4-1_Convolution neural network

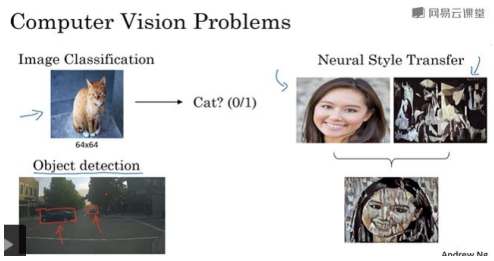
4.1 Computer vision

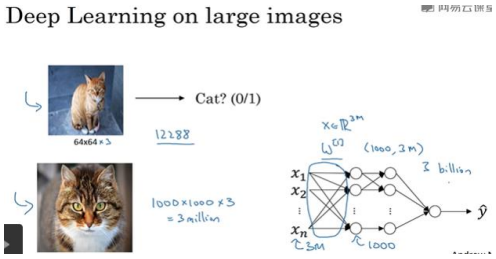
Computer vision is one of the areas that's been advancing rapidly thanks to deep learning. Deep learning computer vision is now helping self-driving cars figure out where the other cars and pedestrians around so as to avoid them.
Deep learning is even enabling new types of art to be created.

Two reasons excited about deep learning for computer vision :
>1. First, rapid advances in computer vision are enabling brand new applications to view, though they just were impossible a few years ago.
And by learning these tools, perhaps you will be able to invent some of these new products and applications.

>2. Second, even if you don't end up building computer vision systems per se,t because the computer vision research community has been so creative and so inventive in coming up with new neural network architectures and algorithms, is actually inspire that creates a lot cross-fertilization into other areas as well.

e.g: while working on speech recognition, actually took inspiration from ideas from computer vision and borrowed them into the speech literature





1. Examples

>1. Image classification
Input is like 64 x 64 image and try to figure out, is that a cat?

>2.object detection_ computer vision problem
e.g. while building a self-driving car, maybe don't just need to figure out that there are other cars in this image. But instead, need to figure out the position of the other cars in this picture, so that your car can avoid them.
In object detection, usually, not just figure out that these other objects but also draw boxes around them- they can be multiple objects/cars in the same picture.

>3. neural style transfer
Want a picture repainted in a different style.
So neural style transfer: have a content image, and a style image. And then have a neural network put them together to repaint the content image, but in a the style of another image.
So, algorithms like these are enabling new types of artwork to be created.

2. Computer vision problem:

The inputs can get really big:
e.g: small image size/feature: 64 x 64 x3=12288--->weight matrix dimension in the first layers: (n1, 12288)
while big image size/feature might be: 1000 x 1000 x3=3million--->first hidden layers, weight matrix dimension is (n1, 3million)
with so many parametes, it's difficult to get enough data to prevent neural network from overfitting, and the memory requirments to train in neural network with three billion parameters is a bit infeasible.

But for computer vision applications, we don't want to be stuck using only tiny little images. we want to use large images.
--->better implement the convolution operation, which is one of the fundamental building blocks of convolutional neural networks.

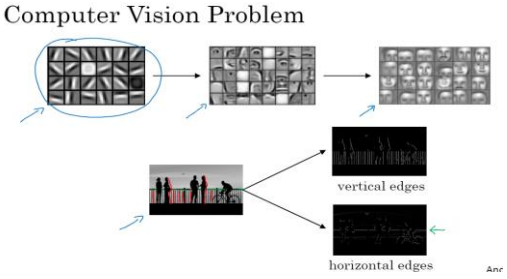
4.1-2 edge detection example

The convolution operation is one of the fundamental building blocks of a convolutional neural network

1. Compute vision problem

In previous videos, get intuition how the early layers of the neural network might detect edges and then the some later layers might detect cause of objects and then even later layers may detect cause of complete objects like people's faces.

this Item see how you can detect edges in an image.

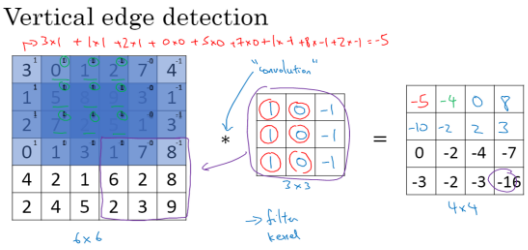


1.1 Edge detect

e.g: detect object in one image, first thing might detect vertical edges in this image:
vertical lines, where the buildings are, as well as kind of vertical lines idea all lines of these pedestrians and so those get detected in this vertical edge detector output.
And might also want to detect horizontal edges so for example, there is a very strong horizontal line where this railing is and that also gets detected sort of roughly here.

2. Vertical edge detection

E.g: grey image 6x6 -->detect vertical edge



>1.method: using a filter/kernal: to convolt the image: kernal : input * kernal

>2.Convolution calculation:
>>>1. do element-wise products between kernal and the areas covered in image
>>>2. Add up all the result,
>>>3. then move kernal horizentally step by step to get different horizental result
>>>4. and then move vertically step by step to get vertical result.

note:
1. '*' here menas convolusion, while in python only means element-wise mulitply.
2. Image , filter, and result matrix have different dimensions

3. Coding: use ' cov_forward' function in Tensorflow: tf.nn.conv2d
all the deep learning framework have good support for convolution fucntion for implementing this convolution operator.

e.g in the last: get to know the edge between light and deep dark area.

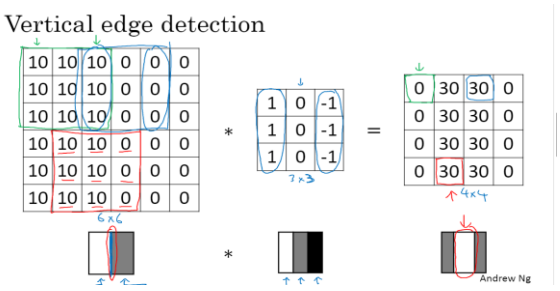
3. Vertical edge detection_example

filter is defined to detect vertical edge: three columns 1, 0, -1 , could visulize as coloer: bright, middel-grey, dark.
Input image: visulize as bright , middle-grey.
Conv. Result: grey-bright-grey: the light region corresponds to having detected the vertical edge in middle of input image. bright region in the middel of result visulizaiton means there is a strong vertical edge right down the middle of the image.

Note: detected edge seems really thick 30, due to working on a very small image . when using a larger image e.g 1000 x 1000, the conv. will do a pretty good job in detecting vertical edge.

one intuition to take away from vertical edge detection:
a vertical edge is a three by three region, since using a 3 x3 filter- where bright pixels on the left, do not care that much what is in the middle, and dark pixels on the right. The middle in this 6 x 6 image is really where there could be bright pixels on the left and darker pixels on the right and that is why it thinks its a vertical edge over there.

The convolution operation gives you a convenient way to specify how to find these vertical edges in an image.



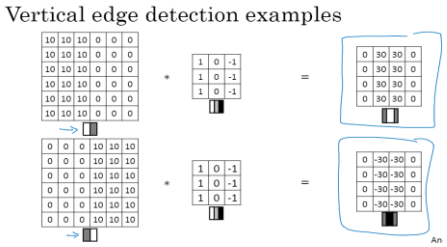
4.1-3 more edge detection examples

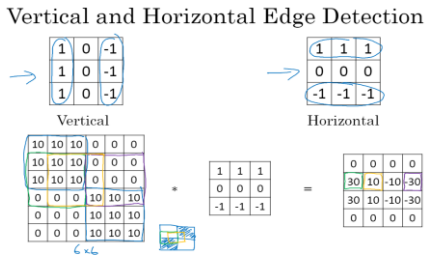
Item purpose: learn the difference between positive and negative edges, that is, the difference between light to dark versus dark to light edge transitions. And also see other types of edge detectors, as well as how to have an algorithm learn.

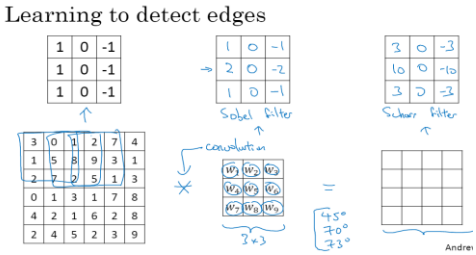
1. Vertical edge detection examples

eg.: image flipped: dark -->bright,
filter no change -->result image: get -30 (shows dark to bright rather than 30-bright to dark)in the middle: grey-dark-grey

if don't care which of these two cases it is, could take absolute values of this output matrix. But this particular filter does make a difference between the light to dark versus the dark to light edges.







2. Horizontal Edge detection:

a horizontal edge : is a 3x3 region where the pixels are relatively bright on top and relatively dark in the bottom row.

Input image: right top and left down part are bright , ohters dark.
Filter: horizental filter: 1; 0; -1: bright-edge -dark.
Result image:
30: corresponds to input image right top area-bright;
-30: correspond to input image left down area-dark;
10, -10: corresponds to transimittion area in the input image. when input image size is bigger, the result value for transmission will be smaller.

Summary, different filters allow you to find vertical and horizontal edges.

3. Detect filter

Sobel filter: 1, 2, 1; put a little bit more weight to central row (central pixl), make it a little bit more robust
Sharr filter: different properties . is vertical kernal, if turn 90 degree, get horizental kernal.

complicated pic edge: kernal /detect fitler number could not by hand-picked, treat these parameter instead learn by using back propagation.
may could learn not only vertical and horizontal edge, but also 45, 30, 71 degree edge, etc.

4.1-4 padding

1. Conv. cons

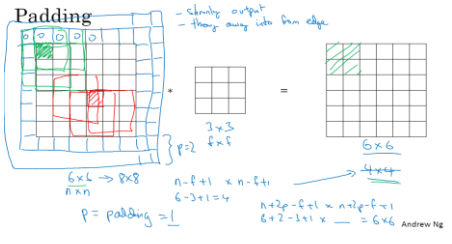
pic dimension: mxn, kernal dimension axb
-->Conv result image dimension: **(m-a+1) x (n-b+1)**, shrink than origianl image
cons:
1. reult shrink: may loose original feature
esp. in deep neural network, if shrink every layer, then end up with a very smaller image.

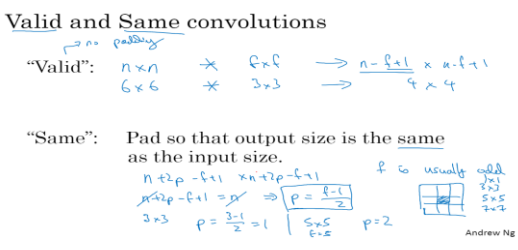
2. image corner used only once while middel pixel in image convoluted many times-->loose/throwing away many info. near the edge of image.

2. Padding

addressing:for convolution cons, padding image

>1. Padding effect:
padding pixels around edges (padding 0)-->increase result dimension: padding influence around pixels in result, which influence by input image corner.
-->this effective maybe not quite throwing away but counting less the information of the edge of a corner of the image is reduce.





3 .Padding method:

>1. 'valid' convolution: no pading

>2. Same convolution: add pading so output size is same as input size
-->padding size p=(f-1)/2

3.1 kernal size alway odd, rarely is even number in computer vision, as:-
>1. if kernal size is even, then need asymeetric padding: paddding more on left side,/right side.
when kernal size is odd, padding size will be symmetric and padd more natural: padding same size o left and right.
>2 odd size kernal has a central position/pixel, could talk about the positon of filter.
sometimes in computer vision its nice to have a distinguisher, it's nice to have a pixel, you can call the central pixel so you can talk about the position of the filter.

Probably get just fine performance even if use an even size kernal, but if stick to the common computer vision convention, usually just use odd number f for kernal size.

4.1-5 strided convolutions

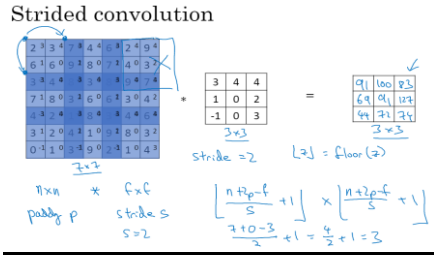
Strided convolutions is another piece of the basic building block of convolutions as used in Convolutional Neural Networks.

1. Strided conv.

filter/kernal moving in the input image, instead move one step , move 's' step every time.

convolution result size: (image size +2* padding size-kernal size)/strided step + 1

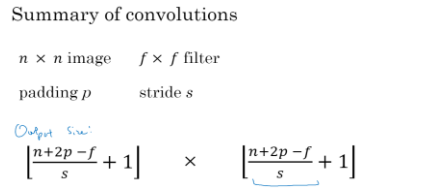
Note:
if result size calculation((n+2p-f)/s+1)is not integer: round down calculation (floor of result size), means: Implement principle: do conv computation only if kernal is fully contained within the image/image +paddding. if any part of kernal hangs outsidev of image+padding, do not convolution computation.



2. Summary of conv

Result image size/dimension: (n+2p-f)/s+1

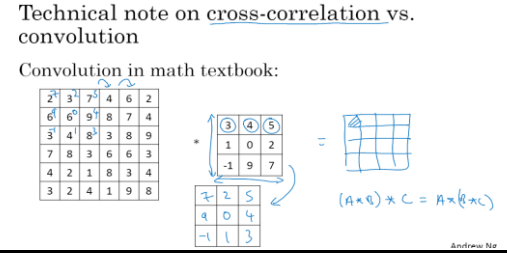
Note: we can choose all of these numbers (n, p, f,s) so that there is an integer result , yet sometimes don't have to do that and rounding down is just fine as well.



3. Techinical note:

convolution in math textbook:
before doing the element-wise product and summing, flip filer/kernal on the horizontal as well as the vertical axis.
offcial convolition is: image * flipped kernal (flipped vertically, and then horizentally-mirro vertical , then mirror horizental).
double flipping enable convoluation operator to enjoy associativity property: (A*B)*C=A* (B*C)

cross-correlation : image * kernal
note: in deep learning will not flipped kernal, so actually is doing cross-correlation convolution, but normally just call it convolution operator.



4.1-6 convoluition over volumes

1. Conv. On RGB images/volume

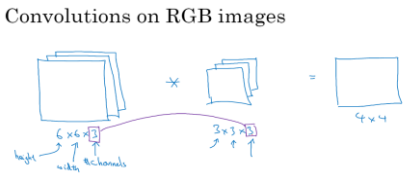
E.g : Detect features instead of on grey image but RGB image: a stack of three 2D images: n_h, n_w, n_c (channel number)

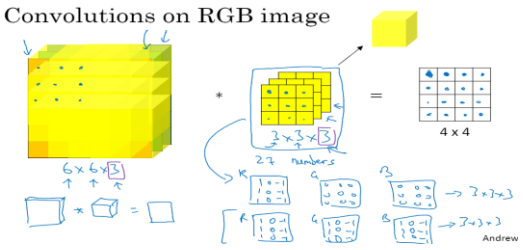
Filter: 3D filter: also have 3 layers/stack number, corresponding to input image: f x f x n_c (input image channel number): have same 3rd dimesnion/channel number as input image channel.

Result image: 2D: [(n_h +2p-f)/s+1, (n_w +2p-f)/s+1, 1 (only one layer-2D)]

Volume conv computation: element-wise product (over all channels) and sum; then move filter same as 2dimension , move kernal to right/down by stride step (3rd dimenson direction no move as cover all 3rd dimension data by kernal (same 3rd dimension size), in each step calculation

Note:
Filter design: could have different design/parameter for each channel.
e.g: if want filter to caputre edge only in the red channel of input image, could set edge detect parameter in the filter layer/channel that corresponding to red layer/channel of input image, and set to fitler layers that corresponds to input green, blue channel.
or if want to caputer edge while do not care color, then could set same fitler parameter in all filter layers/channels.-detect edges in any color of input image.





2. Multiple filters

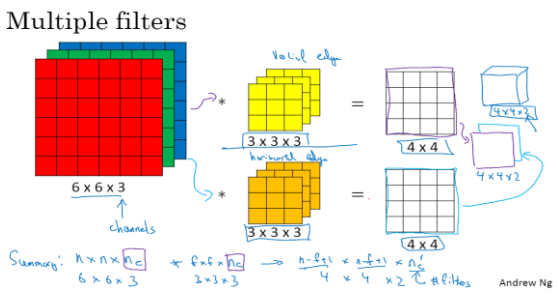
one filter could caputre one feature of input image, if want to capture other feature, use another filter--->use multifple fiters at same time

>2.1 Process:
>>>1. use multiple feature detectors/filters at same time;
>>>2. Stack each filter conv. Result (2D) together

2.2 Volume Result: 3D, 3rd dimension is filter/detector number.
same input with muilit kernal-->stack output together to get volume output
output channels number = feature numbers that are detecting.

2.3 Dimension:
size: input: n x n x n[c], kernal: f x f x n[c], padding: p, stride step: s
-->**output size**: [(n+2p-f)/s+1] x[(n+2p-f)/s+1]x **n[k] (kernal number)**

Question: could use different size filter? while stacking together , use padding?



4.1-7 one layer of convolutional network

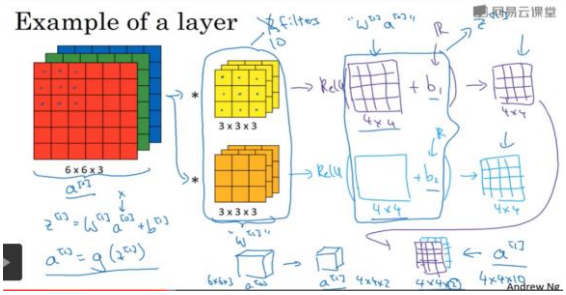
1. One layer of convolutiona neural network

>1. for each filter cov. result: take it as w*x result, add bias b (singal real number-broadcasting while adding to conv. result)-get Z[1], and then apply activation functiong--->final output for each filter A[1]
for i-th filter in neural layer 1:
Z[1]_i= conv result +b [1]_i
A [1] _i= g(Z [1]_i): g function here: just stack Z[1]_i together.
(each filter as one hidden units in l layer: compute Z: conv result + b-bias;
all filters: -->all hidden units in l layer;-->activation function stakcking all hidden units computation result together).

note: bias b [1]_i added for each filter could be different.

>2. Stack up all filter final output A [1] _i, get covolutional neural network output A[1].

Note: cov. function of input image and filters, plays a role similar to W[1]*A[0]-linear operation then add bias, apply activation funciton and output. --this is one layer of a covolutional net.



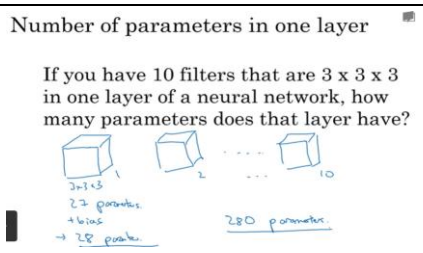
2. Parmaeter number in one layer

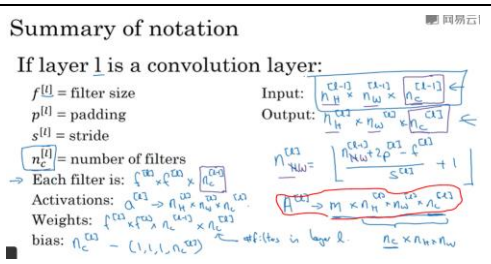
One filter paramter number: (fxfx n_c) + 1 (bias-singal number)
multi filer paramter: filter number x one filter paramter number

Note: parameter number not influenced by input image size. So once learned certain number feature detectors that work, you could apply this even to large images. And the number of parameters still is fixed and relatively small.
e.g: neural network with 10 filters of 3x 3 x3, paramters in ths neural network layer is [3x3x3 (filter parameter)+1 (bias)] *10=280, parameter number will not be influenced by later , even much larger image size-like 1000 x1000x 3.

Convolutional neural nets property: less prone to overfitting.
-->could use filter with small parameters to detect large size image feature--->make algorithm less prone to over fitting.

Summarize:
1. image channel number (last dimension size), same as filter /kernal channel-feature dimension size,
filter numbers (input features specifitly filted by each filter) same as neural network out put channel.
e.g: image color channel is 3, red, blue, green. filter will calculate on all these channels in each step to get 1 feature want to capture.->to consider input all channel in filter, filter need to have same channel number to calculate. and to capture different feature, different filter needed.-->each filter ouput 1 feature, neural network output features number =filter number.





3. Summary of notation:

if layer l is a convolution layer: dimension /note for parameters
inpute : n_h [l-1] x n_w [l-1] x n_c [l-1] (n_c [l-1]: number of channel of last layer output)
filter size: f[l]: f_l x f_l x n_c [l-1] (filter channel same as input channel)
padding: p[l]
stride: s[l]
filter number: n_c [l]
weights:f[l]: [f_l x f_l x n_c[l-1]+1] * n_c [l]
bias: b[l]_i (dimension = n_c[l]): one bias-real number for each filter; in coding for convinence : 1x1x1xn_c [l](four dimension)

output: n_h [l] x n_w [l] x n_c [l] **(output channel same as filter number-feature number)**

n_h [l] = [**n_h [l-1] + 2p[l] -f[l])/s[l] +1**]
n_c [l]: filter number in layer l.

Activation: A[l]: same dimenson: mx n_h [l] x n_w [l] x n_c [l]
m: ordering/index of training example
could search /index in this way, first look for sample ordanance, then check it's output.

Note:
there is no completely univeral standard covention about the ordering of hight, width, and channel.
e.g some may put channel first, and in fact in multiple common frameworks, there's actually a variable or a parameter where do you want to list the number of channels first, or list the number of channels last when indexing into these volumes.
Both of these conventions work okay, so long as you're consistent. And unfortunately maybe this is one piece of annotation where there isn't consensus in the deep learning literature but usually use this convention listed here in this course.

4.1-8 a simple convolutional network example

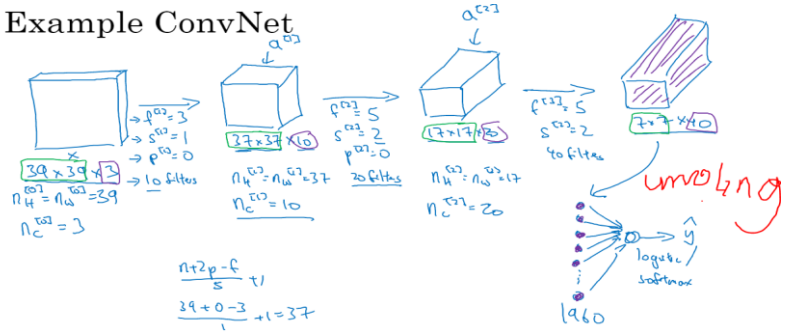
1. Convnet_Example

e.g: image classificaiton use Convnet
>1. inpute: 39x39x3;

>2. conv in layer 1:
filter in layer1 : 3x3x3; ten filters; s=1, p=0
--> layer1 output: 37 x37 x10

>3. Conv in layer2:
filter in layer 2: 5x5 x10, s=2, p=0, 20 filters
-->layer2 output: 17x17 x 20

>4conv in layer3-:
filter : 5x5x20, s=2, p=0, 40 filters
--->layer 3 outptu: 7 x 7 x 40
compute 7x7x40 =1960features of image 39x39x3



>5.unroll/flatten output into one vector (1960,1)--->feed to activation function
activation function: logistic or softmax function (depending on trying to recognize car or not car, or recognize any one of k different objects)
>6. Get final output y^ from activation function.

Note:

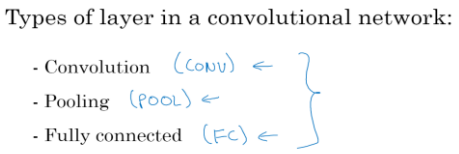
1. A lot of the work in designing convolutional neural net is selecting hyperparameters like: deciding filter size, stride step, padding size and how many filters to use in each layer.

2. General trend in a lot of convolutional neural networks.:
Conv output size decreasing with layer become deeper , while channel increase bigger:
As go deeper in a neural network, the conv result image/visulization's height and width will stay the same for a while and gradually trend down as go deeper in the neural network.
Whereas the number of channels will generally increase.

2. types of layer in convolutional network:

In a typical ConvNet, there are usually three types of layers:
>1. Conv layer
>2. Pooling layer
>3. Fully connected layer- FC.

And although it's possible to design a pretty good neural network using just convolutional layers, most neural network architectures will also have a few pooling layers and a few fully connected layers.
which are a bit simpler than convolutional layers to define.



4.1-9 pooling

Other than convolutional layers, ConvNets often also use pooling layers to: 1. reduce the size of the representation, to speed the computation; 2. as well as make some of the features that detects a bit more robust.

1. Plooling layer: Max pooling:

Max pooling: each of the outputs will just be the max from the corresponding reshaded region in the input image.

E.g: right : input 4x4, filter 2x2, s=2--->result: 2x2
each result corresponds the max of reshaped region-covered by fiter, in the input image.

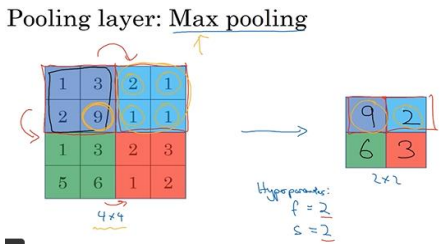
1.1 Intuition of max pooling:
If take each pixel of input image as some set of features or the activations in some layer of the neural network, then a large number in the pixel means that it's maybe detected a particular feature.
e.g: the upper left-hand quadrant (象限) has this particular feature-max value 9. It maybe a vertical edge or maybe a eye, etc. . Clearly, that feature exists in the upper left-hand quadrant.
Whereas this feature in upper right quadrant, maybe is cat eye detector-max value 2, it doesn't really exist in the upper right-hand quadrant.
(note: normally conv followed by pooling:
conv result: each layer result is one filter conv result, capturing one feature of image-->followed by max pooling, to filter out the particular feature caputured by this one filter in image.
max pooling works on each layer seperately, filter out particular feature captured by each filter in image.)

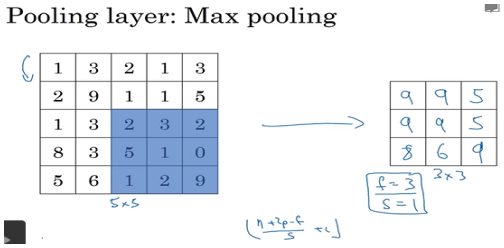
1.1 Intuition of max pooling:

Max operation function : as long as features are detected anywhere in one of these quadrants , it then remains preserved in the output of max pooling.
if these features detected anywhere in this filter, then keep a high number. But if this feature is not detected, so maybe this feature doesn't exist in the upper right-hand quadrant. Then the max of all those numbers is still itself quite small.
So maybe that's the intuition behind max pooling.

note:
max value in one pixel of image could be many pixel's value in pooling result: as this input pixel will be involved in max pooling many times.-below eg. value 9 occure many times in different pixels of result.

