CONVOLUTION NEURAL NETWORK (CNN)
WEEK 1:

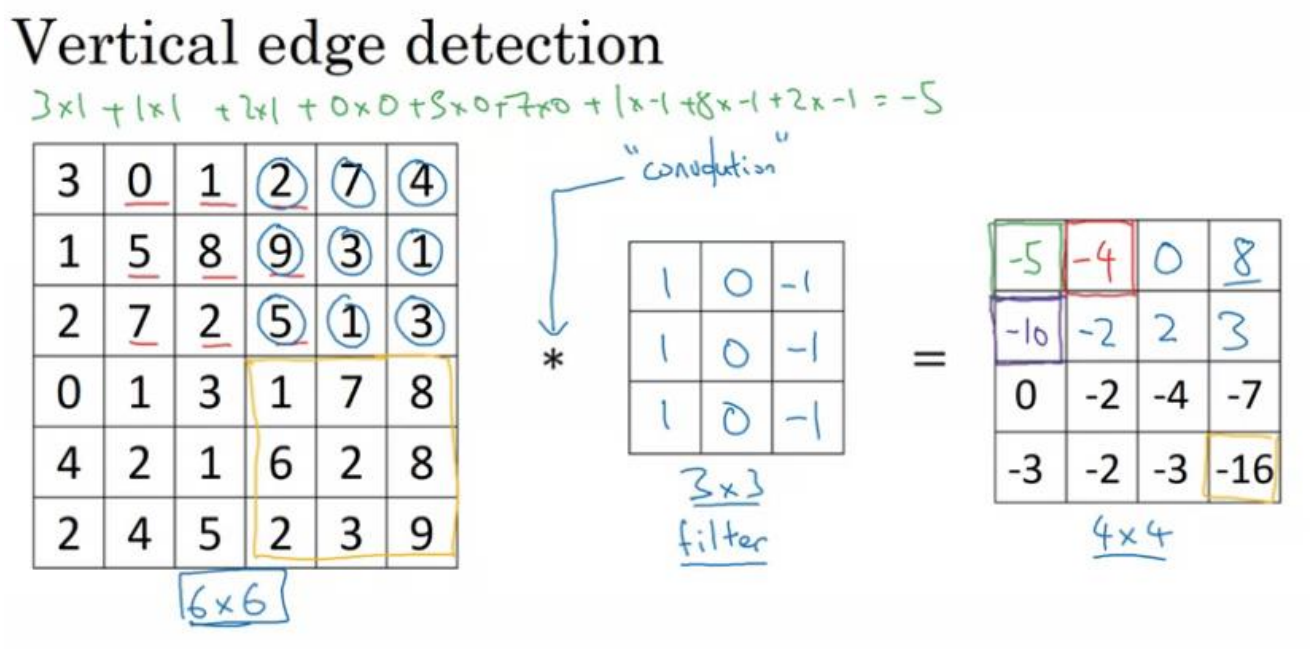
Edge Detection Example

For detecting edge, we convolve the image matrix with a filter or kernel. The way that the convolution works is that, you start from very top left of the image, and multiply it with the filter. Then we go one step to the right and do the same. We keep going to the right until we reach to the end. Then we go back to the first place and go one step down and start the whole operation of multiplication and going right. I mean, this is the convolution operation, nothing weird for real. XD

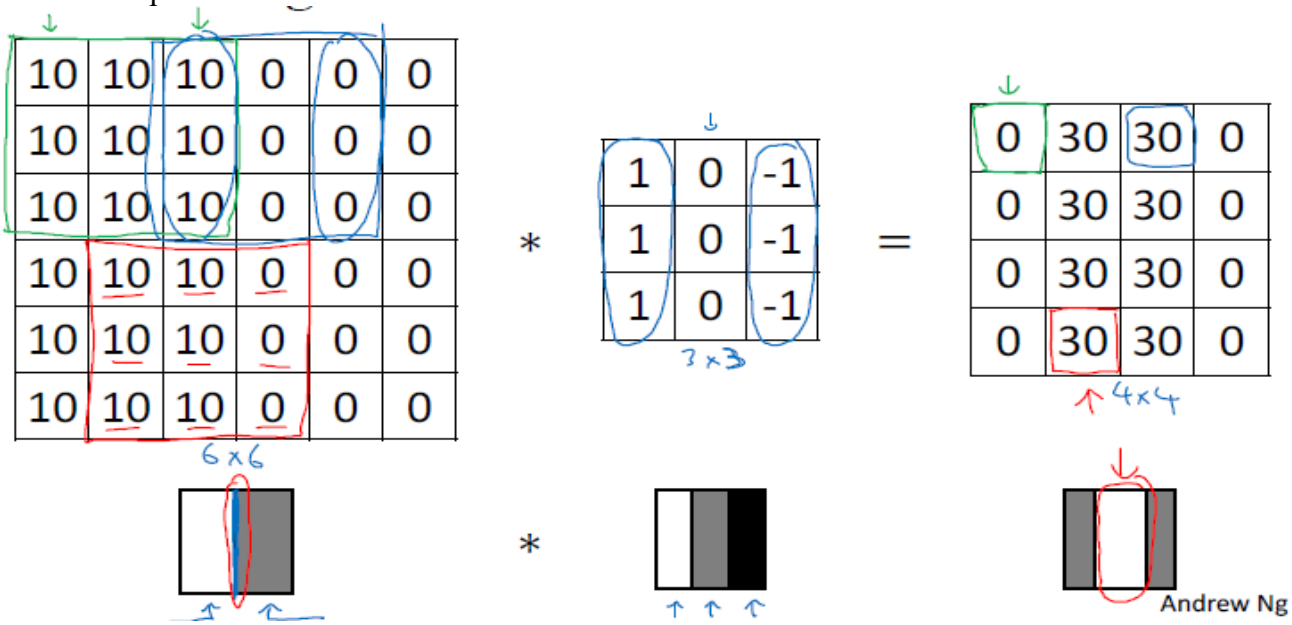
For an example, the very first 3 by 3 cells of the image convolution with the filter would be:

$$3 \times 1 + 1 \times 1 + 2 \times 1 + 0 \times 0 + 5 \times 0 + 7 \times 0 + 1 \times -1 + 8 \times -1 + 2 \times -1 = -5$$

So 6 by 6 image by 3 by 3 filter would be 4 by 4 image.



