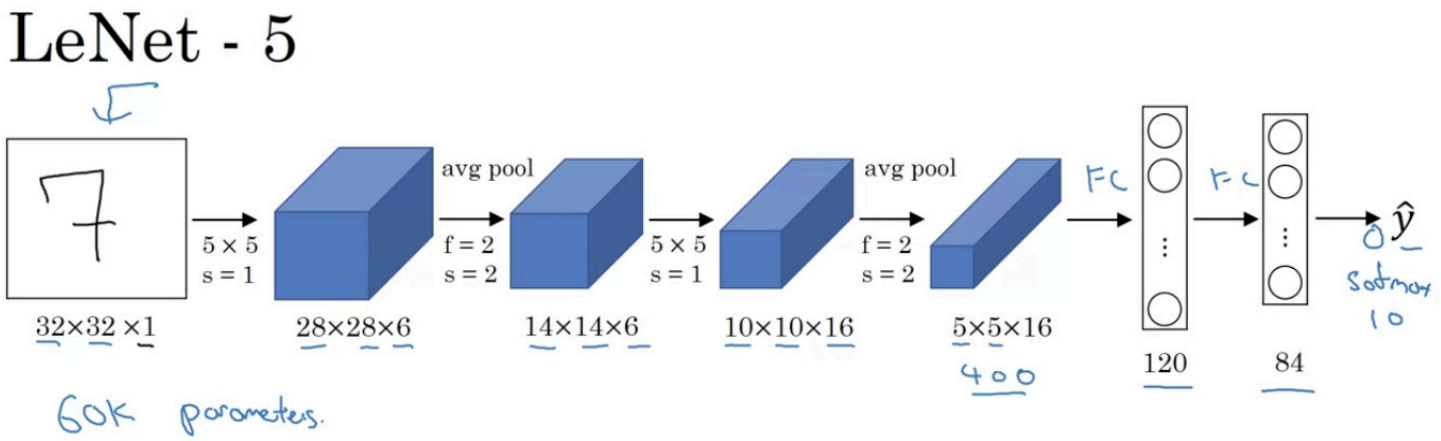
Andrew Ng

Case Studies on CNNs

Classic Networks:

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

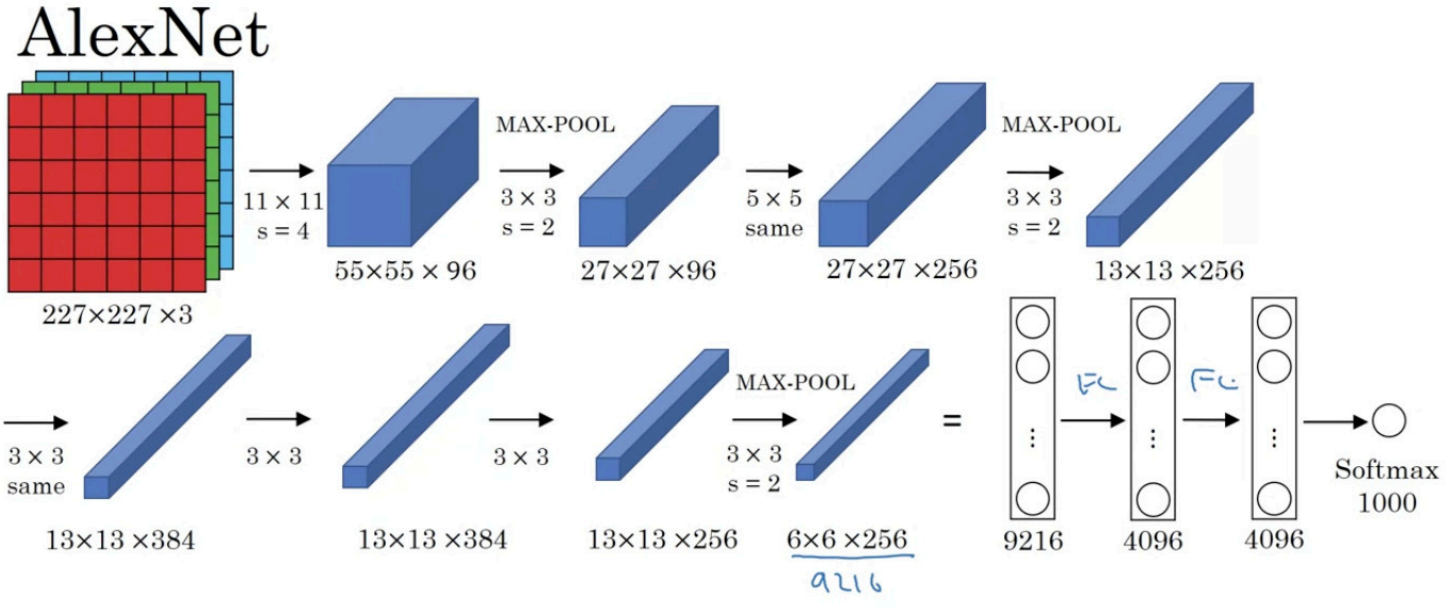
LeNet-5



Modern: not avg. pool, max pool - softmax would be used - much more parameters.
conv - pool - conv - pool - ... - fully connected layers -> pattern is still similar (in some)

- back then sigmoid or tanh, not softmax
- filter sizes are different, computers were slow, different reasons.
- for paper, focus on section 2, quick look at 3.

AlexNet

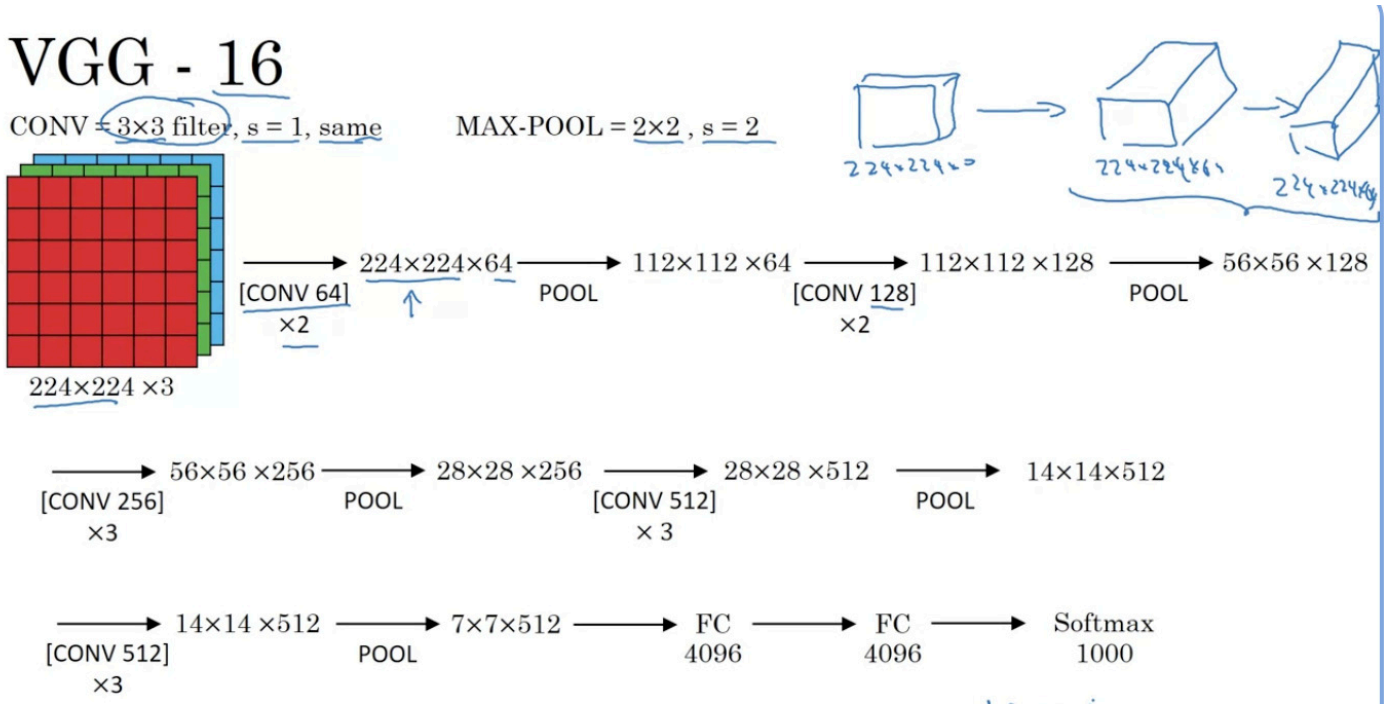


- a lot of similarities with LeNet-5, much bigger (60m parameters, big train dataset)
- relu
- split across 2 different GPUs. makes it complicated, on paper.

- Local response normalization - not much effective but mentioned.
-

VGG-16

- simplified, not many hyperparameters, always use same conv and max pool filter

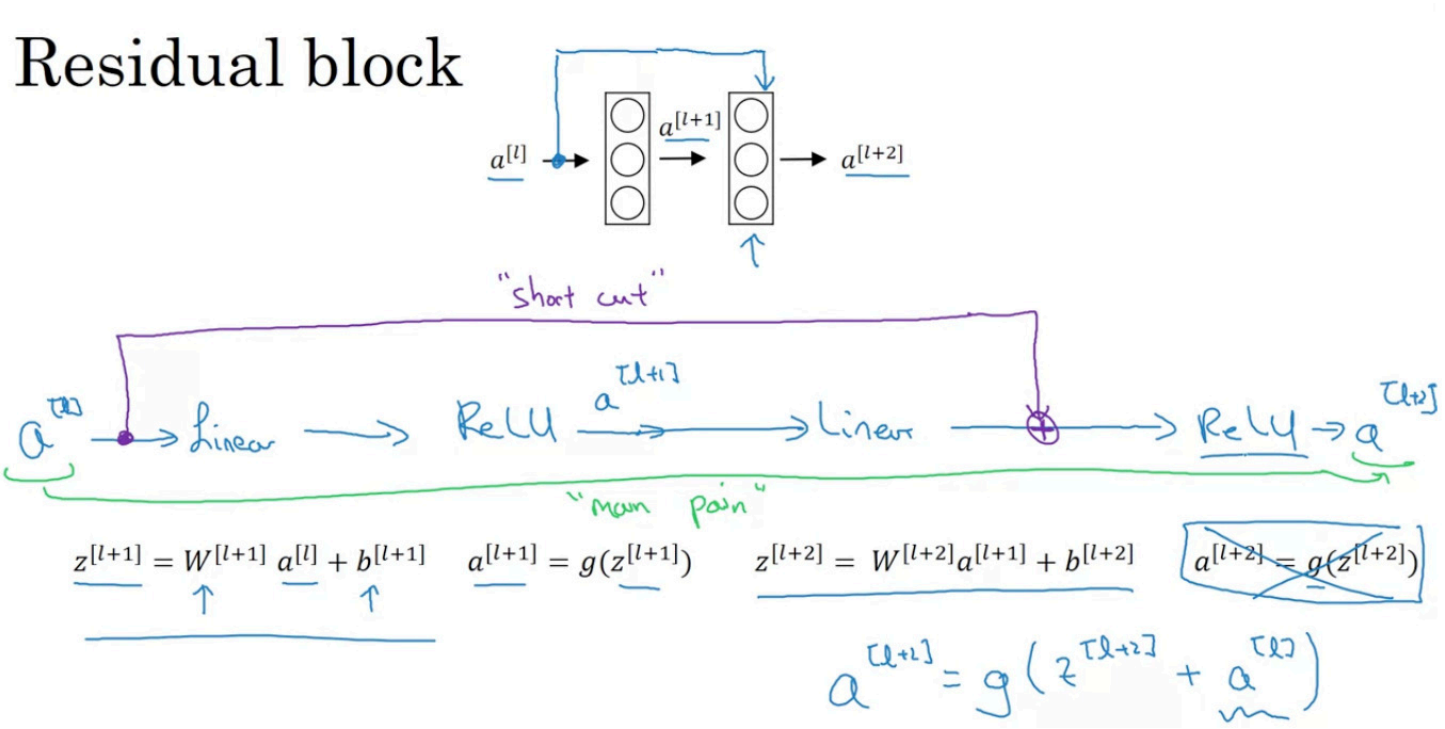


- 138 m parameters, but simple, uniform.
- VGG-19 - bigger version

ResNets

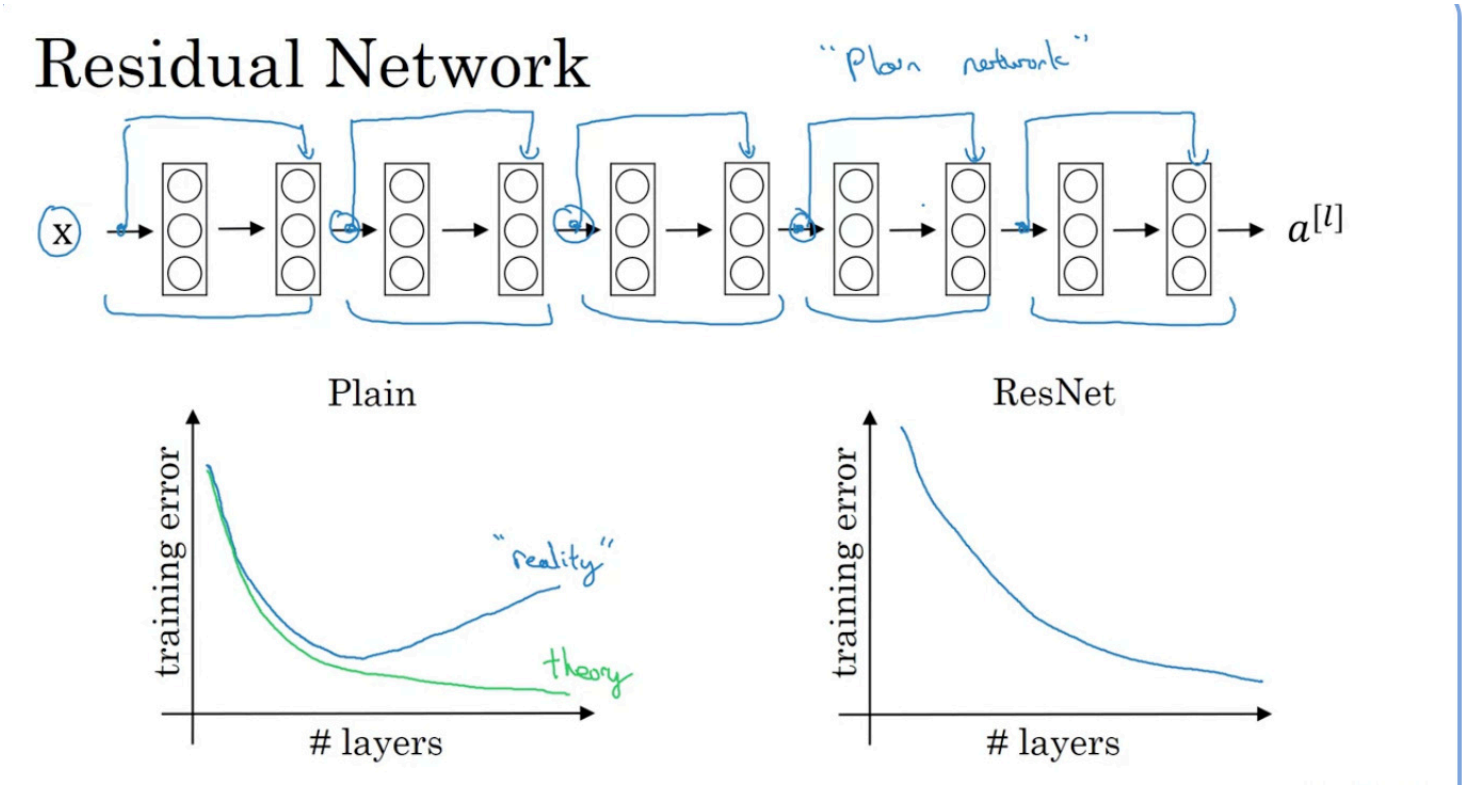
- Big NNs hard to train, vanishing, exploding gradients...

Residual Block



check later

Using residual blocks enable training much deeper neural networks.



Why do they work well?

Residual networks work well because they make it **easier for the network to learn the identity function**.

Let's break down why, using the image:

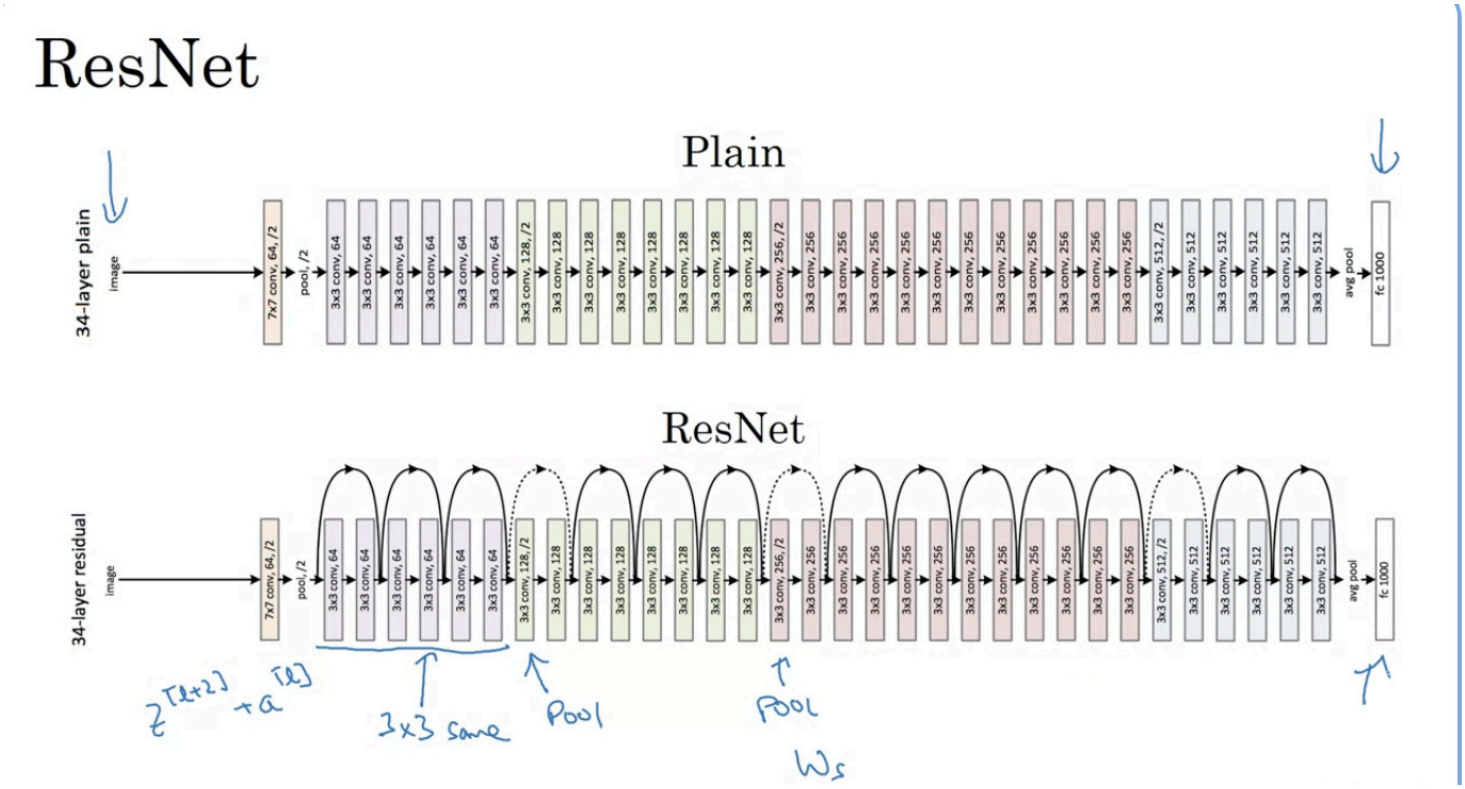
- **Standard Neural Network (Top path):** A "Big NN" directly tries to learn a complex function from input X to output $a^{[1]}$.
- **Residual Block (Bottom path):** Instead of directly learning the output, the residual block learns a **"residual"** or a **"difference"** function.
  - It takes the input $a^{[2]}$ (from a previous layer) and passes it through a few layers (represented by the circles). These layers are meant to learn some function, let's call it $F(a^{[3]})$.
  - **Crucially, it adds back the original input $a^{[4]}$ to the output of these layers.** This is the "residual connection" or "skip connection".
- **Mathematical Formulation:** The image shows the equation:
  $a^{[5]} = g(z^{[6]} + a^{[7]})$
  This means the activation in layer l+2 is the activation function g applied to the sum of:
  $z^{[8]}$ which is the output of the layers within the residual block (the $F(a^{[9]})$ part).
  $a^{[10]}$ which is the original input, the "residual" part.
- **Identity Function Advantage:** The image highlights: "Identity function is easy for Residual block to learn!".
  - If the layers in the residual block are not helpful, or if the optimal function is close to the identity, the network can easily learn to make the weights $W^{[11]}$ and biases $b^{[12]}$ in those layers close to zero.
  - In that case, $z^{[13]}$ becomes close to zero, and $a^{[14]}$ becomes approximately $g(0 + a^{[15]}) = g(a^{[16]})$. If gis ReLU and $a^{[17]}$ is positive (as indicated by "ReLU. a >= 0"), then $g(a^{[18]}) = a^{[19]}$.

- **So, if needed, the residual block can effectively act as an identity function, simply passing the input $a^{[20]}$ directly to the output $a^{[21]}$.**

Why is this good?

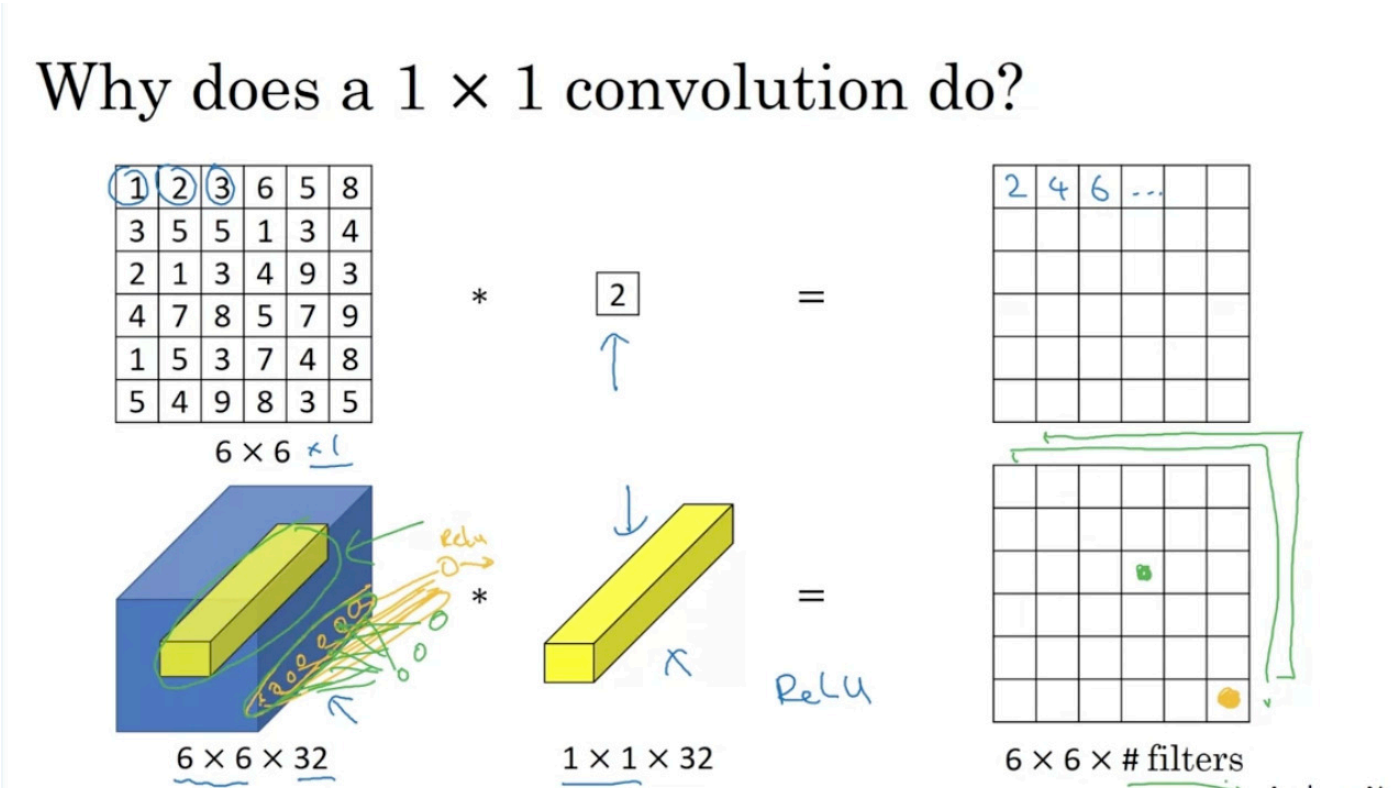
- **Deeper Networks:** In very deep networks, it can be difficult to learn complex functions layer by layer. Residual connections allow you to add layers without hurting performance. If a layer isn't helpful, it can just learn the identity function and not degrade the network. This enables training much deeper networks.
- **Vanishing Gradients:** Residual connections help with the vanishing gradient problem. The direct path from $a^{[22]}$ to $a^{[23]}$ provides an alternative path for gradients to flow backward during training, even if the layers in the residual block have small gradients.

