Coursera Deep Learning Specialization Notes: Convolutional Neural Networks

Annotated by Mohamad Zbib

Version 1.0, November 2022

Contents

Preface

A couple of years ago I completed Deep Learning Specialization taught by AI pioneer Andrew Ng. I found this series of courses immensely helpful in my learning journey of deep learning. After years, I decided to prepare this document to share some of the notes which highlight key concepts I learned in the fourth course of this specialization, Convolutional Neural Networks (CNNs). This course teaches how to build convolutional neural networks and apply them to image data. When it comes to computer vision, CNNs are the bee knees. CNNs are what have given rise to incredible improvements in facial recognition, classifying X-ray reports, and self-driving car systems. Notes are based on lecture videos and supplementary material provided and my own understanding of the topics.

The content of this document is mainly adapted from this GitHub repository. I have added some explanations, illustrations, and visualization to make some complex concepts easier to grasp for readers. This document could be a good reference for Machine Learning Engineers, Deep Learning Engineers, and Data Scientists to refresh their minds on the fundamentals of CNNs. I'm going to prepare and share notes for other courses in this specialization in the near future. Please don't hesitate to contact me via my website ( ) if you have any questions.

Happy Learning!

Annotated by Mohamad Zbib

1 Convolutional Neural Networks

1.1 Fundamentals

Convolution: Applying a **kernel** or **filter** to a matrix. Denoted by operator $*$.

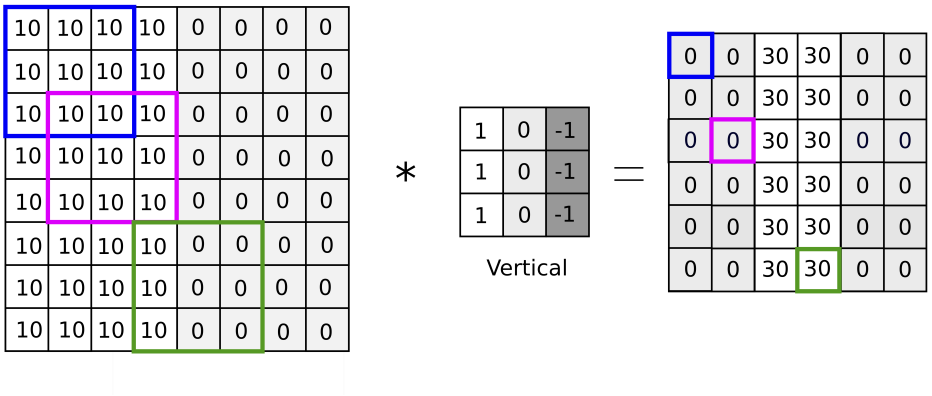

Figure 1: Vertical Edge Detection Example.

- Typical example is edge detection in images (e.g. the Sobel or Scharr algorithms/filters)

Figure 2: Filters.

- Size of the result of convolving an image of size $n \times n$ with a filter of size $f \times f$ is $(n - f + 1) \times (n - f + 1)$.
- **Padding**: adding an extra border of pixels set to 0 to avoid losing information. If $p = 1$ corresponding to a 1 pixel padding border, then the resulting convolved image is $(n + 2p - f + 1) \times (n + 2p - f + 1)$.
  - **Valid Padding** means that no padding is used.
  - **Same Padding** means a padding such that the size of the output image is the same as the input image, so $p = \frac{f-1}{2}$ (from solving $\frac{n+2p-f}{s} + 1 = n$ and assuming a stride of 1).
- **Strided convolution**: use a different step than 1 (e.g. 2) when convolving. Size of the output (for each dimension) becomes $\lfloor \frac{n+2p-f}{s} + 1 \rfloor$ where s is the stride.

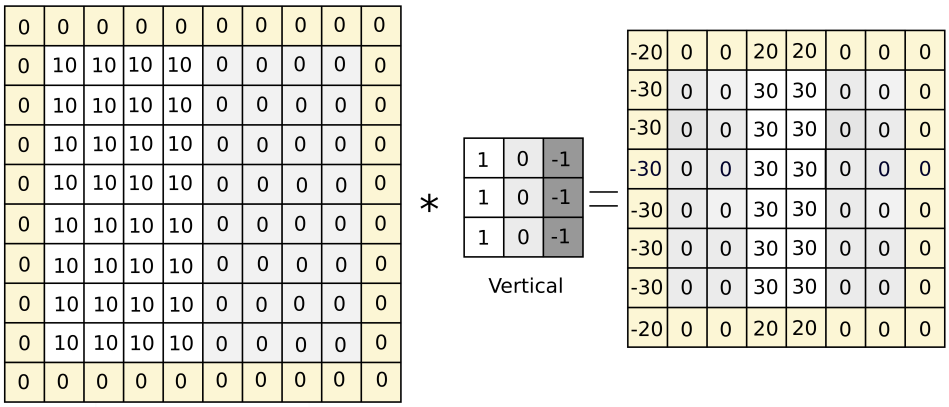

Figure 3: Padding.

- If we need the resulting image to be of a specific (or the original) size, then we need to solve for p the equation $\frac{n+2p-f}{s} + 1 = t$ where t is the target size (of each dimension).
- By convention f is always odd (so it has a central position and we can use the math above, but for no other specific reason).
- In math textbooks convolution is a mirroring step where the filter is flipped in the x and y axes. So technically the convolution used in Deep Learning is called "cross-correlation", but in Deep Learning we still just call it convolution by convention. The flip is used in signal processing to ensure the associativity property of convolution $(A * B) * C = A * (B * C)$, not required in deep learning.
- On RGB images the filter is a cube $f \times f \times f$. Applying convolution is similar, but at each step, we do f^3 multiplications and then we sum all the numbers to get 1 output value. The sizing formulas above apply because the number of channels is not taken into account to determine the size of the output.
- If we have multiple filters (figure (5)), we can apply them to the same image and then stack the multiple outputs as if they were different channels n_C, into a *volume*. The term *depth* is often used to denote the *number of channels*, though that can be more confusing.

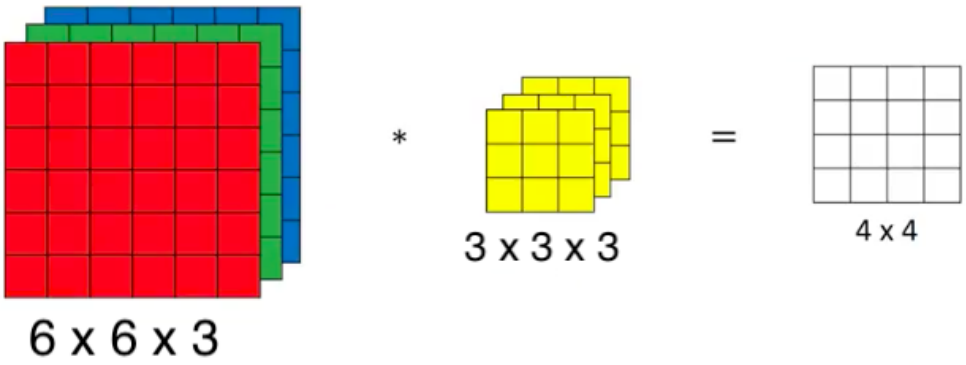

Figure 4: Convolution of a kernel on image

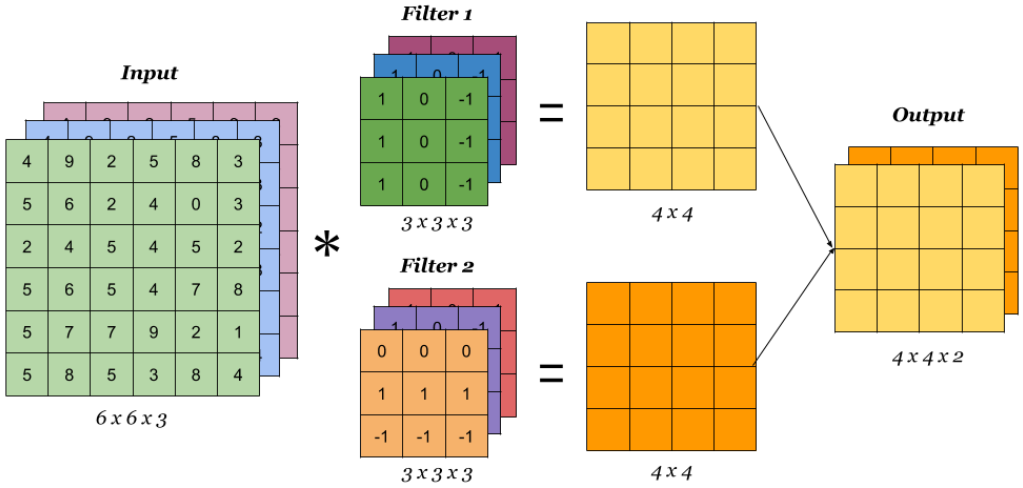

Figure 5: Convolution using multiple filters [2]

You can find an illustrated tutorial on convolution operation on my website:
Understanding 1D, 2D, and 3D convolutional layers in deep neural networks

1.2 One layer of a convolutional network

- Performs the convolution of the input image across all channels n_C, uses multiple filters, and therefore originates multiple convolution outputs.
- A bias term b is then added to the output of the convolution of each filter W and we get the equivalent of the term Z in a normal neural network.
- We then apply an activation function such as ReLU Z (on all channels), and we finally stack the channels to create a "cube" that becomes the network layer's output. See figure (7).