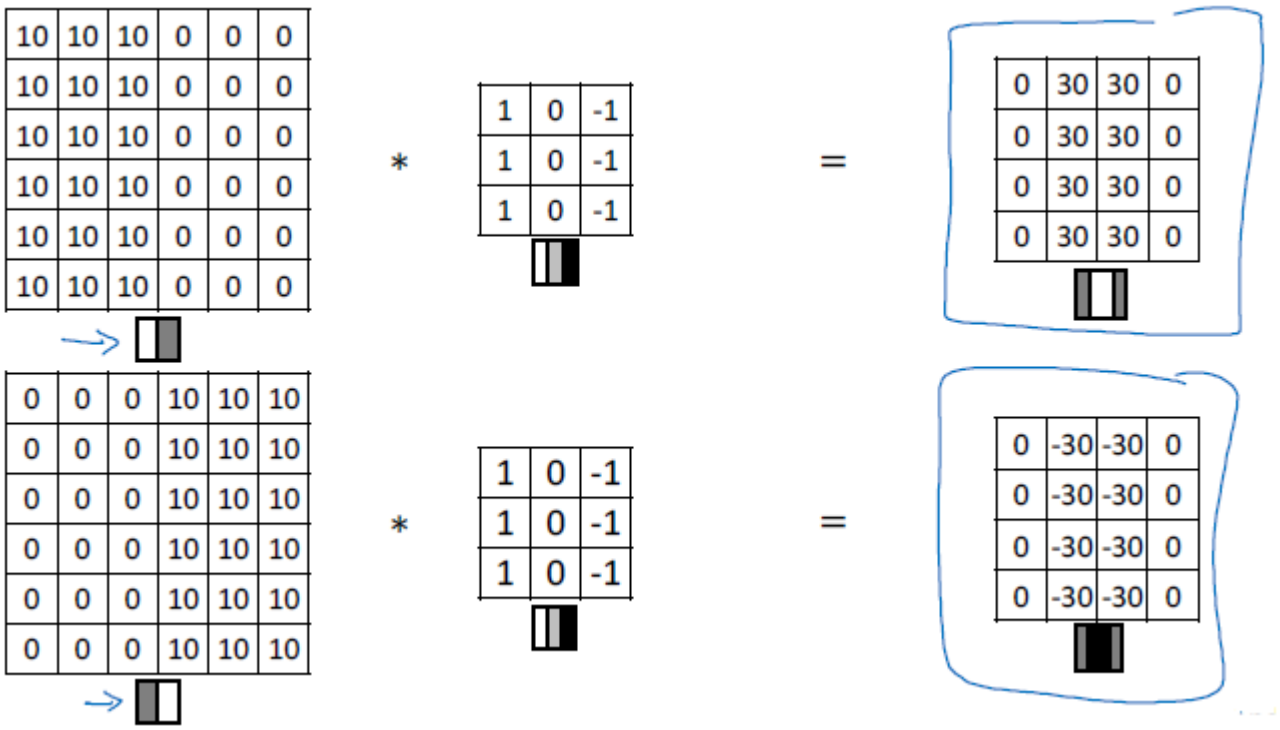
Another example would be:



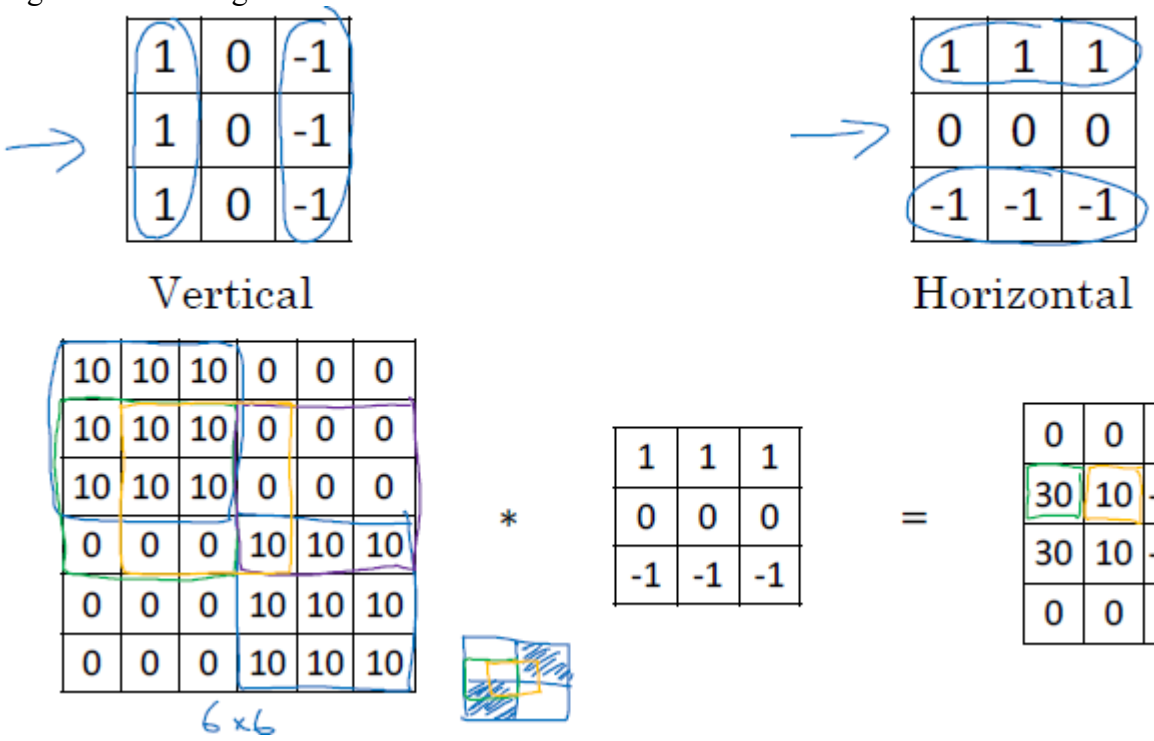
In the example above, the vertical edge of the actual image is so obvious. We see after the convolution, we easily detect it. Of course the edges of the last image is so huge, that's because we use a small image. We can manipulate it later. The concept is important here only.