In short, residual networks work well because they provide a shortcut for information to flow through the network, making it easier to learn the identity function and allowing for the training of deeper and more effective neural networks.



One by one convolution (?)

- seems pointless, but has use:



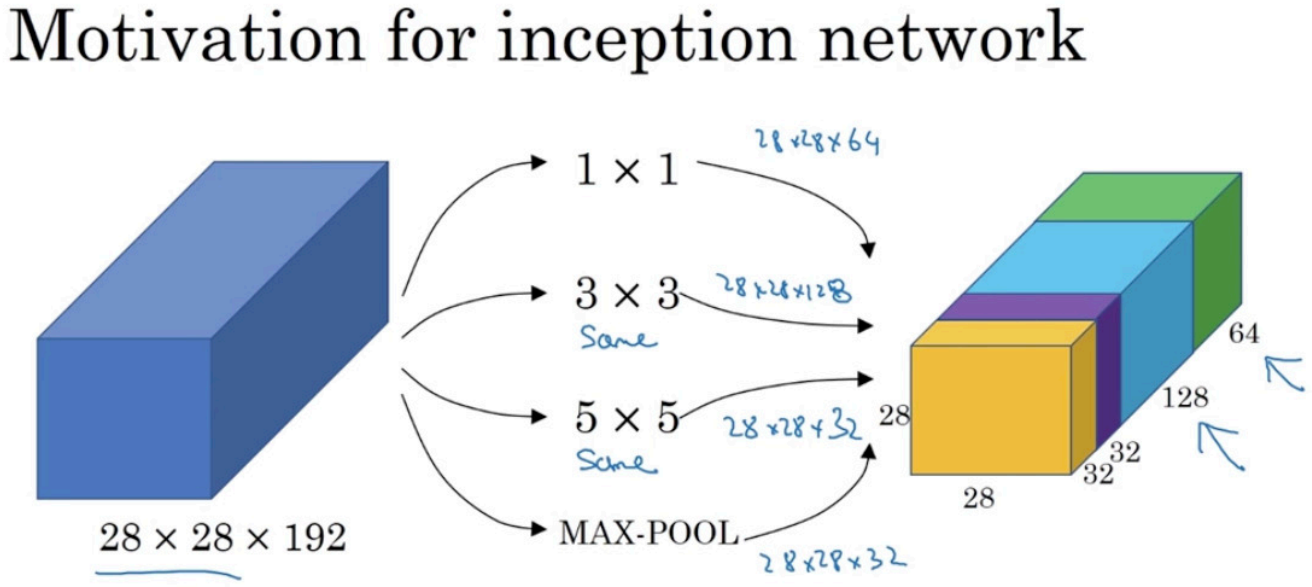
Having a fully connected neural network, for the number of channels of the image/filter.

Network in Network

Where to use it?
Have too much channels, dimensions can be reduced by pooling, we know it.
The # of channels can be reduced by using 1x1 convolutions while preserving the other dimensions.

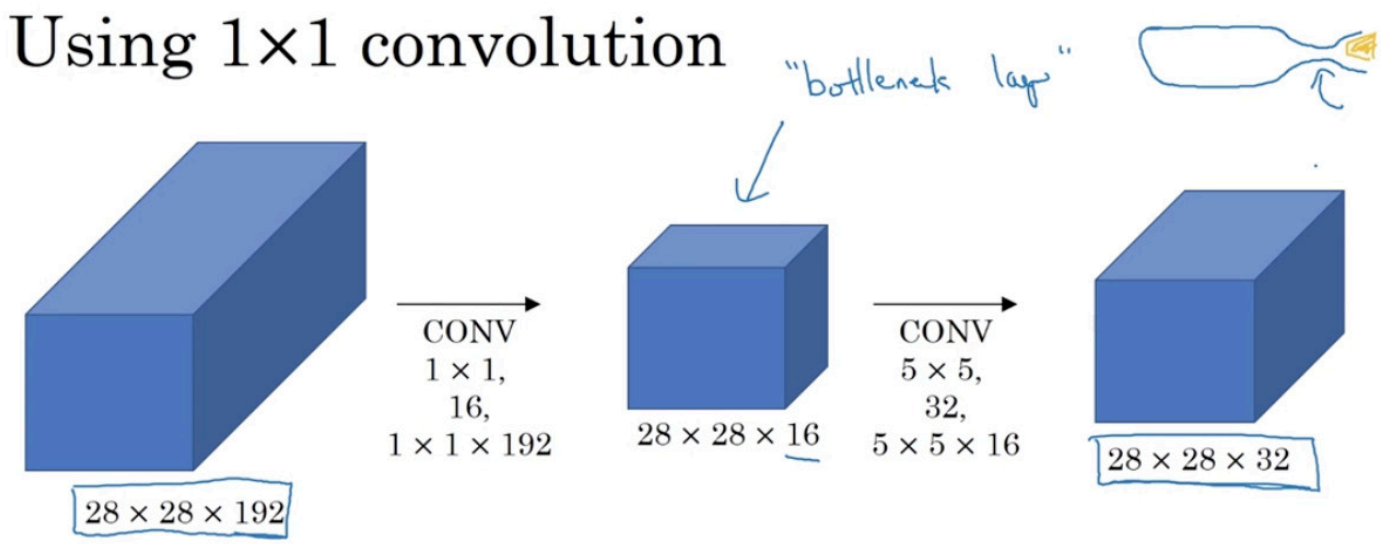
Inception Network Motivation

- inception layer: instead of choosing the filter size, conv layer or pool layer - basically anything - apply all. stack them all together (with the same dimensions).



for dimensions to match, needs padding of some sort
output = 28x28x256
Computational Cost !

Bottleneck Layer:



create the bottleneck layer, to drive down the computational cost.

Bottleneck layers reduce computational cost by **decreasing the number of channels before computationally expensive operations**, like larger convolutional filters. Let's see how it works in the image:

1. **Input Feature Map:** We start with a 28x28x192 feature map. Imagine this is a lot of channels to process.
2. **Bottleneck 1x1 Conv (Reduction):** The first 1x1 convolution layer with **16 filters** reduces the number of channels from 192 down to **16**. The output is now 28x28x16. This is the "bottleneck" - we've squeezed the information into fewer channels.
3. **Expensive 5x5 Conv (Feature Extraction):** The next layer is a 5x5 convolution with **32 filters**. This is where most of the computation would normally happen. However, because we reduced the channels in the previous step, this 5x5 convolution is now performed on a much smaller input depth (28x28x16 instead of 28x28x192).
4. **Bottleneck 1x1 Conv (Expansion):** The final 1x1 convolution with **32 filters** increases the channels back up to the desired output depth (in this case, it seems to stay at 32, 28x28x32). This layer can also be used to increase the channels if needed for the next stage of the network.

Why does this reduce computational cost?

- **Dominant Cost in Conv Layers:** The computational cost in convolutional layers is heavily influenced by the number of input and output channels, as well as the kernel size.
- **Reduced Operations in 5x5 Conv:** The 5x5 convolution is the most computationally expensive part. By reducing the input channels to just 16 using the 1x1 bottleneck, we significantly decrease the number of operations needed for the 5x5 convolution.

Think of it like a pipeline:

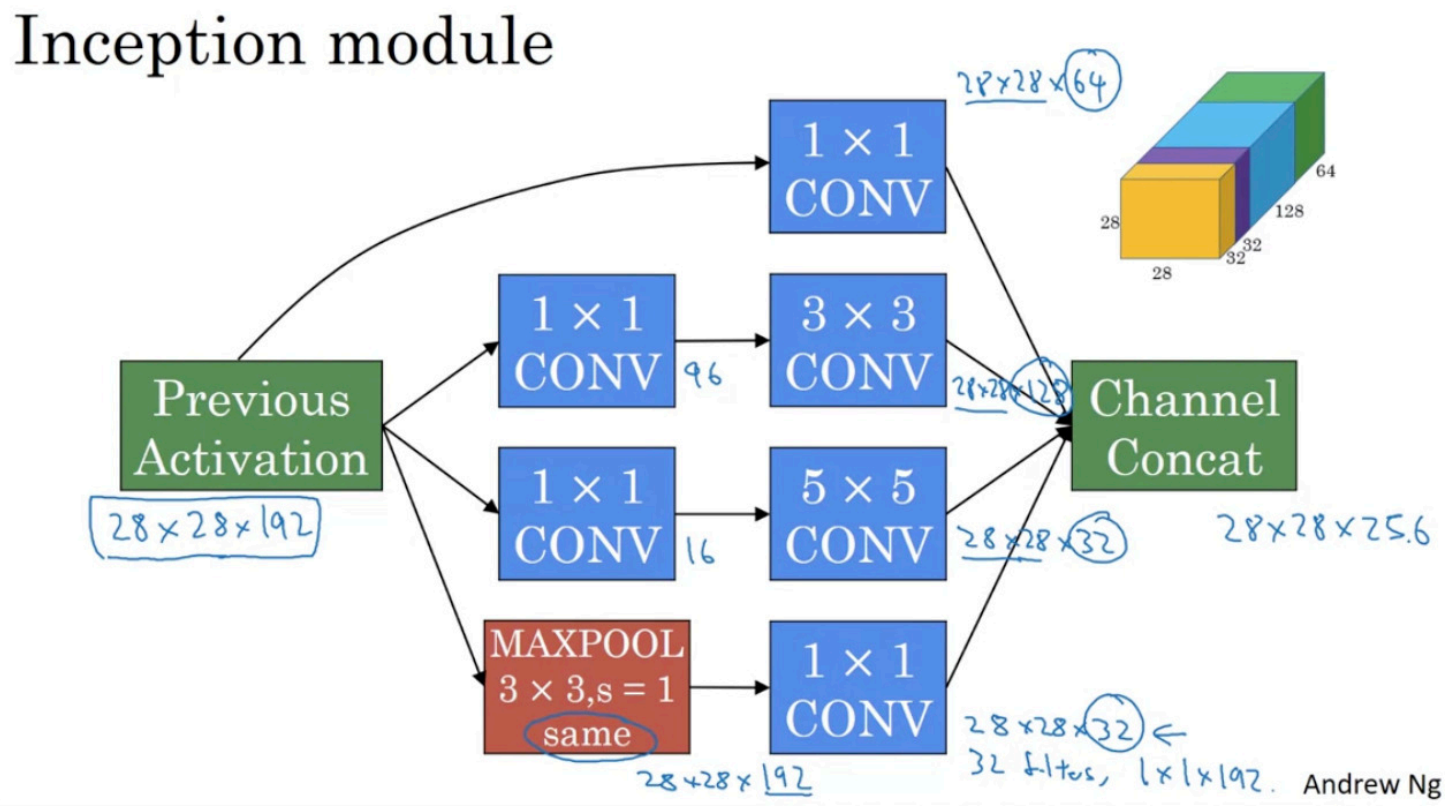
Imagine you are processing a lot of thick pipes (many channels). It's hard and expensive to work with them directly (like a 5x5 conv on 192 channels).

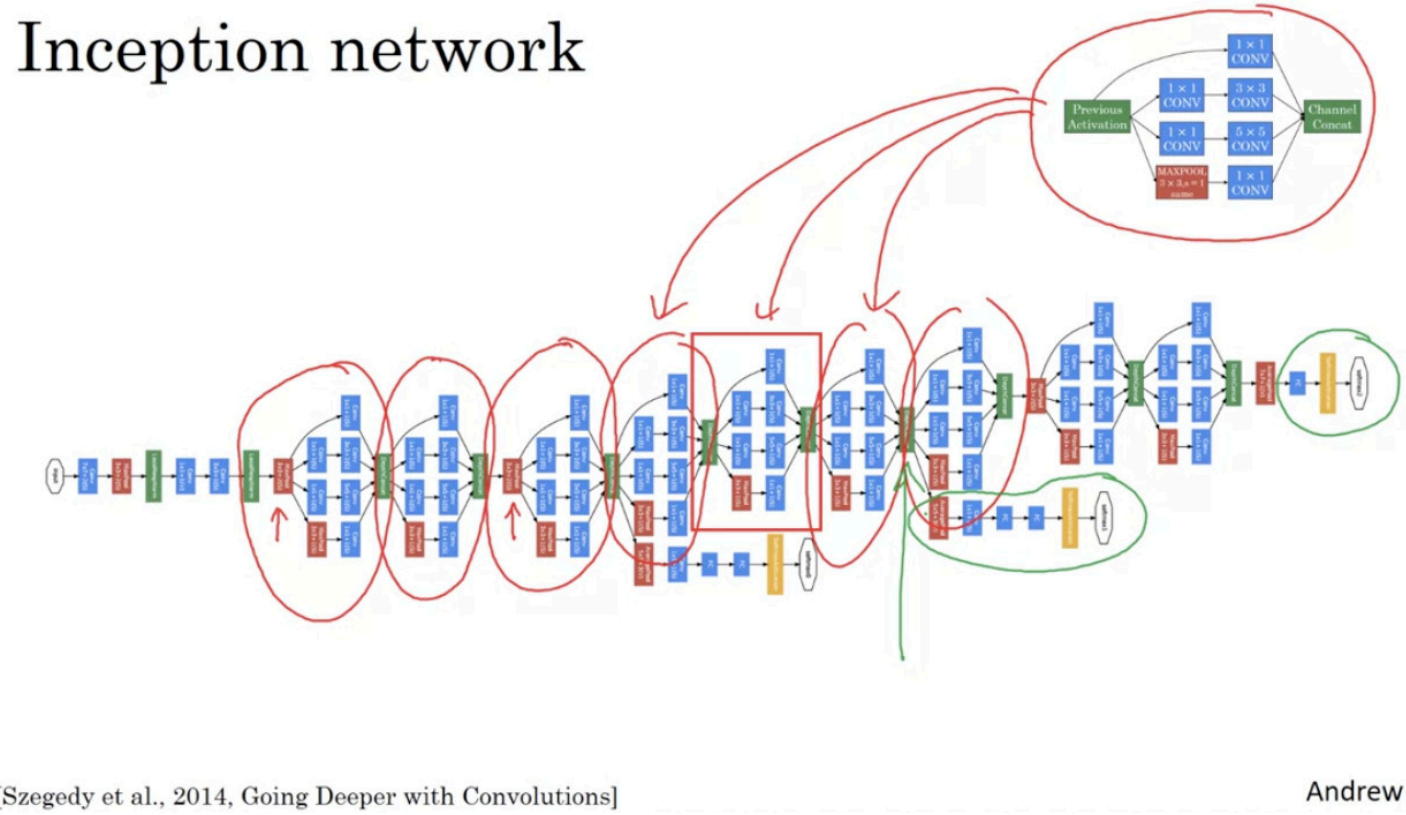
A bottleneck layer is like:

1. **Squeezing the pipes:** The first 1x1 conv squeezes the thick pipes into thinner pipes (reduces channels).
2. **Working on thinner pipes:** It's now much easier and cheaper to process the thinner pipes (the 5x5 conv on fewer channels).
3. **Expanding back if needed:** The second 1x1 conv can expand the pipes back to the desired thickness if necessary (increase/adjust channels for the next layer).

In short, the bottleneck layer uses 1x1 convolutions to strategically reduce the depth of feature maps before expensive convolutional operations, making the overall computation much more efficient.

Building Inception Module





side branches takes a hidden layer, has softmax, tries to predict what is the output label. Another detail in output label. Even at the hidden - middle layers, are not bad to predict the output label from the image. Also can have a use for preventing over fitting.

- GooLeNet
  Inception Network:

http://knowyourmeme.com/memes/we-need-to-go-deeper

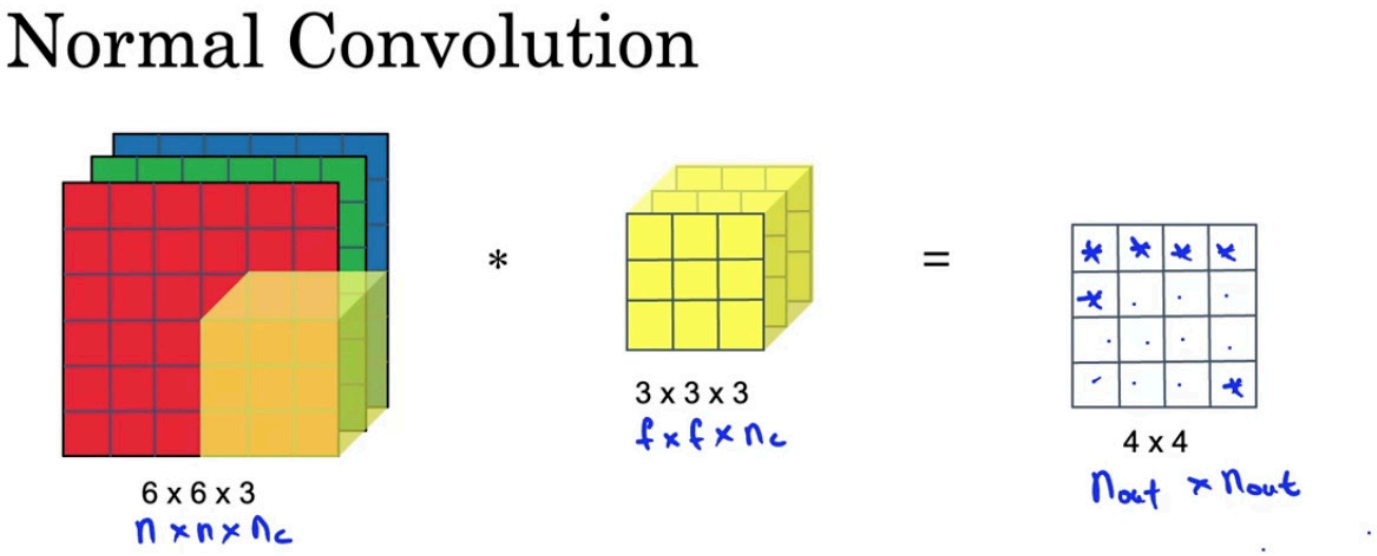
Andrew Ng

MobileNet

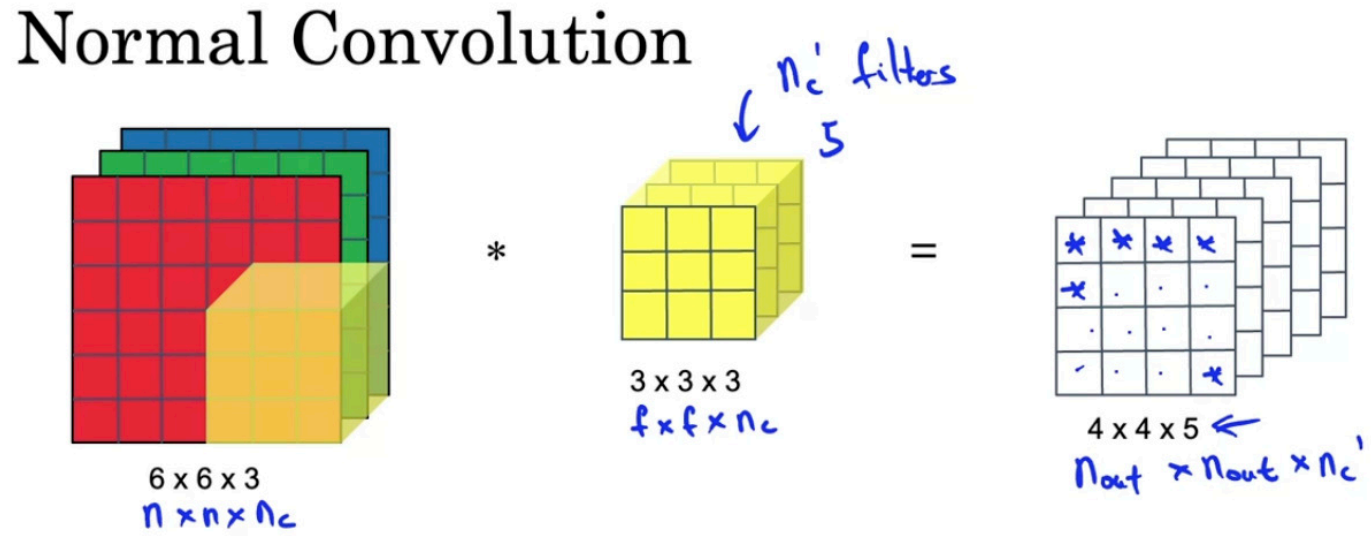
Another foundational CNN architecture.

- works better for low computational, mobile phones.

Normal vs. depthwise separable convolutions

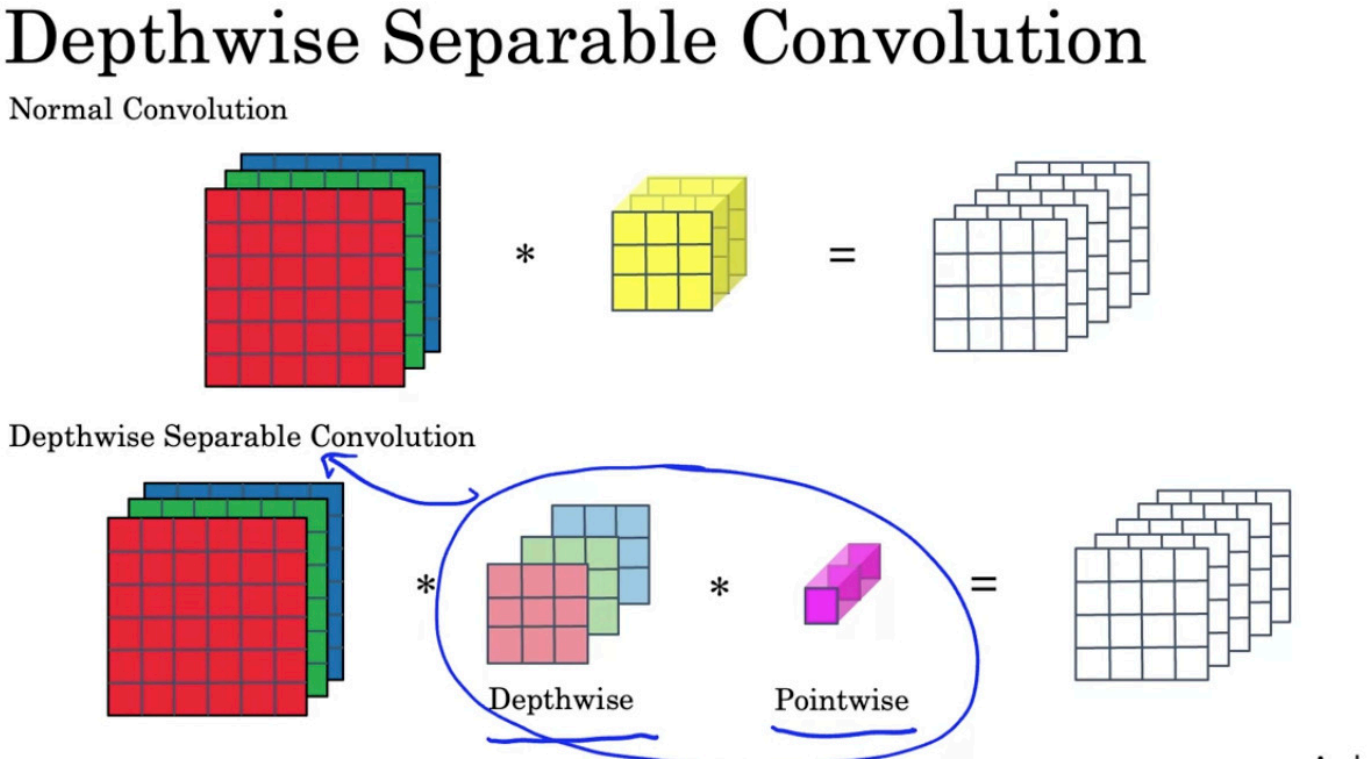


or you may have number of filters



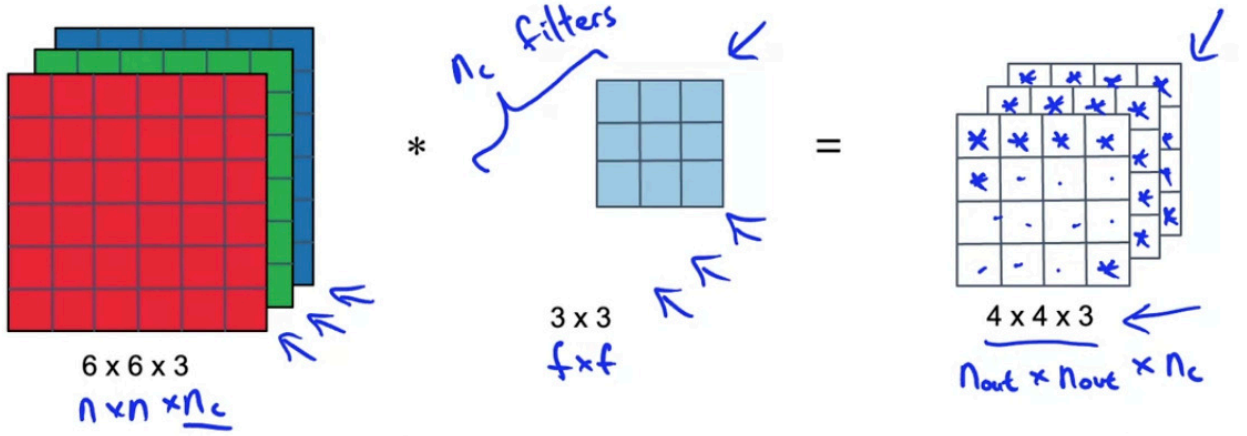
cost = 3x3x3 x 4x4 x 5 = 2160

In Contrast



use a step by step approach, you apply filters one by one, add them on the final output.

