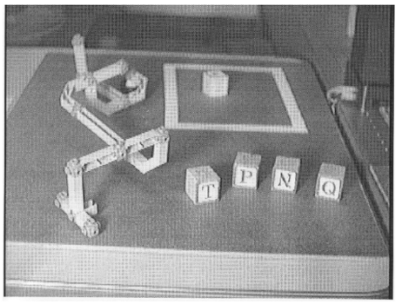
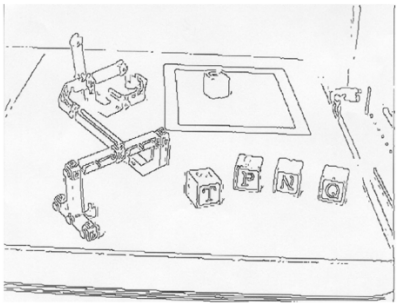
Convolutional Neural Networks

Computer Vision

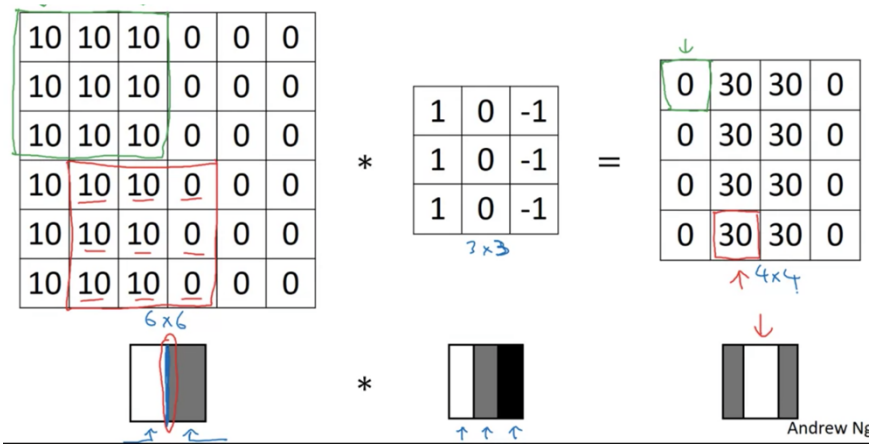
Edge Detection

Edges are significant local changes of intensity in an image. Edges typically occur on the boundary between two different regions in an image. To do Edge Detection, we can produce line drawing of a scene from an image of that scene. Important features can be extracted from the edges of an image, such as corners, lines etc. These features are used by higher-level computer vision algorithms





Example of simple vertical edge detection:



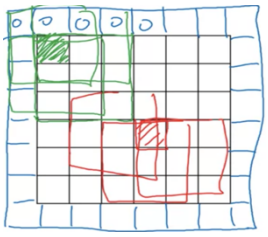
The leftmost matrix represents a simple picture with left half dark and right half bright. Convolute with the filter(middle matrix), with 1s represent bright, -1 represent dark, denotes the edges matrix on the right.

To calculate the right matrix:
*is the convolution notation
1strow 1stcol at right matrix = sum of element-wise product between green circle and middle matrix. And we calculate the 1strow 2ndcol by moving the green circle one col right
Different filters help you find different horizontal and vertical edges. And there are different types of edge filters, such as sobel filter and schar filter.
However, we don't have to choose filter values at the beginning of our model, we can treat the filter matrix as 9 different parameters to train and learning automatically.
But this has some problems:

- Shrinking output: as we can see, as we have more layers, the size of the output image after convolution becomes smaller
- Throwing away info from the edge: as we calculate the convolution, there are many green circles will cover the same entries in the middle of the input image, but edge entries will be covered much less, this led to less info covered from the edge

Padding

To solve the two problems of edge detection, we use padding.
Padding is when we add more entries around the input images, normally with zero.



p is the number of padding, here p =1

Two types of convolutions:
Valid Convolution: without padding -> nxn * fxf -> n-f+1 x n-f+1 , in the previous example, we have 6x6*3*3 -> 4x4
Same Convolution: pad so that the output size is the same as the input size: n+2p-f+1 x n+2p-f+1
To calculate the size of padding required for the image to keep the same size:
P = (f-1)/2
and f is usually odd because allows us to pad the same dimension all around, and to have a central at the center.

Stride Convolutions

$n \times n$ image $f \times f$ filter

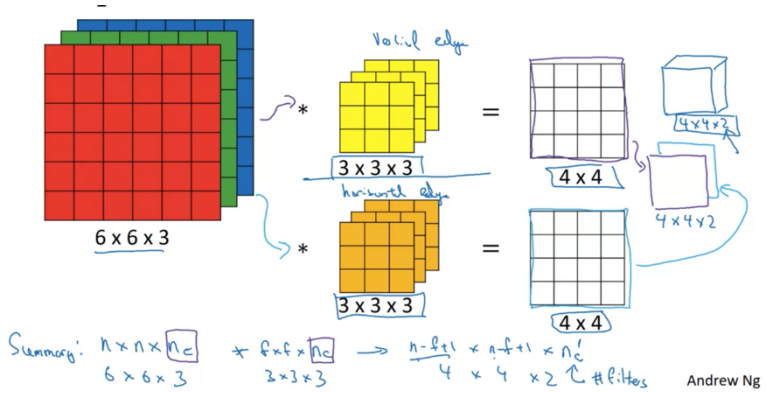
padding p stride s

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

Sometimes we want to make our output image smaller, so we stride our input images by move two steps in row and col instead of one.
So the output image size will be the left shown equation.
Sometimes we get the fraction not an integer, so we use the bracket as a notation meaning get the floor of the fraction

Convolutions over Volume

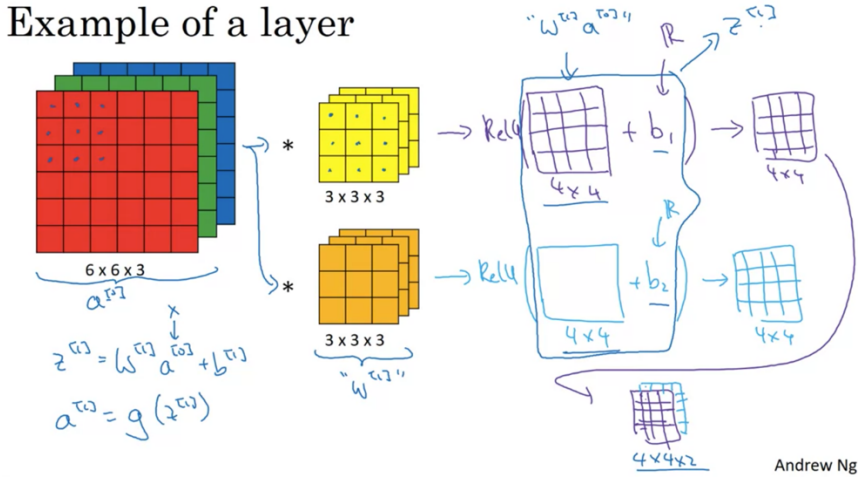
If we want to find edges over RGB images, ie, 3d images, we need to add volume to our edge detection
Similar as before, the only difference right now is the dimensions
nxnxn_c * fxfxn_c -> n-f+1 x n-f+1
note that we must have filter and original images have the same number of channels, n_c.
The output images could be 2d