Main reason people use max pooling is because it's been found in a lot of experiments to work well, and the intuition above is the fancy real underlying reason that max pooling works well in confidence.





1.2 . Max pooling property:

> Parameters:
Only Have hyperparameters, size f, stride step s. yet no parameters to learn-nothing for gradient descent to learn.
once fix f, & stride step, it is a fixed computation and gradient decsent change nothing.

max pooling result size: same computatio as conv. : (n+2p-f/s) + 1.

1.3 Max pooling on volume image:

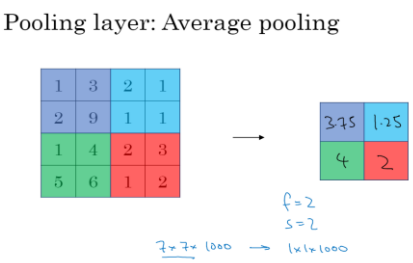
> Max pooling out has same dimension on the 3rd direction, as .input image , when it is 3D dimension
pooling output channe number: =image channel number

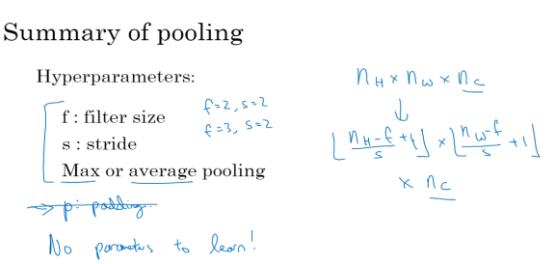
>Computation: comput max pooling on each input channel independently.

1.4. Summarize:
max pooling filter have no channel: calculate image of each channel by same filter, and tacks output together.
while in convolution neural network outupt has no channel for each filter (final output tack all filters' output together, final output channel number =filter number)

> Pooling :
image with channels C--> filter without paramter (once f size and stride step fix), works on each channel seperately-->tack isolate channel output -->final output channel C
(filter out feature captured in each layer-if this layer result is conv result, that means to filter out /find max feature caputured by each fiter)

>convolution:
image with channels C-->filter with same channel-->output only 1 channel.
(each filter works on all input layers, to capture one input feature: by computing on filter covered input region through all channels.
note: while computing image style matrix: element wise same postion of two channels and then sum-squared matrix: channel number x channel number)





2. Pooling layer: Average pooling

instead of taking the maxes within each filter, take the average.

max pooling is used much more often than average pooling with one exception, which is sometimes very deep in a neural network: might use average pooling to collapse your representation from e.g: 7 x 7 x 1,000. An average over all the the spacial extents to get 1 x 1x 1,000.

3. Summary of pooling

3.1 hyperparameters for pooling are:
1> f: the filter size
2> s: stride
Common choices of parameters might be f=2, s=2.
This is used quite often and this has the effect of roughly shrinking the height and width by a factor of above two.

3> the other hyperparameter is: using max pooling or average pooling

4> padding
Very rarely used when you do max pooling, usually, you do not use any padding, although there is one exception that we'll see next week as well.

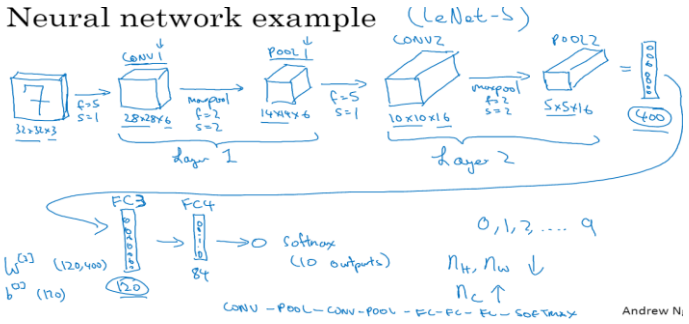
Size:
Input volume size: N_H x N_W x N_C
Output volume size: ((N_H-f/s) +1) x ((N_w-f/s) +1) x N_C.

Note:
1. input channels number = output channels number-because pooling applies to each of your channels independently.
2. In pooling there are no parameters to learn.
So when we implement that crop, you find that there are no parameters that backdrop will adapt through max pooling.
3. In pooling there are just above hyperparameters, that set once, maybe set by hand or set using cross-validation
And then beyond that, you are done. Its just a fixed function that the neural network computes in one of the layers, and there is actually nothing to learn.

4.1-10 convolution neural network example

1. Neural network example_(Inspiered by LeNet-5)

e.g: recognize image number



1.1. Layers
>1. input image: 32 x32 x3

>2: layer 1
> Conv_1: f=5, s=1, 6 filter-->output 28x28 x6
>. Max pooling_1: f=2, s=2, 6 filter--->output 14 x 14 x 6

>3. Layer2
>Conv_2: f=5, s=1 1 filter-->output 10 x 10 x1
>Max Pool_2: f=2, s=2, 16 filter--->output 5 x5 x 16

>4. Flatten result into vector (400, 1)

>5. FC3: fully connected-->output vector (120,1)
Fully connected layer just like the single standard neural network:
flattened vector (400,1) , 400 units densely connected to 120 unts, by weight matrix -dimension (120, 400)

>6. FC4: -->output (84, 1)
feed to activation function.

>7activation function with softmax:

1.2 Note:

1. two convention of defining one layer:
> One convention: convn + pool together is one layer.
> 2nd comvention: conv itself is one layer, pooling layer as one layer.
When people report the number of layers in a neural network usually people just record the number of layers that have weight/parameters. And because the pooling layer has no weights/ parameters, only a few hyper parameters, here use a convention that Conv 1 and Pool 1 shared together, treat that as Layer 1. (when I count layers, I'm just going to count layers that have weights)

2. Guideline for Hyper parametes:
seems like there a lot of hyper parameters, one common guideline is to actually not try to invent your own settings of hyper parameters, but to look in the literature to see what hyper parameters work for others. And to just choose an architecture that has worked well for someone else, and there's a chance that will work for your application as well.

1.3 Activation shape, size and paramters

List in the table , for each layer, the activation/input shape, size, and parameters in neural network

>Activation, a0 has dimension 3072. Well it's really 32 x 32 x 3. And there are no parameters

>Max pooling layers don't have any parameters;

>conv layers tend to have relatively few parameters
And in fact, a lot of the parameters tend to be in the fully collected layers of the neural network.

> activation size tends to maybe go down gradually as you go deeper in the neural network.
If it drops too quickly, that's usually not great for performance as well.
e.g: starts first there with 6,000 and 1,600, and then slowly falls into 84 until finally you have your Softmax output.

A lot of ConvNets will have properties will have patterns similar to above.

1.2 Note:

3. Layer l output:
as go deeper in the neural network, usually layer out size height and width will decrease, whereas the number of channels will increase.

4.Another pretty common pattern in neural networks:
pattern: normaly : cov layer (followed by pooling)+conv+pooling +,.....+FC +activation function (e.g. softmax)

To have conv layers, maybe one or more conv layers followed by a pooling layer, and then one or more conv layers followed by pooling layer. And then at the end you have a few fully connected layers and then followed by maybe a softmax. And this is another pretty common pattern you see in neural networks.

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

Summary:
So basic building blocks of neural networks is , convolutional neural networks, the conv layer, the pooling layer, and the fully connected layer.

A lot of computer division research has gone into figuring out how to put together these basic building blocks to build effective neural networks. And putting these things together actually requires quite a bit of insight. I think that one of the best ways for you to gain intuition is about how to put these things together is see a number of concrete examples of how others have done it. So what I want to do next week is show you a few concrete examples even beyond this first one that you just saw on how people have successfully put these things together to build very effective neural networks.

And through those videos next week I hope you hold your own intuitions about how these things are built. And as we are given concrete examples that architectures that maybe you can just use here exactly as developed by someone else or your own application.

4.1-11: why convolutions?

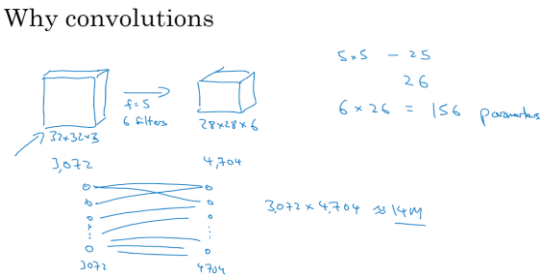
1 . Conv Advantage:

eg.: image 32x32x3 (3072)----: conv: f=5, 6 filters------>conv result 28 x28 x6 (4704)

if use standard neural network/fully connection-->weight matrix is (4704, 3072)-no problem today to train, but this is a quite small image and yet with a lot of parameters. if use larger image, then parameters will be infeasible large.

yet with conv layer: each filter have 5x5 +1-bias=26 paramters, 6 filter have 156 parameters. these small parameters due to:

>1. paramter sharing in conv:
a feature detector (like vertical edge detctor) that's useful in one part of image is probably useful in another part of the image.
e.g: if you've figured out 3x3 filter for detecting vertical edges, you can then apply the same 3x3 filter over here, and then the next position over, and the next position over, and so on. And so, each of these feature detectors, each of these outputs can use the same parameters in lots of different positions in your input image , in order to detect say a vertical edge or some other feature.
this should be true in low level feature, as well as high level feature -e.g maybe detecting the eye that indicates a face or a cat or something is there)
But being with a share in this case the same nine parameters of a filter to compute all 16 of these outputs, is one of the ways the number of parameters is reduced.



The second way that consonants get away with having relatively few parameters is by having sparse connections
>2. sparsity connection:
In each layer, each output value depends only on a small number of input.
(standard neural network: each input unit connected to layer output)

e.g: right pic, conv result first value 0 = filter * right corner region-9 values of input image, decided only by filter and part region of input -not all input image.
each output value in conv result, depend only on a small number of input(filter size)=element-wise and summ of filter and it's specified region in image. while standard NN, each output value/unit z[l]_i value depending on all input= w[l]_i.X

so, through these two mechanisms-sharing parameter and sparsity connection, a neural network has a lot fewer parameters which allows it to be trained with smaller training cells and is less prone to be over-fitting **(over fitting?)**

3. good at capturing translation invariance.
e.g: picture of cat shif couple of pixel to right, still pretty clear a cat.
convolutional structure helps neural network encode the fact that a picture shifted a few pixels should result in pretty similar feature, and should probably be assigned the same output label.
-->the fact that applying the same filter in all the positon of image, both in ealier layers and later layers, hopes network automaticaly learn to be more robust, to better capture the desirabl property of tranlation invariance.

2. Putting it all together and see how to train one of these networks

e.g: build a cat detector
have a labeled training sets: X is an image. y can be binary labels, or one of K causes.

And chosee a convolutional neural network structure:
inserted the image and then having neural convolutional and pooling layers and then some fully connected layers followed by a software output that then operates Y hat.
The conv layers and the fully connected layers will have various parameters, W, as well as bias's B.

Define a cost function similar to previous courses:
>>> Randomly initialized parameters W and B,
>>> Compute the cost J, as the sum of losses of the neural networks predictions on the entire training set.
>>> Train this neural network: use gradient descents or some of the algorithm like, gradient descent momentum, or RMSProp or Adam, or something else, in order to optimize all the parameters of the neural network to try to reduce the cost function J.

---> If do this, can build a very effective cat detector or some other detector

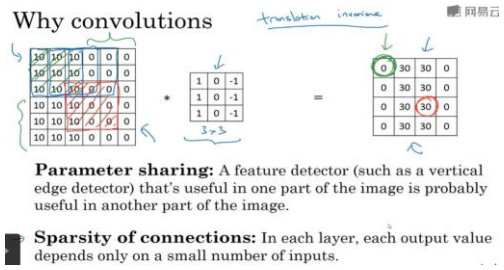
Summary:
This week have seen all the basic building blocks of a convolutional neural network, and how to put them together into an effective image recognition system.
There're just a lot of the hyperparameters in convolution neural networks, next week will show a few concrete examples of some of the most effective convolutional neural networks, so you can start to recognize the patterns of what types of network architectures are effective.
And one thing that people often do is just take the architecture that someone else has found and published in a research paper and just use that for your application. And so, by seeing some more concrete examples next week, you also learn how to do that better.

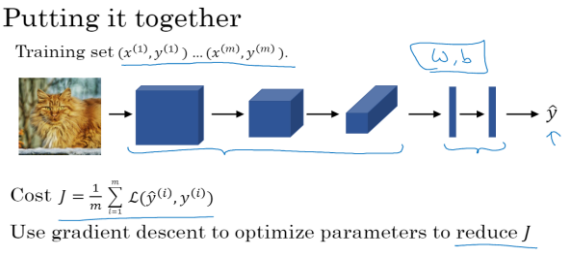
>1. paramter sharing in conv:
Intuition: The same feature seems like it will probably be useful in one position of input image, has a good chance of being useful for other positions of the image. --->So, maybe don't need to learn separate feature detectors for different postion of the image.
And maybe you do have a data set where you have the upper left-hand corner and lower right-hand corner have different distributions, so, they maybe look a little bit different but they might be similar enough, they're sharing feature detectors all across the image, works just fine.

Summarize:
1. Do not need different feature detector /filter /kernal in different posiltion of the image . different postion of image data distribution maybe different, but they might similar engough sharing feature detector all acrros image and works fine.

2. standard neural network: each input unit connected to layer output (by unique weight for each input unit), weight is huge :n[l]x n[l-1]; while conv paramter is only (filter h size x w size x filter channel +bias)x filter number, quite small quant (shared weight parameters for input units).

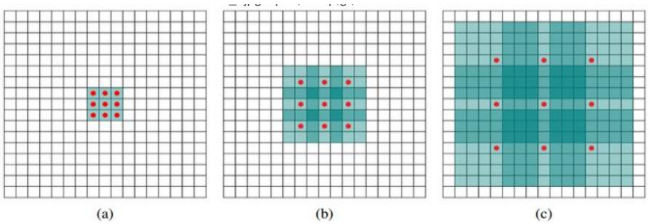
3. conv could use small parameter, as: same paramters can use in different image positioins





Note: 空洞卷积（dilated convolution）

https://blog.csdn.net/sinat_33741547/article/details/79451069?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216286459451678026541330 9%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=162864594516780265413309&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-79451069.pc_search_download_positive&utm_term=%E7%A9%BA%E6%B4%9E%E5%8D%B7%E7%A7%AF&spm=1018.2226.3001.4449



(a)图对应3x3的1-dilated conv，和普通的卷积操作一样，(b)图对应3x3的2-dilated conv，实际的卷积kernel size还是3x3，但是空洞为1，也就是对于一个7x7的图像patch，只有9个红色的点和3x3的kernel发生卷积操作，其余的点略过。也可以理解为kernel的size为7x7，但是只有图中的9个点的权重不为0，其余都为0。 可以看到虽然kernel size只有3x3，但是这个卷积的感受野已经增大到了7x7（如果考虑到这个2-dilated conv的前一层是一个1-dilated conv的话，那么每个红点就是1-dilated的卷积输出，所以感受野为3x3，所以1-dilated和2-dilated合起来就能达到7x7的conv），(c)图是4-dilated conv操作，同理跟在两个1-dilated和2-dilated conv的后面，能达到15x15的感受野。对比传统的conv操作，3层3x3的卷积加起来，stride为1的话，只能达到(kernel-1)*layer+1=7的感受野，也就是和层数layer成线性关系，而dilated conv的感受野是指数级的增长。

dilated的好处是不做pooling损失信息的情况下，加大了感受野，让每个卷积输出都包含较大范围的信息。在图像需要全局信息或者语音文本需要较长的sequence信息依赖的问题中，都能很好的应用dilated conv，比如图像分割[3]、语音合成WaveNet[2]、机器翻译ByteNet[1]中。简单贴下ByteNet和WaveNet用到的dilated conv结构，可以更形象的了解dilated conv本身。

Image size: h x w;
filter size: f xf;
padding: p;
stride: s
diate conv size: d （空洞数）

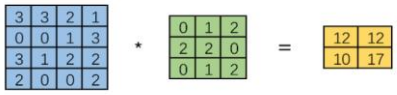
-->result size: { h +2p - (f+d* (f+1)) } / s +1
(standard conv filter + diate conv-->new filter size: f* (1+d) +d

转置卷积

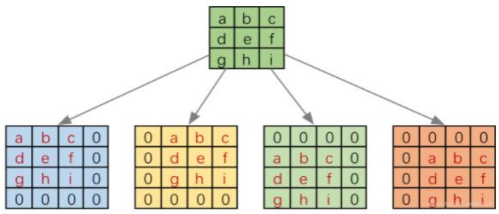
https://blog.csdn.net/tsyccnh/article/details/87357447?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216865030316780357272834%2522%252C%2522scm%2522%253A%2522220140713.130102334..%2522%257D&request_id=168650303167803572728 34&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-87357447.pc_search_download_positive&utm_term=%E8%BD%AC%E7%BD%AE%E5%8D%B7%E7%A7%AF&spm=1018.2226.3001.4449

普通卷积（直接卷积）

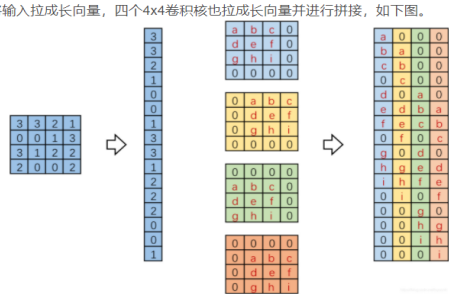
普通的卷积过程大家都很熟悉了，可以直观的理解为一个带颜色小窗户（卷积核）在原始的输入图像一步一步的挪动，来通过加权计算得到输出特征。 如下图。



但是实际在计算机中计算的时候，并不是像这样一个位置一个位置的进行滑动计算，因为这样的效率太低了。计算机会将卷积核转换成等效的矩阵，将输入转换为向量。通过输入向量和卷积核矩阵的相乘获得输出向量。输出的向量经过整形便可得到我们的二维输出特征。具体的操作如下图所示。由于我们的3x3卷积核要在输入上不同的位置卷积4次，所以通过补零的方法将卷积核分别置于一个4x4矩阵的四个角落。这样我们的输入可以直接和这四个4x4的矩阵进行卷积，而舍去了滑动这一操作步骤。



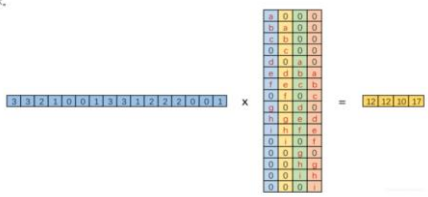
进一步的，我们将输入拉成长向量，四个4x4卷积核也拉成长向量并进行拼接，如下图。



我们记向量化的图像为 I ,向量化的卷积矩阵为C, 输出特征向量为 O
则有：

$$I^T * C = O^T$$

如下图所示。



我们将一个1x16的行向量乘以16x4的矩阵，得到了1x4的行向量。**那么反过来将一个1x4的向量乘以一个4x16的矩阵是不是就能得到一个1x16的行向量呢？** 没错，这便是**转置卷积**的思想。

转置卷积

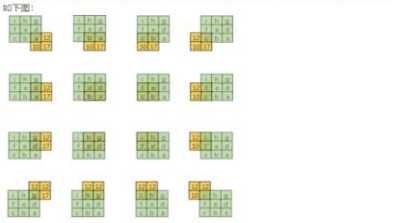
一般的卷积操作（我们这里只考虑最简单的无padding, stride=1的情况），都将输入的数据越卷越小。根据卷积核大小的不同，和步长的不同，输出的尺寸变化也很大。但是有的时候我们需要输入一个小的特征，输出更大尺寸的特征该怎么办呢？比如图像语义分割中往往要求最终输出的特征尺寸和原始输入尺寸相同，但在网络卷积核池化的过程中特征图的尺寸却逐渐变小。在这里转置卷积便能派上了用场。在数学上，转置卷积的操作也非常简单，把正常卷积的操作反过来即可。
对应上面公式，我们有转置卷积的公式：

$$O^T * C^T = I^T$$

如下图所示：

这里需要注意的是这两个操作并不是可逆的，对于同一个卷积核，经过转置卷积操作之后并不能恢复到原始的数值，保留的只有原始的形状。
所以转置卷积的名字就由此而来，而并不是"反卷积"或者是"逆卷积"，不好的名称容易给人以误解。

这是一个很有趣的结果，结合整体来看，仿佛有一个更大的卷积核在2x2大小的输入滑动。但是输入太小，每一次卷积只能对应卷积核的一部分。我们来把更大的卷积核补全，如下图：



这里和直接卷积有很大的区别，直接卷积我们是用一个"小窗户"去看一个"大世界"，而转置卷积是用一个"大窗户"的一部分去看"小世界"。这里有一点需要注意，我们定义的卷积核是左上角为a，右下角为i，但在可视化转置卷积中，需要将卷积核旋转180°后再进行卷积。由于输入图像大小，我们按照卷积核尺寸来进行补零操作，每边的补零数量显而易见是2，即3-1，这样我们就将一个转置卷积操作转换为对应的直接卷积。如下图：

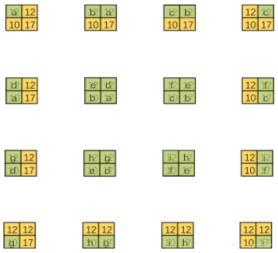
总结一下将转置卷积转换为直接卷积的步骤：（这里只考虑stride=1，padding=0的情况）
设卷积核大小为k*k，输入为方形矩阵

1. 对输入进行四边补零，单边补零的数量为k-1
2. 将卷积核旋转180°，在新的输入上进行直接卷积

形象化的转置卷积

但是仅仅按照矩阵转置形式来理解转置卷积似乎有些抽象，不像直接卷积那样理解的直观。所以我们也来尝试一下可视化转置卷积。前面说了在将直接卷积向量化的时候是将卷积核补零然后拉成列向量，现在我们有了一个新的转置卷积矩阵，可以将这个过程反过来，把16个列向量再转换成卷积核。以第一列向量为例，如下图：

这里将输入还原为一个2x2的张量，新的卷积核由于只有左上角有非零值直接简化为右侧的形式。对每一个列向量都做这样的变换可以得到：



4-2_Examples Convolution neural network

4.2-1 Why look at case studies

1. Why look at case studies
Last week we learned about the basic building blocks, such as convolutional layers, pooling layers, and fully connected layers of confidence. It turns out the past few years of computer vision research has been working on how to put together these basic building blocks to form effective convolutional neural networks. One of the best ways for user gain intuition is to see some of these examples.

It turns out that a neural network architecture that works well on one computer vision tasks often works well on other tasks as well, e.g: If someone else's train a neural network or has figured out a neural network architecture, there's very good at recognizing cats and dogs and people, But you have a different computer vision tasks like maybe you're trying to build a self-driving car, you might well be able to take someone else's neural network architecture and apply that to your problem.
After the next few videos, you'll be able to read some of the research papers from the field of computer vision, and I hope that you might find it satisfying as well.

2. Outline

2.1 Classic networks:

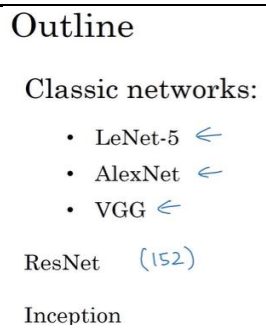
>LeNet-5 network : Came from 1980s
> AlexNet
which is often cited in the VGG network.
>VGG

These are examples of pretty effective neural networks, and you see ideas from these papers that will probably be useful for your own work as well.

2.2 : ResNet or called residual network
ResNet neural network trained a very deep 152 layer neural network. It has some very interesting tricks, interesting ideas how they do that effectively.

2.3 Inception neural network

After seeing these neural networks, I think you have much better intuition about how to build effective convolutional neural networks. Even if you end up not working computer vision yourself, you find a lot of the ideas from some of these examples, such as ResNet, Inception network, many of these ideas are cross fertilizing, are making their way into other disciplines. you'll find some of these ideas very interesting and helpful for your work.



4.2-2classic neural network:

1. LeNet-5

1.2 Summarize_LeNet-5:
1. Lenet5-trained on grey image, image has no channel.
2. at ealier time:
>1. avg pooling use more often , while now max pooling more often used
>2. Padding not used-->result shrink after conv

3. not use softmax as activation, but another one which not in use today.
4. Lenet 5 has 60,000 parameters.
5. Reuslt's height and width goes down while channel number goes up.
6. Pattern still quite common that: conv/several convs +pool+conv+pool+...+fc+fc

Note: only for paper reading:
1. Activation function used is sigmoid/tanh. Not Relu/softmax which use more often today.
2. Due to computation problem, did not use same filter for all input channels(conv filter channel /= input channel): different filters look at different channels of the input block.
3. Non-linearity (sigmoid) was used after pooling
this isn't really done right now

1. LeNet-5

1.1 Architecture_LeNet 5

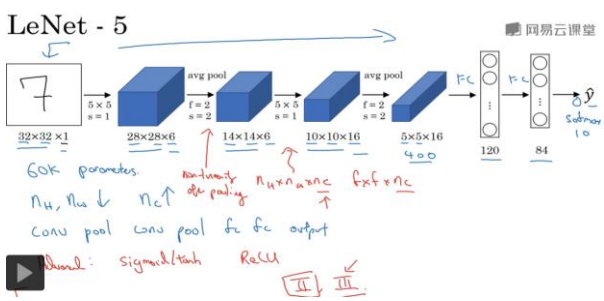
e.g: Recognize hand-written digits. Image: 32 x 32 x1
>Layer 1: conv + pooling
>>> conv1: filter 5x5, s=1, p=0, 6 filters------>result 28x28x6
>>> Avg pool: f=2, s=2 --->result 14x14x6

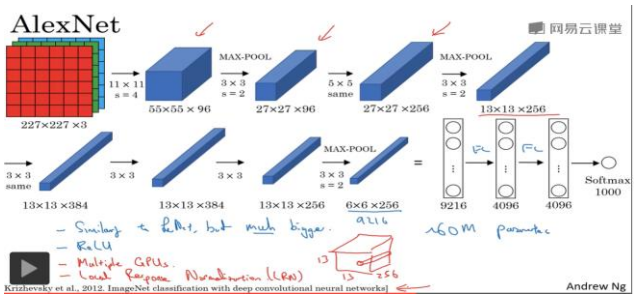
>Layer 2: conv +pooling
>>>conv2: filter 5x5,s=1, 16 filters---->result 10x10x16
>>>avg pool: f=2,s=2--->result 5 x 5 x 16.