Notation and volumes:

- $f^{[l]}$ = filter size of layer l
- $p^{[l]}$ = padding of layer l
- $s^{[l]}$ = stride of layer l
- $n_C^{[l]}$ = number of filters/channels of layer l
- Input size: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- Output size: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- Output volume height: $n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
- Output volume width: $n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
- Each filter volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- Activations volume: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

- Activations volume for M examples: $A^{[l]} \rightarrow M \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]} = M \times a^{[l]}$
- Weights volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$ (# of filters)
- Bias size/number: $n_C^{[l]}$

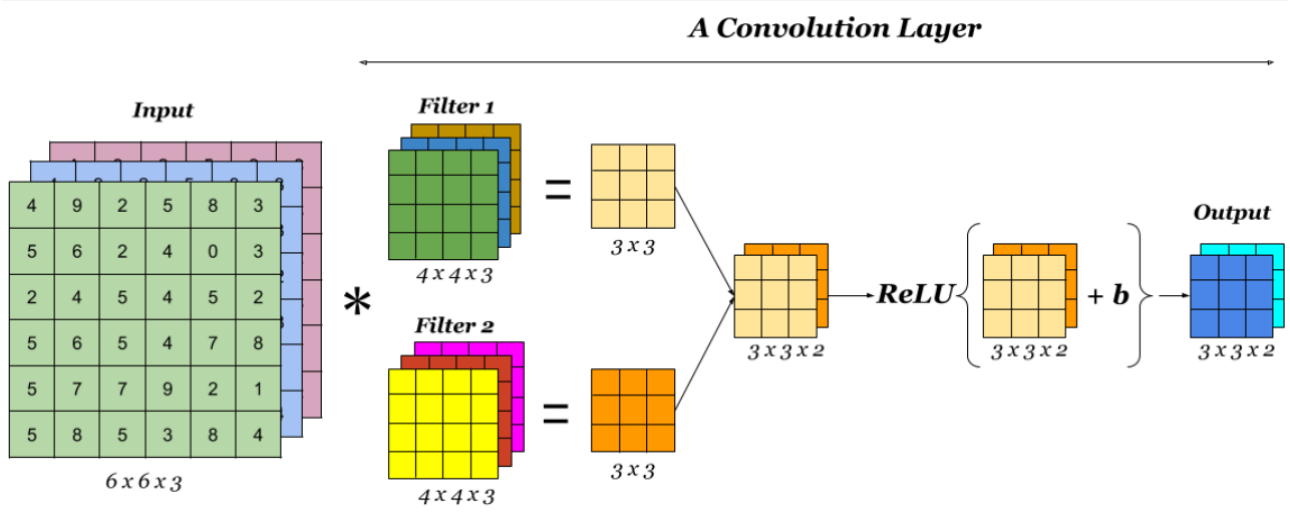

Figure 6: One Convolution Layer [2]

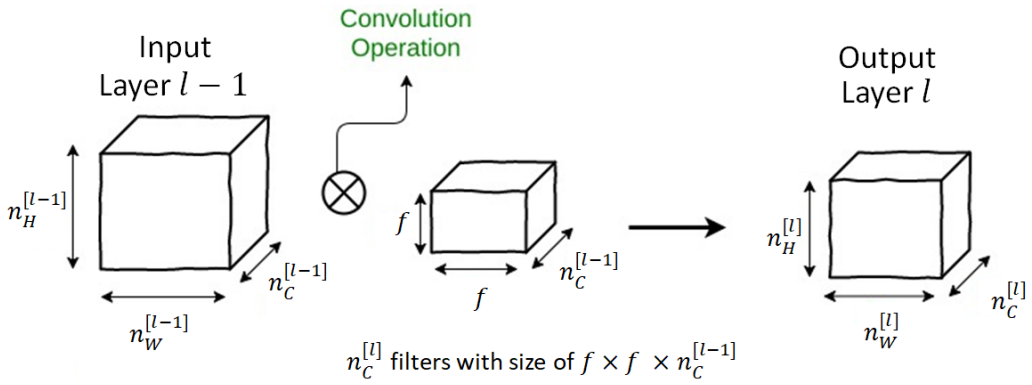

Figure 7: Convolution operator

1.3 Convolutional network

- Stack of layers as above (CONV layers)
- The last layer is a logistic or sigmoid, as required.
- There are also pooling (POOL) layers and fully connected (FC) layers.
- Sometimes POOL layers are not counted as layers since they don't have parameters to optimize. Here we will assume that one layer is a combination of CONV + POOL

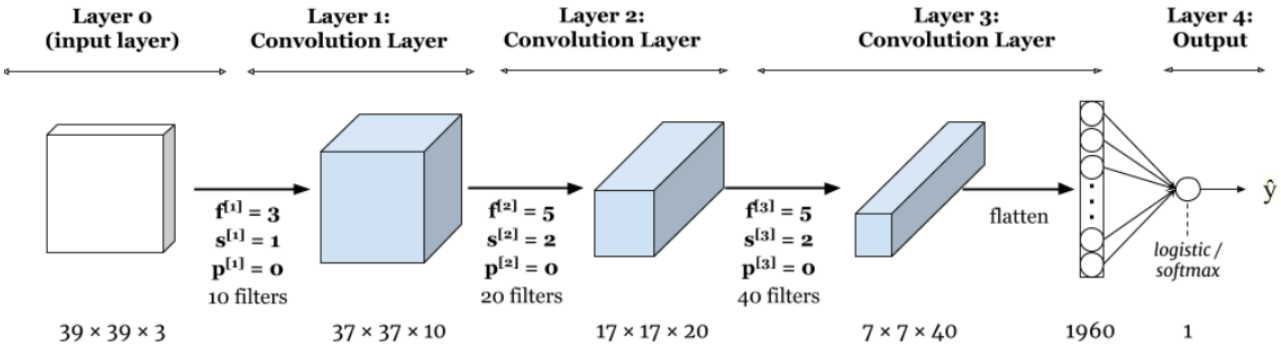

Figure 8: Sample Complete Network [2]

1.4 Pooling layers

- **Pooling** layer is used to reduce the size of the representations (width and height) and to speed up calculations, as well as to make some of the features it detects a bit more robust.
- **Max pooling** defines regions of size f (filter size) and step through them according to stride s, and creates an output matrix containing only the max value of each region. This normally reduces the size of the representation.
- A pooling layer does not change the number of channels (depth) of the input.
- The intuition for using max pooling (except wide acceptance) is that it expresses that if a feature exits in a region, we will express it more evidently.
- Computations are made independently for each channel.
- Average pooling is also possible but less used, except deep in a neural network to collapse a representation, reducing image size.
- There are no parameters to optimize. Hyperparameters: f, s, and the type of pooling (max or average).
- Padding usually is not used for pooling layers (though there are exceptions).
- Size of the output will be calculated by *the same output volume* formulas above. Precisely, the output size of applying a pooling layer with the size of f and stride s to an input with the size of $n_H \times n_W \times n_C$ will be: $\lfloor \frac{n_H - f}{s} + 1 \rfloor \times \lfloor \frac{n_W - f}{s} + 1 \rfloor \times n_C$

1.5 Fully connected layers

- These are layers that take the output of CONV or CONV + POOL layers, and take each individual pixel as an input (instead of a volume with channels/filters) by collapsing the volume into a single vector with an equivalent size.
- A fully connected layer is a normal neural network layer (e.g. with ReLU activation) that takes this collapsed output of the previous layer.
- FC layers have significantly more parameters than CONV.

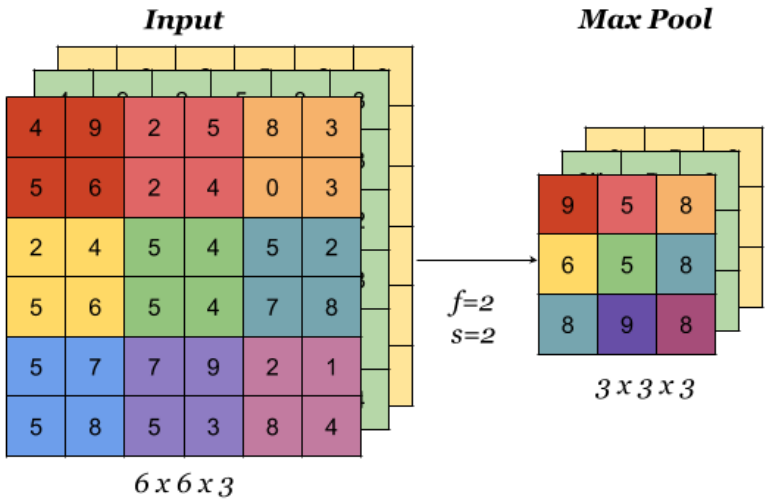

Figure 9: Pooling Layer [2]

1.6 Why convolutions

- Even for a small image, normal NN would use a very high number of parameters per image.
- **Parameter 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.
- **Sparsity of connections**: In each layer, each output value and the number of parameters depend only on the size of filters and it's independent of the size of the input of image/previous layer.