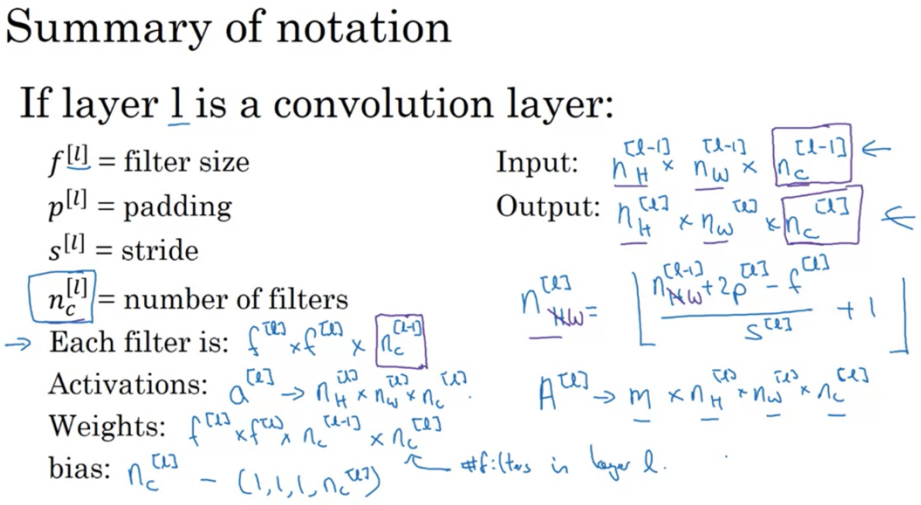
This technique allows us to use filter find the edges of different color. For example, if we want to find edges of red color, we can make all matrix entries equal to zero for n_B, n_G, the matrix for blue and green color
Also, if we want to find edges for different colors, we can make the output images more dimensional as well

The calculation is similar, only difference is when you done calculating the block multiplies by the filter for each channel, you **add them up** to be the one entry of the output

One Layer of Convolutional Network



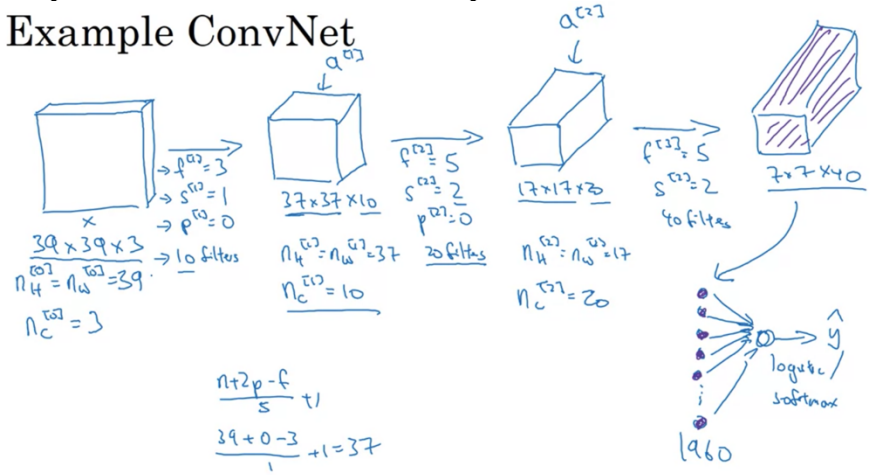
Here, we see an example of how one layer of CNN works.
We take the input image as $a^{[0]}$, multiply it with the filters, so all filters combined together work as the weights. Then we apply the activation function for the output image plus bias. The output image of input images multiply by the filters and plus bias works as $Z^{[1]}$ in our layers. After the activation function, we have output images forming together as 4x4x2 in this example, because we have 2 filters.



This is how we transform the input to output size of width and heights

Note, the output number of channel is different from the input, it equals to the number of filters
While each filter has the number of channels equals to the input number of channels.
The activation result has number of channels same as output channels because it takes input as Z.
If we use mini-batches in our CNN, we will have the size of batch m times the dimension we have for activation right now.

Simple Convolutional Network Example



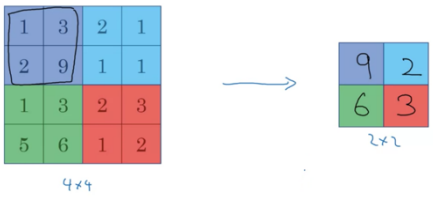
This is a 3-layer example of ConvNet. With input image as 39x39x3 RGB image, with 10 filters and 3 channels.

For the first layer, we have filter size as $f^{[1]}$ = 3x3, stride size as 1 and no padding, so the output size for the second layer would be (39+0-3)/1+1 = 37 x 37 x 10 because there are 10 filters.
For the last layer, what we did is we take the 7x7x40 matrix reshape to 1960 x 1 and feed it into the output activation function, such as logistic and softmax, then we get y_hat
As we building a deeper CNN, we begin with a larger image, and the size of the image will shrink as more layers built. And the number of layers will explode. We can tune the parameters f, s, and p for better performance

There are three types of layers in CNN:

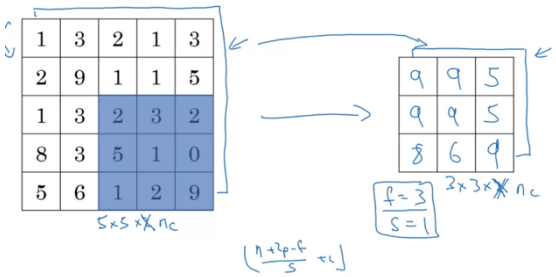
- Convolution (Conv)
- Pooling (Pool)
- Fully Connected (FC)

Pooling Layers



A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down sample the feature maps. This has the effect of making the resulting down sampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase "local translation invariance."