>Flatten result to vector (400,1)

>Layer 3: FC: weight (400, 120)-->result vector (120,1)
>Layer 4: FC: weight (84, 120)-->result vector (84,1)

>Softmax-->prediction y^, 10 outputs probability





2. AlexNet

2.2 Compare vs LeNet:

Much better performance than LeNet , due to:
>1. similar to Lenet, but much bigger
LeNet has about 60,000 parameters, , while AlexNet has 60 Million parameters;
And the fact that they could take pretty similar basic building blocks that have a lot more hidden units and training on a lot more data, they trained on the image that dataset that allowed it to have a just remarkable performance.

>2. Relu activation function used

>>Had a complicated way of training on two GPUs
when this paper was written, GPUs was still a little bit slower, so it had a complicated way of training on two GPUs.
And the basic idea: was that, a lot of these layers (e.g layer 1, layer2, layer 3)was actually split across two different GPUs and there was a thoughtful way for when the two GPUs would communicate with each other.

>>Had Local Response Normalization layer. And this type of layer isn't really used much, which is why I didn't talk about it.
LRN basic idea: e.g take one blocks/layer result,e.g 13 x 13 x256, look at one position on the 2D 13 x 13, So one position height and width, and look down this across all the channels, look at all 256 numbers and normalize them.
LRN motivaiton: for each position in this 13 x 13 image, maybe you don't want too many neurons with a very high activation.
note: many researchers have found LRN doesn't help that much so this is one of those ideas I guess I'm drawing in red because it's less important for you to understand this one. And in practice, I don't really use local response normalizations really in the networks language trained today.

Before AlexNet, deep learning was starting to gain traction in speech recognition and a few other areas, but it was really just paper that convinced a lot of the computer vision community to take a serious look at deep learning to convince them that deep learning really works in computer vision. And then it grew on to have a huge impact not just in computer vision but beyond computer vision as well

AlexNet had a relatively complicated architecture, there's just a lot of hyperparameters.

--take pretty similar basic building block as LeNet and just have a lot more hidden units and trained on a lot more data, which trained on the image and data-->remarkable performance

2. AlexNet

2.1 Architecture

input image:color 227 x227 x3

>Layer 1: conv + pooling
>>> conv1: filter 11x11 s=4, p=0, 96 filters------>result 55 x 55 x 96
s=4, shrink image size 4 times
>>> Max pool: f=3, s=2 --->result 14x14x6

>Layer 2: conv +pooling
>>>conv2: filter 5x5 same padding, 256 filters---->result 27 x27 x256
>>>Max pool: f=3,s=2--->result 13 x 13 x 256.

>Layer3: Conv3: same padding 384 filters-->13x13x384

>Layer4: conv only : same padding, 384 filters-->13 x13 x 384

>Layer5: Conv only: same padding: 256 filters, -->13 x13 x256

>Layer 6: max pool: f 3x3, s=2--->6 x 6 x 256

>Flatten result to vector (9216,1)

>Layer 7: FC: weight (4096, 9216)-->result vector (120,1)
>Layer 8: FC: weight (4096, 4096)-->result vector (4096,1)

>Softmax-->prediction y^, 1000 outputs probability

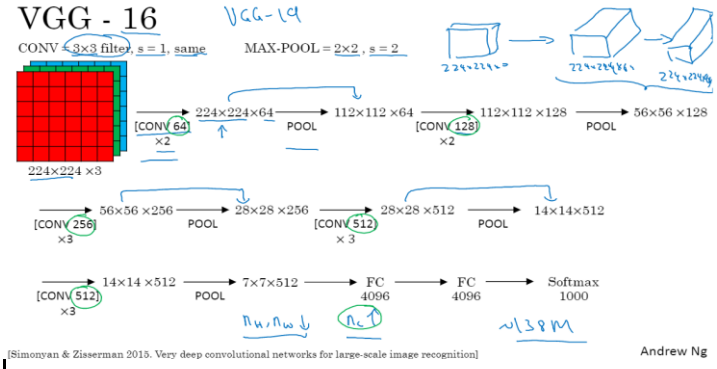
3. VGGNet-16

3.0 Intuition

A remarkable thing about the VGG-16 net is instead of having so many hyperparameters, use a much simpler network:
Focus on just having:
Conv-layers= 3x3 filters s=1, same padding
max poolling layers : 2x2, s=2

One very nice thing about the VGG network was it really simplified this neural network architectures.

VGGNet-16: has 16 layers that have weights: 13 conv layers, 2 FC layer, 1 softmax layer.



3. VGGNet-16

3.1 Architecture

input image:color 224 x224 x3

>Layer 1: conv x 2 +pooling
>>> Conv x2 (two conv layers with 64 filters)---> 224 x224 x64
VGG is a relatively deep network (many channels)
>>>Max pool:-->112 x 112 x64

>Layer 2: conv x 2 +pooling
>>> Conv x2 (two conv layers with 128 filters)---> 112 x112 x128
>>>Max pool:-->56x 56 x128
....
>Flatten result to vector (7x7x512,1)

>Layer : FC: weight (4096, 7x7x512)-->result vector (4096,1)
>Layer 8: FC: weight (4096, 4096)-->result vector (4096,1)

>Softmax-->prediction y^, 1000 outputs probability

note:
conv layer-->size no change, channel twice number ;
pooling layer: size 1/2, channel number no change.

3. VGGNet-16

3.2 Summary: VGG-16

1. Large network: 138 million parameters; (Le-Net 60,000 , Alex Net-60 million)
2. Architecture's simplicity made it quite appealing: quite uniforme
>>1. Few conv-layers followed by a pooling layer
which reduces the height and width,
>>2. Conv layer filter number is double: from 64 to 128, to 256, 512
Maybe authors thought 512 was big enough on the game here.
This sort of roughly doubling on every step, or doubling through every stack of conv-layers was another simple principle used to design the architecture of this network.
(channel double by conv-size no change, size half by pooling-channel number no change)

The relative uniformity of this architecture made it quite attractive to researchers.

Disadvantage of VGG-16:
it Is a pretty large network in terms of the number of parameters had to train.

3. VGGNet-16

3.2 Summary: VGG-16

Summarize:
There is VGG-19, an even bigger version of VGG-16. But because VGG-16 does almost as well as VGG-19. A lot of people will use VGG-16.

VGG-16/19 pattern: as go deeper and height and width goes down by a factor of two each time for the poolling layers whereas the number of channels increases roughly goes up by a factor of two every time have a new set of conv-layers.
So by making the rate at which height and width goes down and channel go up very systematic, I thought this paper was very attractive from that perspective.

Recommend reading paper: starting with the AlexNet paper followed by the VGG net paper and then the LeNet paper is a bit harder to read but it is a good classic once you go over that.

4.2-3 Residual network.

Very, very deep neural networks are difficult to train because of vanishing and exploding gradient types of problems.
Skip connections which allows you to take the activation from one layer and suddenly feed it to another layer even much deeper in the neural network. And using that, you'll build ResNet which enables you to train very, very deep networks. Sometimes even networks of over 100 layers.

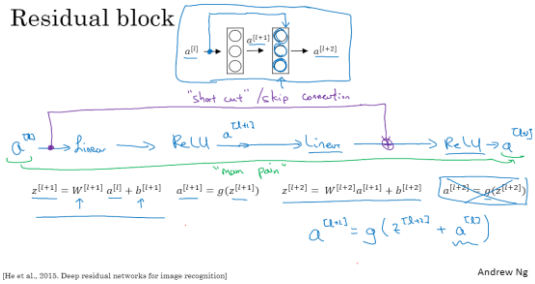
1. Residual block

e.g: Two layers :
>1. Main path is as below: from a[l] to a[l+2], need to go through all the steps.

a[l]--------->Linear function: Z[l+1]= W[l+1].a[l]+ b[l+1] ------------->activation function Relu: a[l+1]= g[Z[l+1]) ---
----------Linear function: Z[l+2]= W[l+2].a[l+1]+ b[l+2] ------------->activation function Relu: a[l+2]= g[Z[l+2])

>2. Residu net: take a[l] and just forward it, copy it, much further into the neural network
Add a[l], before applying activation funciton (e.g the ReLU non-linearity): a[l+2]= g[Z[l+2] + a[l])
So rather than needing to follow the main path, the information from a[l] can now follow a shortcut to go much deeper into the neural network.
Note: a[l] is being injected in the a further layer: after linear part, but befor activation part.

Also called: short cut/ skip connection: refers to a[l] just skipping over a layer or kind of skipping over almost two layers in order to process information deeper into the neural network.



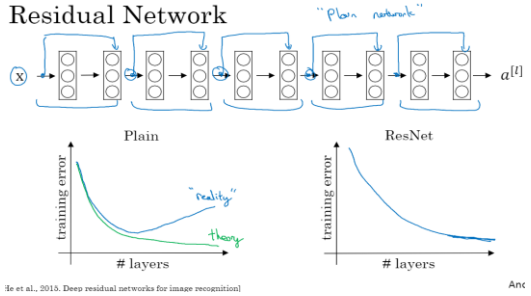
2. Residual Network:

using residual blocks allows you to train much deeper neural networks. And the way you build a ResNet is by taking many of these residual blocks, blocks like these, and stacking them together to form a deep network.

Residual block: one layer output A[l]skip few layers to add directly befor activatin function.
add in the a few further layer: between linear part and activation part: Influence layer output =g(Z[L-influenced]+ A[l])
Z': now actually have two patch: main patch (normal one)+ short patch (skipped patch).

Residual network movtivation: train deep network.

Residual network: tacks many residual block together to form deep network.
e.g : To turn this into a ResNet, add all those skip connections although those short like a connections like so: every two layers ends up with that additional change that we saw on the above example to turn each of these into residual block.
So above this picture shows five residual blocks stacked together, and this is a residual network.



3. Training error_Residual Network

If use standard optimization algorithm such as a gradient descent or one of the fancier optimization algorithms to the train algorithm:

>1. Training error for network without residual block:
for a plain (no residula connection) network, without all the extra residua/ short cuts or skip connections,:
>>> Empirically, as increase the number of layers, the training error will tend to decrease after a while but then they'll tend to go back up.
>>>While in theory as make a neural network deeper, it should only do better and better on the training set.

In theory, having a deeper network should only help, but in practice having a plain network, so no ResNet, very deep means that all your optimization algorithm just has a much harder time training.

so, in reality, your training error gets worse if you pick a network that's too deep.

>2. Training error for network with ResNet:
Even as the number of layers gets deeper, training error kind of keep on going down. Even if we train a network with over a hundred layers/over a thousand layers.

Summary:
By taking these activations be at X or these intermediate activations, and allowing it to go much deeper in the neural network, this really helps with the vanishing and exploding gradient problems and allows you to train much deeper neural networks without really appreciable loss in performance (maybe at some point, this will plateau / flatten out, and it doesn't help that much deeper and deeper networks).

But ResNet has certainly been effective at helping train very deep networks.

4.2-4 why Residual network works

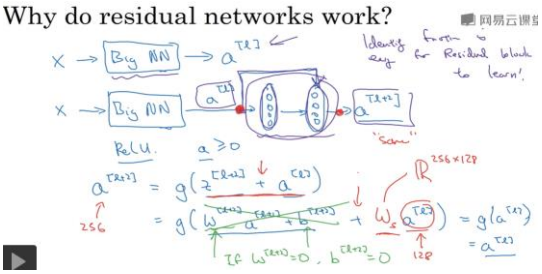
Intuition: how you can make algorithm deeper and deeper without really hurting your ability to at least get them to do well on the training set.
Doing well on the training set is usually a prerequisite to doing well on your hold up or on dev / test sets. So, being able to at least train ResNet to do well on the training set is a good first step toward that.

1. Why do residual networks work

If make a network deeper, it can hurt the ability to train the network to do well on the training set. And that's why sometimes don't want a network that is too deep. But this is not true or at least is much less true when you training a ResNet.

e.g:
Algorithm 1-without residual block: X---->Bin NN------->a[l]
Algorithm 2-with residual block: X---->Bin NN------->a[l] (also skip to a[l+2]------->a[l+1]---------->a[l+2]

Algorithm 2 is modified based on algorithm 1 to a little bit deeper with a ResNet block, a residual block with that extra short cut.
For the sake our argument, using Relu activation functions in the neural network. So, all the activations a[l]>=0,
forn algorithm 2, a[l+2] = g(w[l+2],a[l+1] + b[l+2] + a[l]).
if using regulization, w[l+2], b[l+2] close to 0, then a[l+2] ~= a[l] , so just get back, a[l].



1.1 Reason: Residual network works , due to:
>1. identity function a[l+2] ~= a[l], is easy for residual block to learn.

It's easy to get a[l+2]= a[l] because of this skip connection. Means that adding these two layers in neural network,doesn't really hurt neural network's ability to do as well as this simpler network without these two extra layers, because it's quite easy for it to learn the identity function to just copy a[l] to a[l+2] using despite the addition of these two layers.

And this is why adding two extra layers, adding this residual block to somewhere in the middle or the end of this big neural network it doesn't hurt performance.

>2. If all of these hidden units actually learned something useful then maybe you can do even better than learning the identity function.

>3.For very deep plain nets in very deep network without this residual of the skip connections , when network is deeper and deeper, it's actually very difficult for it to choose parameters that learn even the identity function, which is why a lot of layers end up making your result worse rather than making your result better.

Main reason the residual network works is that it's so easy for these extra layers to learn the identity function, kind of guaranteed that it doesn't hurt performance and then a lot the time you maybe get lucky and then even helps performance. At least is easier to go from a decent baseline of not hurting performance and then great in decent can only improve the solution from there.

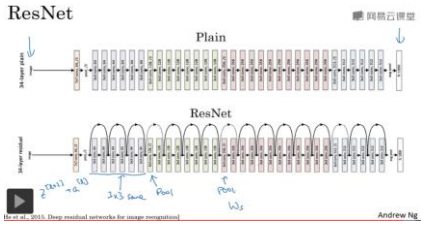
Note: skip dimension issue:
>1. In the e.g , assuming that a[l+2] and a[l] have the same dimension.
While in ResNet is a lot of use of same convolutions so the dimension of a[l] is equal to the a[l+2]dimension , then we can actually do this short circle connection, because the same convolution preserve dimensions, and so makes that easier for you to carry out this short circle and then carry out this addition of two equal dimension vectors.

>2. In case the input a[l] and output a[l+2] has different dimensions , add an extra matrix Ws, Ws.a [l] , have same dimension as a[l+2] .
Ws could be a matrix of parameters we learned, or a fixed matrix that just implements zero paddings to make a[l] have same dimension as a[l+2] -if a[l+2] dimension is bigger

2. ResNet_example on images
e.g on images

a plain network: an image -----> a number of conv layers ---> softmax output at the end.
Turn this into a ResNet: add those extra skip connections in plain network

There are a lot of 3x3, same convolutions and that's why you're adding equal dimension feature vectors.
So rather than a fully connected layer, these are actually convolutional layers but because the same convolutions, the dimensions are preserved and so the z[l+2] plus a[l] by addition makes sense.
if skip to the pooling layers,
1. make an adjustment to the dimension with matrix Ws: a[l+2] dimension smaller than a[l], matrix Ws , parameters learned to learn?;
2. or could 0 padding z[l+2] to a[l] dimension?
if a[l+2] next layer is conv, and use same padding, then a[l+2] need 0 padding for same padding-->possible to 0 padding z[l+2] first, adding a[l]?



4.2-5 network in network and 1x1 convolution

1. for input 2D data:
Input just multiply filter value, not much useful.
e.g: right : filter value=2, -->each pixel of input just multiply 2.

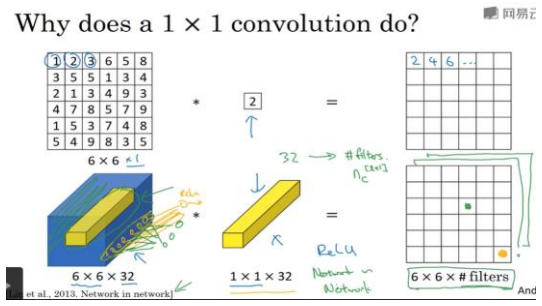
2. 1x1 convolution for input volume data:
Filter size: 1 x 1 x channel number (= input channel)
Comutation: take one postion in input, and look at it through all channels: do the element-wise product and then sum, -->then use Relu.
(why use Relu-non-linear activation for 1x1 conv?
standard conv fxf size: activation function is just stacking conv result of different filters together.)

3. 1x1 conv. intuition:
take right e.g: input image 6 x6 x32, filter 1x1x32

Intuition:
basically having a fully connected neural network that applies to each of the 32 different positions (channels). Fully connected neural network inputs 32 numbers and outputs number of filters, outputs. By doing those at each of the 36 positions, each of the six by six positions, end up with an output that is 6x6x number of filters. **This can carry out a pretty non-trivial computation on your input volume.**
(same padding, with standard conv f x f x n_c: trival computation on input volume-each output compute on input region covered by filter size, while 1x1 conv only compute one pixel and not repeated.)

Take each postion of input (through all channels): 1 x1 x32, as an input, and one filter (1x1x32) as the weight of one unit/neuron in the hidden layer, different filters as different units in the hidden layer, with different weight parameter (1x1x32), to capture input different feature.

As if have one neuron that is taking input 32 numbers. Multiplying each of these 32 numbers in one slice in the same position, height, and width, but these 32 different channels, multiplying them by 32 weights. **In applying a ReLU, nonlinearity to it and then outputting the corresponding value in the result image.**
If have multiple filters, then it's as if you have multiple units to taking as input all the numbers in one slice and then building them up into an output, 6 x 6 x number of filters.



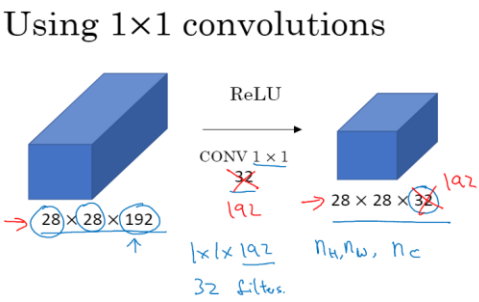
Summarize: 1x1 conv
Fully connction in channel direction, applied to each postionis of image,
output dimension: H(input) x W(input) x filter numbers
input channel number :could only shrink by convolution (filter numbers)

2. Using 1 x 1 CONV.
E.g: input 18x28 x 192: -->filter cov 1x1 x32 (with Relu) ---->result 28 x 28 x32

if shrink input height & width--->pooling layer
if shrink channel -->conv layer: standard conv layer or 1x1 CONV.

The effect of a one-by-one convolution is it just has nonlinearity: allows to learn a more complex function of network by adding another conv layer, and allows to shrink the number of input channels or keep it the same or even increase it if you want.
(linear computation on each postion through all channels and then apply Relu-->learn complex function;
shrink input channels by appling multi filters)

note: when to use 1x1 conv: region considered is narrower than standard conv, only complex funciton learned by added Relu.??



4.2-6 Inception network motivation

When designing a layer for a ConvNet, have to pick, filter size, add pooling layer or not, etc.
Inception network make the decision by it own, this makes the network architecture more complicated, but it also works remarkably well.

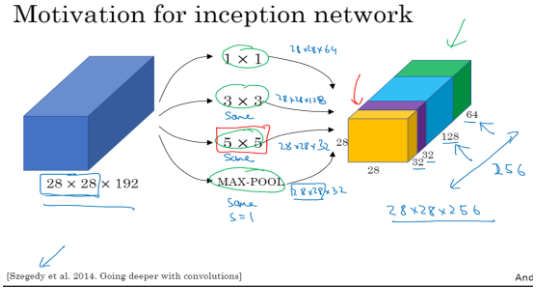
1. Motivation for inceptionnetwork

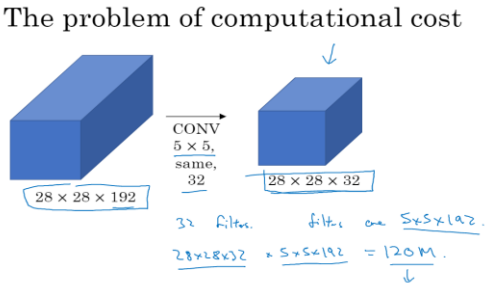
choosing filter size in conv, or whether or not adding conv/pooling layer-->inception do it by itself.

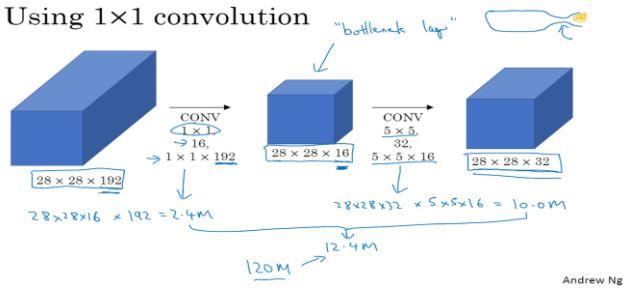
E.g: input 28x28x192:
use conv: 3 filters size: 1x1x64, 3x3, 128filter-same padding, 5x5 32 filer-same padding-->output H, W same, only different channel.
use one max-pool : and use same padding for pooling result having same dimension as conv result.--> this is the only case when to use same padding for max-pool .

stacks result together--->28x28x256

Above this is the heart of the inception network : use different size filter, max pool.
The basic idea: instead to pick one of these filter sizes or pooling, do them all and just concatenate all the outputs, and let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants.
(fixed hyperameters: filter size, step, padding as right e.g?)







2. Computational cost:

above e.g:
>1. conv filter 5x5x192, 32 filter
input 28x28x192--->5x5 filter conv, 28 x28 x32.
computation cost:
multiplication: out value numbe x comutation for each value: 28x 28x32 x (5x5x192) = 120million (expension computation)

Note: 1x1 CONV will reduce compuation by 10 times.

>2. 1X1 convolution: filter size: 1x1x192, 16 filter, + 5x5x16, 32 filter
input 28x28x192--->1X1 cov, output 28x28x16------>5x5 filter conv, 28 x28 x32.
input dimension and output dimension same as above con filter 5x5x192, but only add a much smaller intermediate volume 28x28x16-also called bottleneck layer, only 16 channel instead of 192 channel of input.

computation cost:
bottleneck: output value number x each number computation = 28 x28 x16 x 192 = 2.4 million
conv 5x5 layer computation: 28x28x32x(5x5x16)=10million
Total computaiton: 12million

-->reduce computation from 120 million mulitiplication to 12milion multiplication.
here only consider mulitiplicaton number, while adding funciton is same number

Summary:
When building a layer of a neural network and you don't want to have to decide filter size, or pooling layer, the inception module do them all, and concatenate the results.

And then we run to the problem of computational cost, while using a 1 by 1 convolution can create this bottleneck layer thereby reducing the computational cost significantly.
And It turns out that so long as you implement this bottleneck layer within reason, you can shrink down the representation size significantly, and it doesn't seem to hurt the performance, but saves you a lot of computation.

So these are the key ideas of the inception module.

(Note: computation cost saving by 1x1 conv:
1x1 cov: shrink input channel very much (e.g: from 192 to 16): this bottleneck is caused by shrinked, much smaller channel.-while size no change and only compute one time for each position.
could take it as capturing input feature in each position (through all channels-combining all original features),use linear function/fully connection + Relu.-->get much smaller , yet complex features for each position by small number of filters.
then use standard conv (same padding): to caputuring more features (channel increase from 16 to 32):
but combining multi-positions of input (filtr size region), and all of these positons' features/channel.)

motivation for 1x1 conv: only for computation saving? When to intro. ?by shrinking channles/features and then increase again, will original features/info. lost? how to choose 1x1 conv filter number?)

4.2-7 Inception network

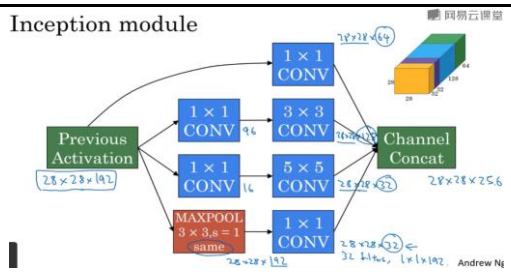
1.Inception module:

take input from some previous layer, e.g 28x28x192, using filters as below:
>1. 1x1 conv + standard conv: for save computation, add 1x1 conv before standard cov filter
>2. only 1x1 conv:
>3.Pooling: + 1x1 conv
>>>pooling use same padding here , for keep result same dimenson with above conv result dimension, for stack result together
Note: channel number do change change by pooling, still 192.
>>> 1x1 conv, used to shrink channels, as pooling do not shrink channel.
note: maxpooling + 1x1 conv: -->get 'max' value for each feature/channel, then compute more complex, new feature(by 1x1 conv filter)
better than 1x1 conv + maxpool??, (or 1x1 conv only for saving computation , add after maxpool is for future compuation saving in next layer?

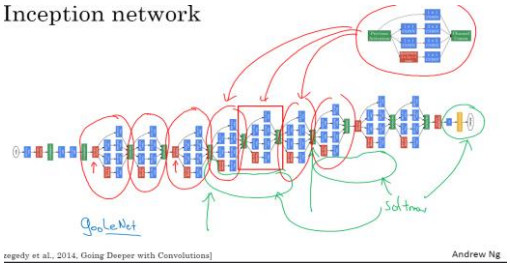
>4. Channel concat: Stack above result together

So this is one inception module, and what the inception network does, is, more or less, put a lot of these modules together.

(inception module output: size no change, only channel number changed)



2. Inception network:
Put inception modules together.



Right example: looks really complicated, But for the one block there, is basically the inception module that described above.
The inception network is just a lot of these blocks repeated to different positions of the network. (note: there is one positon, adding max pooling between two blocks for shrinking height and width).

Note:
There are side branches added in the inception network: Regularizing effect and prevent overfitting (how?):
takes some hidden layer and it tries to use that to make a prediction- softmax output a: take hidden layer out and passes through a few layers like a fully connected layers. And then has the softmax try to predict what's the output label.
hidden layer output: ->full connection-->activation fucntion-->output.
This is to ensure that features computed even in the hidden unites, even in intermediate layers , are not too bad to predicting output class of a image.
note: side branches: if this result ok, then could just take these result as final result-ignore future layers output (kind of reduce layer number, prevent overfitting?)?