More Edge Detection

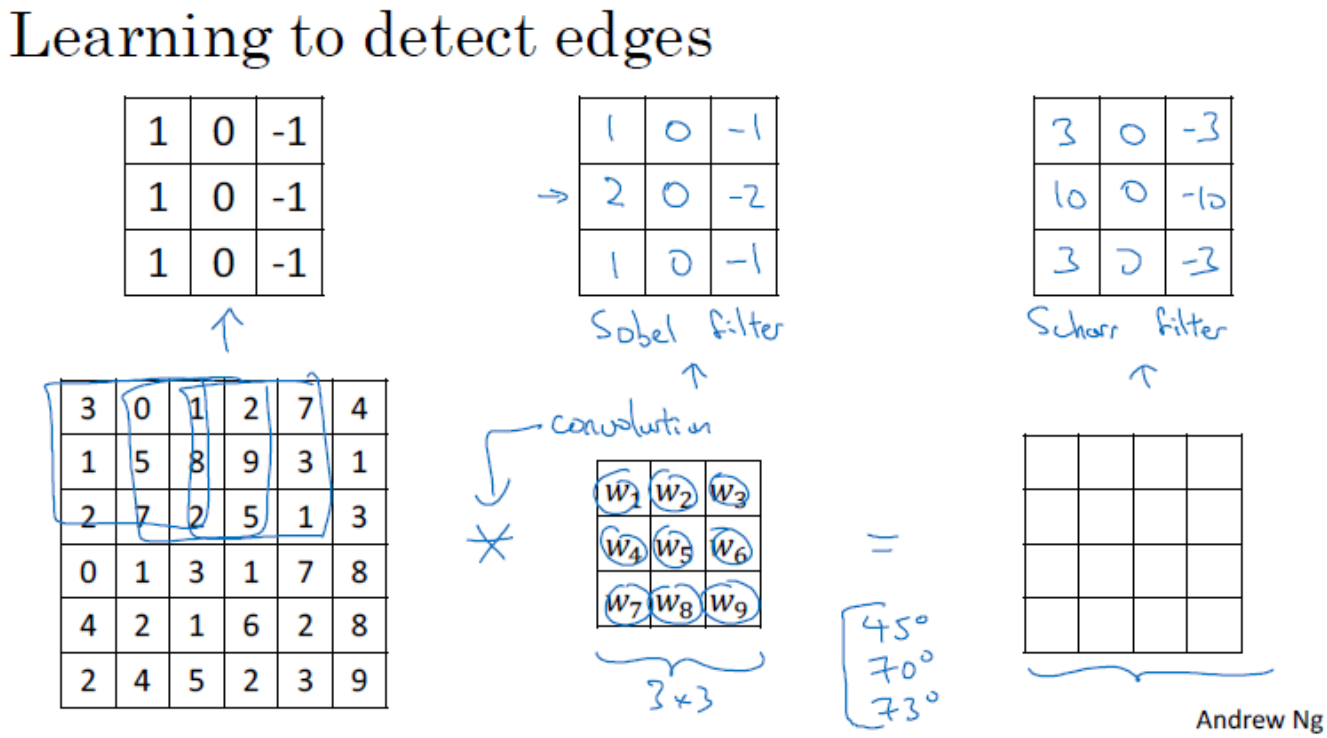
For the dark to light or light to dark edge detection, we can see below example. In many application, it might not be a big deal for us, so we can take the absolute value of the end result image.



So for horizontal edge detection, we also have the below. As we saw before, the positive values are brighter and the negative ones are darker.