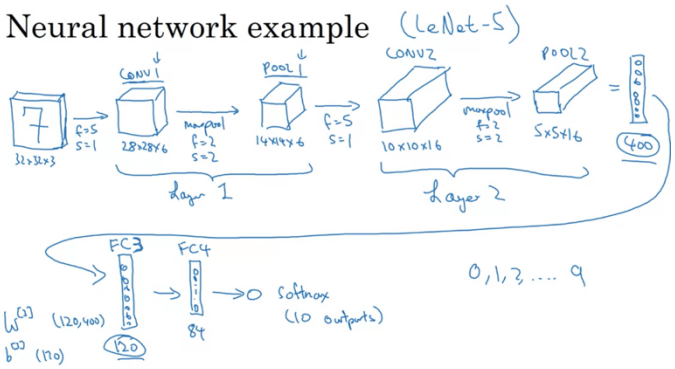
Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map



An example of max pooling:
the output matrix size = 5+2*0-3/1+1 = 3
the 1st row 1st col in output matrix =
max(1+3+2+9+1+1+3+2) = 9, and since s = 1, so we calculate the next entry by moving the square by one col
if we have multi channels, we calculate the pooling by each nc channels separately

Hyperparamters: f(filter size), s(stride), types of pooling(max,average), and there is no parameters to lean in Pooling

A complete CNN Example



we usually count layers with weights, so since layer for pooling doesn't have weights or parameters, so we combine pooling and conv together as one layer

Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

typos:
208 should be 608

416 should be 3216

48001 should be 48120
10081 should be 10164
841 should be 850

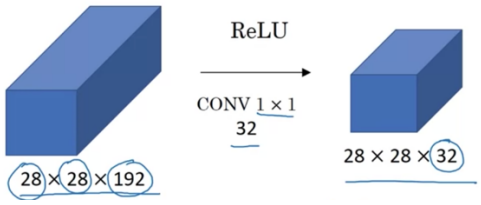
Calculation details:
Activation size = $\prod n^w, n^h, n^c$ -> 3072 = 32*32*3
$f^{[l]c} = n^{[l-1]c}$
#parameters for Conv1 = (5*5*3+1)*8 =($f^H * f^W * f^c$+bias)*n^c = 608
# of parameters for Conv2 = (5*5*8+1)*16 = 3216
# of parameters for FC3 = 400*120 +120 = 48120, since bias should have 120 parameters
# of parameters for FC4 = 120*84+84 = 10164
# of parameters for softmax = 84*10+10 = 850
# of parameters for Conv layer here are referring to weights+bias, and in CNN weights are actually all filters together, so for example, for Conv1, there are 5 filters, and each filter has size 5x5, so 5x5x3 + 1 bias, since there are 8 channels, each filter and bias should have same number of channels as well, so x8

Why Convolutional is Useful

Parameter Sharing (reduced #of features): A feature detector such as vertical edge detector that's useful in one part of the image is probably useful in another part of the image
Sparsity of connections (reduced overfitting): In each layer, each output value depends only on a small number of inputs
Translation Invariance: And that's the observation that a picture of a cat shifted a couple of pixels to the right, is still clearly a cat. And convolutional structure helps the neural network encode the fact that an image shifted a few pixels should result in similar features and should probably be assigned the same oval label. And the fact that you are applying to same filter, knows all the positions of the image, both in the early layers and in the late layers that helps a neural network automatically learn to be more robust or to better capture the desirable property of translation invariance.

One by One Conv Layer

While Pooling helps us shrink the width and height, 1x1 conv layer can help us shrink, keep the same or increase the number of channels.
One by One convolution is when we use size 1 filter in the conv layer.



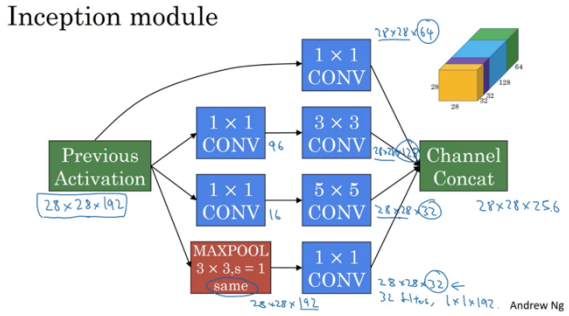
Inception Network

An inception network is a deep neural network with an architectural design that consists of repeating components referred to as Inception modules.

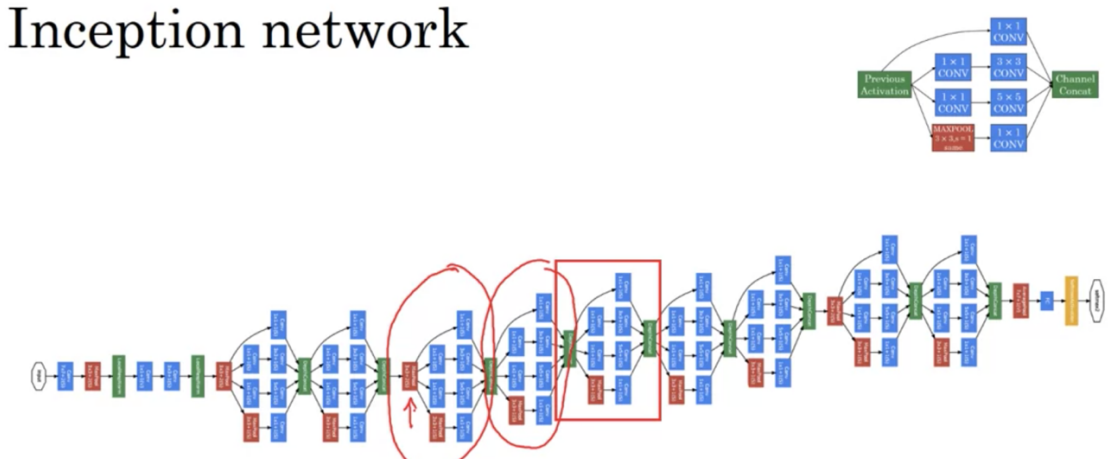
Benefits of the Inception Module

- High-performance gain on convolutional neural networks
- Efficient utilization of computing resource with minimal increase in computation load for the high-performance output of an Inception network.
- Ability to extract features from input data at varying scales through the utilization of varying convolutional filter sizes.
- 1x1 conv filters learn cross channels patterns, which contributes to the overall feature extractions capabilities of the network.

more details..



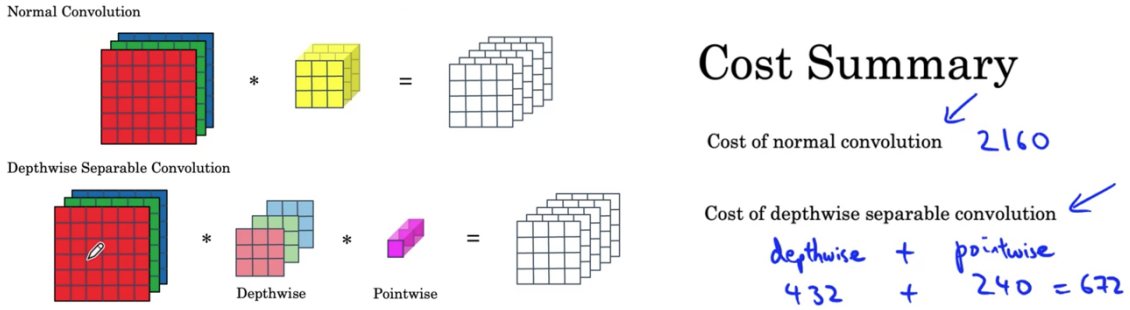
The Inception Module repeated multiple times in Inception Network. With even some side layers added