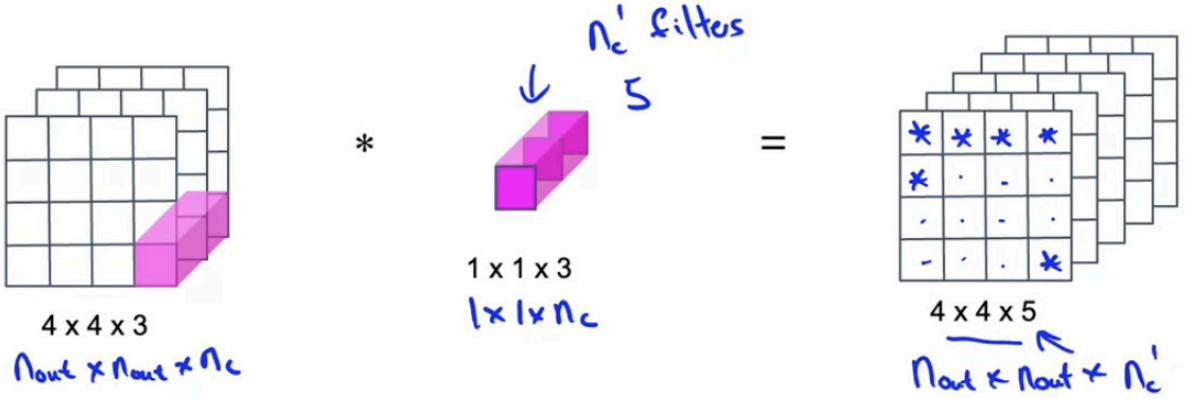
Depthwise Convolution



Computational cost	=	#filter params	x	# filter positions	x	# of filters
432	=	3x3	x	4x4	x	3

next step: take 4x4x3 and carry out the pointwise convolution

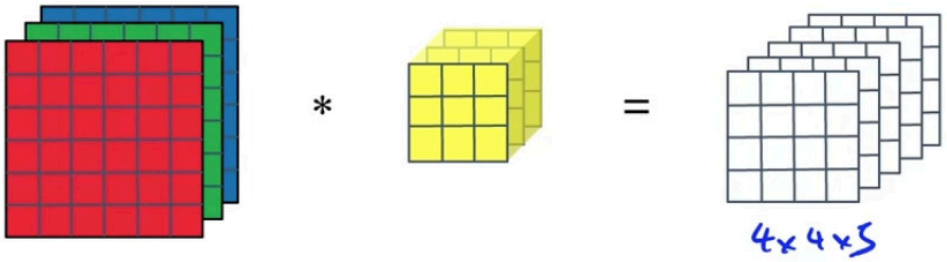
Pointwise Convolution



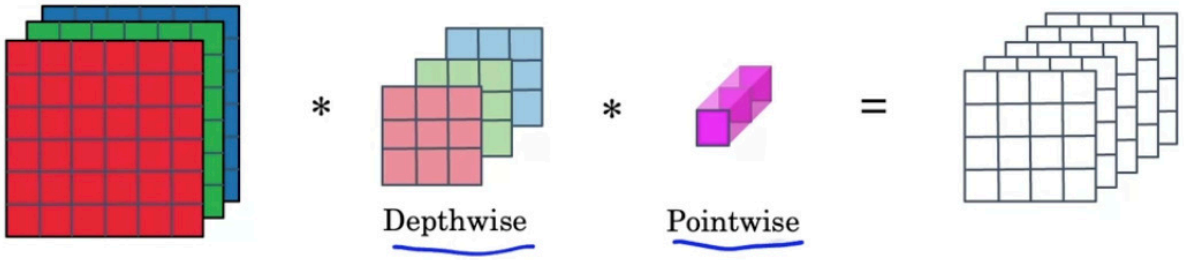
Computational cost	=	#filter params	x	# filter positions	x	# of filters
240	=	1x1x3	x	4x4	x	5

Depthwise Separable Convolution

Normal Convolution



Depthwise Separable Convolution



Cost Summary

Cost of normal convolution 2160

Cost of depthwise separable convolution

depthwise + pointwise

432 + 240 = 672

$$\frac{672}{2160} = 0.31$$

$$= \frac{1}{n_c} + \frac{1}{f^2}$$

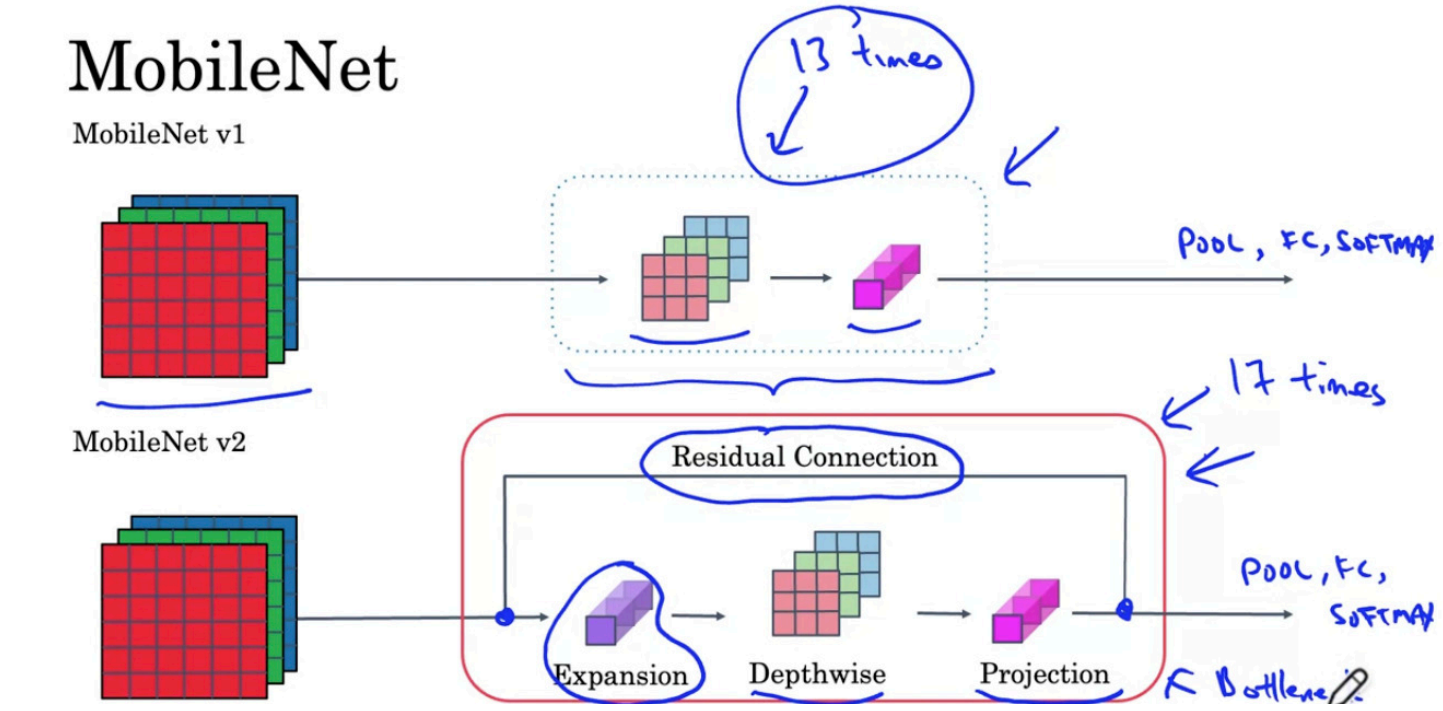
$$\frac{1}{5} + \frac{1}{9}$$

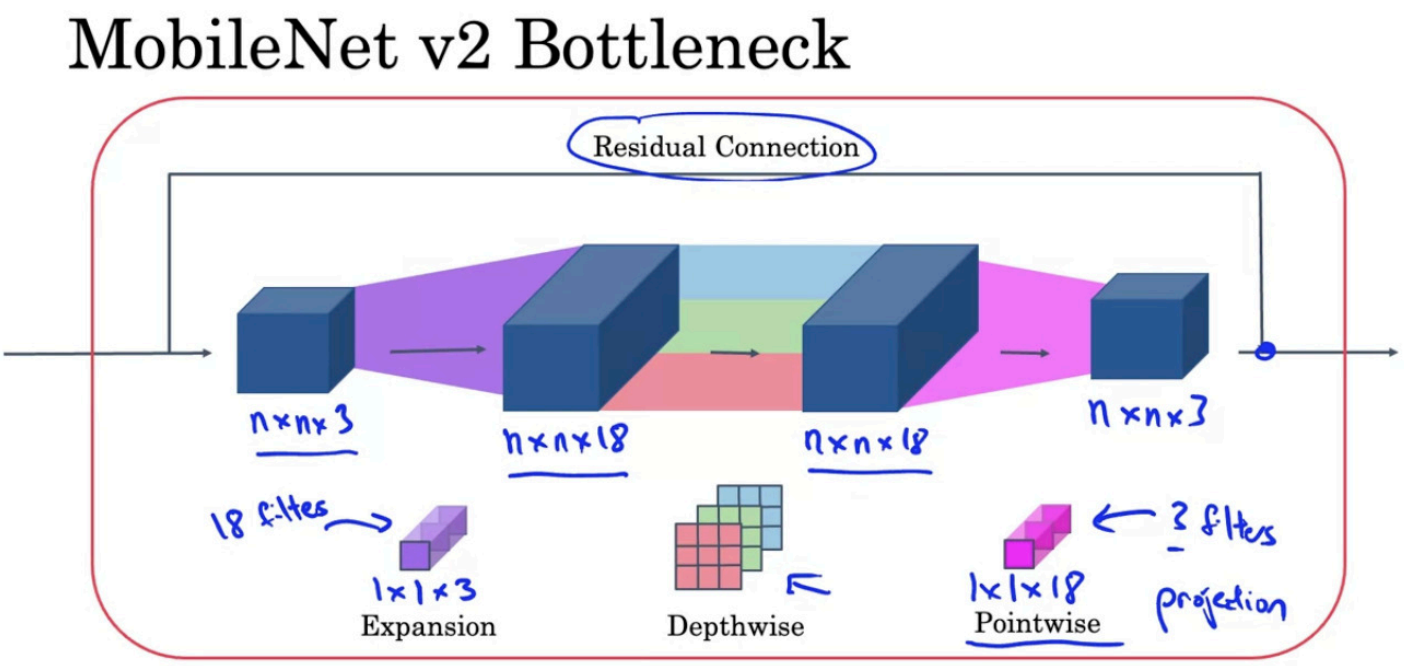
$$= \frac{1}{512} + \frac{1}{3^2}$$

~10 times cheaper

nc is much higher in practical cases.

MobileNet Architecture



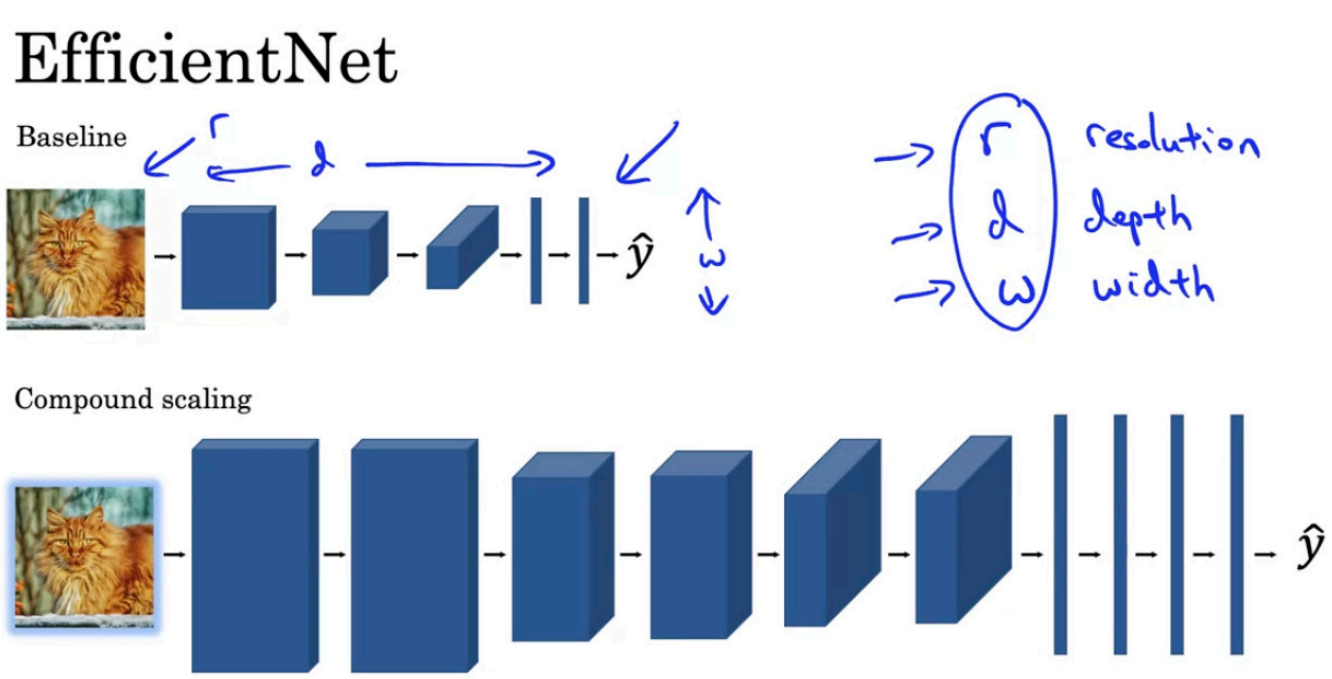


- Richer set of computations, while keeping the amount of memory needed in between the layers low (thanks to expansion - projection)

EfficientNet

how to automatically scale down or up, dependent on the hardware?

what is a good choice of r,d,w?



?

Practical Side

Using Open-Source Implementations

...kaggle, github, general suggestions.

Transfer Learning

- have a task in hand -> download a network itself with its code.
- get rid of the softmax layer and create your own softmax layer. (all the previous layers become frozen, you train only the parameters on the softmax layer)
- -> may get a good performance even with a small dataset. (DatathonAI'24)
- or you may train a shallow layer. (*save to disk* method)
  In the case of larger dataset for your task:
- Freeze fewer layers.
  In the case of a lot of data:
- Just like initializing the network, train the whole NN.

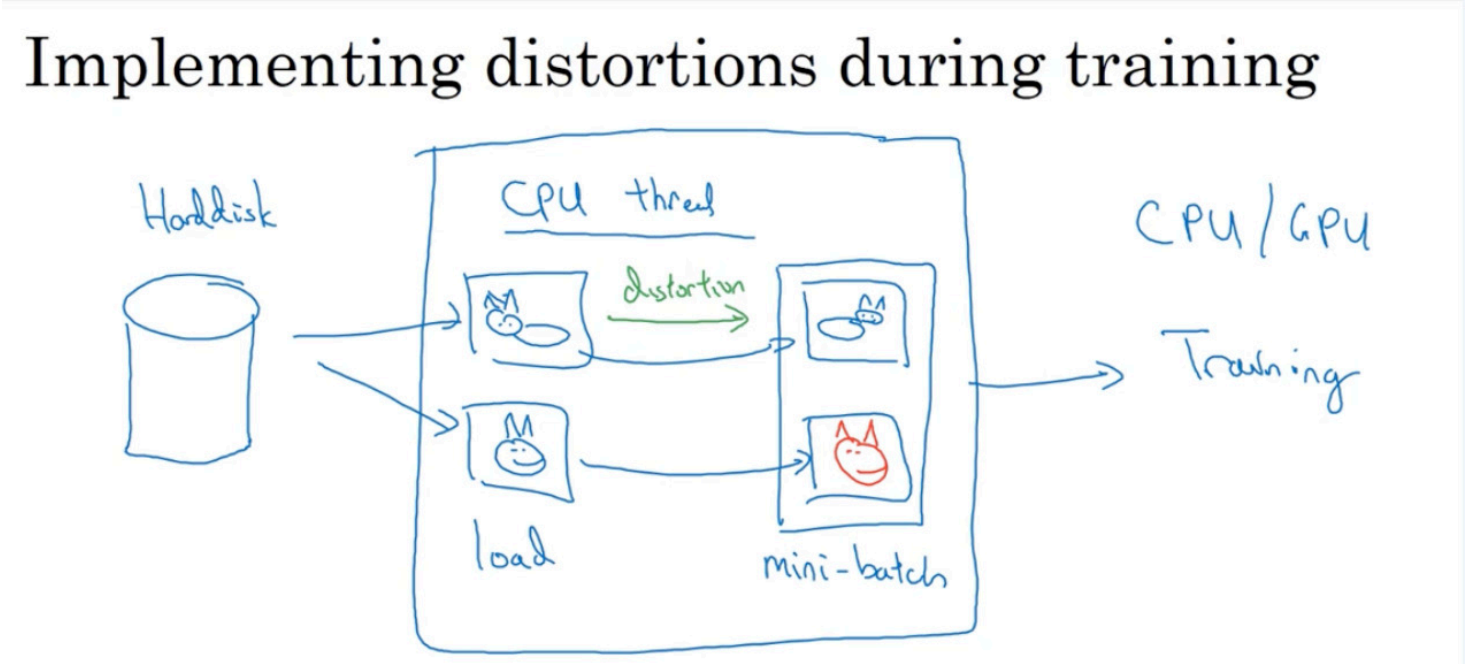
Since they are trained on vast amounts of data, most probably using a pretrained model always outperforms.

Data Augmentation

- Mirroring, Random Cropping, (from here, less used, less gains usually) Rotation, Shearing, Local Warping...
- Color shifting. (Advanced method: PCA - a different approach to sampling RGB. Referred in AlexNet paper. PCA Color Augmentation: a lot of red and blue, too less green, it changes Red and Blue much more than the green. *duh*)

Implementing Distortions During Training

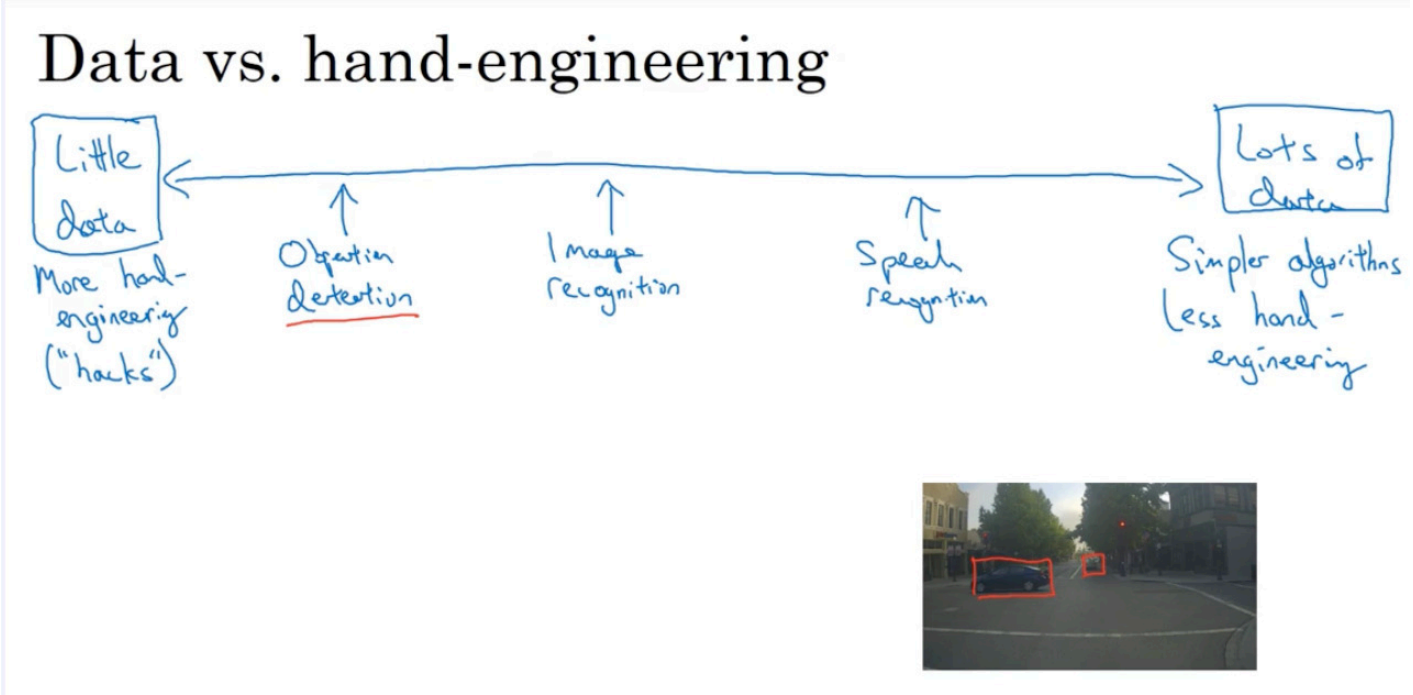
Small dataset, no problem. But with data augmentation, scales up quickly.



State (Status) of Computer Vision - may be outdated

Two sources of knowledge:

- labeled data
- hand engineered features/network architectures/other components



Tips for doing well on benchmarks/winning competitions

Andrew pretty much never use in practice

- Ensembling (3-15 networks used, not so practical for little gains.)
- Multi-crop at test time. (Run classifier on multiple versions of test images and average results. ex: 10-crop) ***

From the notebooks

What you should remember:

- Very deep "plain" networks don't work in practice because vanishing gradients make them hard to train.
- Skip connections help address the Vanishing Gradient problem. They also make it easy for a ResNet block to learn an identity function.
- There are two main types of blocks: The **identity block** and the **convolutional block**.
- Very deep Residual Networks are built by stacking these blocks together.

Notes on Transfer Learning with MobileNetV2

Core Goal: Efficiently build an image classifier (Alpaca vs. Not Alpaca) by leveraging a pre-trained Convolutional Neural Network (CNN), MobileNetV2, trained on a massive dataset (ImageNet).

1. The Power of Transfer Learning: Why and When

- **Efficiency and Reduced Data Needs:** Transfer learning allows you to build accurate models with significantly less training data and computational resources. Instead of training a deep network from scratch, you reuse knowledge learned from a large, general dataset (like ImageNet) and adapt it to your specific task.
- **Leveraging Pre-trained Features:** Pre-trained models have learned hierarchical feature representations from vast amounts of data. Early layers learn general features (edges, textures), while later layers learn more complex, task-specific features. Transfer learning reuses these learned features, especially the general ones, which are often beneficial across different image recognition tasks.
- **When to Use:** Transfer learning is most effective when:
  - You have a relatively small dataset for your new task.
  - Your new task is related to the task the pre-trained model was trained on (e.g., both are in image recognition domain).

2. Data Preparation: Setting the Stage for Effective Learning

- **Dataset Creation (image_dataset_from_directory):**
  - **Methodology:** Use TensorFlow's utility to efficiently create datasets directly from image directories. This simplifies data loading and management.
  - **Key Parameters:**
    - validation_split: Crucial for evaluating model generalization. Splits data into training and validation sets.
    - subset='training' and subset='validation': Specify which portion of the data to load for each dataset.