1.7 Historical/Classic architectures

- LeNet-5
  - INPUT -> CONV -> POOL -> CONV -> POOL -> FC -> FC -> OUTPUT
  - Uses Sigmoid/tanh instead of ReLU
  - Average pooling (not max pooling)
  - Includes not-linearity after the pooling
  - Different filters for each channel (not fully understood unless reading the paper)
  - Paper: "Gradient-based learning applied to document recognition"
- AlexNet
  - INPUT (227x227x3) -> CONV -> POOL -> CONV -> POOL -> CONV -> CONV -> CONV -> POOL -> FC -> FC -> Softmax -> OUTPUT
  - Similar to LeNet but much bigger
  - Uses max pooling
  - Most convolutions are "same"
  - Uses ReLU activation

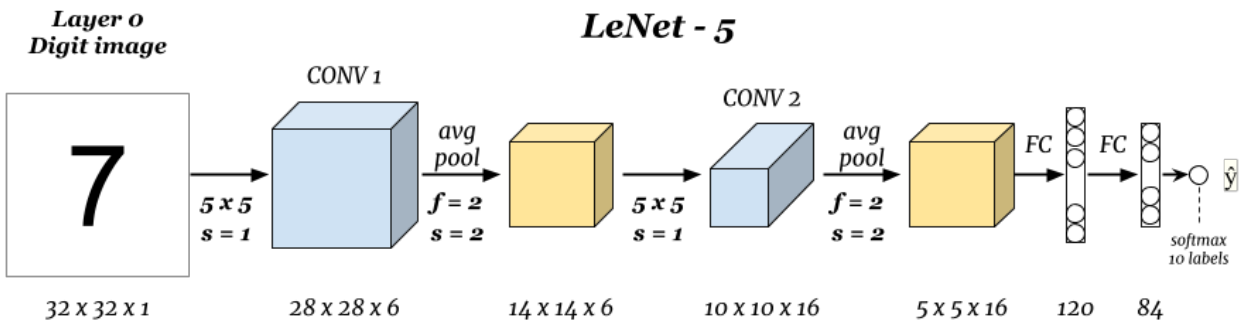

Figure 10: Lenet 5 [2]

- Local Response Normalization -> normalizes each position in a volume across all channels of the volume (not very used in practice)
- Paper: "ImageNet classification with deep convolutional neural networks"
- $\sim$ 60 million parameters

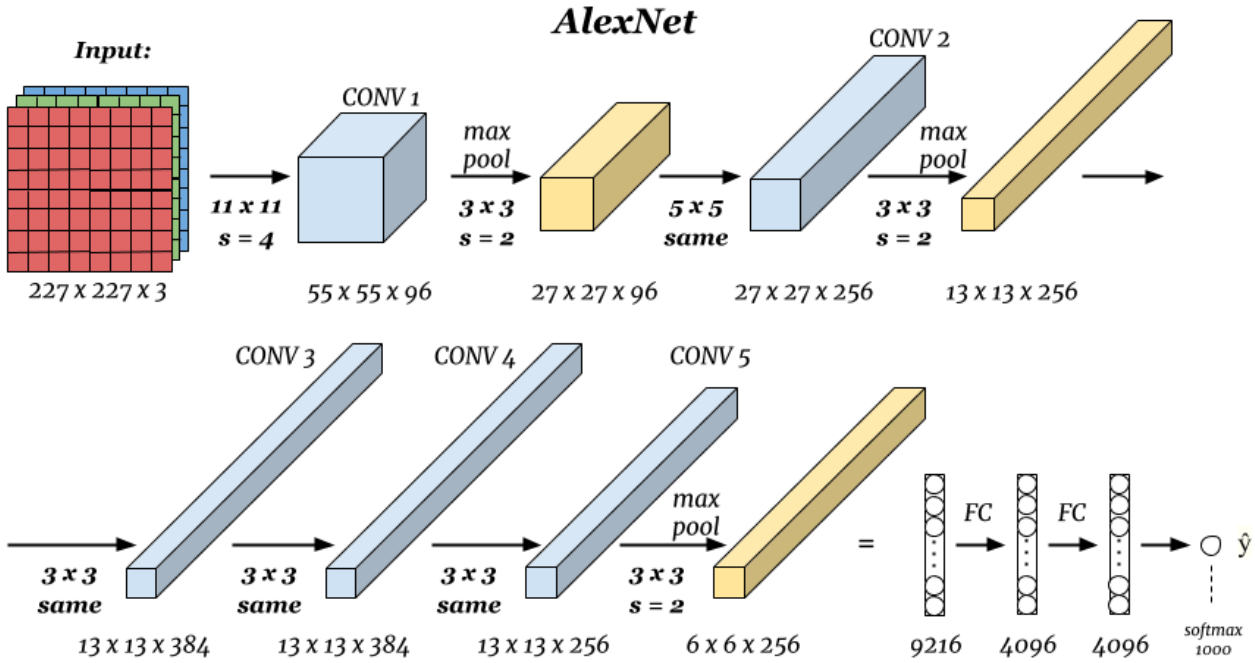

Figure 11: AlexNet [2]

- VGG-16
  - INPUT (224x224x3) -> CONV (64) -> POOL -> CONV (128) -> POOL -> CONV (256) -> POOL -> CONV (512) -> POOL -> CONV (512) -> POOL -> FC (4096) -> FC (4096) - Softmax (1000)
  - All CONV = 3x3 filter, s = 1, same
  - All MAX-POOL = 2x2, s = 2

– $\sim$ 128 million parameters
– Paper: "Very deep convolutional networks for large-scale image recognition"

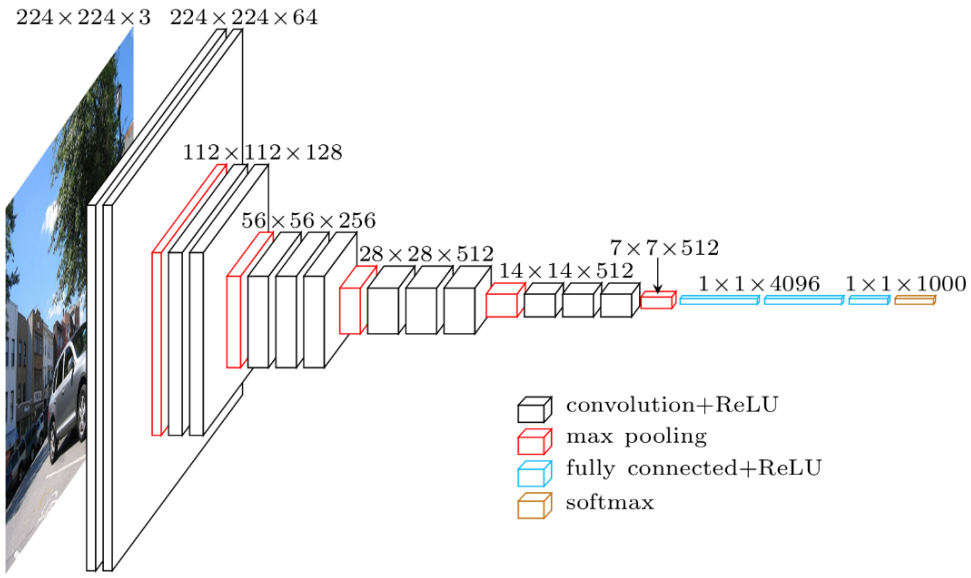

Figure 12: VGG 16

1.8 ResNet

- Paper: "Deep residual networks for image recognition"
- Add "shortcut"/"skip connection" to the network (More details on Skip Connection), where activation of past layers is included when calculating the activation of layers in the future:

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$$

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

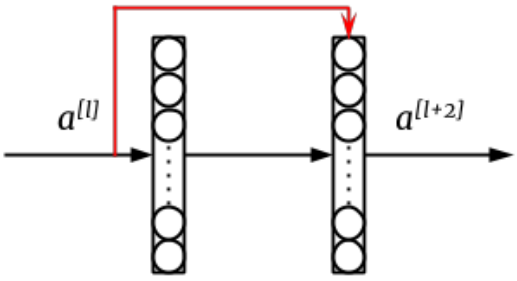

Figure 13: Skip Connection [2]

- These are residual blocks. A ResNet will have a sequence of these blocks.
- This allows training much deeper networks because the training error always goes down with an increase in the number of layers (which is not the case with traditional CONV nets):

  - "Learning the identity function is easy" - This means that it doesn't hurt to add to layers even if they do not contribute to the activation of previous layers. When regularization is applied, then $W^{[l+2]} \approx 0$ and $b^{[l+2]} \approx 0$ and then $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$
  - This means that a larger network will at least do as good as a smaller network, never worse.
  - During training of a traditional deep convolutional network we might see the magnitude (or norm) of the gradient for the earlier layers decrease to zero very rapidly as training proceeds. In ResNets the "shortcut"/"skip connection" allows the gradient to be directly backpropagated to earlier layers, thus solving the problem of **vanishing gradients**.

- For the operation $z^{[l+2]} + a^{[l]}$ to be possible, "same" convolutions must be used. Alternatively, there could be a W_s matrix such that $z^{[l+2]} + W_s a^{[l]}$ makes the dimensions match. This matrix could also add padding, or it could be just additional parameters to learn.

- Another similar (mentioned to be more "powerful") design allows for skipping 3 layers instead of 2.

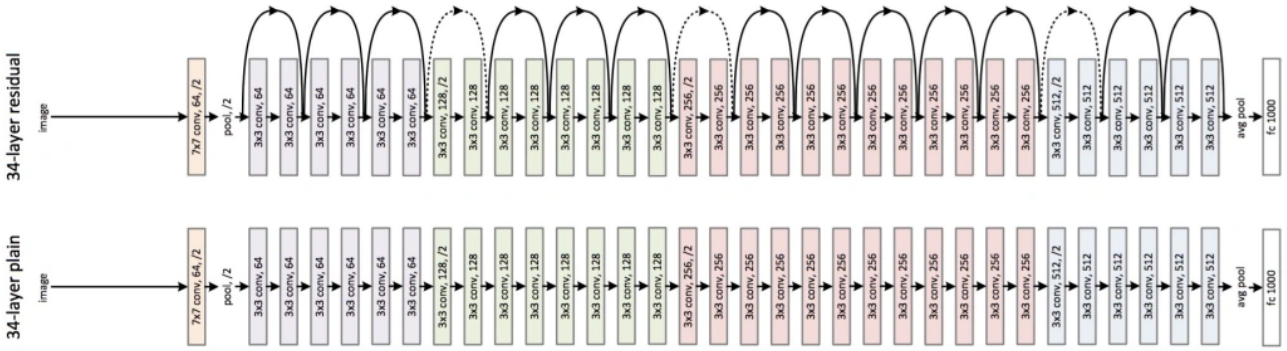

Figure 14: ResNet

1.9 Networks in Networks / 1x1 Convolution

- Useful only when the input has multiple channels.
  - Like having a fully connected layer for each pixel in the input.
- Useful to shrink the number of channels in a volume. Whereas pooling only reduces height and width.