MobileNet

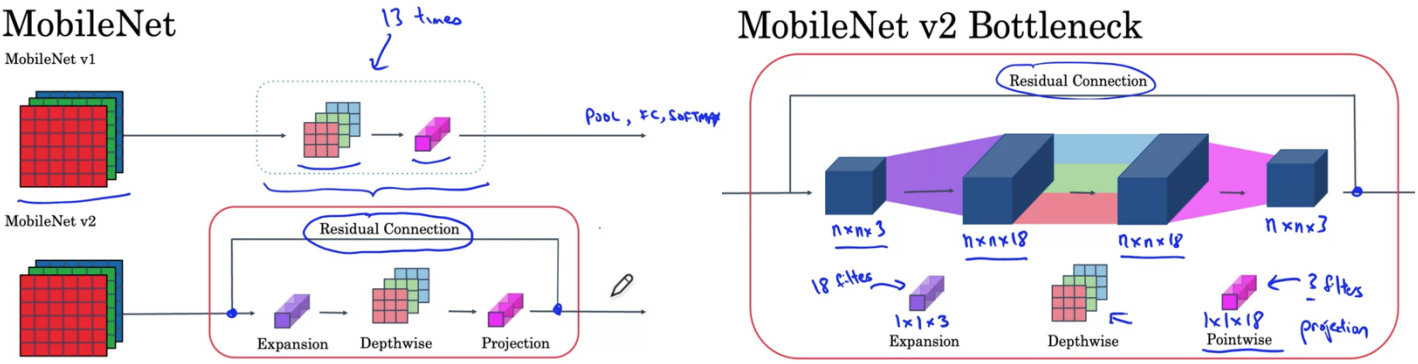
As the name applied, the MobileNet model is designed to be used in mobile applications, and it is TensorFlow's first mobile computer vision model.
MobileNet uses depthwise separable convolutions. It significantly reduces the number of parameters when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks.
MobileNet allows us to build NN even under low computational resources.

Compare Depthwise Separable Convolution with Normal Convolution



In DSC, we have two separable steps, Depthwise and pointwise, to combine as the convolution step.
Computational Cost Compare:
DSC = Depthwise +Pointwise
Depthwise = n_{out} x n_{out} x nc x fxf =#filter positions x #filters/channels x #filter params
Pointwise = = n_{out} t x n_{out} x nc' x 1x1 = #filter positions x #filters/channels x #filter params
So DSC = n_{out} x n_{out} x n_c x fxf + n_{out} x n_{out} x n_c,
Normally, the ratio of the cost of DSC compared to Normal Convolution = $1/n_c$, $+1/f^2$

MobileNet v2



In v2, we repeat the Residual connection, which called the Bottleneck several times.
In the Bottleneck, we first apply the **Expansion** to the input image, which expanded its channel/dimension, then we apply the **depthwise** step to the matrix with padding to keep the same dimension. Finally, we apply **Pointwise** step with expected number of channels for the output to be the number of filters, in this example it's 3. Since we shrink the dimension of the input in this step, so we can call this step as **Projection**
We increase the dimension of the image in the middle of the residual connection, to allow the NN learn richer and complex models, but we shrink the dimension after it to allow the amount of memory needed to store these values is reduced back down.

EfficientNet

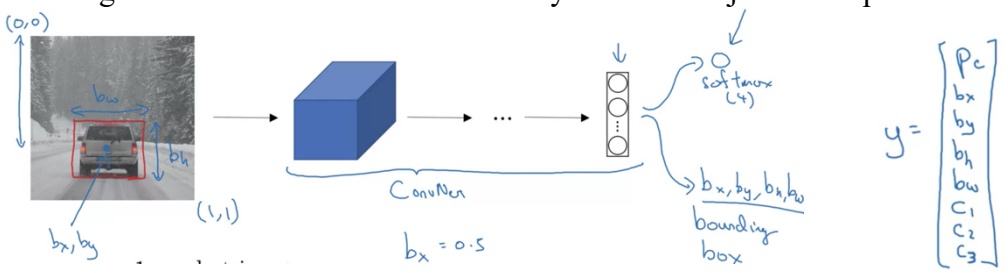
EfficientNet is a convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient.

Data Augmentation (in computer vision)
Mirroring, random cropping, rotation etc
And color shifting -> taking different values in RGB
We also have hyperparameters in data augmentation, such as which direction to rotate, how much to crop etc.
We usually use one thread to do data augmentation/distortions and one thread to train, and two threads can run parallelly.

What are localization and detection?

Image classification | Classification with localization | Detection

Lets begin with the situation where we only have one object in the picture



Normal image classification will have an input image, apply convNet layers and have one output layer to fed in nonlinear activations, such as Softmax.

If we want to have a box output around the object, we need to define parameters: bx, by, bh, bw

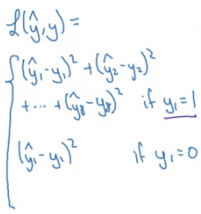
Where (bx, by) defines the center of the object, bh is the height and bw is the width

The output y consists [pc,bx,by,bh,bw,c1,c2,c3] assuming we have 4 classes to predict, and one of them denotes 'no image detected', if such case, we don't care about the other values in the output vector

pc: is there any object detected, ie, any classes belong to c1,c2,c3

if the object is predicted as c1, then c2=c3 = 0, c1 =1 (similar as OHE)

Define Loss function:

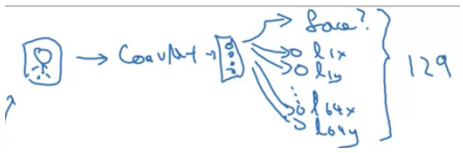


note that since when y1=0, means we didn't catch an object, so we don't care about the rest inputs, so the loss will only be the square error of y1

Landmark Detection

landmark detection is a computer vision task where we want to detect and track key points from an object.

For example, we can use the key points for detecting a human's head pose position and rotation. With that, we can track whether a driver is paying attention or not. Also, we can use the key points for applying an augmented reality easier. And there are so many solutions that we can generate based on this task.



Here, for example, if we want to locate 64 points around the eyes, we have l_{1x}, l_{1y} denotes the x,y coordinates for first point, and etc. so we will have 128 output units for landmark units, and one for telling you if this is a face or not. So in total 129 units for the output for this one example pic.