- seed: Ensure consistent splitting for reproducibility and to prevent overlap between training and validation sets.
- batch_size: Process data in batches to optimize memory usage and training speed.
- image_size: Resize images to a consistent size expected by MobileNetV2.

```
train_dataset = image_dataset_from_directory(directory, validation_split=0.2, subset='training', seed=42, image_size=IMG_SIZE, batch_size=BATCH_SIZE)
```

- **Preprocessing (preprocess_input):**
  - **Methodology:** Apply the same preprocessing steps used to train the pre-trained model. This ensures your input data is in a format that the model "understands" and for which its weights are optimized.
  - **For MobileNetV2:** Use tf.keras.applications.mobilenet_v2.preprocess_input. This typically involves normalizing pixel values to a specific range (e.g., [-1, 1]).

```
`preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input`
```

- **Data Augmentation (data_augmenter & Sequential API):**
  - **Methodology:** Artificially increase the diversity of the training data by applying random transformations to images (flips, rotations, zooms, etc.). This helps the model generalize better and reduces overfitting.
  - **Keras Sequential API:** A convenient way to define a series of data augmentation layers as a model itself.
  - **Example Augmentations:** RandomFlip('horizontal'), RandomRotation(0.2).

```
`def data_augmenter():     data_augmentation = tf.keras.Sequential()
data_augmentation.add(tf.keras.layers.experimental.preprocessing.RandomFlip("horizontal"))
data_augmentation.add(tf.keras.layers.experimental.preprocessing.RandomRotation(0.2))     return data_augmentation`
```

- **Prefetching (prefetch):**
  - **Methodology:** Optimize data loading pipeline by preparing the next batch of data while the current batch is being processed by the model. This prevents input bottlenecks and speeds up training.
  - **AUTOTUNE:** Let TensorFlow dynamically optimize prefetching parameters for best performance.

```
`AUTOTUNE = tf.data.experimental.AUTOTUNE train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)`
```

3. Transfer Learning - Feature Extraction: Adapting MobileNetV2 for Alpaca Classification

- **Using MobileNetV2 as a Feature Extractor:** Leverage the pre-trained MobileNetV2 model as a powerful feature extraction engine. The idea is that MobileNetV2's convolutional base has already learned to extract useful features from images.
- **Removing the Top Classification Layer (include_top=False):**
  - **Rationale:** The original top layer of MobileNetV2 is trained to classify 1000 ImageNet classes, which are irrelevant to the Alpaca vs. Not Alpaca task. We discard this layer and replace it with a new classification layer tailored to our problem.
  - **Implementation:** Set include_top=False when loading MobileNetV2.

```
`base_model = tf.keras.applications.MobileNetV2(input_shape=input_shape,
include_top=False,
weights=base_model_path)`
```

- **Freezing the Base Model (base_model.trainable = False):**
  - **Rationale:** Prevent the weights of the pre-trained MobileNetV2 convolutional base from being updated during initial training. This preserves the valuable features learned from ImageNet and focuses training on the newly added classification layers. This is "feature extraction" – we're using the pre-trained network as is to extract features.
  - **Implementation:** Set base_model.trainable = False.
- **Adding a New Classification Layer (Functional API):**
  - **Methodology:** Build a new model using the Keras Functional API. This involves:
    1. **Input Layer:** Define the input shape for your images.
    2. **Data Augmentation Layer:** Apply the data_augmenter sequentially.
    3. **Preprocessing Layer:** Apply preprocess_input.
    4. **Base Model (Frozen):** Pass the preprocessed input through the frozen base_model.
    5. **Global Average Pooling (GlobalAveragePooling2D):** Reduce the spatial dimensions of the feature maps from the base model to a feature vector.
    6. **Dropout (Dropout):** Regularization to prevent overfitting.
    7. **Dense Output Layer (Dense(1)):** A single neuron with linear activation for binary classification (use from_logits=True in BinaryCrossentropy loss).

```
`def alpaca_model(image_shape=IMG_SIZE,
data_augmentation=data_augmenter()):     inputs =
tf.keras.Input(shape=input_shape)     x = data_augmentation(inputs)     x =
preprocess_input(x)     x = base_model(x, training=False) # training=False
for batch norm     x = tf.keras.layers.GlobalAveragePooling2D()(x)     x =
tf.keras.layers.Dropout(0.2)(x)     outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)     return model`
```

- **Training the New Classifier Layer:** Compile and train the model. Initially, train only for a few epochs as only the newly added classification layers are being trained.

```
    `model2.compile(optimizer=tf.keras.optimizers.Adam(lr=base_learning_rate),
loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
```

```
metrics=['accuracy']) history = model2.fit(train_dataset,
validation_data=validation_dataset, epochs=initial_epochs)`
```

4. Fine-tuning: Refining the Model for Better Accuracy

- **Unfreezing Layers:** After initial training of the classifier, unfreeze some of the top layers of the base_model(MobileNetV2). This allows these layers to be adjusted to the specific nuances of the Alpaca/Not Alpaca dataset.
- **Rationale for Fine-tuning Later Layers:**
  - **Hierarchical Feature Learning:** CNNs learn features in a hierarchical manner. Early layers capture general features (edges, textures), while later layers learn more complex, task-specific features.
  - **Adapting High-Level Features:** Fine-tuning the later layers allows the model to adapt the more complex features to be specifically relevant for distinguishing alpacas, while still benefiting from the general feature extraction capabilities of the earlier, frozen layers.
- **Choosing Layers to Fine-tune (fine_tune_at):** The number of layers to unfreeze is a hyperparameter. Start by unfreezing a small number of the top layers and experiment. The notebook suggests fine-tuning from layer 120 onwards in MobileNetV2.

```
    `base_model = model2.layers[4] # Get the base MobileNetV2 model
base_model.trainable = True # Unfreeze the entire base model  fine_tune_at =
120 for layer in base_model.layers[:fine_tune_at]: # Freeze layers before
fine_tune_at     layer.trainable = False`
```

- **Lower Learning Rate for Fine-tuning:** Use a significantly smaller learning rate (e.g., 1/10th of the initial learning rate) during fine-tuning.
  - **Rationale:** Avoid disrupting the pre-trained weights too much. Smaller steps allow for finer adjustments and prevent overfitting to the smaller, new dataset.

```
`optimizer = tf.keras.optimizers.Adam(learning_rate=0.1 *
base_learning_rate) # Reduced learning rate`
```

- **Re-compile and Continue Training:** Re-compile the model with the new optimizer and loss, and continue training for a few more epochs (fine_tune_epochs).

```
    `model2.compile(loss=loss_function, optimizer=optimizer, metrics=metrics)
history_fine = model2.fit(train_dataset,
epochs=total_epochs,                         initial_epoch=history.epoch[-1],
# Continue from last epoch
validation_data=validation_dataset)`
```

5. Key Takeaways - Methodological Steps for Transfer Learning & Fine-tuning

1. **Choose a Pre-trained Model:** Select a model pre-trained on a large, relevant dataset (e.g., ImageNet for image tasks). MobileNetV2 is a good choice for efficiency.
2. **Prepare Your Data:** Create datasets, apply necessary preprocessing (like normalization), and use data augmentation to increase data diversity.
3. **Feature Extraction Phase:**
   - Load the pre-trained model, excluding the top classification layer (include_top=False).
   - Freeze the base model (base_model.trainable = False).
   - Add your custom classification layers on top.
   - Train only the newly added layers with a suitable learning rate.
4. **Fine-tuning Phase (Optional but often beneficial):**

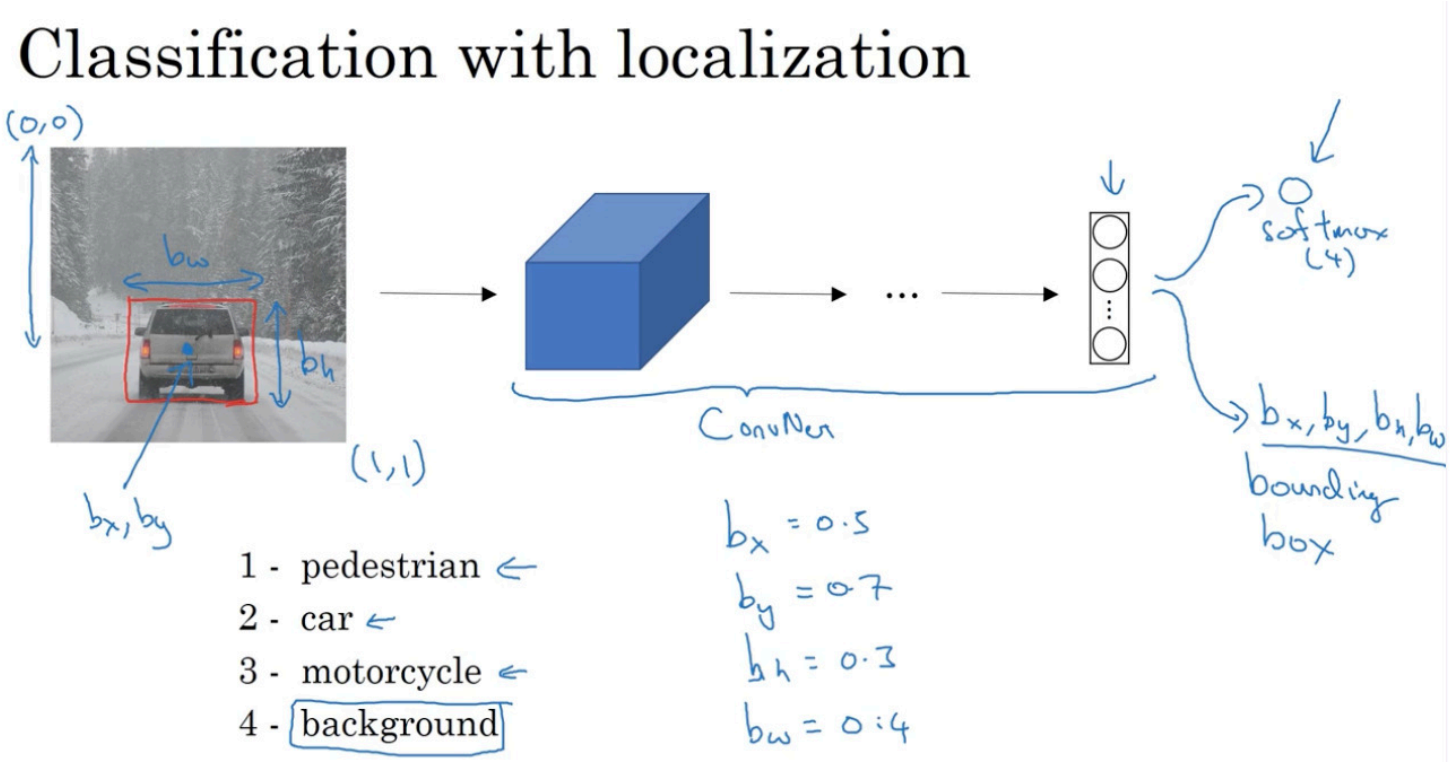
- Unfreeze some of the top layers of the pre-trained base model (base_model.trainable = True).
- Freeze layers below a certain point in the base model to retain general features.
- Use a much lower learning rate for fine-tuning.
- Continue training to subtly adjust the unfrozen layers to your specific task.

Object Detection

Object Localization and Detection

Classification - Classification with localization: 1 object, localize and recognize.
Detection - Multiple objects within a single image.

Classification with Localization



pedestrian, car, motorcycle, background(?)

Defining the target label y

Object Categories:

Here are the categories of objects being considered, labeled 1 through 4:

- 1 - pedestrian
- 2 - car <- (Indicates 'car' is likely the object in the example image)
- 3 - motorcycle
- 4 - background <- (Indicates 'background' is also a considered category)

  Output Requirements:

  For each detected object, the model needs to output the following:
- bx, by, bh, bw: These represent the bounding box coordinates. It's likely:
  - bx, by: Center coordinates of the bounding box
  - bh: Height of the bounding box
  - bw: Width of the bounding box
- class label (1-4): The category of the object, corresponding to the list above (pedestrian, car, motorcycle, background).

  Input Images (X):

The input X consists of images. Examples shown:

- [Image of a car with a red bounding box drawn around it] - This image contains a car, and the red box visually highlights the bounding box.
- [Image of a snowy landscape/road] - This image likely represents a scene where objects might be detected.

  Loss Function L(y_hat, y):

  This defines how the model's prediction (y_hat) is compared to the true target label (y). It's presented as a piecewise function:
- **If y_1 = 1 (Object is present):**

  The loss is calculated as the sum of squared differences between predicted and true values for components 1 through 8.

  This is represented as: (y_hat_1 - y_1)^2 + (y_hat_2 - y_2)^2 + ... + (y_hat_8 - y_8)^2
- **If y_1 = 0 (No object present):**

  The loss is only calculated on the first component.

  This is represented as: (y_hat_1 - y_1)^2

  Target Label Vector 'y' Structure:

  The target label y is a vector, structured as follows:

  y = [Pc / bx / by / bh / bw / c1 / c2 / c3]

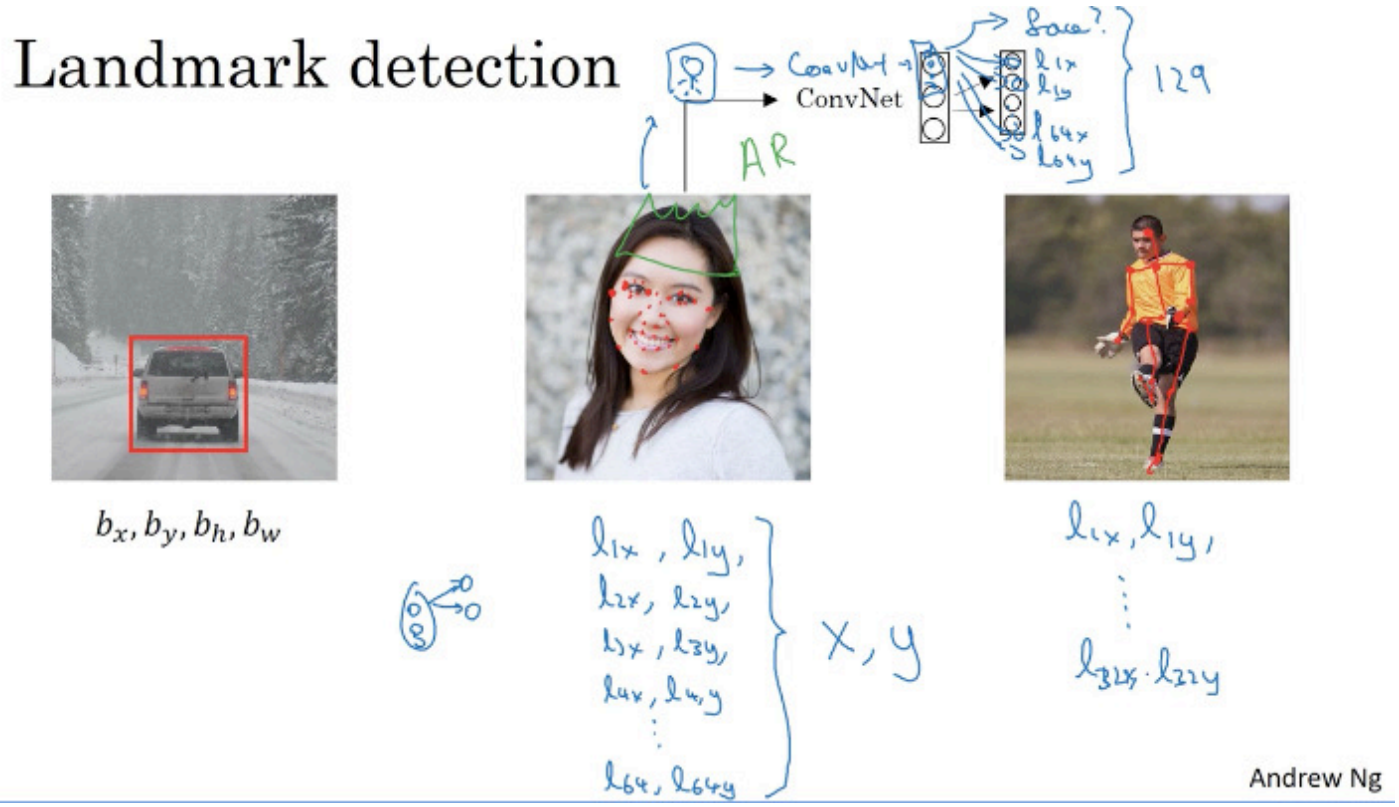
  Where:
- Pc: Probability of an object being present. It seems Pc = y_1 from the Loss Function definition.
- bx, by, bh, bw: Bounding box parameters, as defined earlier.
- c1, c2, c3: Class probabilities. It's implied that only one of these will be 1 and the rest 0 to indicate the class label (though the exact mapping to classes 1-4 isn't explicitly shown in this vector).

Example Target Vectors (x, y):

- **Object Present Case:**
  y = [1 / bx / by / bh / bw / c1 / c2 / c3]
  Here, Pc = 1, indicating an object is present. The rest of the vector provides the bounding box and class information.
- **"Don't Care" / Background Case:**
  [0 / ? / ? / ? / ? / ? / ? / ?]
  Here, Pc = 0, indicating no object is present (or "don't care"). The question marks ? suggest that the values for bounding box and class probabilities are not relevant or "don't care" when no object is detected.

Landmark Detection

- *where is the corner (or all 4 corners of both eyes) of the eye?*



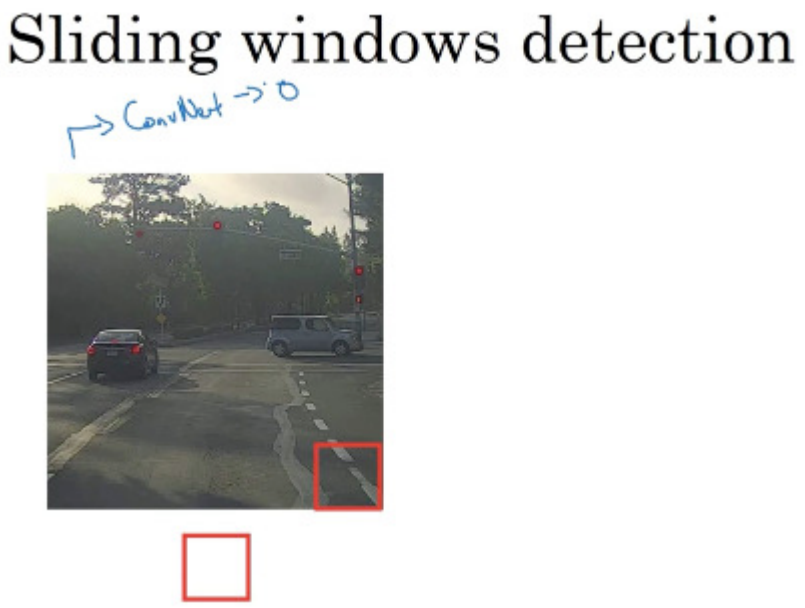
landmarks/labels should be consistent always.

Object Detection

Sliding windows detection algorithm

- Train a ConvNet, that takes in centralized photo, car or not 0/1

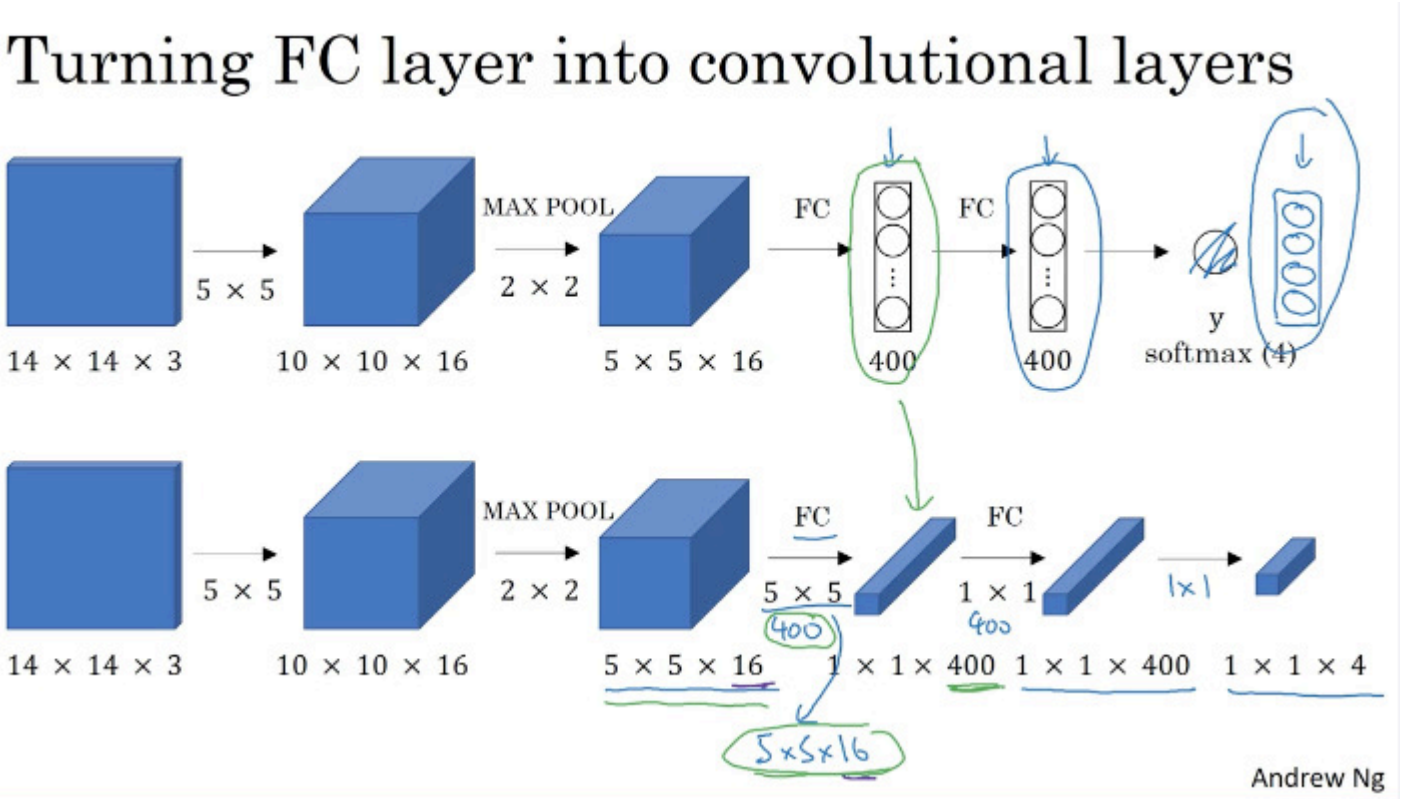
- Then take windows with sliding on the image and feed each of them to the ConvNet.



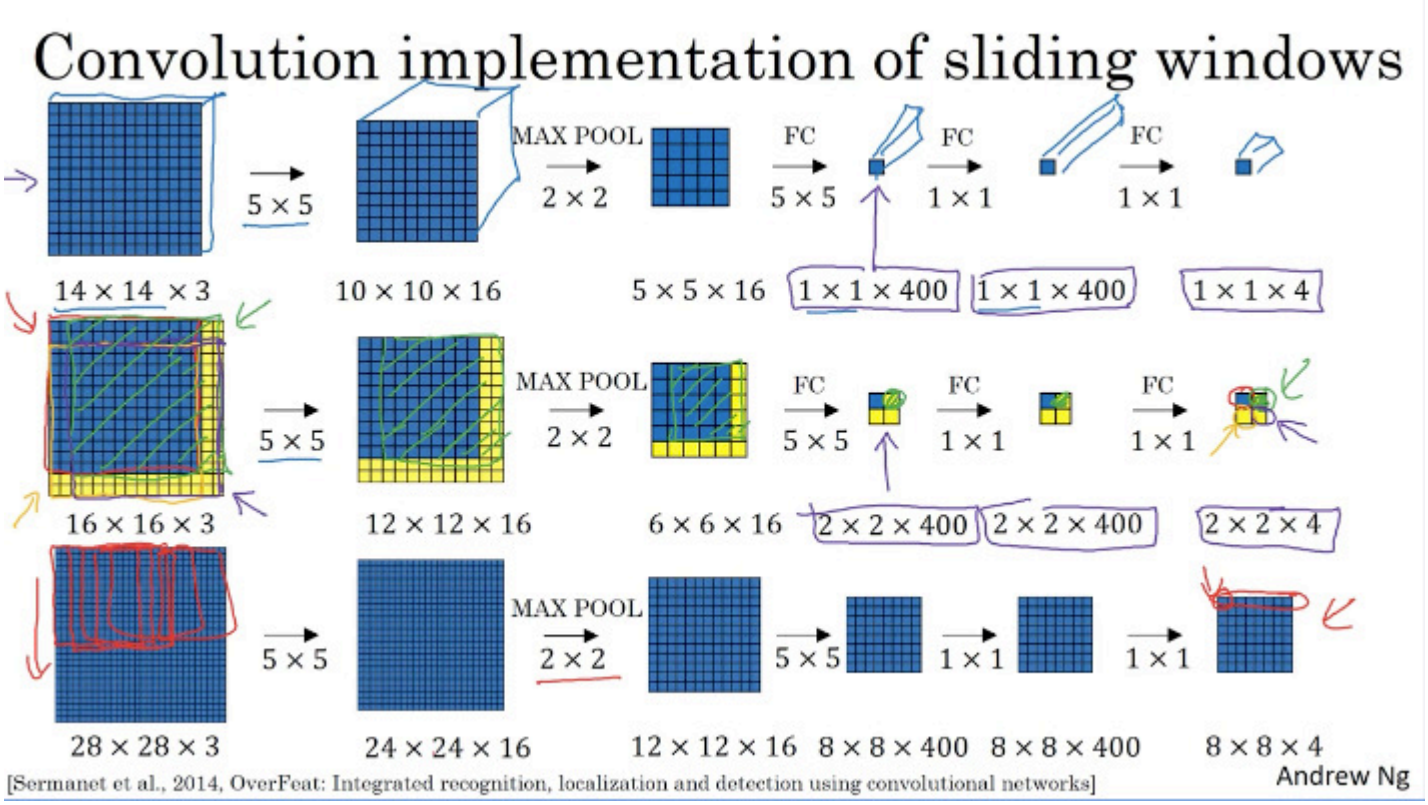
*Huge Disadvantage: Computation Cost

- Stride Big: Hurts Performance
  simpler algorithms - linear fncs - sliding window was OK, then algorithms became more complex

How to implement it convolutionally?



converting FC layer to Conv layer



instead of forcing to run forward prop separately, while using sliding window approach, apply it to later versions of layers.