1.10 Inception network (aka GoogleNet)

- Instead of choosing which filters to use, we used them all and the network chooses what's important!
- In a single layer, applies several types of filters, including:
  - 1x1 convolutions
  - Several "same" convolutions with different filter sizes (and possible stride sizes)

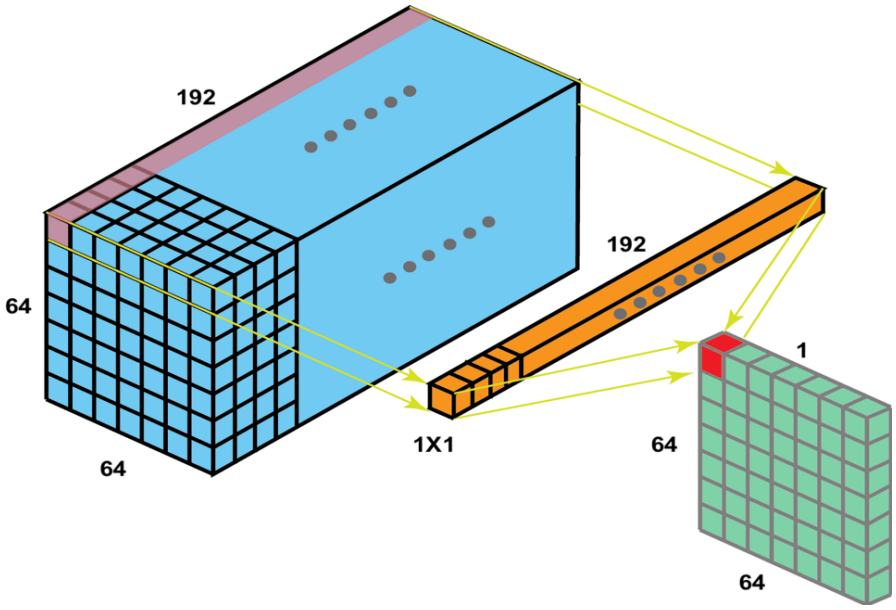

Figure 15: 1×1 Convolution

  - MAX-POOL directly from the input, in which case **padding must be added before the MAX-POOL** to ensure that the height and width of the output are the same as the input - this is an exceptional case since padding is not usually added when doing POOL.

- To reduce the computation cost of convolutions on many channels, 1x1 convolutions with a lower number of channels are used as an intermediate step or "bottleneck" for convolutions with a larger number of channels (reducing the number of operations by a factor of 10, compared with doing the larger convolution on the input directly). See figures (17) and (18).
- Similarly, after the MAX-POOL step, a 1x1 CONV is used to reduce the number of channels in its output.
- So, 1x1 CONV are used as intermediate (or final in the case of MAX POOL) steps in each layer either as bottlenecks are to reduce the number of channels at the output of MAX-POOL.
- One layer containing these 1x1 CONV, the larger CONV and the MAX-POOL (in parallel) is called an "Inception Module" (19) of which there are many in a network.
- There is a final FC and Softmax layer.
- Side-branches (feature of inception networks): Take some hidden layers and use FC and then Softmax to predict a label. This ensures that even hidden layers are not too bad to predict the output cost. These have a regularizing effect pn the network.
- Paper: "We need to go deeper".

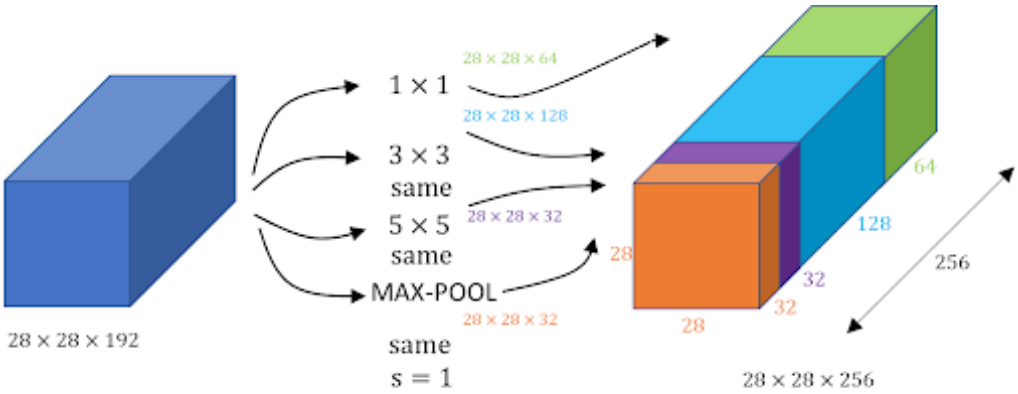

Figure 16: Inception Block

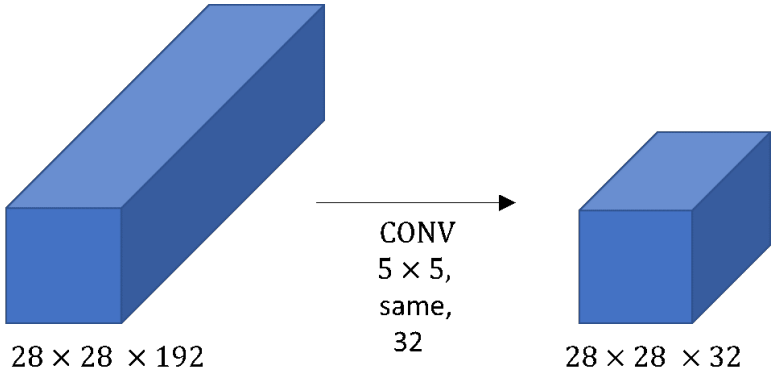

Figure 17: Huge computations in inception

1.11 Using Transfer Learning

- Use transfer learning/pre-trained models. Example: *Imagenet* (1000 output classes).
- Put your own Softmax output layer:
  - Freeze the parameters in the other layers (e.g. trainableParameter=0 or freeze=1, depending on framework).
  - (with small data) Pre-compute the output of the network before the new Softmax layer across training examples and save them to disk (so you don't have to compute these on every epoch of the shallow training). Then train only the shallow Softmax model.
  - (with more data) Include a few of the last few layers in the training, not just Softmax. We may want to retain or not the weights of some of the unfrozen layers (used them as initialization) and add just some additional extra training there. Alternatively, the last unfrozen layers could be randomly initialized, re-designed, etc. The more data you have, the fewer layers you freeze.

1.12 Data Augmentation

- Having more data helps in the majority of computer vision tasks (not the same for other areas of ML).

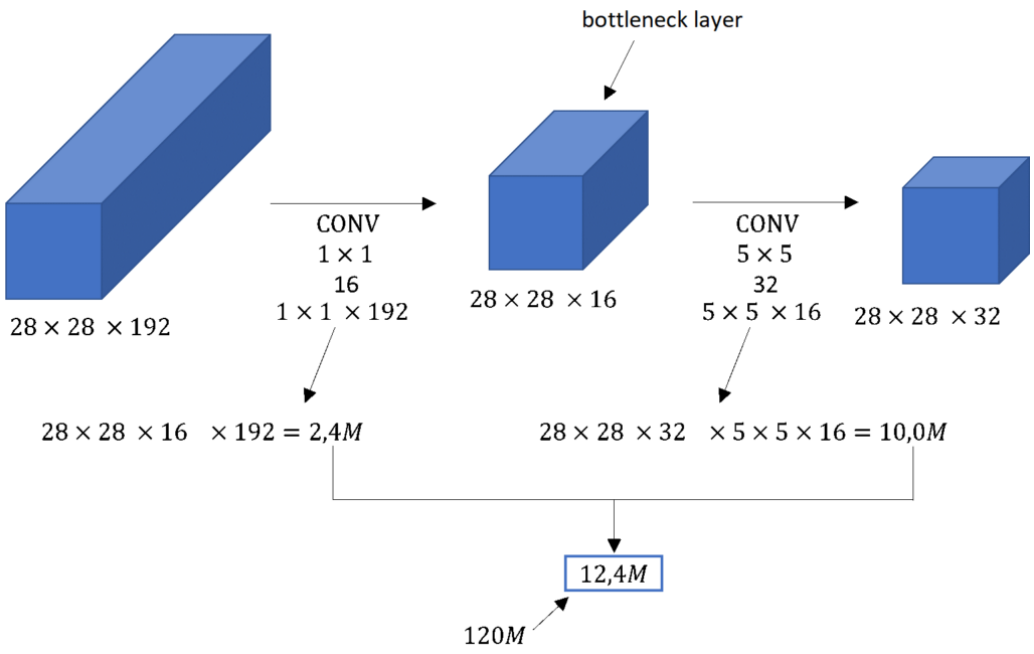

Figure 18: Reduce computations by 1×1 CONV

- Methods (from existing images):
  - Mirroring
  - Random cropping (works as long as the crops are reasonable)
  - Rotation
  - Shearing
  - Local warping
  - Color shifting: change the RGB color balance according to some probability distribution. (it is possible to use PCA to choose RGB - check AlexNet paper for "PCA color augmentation")
- A common way of implementing augmentation is to have one (or multiple) CPU threads do the augmentation/distortion, and then run the network training in parallel in another thread or GPU.

1.13 State of computer vision

little data ← object detection - image recognition - speech recognition → **lots of data**

Two sources of knowledge:

- Labeled data (x,y)
- Hand-engineered features/network architecture/other components

Often for computer vision, there is not as much data as needed. So hand-engineered features are more needed.

- Even though the amount of data increased dramatically in the last few years there is still a lot of hand engineering, specialized components and hyperparameters for historical reasons.