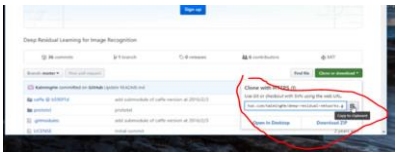
Note:
Inception 'name': also called GooglNet, developed by authors at Google.
there is a movie 'inception', cite this name as motivation for needing to build deeper new networks

Summary:
Inception network, is largely the inception module repeated a bunch of times throughout the network, now these later versions like inception v2, inception v3, inception v4, combined with the residualt idea of having skipped connections, and that sometimes works even better.

But all of these variations are built on the basic idea of coming up with the inception module and then stacking up a bunch of them together.

4.2-8 open source implements

It turns out that a lot of these neural networks are difficult or finicky to replicate because a lot of details about tuning of the hyperparameters such as learning decay and other things that make some difference to the performance.
And it's sometimes difficult even at the top universities to replicate someone else's published work just from reading their paper. Fortunately, a lot of deep learning researchers routinely open source their work on the Internet, such as on GitHub.
Recommend to use open source impletation/architecture(open source internet: GitHub) as a starter.



Search on Github:
> 1.1 search in Goolge: e.g 'Resnet GitHub'-->open the one gonna to copy and click 'download'-->copy 'https', get URL and copy it.
>1.2: in cp commend type: tyep 'git clone' +copied address, enter;-->source downloaded to local dic.
>13. Open the download filte to look into detail for the coding.
use transfer appliton to use architecture others have already training with lage data/ multi GPUs...

Summary:
When developing a computer vision application, a very common workflow would be:

>1. Pick an architecture that you like,(maybe one of the ones you learned about in this course. Or maybe one that you heard about from a friend or from some literature.)

>2. Look for an open source implementation and download it from GitHub to start building from there.
One of the advantages of doing so also is that sometimes these networks take a long time to train, and someone else might have used multiple GPUs and a very large dataset to pretrain some of these networks. ity. But because so many vision researchers have done so much work implementing these architectures, I found that often starting with open-source implementations is a better way, or certainly a faster way to get started on a new project.
And that allows to do transfer learning using these networks .

but if implementing computer vision from scratch, then your workflow will be different.

4.2-9 practical advice of convnets _transfer learing

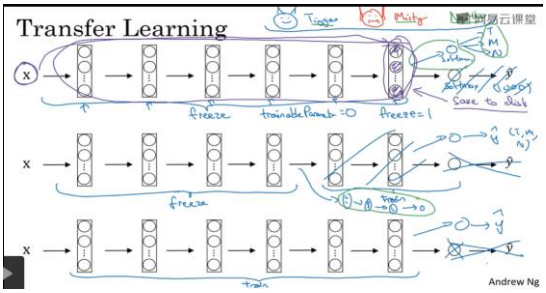
If you're building a computer vision application rather than training the ways from scratch, from random initialization,often make much faster progress if you download weights that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in.
The computer vision research community has been pretty good at posting lots of data sets on the Internet like Image Net, or MS COCO, or Pascal types of data sets, these are the names of different data sets that people have post online and a lot of computer researchers have trained their algorithms on.

And use transfer learning to sort of transfer knowledge from some of these very large public data sets to your own problem.

1. Transfer learning

download source (code and source) others have trained , use it as pre-traing then transfer to own task.
E.g: cat classifcation: is A, B or neither, _small training set.

1.1 small training set for new applicaton/task
> 1. download Imagenet:(both code and weight), which have 1000 classfication (softmax) ,output; softmax-->1000ouput
> 2. Remvoe softmax layer and oupt, create new softmax layer and output
>3. Frozen ealier layers' parameter, train parameters associated with your own softmax layer with three possible outputs, on the small data real cared.



Note:
1. frozen open source paramater method: different framework have differnt method:to specify whether or not to train paramter associated with a particular layer. like: put trainableParameter=0, means do not train weight in this layer. or freeze =1

2. Trick for fasting computing /training on softmax layer: could save the last frozen weight layers ouput (based on all real cared data), take it as 'input' to train shallow neural netork-softmax layer.: do not need to computated every time on previous layer activation of evey epoch or taking pass through training set.

1.2 Large training set for new applicaton/task

Freeze less low layers, and train more top later layers, and the later layer and softmax new created.
Having your own output unit: could take the last few layers away and just use that as initialization and do gradient descent from there or you can also blow away these last few layers and just use your own new hidden units and in your own final softmax outputs.
Either method worth trying, maybe one pattern is if you have more data, the number of layers you've freeze could be smaller and then the number of layers you train on top could be greater.

If have bigger data set, maybe of enough data not just to train a single softmax unit, but to train some other modest-sized neural network that comprises the last few layers of this final network that you end up using.

1.3 Have a lot of data for new application:
Could: take this open source network and weights and use the whole thing just as initialization and train the whole network, with your own softmax output.
-->Use the weights download just as initialization, replace random initialization and then could do gradient descent, training updating all the ways and all the layers of the network.

Summary:
This's transfer learning for the training of ConvNets. In practice, because the open data sets on the internet are so big and the ways you can download that someone else has spent weeks training has learned from so much data, you find that for a lot of computer vision applications, you just do much better if you download someone else's open source ways and use that as initialization for your problem.

In all the different disciplines, different applications of deep learning, computer vision is one where transfer learning is something that should almost always do unless, you have an exceptionally large data set to train everything else from scratch .
But transfer learning is just very worth seriously considering unless you have an exceptionally large data set and a very large computation budget to train everything from scratch by yourself.

4.2-10 practical advice of convnets _data augmentation

Most computer vision task could use more data. And so data augmentation is one of the techniques that is often used to improve the performance of computer vision systems.
Computer vision is a pretty complicated task: input this image, all these pixels and then figure out what is in this picture. And it seems like you need to learn the decently complicated function to do that.
In practice, there almost all competing visions task having more data will help. This is unlike some other domains where can get enough data.

Today, state of computer vision is : for the majority of computer vision problems, we feel like we just can't get enough data. And this is not true for all applications of machine learning, but it does feel like it's true for computer vision.

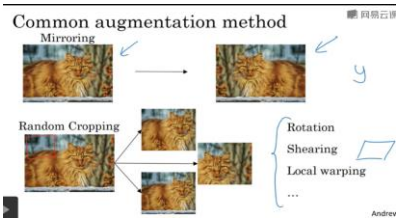
That means when you're training in computer vision model, often data augmentation will help. And this is true whether you're using transfer learning or using someone else's pre-trained ways to start, or whether you're trying to train something yourself from scratch.

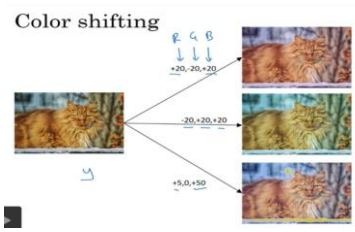
1. Common Data augmentation:

>1. Mirroring on the vertical
good to use if mirrored pic still keeps the feature you want
>2. Random cropping
not good option , as have chance to crop content /features mattered.
Yet in practice works well as long as the random craps are **reasonable large subset** of the actual image

Mirroring and random cropping are frequicly used.

3. Rote, shearing, local warping.. : a littel bit less used may due to complexity.

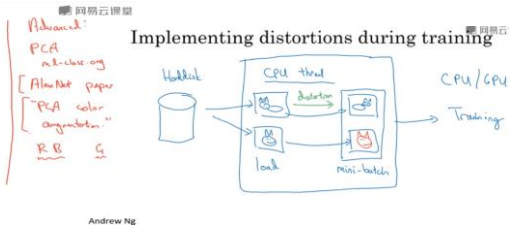




2. color shifting: Special Data augmentation:_commonly used

>1. color shifting: take different values of color channel (R, G, B), and use them to distort color channels.
color shifting moviton: image color could be changed by enviroment (sunlight, indoor illumination)-->algorithm more rubst to image color change.

Note: there are different ways to sample color R , G, B, one is PCA
color distortion: PCA (principal component analysis) color augmentation: keep overall color of the tint same （使总体的颜色保持一致）: weight of color RGB same: if image too purple: R&G weight high, PCA will add a lot subtract to R, G, but little to B.



3. Implementing distortions during training:

>1. Harddisk
Have training data stored in a hard disk .
If have a small training set, you can do almost anything and it is okay. But the very last training set and below is how people will often implement it.

>2. CPU thread
A CPU thread is constantly loading images of hard disk as well as implementing whever distortions are needed to form a batch or really many batches of data.
And this data is then constantly passed to some other thread or some other process for implementing training and this could be done on the CPU or really increasingly on the GPU if you have a large neural network to train.

>3. CPU/GPU

So, a pretty common way of implementing data augmentation is to really have one thread /multiple threads that is responsible for loading the data and implementing distortions, and then passing that to some other thread or some other process that then does the training.

store original data in hard disk-->**1 /more thread : resp. for loading data and add distortion (data augamentation)**-->pass to other thread/process for training

note: CPU thread for loading+ adding distoring, and training on CPU/GPU, could run in parallel.

Summary:
Similar to other parts of training a deep neural network, the data augmentation process also has a **few hyperparameters**: such as **how much color shifting** to implement and exactly what **parameters to use for random cropping.**

So, similar to elsewhere in computer vision, a good place to get started might be to use someone else's open source implementation for how they use data augmentation.

But if you want to capture more in variances, while someone else's open source implementation isn't, it might be reasonable also to tune these hyperparameters yourself.

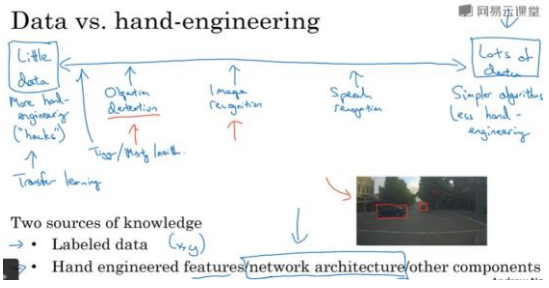
4.2-11 State of computer vision

Deep learning has been successfully applied to **computer vision, natural language processing, speech recognition**, **online advertising, logistics, many problem**s. There are a few things that are unique about the application of deep learning to computer vision, about the status of computer vision.

1. Data vs. hand-engineering
Can think of most machine learning problems as falling somewhere on the spectrum between having relatively little data to have lots of data:
1.1: Data spectra
>1. Speech recognition:
today we **have a decent amount of data for speech recognition** and it's relative to the complexity of the problem.

>2. Image recognition:
even though there are reasonably large data sets today for image recognition or image classification, because image recognition is just a complicated problem to look at all those pixels and figure out what it is. It feels like even though the online data sets are quite big like over a million images, feels like we **still wish we had more data.**

>3. Object detection
Object detection: look in the picture and actually putting the bounding boxes, telling where in the picture the objects are. As the cost of getting the bounding boxes is just **more expensive** to label the objects and the bounding boxes, we tend to have less data for object detection than for image recognition.



1.3: Algorithm Two source of knowledge:

1. labeled data (x, y) , use for supervised learning

2. hand engineering

There are lots of ways to hand-engineer a system: carefully hand **designing features**,carefully hand designing **network architectures** or maybe other components of system.
So when don't have much labeled data, have to call more on hand-engineering .
As computer vision is trying to learn a really complex function, and it often feels like we don't have enough data for computer vision. Even though data sets are getting bigger and bigger, often we just don't have as much data as we need.

This is reason why:
1. computer vision historically and even today has relied more on hand-engineering.

2. Computer vision has developed rather complex network architectures.
In the absence of more data, the way to get good performance is to spend more time architecting.
when you don't have enough data hand-engineering is a very difficult, very skillful task that requires a lot of insight. And someone that is insightful with hand-engineering will get better performance, and is a great contribution to a project.

1.2 Note: on average:
>1. when have a lot of data: tend to use simpler algorithms as well as less hand-engineering.
So, there's just less needing to carefully design features for the problem, but instead you can have a **giant neural network, even a simpler architecture,** and have a neural network, Just learn whatever it wants to learn when you have a lot of data.
>2. Don't have that much data : tend to engaging in more hand-engineering. And there are more hacks(黑客攻击) . how??
when don't have much data then hand-engineering is actually the best way to get good performance.

1.4 Computer vision status

>1. Historically due to very small data sets, computer vision literature has relied on a lot of hand-engineering.
>2. Latest: still a lot of hand-engineering of network architectures in computer vision.
in the last few years, the amount of data with computer vision task has increased dramatically,resulted in a significant reduction in the amount of hand-engineering that's being done. yet the data is not big enough, so still a lot of hand-enginerring architecture.
-->very complicated hyper parameters choices in computer vision, more complex than a lot of other disciplines.

Object detection:
And usually have smaller object detection data sets than image recognition data sets, and the algorithms become even more complex and has even more specialized components.

Transfer learning:
Fortunately, one thing that helps a lot when you have little data is transfer learning. **used a lot for when you have relatively little data.**

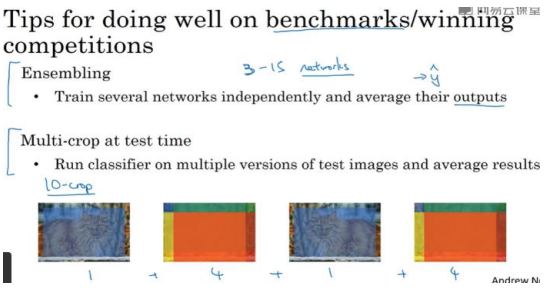
2. Tips for doing well on benchmarks/winning competitions

Computer vision researcher are really into doing well on standardized benchmark data sets and on winning competitions: easier to get the paper published. So, there's just a lot of attention on doing well on these benchmarks.

>Positive side : helps the whole community figure out the most effective algorithms.
> yet, in the papers people do things that allow you to do well on a benchmark, but that wouldn't really use in a production or a system .

Tips for doing well on benchmarks:
>1. Ensembling
Traing several neural networks (differnt weight, layers) independently and average their output (y^) (not average weights of diffent network): maybe 1% or 2% better performance on benchmark.
waisting computatin and time, not use in real customer:
As ensembling means that to test on each image, might need to run an image through may 3 to 15 different networks quite typical. This slows down your running time by a factor of 3 to 15, or sometimes even more.

Ensembling is one of those tips that people use doing well in benchmarks and for winning competitions. But almost never use in production to serve actual customers. I guess unless you have huge computational budget and don't mind burning a lot more of it per customer image



2. Tips for doing well on benchmarks:

>2. Multi-crop at test time:
Run classifier on multiple versions of test image and average test result. (slow quite computation time quite a bit)
Multi-crop is a form of applying data augmentation to your test image as well.
e.g: take a cat image and just copy it four times including two more versions. then use one technique called the 10-crop: which basically take middel, right top, righ down, left top, left down, of test image and mirrored test image-->get 10 cropped pic. and then run these 10 images through classfier and average the results.

If have the computational budget, could do this. Maybe you don't need as many as 10-crops, you can use a few crops.
And this might get you a little bit better performance in a production system for actual users.

2. Tips for doing well on benchmarks:

Summarize:
Emsembling and multi-crop are techniques that are used much more for doing well on benchmarks than in actual production systems.
And one of ensembling problems is that you need to keep all these different networks around, this takes up a lot more computer memory.
Multi-crop at least keep one network around. So it doesn't suck up as much memory, but it still slows down run time quite a bit.

3.Use open source code:

>1. Use architectures of networks published in the literature
In fact, do not tend to use these two techniques when building production systems even though they are great for doing better on benchmarks and on winning competitions. Because a lot of the computer vision problems are in the small data regime, other people have done a lot of hand-engineering of the network architectures.

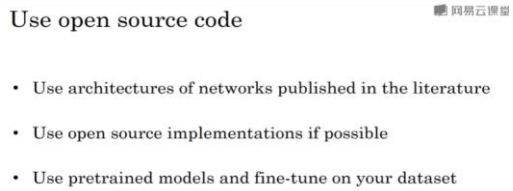
And a neural network that works well on one vision problem often may be surprisingly, work on other vision problems as well.

So, to build a practical system often you do well **starting off with someone else's neural network architecture**. And

>2. Use an o**pen source implementation** if possible
Because the open source implementation might have figured out all the finicky details like the learning rate, case scheduler, and othe**r hyper parameters.**

>3. use pretrained model and fine-tune on your data set
Someone else may have spent weeks training a model on half a dozen GPU and on over a million images. By using someone else's pretrained model and fine tuning on your data set, you can often get going much faster on an application.

But have the compute resources and the inclination, or want to invent your own computer vision algorithm , could train networks from scratch.



4.2-13 Mobile

Have learned about the ResNet architecture, Inception net. While MobileNets is also another foundational convolutional neural network architecture used for computer vision.
Using MobileNets will allow you to build and deploy new networks that work even in low compute environment, such as a mobile phone.

1. Why need another neural network architecture
Other neural networks have learned about so far are quite computationally expensive. If you want your neural network to run on a device with less powerful CPU or a GPU at deployment, then there's another neural network architecture called the MobileNet that could perform much better.

>1. Normal convolution function:
input image (n x n x n_c-number of channels)--->filter (f x f x n_c)
e.g 6x6x3, convolve it with a filter3x3x3. The way you do this is you would take this filter which I'm going to draw as a three-dimensional yellow block and put the yellow filter over there. There are 27 multiplications you have to do, sum it up, and then that gives you this value. Then shift a filter over, multiply the 27 pairs of numbers, add that up that gives you this number, and you keep going to get this, to get this, and then this, and so on, until you have computed all four by four output values. We didn't use padding in this case, and we use a stride of one, which is why the output size a nout by a nout is a bit smaller than the input size. That is this four by four instead of six by six. Rather than just having one of these three by three by three filters, you may have some n_c prime filters and if you have five of them, then the output will be four by four by five, or a nout by a nout by n_c prime. Let's figure out what is the computational cost of what we just did. It turns out the total number of computations needed to compute this output is given by the number of filter parameters which is three by three by three in this case, multiplied by the number of filter positions. That is a number of places where we place this big yellow block, which is four by four, and then multiplied by the number of filters, which is five in this case. You can check for yourself that this is the total number of multiplications we need to do, because at each of the locations we plopped down the filter, we need to do this many multiplications, and then we have this many filters. If you multiply these numbers out, this turns out to be 2,160. We'll come back. You see this number again later in this video, when we come up with the depthwise separable convolution that will be able to take as input a six by six by three image and outputs four by four by five set of activations that were fewer computations than 2,160. Let us see how the depthwise separable convolution does that. In contrast to the normal convolution which you just saw, the depthwise separable convolution has two steps. You're going to first use a depthwise convolution, followed by a pointwise convolution. It is

4-3_Object Detection

4.3-1 Object localization

Object detection is one of the areas of computer vision that's just exploding and is working so much better than just a couple of years ago. In order to build up to object detection, you first learn about object localization.

1. What are localization and detection

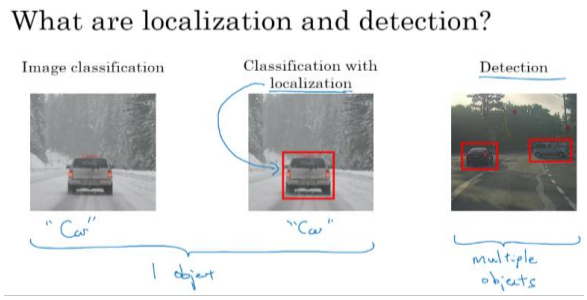
1.1 Defination

>1. Image classification task: is an algorithm looks at this picture and might be responsible for saying this is a car. So that was classification.

The problem you learn to build in your network to address later on this video is classification with localization.
>2. classification with localization.: Which means not only do you have to label this as say a car but the algorithm also is responsible for putting a bounding box, or drawing a red rectangle around the position of the car in the image.

The term localization refers to figuring out where in the picture is the car you've detected.
Later this week, you then learn about the detection problem.

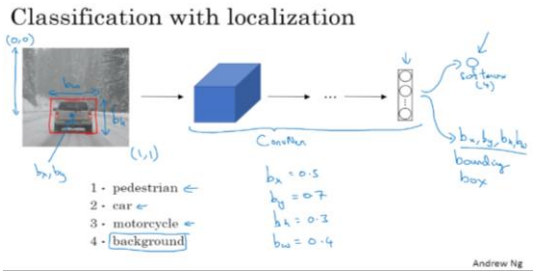
>3. Detection problem: there might be **multiple objects i**n the picture and you have to detect them all and and localized them all.
If doing this for an autonomous driving application, then you might need to detect not just other cars, but maybe other pedestrians and motorcycles and maybe even other objects.



1.2 objects number in image:
1. The classification and the classification of localization problems usually have **one objec:** one big object **in the middle of the image t**hat you're trying to recognize or recognize and localize.

2. In **detection** problem there can be multiple objects.
And in fact, maybe even **multiple objects of different categories** within a single image.

So the ideas learned about for image classification will be useful for classification with localization. And that the ideas learned for localization will then turn out to be useful for detection.

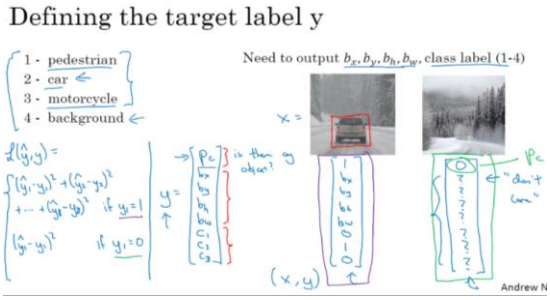


2. Classification with localization:

2.1 Standard classificaiton pipeline:
Input a picture into a ConvNet with multiple layers , and this results in a vector features that is fed to **maybe a softmax unit** that outputs the predicted class.
e.g: Building a self driving car, maybe object categories are a pedestrian, or a car, or a motorcycle, or a background. So these are algorithm classes, they have a softmax with four possible outputs.

2.2 Localize object in the image: modify on the standard classification pipeline
To recognize and also localize object in the image, change neural network to have a few more output units that output a bounding box.
e.g: can have the neural network output four more numbers, (bx, by, bh, and bw), and these four numbers parameterized the bounding box of the detected object.

Notational convention: upper left of the image is (0,0), right down corner (1,1), object central positon and size info. labeled based on image size.
Specifying the bounding box by: box midpoint (bx, by) + box size (bh, bw)



2.3 Define target lable:

Training set: (x, y)
x: image;
y: labeled: object classfication + box location (bx, by, bh, bw)
Training set contains not just the object cross label, which a neural network is trying to predict up here, but it also contains four additional numbers. Giving the bounding box then you can use supervised learning to make your algorithm outputs not just a class label but also the four parameters to tell you where is the bounding box of the object you detected.

Target label y defination:
y^=[Pc, bx, by, bh, bw, c1, c2, c3, c4]T
>1. Pc: is there a object or not (1: yes, 0 :no)
(there is a object probabilty that one of classes tryting to detect is there.). if Pc is 0, then do not care rest output b,c ,? , could be whatever.
>2. bx, by, bh, bw: object position and size. If Pc is 0, then bx, by, bh, bw is 0
>3. C1, c2, c3: object category , detected object is which category, if c1,then 1,0,0; at most one of these class category should be 1.
(only one project/class in image, softmax output is probabiify of each class-probability sum =1, take the max probability as prediction clas, and loss = y(k)* log (h(k))

training set: image, + labled target (only have two case: Pc is 1, Pc is 0)
label training set need bounding boxes (bx, by, bh, bw) in the labels.

e.g1: left pic,_there is one car in image ->label y = [1; bx; by; bh; bw; 0; 1; 0]
e.g2: right pic_no car/object in image-->label y=[0; ?; ?; ?; ?; ? ;?; ?], pc set to 0, and rest part do not care. (make randn value?? in label data??)

3. Loss fucntion (one input/image)

output y^=[y^1, y^2, y^3, y^4,..y^8]T (8 dimension)
labeled target y=[y1, y2, y3, y4,..y8]T (8 dimension)

Loss for one signal example:
>`. if y1=1 (labeled pc=1) -->L(y^, y)=sum from 1-8, (y^i-yi)^2: sum square error o ver all dimension
note: loss function here is simplified by using squared error to all output unit.
in reality could use different method for different y^i: Pc could use logistic regression, object postion and size could use squared error, classfication (c1,c 2, c3)could use log loss (output one value)

>2. if y1=0 (labeled pc=0)-->L(y^, y)=(y^1-y1)^2,
as when there is no object, all cared is how accurately the neural network output Pc.

Summary:
So that's how you get a neural network to not just classify an object but also to localize it. The idea of having a neural network output a bunch of real numbers to tell you where things are in a picture turns out to be a very powerful idea.

4.3-2 Lanmark

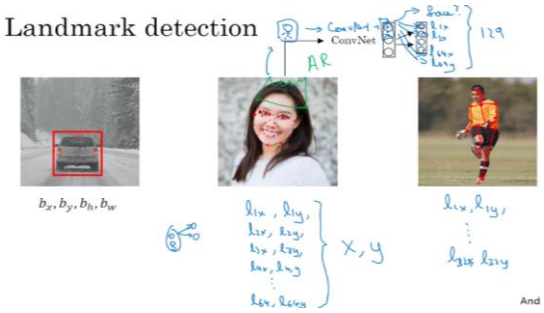
neural network output four numbers of bx, by, bh, and bw to specify the bounding box of an object you want a neural network to localize. In more general cases, you can have a neural network just **output X and Y coordinates of important points and image, sometimes called landmarks,** that you want the neural networks to recognize.

1. Landmark detection:

1.1 Example
e.g1. Face detection :
Right pic, output coordinance of 4 corners of eyes.
-->label output add 8 number for the corner coordinance: l1x, l2y , ...l4x, l4y., train neural network.
Take 64 points as landmark in image , -->Neural notwork output 128+1 value, tell if there is a face as well as all the landmarks on the face.

E.g 2: pose detection:
Could define a few key positions like the midpoint of the chest, the left shoulder, left elbow, the wrist, and so on, and just have a neural network to annotate key positions in the person's pose as well.
By having a neural network output, all of those points I'm annotating, you could also have the neural network output the pose of the person.
And of course, to do that you also need to specify on these key landmarks and use their coordinates to specify the pose of the person.