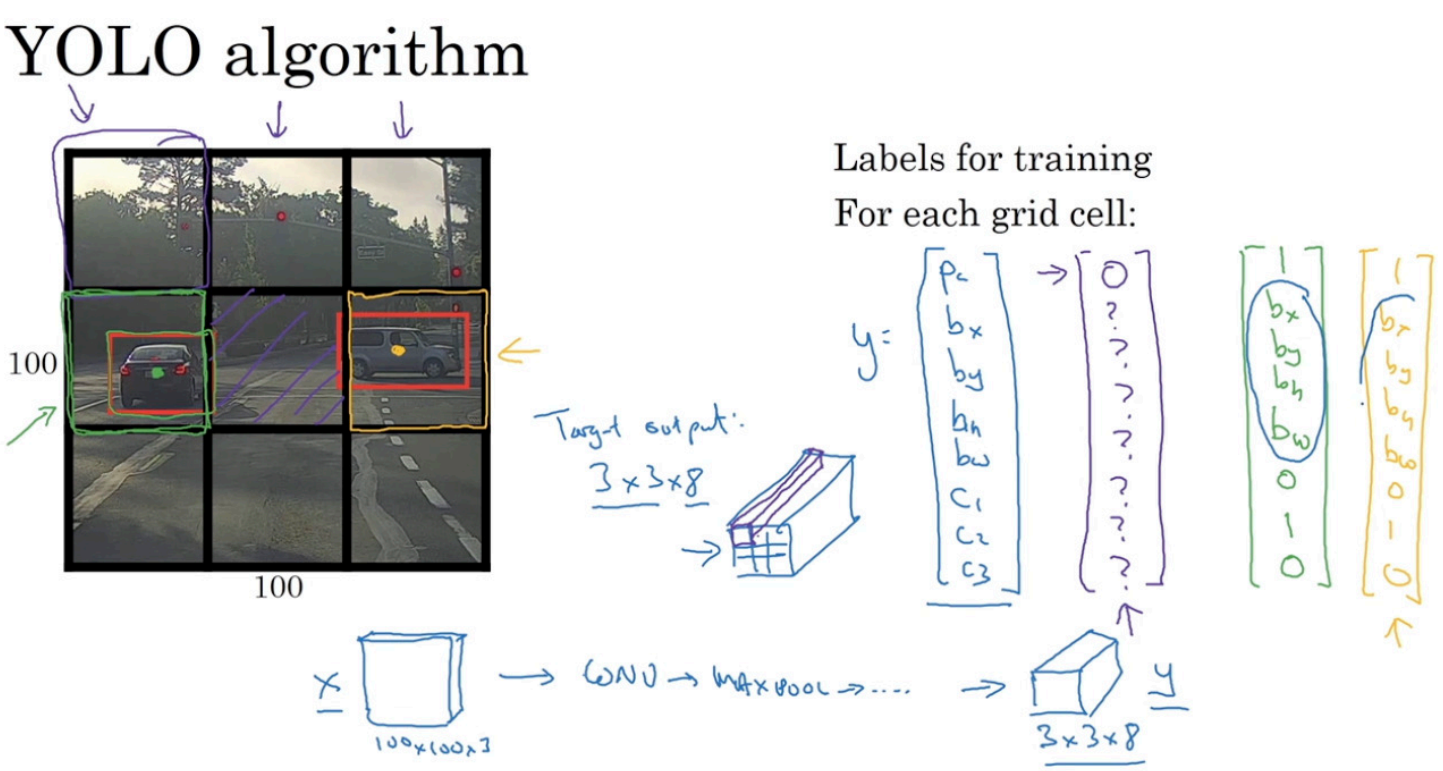
- Previously, run & slide, run & slide...
- Now instead of sequentially, make predictions on the image simultaneously, process the image at the same time.

none of the boxes may not still match perfectly to the target object.

Bounding Box Predictions

YOLO Algorithm

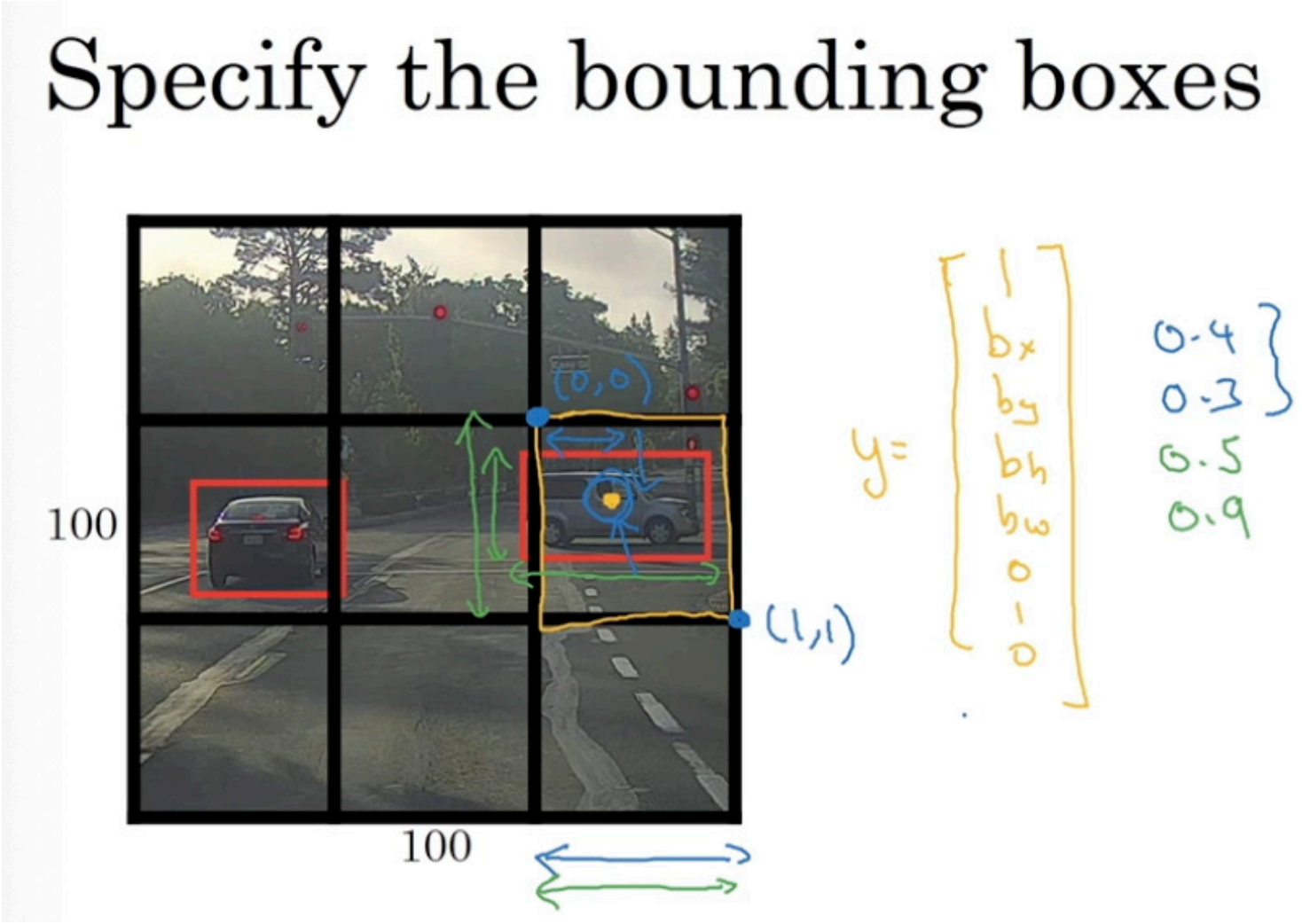
- place a grid.
- apply localization algorithm for each of the grid cell
- labels for training, pc (exists? 0/1), bx,by,bz,bh,bw,c123
- if some grid includes some parts of objects, though not their centers, pc = 0. An item is only written for single grid.
- 3x3x8: 3x3 grid, 8 features.



[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

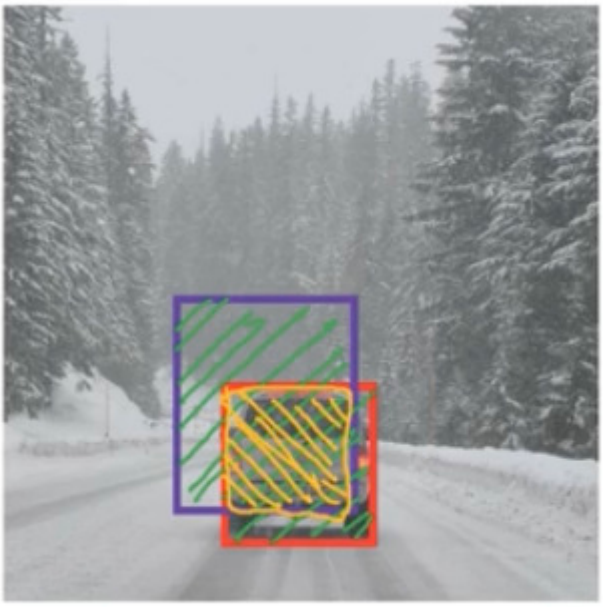
Andrew Ng

you get a model architecture that matches your grid structure (here 3x3x8)
in practice, much finer grid



location parameters must be between 0-1, size parameters can be bigger than 1 - intuition

Intersection Over Union - IOU



as the name suggests.
if => 0.5, 'correct' - general approach
comparison of sizes

Non-max Suppression

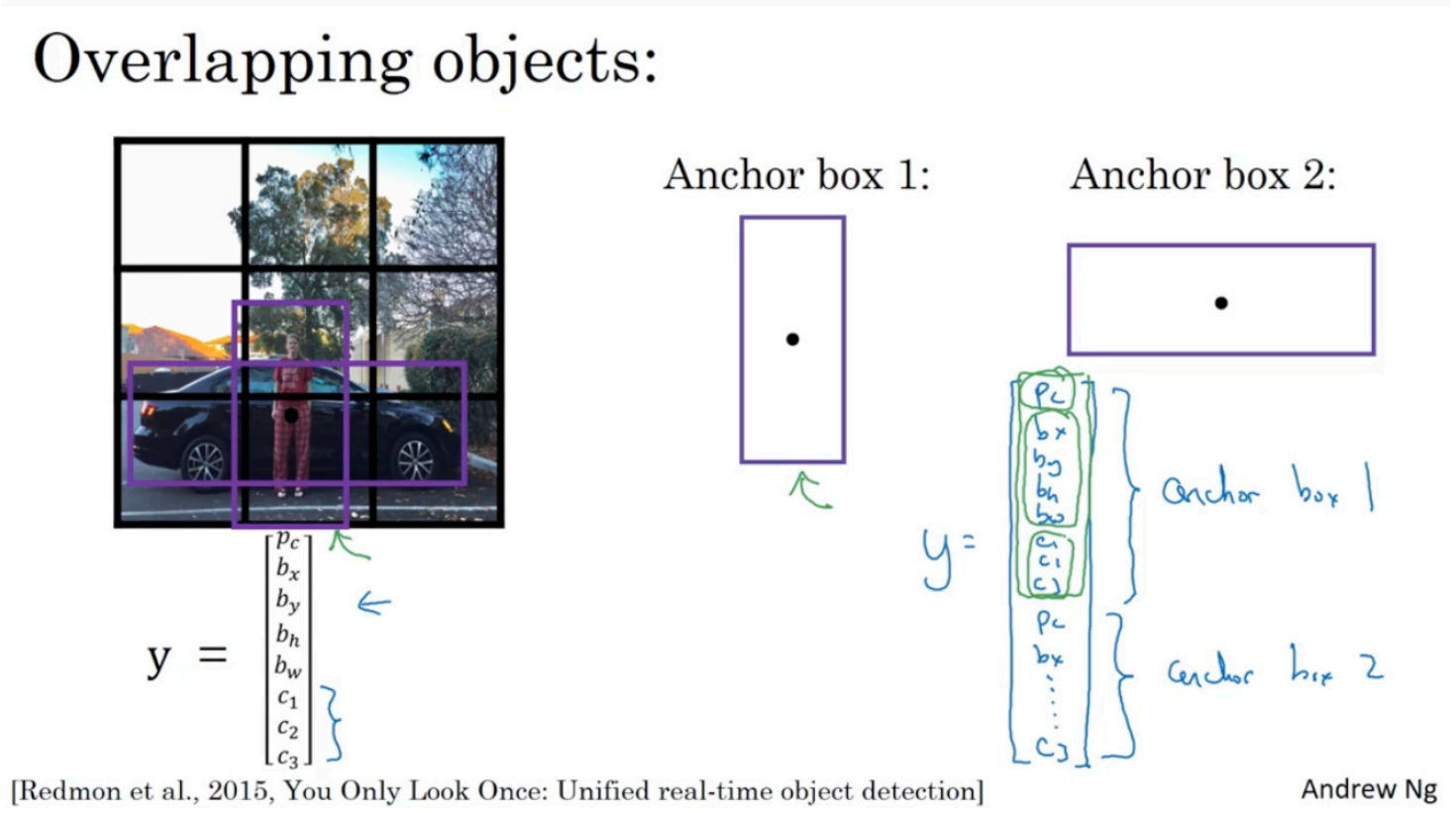
May detect each (some) object more than once.
19x19: other boxes may think they have the center, objects gets distributed to many grids.
understandable
Non-max Suppression: cleans up mirrored detections.
choose high possibility -> then clear high IOU ones

- discard all boxes pc < 0.6
- pick largest pc
- discard any remaining box with IoU > 0.5 with the box outputs in the previous step

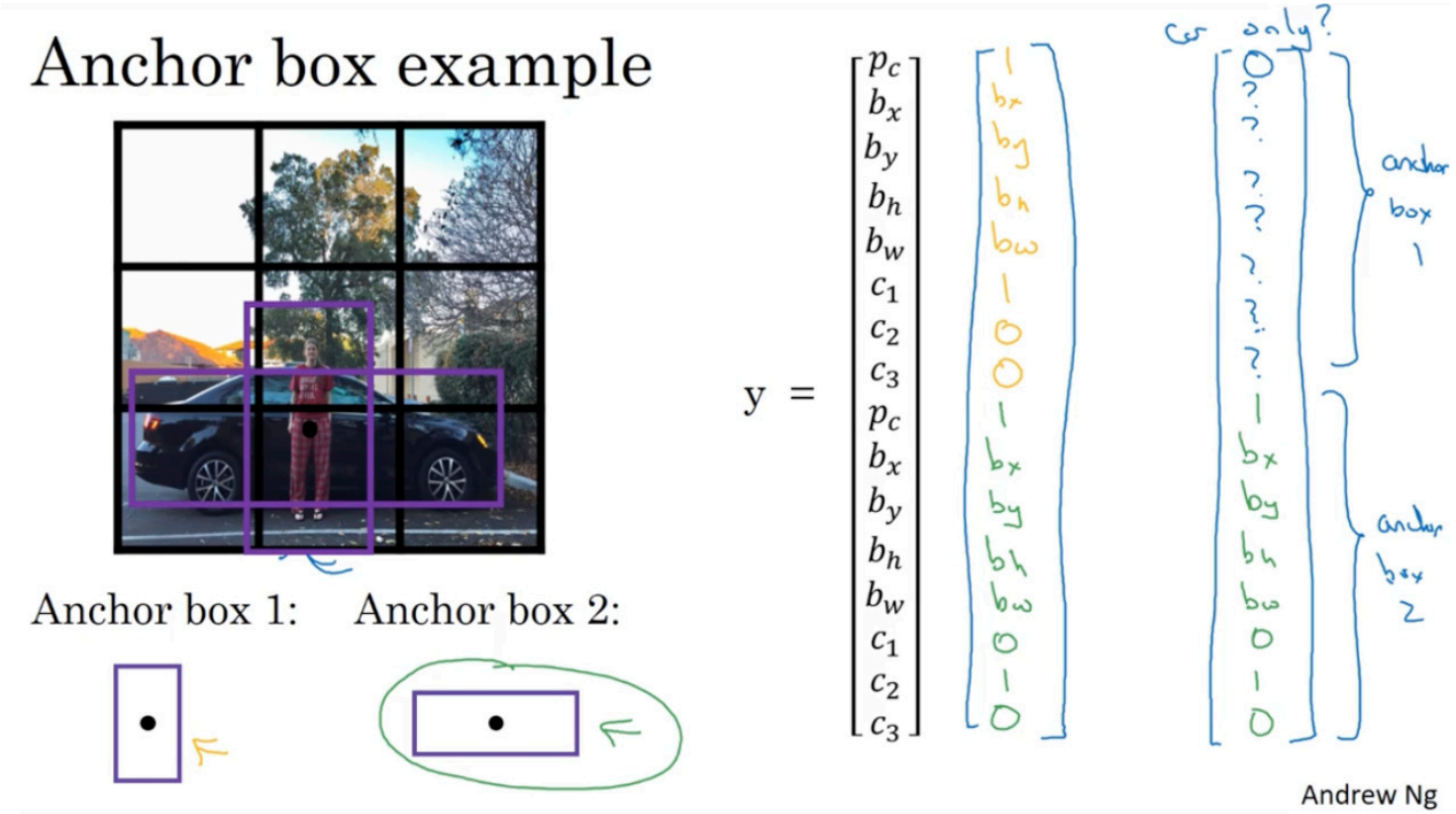
Anchor Boxes

Overlapping Objects
pre define boxes, *anchor shapes*
now you are able to associate predictions with these anchor boxes.

(assuming 2 anchor boxes) double the amount of features in *y*



- *previously: each object in training image is assigned to grid cell that contains that object's midpoint*
- *with two anchor boxes: each object is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU*

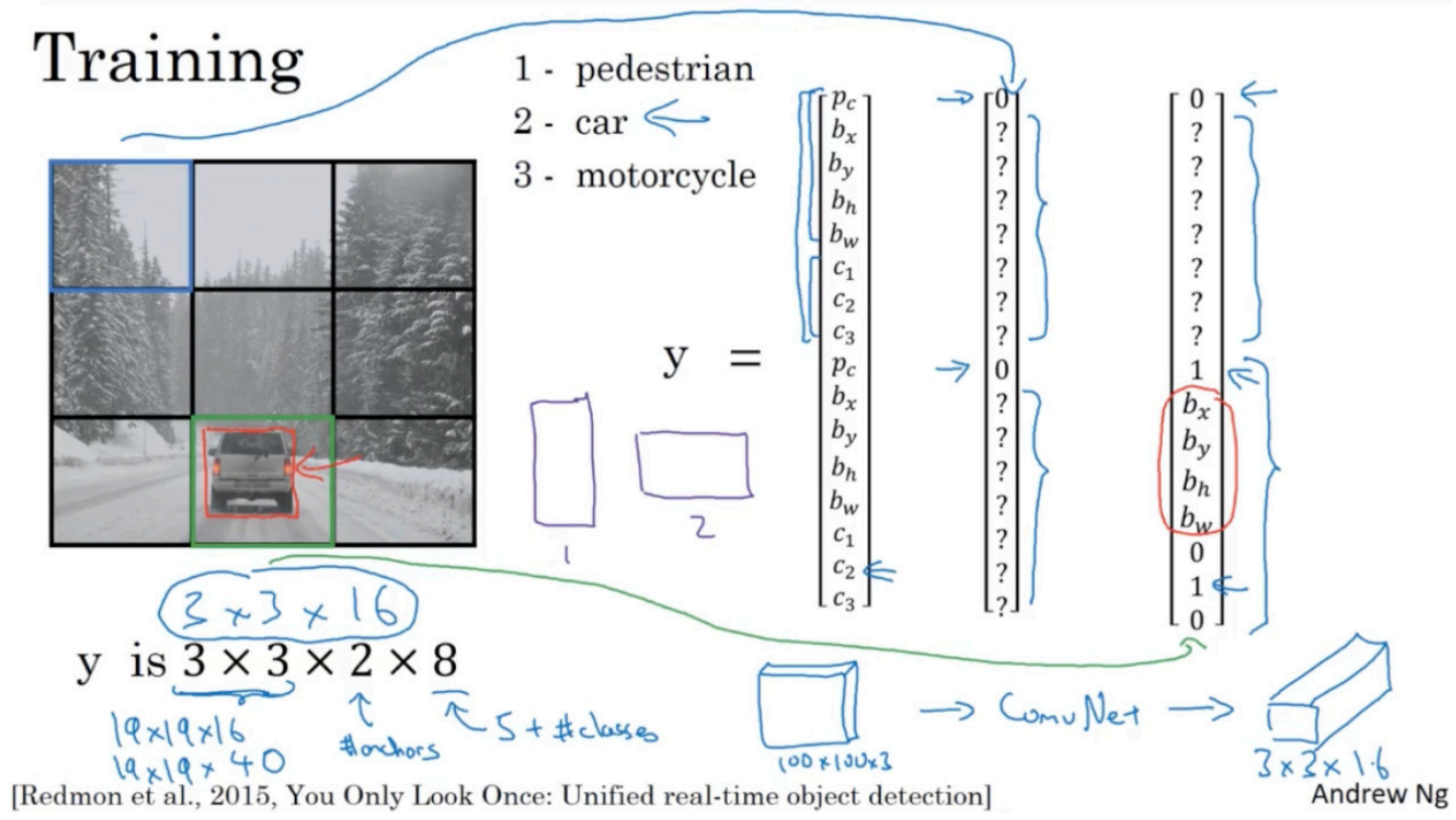


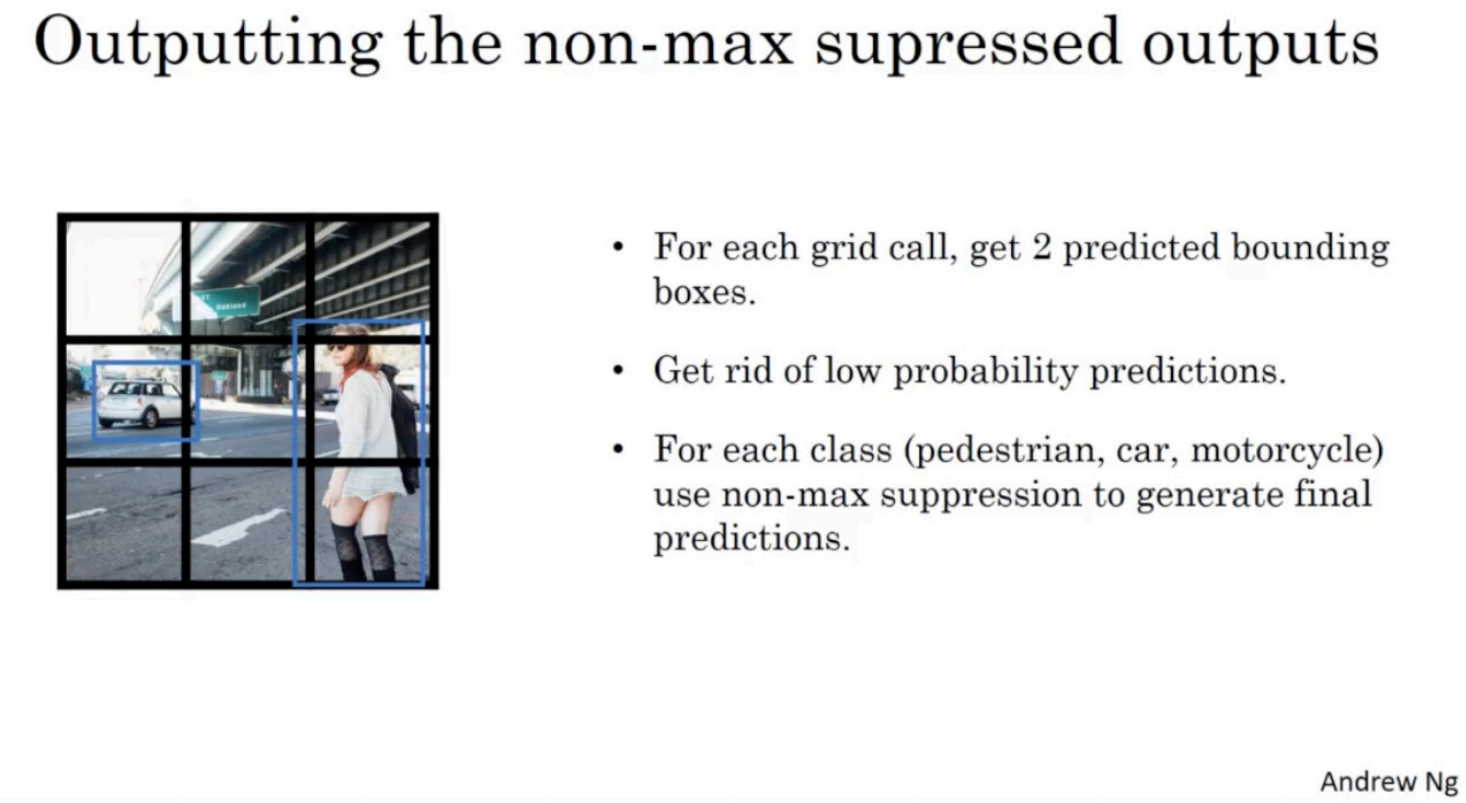
The algorithm doesn't handle these two cases well:

1. **Multiple objects in the same grid cell, exceeding the number of anchor boxes:** If a grid cell contains more objects than available anchor boxes, the algorithm struggles to represent all objects accurately. It has no way to associate more than one object with a single anchor box in a given grid cell.