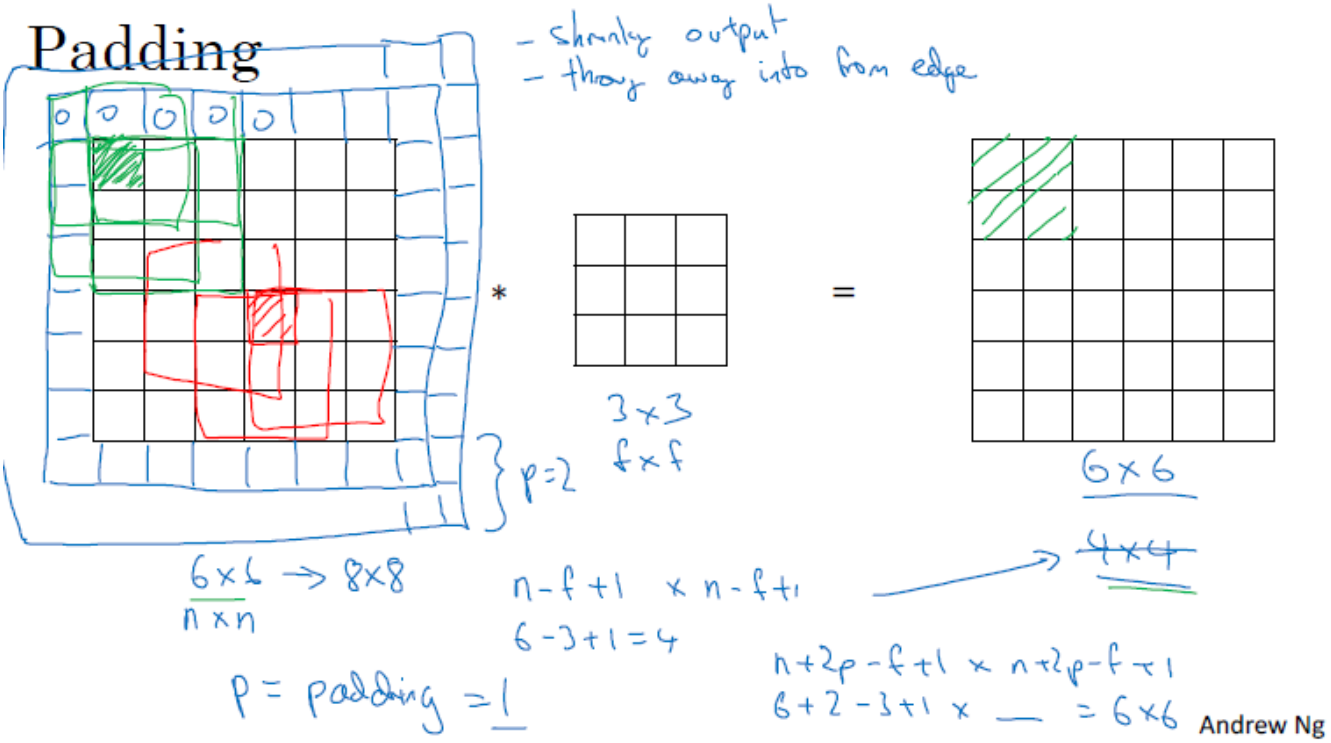


We historically have a “Sobel filter” which gives higher weight to the central pixels and it is a little bit more robust. The idea is the same, and we can get other filters like “Scharr filter”.
The other thing is that, we don’t need to choose the 9 numbers. We can learn them and use them as parameters to learn at the very first place. The goal would be giving the image and the final detected edges and learn 9 values. Its good thing is that we can come up with more robust filters. It can also learn other than vertical and horizontal edges.



Padding

So far, we saw how filters work. But we also need to know its dimensions before hand. As obvious it is, we can come up with this formulation: having $n \times n$ image, convolving with $f \times f$ filter or kernel, we get $(n - f + 1) \times (n - f + 1)$ image.
So, as the formula suggests, the operation shrinks the image, it is more critical when we use deep networks with many many layers. If it shrinks every time, we will end up with a very small image. Moreover, not all pixels in the image are contributing to the end result with the same weight. For example the corner pixels only contribute once as the middle ones contribute many more times. So we loose some information.
To solve the shrinking problem, we can use "padding" which is an extra border. Doing this we gonna have a new formulation as: having $(n + 2p) \times (n + 2p)$ image, convolving with $f \times f$ filter or kernel, we get $(n + 2p - f + 1) \times (n + 2p - f + 1)$ image. The padding can be any number.



For a "Valid" convolution we have no padding which is our very first formulation. For a "Same" convolution, padding will be added in a way that output size stays the same as the input size.

Note that, the only way that the padding is symmetric is that the size of the filter is odd, otherwise it would be asymmetric.
To clear it out, if the n is even and as $2p$ (symmetric) is even, adding 1 will make it odd. Now the filter should be odd to have an even output size as the input size. If n is odd, adding it up a 1 and an even value will give us an even value. If the filter size is an even value again, the output size would be odd as well with symmetric padding again. It won't be the case when the filter has an odd size and the padding will be asymmetric then:

$$\pm odd\ size\ filter + 1 = even \mid symmetric\ padding\ is\ even$$

$$even + even = even$$

$$even + odd/even\ input = odd/even\ output$$

Now if the filter size is even, the padding size should be odd to save the same property of last formula. It is only a convention at any event.

Valid and Same convolutions

"Valid": $n \times n \quad * \quad f \times f \quad \rightarrow \quad n-f+1 \times n-f+1$

$6 \times 6 \quad * \quad 3 \times 3 \quad \rightarrow \quad 4 \times 4$

"Same": Pad so that output size is the same as the input size.