Being able to **detect these landmark**s on the face,this is also a key building block for the computer graphics effects that warp the face or drawing various special effects like putting a crown or a hat on the person.



1.2: Training set

for training set (X, Y): , images X , labels Y -People have to go through all the images X and **laboriously annotate all of these landmarks.**

> Landmark: Output x, y coordinance of important points in image
>Method:
Modify output to add more output in y^ for landmark coordinance.:adding a bunch of outpput units to output the x, y coordinanates of different landmarks want to recognize.

Note:
1. Label training set need also contains all of these landmarks' coordinance
>2. the identity of landmark 1 must be consistent across different images. labels of landmark have to consistent across differnt images (landmarks have same postion meaning in all input image, if landmark1 means left eye right corner position, then all images landmark 1 also eye right corner position)

Summary:

This idea might seem quite simple of just adding a bunch of output units to output the X,Y coordinates of different landmarks you want to recognize.

4.3-3 Object detection_slid window

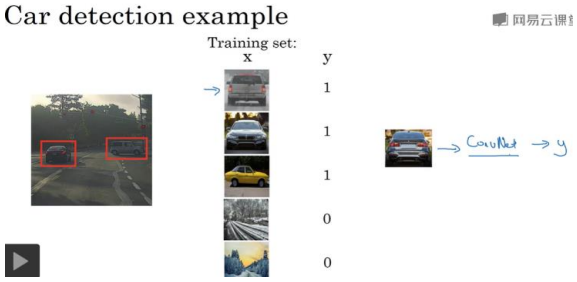
use a ConvNet to perform object detection , Sliding Windows Detection Algorithm.

1. Car detection example

Process:
>1. create a label training set, so x and y with closely cropped examples of cars.
for the purposes in this training set, you can start off with the car closely cropped images: Meaning that x is pretty much only the object car. So, you can take a picture and crop out and just cut out anything else that's not part of a car. So you end up with the car centered in pretty much the entire image

>2. Train convNet on training set, output there is car or not y=0 or 1.

>3. Use trained ConvNet classfire in sliding windows detection



2. Sliding windows detection

used trained conv neural network (classfier) in sliding window detection:
idea: go through every image region of sliding window size with some stride step, and pass lots of little cropped images into covnet, have it classified 1/0 for each postion and/at some stride.

2.1: Process
>1. picking a certain size sliding window,
>2. puting this size red squre input into conv-->conv network predict for that little red squre input, contain object or not.
>3. take a second image: by moving this red squre shift a littlbe bit over (in spefied stride step) itnto conv network and predict contain object or not
keep going , until: slid red squre across every position of the image.

Note: slide window size could be choosed acc.to Convnet requirement.(bigger, or smaller)

>4. then repeat above process 1-3 with larger size window, with speficied stride step, until run througnout the entier image until get to the end.
for the larger image region of sliding window size, resize this region into whatever input size the ConvNet is expecting, feed that to ConvNet to preditc have object or not.
>5. repeate 3rd time with more large size slide window, with speficied stride step

If do like above, then so long as there is a car somewhere in the image, there will be a window fed into convnet, will have output 1 for that image region. so detect there is a car there.

So this algorithm is called Sliding Windows Detection because you take these windows and slide them across the entire image and classify every square region with some stride as containing a car or not.

2.2 Disadvantage: Huge computation cost:
As cropping out so many different square regions in the image and run them through Convnet independently.
(e.g: pixel 'a' in 3 cropped images: cropped1, 2,3, then for each cropped image, pixel 'a' is computed by conv network-compute may many times repeately by conv same one filter.-->pixel a is computed repeately firstly in convnet classfier by same filter, then computed again repeatly in different cropped image by again same filter in conv net.)

>And if use a very coarse stride / big stride, then will reduce the number of windows you need to pass through the conNet, yet this coarser granularity may hurt performance.

>If use a very fine granularity or a very small stride, then a huge number of all these little regions will passg through the ConvNet, means there is a very high computational cost.

Summary:
>1. before the rise of Neural Networks people used to use much simpler classifiers like a simple linear classifier over hand engineer features in order to perform object detection. And in that era because each classifier was relatively cheap to compute, just a linear function, Sliding Windows Detection ran okay.it was not a bad method.
>2. but now with convnet, running a signal classfication task is much more expensive, and sliding window in this way is too slow.and unless use fine granularity (little cropped image) or very small stride, will not able to localize the object that actually within the image as well.

This problem of computational cost has a pretty good solution. In particular, the Sliding Windows Object Detector can be implemented convolutionally or much more efficiently.

4.3-4 Convolutional implementation of sliding window

Sliding windows object detection algorithm using a convnet is too slow, could implement this algorithm convolutionally.

1 . Turning FC layer into Conv layers

(motivation: getting result reprenting/corresponding to multi image position)

e.g: image 14x14x3-->conv 5x5 --->Max pool (5x5x16) --->FC (400,1)---->FC(400,1) ---->Softmax, output y (4 classes here)(4,1)

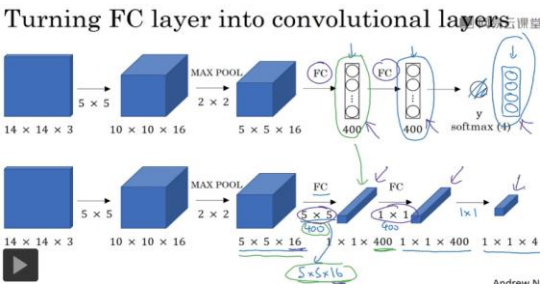
Turn FC--->FC--->Softmax into convolutional layers:
> Turn FC into. convolutional layer: conv filter size: 5x5x16, 400 filters-->result : 1x1x400 (400 channels)
filter number = FC units number, filter size same as input size (height, width, channel number).
(original no weight, current have 5 x 5 x16, x 400 weight parameters-->why could not just unrolled?)

>Turn 2nd FC into convlutional layer: filter 1x1x400 convluation, 400 filters. reslut (1x1x400)
(current and original weigh number is 400 x 400, and still fully connection:
FC -FC, change to cov layer using 1x1 conv : do not change FC property.
as FC is fully connect , each input have a unique weight for each output value. while 1x1 conv also adding unique weight to each input position for each output, different output generated by different 1x1 conv filters-adding different, unique weights to each input positions)

>Turn softmax funciotn into convlutional layer: filter 1x1 x400, 4,result (1x1x4)-
(current and original weight number both is 4 x 400, and still fully connection)

Note:
>1. FC output vector (400,1) turn into actually conv. result (1x1x400): 400 channels.
>2. Mathmatically change FC to conv is same as FC layer:, as each of output is arbitary linear function of 5x5x16(filter size) activations ,from the previous layer: filter fucntion on input position

>L3: FC, FC, softmax: change Fully connection (standard neural network weight matrix) into 1x1 conv : same



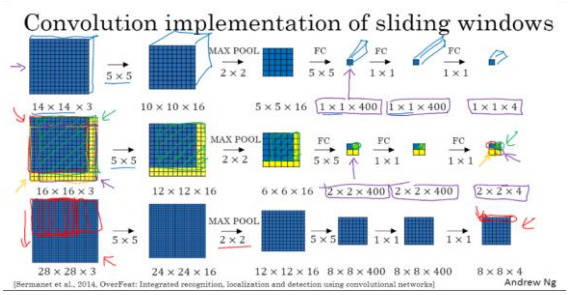
2. Convolutional implementation of sliding window

2.1 Example: convolution implement of sliding windows
sliding window ConNet inputs 14x14x3 image:
image 14x14x3-->conv 5x5 --->Max pool (5x5x16) --->FC (1x1x400)---->FC(1x1x400) ---->Softmax, output y (4 classes here)(1x1x4)

test image: 16x16x3
>1. In original sliding windows algorithm: input the blue region 14x14x3 into a convnet and run that once to generate a classification 0/1 and then slightly down (horizentally, vertically, 1 step here) to every positon of the image: run this convnet four times in order to get four labels. But it turns out a lot of this computation done by these four convnets is highly duplicative.

(e.g: pixel 'a' in these 4 cropped images: cropped1, 2,3,4. then for each cropped image, pixel 'a' is computed by conv network-compute many times repeately by conv same one filter.-->pixel a is compted repeately firstly in singal convnet classfier by same filter, then computed again repeatly in different 4 cropped image by again the same filter in conv net.)

>Convolutional implementation of sliding windows: allows these 4 forward passes in the convnet to share a lot of computation.
image 16x16x3-->conv 5x5 --->Max pool (5x5x16) --->FC (1x1x400)---->FC(1x1x400) ---->Softmax, output y (4 classes here)(2x2x4)
(could take it as using a sliding window same size as input image, -->but will not shrink image size to conv classfire expected size 14x14 x3, instead run conv filters on whole image-->pixel 'a' will go through only one time of conv computation-also repeated compuation by filters in strides involve this pixel, but will not go through 4 times-conv computation-
pixel 'a' may computed more time than seperated one conv computation in cases : pixel a in cropped image-conv classfier expected size 14x14 x3, covered by filter, but filter now is outside cropped image:-then no computaion. but in covolution implementation-for the same filter positon, filter may still inside image, then will compute fixel a also.
-may not cause computation more, as if filter still within image, then there are may cropped image-decided by sliding window and window stride, covered pixel a and also same position filter-->compute pixel a in another cropped image, computing not less than convolution implementation.



2.1 Example: convolution implement of sliding windows

method: use same conv filter parameter of sliding ConvNet run on the entier test image, instead of intputting high duplicated image regions into sliding ConvNet.

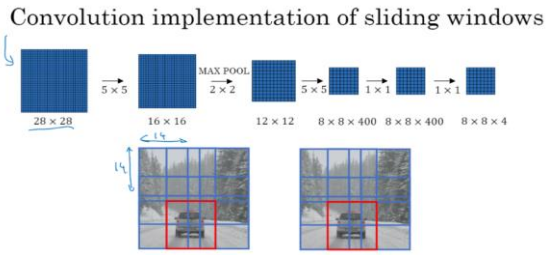
Convolution implementation: instead of forcing to run four propagation on four subsets of the input image independently, Instead, it combines all four into one form of computation and shares a lot of the computation in the regions of image that are common. So all four of the 14 by 14 patches in this e.g.

(common regions of input: shared in different(at least two) cropped windows, + could covere conv classfier filter size, this region is saved from duplicatedcomputation, go through conv classfier only once.)

2.2 Summary:
>1. Principle of convolution implementation of sliding windows:
instead of forcing to run forward propagation on all sebsets of the input image independently, combine all subset into one computation, and shares a lot of the computation in the regions of the image that are common.-->make all the predictions (whether there is object in subset region of image) at the same time by one computation (as the input image pass through the CovNet).

Note:
1. the slid window (red square (14 x14x3)) cropped test imapge into 4 subset (right top, right down, left top, left down) with sliding stride step is =classfier's conv filter step * pooling step=1*2=2
(sliding window size should have relation to conv filter step?)
2. if go convnet for each sebset, need go through 4 times computation for the common areas in the test image-->conv. allows to share common postion computation by taking it as one image instead of cropped into 4.

2.2: Example_bigger image: 28x28x3
Run forward prop the same way (conv filter , pooling and FC parameters same as sliding window conv) :
image 28x28x3-->conv 5x5 --->Max pool (5x5x16) --->FC (1x1x400)---->FC(1x1x400) ---->Softmax, output y (4 classes here)(8x8x4)
(note: image convolution implementation result size = image size -conv classfier expected size (classfier result size is 1 x 1 x channels)/sliding window step +1:
e.g: image size: 28x28, conv classfier 14x14, sliding window step s=2-->image convolution implementation result size = (28-14)/2 +1 =8)
>> sliding window step = multiplication of step of each classfire filter in conv, max, etc.; e.g above: conv stride is 1, max pool stride is 2 -->1x2 = 2,is sliding window step.



3. Summary:
for implementing sliding windows, previously, need to crop out a region (e.g 14x14) of input image, and run that through your convnet and do that for the next region over,and so on, until hopefully that one recognizes the car.

Now instead of doing it sequentially, with this convolutional implementation ,you can implement the entire image (e.g 28x28) and convolutionally make all the predictions at the same time by one forward pass through this big convnet and hopefully have it recognize the position of the car.
This makes the whole thing much more efficient.

Yet still has one weakness: the position of the bounding boxes is not going to be too accurate.
(bounding boxes position: is limited image subregion-size of classfier trained and required.??
classfier is trained with croped image, work on this size image will get one output: [pc, bx, by, ..c1, c2, c3]:-->
in image convolutional implementation, each result: of position hxw is: 1x1xc: is one classfire result: correspoinding also to the size of cropped image used for training classfire, on test image.

4.3-5 Bounding box predictions

Convolutional implementation of sliding windows is more computationally efficient, but it still has a problem of not quite outputting the most accurate bounding boxes.
Need to improve getting more accurate bounding boxes.

1. Output accurate bounding boxes

1.1 Bounding boxes accurattion problem:
With sliding windows, take discrete set of locations and run the classfier through it.In this case, none of the boxes really match up perfectly with the position of the car. And also the perfect bounding box isn't even quite square,which based on object shape.
(convolution implementation of sliding window:
1. each result position get output y^ =[; c1;c2...], corresponding to that cropped image by slding window, have which object.
2. will try different sliding window size on image, and the size that cropped image have object, will take it as object size (no-have bx, by, bw, bh : limited in this cropped size).
1. from convolution implementation of sliding window result, how to match corresponding image region?
e.g: image 28x28 x3conv implement result is 8x8x4, if one position result have object,how to find corresponding position in original image, and region size-sliding window size (window size seems not involved in conv implementation at all)
2. why not use a traied classfier +localization: prediction output y^ = [pc, bx, by , bh, bw, c1, c2..]-->get box size??-->too many output, computation cost? -while YOLO may only 19x19 output vector
)

Convolution implementing:
not try different sliding window size on image-->result hxwxc: decided by image size required by trained classfire and classfire conv & max pool step. each postion in hxw, corresponding to origian input image classfire required size region, and the region position could be localized by result positon, classfire image size, and classfire step.
e.g.of above, result 8x8x4(/9), first postion in second row-->region in input image 28x28x3: region size: 14x14x3, region location: shift sliding wind of 14x14 down by step2.

2. YOLO algorithm

YOLO: stands for you only look once.

2.1 Basic idea:
Place down a grid on input image, (e.g 3 bx 3 grid. in an actual implementation,use a finer one, like maybe a 19 x 19 grid, and then apply the image classification and localization algorithm to each of the image grids
(note: seems more efficient than convolutioning implementation: as this 19x19 grid sub-image has no common area to go through clasfire: less duplicated computation. while in convolutioning implemention, 'common area' may involved in conv calculation several times.)

2.2 Label for training
For each grid cells, specify a label Y
>1. Label Y is this eight dimensional vector: y=[pc; bx; by; bh; bw; c1; c2; c3]T
PC: 0/ 1 depending on whether or not there's an object in that grid cell
BX, BY, BH, BW: specify the bounding box if there is an object, associated with that grid cell.
C1, C2, C3: classes traing to recoginize.
e.g pedestrian's class, motorcycles and the background class. Then C1 C2 C3 can be the pedestrian, car and motorcycle classes.
e.g right pic: y1 (first grid) = [0; ? ; ? .....] (no object in the first cell, and then not care other values in this vector)
e.g : grid has object-->y=[1; bx; by; bh; bw; 0;1;0]T

2.3 YOLO algorithm example:
Use YOLO to train neural networ:
input 100 x 100 x 3--->convnet----> max pool layers--->.....------>eventually maps to 3x 3 x 8 output volume.

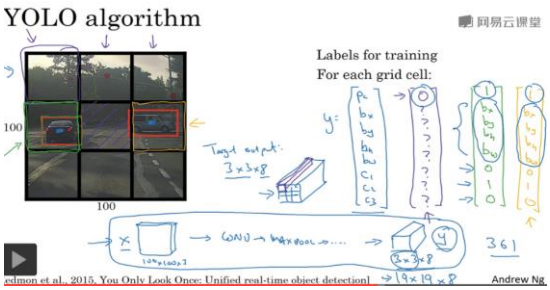
Have training set: image - 100 x 100 x 3, and target labels Y which are 3 x 3 x8, (labeled y_bh, y_bw may occupy more than one grid)
--->use back propagation to train the neural network to map from any input X to this type of output volume Y.
(note: above conv. implementation of sliding window is using a trained conv classfire to predict image., classfiere trained on cropped image with labeld y-only for classification c1, c2,..., --no, could add bounding box ino.; bx, by, bh, bw, but only bounding box dimension and position limited to cropped image size that for classfire training)

YOLO algorithm advantage: neural network outputs precise bounding boxes (bx, by, bh, bw),

At test set: feed an input image X and run forward prop until you get this output Y.
then for each of the outputs, can then just read off 1 or 0: Is there an object associated with that image positions/grid. if there is an object, what object it is, and where is the bounding box for the object in that grid cell.

Note:
1. so long as do not have more than one object in each grid cell, YOLO algorithm should work okay.
in practice, could use a much finer grid e.g (19 x 19x 8), this reduces the chance that there are multiple objects assigned to the same grid cell.

2. If the object spends multiple grid cells, it is assigned only to one grid cells
As the way to assign an object to grid cell is looking at the midpoint of an object and then assign that object to whichever one grid cell contains the midpoint of the object



2.2 Label for training
For each grid cells, specify a label Y

>2. for objects in image:
YOLO algorithm takes the midpoint of each objects and then assigns the object to the grid cell containing the midpoint.

Note:
1. For the grid that has part of the object yet not contains object midpoints:
YOLO will pretend the this grid cell has no interesting object , class label Y vector has no object
(for one grid computation: how to judge it is part of one object, and how to find midpoints if object occupy more than one grid?)

2. Totol output dimension= grid number x 8 (y dimension is 8 in this case: pc, ..c2);
each grid of input image corresponds to 8 dimension output y_i.

2.4 YOLO algorithm features:
>1. A lot like the image classification and loclization algorithm:
output the bounding box coordinates explicitly, allows the neural network to output bounding boxes any aspect ratio as well as output much more precise coordinates.-->not dictated (限制) by the stride size of sliding windows classfier.
>2. Is single Convolution implementation: do not run conv calulation on each grid seperately, but run on input image only once.

Not implementing this algorithm e.g 9 times on the 3 x 3 grid or if 362 times if using a 19 x 19 grid. Instead, this is one single convolutional implantation, where use one ConvNet with a lot of shared computation between all the computations needed for all of your 3 by 3 or all of your 19 by 19 grid cells., this is a pretty efficient algorithm

question:
1. each grid has no common region, how to share computation?
2. image grid number: has to be fixed value for one algorithm?-algorithm trained on fixed grid number (output dimension decided by grid number), what if want to change grid number, re-train algorithm-filter number/size need to change as well?
algorithm output dimension decide by grid number x dimension, grid number -->decide conv filter hyper parameter (filter size)+ input image size.-->input image size and fitler hyperamter then is fixed by grid number. if grid number change, algorithm need to change.

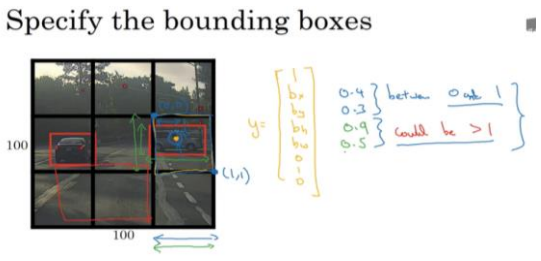
>3. popular: real-time object detection: conv.calculation actually runs very fast , so this works even for real-time object detection.

note: YOLO seems a special convolutioning implementation: trained classfire, conv the while input image one time.

Trained YOLO is a specail classfire: result grid number = { input (H) - classfire size (f) } /classfire's conv/& max_pool step +1
since each result corresponding to input image grid position, no shared positon of each grid-->
classfire step = classfire's size f
-->grid number(g) = input(H)/ f
--> classfire size f = input H / grid number

different: YOLO output bounding box dimention not limited to grid/classfier image size..

3. Specify the bounding boxes : bx, by, bh, bw



for each grid cell, define left up corner coordinance (0,0), right down corner (1,1), bounding box (bx, by, bh, bw) are specified relative to the grid cell size (h, w)

bx, by: Object middle point positon in grid cell: must between 0-1, grid cell whose coordinance defied as: left up corner coordinance (0,0), right down corner (1,1).
if bx/by not in range 0-1, that means in other grid, then current grid has no object pc=0, no care bx, by or other dimenion output.

bh, bw: pecified as fraction of grid cell size (h, w)
bx, by must between 0-1; but bh, bw may over 1 as object may occupy two or more grids.
Note: have many other paramerized methods of specifying bounding boxes , but above one convensional one works ok.

YOLO:
1. Training set:
x: image, fix all image size and set grid number;
labeled y: grid number gi, labeled y matrix: [g1_pc; g1_bx; ..g1_cn, g2_pc; g2_bx; ..g2_cn, ...gn_pc; gn_bx; ..gn_cn];

2. Algorithm:
> choose architecture: image-->inteception conv (1x1 conv, 1x1 conv + 3x3 conv, 1x1 conv +5x5 conv, pooling +1x1 conv)---->1x1 conv (FC-FC turn into conv)--->output activation function--->output.
choose: hyperperameter: filter size, output activation function(as only one object, c1, c2, ..-->softmax? pc-sigmoid? bx, by, bw, bh-linear function?)

3. Cost function: for a signal example:
pc-logistic regression, box (bx, by, bh, bw)-squared error, class c1, c2...-softmax loss y_k*log(h_k)

4.3-6 Intersectioin over union

Intersectioin over union fucntion: evaluating object detection algorithm and add another component to algorithm to make it perform better

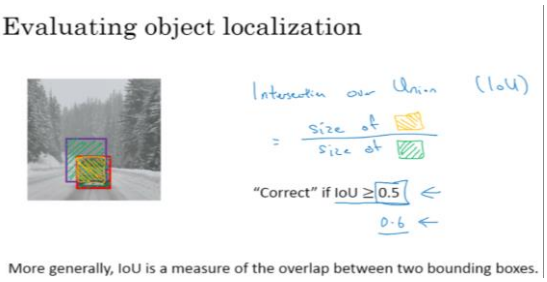
1. Evaluationg object detection

Intersection over union/**IoU**:
Method: Comute the intersection （交集）over union （并集）of output bouding box and labled output boudning box （计算两个边界框交集和并集之比）=**size of intersection/size of union**

Threshod: If IoU>=0.5, -->object detection is correct
0.5 value: human-choosen convension, no particularly deep theorectical reason for it, the higher the IOU , the more accurate the bounding box. higher threshod, more stringent. (rarely lower than 0.5)

IoU is one way to map localization to accuracy:
algorithm accuray: count up the number of times an algorithm correctly detects and localizes an object where you could use a definition like this (IOU >=0.5, correct detection and localizaiton).

Note:
IOU is also a way of measuring how similar two boxes are to each other.
IOU will be used in max-supressio method.



4.3-7 Non-max suppression

One problems of Object Detection is that algorithm may find multiple detections of the same objects. Rather than detecting an object just once, it might detect it multiple times.
Non-max suppression is a way to make sure that algorithm detects each object only once.

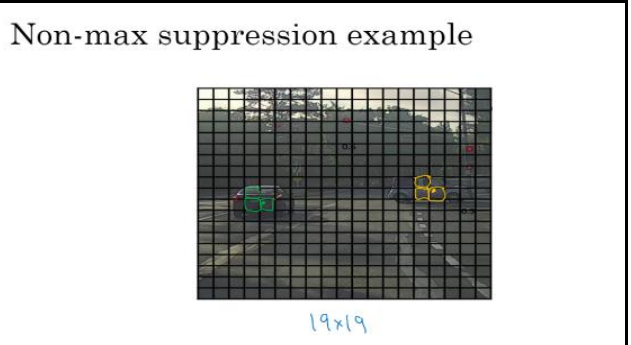
1. Non-max suppression example

e.g: input image grid is 19x19, detect pedestrian, car, etc. in image.

Multiple detection problem:
>1. Technically this one object/car has just one midpoint, so it should be assigned just one grid cell.
So technically only one of those grid cells should predict that there is specified object/a car.

>2. In practice, one object might extrends more than one grid,and these grids might take midllepoint of that sub-part as object middle point and make prediction detected one object
multiple grids may predit have one object-which is actually only sub-part of the same object

As we are running the image classification and localization algorithm on every grid cell, on 361 grid cells, it's possible that many of them will raise their hand and say, "my chance of thinking I have an object in it is large."
-->
Might end up with multiple detections of each object.
while Non- Max suppression: will cleans up these detections. So they end up with just one detection per one object, rather than multiple detections per object.

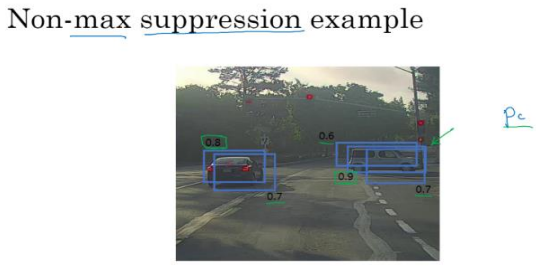


2. Non-max suppresion Pocedure

for one object catagory:
>1. 1rst: look at the probabilities associated with each of these detections count on the pc (is actually pc times c1, or c2 or c3)
>2: discard all box with pc <=0.6 (think at least 0.6 chances it is an object there)
> 3: take the box has the largest pc.
>4: looks at rest remaining boxes: all the ones with a high overlap -a high IoU with this largest pc box, will get suppressed.

>5: go thround remaining rectangles, repeat step 3, &4 above.
finally till all boxes(not disgarded: pc >0.6) dealed: either be gotten rid of as suppressed rectangles, or output as the rectangles for diffrent objects.

Non-max suppresion: means output maximal probabilities classifications but suppress the close-by ones that are non-maximal.
(key point: take the largest pc: as object
if object fully occupy many grids, is it possible largest pc in the grid not containing the whole object midpoint-how does this influence localization-bounding box position and size-no much influence??)



2. Non-max suppresion Pocedure

Note: if there is more than one object catogory: c1, c2, c3...then independently carry out non-max suppression for each catagory (e.g. times for 3 catagory): (pc actually is-times catagory) -in week exercise

If actually tried to detect more than one objects e.g pedestrians, cars, and motorcycles, then the output vector will have more (e.g3) additional componentS (C1, C2, ...CN). And it turns out, the right thing to do is to independently carry out non-max suppression N times-N classes category number, one on each of the outputs classes.