2. **Multiple objects in the same grid cell with similar anchor box shapes:** Even if the number of objects is within the limit of anchor boxes, if multiple objects have shapes that best match the *same* anchor box, the algorithm has difficulty distinguishing and accurately bounding them. It might assign both objects to the same anchor box, leading to inaccurate or merged bounding boxes.

YOLO Algorithm





Region Proposals: R-CNN

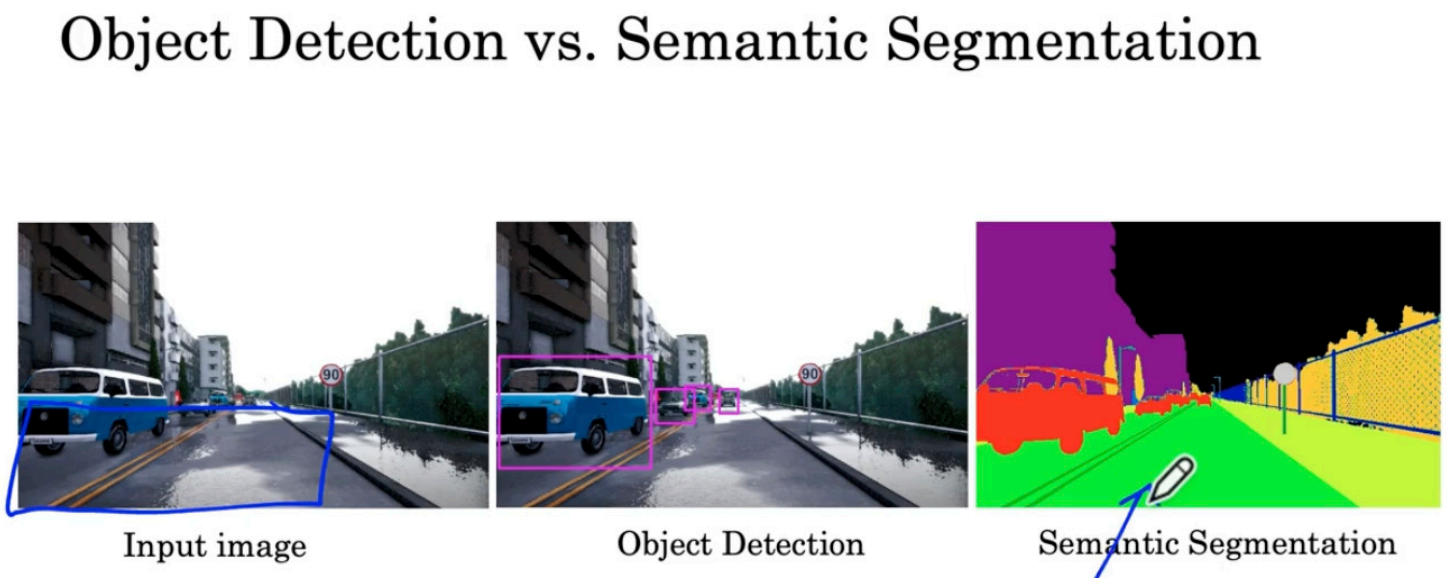
instead of running algorithm on every single window, only run on *interesting* parts.

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.

Semantic Segmentation with U-Net

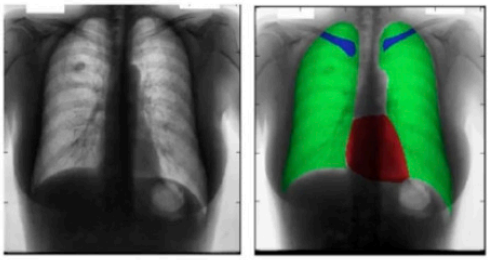
which pixels belong to the object?



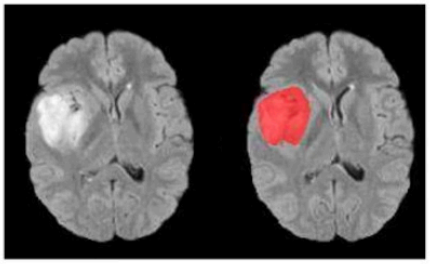
every single pixel is labeled

- used by some self driving teams

Motivation for U-Net

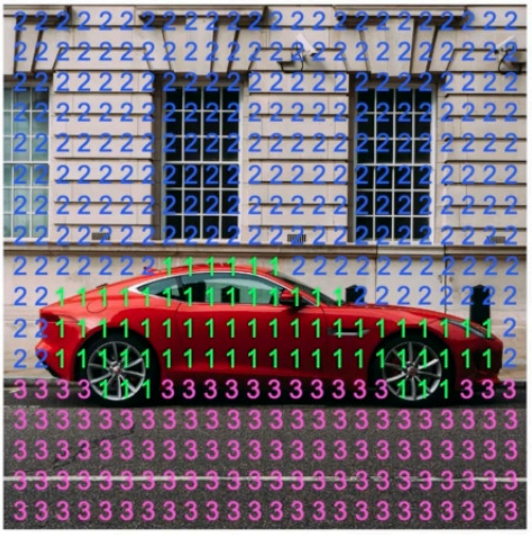


Chest X-Ray



Brain MRI

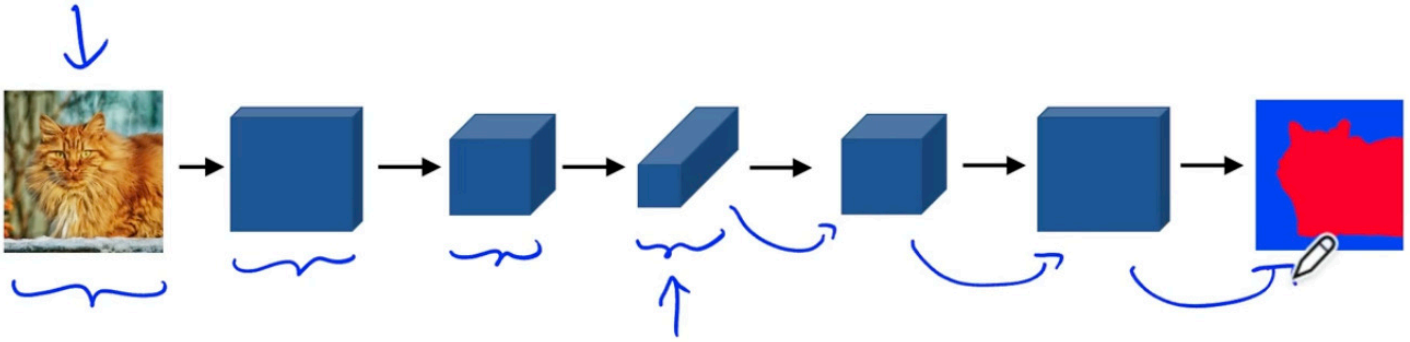
Per-pixel class labels



1. Car
2. Building
3. Road

Segmentation Map

Deep Learning for Semantic Segmentation

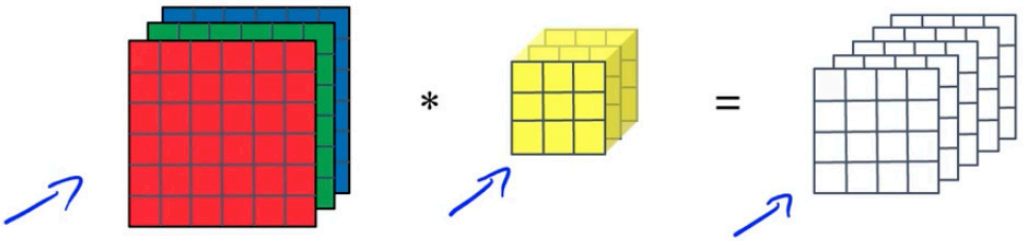


transpose convolution, need to cover all pixels, so network needs to become bigger again, less channels

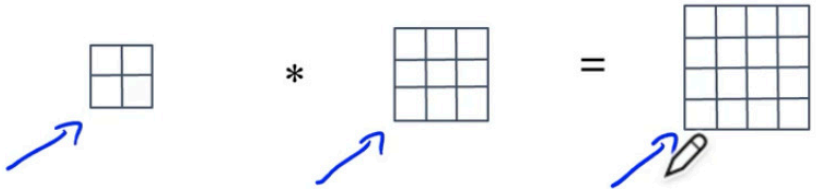
Transpose Convolutions

Transpose Convolution

Normal Convolution

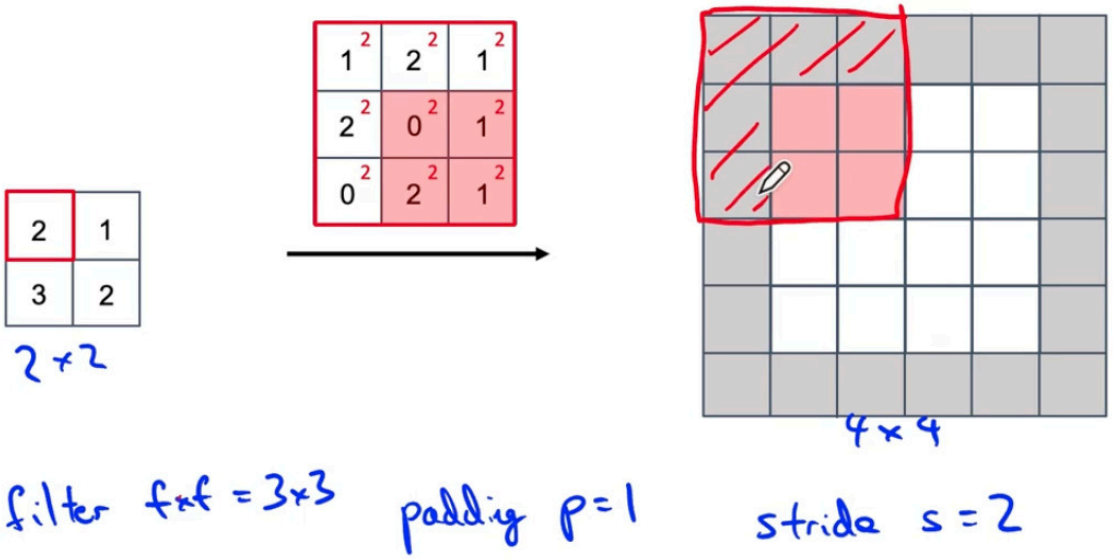


Transpose Convolution



instead filter on the input, place filter on the output

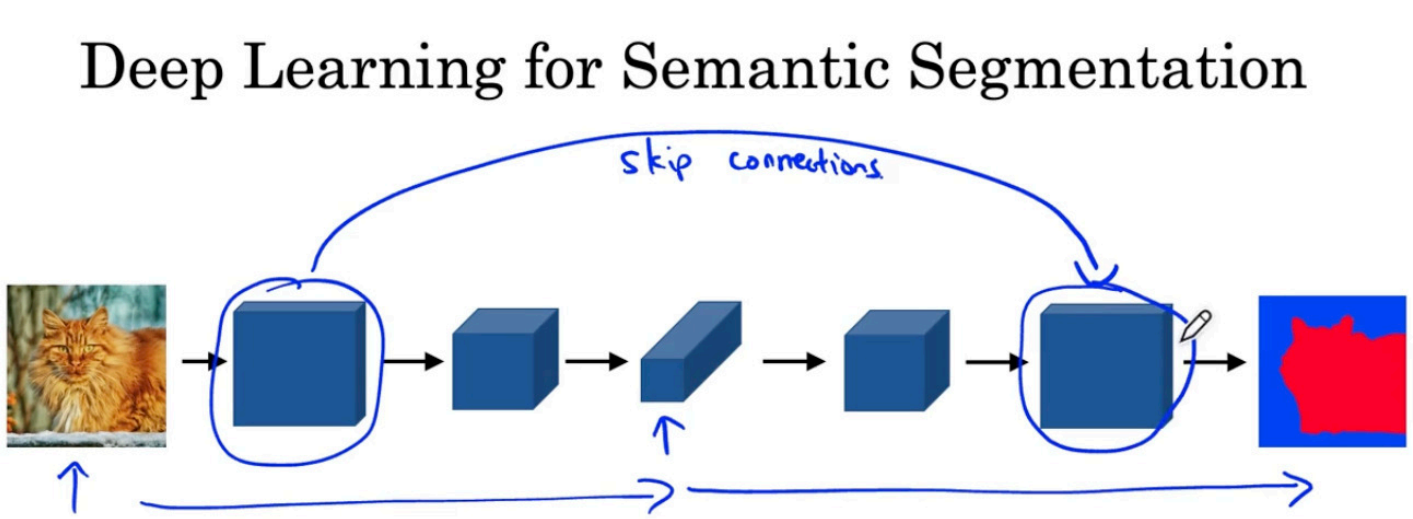
Transpose Convolution



padding area is neglected

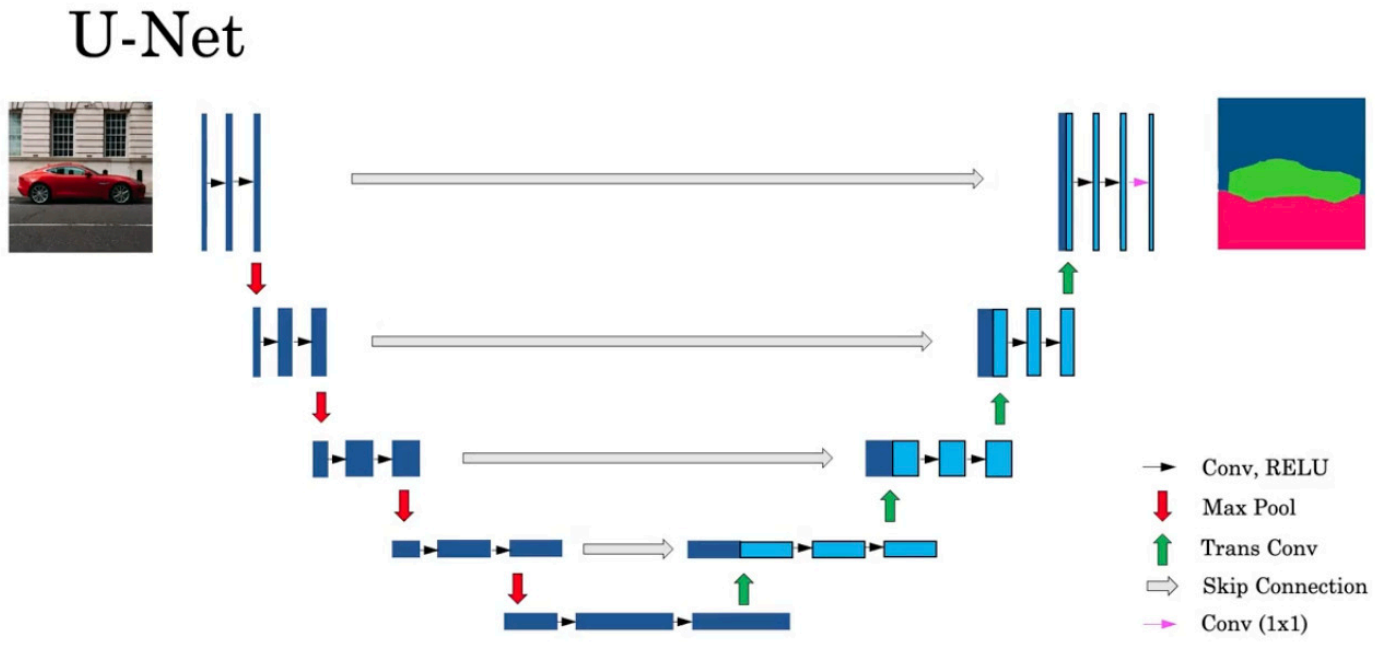
U-Net

U-Net Architecture Intuition



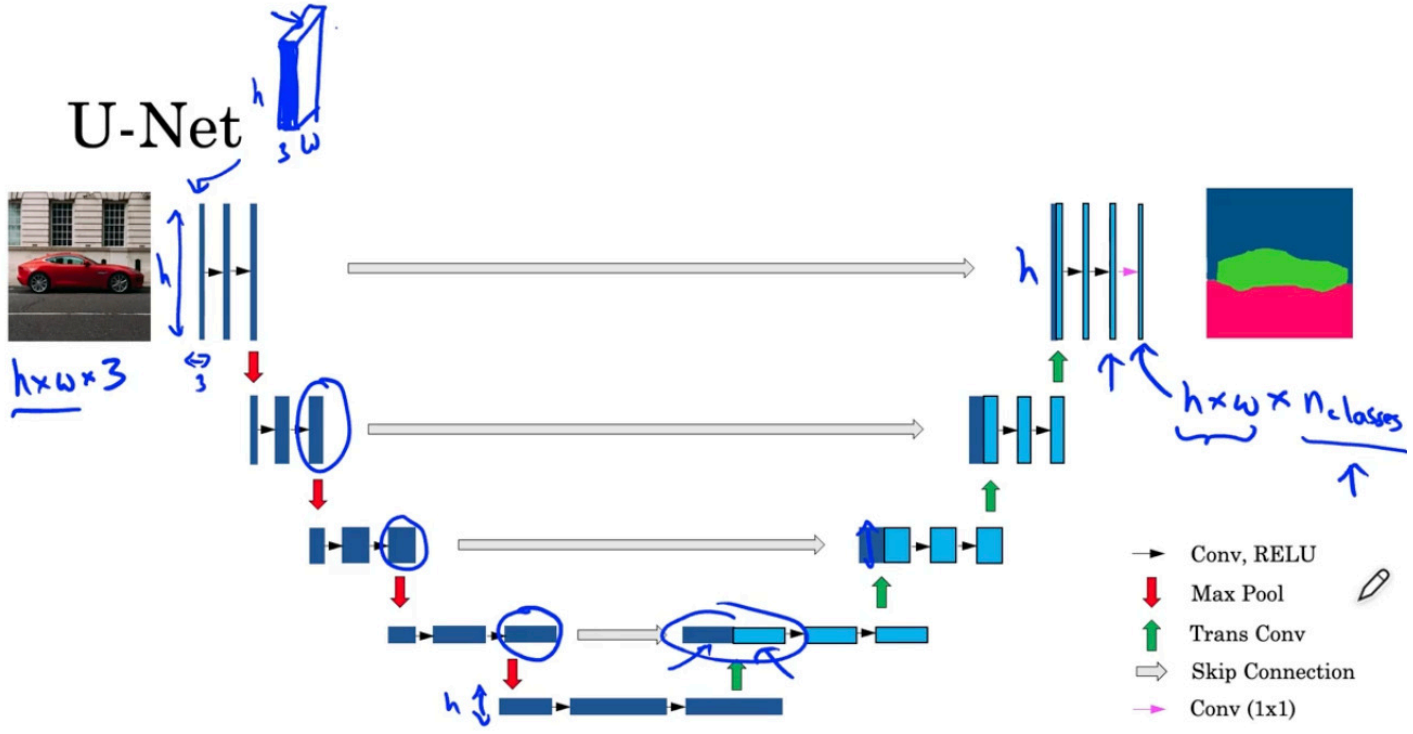
in the first part detail is lost, though more info. is stored within diff. channels, to build U-Net, skip connections are used.
allows NN to pass that detail information. so has both, high level details and high level spatial and contextual information

U-Net



u
biomedical image segmentation

- first operations, conv layer with ReLU activation fncs.
- thin images are side perspective of the image.
- at the first half, normal convolution & activation operations, with occasional max pooling layers
- trans/conv layer, second half begins. channel size decreases & added in skip connection
- copies to the right.
- and iterate



[Ronneberger et al., 2015, U-Net: Convolutional Networks for Biomedical Image Segmentation] Andrew Ng

What you should remember:

- YOLO is a state-of-the-art object detection model that is fast and accurate
- It runs an input image through a CNN, which outputs a 19x19x5x85 dimensional volume.
- The encoding can be seen as a grid where each of the 19x19 cells contains information about 5 boxes.
- You filter through all the boxes using non-max suppression. Specifically:
  - Score thresholding on the probability of detecting a class to keep only accurate (high probability) boxes
  - Intersection over Union (IoU) thresholding to eliminate overlapping boxes
- Because training a YOLO model from randomly initialized weights is non-trivial and requires a large dataset as well as lot of computation, previously trained model parameters were used in this exercise. If you wish, you can also try fine-tuning the YOLO model with your own dataset, though this would be a fairly non-trivial exercise.

What you should remember:

- Semantic image segmentation predicts a label for every single pixel in an image
- U-Net uses an equal number of convolutional blocks and transposed convolutions for downsampling and upsampling
- Skip connections are used to prevent border pixel information loss and overfitting in U-Net

Face Recognition vs Face Verification

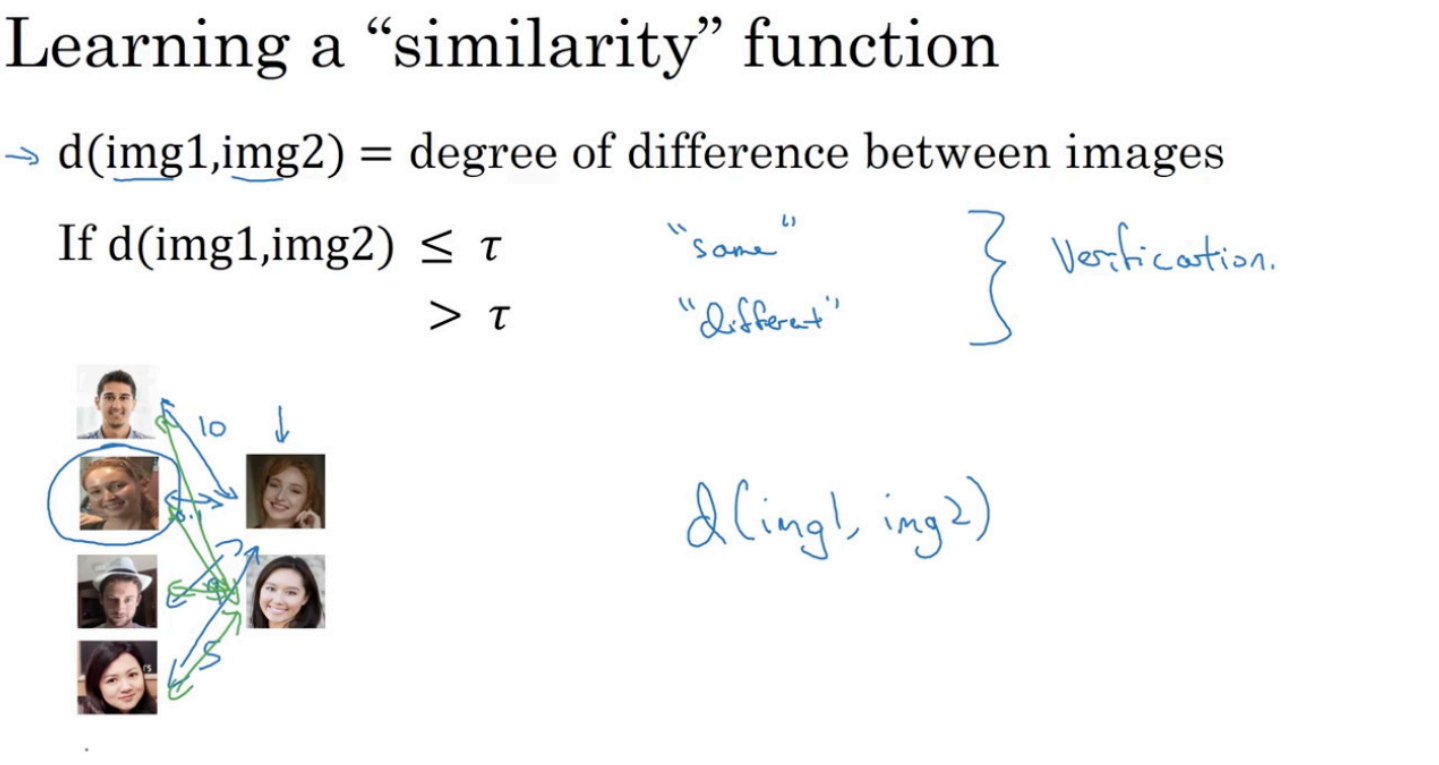
who is this person? / is this the claimed person?
maybe even %99.9 accuracy is not enough, thousands of people in database

One Shot Learning

you should resolve this problem.
you should understand the person from a single photo