Object Detection

We use **Sliding Windows Detection** when we try to put a box on the object(s)

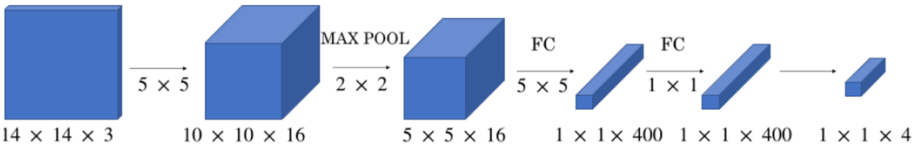
this algorithm is called Sliding Windows Detection because you take these windows, these square boxes, and slide them across the entire image
and classify every square region with some stride as containing an object or not. And we make the boxes bigger each time

Disadvantage: Computational Cost

If we increase the step size, the strides, or make the boxes bigger may help with the computational cost, but it may affect the performance.

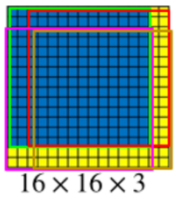
Convolutional Implementation of Sliding Windows

Assuming our development dataset is 14 x 14 x 3, and we output a 1 x 1 x 4 vector



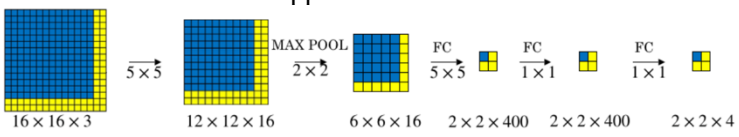
If in our test dataset, we get the image of size 16 x 16x3, so there are two ways:

1. Sliding window approach



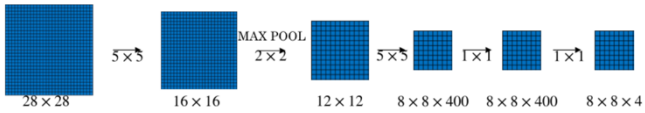
we pass through the 14x14x3 image size in the test image, and use ConNet to make prediction. Then move along to cover the whole image. So in this example, we need to predict 4 times of the output

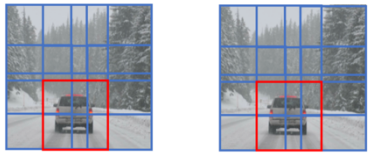
2. Convolutional Approach



We apply 400 5x5 filters during our fully connected layer to help we make the vector 3d. So rather than viewing these 400 as just a set of nodes, we're going to view this as a 1 x1 x400 volume.

Here, we output 2 x2 x4 image, the upper left entry represents the output of green box, the upper right entry represents the output of red box, the lower left box represents the purple box, and the lower right entry represents the yellow box. So the Convolutional approach enables us to predict the boxes in parallel.



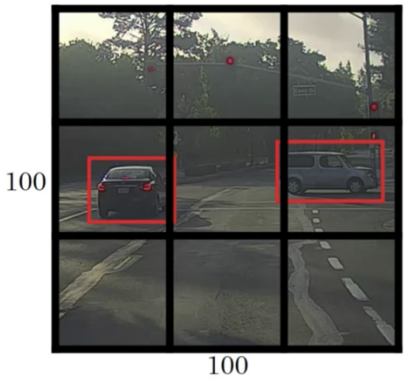


Let's understand this in a 28x28x3 input image. Since our dev set is 14x14, so our output should be 8x8(we can form 8 different box with size 14x14).

with this convolutional implementation that you saw in the previous slide, you can implement the entire image, all maybe 28 by 28 and convolutionally make all the predictions at the same time by one forward pass through this big convnet and hopefully have it recognized the position of the car.

Disadvantage: not accurate enough

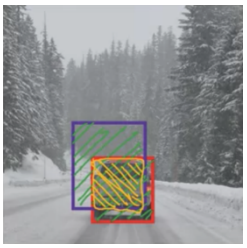
Bounding Box Predictions – YOLO(You Only Look Once)



This is a convolutional implementation, because we implement it once, instead of 9 times in the example.
For a 100 x 100 size input image, we create grids on it, for this example, we created a 3 x 3 size grids. So hopefully we catch object in some of the grids. We define if an object in a grid by its center, so even though some of the objects may intersect with multiple grids, it only belongs to one of the grids. We may increase the size of grids in practice so that can reduce the possibility of object crossing multiple grids.

- **Details in output**: For each grid cell, we output y = [p_c, b_x, b_x, b_y, b_h, b_w , c1, c2, c3].T, so in total 8 input units. Since there are 3x3 grid cells, the target output size should be 3x3x8, each 1x1x8 input unit represents the output of one grid cell. So, for each grid, we can know, if there is an object or not, if it is, where is the box and which class it belongs to.
- **Specify the bounding boxes**: b_x, b_y, b_h, b_w evaluated relative to the specific grid cell. The upper left point in the grid cell has coordinates (0,0) the lower right point is (1,1), so b_x, b_y should between 0 and 1, we will involving a sigmoid function to make sure of that, but b_h, b_w can be bigger than 1 when the bounding box exceeds the grid cell, ie, when the object covered in two grid cells, we will include an exponential function to make sure of nonnegativity.

Intersection Over Union



How do you tell if your object detection algorithm is working well? How do you know if the bounding box is good enough?
We can calculate the Intersection Over Union (IoU) value.

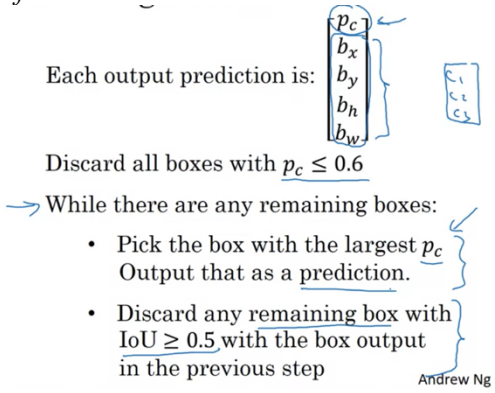
$$\text{IoU} = \frac{size\ of\ intersection\ of\ true\ bounding\ box\ and\ predicted\ bounding\ box}{size\ of\ union\ of\ true\ bounding\ box\ and\ predicted\ bounding\ box}$$

We say the predicted bounding box is 'correct' with IoU >=0.5 by convention

IoU is the metric evaluating the overlapping of two bounding boxes. It can be used to determine how similar two bounding boxes are.

Non-max Suppression

One of the problems of Object Detection as you've learned about this so far, is that your algorithm may find multiple detections of the same objects. Sometimes multiple grid cells will detect the center of the same object, remember we said we determine if an object belongs to a grid cell by its center. Non-max suppression is a way for you to make sure that your algorithm detects each object only once.
Let's see an example of how it works:



P_c denotes to the probability of the object is a car (in this example, we don't have multiple classes).
In the while loop, we pick the boxes with the largest probability, and remove those boxes with high similarity (IoU) with the boxes with largest probability. And we repeat it until there is one box left. If we have multiple classes, we then need to run the non-max suppression multiple times for each class.