3. Non-max suppression algorithm

e.g: simplified algorithm :
only do car detection: each grid output vector get rid of c1, c2, c3 : [pc; bx; by; bh; bw]T
input image: 19x19 grid -->total output 19x19x 5

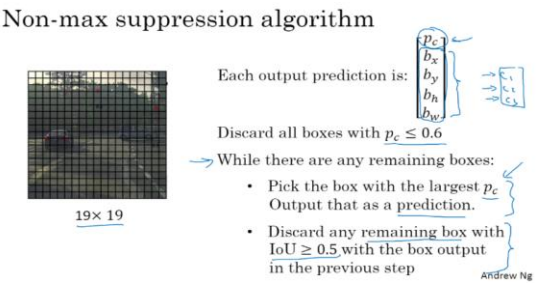
Process:
>Each output of 19x19 grid/positions, get an output prediction [pc; bx; by; bh; bw]T

>1. Discard all the predictions of the bounding boxes with Pc less than or equal to some threshold, e.g pc<= 0.6.
Means unless there's at least a 0.6 chance it is an object there, let's just get rid of it.

The way to think about this is for each of the 361 positions, you output a bounding box together with a probability of that bounding box being a good one.

>3. while there are any remaining bounding boxes:
>>>1. Repeatedly pick the box with the highest probability, with the highest Pc, and then output that as a prediction.
Commit to outputting that as a prediction for that there is an object/car there.
>>>2. Discard any remaining box with a high overlap,a high IOU (e.g >0.5), with the box that you just output in the previous step.

keep doing this while there's still any remaining boxes that you've not yet processed, until you've taken each of the boxes and either output it as a prediction, or discarded it as having too high an overlap, or too high an IOU, with one of the boxes that you have just output as your predicted position for one of the detected objects.



4.3-8 Anchor boxes

One of the problems with object detection as you have seen it so far is that each of the grid cells can detect only one object. when a grid cell wants to detect multiple objects, can use the idea of anchor boxes.

1. Overlapping objects
e.g: image have two objects: person and car, and the midpoint of these two objects are in almost the same place and both of them fall into the same grid cell.

While grid cell Y outputs= [pc; bx; by; bh; bw; c1;c2;c3]T.- 3 classess, pedestrians, cars and motorcycles,
-->it won't be able to output two detections: only set 1 to one of 3 classes.
So have to pick one of the two detections (person or car)to output.

Anchor boxes idea: pre-define two different shapes anchor boxes

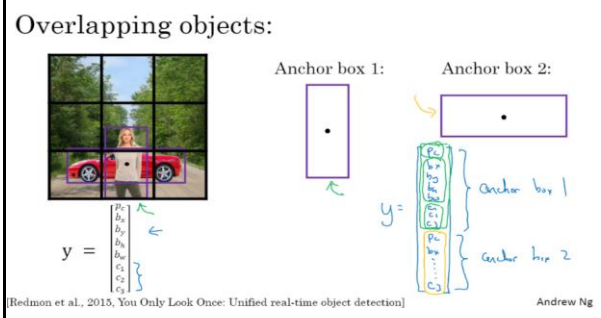
2. Anchor box algorithm:

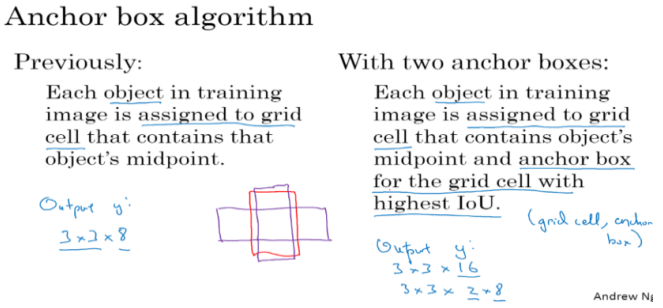
>1. Anchor box: Associate two predictions with the two anchor boxes.
In general, you might use more anchor boxes, maybe five or even more.

>output y: =[y1;y2]T
y1 =[pc; bx; by; bh; bw; c1;c2;c3]T
y2 =[pc; bx; by; bh; bw; c1;c2;c3]T
y1 associated with anchor box 1, and y2 associated with anchor box 2.

e.g:
Pedestrian shape is more similar to anchor box 1, then can encode y1 to make prediction:
pc=1: there is a pedestrian.
Use (bx, by, bh, bw) to encode the bounding box size around the pedestrian, Use (c1, c2,c3) to encode that the object is a pedestrian.

As the car shape is more similar to anchor box 2, than encode y2 to make prediction.





2. Anchor box algorithm:

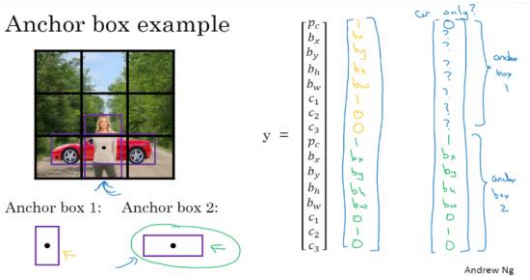
Summary:

>1. Previously: each object in training image is assigned to the grid cell that corresponds to that object's midpoint.
The output Y is 3x 3 x 8 : as has a 3 x3 grid.
And for each grid position, output vector which is PC, then the bounding box, and C1, C2, C3.

>2. With the anchor box,
Each object is assigned to the same grid cell as before, assigned to the grid cell that contains the object's midpoint and anchor box (y1/y2/y3/..) with the highest IoU with the object's shape.
e.g: each grid has two anchor boxes, then the one has a higher IoU with object shape,, will be drawn through bounding box.
(for one grid:
1. if only have one ojbect, in the prediction output, have box posiztion bx, by and size bh, bw -->could just create a rectangular box of this size and put around object.
2. if have two object-no anchox box: no idea how to assign y1 and y2 for which object.
3. if have two different shape object-have different shape anchox box associated with y1 and y2: y1 will assign to object that shape (bx,by, bh, bw) is higher overlap with y1 box--object box size still (bx, by, bh, bw)
4. if have two same shape objects-have different shape anchox box associated with y1 and y2: could not tell / fix y1/y2 for which object.
5. if have more than two shape objects-->assign more than two different anchox box with y1, y2, y3..: each object prediction y_i have a unique anchox box, to assign to the similar shape object.

Anchox box -->use shape similarity of object and y_i to locate multi object to corresponding output position y_i.
how to set anchox box size, ratio of grid cell,how to set this ratio?
)

The object then gets assigned not just to a grid cell but to a pair. It gets assigned to grid cell comma anchor box pair. And that's how that object gets encoded in the target label.



2. Anchor box algorithm:

E.g:
now the output Y is going to be 3 x 3 x16. or 3 x 3x 2 x 8 .
As there are now two anchor boxes and Y is eight dimensional. dimension of Y being eight was because we have three objects classes, if you have more objects then the dimension of Y would be even higher.

e.g1: image has both person and car, total output Y=[1;bx;by;bh;bw;1;0;0; 1;bx;by;bh;bw;0;1;0;]T
e.g1: image has only car, total output Y=[0;?;?;?;?;1;0;0; 1;bx;by;bh;bw;0;1;0;]T

3. Cases Anchor box algorithm doesn't handle well:
>1. In the same grid, objects number bigger than anchor boxes number
e.g: have two anchor boxes but three objects in the same grid cell

>2.In the same grid, objects have same anchor box shape
e.g: have two objects associated with the same grid cell, but both of them have the same anchor box shape.

That's cases that this algorithm doesn't handle well. it won't happen much at all, but if it does, this algorithm doesn't have a great way of handling it, better influence some default tiebreaker for those cases.

4. summary:
4.1 Motivation of anchor boxes:
>1. As a way to deal with what happens if two objects appear in the same grid cell.
In practice, that happens quite rarely, especially if you use e.g a 19x 19 rather than a 3 x 3 grid. The chance of two objects having the same midpoint out of these 361 cells, it does happen, but it doesn't happen that often.

>2. Even better motivation or even better results: that anchor boxes allows learning algorithm to specialize better(different shape objects).
In particular, 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.

4. summary:
4.2 Anchor box design
>1. People used to just choose them by hand or choose maybe five or 10 anchor box shapes (then 5 or 10 y_i are associated) that spans a variety of shapes that seems to cover the types of objects you seem to detect.

>2. A much more advanced version/ way to do this in one of the later YOLO research papers, is to use a K-means algorithm:
Group together two types of objects shapes you tend to get, and then to use that to select a set of anchor boxes that this most stereotypically representative of the maybe multiple, of the maybe dozens of object causes you're trying to detect.

4.3-9 YOLO Detection_putting above together

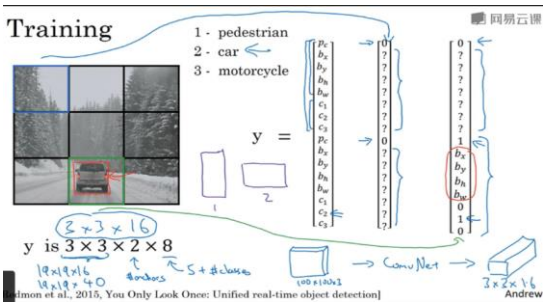
Put all the components together to form the YOLO object detection algorithm.

1. Traing set:
E.g:
Objects to detect: suppose trying to train an algorithm to detect three objects: pedestrians, cars, and motorcycles. And you will need to explicitly have the full background class, so just the class labels
Anchor box: . use two anchor boxes,
Grid cell: 3x3 grid

-->outputs y 3x3x8 x2 (-->fix algorithm architecture hyper paramter : filter size, also fix input image size)
. Eight is actually five plus the number of classes.Five is Pc and then the bounding boxes bx, by, bh, bwand then C1, C2, C3. 2 anchor box.
or either view this as 3x3x2x8,

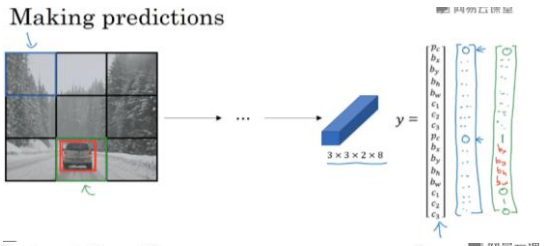
Traning set: x: image, y: go through each of these nine grid cells and form the appropriate target vector y.
e.g first grid label y:=[0,?,?,?,?,?,?,?, [0,?,?,?,?,?,?,?,]
grid has car: [0,?,?,?,?,?,?,?; 1; bx; by, bh; bw; 0;1;0]

2. Train a conv net on the input:
e.g image 100x100x3, conv net would output 3x3x16
used a pre-trained classfier with cropped image? no, this is for conv implementation for sliding window.
YOLO no pre-trained/trained classfier, just train the conv net using training set.



3. Making predictions

Given an image, neural network output 3x3x16 volume:
expected output: grid does not have object: y=[0;..... 0 ; ....]
once pc=0, then the following 7 numbers has random value-as not care these values)
for grid has car: y=[0;.....;1, bx, by, bh, bw, 0;1;0]



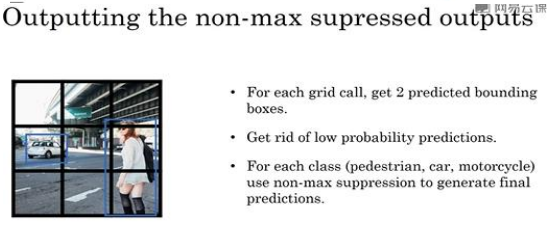
4. Outputing the non-max supressed outputs

>1.If use two anchor boxes, then for each of the grid cells, get two predicted bounding boxes.
Note: some of the bounding boxes(bx, by) can go outside the height and width of the grid cell that they came from.

>2. Get rid of low propabilty prediction: pc(like <0.6)
>.3. for each object classes, independently run non-max suppression to generate final predictions
e.g three catagory (pedestrian, car, motorcycle)-->run non-max suppression 3 times, seperately for each class.

Final output hopefully will detect all the cars and all the pedestrians in this image.

(Note: is it possible mis supressed object?: e.g: one grid cell have two objects of same class, only object shape different belong to differnt anchox box.
if these two objets box (bx, by, bh, bw) have high IOU-->one object will supressed.
these objects box compare with anchox boxes, assigne to different anchox.: anchoxes could tell these two object apart by their IOU,-Objects shape are different engough to assign to different anchox box, and to get output both objects. in this case, will these two objects shape are similar engough (IOU), to suppress the other one??-decided by anchox box shape??)



5. Summary:
YOLO object detection algorithm is really one of the most effective object detection algorithms, that also encompasses many of the best ideas across the entire computer vision literature that relate to object detection

4.3-10: Region proposal_R-CNN (Regions with convolutional networks)

If you look at the object detection literature, there's a set of ideas called region proposals that's been very influential in computer vision as well.
Make this video optional because tend to use the region proposal algorithm a bit less often but nonetheless, it has been an influential body of work and an idea that you might come across in your own work.

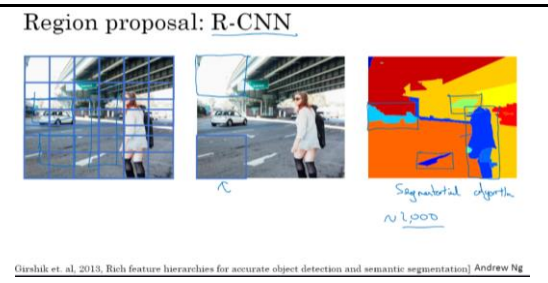
1. Region proposal: R-CNN

Sliding windows idea is: take a trained classfier and run it across all of these different windows and run the detector to see if there's a object (e.gcar, pedestrian, or maybe a motorcycle).
Run sliding window algorithm convolutionall: but one disadvantage is it classfies a lot of the regions where there's clearly no object.

R-CNN stands for Regions with convolutional networks or regions with CNNs: tries to pick just a few regions that makes sense to run your conv net classfire on.
So rather than running your sliding windows on every single window, instead select just a few windows to run conv net classfire on just a few window.

(sliding window: using a trained classfier (trained with cropped image) on image.
conv classfier parameter: will not influce by image size; and image size do not need to be fixed value.(could reshape image size)

YOLO:No trained algorithm: detector(not classfier as more than one object) +localization.
instead setting training set (X, Labled Y-acc. to grid number , class number, , anchox box number); train a algorithm to map X to Y.)

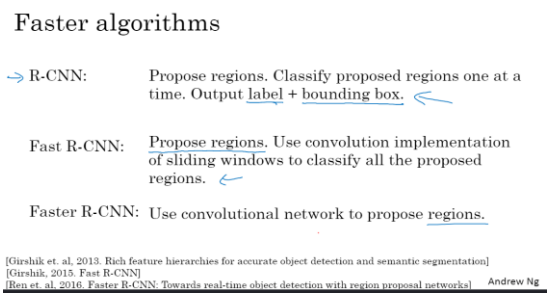


2. R-CNN

Segmentation algorithm: R-CNN use segmentation algorithm to perform the region proposals
Detail: segmentation algorithm find maybe 2000 blobs and place bounding boxes around about 2000 blobs and value classfiers on just those 2000 blobs, and this can be a much smaller number of positions on which to run conv net classfier, than to run it at every single position throughout the image.

e.g: in the Segmentation result, may finds some blobs -blu, green, etc. color over here, and might pick that bounding boxes and run classfier to see if there's some interesting there.

Note: could run conv net not just on square-shaped regions but also on tall skinny regions to try to find pedestrians or running them on your white fat regions try to find cars and running them at multiple scales as well.



3. Faster algorithm

Now, it turns out the R-CNN algorithm is still quite slow. So there's been a line of work to explore how to speed up this algorithm.

3.1. Basic R-CNN:

>>>1. Using algorithm to find proposed regions,
>>>2. then classify the proposal regionn s one at a time
(use a trained classfier for classfication + localization, like sliding window with grained classfire.
No: like YOLO without anchox box, and only one grid for the input image).
>>>3. For each of the regions, output: label+bounding box
> Label:
So is there a car? Is there a pedestrian? Is there a motorcycle there?
>also outputs a bounding box:
so can get an accurate bounding box if there is a object in that region.

Basic R-CNN Note:

1. R-CNN algorithm doesn't just trust the bounding box it was given.
It also outputs a bounding box,bx, by, bh, bw, in order to get a more accurate bounding box than whatever happened to surround the blob that the image segmentation algorithm gave it.

So it can get pretty accurate bounding boxes.
Now, one downside of the R-CNN algorithm was that it is actually quite slow.

3. Faster algorithm

3.2 Fast R-CNN

Fast R-CNN is basically the R-CNN algorithm but with a convolutional implementation of sliding windows to classify all the proposed regions.

The original implementation would actually classify the regions one at a time. while fast R-CNN use a convolutional implementation of sliding windows, and this speeds up R-CNN quite a bit.
(note: classify the regions one at a time: these regions has no common area, then no duplicated computation.-->convolutional implementation will not be faster. or ??)

Fast R-CNN algorithm problem:
The clustering step to propose the regions is still quite slow.

3.3: Faser R-CNN:

uses a convolutional neural network instead of one of the more traditional segmentation algorithms to propose a blob on those regions, and that wound up running quite a bit faster than the fast R-CNN algorithm.

Yet faster R-CNN algorithm, most implementations are usually still quit a bit slower than the YOLO algorithm.

4. Summary

The idea of region proposals has been quite influential in computer vision, and there are people still use these ideas.
Person opinion: not the opinion of the computer vision research committee as a whole:
Region propose is an interesting idea but that now having two steps: first proposed region and then classfier it, being able to do everything more or at the same time, similar to the YOLO -you Only Look Once algorithm , that seems to me like a more promising direction for the long term.

4-4_ConvNet Special use

4.4-1 Face recognition

This week is to show a couple important special applications of conv net. We'll start the face recognition, and then go on later this week to neurosal transfer,

1. Face recognition example

e.g: living people/face get acess to pass door , while using picure only did not get perssion to pass.
This example demoted: **face detection , as well as liveness detection (**make sure you are a live human)

liveness detection can be implemented using supervised learning as well to predict live human versus not live human.

2. Face verificaiton vs face recognition

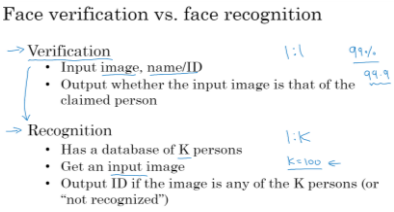
2.1 Face verfication: 1:1 prbolem: check if the person is the person claimed to be.
Given an input (image, name/ID), output whtether the input is that of the claimed person.

face verification is just want to know if the person is the person they claim to be.

2.2: Face recognition: 1:K problem

Face recognition problem is much harder than the verification problem:
e.g if have a verification system that's 99%t accurate, might not be too bad.
but now suppose that K i= 100 in a recognition system. If you apply this system to a recognition task with a 100 people in your database, you now have a hundred times of chance of making a mistake .
then if you want an acceptable recognition error, you might actually need a verification system with maybe 99.9 or even higher accuracy to work well on the 100 data.

Note:
recognition is much harder as chances to get error is k times of face verificaiton.-->if face recognition works well , requirement on face verificaton accuracy should be much higher.
1. Has a database of K persons;
2. Get an input image;
3. Output ID if the image is any of the K persons (or 'not recognized')



Summary:
In below items we will focus on building a face verification system as a building block and then if the accuracy is high enough, then you probably use that in a recognition system as well.

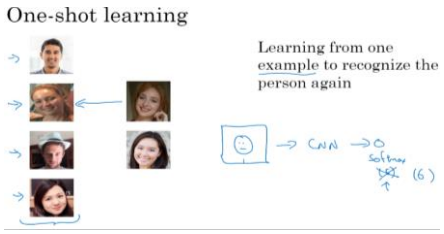
4.4-2: one-shot learning_face recognition

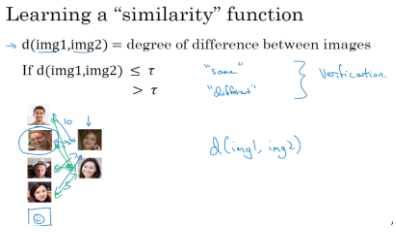
One of the challenges of face recognition is that you need to solve the one-shot learning problem:
What that means is that for most face recognition applications you need to be able to recognize a person given just one single image, or given just one example of that person's face. while historically, deep learning algorithms don't work well if you have only one training example.

1. One-shot learning

e.g: having image database of multi different people, to recognize if the person occured now is one of them.

In the one shot learning problem, you have to learn from just one example to recognize the person again.
And you need this for most face recognition systems because you might have only one picture of each of your employees or of your team members in your employee database.





2. One-shot learning approach:

2.1: Conv net: image-->conv net-->softmax, output people classes.
Input the image of the person, feed it to a ConvNet. And have it output a label, y, using a softmax unit with four outputs or maybe five outputs corresponding to each of these four persons or none of the above.

Note:
This really doesn't work well:
1. Having such a small training set, it is really not enough to train a robust neural network for this task.
2. If a new person joins your team, have to retrain the ConNet every time:
e.g : if now you have 5 persons you need to recognize, when new one join, there should now be six outputs, have to retrain the ConvNet every time.

So instead, to make this work, learn a similarity function.

2.2: Learning a 'Similarity' function:

'Similarity' function: neural network to learn a function D: inputs two images and outputs the degree of difference between the two images.

>1. If the two images are of the same person, D output a small number.
>2. If the two images are of two very different people, D output a large number.

If d<=t , --> predict same person
if d>t, --> predict different person
Threshold t: is a hyperparameter

This is how to address the face verification problem.

3. Face recogniton
to use similarity function on face recognition problem:
with given new image:
for each image in datase: use similarity function to compare new image with this database image, and output prediciton of these two images' similarity.

use similarity to compare new input image and each images in database, predict their similarity (big value or small value).

if the new image/person in database-->d will has small value, otherwise d have big value.

So long as you can learn this function d, which inputs a pair of images and tells basically, if they're the same person or different persons. Then if you have someone new join your team, you can add a new image to database, and it just works fine.

Summary:
Similarity function d, inputs two images, allows to address the one-shot learning problem.

4.4-3 Siamese network_face recognition

The job of the function d, is to input two faces and tell you how similar or how different they are. A good way to do this is to use a Siamese network.

1. Siamese network

standard image conv net: image--->a sequence of conv layerl ----> polling ----->fully connected layers---->FC: feature vector ----->softmax, output

1.1: Encoding image:
Modify on standard conv net: remove softmax output layer, only take the final feature vector as an encoding of the input image x: f(x)
image--->a sequence of conv layerl ----> polling ---->fully connected layers---->FC: feature vector

e.g: input image x1, then final get image feature f(x1).

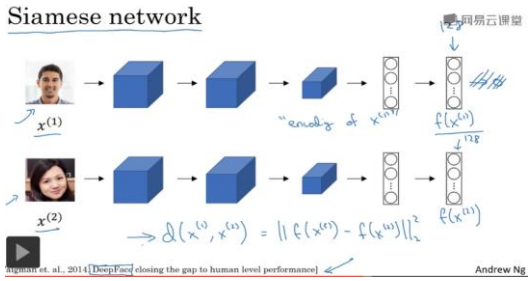
compare two pic:
The way to build a face recognition system is then that if you want to compare two pictures: image x1 ,image x2. What you can do is feed this second picture x2 to the same neural network with the same parameters that generated image x1 feature vector f(x1) and get a different vector f(x2), which encodes this second picture.

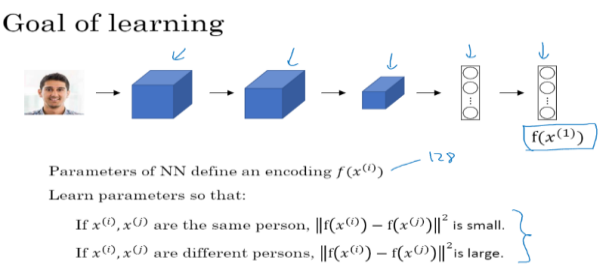
1.2: compare two pic: similarity

if believe that these encodings f(x1), f(x2) are a good representation of these two images, then can define the D-
distance between x1 and x2 as the norm of the difference between the encodings of these two images f(x1), f(x2):

d(x1, x2) ~= ||f(x1)-f(x2)||^2

Siamese neural network architecture: running two identical, convolutional neural networks on two different inputs and then comparing their encoding.





2. Train Siamese network

Note: two neural networks used for two images, have the same parameters.

2.1 Training purpose:
Train the neural network parameters so that the encoding those parameters compute results in a function d that tells whether two pictures are of the same person.

2.2 Training paramter and condtions to meet
Parameters of the neural network define an encoding f (x_i).
So given any input image xi, the neural network outputs multi dimensional encoding f(x_i).

learn parameters to generate f(x1), f(x2), meet below conditions:
If x1, x2 are same person, distance ||f(x1)-f(x2)||^2 is small; condition (1)
If x1, x2 are different persons, distance ||f(x1)-f(x2)||^2 is large. condition (2)

while vary the parameters in all of these layers of the neural network, will end up with different encodings f(x_i).
--> use back propagation and vary all those parameters in order to make sure above conditions (1) and (2) are satisfied.

4.4-4 Triplet loss method_for face recognition

One way to learn the parameters of the neural network so that it gives you a good encoding for your pictures of faces is to define an applied gradient descent on the **triplet loss function.**

1. Learning objective

1.1: Triplet loss:

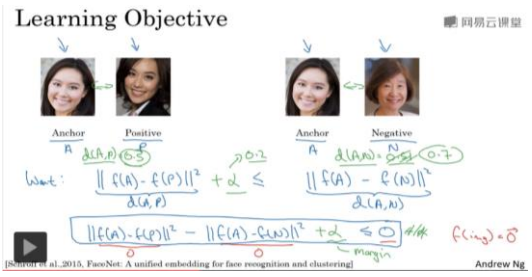
To apply the triplet loss, need to compare pairs of images.
e.g: given a pair of images, want their encodings to be similar when they are the same person. Whereas, given another pair of images, want their encodings to be quite different as they se are different persons.