$n+2p-f+1 \times n+2p-f+1$

$n+2p-f+1 = n \Rightarrow p = \frac{f-1}{2}$

$3 \times 3 \quad p = \frac{3-1}{2} = 1$

$5 \times 5 \quad p = 2$

f is usually odd

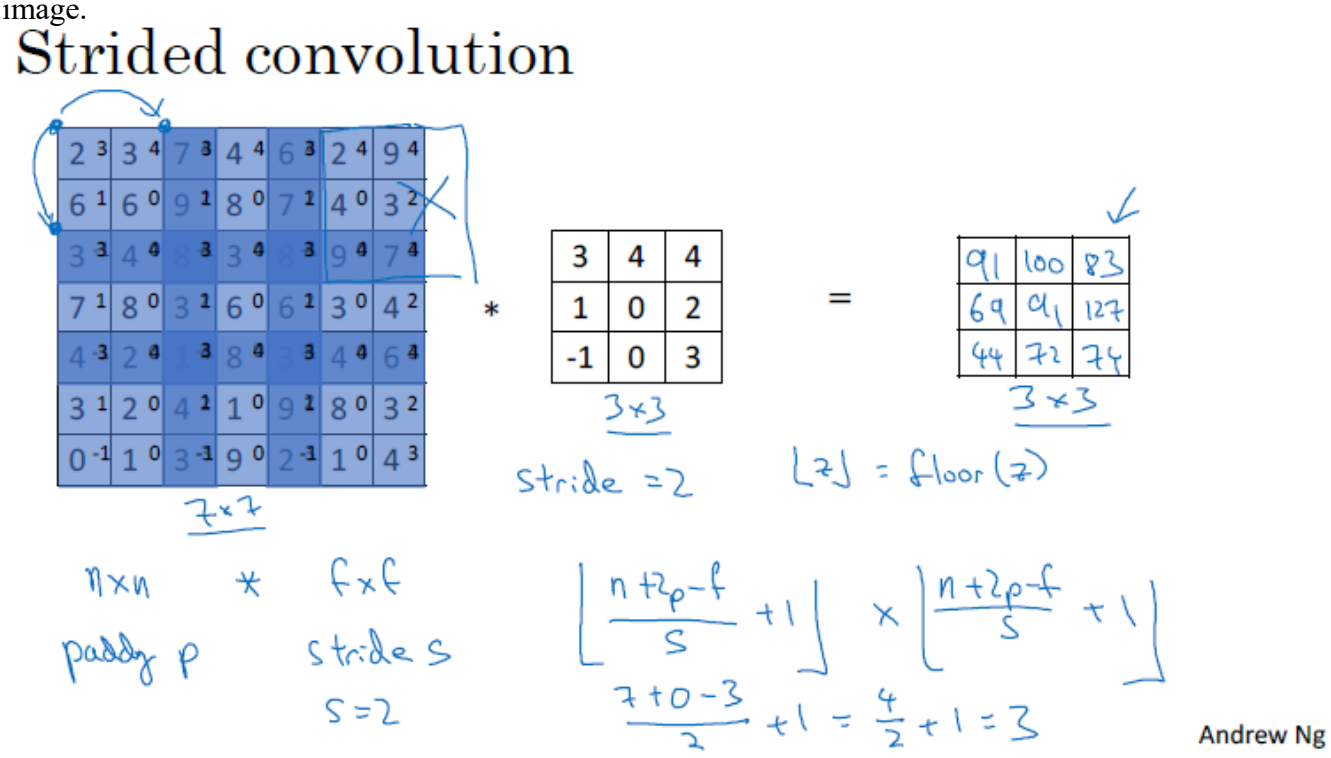
1x1
3x3
5x5
7x7

Andrew Ng

Strided Convolutions

The "stride" is about jumping over the actual image for the convolution task. So far the assumption was that we have stride as 1. The new formulation would be:

having $n \times n$ image, convolving with $f \times f$ filter or kernel, we get $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ size image.



Summary of convolutions

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

padding p $\quad$ stride s

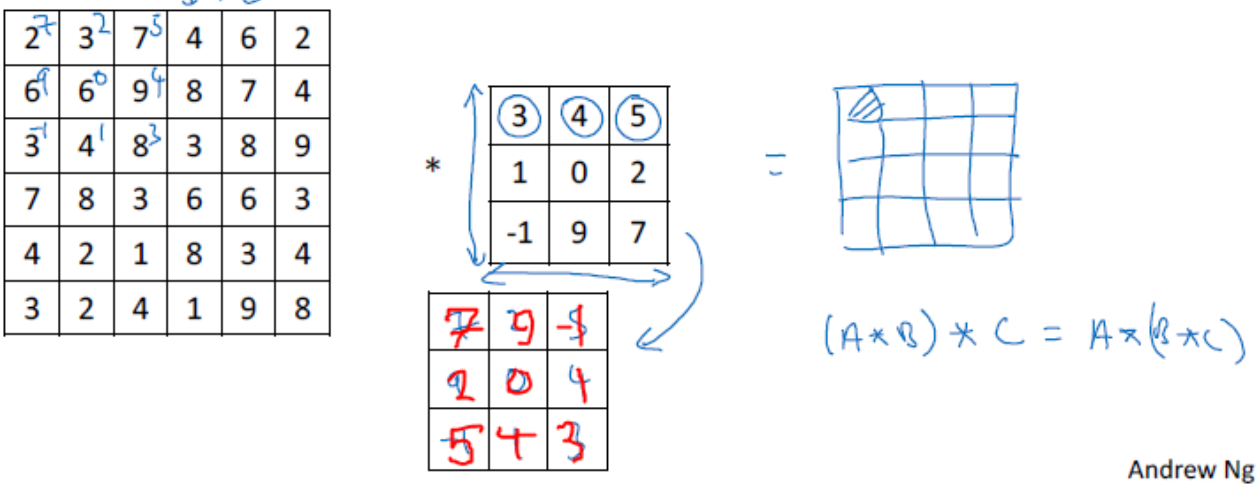
Output Size:

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

There is one technicality. Mostly, in signal processing literature, the convolution happens after inverting the row and column (flipping) of one of the matrices, and then passing them through each other. If we skip this step, it is cross-correlation, not convolution. However, both are the same for us as we learn the filter through parameterizationing of it. So we can still call it convolution.

Technical note on cross-correlation vs. convolution

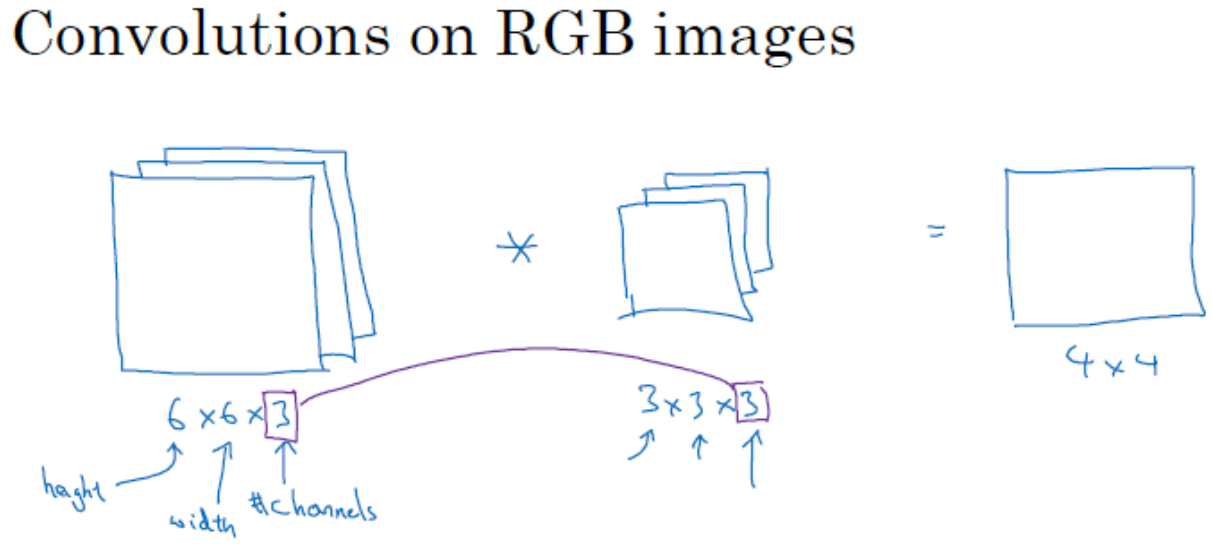
Convolution in math textbook:



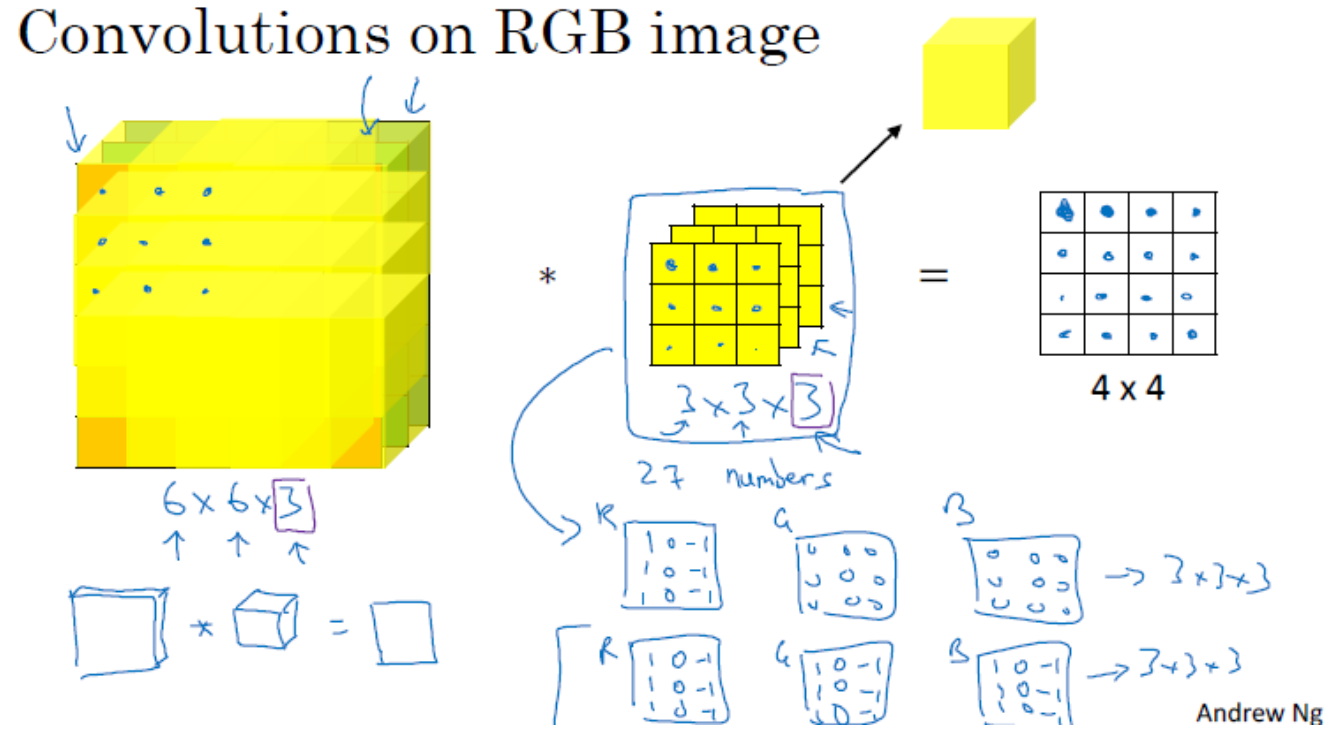
Andrew Ng

Convolutions Over Volume

As our images have three colors of RGB, we use 3 channels for this purpose. To this end, we need our filters to have the same number of channels.