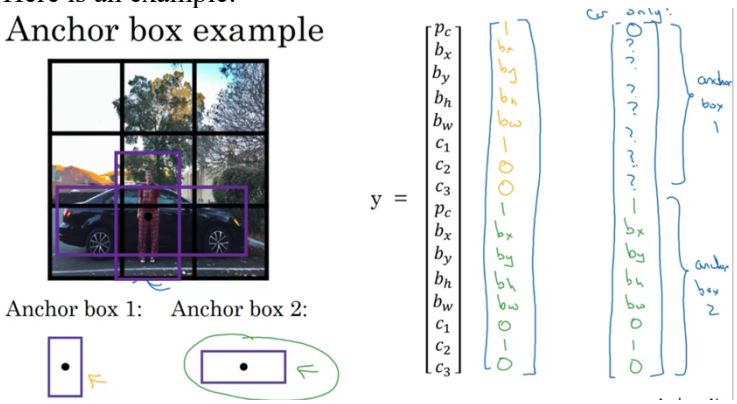
Anchor Box

Sometimes one grid cell will contain multiple objects' center, so that we will want to predict and locate multiple objects in one grid cell, our previous YOLO algorithm won't work.

What we will do then is to define multiple **Anchor Boxes** with different shapes in each grid cell, and assign (grid cell, anchor box) to an object based on the location of its center and the highest IoU of its bounding box and anchor box.

The output y then will contain multiple Anchor Boxes. For example, as in our previous example, output y has 8 output units, if we have two anchor boxes, y will have 2x8 = 16 output units

Here is an example:



the 2nd vector represents when we detect human and the car in our image, since human has similar shape with Ancho Box1, the yellow half represents the detection for the human so c1 =1, similarly, the green half represents the detection of the car so c2 = 1. But what if there is only car in the pic? So we will have no object with the highest IoU with the Anchor box1, then first pc will equal to 0

Maybe even better motivation or even better results that anchor boxes gives you is it allows your learning algorithm to specialize better. If your data set has some tall, skinny objects like pedestrians, and some white objects like cars, then this allows your learning algorithm to specialize so that some of the outputs can specialize in detecting white, fat objects like cars, and some of the output units can specialize in detecting tall, skinny objects like pedestrians.

Region Proposal – R-CNN

One drawback of sliding window, even with convolutional implementation, is it trains on grids with nothing there, like sky, air, etc. To solve this issue, R-CNN implements **Segmentation Algorithm** to figure out what could be an object.

Just to be clear, the R-CNN algorithm doesn't just trust the bounding box it was given. It also outputs a bounding box, B X B Y B H B W, to get a more accurate bounding box and whatever happened to surround the blob that the image segmentation algorithm gave it. So, it can get pretty accurate bounding boxes.

One downside of R-CNN is its speed, so there are more faster algorithms proposed later

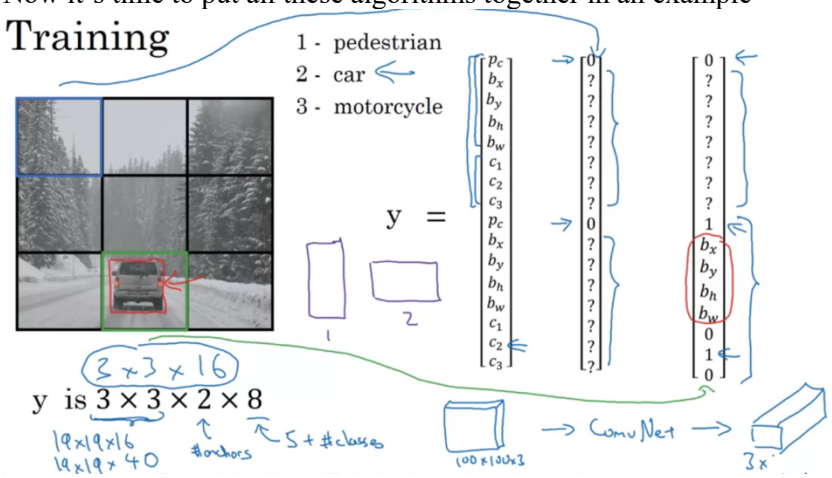
R-CNN: Propose regions. Classify proposed regions one at a time. Output label + bounding box.

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.

Put Them All Together..

Now it's time to put all these algorithms together in an example



We have an input image with shape 100x100x3, with 3x3 grid cells, and three classes to be predicted: pedestrian, car and motorcycle. With 2 anchor boxes, our expected output y would have the size 3x3x2x8 = grid cell size x # of anchor box x (5+#of classes)

To train this model, we put the image into ConvNet and output the 3x3x16 matrix.

If we have a grid that detects nothing, it will output with all pc for each anchor box equals 0, and the rest units as 'don't care'

During the one pass of forwards propagation, YOLO determines the probability that the cell contains a certain class. The equation for the same is :

$$score_{c,i} = p_c * c_i$$

The probability that there is an object pc times the probability that the object is certain class ci

The class with the maximum probability is chosen and assigned to that grid cell. Similar process happens for all the grid cells present in the image.

- let's say for box 1 (cell 1), the probability that an object exists is $p1$=0.60p1=0.60. So there's a 60% chance that an object exists in box 1 (cell 1).
- The probability that the object is the class "category 3 (a car)" is $c3$=0.73c3=0.73.
- The score for box 1 and for category "3" is $score1,3$=0.60×0.73=0.44score1,3=0.60×0.73=0.44.
- Let's say you calculate the score for all 80 classes in box 1, and find that the score for the car class (class 3) is the maximum. So you'll assign the score 0.44 and class "3" to this box "1".

Now, let's see how we make predictions

After our training, the next step is **non-max suppression**, it helps the algorithm to get rid of the unnecessary anchor boxes, like you can see that in the figure below, there are numerous anchor boxes calculated based on the class probabilities.

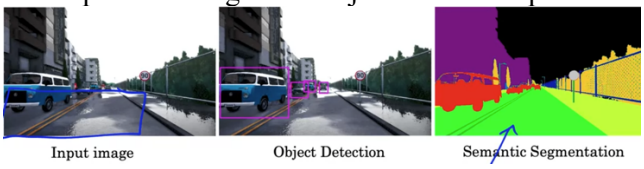
To resolve this problem non-max suppression eliminates the bounding boxes that are very close by preforming the IoU (Intersection over Union) with the one having the highest class probability among them.

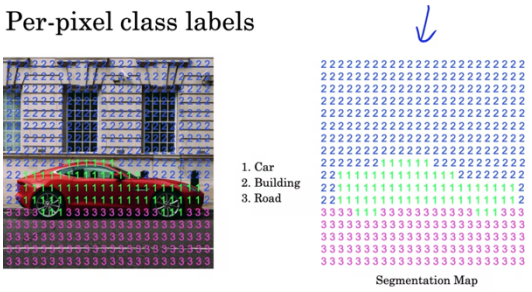
It calculates the value of IoU for all the bounding boxes respective to the one having the highest class probability, it then rejects the bounding boxes whose value of IoU is greater than a threshold. It signifies that those two bounding boxes are covering the same object but the other one has a low probability for the same, thus it is eliminated.