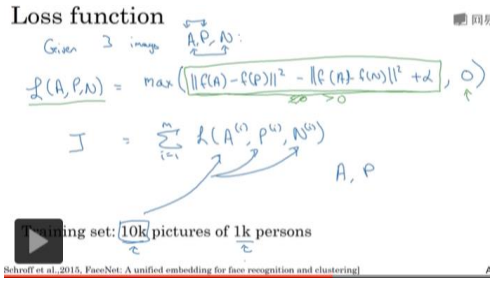
Triplet loss function: always look at one anchor image and want distance between the anchor and the positive image (positive meaning is the same person) to be similar. Whereas want the anchor when pairs are compared to the negative example for their distances to be much further apart.

So, this is what gives rise to the term triplet loss, which is that you'll always be looking at three images at a time.Will be looking at an A:anchor image, a P: positive image, as well as a N:negative image.

Triplet loss , want to meet condtions: d(A, P) + a<= d(A,N)
d(A,P): distance of anchor imge encoding f(A) and postive image encoding f(P).
a: margin parameter, is a hyperparameter: prevents a neural network from outputting the trivial solutions, by making:
>1.Neural network does not just output 0 for all the encoding f(x_i)==0. (then d==0);
>2. Neural network does not encoding every image identical to other image (then d==0)

A margin parameter pushes the anchor positive pair and the anchor negative pair further away from each other, as to satisfy d(A, P) + a<= d(A,N), tend to make d(A,N) bigger, while d(A,P) smaller.





2. Loss function:

Triplet loss function is defined on triples of images. Given three images, A, P, and N, the anchor positive and negative examples. So the positive examples is of the same person as the anchor, but the negative is of a different person than the anchor.

Loss function on a singal triplet: = max (d(A,P) - d(A,N) +a, 0)
loss =0, if d(A,P) - d(A,N) +a <=0;
loss = d(A,P) - d(A,N) +a, if d(A,P) - d(A,N) +a>0
By minimizing loss fucntion, will pull d(A,P) - d(A,N) +a <=0, and does not care how negative it is once it is less than 0.

Overall cost function for neural network: sum over a training set of these individual losses on different triplets. (not average then 1/m?)

e.g: training set is10,000 pictures with 1,000 different persons, then take 10,000 pictures and use it to generate, to select triplets (A,P,N) and then train your learning algorithm using gradient descent on cost function, which is really defined on triplets of images drawn from your training set.

3. Training set:

Training set: have multiple pictures of the same person at least for some people
In order to define this dataset of triplets, do need some pairs of A and P. Pairs of pictures of the same person.
So for the purpose of training system, do need a dataset where have multiple pictures of the same person. If you had just one picture of each person, then you can't actually train this system.

after training, could apply the trained system to one shot learning problem where for your face recognition system, maybe you have only a single picture of someone you might be trying to recognize.
But for training set, do need to make sure having multiple images of the same person at least for some people in training set so that you can have pairs of anchor and positive images.

Note: each triplet is independently of other triplets(A_i, P_i, N_i)

does A need to be same one in the whole training set?-no, each triplet is indenpendently of other triplets (coudl one image A use in different triplet?-no neccasry,as will choose hard training pairs, no necassry to train on muliti same A, pairs.)

4. Choosing the triplets A,P,N

4.1 Random choose problem: constraint d(A,P) +a <= d(A,N) is easy to satisfy.

If choose A, P, and N randomly from training set subject to A and P being from the same person, and A and N being different persons, one of the problems is this constraint d(A,P) +a <= d(A,N) is very easy to satisfy.

Reason:
given two randomly chosen pictures of people, chances are A and N are much different than A and P.
if A and N are two randomly chosen different persons, then there is a very high chance that d(A,N) will be much bigger more than d(A,P) +a , And so, the neural network won't learn much from it.

4.2: Manually choose: choose triplets that are 'hard' to train on

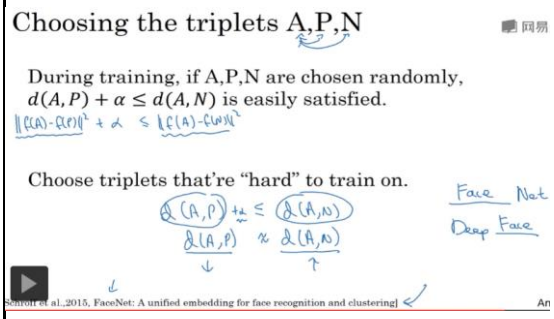
Expectation for all triplets is satisfiying constraint d(A,P) +a <= d(A,N) be satisfied.

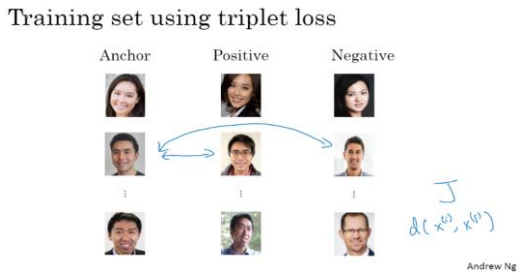
Hard triplet : is choosing A, P,N, d(A, P) is actually quite close to d(A,N). (-->P, N image that are close)
So in this case, the learning algorithm has to try extra hard to make d(A,N) bigger and try to push d(A,P) down so that there is at least a margin of a between them.

Effect of choosing hard triplets: increasing the computational efficiency of your learning algorithm.
If you choose your triplets randomly, then too many triplets would be really easy, and so, gradient descent won't do anything because your neural network will just get them right, pretty much all the time. And it's only by using hard triplets that the gradient descent procedure has to do some work to try to push d(A,N) further away from d(A,P).

Parameter update: θ:= θ- a * dJ/dθ
randomly choose triplet, easys to meat d (A, N) > d (A, P) + a ---> J =0. ~~, -->dJ/dθ =0, neural network parameter θ has no update.~~
hard triplet: d (A, N) may < d (A, P) + a -->J = d (A, P) + a- d(A, N) >0, function of parameter -->dJ/dθ may !=0, -->neural network parameter update more.

Note: naming in deep learing in literature: 'bland' net or deep 'blank'.
E.g: face net/ deep face





4. Summary
This is for the triplet loss and how you can train a neural network for learning an encoding for face recognition.
To train on triplet loss, need map training set to a lot of triples: with an anchor and a positive, and a negative. where the anchor and positive are of the same person but the anchor and negative are of different persons and then use gradient descent to try to **minimize the cost function J , which will have the effect of that propagating to all of the parameters of the neural network in order to learn an encoding** so that d of two images will be small when these two images are of the same person, and they'll be large when these are two images of different persons.

Note:
Face recognition systems especially the large-scale commercial face recognition systems are trained on very large datasets, which are not easy to acquire. yet some of these companies have trained these large networks and posted parameters online, so it might be useful to download someone else's pre-train model, rather than do everything from scratch yourself.

4.4-5 Binary classification_face verification

Triplet Loss is one good way to learn the parameters of a conv net for face recognition. There's another way to learn these parameters, **pose face recognition as a straight binary classification problem.**

1. Learning the similarity function

1.1: Treating face recognition as Bianary classificaiton
Alternative to the triplet loss for training a system , treatting face recognition just as a binary classification problem:
Train a neural network, take a pair of same Siamese Network and compute two image embeddings, maybe multi dimensional embeddings, and then feed these embeddings to a logistic regression unit to make a prediction.
Target output will be 1 if both of these are the same persons, and 0 if both of these are of different persons.

1.2: Final logistic regression:

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

output y^ will be a sigmoid function, applied to some set of features: y^ =g (f(x1), f(x2))

Note:
1. Rather than just feeding in image these encodings to sigmoid function, take the differences between the encodings.
y^ =g (sum over k , of w_i * |f(x1)_k - f(x2)_k| +b)
a sum over K equals 1 to 128 of the absolute value, taken element wise between the two different encodings.

2. could take image embedding f(x) as features, and logistic regression have additional parameters w_i, and b similar to a normal logistic regression unit to each feature fed in. and then train appropriate weight on these e.g 128 features in order to predict whether or not these two images are of the same person or of different persons.
So, this will be one pretty useful way to learn to predict zero or one whether these are the same person or different persons. **y^ = g (W* |f(x1)-f(x2)| +b)**

Note:
There are a few other variations on how you can compute w_i * |f(x1)_k - f(x2)_k|.
e.g: another formula -k square formula: squared feature distance / f(x1)_k - f(x2)_k.

(note: triplet loss, from image feature to output d, only have one **hyper parameter a-margin value.**
while binary classficiation loss , from image feature to output 1/0, have **parameter weight w_i and b-bias-could learn by back prop**.)

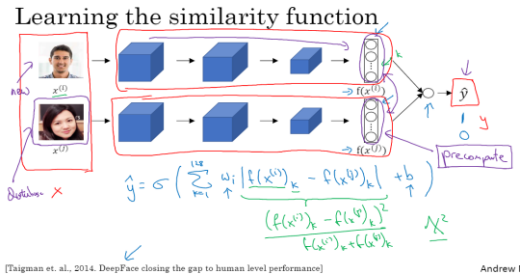
1.3 Summarize:

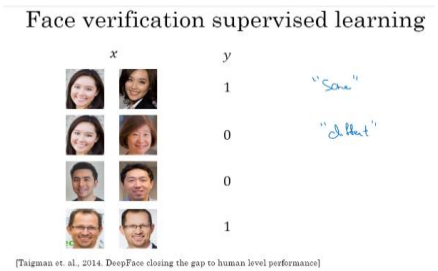
So in this learning formulation, the input x is a pair of images, and the output y is either zero or one depending on whether you're inputting a pair of similar or dissimilar images.

Note: Same Siamese Network for the input pair of images.

computational trick:
for the database images, pre-compute their codings and store them.
so instead of having to compute this embedding every single time, when the new employee walks in, could just compute this new image coding and compare it to pre-computed database encoding and then to make a prediction y^,

As don't need to store the raw images and if have a very large database of employees, you don't need to compute these encodings every single time for every employee database.
This idea of pre- computing can save a significant computation, and both for this type of Siamese Central architecture where you treat face recognition as a binary classification problem, as well as, when you were learning encodings maybe using the Triplet Loss function as described in the last couple of videos.





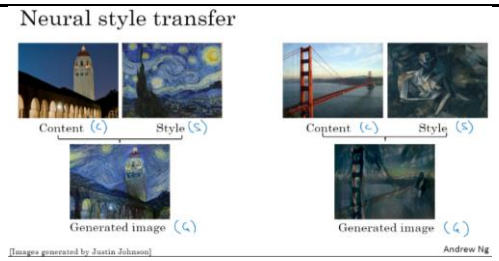
Summary:
For treating face verification supervised learning: treate a training set of just pairs of image , instead of triplets mages: target label is 1 When these are a pair of pictures of the same person and where the target label is 0, when these are pictures of different persons.
And use different pairs to train the Siamese netwok using back propagation.
Treating face verification and by extension face recognition as a binary classification problem, this works quite well as well.

4.4-6 neural style transfer

One of the most fun and exciting applications of ConvNet recently has been Neural Style Transfer.

e.g: take one image-C content image, want this picture recreated in the style of another image-S styple image, Neural Style Transfer allows to generated new image-G: have C image content yet painted in S image styple.

In order to implement Neural Style Transfer, you need to look at the features extracted by ConvNet at various layers, the shallow and the deeper layers of a ConvNet, get better intuition about whatever all these layers of a ConvNet is really computing.



4.4-7 Deep convnet visualizing

Item purpose:
share visualizations that will help get intuition about what the deeper layers of a ConvNet really are doing. And this will help us think through how to implement neural style transfer as well.

1. Visualizing what a deep network is learning

1.1 Idea of visulizing hidden unit :

>1. Pick a hidden unit in layer l , put training set through neural network, and figure out what is the image that maximizes that particular unit's activation.
(activation result of unit i in layer l, for all training examples m : a[l]_i = [a[l]_i_1, a[l]_i_2.....a[l]_i_m], find the max few values a[l]_i_?-->correspondign image)

e.g: for a **trained** neural network like a ConvNet, pick a hidden unit in layer l, scan through training sets and find out what are the images or what are the image patches that maximize that unit's activation.

Note:
a hidden unit in layer 1, will see only a relatively small portion of the neural network. So if to visualize- plot what activated unit's activation, it makes sense to plot just a small image patches, because that's all of the image that this particular unit sees.
e.g:
If pick one hidden unit and find 9 input images that maximizes that unit's activation, nine image in the right: looks like this particular hidden unit is looking for an egde or a line

1.2 Note:
1. different hidden units in the same layer, tend to activate in similar image patches: catching similar type of features.

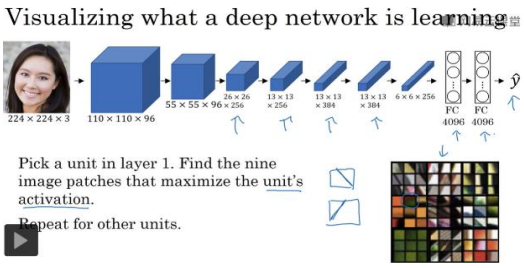
2.Trained hidden units in shallow layers, tend to look for relatively simple features such as edge or a particular shade of color.

3. In the paper, some other more sophisticated ways of visualizing when the ConvNet is running is also used..

1.1 Idea of visulizing hidden unit :

>2. Repeate for other units
e.g:
pick a different hidden unit in layer 1 and do the same thing. looks like this second unit, represented by another 9 image patches:looks like this hidden unit is looking for a line sort of in that portion of its input region.

1.2 Note:
1. different hidden units in the same layer, tend to activate in similar image patches: catching similar type of features. (?? for conv, each channel generated by different filter, means different feature)
2.**Trained** hidden units in shallow layers, tend to look for relatively simple features such as edge or a particular shade of color.
3. In the paper, some other more sophisticated ways of visualizing when the ConvNet is running is also used..



2. Visualization deep layers: Layer 5

In the deeper layers, a hidden unit will see a larger region of the image.
Extreme cases, each pixel could hypothetically affect the output of these later layers of the neural network.
So later units are actually seen larger image patches.

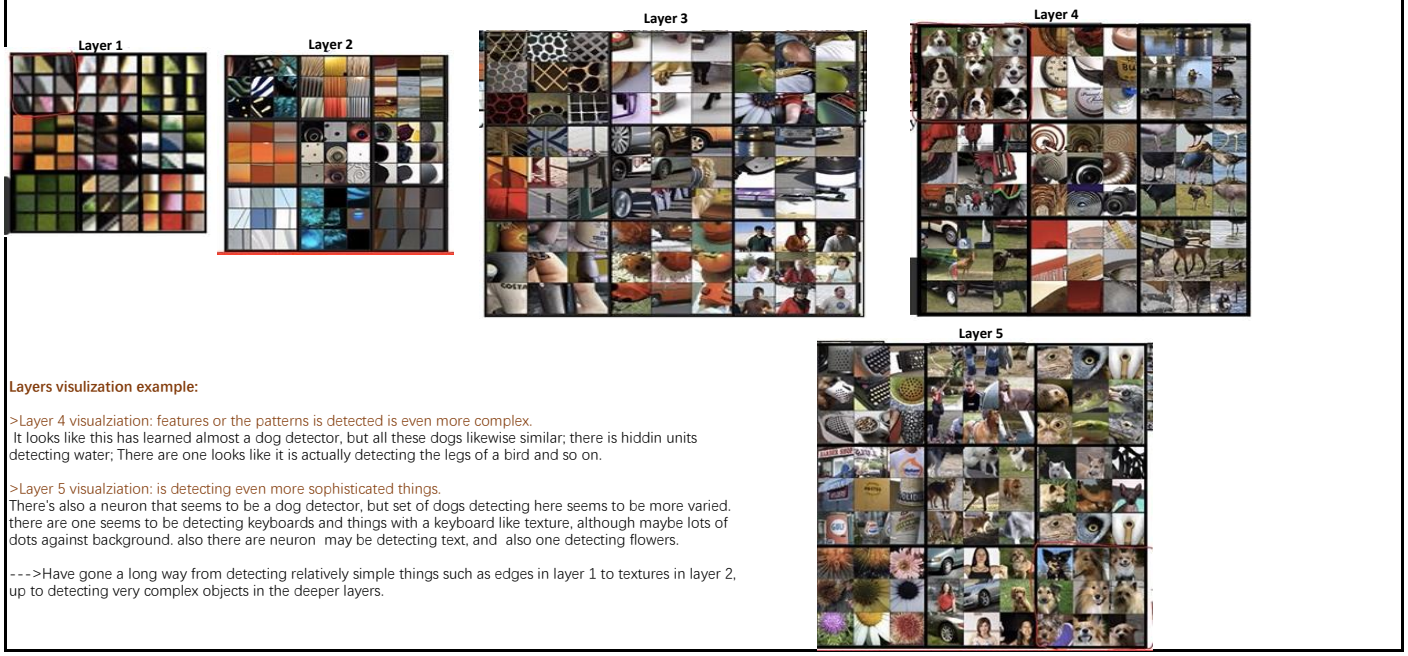
2.1 Example:
Plot the image patches as the same size -see right pic. each layer have 9 units, and each units have 9 imges that maixmally activate it. Repeated for all layers.
If repeat this procedure, get image paches that maximally activates all the hidden units.

Layers visulization example:

> Layer 1 visualization: there's an edge maybe at that angle in the image;

>Layer 2 visualziation: looks it's detecting more complex shapes and patterns.
e.g: looks like it's looking for a vertical texture with lots of vertical lines. This hidden unit looks like its highly activated when there's a rounder shape to the left part of the image. Here's one that is looking for very thin vertical lines and so on.
And so the features the second layer is detecting are getting more complicated.

>Layer 3 visualziation: clearly starting to detect more complex patterns
It looks like there is a hidden unit that seems to respond highly to a rounder shape in the lower left hand portion of the image, maybe. So that ends up detecting a lot of cars, dogs and wonders is even starting to detect people. And this one look like it is detecting certain textures like honeycomb shapes, or square shapes, this irregular texture. And some of these it's difficult to look at and manually figure out what is it detecting, but it is clearly starting to detect more complex patterns.



Layers visulization example:

>Layer 4 visualziation: features or the patterns is detected is even more complex.
It looks like this has learned almost a dog detector, but all these dogs likewise similar; there is hiddin units detecting water; There are one looks like it is actually detecting the legs of a bird and so on.

>Layer 5 visualziation: is detecting even more sophisticated things.
There's also a neuron that seems to be a dog detector, but set of dogs detecting here seems to be more varied. there are one seems to be detecting keyboards and things with a keyboard like texture, although maybe lots of dots against background. also there are neuron may be detecting text, and also one detecting flowers.

--->Have gone a long way from detecting relatively simple things such as edges in layer 1 to textures in layer 2, up to detecting very complex objects in the deeper layers.

4.4-8 cost function_neural style transfer

Item purpose: To build a Neural Style Transfer system, define a cost function for the generated image and by minimizing this cost function, we can generate the image want.

1. Neural style transfer cost function

1.1 Neural style transfer: given a content image C + a style image S, and the goal is to generate a new image G.

1.2 Cost function J(G): purpose
Defined in order to implement neural style transfer, measures how good is a particular generated image G . And use gradient descent to minimize J of G in order to generate this image.

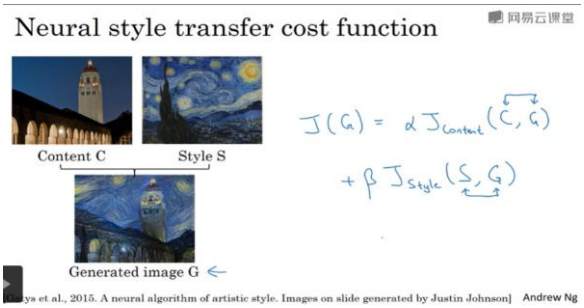
1.3 Cost function part: J(G) = a *J(C, G) +β* J (S, G)
Define two parts to this cost function.

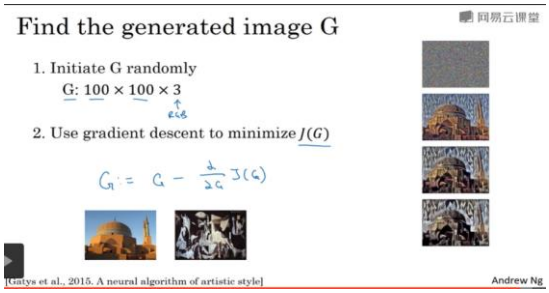
>1. Content cost: J(C, G)
defination: funciton of the content image and generated image.
Function: measure the similarity of the contents of generated image to the content of the content image C.

>2. Style cost: J (S, G)
defination: funcion of style image and generated image
Fucntion: measure the similarity of the style of generated iamge to the style of the style image S.

1.3 Cost function part: J(G) = a *J(C, G) +β* J (S, G)
Define two parts to this cost function.

>3. hyper parameters: a, β:
Function: specify the relative weighting between the content costs and the style cost.
It seems redundant to use two different hyper parameters to specify the relative cost of the weighting. One hyper parameter seems like it would be enough .but the original authors of the Neural Style Transfer Algorithm, use two different hyper parameters, so just follow their convention here.





1.4 Process: Algorithm have to find the cost function J (G) in order to actually generate a new image

>1. Initialize the generated image G randomly
might be 100 x 100 x 3 or 500 x 500 x 3 or whatever dimension you want it to be. (then scale dimension to Siamese conv network to generate image embedding?)

>2. Define the cost function J (G): J(G) = a *J(C, G) +β* J (S, G)

>3. use gradient descent to minimize cost , so can update G := G- d(J) /d(G).

Note: iIn this process of running gradient decsent and updating G , you're actually updating the pixel values of this image G which is a 100 x 100 x 3 maybe GRB channel image (image paramters: 100x100x3=30,000).

E.G:
Given content image and style image, initialize G randomly. As run gradient descent, minimize the cost function J(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.

4.4-9 content cost of cost function_neural style transfer

1. Content cost function

Overal cost function: J(G) = a *J(C, G) +β* J (S, G)

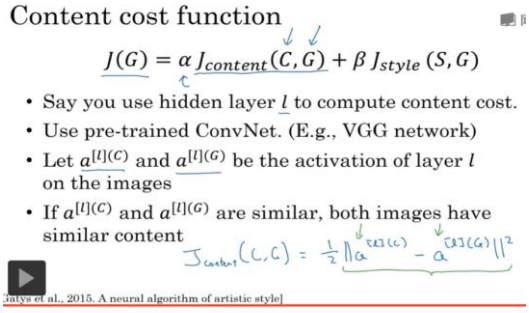
1.1 Content cost process/Idea:
>1. Use hidden layer l to compute the content cost.
Layer l chosen somewhere in between. It's neither too shallow nor too deep in the neural network.
>>>1. If l is a very small number, like layer 1, then it will really force algorithm to generate image to pixel values very similar to your content image.
>>>2. If use a very deep layer, then it's just asking, "Well, if there is a dog in your content image, then make sure there is a dog somewhere in the generated image.

Note: usually choose layer to be somewhere in the middle of the layers of the neural network, neither too shallow nor too deep.

>2. Use a pre-trained ConvNet () (Siamese network??),maybe a VGG network, measure similarity in content of content image C and generated image G.

>3. Let a[l]_C, a[l]_G be the activations of layer l on these two images, on the images C and G.

>4. if these two activations a[l]_C, a[l]_G are similar, then that would seem to imply that both images have similar content.



1.2 Content cost defination:
Take the element-wise difference between these hidden unit activations in layer l, between when you pass in the content image compared to when you pass in the generated image, and take that squared.

J (C, G)_content = 1/2 * || a[l]_C - a[l]_G|| ^2

Note:
1. could have a normalization constant J formaular or not., so it's just 1/2, or something else.
It doesn't really matter since this can be adjusted as well by this hyperparameter a. S

2. Both a[l]_C & a[l]_G have been unrolled into vectors,
-->content cost is really just the element-wise sum of squared differences between these two activations in layer l, between the images in C and G.

3. when later perform gradient descent on J (G) to try to find a value of G, so that the overall cost is low, this will incentifize /force the algorithm to find an image G, so that it's hidden layer activations are similar to that layer activations of the content image.

4.4-10 styple cost of cost function_neural style transfer

1. Image style:
Example: input iamge , use conv net to compute features that there are in different layers.

>1. Chosen some layer l, to define the measure of the style of an image.
>2. Define image style as the correlation between activations across different channels in this layer l activation.
could take different channels as different hidden units in layer l, image style defined in the layer l, means each hidden units value relation with other hidden units.

1.1 Style intuition:
for layer l activation: nh x nw x nc and check how correlated are the activations across different channels.

Example:
has five channels, look at the first two channels and see how correlated are activations in these first two channels.

>1. Idea:
for the same positon in channel 1, and channel 2, get a pair of number.and see when you look across all of these positions, all of these nh x nw positions, how correlated are these two numbers.

This capture style: take each channel as one hidden unit in layer l.
e.g: in the layer l visualizations, the red channel's maxi activation image is vertical texture-->red channels corresondpong to neurons trying to figure out if there's this little vertical texture in a particular position in the nhxhw.
This yellow second channel corresponds to this neuron, which is vaguely looking for orange colored patches.

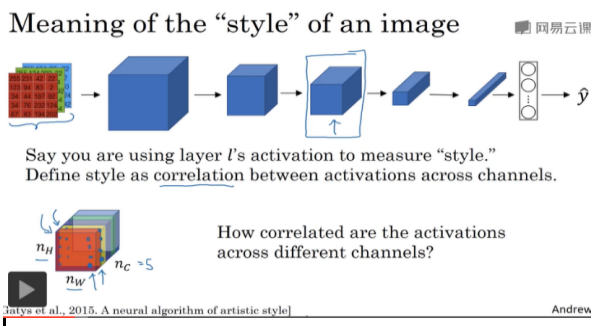
If these two channels are highly correlated, that means whenever part of first channel maxi activation image has this type of subtle vertical texture, that part of second channel maxi activation image will probably have these orange-ish tint.
if they are be uncorrelated, , it means that whenever there is this vertical texture, it's probably won't have that orange-ish tint.

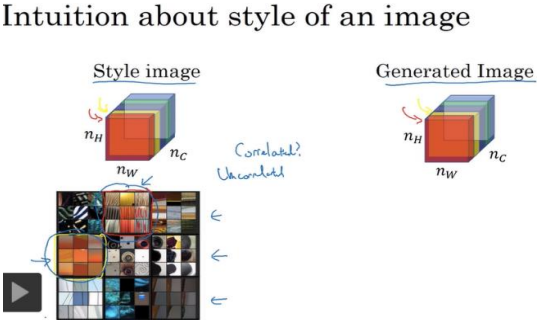
>1. Idea:

Summarize:
The correlation tells which of these high level texture components tend to occur or not occur together in part of an image and the degree of correlation 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 and don't occur together in different parts of an image.