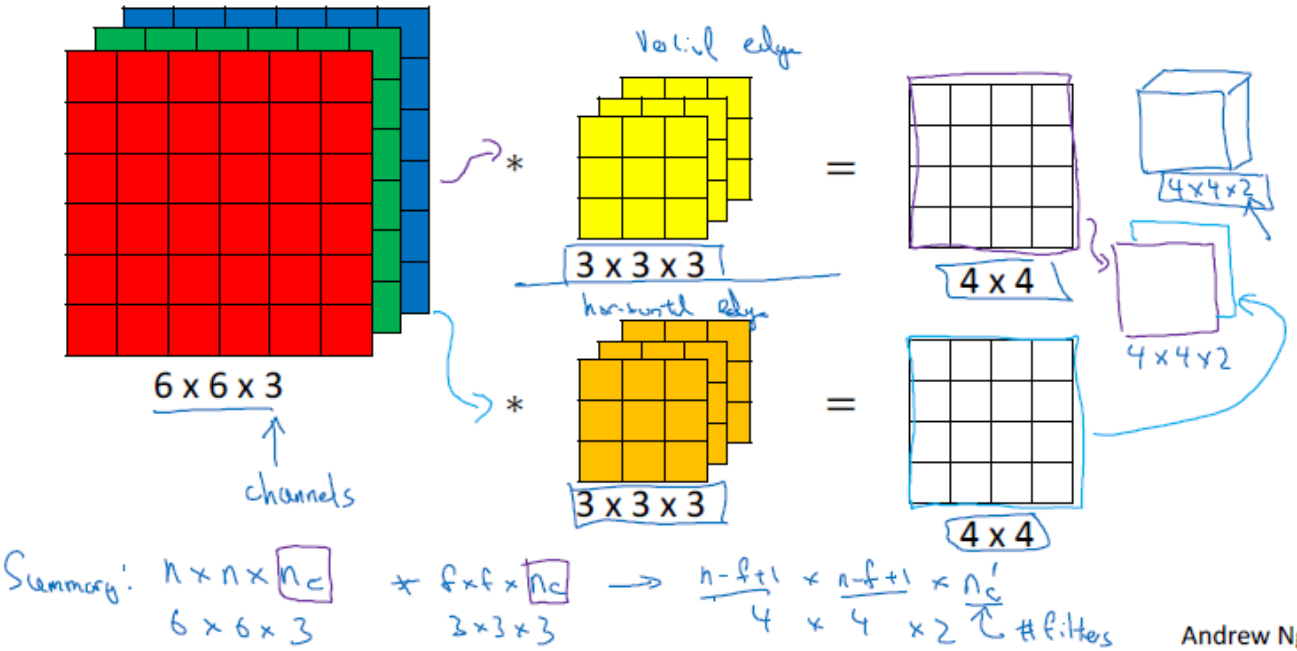


For detecting the edges for each color or for all of them, the technique stays the same.



Now we may want to use filters for detecting edges with different angles. Then we convolve the image with different filters and then stack them their output.

Multiple filters



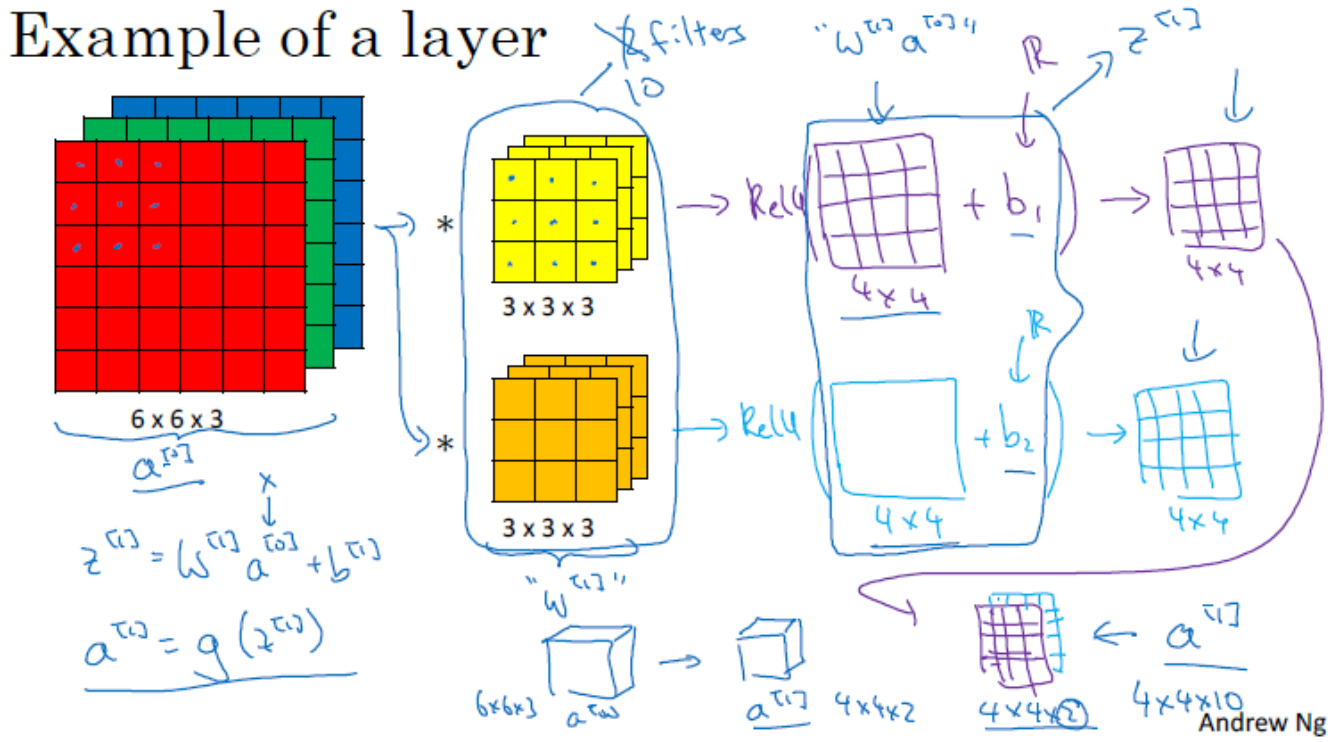
Summary: $n \times n \times n_c$ * $f \times f \times n_c$ → $(n-f+1) \times (n-f+1) \times n_c'$

$6 \times 6 \times 3$ * $3 \times 3 \times 3$ → $4 \times 4 \times 2$ (# filters)