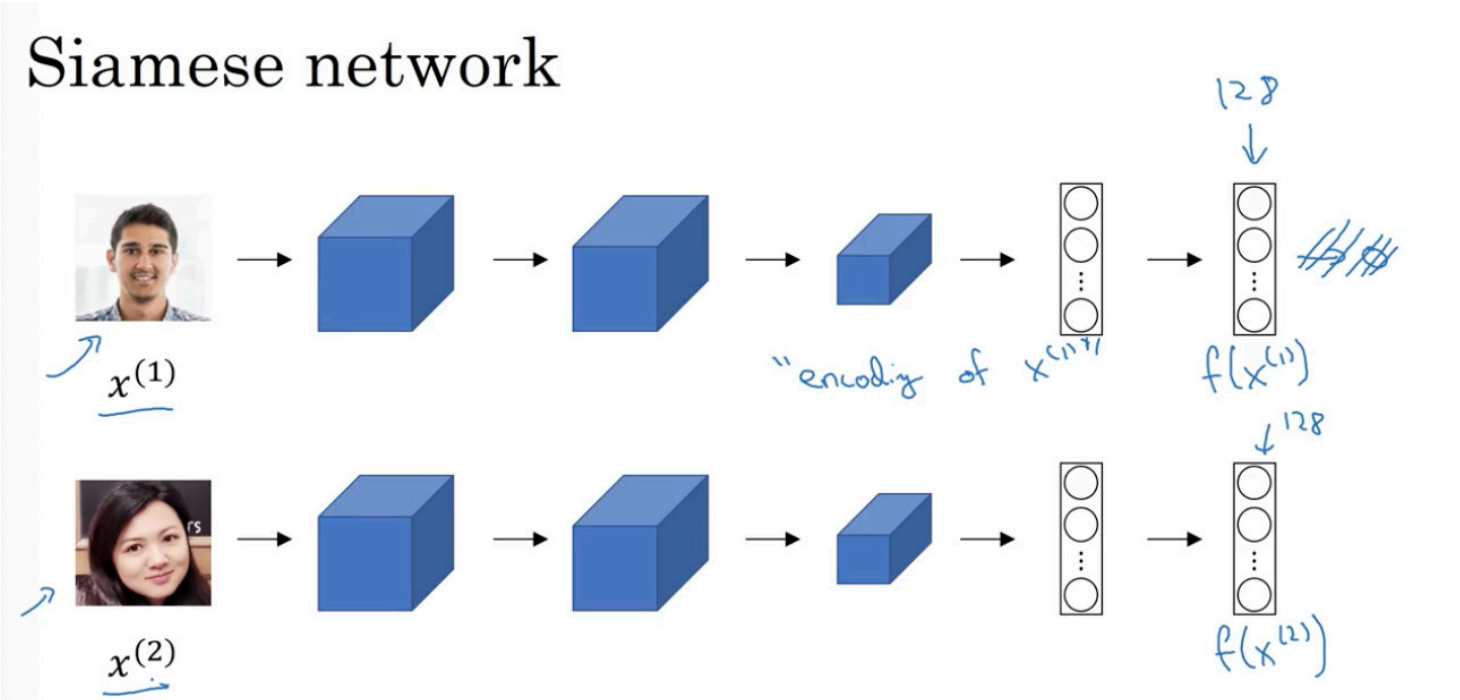
- at first 5 people at company, normal conv net, 5 unit softmax layer, someone joins in - do we need to train the whole NN?
  soln:

'Similarity' Function



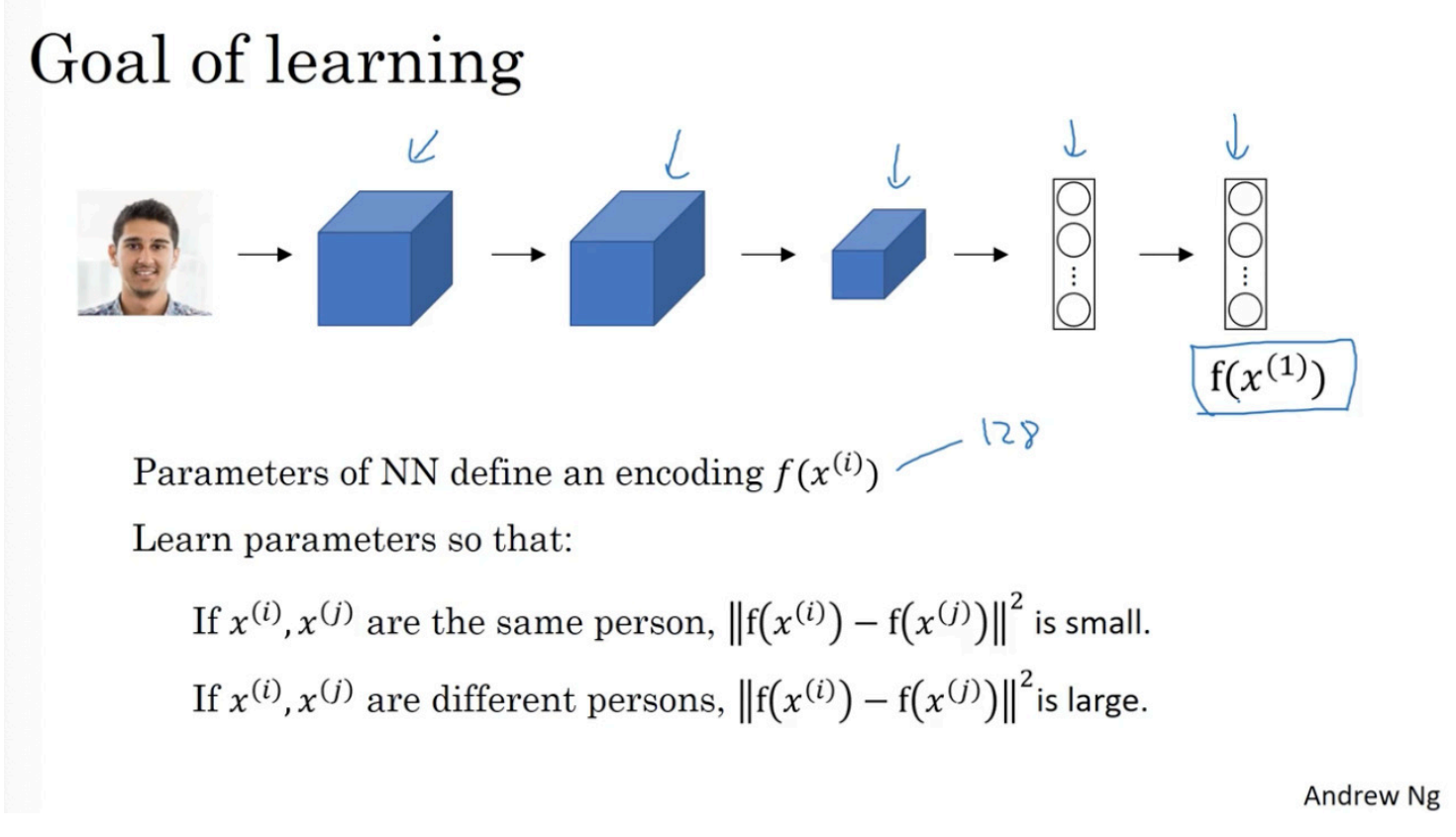
Andrew Ng

Siamese Network

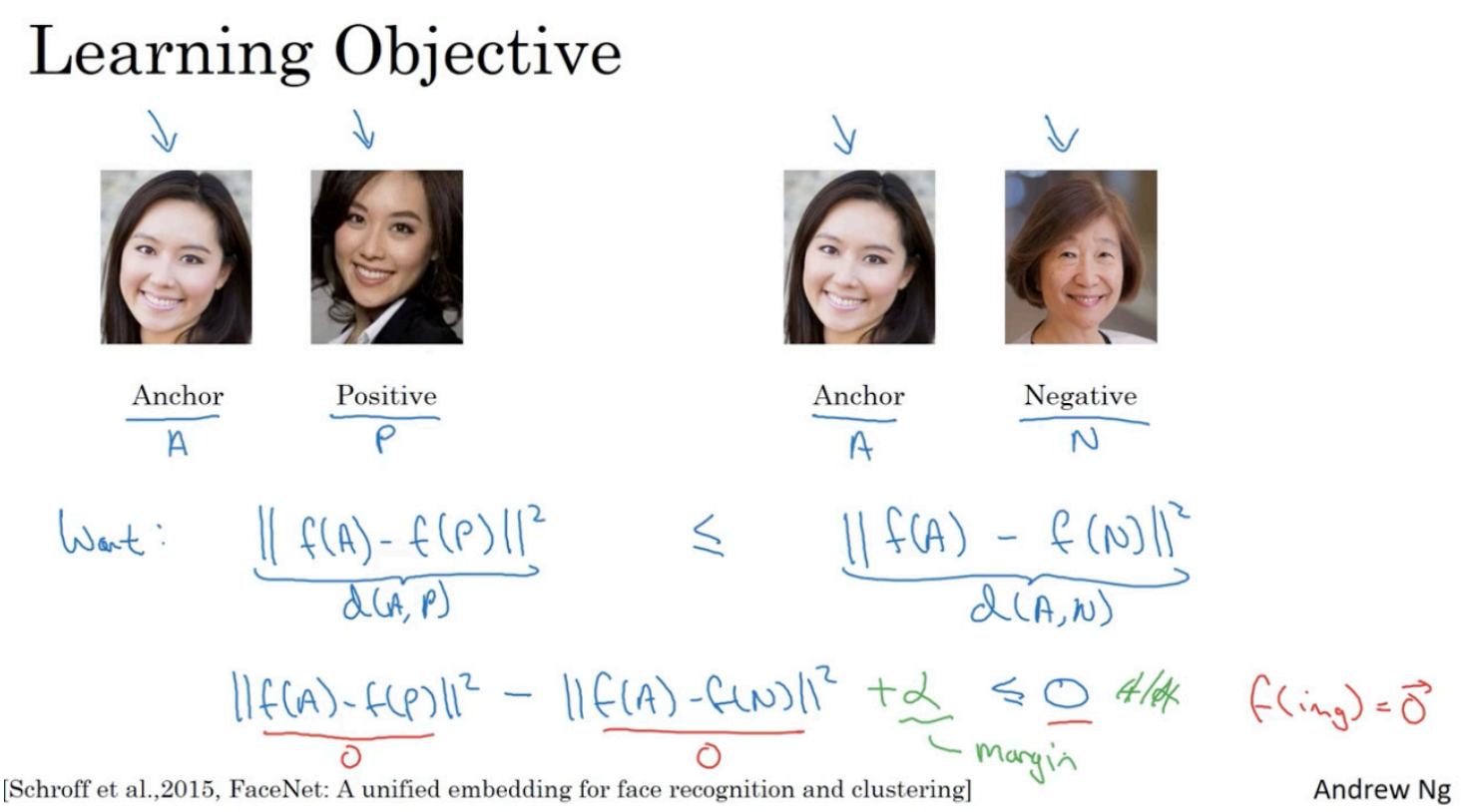


[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

Andrew Ng

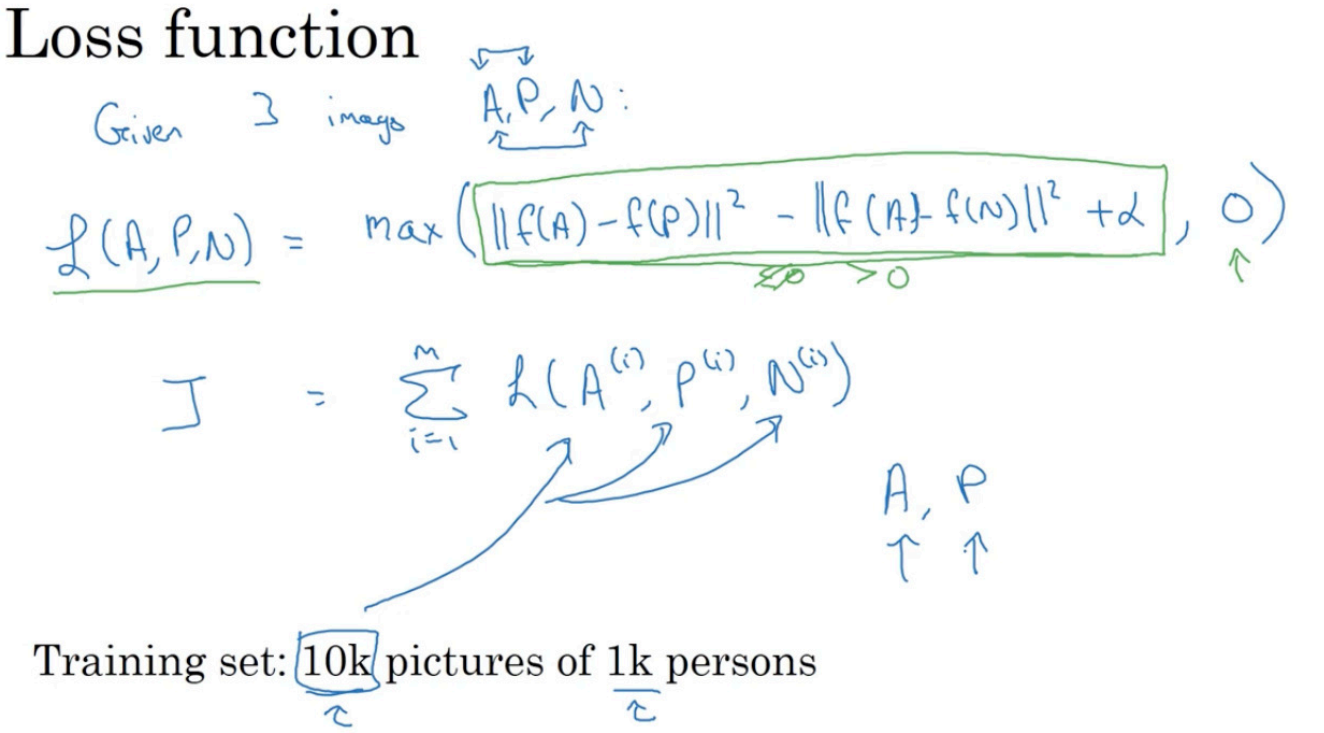


Triplet Loss



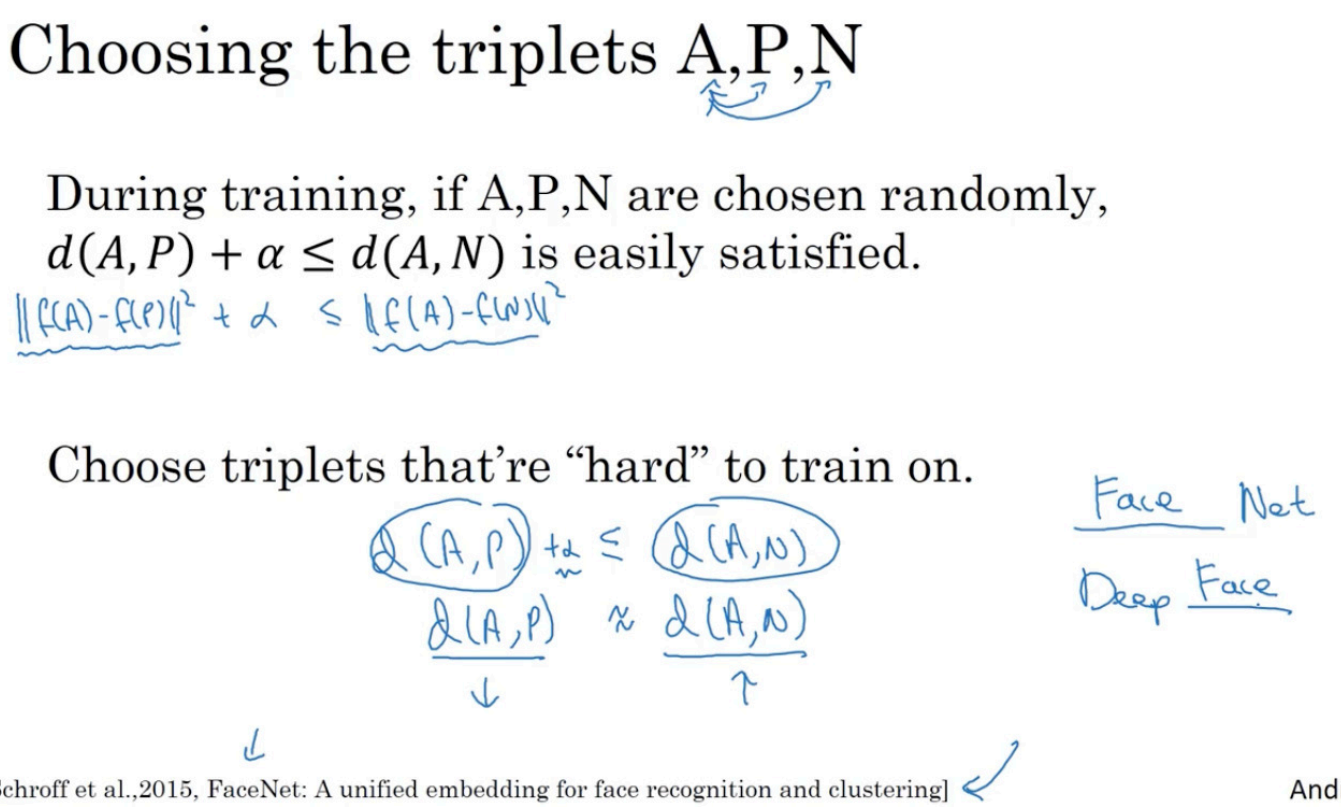
margin

Loss Function of Triplet Loss



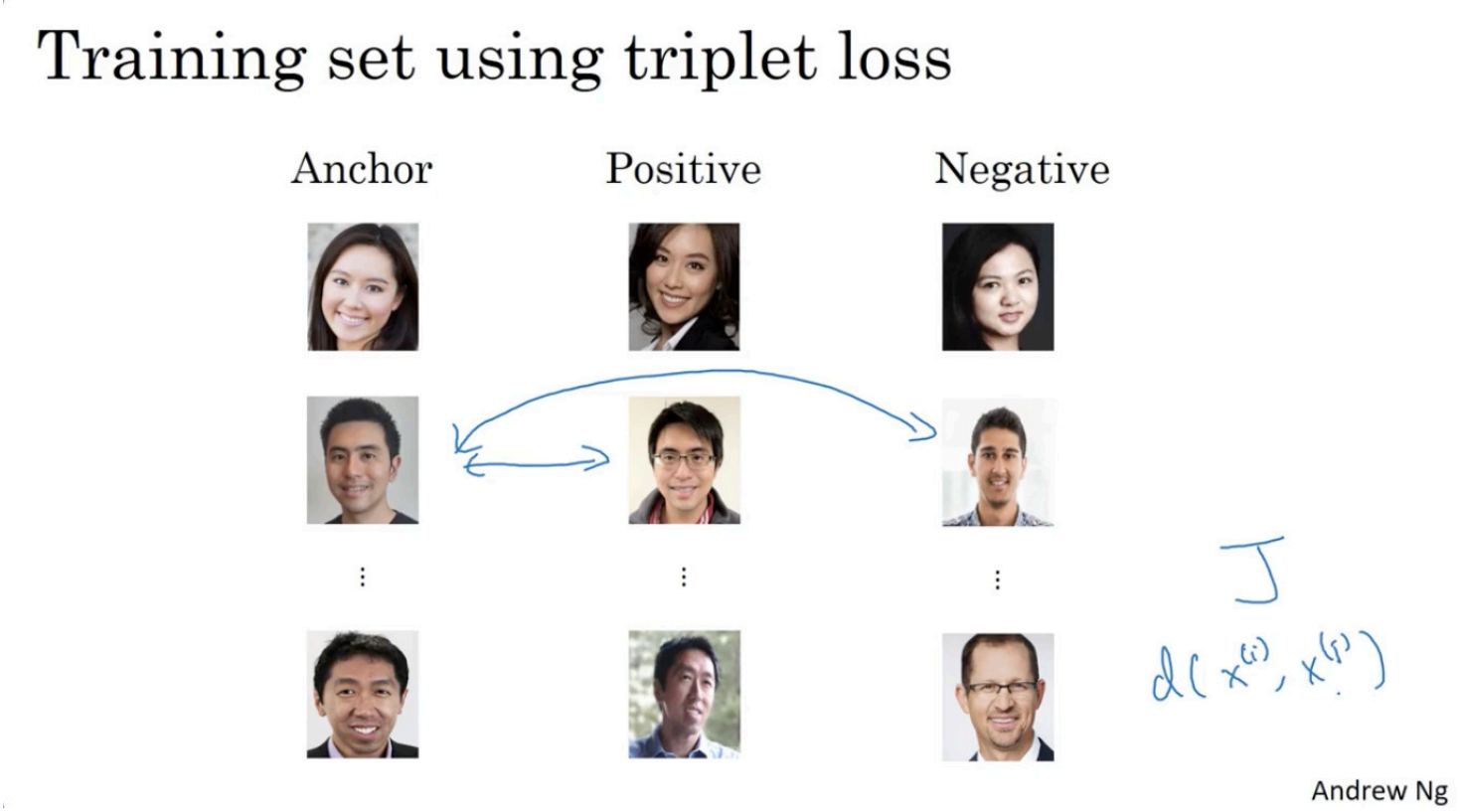
[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering] Andrew Ng

Choosing the Triplets: A, P, N

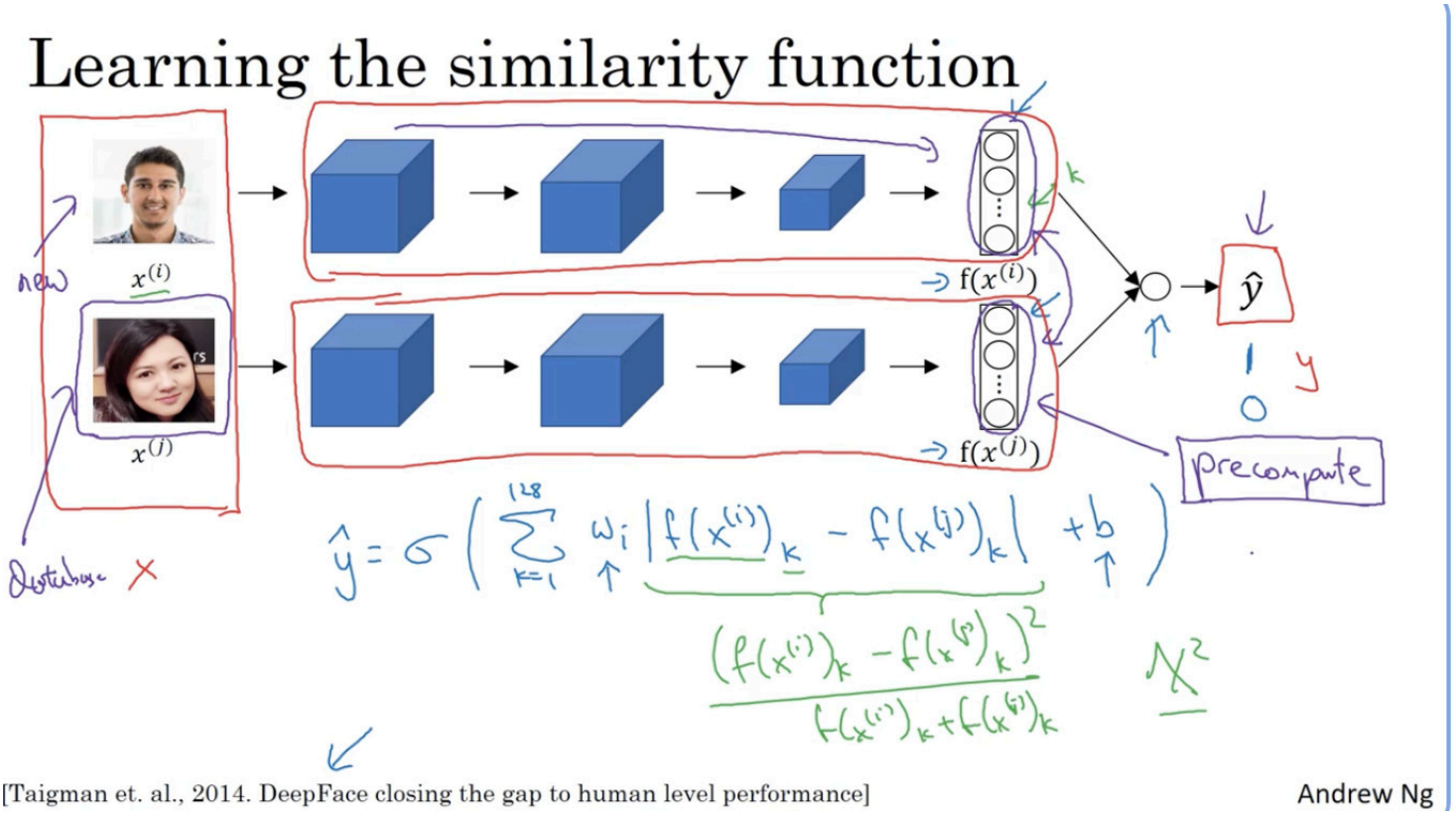


[Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering] Andrew Ng

net / deep net



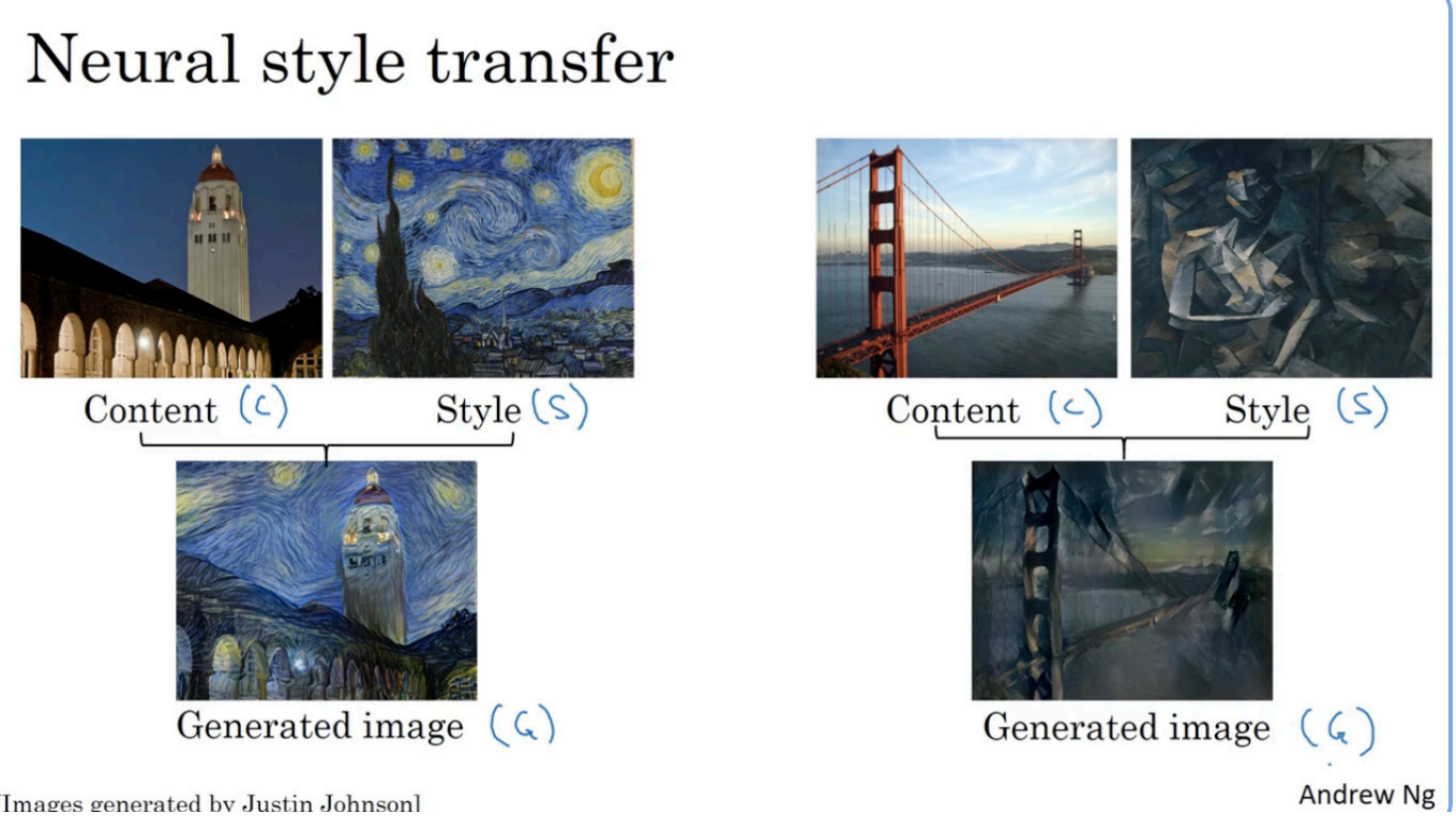
Face Verification and Binary Classification