Generated image:
If **use the degree of correlation between channels as a measure of the style**, then measure the degree in generated image.
This first channel is correlated or uncorrelated with the second channel and that will tell you in the generated image how often this type of vertical texture occurs or doesn't occur with this orange-ish tint .

This gives a measure of how similar is the style of the generated image to the style of the input style image.



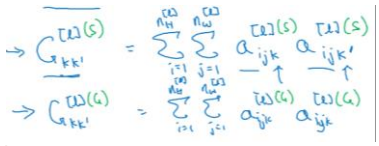


2. Style matrix:

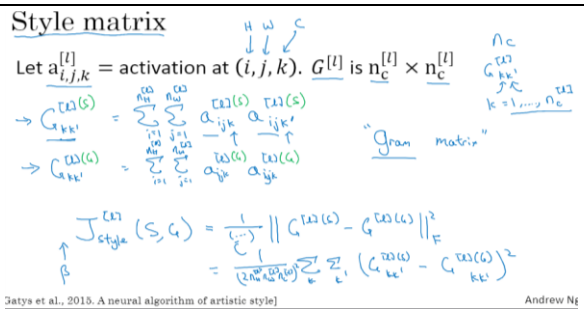
layer : l; channel: k; p
activation a[l]_i,j,k: a

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

>1. Channel k, k' correlation for style image and generated image:G[l]_k, k'
Measure how correlated are the activations in channel k compared to the activations in channel k'.
Take a pair of activation values **in same postioin of channel k, k'" and multiply them a[l]_ij,k * a[l]_i,j,k'** and then sum over all nh x nw postions'



>2. Styple matrix for one image: G[l]_S, G[l]_G
take layer l activations to compute image style: G[l][s] is n[l]_c x n[l]_c dimension.
take Channel k, k' correlation in layerl activation, , go through all channels in layer l activation. As have nc channels so have an nc x nc dimensional matrix in order to measure how correlated each pair of them-channel is.
(note: styple matrix is 实对称:
[G_11, G_12 , G13, ...G_1k', ...G_1n_c
G_21, G_22 , G23, ...G_2k', ...G_2n_c
....
G_n_c1, G_n_c2 , Gn_c3, ...G_n_ck', ...G_n_cxn_c]

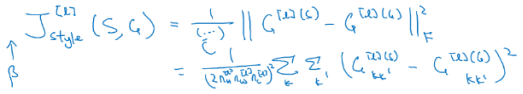


Note:
1. if both of these activation tend to be large together, G[l]_k,k' will be large. if they are uncorrelated, then G[l]_k,k' might be small.
(how to veryfy this is good representation of image style?)

2. This is actually the unnormalized cross of the areas because we're not subtracting out the mean and this is just multiplied by these elements directly.
(no need to normalize all channels in layer l ?)

(G[l]_k,k' : kind of like conv computation: filter same size as input volume (only two channel k, k' here)-element wise filter and channel values and then sum, result in one value representing image feature caputured by filter;
while G[l]_k,k' no filter, element wise channel's own values and then sum, representing image styles in this layer.).

3. Style cost function in layer l:

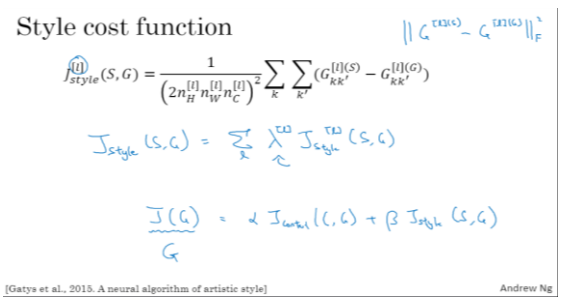


Defination: the difference between these two matrices, G[l]_S, G[l]_G: the sum of squares of the element wise differences between these two matrices : square of (G[l]_k,k'_S - G[l]_k,k'_G) , and sum over k, k'.
= norm||G[l]_s - G[l]_G||

Note:
1. paper authors actually used normalization constant 1/ (2*nh *nw*nc)^2.
But a normalization constant doesn't matter that much because this causes multiplied by some hyperparameter β anyway.

Summary:
This style cost function defined using layer l, cost function is basically the Frobenius norm between the two style matrices computed on the image s and on the image G , Frobenius on squared and divide by normalization constants, which isn't that important.

And it turns out that you get more visually pleasing results if use the style cost function from multiple different layers.



4. Overall style cost functionJ (S,G)_style:

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

Defination: sum over all the different layers of the style cost function for that layer J[L]_(S,G)_style, weighted by some set of additional hyperparameters, λ[l].
So it allows to use different layers in a neural network: both the early ones, which measure relatively simpler low level features like edges as well as some later layers, which measure high level features and **cause neural network to take both low level and high level correlations into account when computing style.**

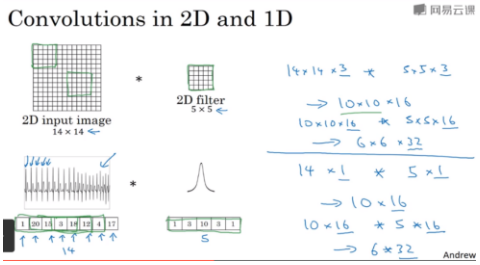
And, in the following exercise, you gain more intuition about what might be reasonable choices for this type of parameter λ[l] as well.

5. Overal cost function: J(G) = a *J(C, G) +β* J (S, G)
Could use gradient descent or a more sophisticated optimization algorithm if you want in order to try to find an image G that to minimize this cost function J(G). and if you do that you'll be able to generate some pretty nice novel artwork.

4.4-11: 1D & 3D generalizations of models.

Have learned a lot about ConvNets, everything ranging from the architecture of the ConvNet to how to use it for image recognition, to object detection, to face recognition and neural-style transfer.
Yet most of the discussion has focused on images, on sort of 2D datae. It turns out that many of these ideas have learned about also apply, not just to 2D images but also to 1D data as well as to 3D data.

1. convolution in 2D & 1D



>1. 2D convolution:
e.g: image 14x14, filter 5x5, 16 filter-->result 10x10x16
or image 14x14x3, --filter 5x5x3, 16 filter -->result 10x10x16.

Same method could appply on 1D data.

>2. 1D convoluation:
Mehtod for 2D convo. (n_h, h_w) * n_c, could same used in 1D (n_h)*n_c
just rememove n_w dimension, same calcualtion. Just Take n_w=1.
e.g : EKG signal: heart beating vs time series. different postion (-diffrent channel)heart beating vs time

Note: filter channel still need to be same as 1D image channel.
>2. 1D convoluation:

E.g: EKG signal- electrocardioagram. heart beating vs time, showing the voltage at each instant in time.
input is 14 (in sequence), ---> convolve this with a 1 dimensional filter, rather than the 5 x 5, just have 5 (x1) dimensional filter.

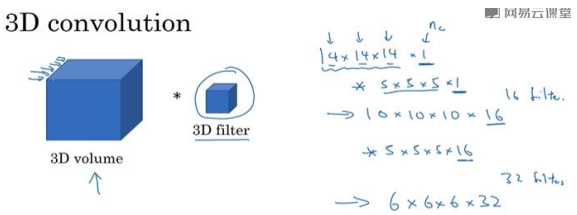
In 2D data convolution, apply the same 5 x 5 feature detector across at different positions throughout the image. And end up with your 10 x 10 output.
In 1D data covolution: apply 5 dimensional filter similarly in lots of different positions throughout this 1D signal.will end up 10 dimensional output (10x1).

Note:
If have multiple channels in 1D data, filter have same channel number.
e.g: input 10 x 16 dimensional input, filter -5 dimensional with 16 channels, 32 filters, then end up 6 x 32,

So generalization from 2D to 1D, all of 2D convolution ideas apply also to 1D data, having the same feature detector apply to a variety of positions

2. 3D Convolution:

>1. Input: 3D block, a three dimensional input
e.g: CT scan, three dimensional model of body. so as scan through a CT scan, you can look at different slices of the human torso to see how they look .



>2. 3D convolution idea:
Same idea of 2D convolution, to apply a ConvNet to detect features in this three dimensional image
e.g: input 14 x 14 x 14 x32 32 channel, filer 5x5x5, 32 channel, 16 filter, end up with a 6 x 6 x 6 volume across 32 channels.

So 3D data can also be learned on, sort of directly using a three dimensional ConvNet. And what these filters do is really detect features across your 3D data

e.g2: 3D movie , different slices 3D data in time through a movie. Could use this to detect motion or people taking actions in movies.

>3. Note:
1. 3D input convlution: filter is also 3D. yet filter dimension not same as input dimension esp. in the 3rd direction: only filter channel is = input channel.
Apply same volume filter in different position of horizental, veritial and volume directions.

5.1 - Convolutional layer backward pass

5.1.1 - Computing dA:

This is the formula for computing dA with respect to the cost for a certain filter W_c and a given training example:

$$dA += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw} \tag{1}$$

Where W_c is a filter and dZ_{hw} is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down). Note that at each time, we multiply the the same filter W_c by a different dZ when updating dA. We do so mainly because when computing the forward propagation, each filter is dotted and summed by a different a_slice. Therefore when computing the backprop for dA, we are just adding the gradients of all the a_slices.

In code, inside the appropriate for-loops, this formula translates into:

```
da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
```

5.1.2 - Computing dW:

This is the formula for computing dW_c (dW_c is the derivative of one filter) with respect to the loss:

$$dW_c += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw} \tag{2}$$

Where a_{slice} corresponds to the slice which was used to generate the acitivation Z_{ij}. Hence, this ends up giving us the gradient for W with respect to that slice. Since it is the same W, we will just add up all such gradients to get dW.

In code, inside the appropriate for-loops, this formula translates into:

```
dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
```

5.1.3 - Computing db:

This is the formula for computing db with respect to the cost for a certain filter W_c:

$$db = \sum_h \sum_w dZ_{hw} \tag{3}$$

As you have previously seen in basic neural networks, db is computed by summing dZ. In this case, you are just summing over all the gradients of the conv output (Z) with respect to the cost.

In code, inside the appropriate for-loops, this formula translates into:

```
db[:,:,:,c] += dZ[i, h, w, c]
```

5.2 Pooling layer - backward pass

Next, let's implement the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, you still need to backpropagation the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

5.2.1 Max pooling - backward pass

Before jumping into the backpropagation of the pooling layer, you are going to build a helper function called `create_mask_from_window()` which does the following:

$$X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \quad \rightarrow \quad M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{4}$$

As you can see, this function creates a "mask" matrix which keeps track of where the maximum of the matrix is. True (1) indicates the position of the maximum in X, the other entries are False (0). You'll see later that the backward pass for average pooling will be similar to this but using a different mask.

Exercise: Implement `create_mask_from_window()`. This function will be helpful for pooling backward.
Hints:

- np.max() may be helpful. It computes the maximum of an array.
- If you have a matrix X and a scalar x: `A = (X == x)` will return a matrix A of the same size as X such that:

```
A[i,j] = True if X[i,j] = x
A[i,j] = False if X[i,j] != x
```

- Here, you don't need to consider cases where there are several maxima in a matrix.

复制

```
def create_mask_from_window(x):
    """
    Creates a mask from an input matrix x, to identify the max entry of x.

    Arguments:
    x -- Array of shape (f, f)

    Returns:
    mask -- Array of the same shape as window, contains a True at the position corresponding to the max entry of x.
    """

    ### START CODE HERE ### (≈1 line)
    mask = (x == np.max(x))
    ### END CODE HERE ###

    return mask
```

```
np.random.seed(1)
x = np.random.randn(2,3)
mask = create_mask_from_window(x)
print('x = ', x)
print("mask = ", mask)
```

```
x =  [[ 1.62434536 -0.61175641 -0.52817175]
 [-1.07296862  0.86540763 -2.3015387 ]]
mask =  [[ True False False]
 [False False False]]
```

Why do we keep track of the position of the max? It's because this is the input value that ultimately influenced the output, and therefore the cost. Backprop is computing gradients with respect to the cost, so anything that influences the ultimate cost should have a non-zero

gradient. So, backprop will "propagate" the gradient back to this particular input value that had influenced the cost.

5.2.2 - Average pooling - backward pass

In max pooling, for each input window, all the "influence" on the output came from a single input value–the max. In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix} \tag{5}$$

This implies that each position in the dZ matrix contributes equally to output because in the forward pass, we took an average.

Exercise: Implement the function below to equally distribute a value dz through a matrix of dimension shape. Hint

```
def distribute_value(dz, shape):
    """
    Distributes the input value in the matrix of dimension shape

    Arguments:
    dz -- input scalar
    shape -- the shape (n_H, n_W) of the output matrix for which we want to distribute the value of dz

    Returns:
    a -- Array of size (n_H, n_W) for which we distributed the value of dz
    """

    ### START CODE HERE ###
    # Retrieve dimensions from shape (≈1 line)
    (n_H, n_W) = shape

    # Compute the value to distribute on the matrix (≈1 line)
    average = dz / (n_H * n_W)

    # Create a matrix where every entry is the "average" value (≈1 line)
    a = average * np.ones(shape)
    ### END CODE HERE ###

    return a
```

```
a = distribute_value(2, (2,2))
```

```
print('distributed value =', a)
```

```
distributed value = [[ 0.5  0.5]
 [ 0.5  0.5]]
```

Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different. You are going to implement both of them.

2.1 - The identity block

The identity block is the standard block used in ResNets, and corresponds to the case where the input activation (say $a^{[l]}$) has the same dimension as the output activation (say $a^{[l+2]}$). To flesh out the different steps of what happens in a ResNet's identity block, here is an alternative diagram showing the individual steps:

Figure 3 : **Identity block.** Skip connection "skips over" 2 layers.

2.2 - The convolutional block

You've implemented the ResNet identity block. Next, the ResNet "convolutional block" is the other type of block. You can use this type of block when the input and output dimensions don't match up. The difference with the identity block is that there is a CONV2D layer in the shortcut path:

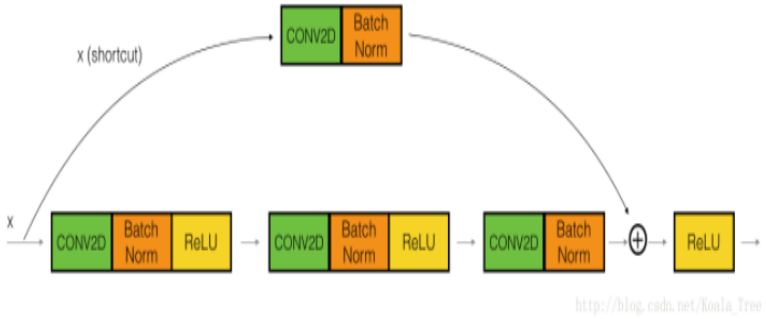


Figure 4 : **Convolutional block**

The CONV2D layer in the shortcut path is used to resize the input x to a different dimension, so that the dimensions match up in the final addition needed to add the shortcut value back to the main path. (This plays a similar role as the matrix W_s discussed in lecture.) For example, to reduce the activation dimensions's height and width by a factor of 2, you can use a 1x1 convolution with a stride of 2. The CONV2D layer on the shortcut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.

2 - YOLO

YOLO ("you only look once") is a popular algoritm because it achieves high accuracy while also being able to run in real-time. This algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

2.1 - Model details

First things to know:

- The **input** is a batch of images of shape (m, 608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

We will use 5 anchor boxes. So you can think of the YOLO architecture as the following: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

Lets look in greater detail at what this encoding represents.

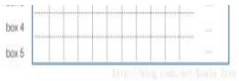


Figure 2 : Encoding architecture for YOLO

If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Since we are using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height.

For simplicity, we will flatten the last two last dimensions of the shape (19, 19, 5, 85) encoding. So the output of the Deep CNN is (19, 19, 425).

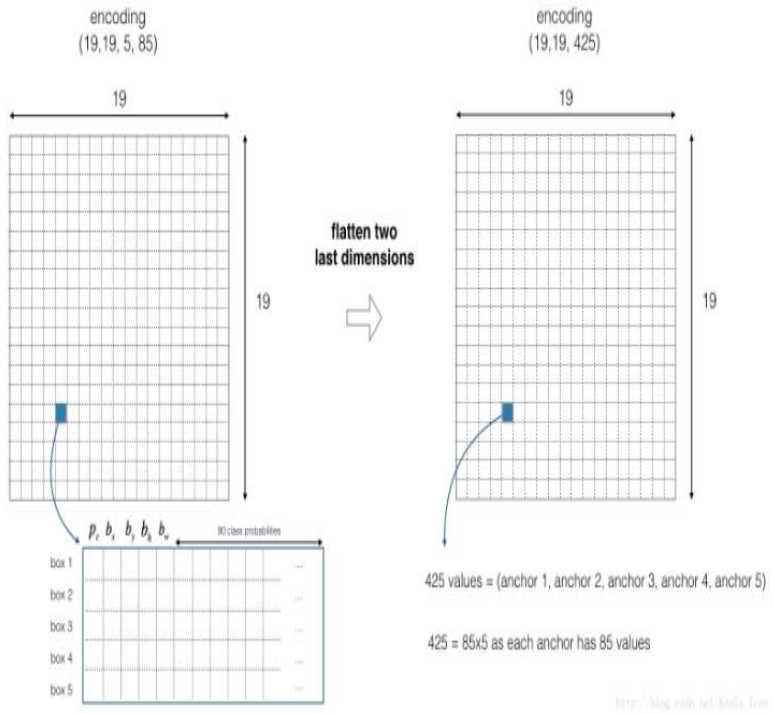


Figure 3 : Flattening the last two last dimensions

Now, for each box (of each cell) we will compute the following elementwise product and extract a probability that the box contains a certain class.

3 - Neural Style Transfer

We will build the NST algorithm in three steps:

- Build the content cost function $J_{content}(C,G)$
- Build the style cost function $J_{style}(S,G)$
- Put it together to get $J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$.

3.1 - Computing the content cost

In our running example, the content image C will be the picture of the Louvre Museum in Paris. Run the code below to see a picture of the Louvre.

** 3.1.1 - How do you ensure the generated image G matches the content of the image C?**

As we saw in lecture, the earlier (shallower) layers of a ConvNet tend to detect lower-level features such as edges and simple textures and the later (deeper) layers tend to detect higher-level features such as more complex textures as well as object classes.

We would like the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image. In practice, you'll get the most visually pleasing results if you choose a layer in the middle of the network–neither too shallow nor too deep. (After you have finished this exercise, feel free to come back and experiment with using different layers, to see how the results vary.)

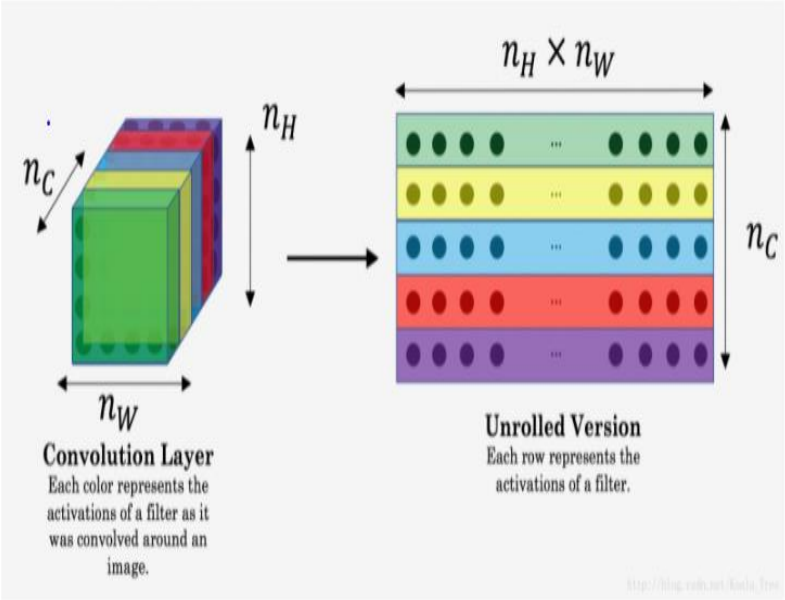
So, suppose you have picked one particular hidden layer to use. Now, set the image C as the input to the pretrained VGG network, and run forward propagation. Let $a^{(C)}$ be the hidden layer activations in the layer you had chosen. (In lecture, we had written this as $a^{[l](C)}$ but here we'll drop the superscript $[l]$ to simplify the notation.) This will be a $n_H \times n_W \times n_C$ tensor. Repeat this process with the image G: Set G as the input, and run forward progation. Let

$$a^{(G)}$$

be the corresponding hidden layer activation. We will define as the content cost function as:

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

Here, n_H, n_W and n_C are the height, width and number of channels of the hidden layer you have chosen, and appear in a normalization term in the cost. For clarity, note that $a^{(C)}$ and $a^{(G)}$ are the volumes corresponding to a hidden layer's activations. In order to compute the cost $J_{content}(C, G)$, it might also be convenient to unroll these 3D volumes into a 2D matrix, as shown below. (Technically this unrolling step isn't needed to compute $J_{content}$, but it will be good practice for when you do need to carry out a similar operation later for computing the style const J_{style}.)



Exercise: Compute the "content cost" using TensorFlow.

Instructions: The 3 steps to implement this function are:

1. Retrieve dimensions from a_G:

- To retrieve dimensions from a tensor X, use: `X.get_shape().as_list()`

2. Unroll a_C and a_G as explained in the picture above

- If you are stuck, take a look at Hint1 and Hint2.

3. Compute the content cost:

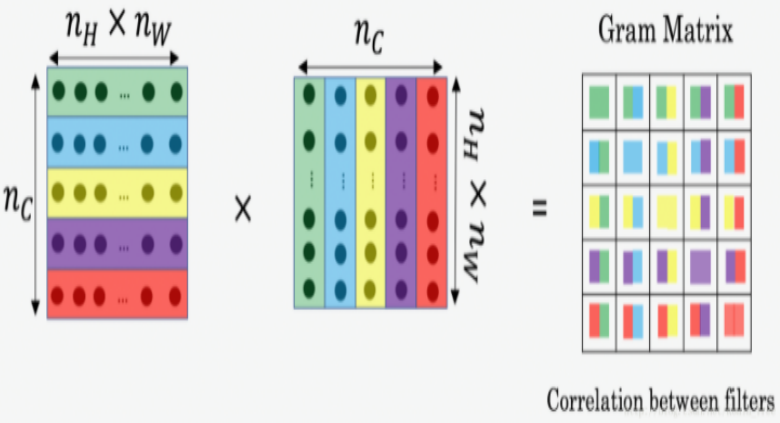
- If you are stuck, take a look at Hint3, Hint4 and Hint5.

3.2.1 - Style matrix

The style matrix is also called a "Gram matrix." In linear algebra, the Gram matrix G of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$. In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large.

Note that there is an unfortunate collision in the variable names used here. We are following common terminology used in the literature, but G is used to denote the Style matrix (or Gram matrix) as well as to denote the generated image G. We will try to make sure which G we are referring to is always clear from the context.

In NST, you can compute the Style matrix by multiplying the "unrolled" filter matrix with their transpose:



The result is a matrix of dimension (n_C, n_C) where n_C is the number of filters. The value G_{ij} measures how similar the activations of filter i are to the activations of filter j.

One important part of the gram matrix is that the diagonal elements such as G_{ii} also measures how active filter i is. For example, suppose filter i is detecting vertical textures in the image. Then G_{ii} measures how common vertical textures are in the image as a whole: If G_{ii} is large, this means that the image has a lot of vertical texture.

By capturing the prevalence of different types of features (G_{ii}), as well as how much different features occur together (G_{ij}), the Style matrix G measures the style of an image.

3.2.2 - Style cost

After generating the Style matrix (Gram matrix), your goal will be to minimize the distance between the Gram matrix of the "style" image S and that of the "generated" image G. For now, we are using only a single hidden layer $a^{[l]}$, and the corresponding style cost for this layer is defined as:

layer is defined as:

$$J_{style}^{[l]}(S,G) = \frac{1}{4 \times {n_C}^2 \times (n_H \times n_W)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G_{ij}^{(S)} - G_{ij}^{(G)})^2 \tag{2}$$

where $G^{(S)}$ and $G^{(G)}$ are respectively the Gram matrices of the "style" image and the "generated" image, computed using the hidden layer activations for a particular hidden layer in the network.

Exercise: Compute the style cost for a single layer.

Instructions: The 3 steps to implement this function are:

1. Retrieve dimensions from the hidden layer activations a_G:

- To retrieve dimensions from a tensor X, use: `X.get_shape().as_list()`

2. Unroll the hidden layer activations a_S and a_G into 2D matrices, as explained in the picture above.

- You may find Hint1 and Hint2 useful.

3. Compute the Style matrix of the images S and G. (Use the function you had previously written.)

4. Compute the Style cost:

- You may find Hint3, Hint4 and Hint5 useful.

3.2.3 Style Weights

So far you have captured the style from only one layer. We'll get better results if we "merge" style costs from several different layers. After completing this exercise, feel free to come back and experiment with different weights to see how it changes the generated image G. But for now, this is a pretty reasonable default:

```
STYLE_LAYERS = [
    ('conv1_1', 0.2),
    ('conv2_1', 0.2),
    ('conv3_1', 0.2),
    ('conv4_1', 0.2),
    ('conv5_1', 0.2)]
```

You can combine the style costs for different layers as follows:

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

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

where the values for $\lambda^{[l]}$ are given in `STYLE_LAYERS`.

We've implemented a compute_style_cost(...) function. It simply calls your `compute_layer_style_cost(...)` several times, and weights their results using the values in `STYLE_LAYERS`. Read over it to make sure you understand what it's doing.

3.3 - Defining the total cost to optimize

Finally, let's create a cost function that minimizes both the style and the content cost. The formula is:

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

4 - Solving the optimization problem

Finally, let's put everything together to implement Neural Style Transfer!

Here's what the program will have to do:

1. Create an Interactive Session
2. Load the content image
3. Load the style image
4. Randomly initialize the image to be generated
5. Load the VGG16 model
6. Build the TensorFlow graph:
    1. Run the content image through the VGG16 model and compute the content cost
    2. Run the style image through the VGG16 model and compute the style cost
    3. Compute the total cost

3. Compute the total cost

4. Define the optimizer and the learning rate

7. Initialize the TensorFlow graph and run it for a large number of iterations, updating the generated image at every step.

,

d

,

l)