It is done until we are left with one bounding box for each object.

Semantic Segmentation with U-Net

The goal is to draw a careful outline around the object that is detected so that you know exactly which pixels belong to the object and which pixels don't.

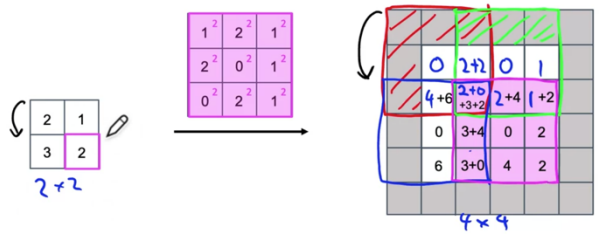




Taking the per-pixel labels and shifting it to the right, this is the output that we would like to train a unit table to give.
We want to identify each pixel which label should we give to it.

Transpose Convolution

Transpose Convolution is a key part in semantic segmentation. Unlike traditional ConvNet, instead of making the image smaller, the transpose convolution can explode an image.



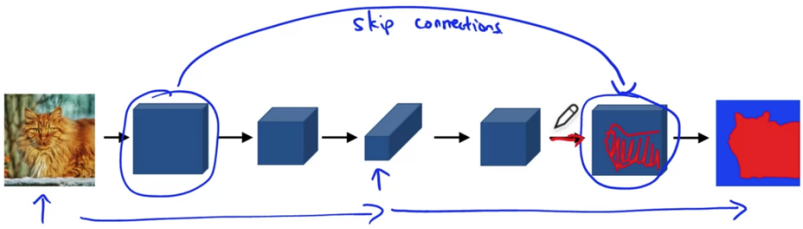
We will explain the calculation with this example, with input is 2x2 and we want to output a 4x4 matrix
Apply 3x3 filter, strides = 2 and padding = 1

For each element i in the input matrix:

1. Multiply each element in the filter with i
2. Take this 3x3 multiplied matrix to into the output (padded) matrix, and ignore the padding part
3. Take two steps right/down (strides = 2) when apply the next i
4. If there is overlapping, sum values together

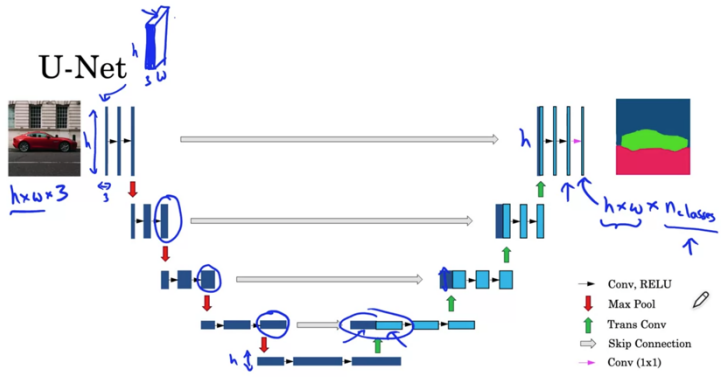
The elements in the filter are learned by the training algorithm

U-Net Architecture



Here is a rough diagram of the NN architecture for Semantic Segmentation. For the first half, we apply normal CNN , which transfer from very large image to one where the height and width for activation is much smaller. You will lose a lot of spatial information because shrinkage of the size, but it's much deeper. For example, the middle matrix may look like there's a cat roughly in the lower right-hand portion of the image, but we lost the details of the cat's fur and etc.
The second half is **transport convolution**, which is used to blow the representation size up back to the size of the **original input image**. So we blow up the width and height, but shrink the depth.
There is one more step we need to add – **skip connection**. We add the very beginning representation to the last one. What this does is it allows the NN to take this very high-resolution, low-level feature information (the beginning representation) where it could capture for every pixel position, like how much fairy stuff is there in this pixel. And used to skip connection to pause that directly to the latter layer. And so the last representation has the high level, spatial, high level contextual information (last representation), as well as low level but detailed texture information to make a decision as to whether a certain pixel is part of a cat or not.

How it works?



The left part is traditional ConvNet. We shrink the size of our input image but blow the depth. While when we reach the bottom, we scale up the size of our image using Trans Conv to match up every step of the Conv. During our Trans Conv steps, we copy the output of every step in Conv and do Conv to the combined output. During the final step, we use 1x1 Conv to scale the depth so the final output has the depth = # of classes we predict

Face Verification : (is this the claimed person?)

- Input image, name/ID
- Output whether the input image is that of the claimed person

Face Recognition: (who is this person)

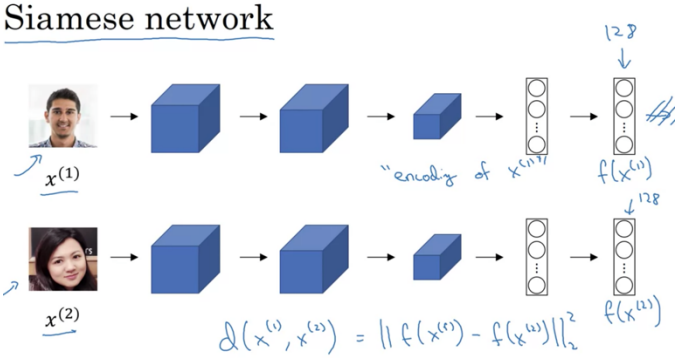
- Has a database of K persons
- Get an input image
- Output ID if the image is any of the K persons / not recognized.

One-Shot Learning

Learn from one example to recognize the person again.
In such case, one example is not enough to train a Deep NN. So we want to learn **a 'similarity' function**
Let **d(img1, img2) = degree of difference between images**
If $d \leq \tau$(some threshold), we claim it's the same person, otherwise not.
This approach helps us solve the problem when a new person come in the database, we don't need to run the whole model again, we just need to calculate the degree of difference between two images.
But how do we calculate the difference?

Siamese Network

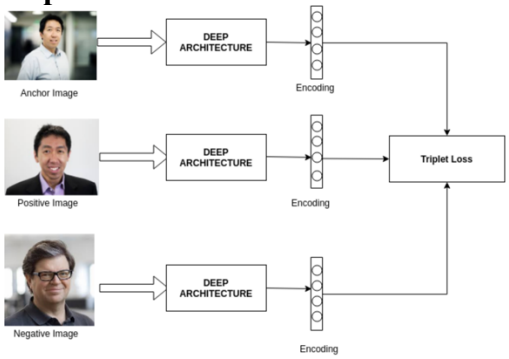
Traditionally, a neural network learns to predict multiple classes. This poses a problem when we need to add/remove new classes to the data. In this case, we must update the neural network and retrain it on the whole dataset. Also, deep neural networks need a large volume of data to train on. SNNs, on the other hand, learn a similarity function. Thus, we can train it to see if the two images are the same. This enables us to classify new classes of data without training the network again.