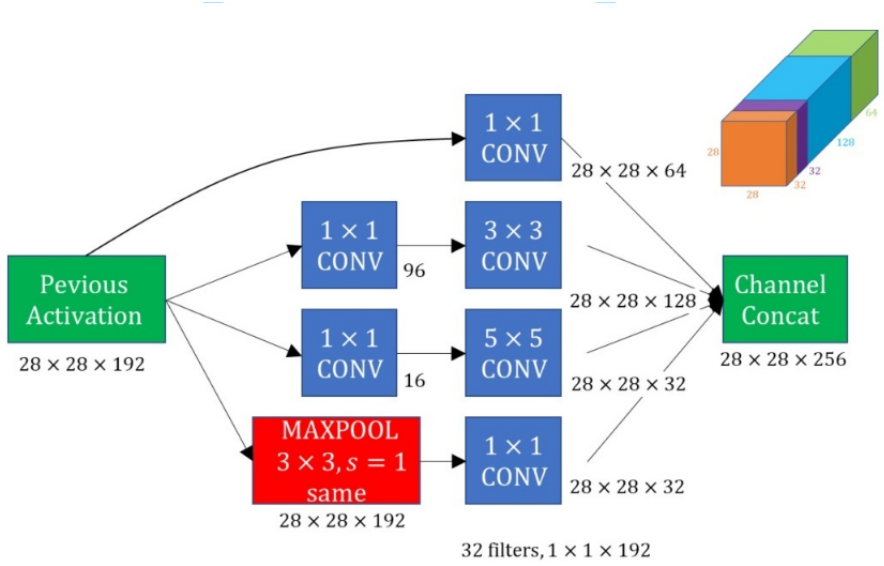

Figure 19: Inception Module

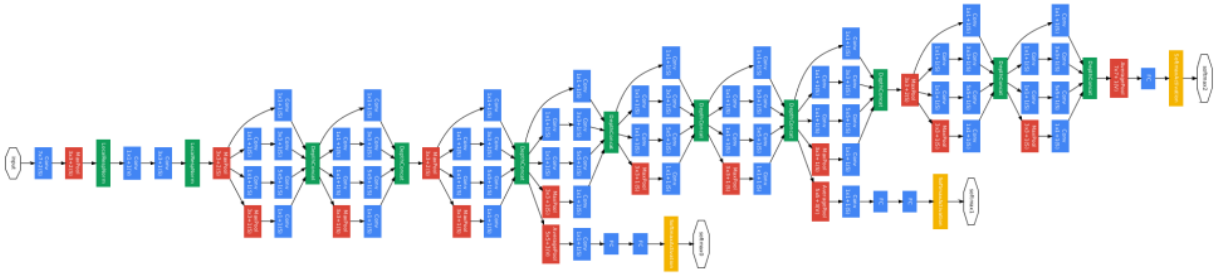

Figure 20: Inception Network

1.14 Tips for doing well on benchmarks/winning competitions

(Not for production environments)

- **Ensembling**: train several networks independently and average their output. (typical 3-15 networks, lots of CPU time) - usually not used in production.
- Multi-crop at test time (data augmentation) - run the classifier on multiple versions of test images and average the results - more used in benchmarks rather than production systems.

1.15 Classification with localization

- Identify the object and localize it in the picture.
- Add 4 more outputs (in addition to the object classes) that are the bounding box coordinates b_x, b_y, b_h, b_w, that is the bounding box center coordinates and its height and width to the labels vector, respectively. Add another output P_c that indicates if there is an object in the image. For instance, in an object detection problem with three classes (c_1, c_2, c_3), the label vector for

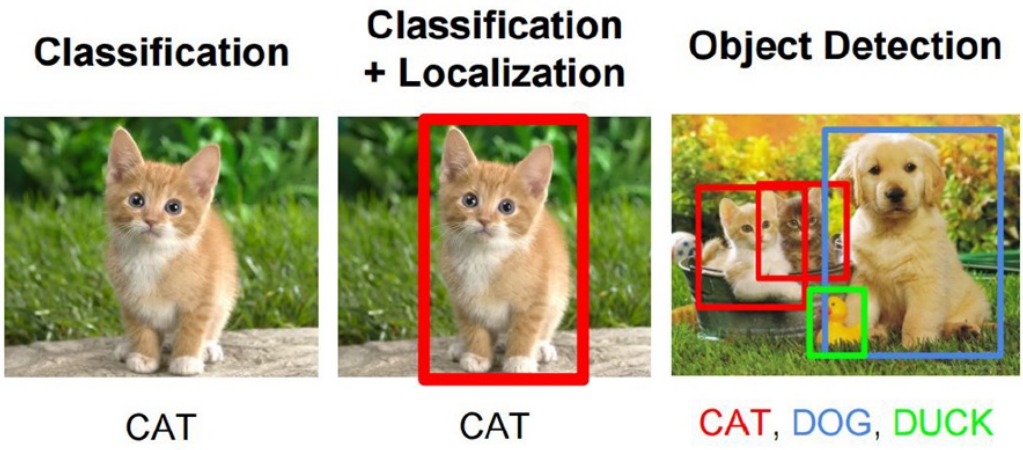

Figure 21: Object Classification, Localization, and Detection

each sample would be as follows:

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

- If P_c is 0 (no object identified) then the loss function must take that into consideration, basically just calculating $(\hat{y}_1 - y_1)^2$ and ignoring the remaining components (this example uses squared error, but it could be a log-likelihood, etc.).

$$L(\hat{y}, y) = \begin{cases} (\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2 + \cdots + (\hat{y_8} - y_8)^2, & \text{if } y_1 = P_c = 1 \\ (\hat{y_1} - y_1)^2, & \text{if } y_1 = P_c = 0 \end{cases}$$

1.16 Landmark detection

- Basically use a conv network to output the coordinates of the points of interest (e.g. find the corners of the eyes in a person's picture). It can be a high number of points of interest though.
- Landmark labels must be consistent between training/test examples.

1.17 Object detection

- First build a conv net that identifies a certain class(es) of objects, and train it only on pictures where the positive examples include a single object (e.g. a single car).

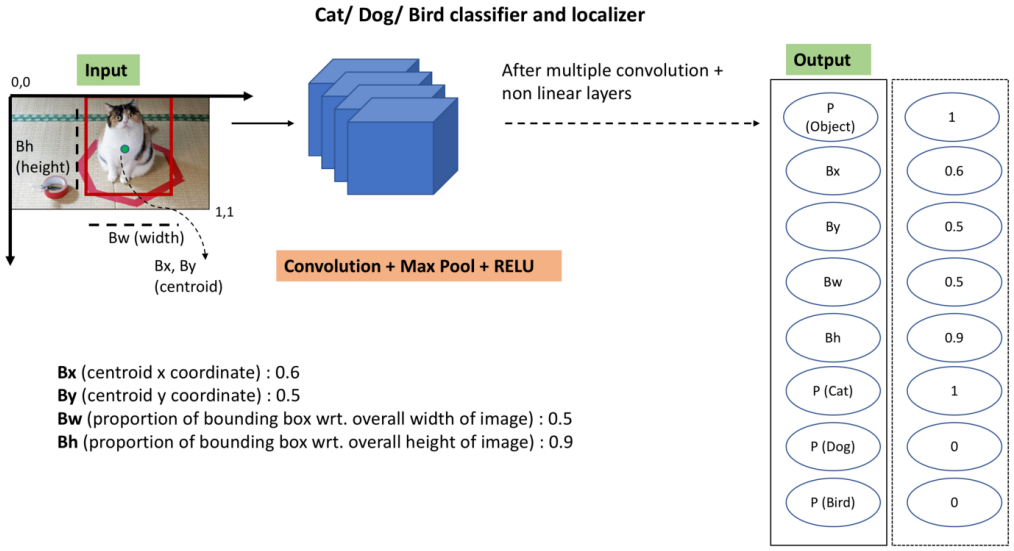

Figure 22: Label representation in Object Detection [1]

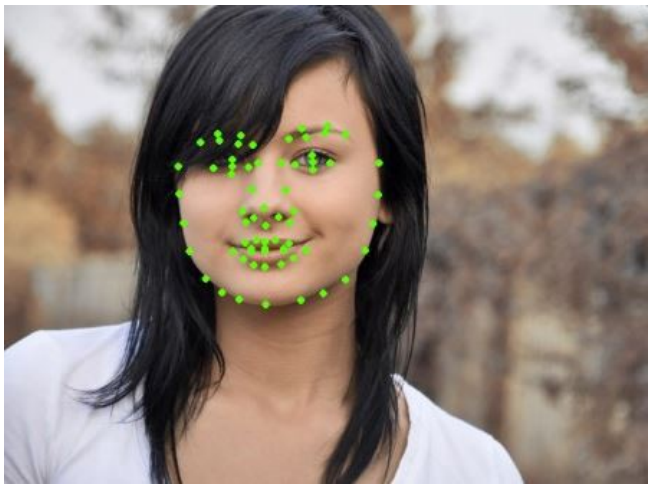

Figure 23: Landmark Detection

- To identify multiple cars, use a sliding window approach (with windows of varying sizes, over multiple passes), and run each crop via the conv net. However, this is extremely inefficient because we don't know the stride or the size of the crops, and we need to evaluate a high number of crops.

1.18 Convolutional implementation of sliding windows

First a useful tool: Turning FC layers into convolutional layers (25):

- Basically just use a filter with the same height and width as the input volume (no padding and or stride used), e.g. 5x5 in this example. The number of channels/filters can then correspond the number of units that we would have in an FC layer (e.g. 400 in this example). The output will be a 1x1x400 volume, which is equivalent to having a fully connected layer with 400 units!