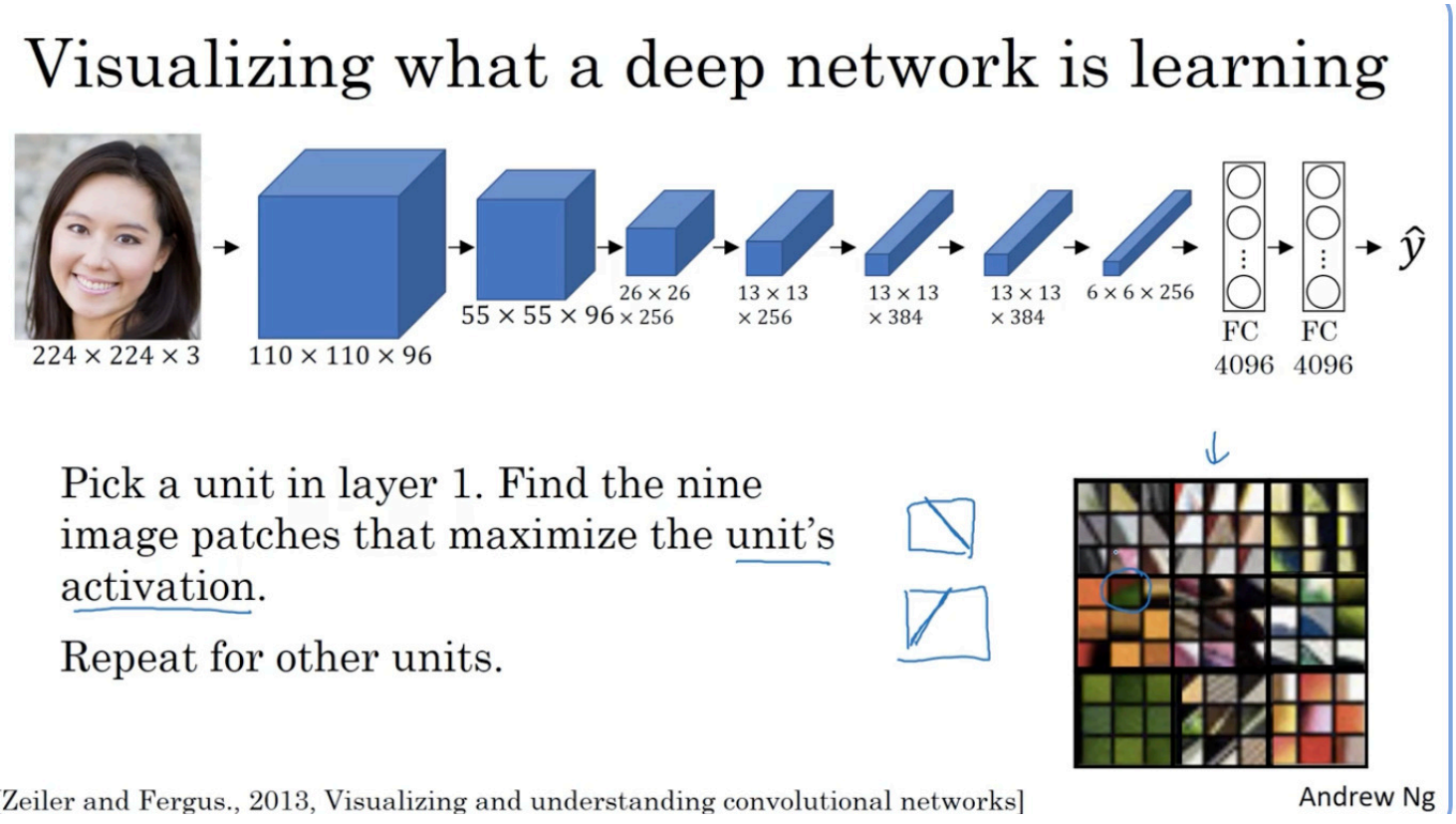
alternative to face recognition to treat it like binary classification problem

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

Neural Style Transfer



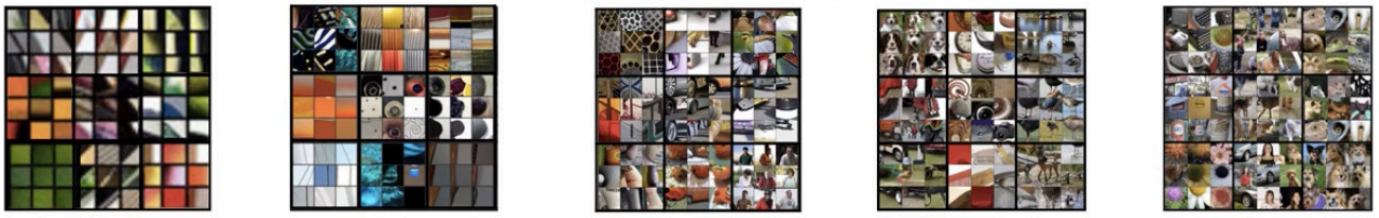
What Are Deep ConvNets Learning?

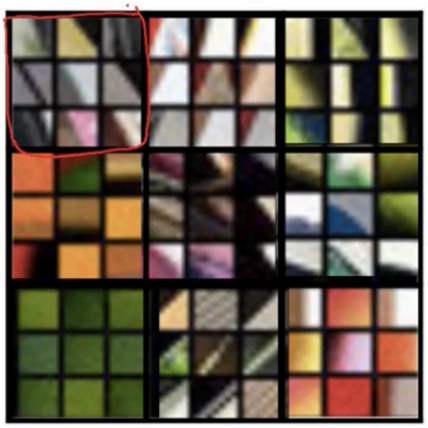


what activates some layer? when checked, some images patches are found - how(?)

- first hidden layers look for simple details - ex: edges
- patch: 9 image, that altogether passes through a layers (here again 9 hidden units)

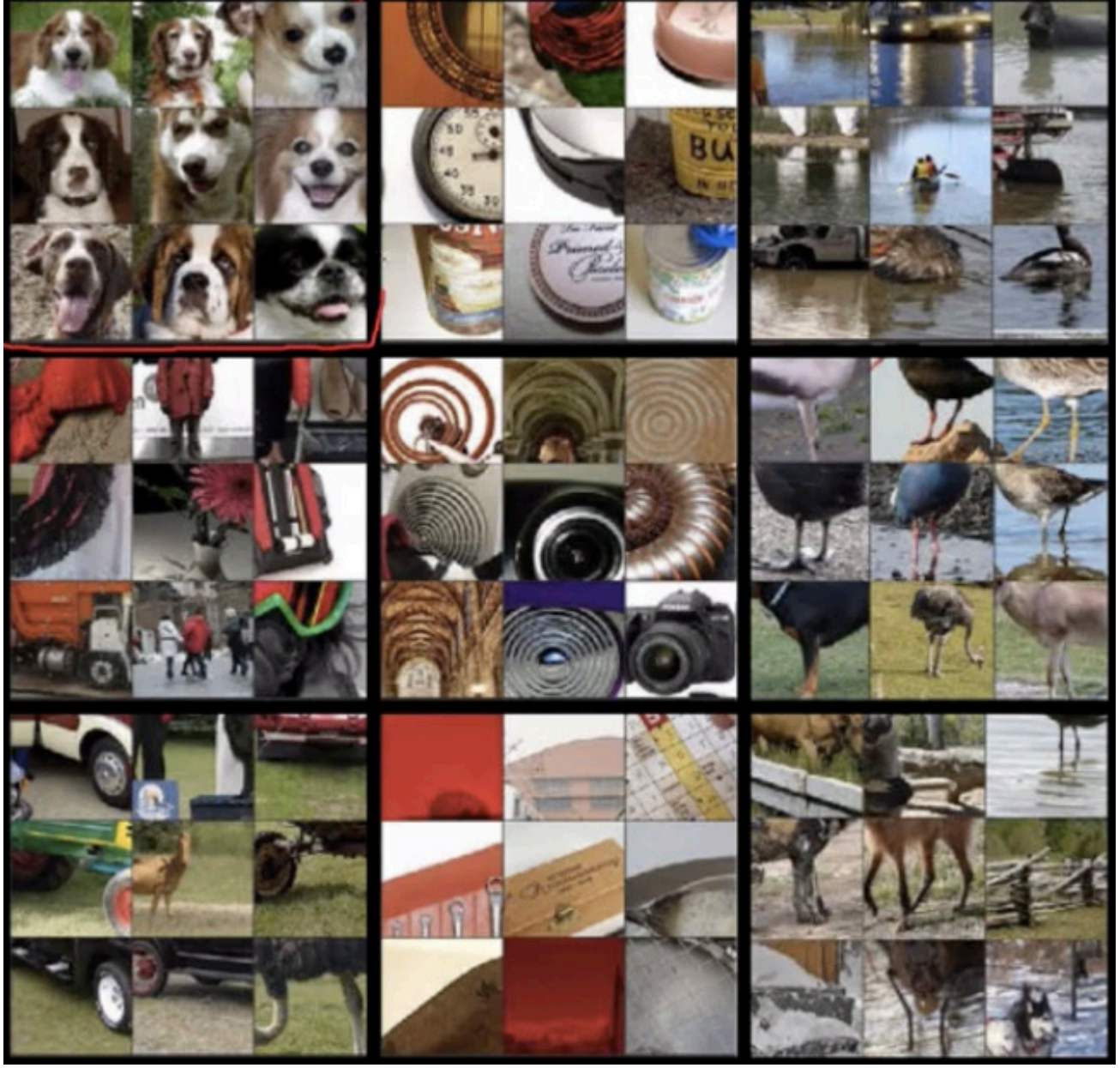
Visualizing deep layers: Layer 1





Andrew Ng

example: layer 3



example: layer 4

shows the difference of shallows vs deeper layers

Cost Function

Neural style transfer cost function

Content C

Style S

Generated image G

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

[Gatys et al., 2015. A neural algorithm of artistic style. Images on slide generated by Justin Johnson] Andrew Ng

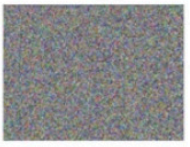
Find the generated image G

1. Initiate G randomly

   G: $100 \times 100 \times 3$ (RGB)

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

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



[Gatys et al., 2015. A neural algorithm of artistic style] Andrew Ng

Content Cost Function

Content cost function

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

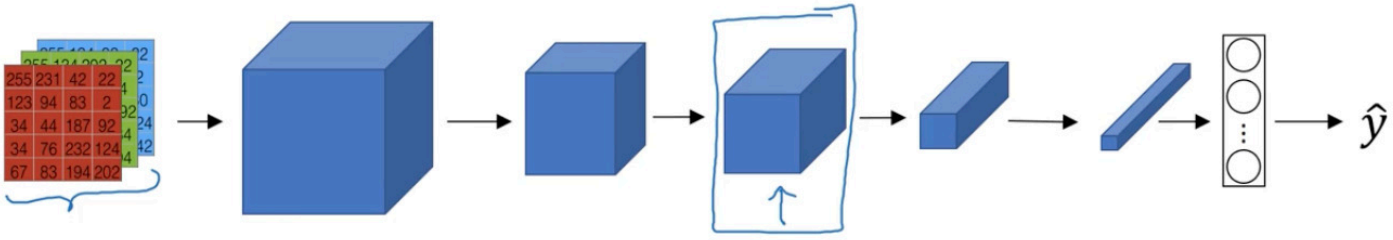
- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

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

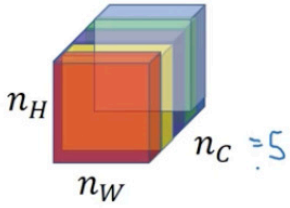
[Gatys et al., 2015. A neural algorithm of artistic style]

Style Cost Function

Meaning of the "style" of an image



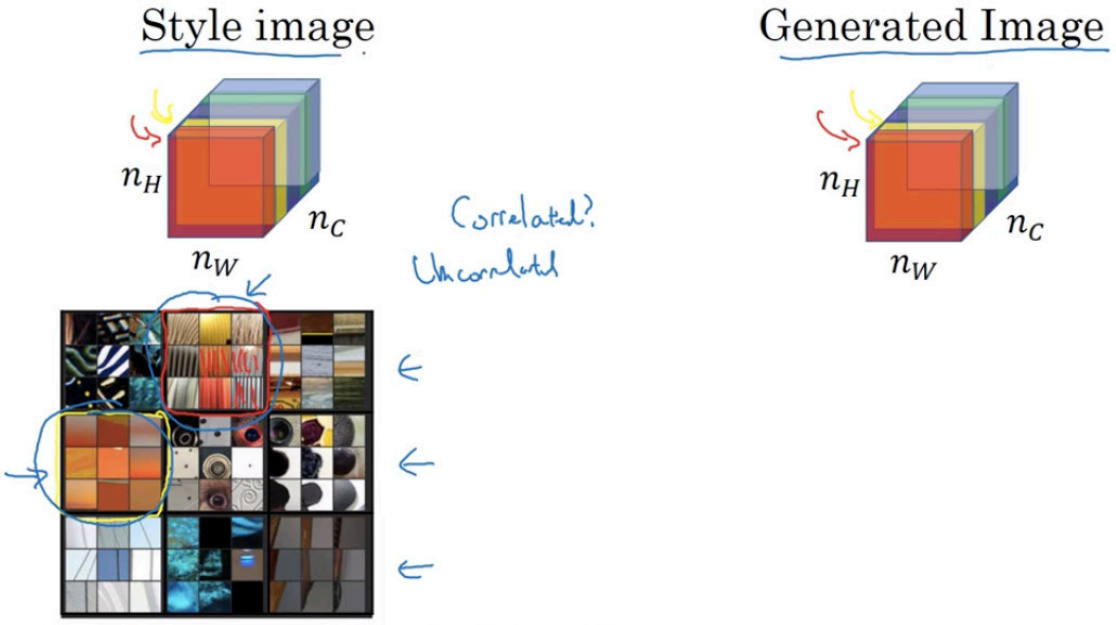
Say you are using layer l's activation to measure "style."
Define style as correlation between activations across channels.



How correlated are the activations across different channels?

[Gatys et al., 2015. A neural algorithm of artistic style]

Intuition about style of an image



[Gatys et al., 2015. A neural algorithm of artistic style]

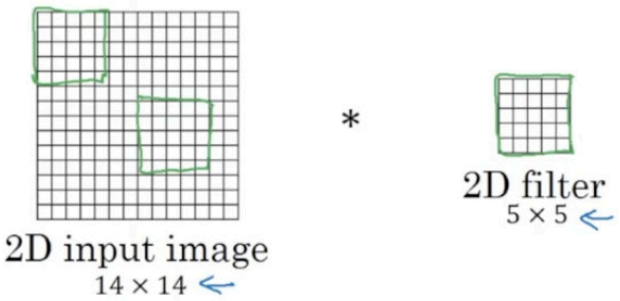
Andrew Ng

$$J^{[l]}_{style}(S,G) = \frac{1}{(2n^{[l]}_H n^{[l]}_W n^{[l]}_C)^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2$$

1D and 3D Generalization

rules on 2d (images) also generally apply to both 1D and 3D.

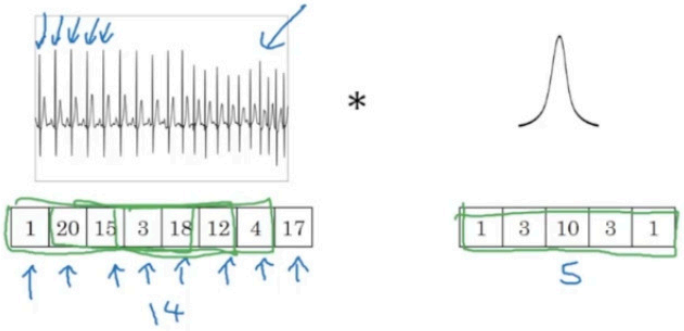
Convolutions in 2D and 1D



$14 \times 14 \times 3 \quad * \quad 5 \times 5 \times 3$

$\rightarrow 10 \times 10 \times 16$

$10 \times 10 \times 16 \quad *$



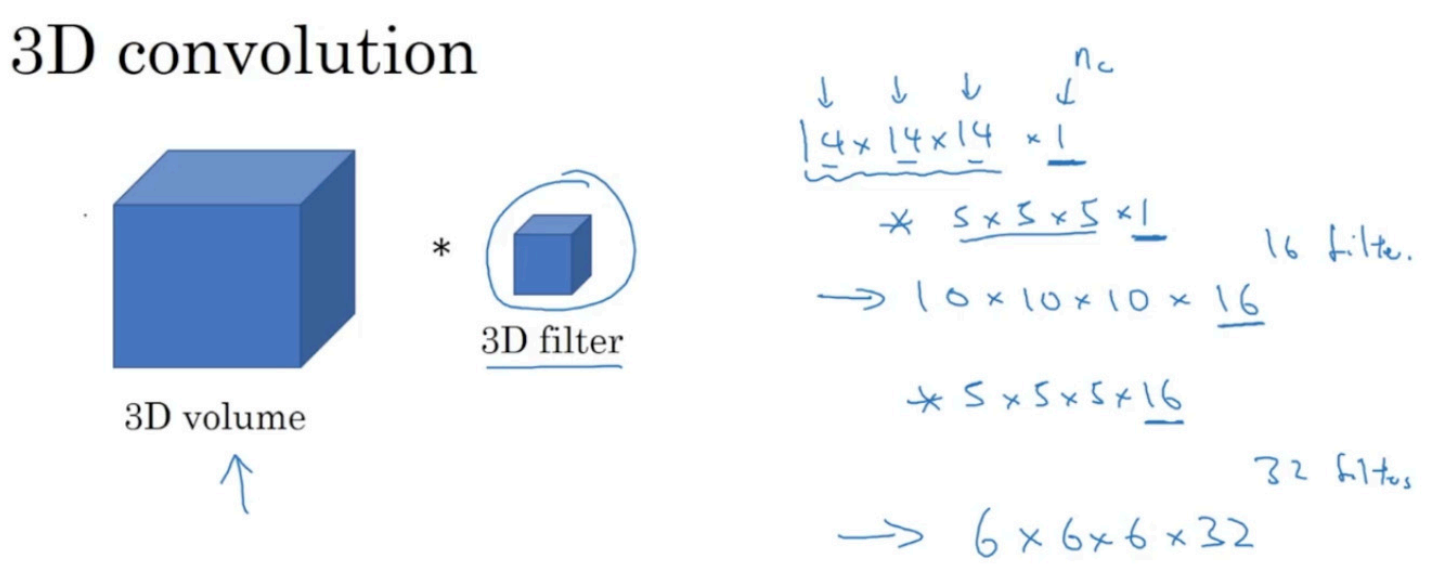
$14 \times 1 \quad * \quad 5 \times 1$

$\rightarrow 10 \times 16$

$10 \times 16 \quad * \quad 5 \times 16$

$\rightarrow 6 * 32$

Andrew Ng



may also imagine this as a movie data, each channel being a different frame of the movie/video.

Triplet Loss (Short Note)

- **Purpose:** Learns embeddings where similar items are close, and dissimilar items are far apart. Used in face recognition, image retrieval, etc.
- **Components:**
  - **A (Anchor):** Baseline input.
  - **P (Positive):** Input *similar* to A.
  - **N (Negative):** Input *dissimilar* to A.
- **Goal:** Make distance(A, P) small, and distance(A, N) large.
- **Formula:** `Loss = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + α, 0)`
  - `f(x)` : Embedding function.
  - `α` : Margin (enforces separation).
  - || ||^2 is the euclidian distance.
- **High Loss:** When A is closer to N than to P (bad!). The `||f(A) - f(P)||^2 - ||f(A) - f(N)||^2` part of the equation becomes positive.
- **Low Loss:** When A is closer to P than to N (by at least the margin α) (good!). The `||f(A) - f(P)||^2 - ||f(A) - f(N)||^2` part of the equation becomes negative.
- **Loss is larger when:** The encoding of A is closer to the encoding of N than P.

What you should remember:

- Face verification solves an easier 1:1 matching problem; face recognition addresses a harder 1:K matching problem.
- Triplet loss is an effective loss function for training a neural network to learn an encoding of a face image.
- The same encoding can be used for verification and recognition. Measuring distances between two images' encodings allows you to determine whether they are pictures of the same person.

- The content cost takes a hidden layer activation of the neural network, and measures how different $a(C)$ and $a(G)$ are.
- When you minimize the content cost later, this will help make sure G has similar content as C.

- The style of an image can be represented using the Gram matrix of a hidden layer's activations.
- You get even better results by combining this representation from multiple different layers.
- This is in contrast to the content representation, where usually using just a single hidden layer is sufficient.
- Minimizing the style cost will cause the image G to follow the style of the image S.

- The total cost is a linear combination of the content cost $Jcontent(C,G)$ and the style cost $Jstyle(S,G)$.
- α and β are hyperparameters that control the relative weighting between content and style.

- Neural Style Transfer is an algorithm that given a content image C and a style image S can generate an artistic image
- It uses representations (hidden layer activations) based on a pretrained ConvNet.
- The content cost function is computed using one hidden layer's activations.
- The style cost function for one layer is computed using the Gram matrix of that layer's activations. The overall style cost function is obtained using several hidden layers.
- Optimizing the total cost function results in synthesizing new images.

---

1. I ↩
2. I ↩
3. I ↩
4. I ↩
5. I+2 ↩
6. I+2 ↩
7. I ↩
8. I+2 ↩
9. I ↩
10. I ↩
11. I+2 ↩
12. I+2 ↩
13. I+2 ↩
14. I+2 ↩
15. I ↩
16. I ↩
17. I ↩
18. I ↩

19. l ↩
20. l ↩
21. l+2 ↩
22. l ↩
23. l+2 ↩