We will train the input images with the same parameters and output with 128 units for each person, say $f(x^{(1)})$ and $f(x^{(2)})$
Calculate the $d(x^1, x^2)$ based on the Norm.
So the goal of this network would be learn parameters so that, if x^1, x^2 are same person, the norm should be small, otherwise large.

But how do we define an objective(loss) function to achieve this?

Triplet Loss



Our goal is to let d(A,P) <= d(A,N), but how do we prevent NN to assign all output to be zero so that two degrees equal to each other?
We should add a margin to the equation

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

Given three images input, A, P, N, we will have the Loss function:

$$L(A,P,N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$$

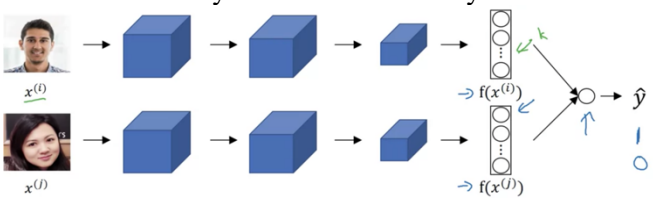
With this loss function, when our degree of difference negative, which achieves our goal, it will output 0, but when it is positive, it will output a positive number, so we know it does not met our expectation.
The Cost function would be:

$$J = \sum_{i=1}^{m} L(A^i, P^i, N^i)$$

Notice that our training set might be 10k pictures of 1k persons, that means, we have multiple pictures for each person, because we need that for training, but after we train the learning, we can input one pic of the person and the algorithm can output if it's verified.
And we need to choose triplets A, P, N that're hard to train on, ie, the pics are similar. And choosing triplets randomly can cost too much for computational resources, if we choose them on purpose, it can reduce our computation cost.

Binary Classification for Face Verification

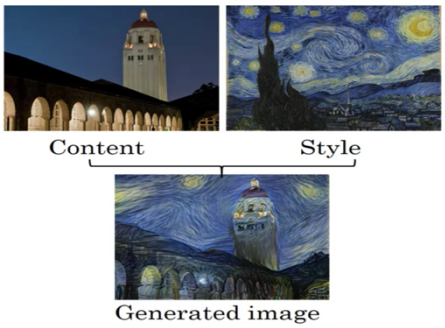
An alternative way to train the similarity function



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

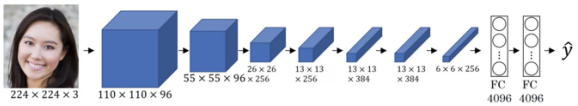
We have two samples (one pair) of images, and we fed them into our Conv Net (enconding), and calculated our y hat with logistic function to output 1 if two are the same person, otherwise 0.

Neural Style Transfer

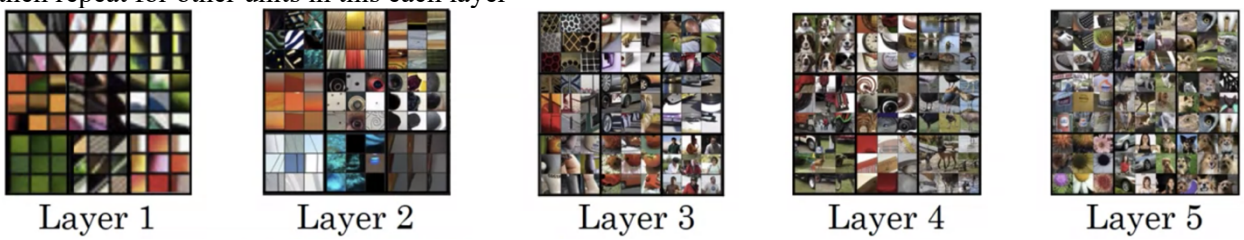


generated new image like the one below which is a picture of the Stanford University Campus that painted but drawn in the style of the image on the right.

What are deep ConvNet are learning



Lets say you are training a AlexNet like NN, but what are the layers learning exactly?
To visualize this, we can find out what are the images/patches that maximize the unit's activation. And then repeat for other units in this each layer



See in early layers, we are learning more simple patterns, like vertical lines, colors, and etc, but in later layers, our patches become larger, so that we can learn more sophisticated things, like human, dog and flowers.

Cost Function

To build a Neural Style Transfer system, let's define a cost function for the generated image, which can generate the style transferred imaged we want by minimizing the cost function

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

$$\alpha, \beta: weights\ for\ the\ two\ cost\ functions\ for\ content\ and\ style$$

G: Generated image. C: input Content image. S: input Style image

Steps to find the G we want:

1. Initiate G randomly, eg: 100x100x3
2. Use Gradient Descent to minimize J(G) -> $G := G - \frac{\partial}{\partial G} J(G)$
3. As you run gradient descent, you minimize the cost function J of G slowly through the pixel value so then you get slowly an image that looks more and more like your content image rendered in the style of your style image.

Content Cost Function

- 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

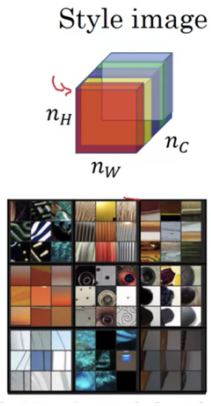
$$J_{content}(C,G) = \frac{1}{2}\left\| a^{[l](C)} - a^{[l](G)} \right\|^2$$

Normally, we choose layer to be not too shallow(will generate the image prone to C) nor too deep (will generate the image too away from C)

Style Cost Function

The first question is: How do we define and quantify style?

Say you are using layer l's activation to measure style, define style as correlation between activations across channels



Similar as we discussed what are we predicting in each layer , each unit. Maybe in this layer l, we have the red channel predicting the upper left neuron to figure what shape does this particular position has, and the yellow channel corresponds to the middle left neuron to figure if this position has orange color.

If two channel activations are highly correlated: whatever type of this image has this type of texture as in the upper left neuron, will have the similar orange color as in the middle left neuron.

so the correlation tells you which of these high level texture components tend to occur or not occur together in part of an image and that's the degree of correlation that gives you one way of measuring how often these different high level features, such as vertical texture or this orange tint or other things as well, how often they occur and how often they occur together or not in different parts of an image

If we use the degree of correlation between channels as a measure of the style, this tells you how similar of the style is your input image and your generated image.

Style matrix

Let $a^{[l]}_{i,j,k}$ = activation at (i,j,k). $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

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

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

"Gram matrix"

We compute $G^{[L]}$ for both the style image and generated image

We sum over the i and j because we calculate the correlation across all positions in the image.

Style cost function

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

$$\left\| G^{[l](S)} - G^{[l](G)} \right\|_F^2$$