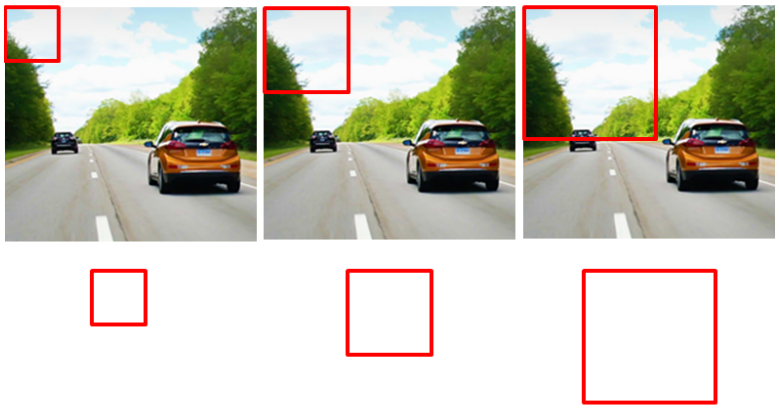

Figure 24: Sliding Window

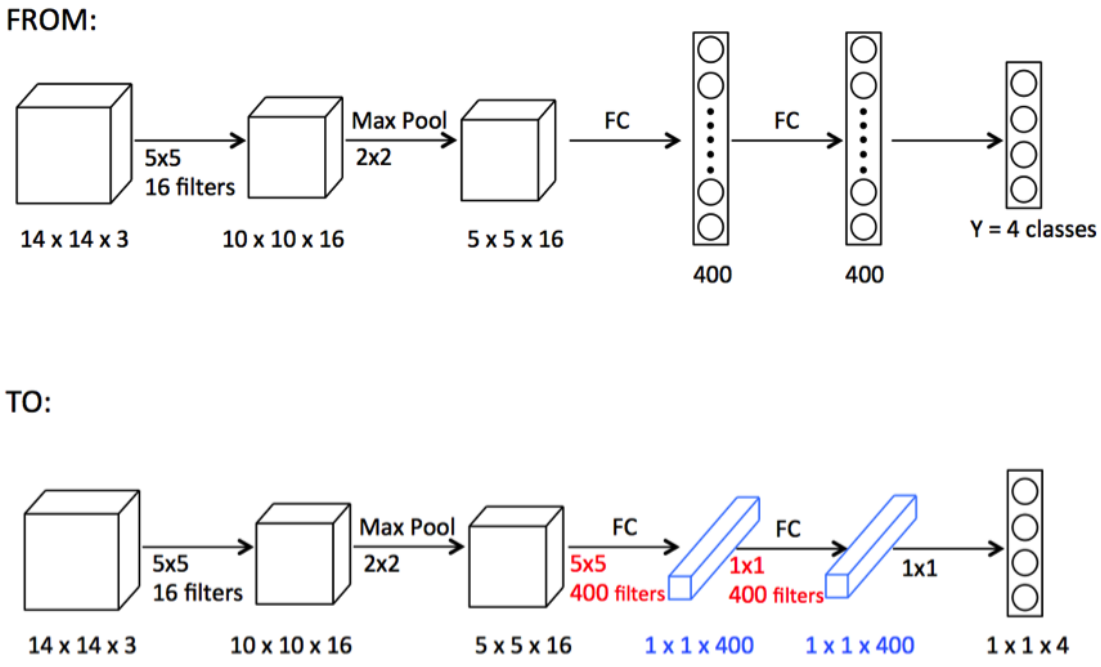

Figure 25: Converting Fully Connected layers to Convolutional Layers

And now the convolutional implementation for sliding windows:

- The objective is to implement a convolutional network that **in a single pass** calculates the equivalent of the convolution of a number of sliding windows, of a certain size and with a certain stride. If our test image is of dimension 16x16x3 and we had to perform the "regular" sliding window we would have to create 4 different windows of size 14x14x3 out of the original test image and run each one through the ConvNet.
- This is computationally expensive and a lot of this computation is duplicated. We would like, instead, to have these four passes to share computation. So, with the convolutional implementation of sliding windows we run the ConvNet, with the same parameters and same filters on

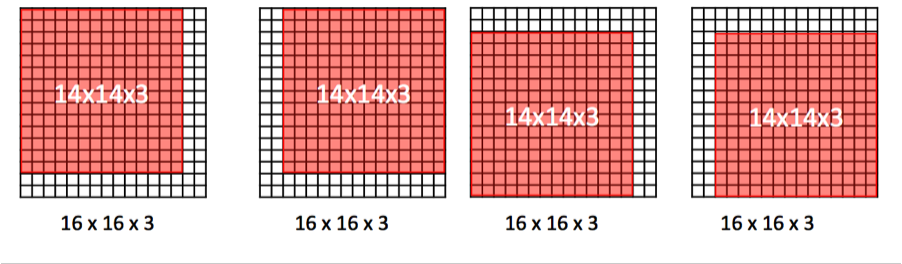

Figure 26: Sliding windows using Conv Net

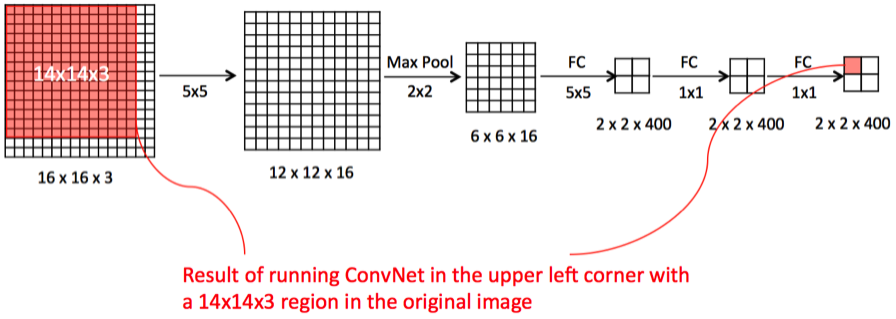

Figure 27: Computing all windows with one feed forward

the test image and this is what we get in figure (27).

- Each of the 4 subsets of the output unit is essentially the result of running the ConvNet with a 14x14x3 region in the four positions on the initial 16x16x3 image.
- Think about an input image of 28x28x3. Going through the network, we arrive at the final output of 8x8x4. In this one, each of the 8 subsets corresponds to running the 14x14x3 region 8 times with a slide of 2 in the original image.
- One of the weaknesses of this implementation is that the position of the bounding box we get around the detected object is not overly accurate.

1.19 YOLO algorithm

- Efficient algorithm, convolutional implementation, runs very fast. YOLO paper: "You Only Look Once: Unified real-time object detection".
- A single CNN simultaneously predicts multiple bounding boxes and class probabilities for those boxes.
- First divide the input image into a grid (say 19x19). We want to use a conv net for each volume in a grid, or better do one single conv net pass that performs the equivalent of that. Say that for each grid division we want to have an output vector $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$ (assuming 3 classes), we want to obtain an output volume of dimensions 19x19x8 (8 is the dimension of the vector for each grid cell)
  - This can be achieved in a single convolution by setting the appropriate filter and max pool sizes/strides.

- YOLO considers that the grid cells that have an object are those that contain the center coordinates of the bounding box for the object, b_x and b_y.
- How to encode bounding boxes (that may go beyond the grid cell)? See figure (28).
  - b_x and b_y are between 0 and 1 and are specified relative to the bounding box start point in the upper left corner (0,0).
  - b_h and b_w are specified as a proportion in relation to the **grid cell size**, and therefore can be greater than 1.

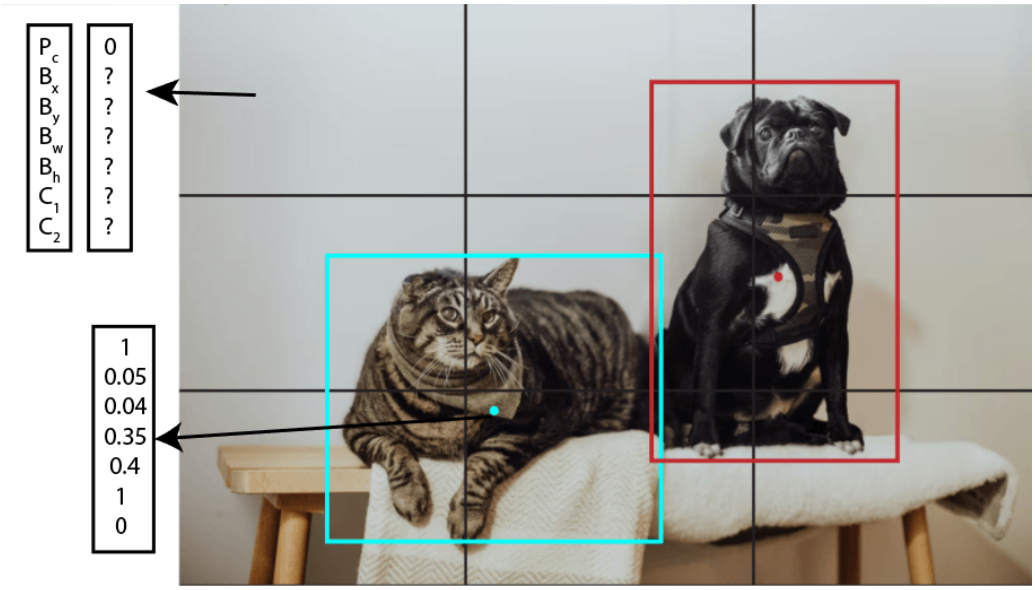

Figure 28: Bounding Boxes

Intersection over Union (IoU) - to evaluate object localization

- Given 2 overlapping bounding boxes, we define:

$$IoU = \frac{size\ of\ intersection}{size\ of\ union}$$

- We say that a bounding box is "correct" if its IoU with the ground truth bounding box is higher than 0.5 ($IoU \geq 0.5$). This is just a convention. 0.5 may be too low if we are more demanding (but never lower than 0.5).

Non-max suppression

- One issue: the algorithm may predict several bounding boxes for the same object. Non-max suppression helps to select only one box per class, that has the highest probability.
- First of all we discard any bounding boxes with $P_c \leq 0.6$.
- Then we choose the highest probability bounding box first, and then iteratively eliminate the overlapping bounding boxes with $IoU \geq 0.5$, then select the next highest P_c, eliminate overlapping boxes with $IoU \geq 0.5$, and so on, while there are still boxes remaining.

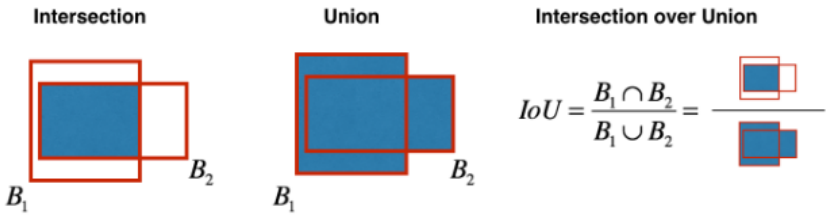

Figure 29: IoU

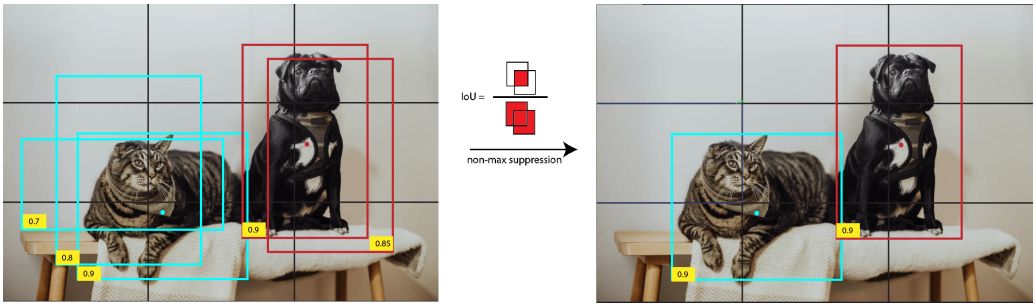

Figure 30: Non-max Suppression

Anchor boxes

- Goal: Detect several objects within the same grid. Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect.
- Anchor boxes increase the number of output parameters to encode multiple possible object detection. E.g. with two anchor boxes (and for 3 classes as before) the output vector is $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$, notice that all parameters are doubled, one of each anchor box.
- Previously: Each object in the training image is assigned to the grid cell that contains that object's midpoint.
- With anchor boxes: Each object in the training image is assigned to the grid cell that contains the object's midpoint and anchor box for the grid cell with the highest IoU.
- What if you have 2 anchor boxes but 3 objects? The algorithm doesn't work well in this case.
- If there are 2 objects but both have the same anchor box shape the algorithm won't work well either.
- One way to choose anchor box shapes is by doing K-Means clustering over the anchor boxes of the training set.

YOLO algorithm (second take)

- If we have a grid of 3x3, 3 classes and 2 anchor boxes, our output volume will be $3 \times 3 \times (5+3) \times 2$, where 5 comes from the components P_c, b_x, b_y, b_h, b_w.
- With 2 bounding boxes for each grid cell, get 2 predicted bounding boxes.

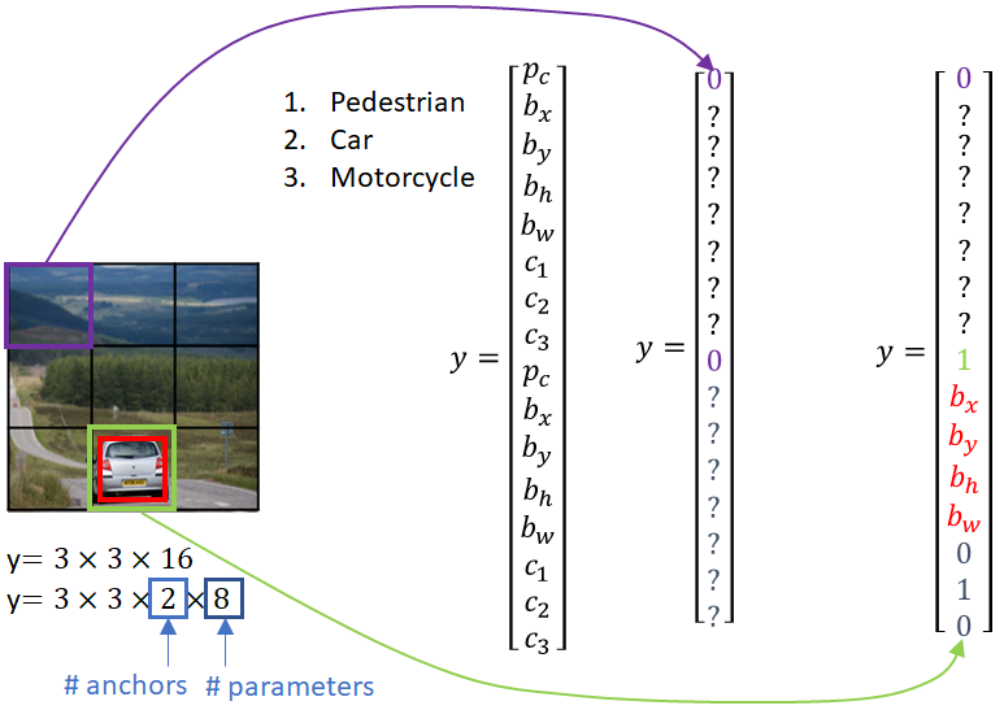

Figure 31: Anchor Boxes

- Get rid of low probability predictions.
- For each class, use non-max suppression to generate final predictions.
- Find out more details on my website:
  Understanding Object Detection using YOLO with Python implementation.

Region proposals

- **R-CNN** (R as in regions) Select only a few regions/windows that seem relevant (contain objects) instead of sliding window through the entire image:
  - To find the regions, a **segmentation algorithm** is used, to find "blobs" that are candidates to be enclosed by bounding boxes.
  - 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 windows.
  - Faster R-CNN: Use a convolutional network to propose regions. Still slower than YOLO.
  - Find out more details on my website: RCNN, Fast RCNN, and faster RCNN algorithms for Object Detection Explained.

1.20 Face recognition

- **Validation** (1:1 problem): Is he James? much easier problem than:
- **Recognition** 1:K persons. Who is that person? requires a database with recognized individuals.

- **One-shot learning**: Learning from 1 example to recognize the person again.
- Solution: Learning a **similarity function** for face verification instead of training a conv classifier on a small dataset:
  - $d(img1, img2)$ = degree of difference between images
  - if $d(img1, img2) \leq \tau$ same, otherwise different.

Siamese network

- Paper: "DeepFace closing the gap to human-level performance".
- Idea: run each photo through a convolutional neural network and use the output as an "encoding" of the picture $x^{(1)}, ..., x^{(m)}$ as $f(x^{(1)}), ..., f(x^{(m)})$.
- Then calculate the distance between two encodings that is the norm between the difference of the two encodings:

$$d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(1)})||_2^2$$

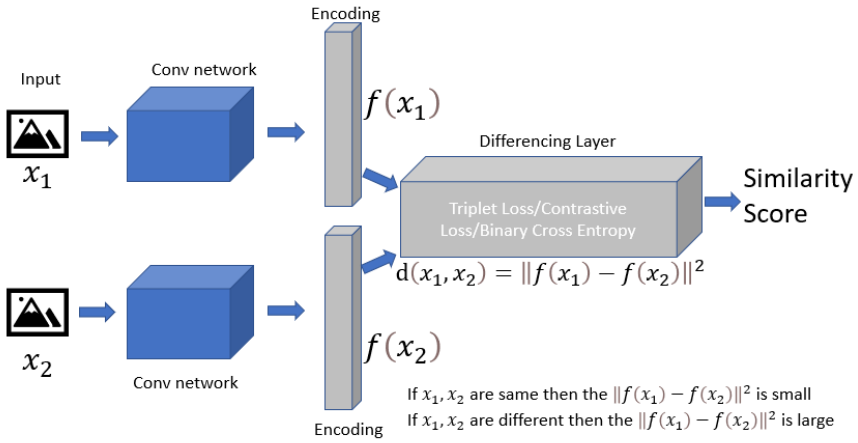

Figure 32: Siamese Network

The Triplet Loss function to learn parameters of the conv network

- Learning objective - Given an image A (*Anchor*), we want to find a function d such that d is small for another image of the same person P (*Positive*) and at the same time higher by an α **margin** when compared with an image of a different person N (*Negative*). In other words:

$$d(A, P) - d(A, N) + \alpha \leq 0$$

- Loss function - Given 3 images A, P, N (*triplet*) we define the loss as:

$$\mathcal{L}(A, P, N) = \max(d(A, P) - d(A, N) + \alpha, 0)$$

- We do need a dataset where we have multiple pictures of the same person (suggested 10 pictures per person on average).

- Choosing the triplets randomly, then it makes it too easy for the training.
- We should instead choose pictures where the negative is as close as possible to the positive, as these will be the best triplets to train on.
- Paper: "A unified embedding for face recognition and clustering".

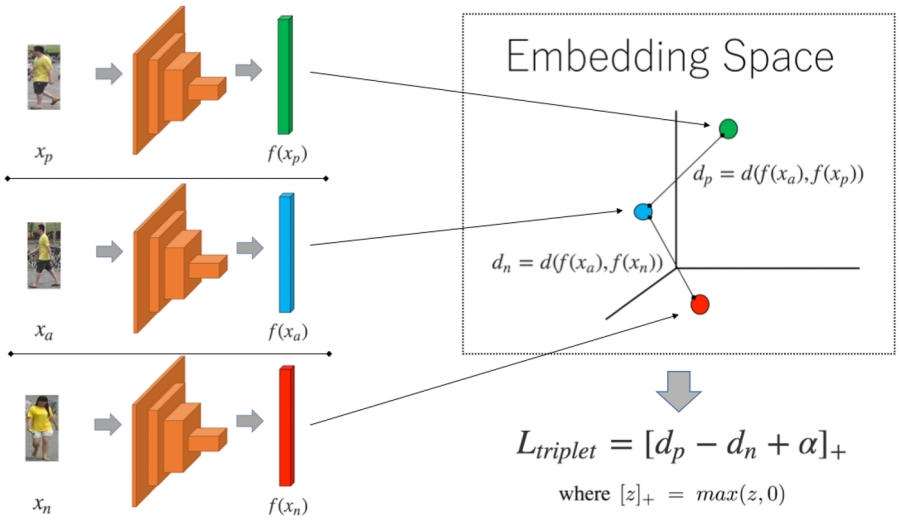

Figure 33: Triplet Loss

1.21 Face Verification and Binary classification

- Another way to learn weights of conv net.
- Example of **Siamese networks**, with a logistic regression unit for the output. That unit implements:

$$\hat{y} = \sigma(\sum_{k=1}^{N_f} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b)$$

- In this formula, $x^{(j)}$ and $x^{(i)}$ are training examples, w_i and b are the parameters of the logistic unit, N_f is the number of input features in each vector input to the logistic unit, k is one single feature (pair of nodes from each of the Siamese networks), and $f()$ is the function calculated by each of the Siamese networks.
- Take difference of embeddings as features of a logistic regression model.
- In some variations the χ^2 formula is used instead of the absolute value of the difference:

$$\chi^2 = \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

- Pre-compute encodings as a computational trick. We don't need to store the original images in a production verification system. Only the encodings of the images (e.g. for each individual).
- The Siamese network is trained using matching and mismatching pairs of pictures.

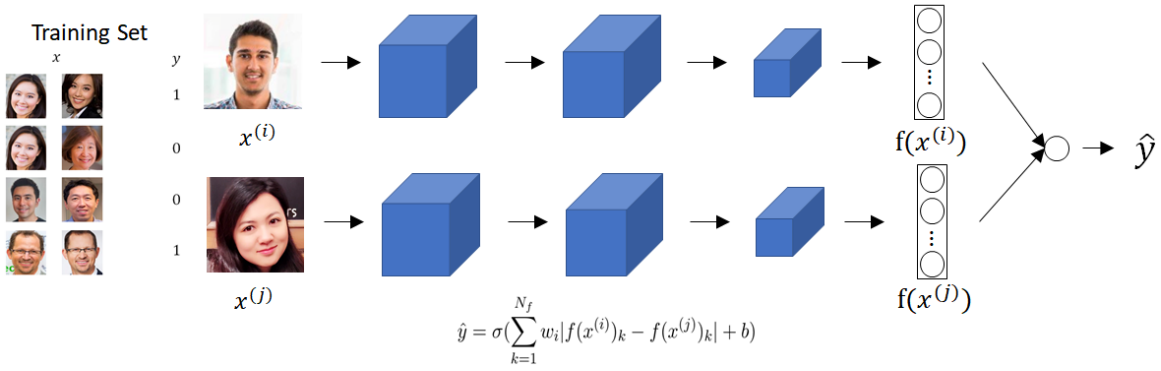

Figure 34: Face verification as a supervised learning problem

1.22 Neural style transfer

- Given a Content image C (a photo) and a style image S (e.g. a painting) generate a new version of the original content reflecting the style in the painting, the "Generated image" G.

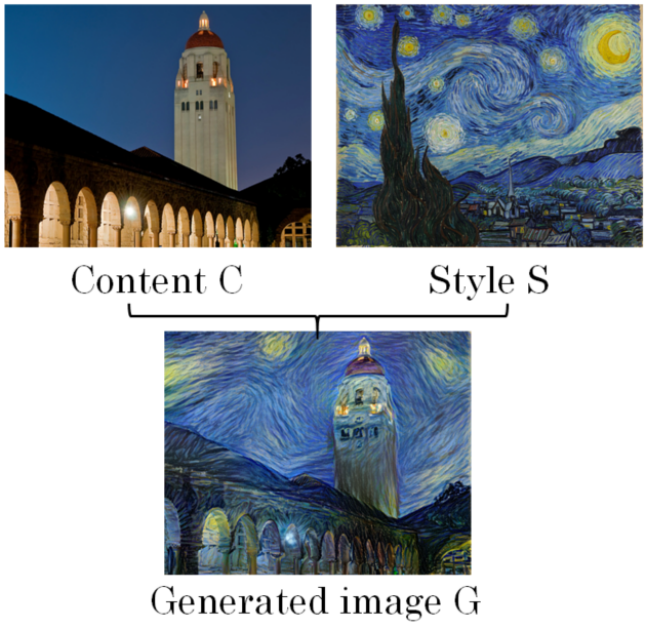

Figure 35: Neural Style Transfer

What are deep Conv Nets learning?

- Book/paper: "Visualizing and understanding convolutional networks".
- Pick a unit in layer 1, and find the nine image patches that maximize the unit's activation. Repeat for other units.
- Deeper units learn increasingly more complex/high-level features and shapes.

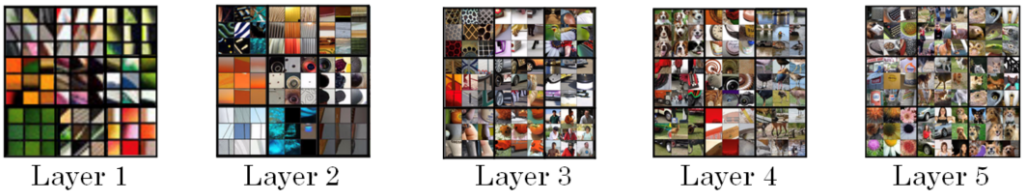

Figure 36: Visualizing deep layers

Cost function for neural style transfer

- Paper: "A neural algorithm of artistic style".

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

- Two components measure the difference between the content image to the generated and the style image to the generated image, where α and β are hyperparameters.
- Algorithm:
  1. Initialize G randomly (e.g. G: 100x100x3 randomly initiated)
  2. Use gradient descent to minimize $J(G)$ and update G (where α is the learning rate):

$$G = G - \frac{\alpha}{2G} J(G)$$

Content Cost Function

- Choose a layer l somewhere in the middle of the layers of the network (neither too shallow nor too deep).
- Use pre-trained ConvNet. (E.g., VGG networks) to measure how similar the content and generated images are.
- 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.
- Then calculate the L2 norm (element-wise sum of squared differences) of the activation vectors (activations are unrolled into a vector), with:

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

- Finally, when we are generalizing in a $n_H \times n_W \times n_C$ volume we can use a more appropriate normalization constant:

$$J_{content}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{[l](C)} - a^{[l](G)})^2$$

Style cost function

- Using l's activation to measure style, we define style as the **correlation between activations *across channels***.
- Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k).
  - The **Style Matrix** ("Gram Matrix" in algebra) is $G^{[l]}$ and is $n_c^{[l]} \times n_c^{[l]}$. k and k' are coordinates of the style matrix corresponding to the correlation between channels k and k'.
- We calculate style matrices for the style image and the generated image:

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \times a^{[l](S)}_{ijk'}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} \times a^{[l](G)}_{ijk'}$$

- Finally the style cost function can now be defined by the Frobenius norm (sum of squares of the element-wise differences) multiplied by a normalization constant (the fraction portion):

$$J^{[l]}_{style}(S, G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} ||G^{[l](S)} - G^{[l](S)}||^2_F$$

$$J^{[l]}_{style}(S, G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})$$

- Finally style transfer is better if all the layers are used. So we get the final style cost function (with additional hyperparameter vector λ):

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

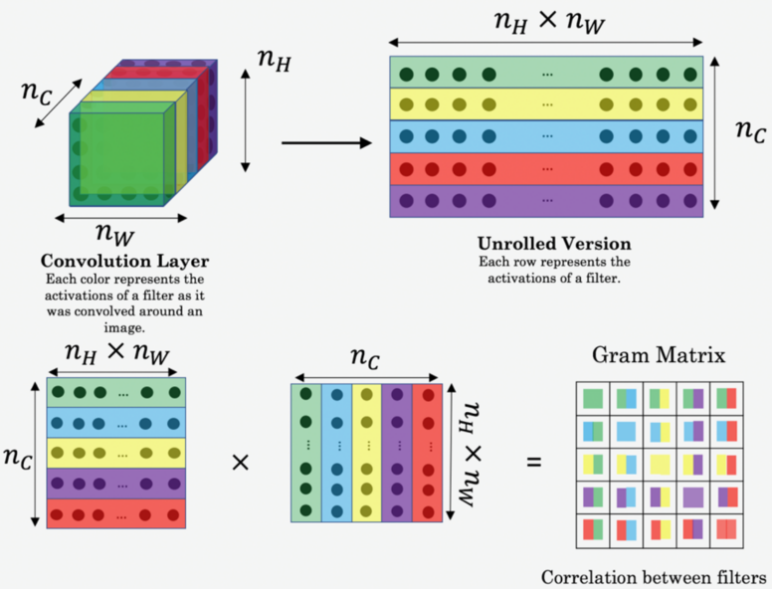

Figure 37: Style/Gram Matrix

References

[1] P. Grover. Evolution of Object Detection and Localization Algorithms. https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad.

[2] IndoML. Student Notes: Convolutional Neural Networks (CNN) Introduction. https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/.