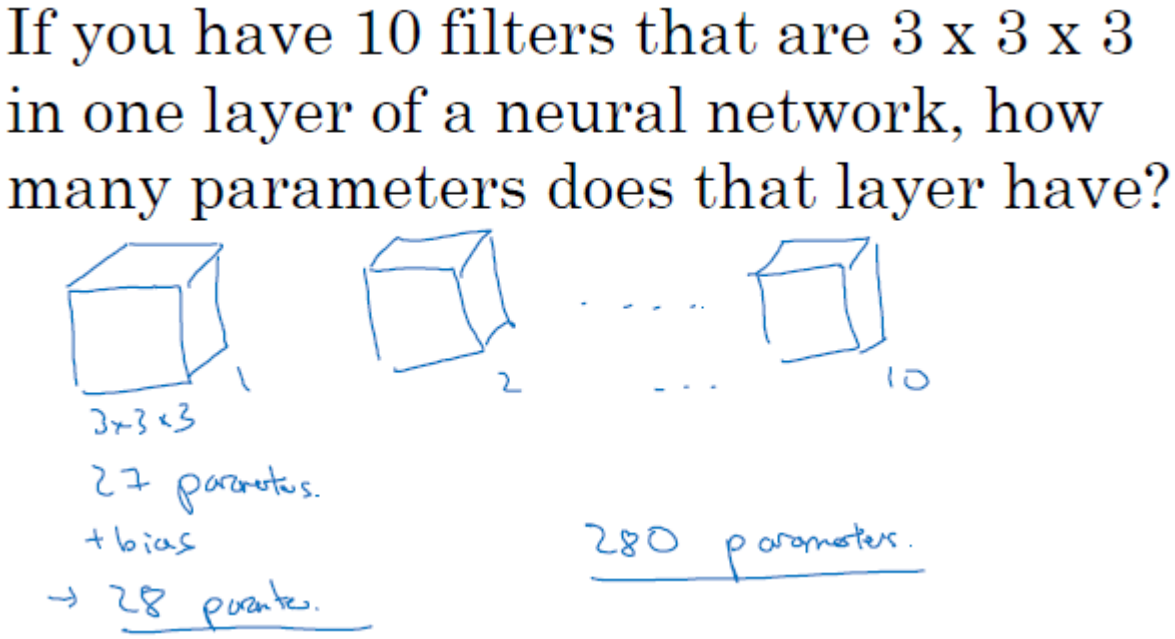
Andrew Ng

One Layer of a Convolutional Network

So for making the layers of convolution network, after applying filter on the input (previous layer), we use a bias and a nonlinearity (activation function). So rest of what we did so far stays the same. We can say the filter is playing the role of weights to some extent (intuitionally).



Let's go through an example. Having 10 filters of 3 by 3 by 3 in one layer of neural network, the number of parameters would be:
27 parameters for a filter times 10 filters, we get 270 parameters. Each one of the filters also have a bias, so it would be 280 parameters. So the input size does not even matter anymore which is one of the best properties of the CNN, increasing size of the input has no effect on number of parameters.



The notation summary is shown below. Note that the superscript is number of layer. For the input, we use a layer before the one we it is the activation from the previous layer. Also, “w” is for width, “H” height, and “c” the channel. So we can see the size of the new layer is computed with size of the previous layer with new layer padding, filter, and stride size.

Summary of notation

If layer l is a convolution layer:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = number of filters

→ Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

bias: $n_c^{[l]} - (1,1,1,n_c^{[l]})$ ← #filters in layer l.

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ ←

Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ ←

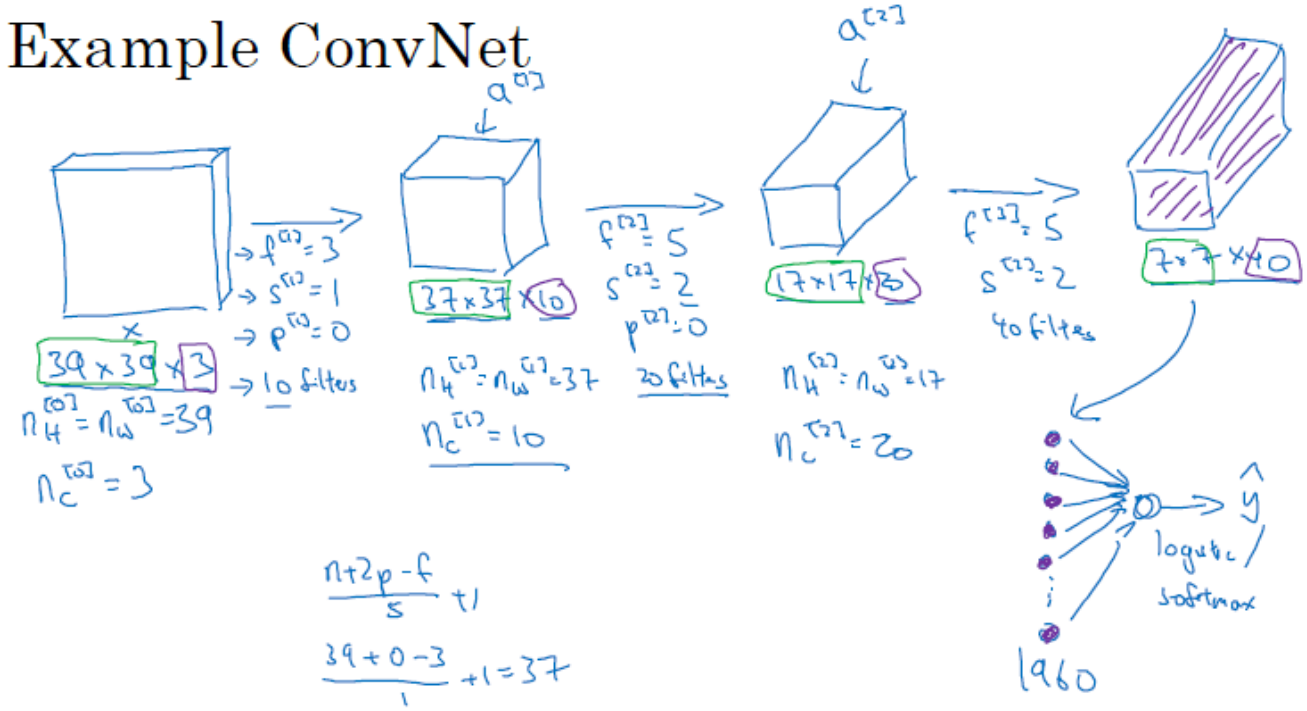
$$n_{H,W}^{[l]} = \left\lfloor \frac{n_{H,W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$n_c \times n_H \times n_W$

Simple Convolutional Network Example

Let's see a thorough example. Note that the last layer is converted to a typical layer of NN for prediction using Softmax activation function. This layer been called fully connected (FC) network. One more thing is that, we started from 39 by 39 by 3 and ended up with 7 by 7 by 40.



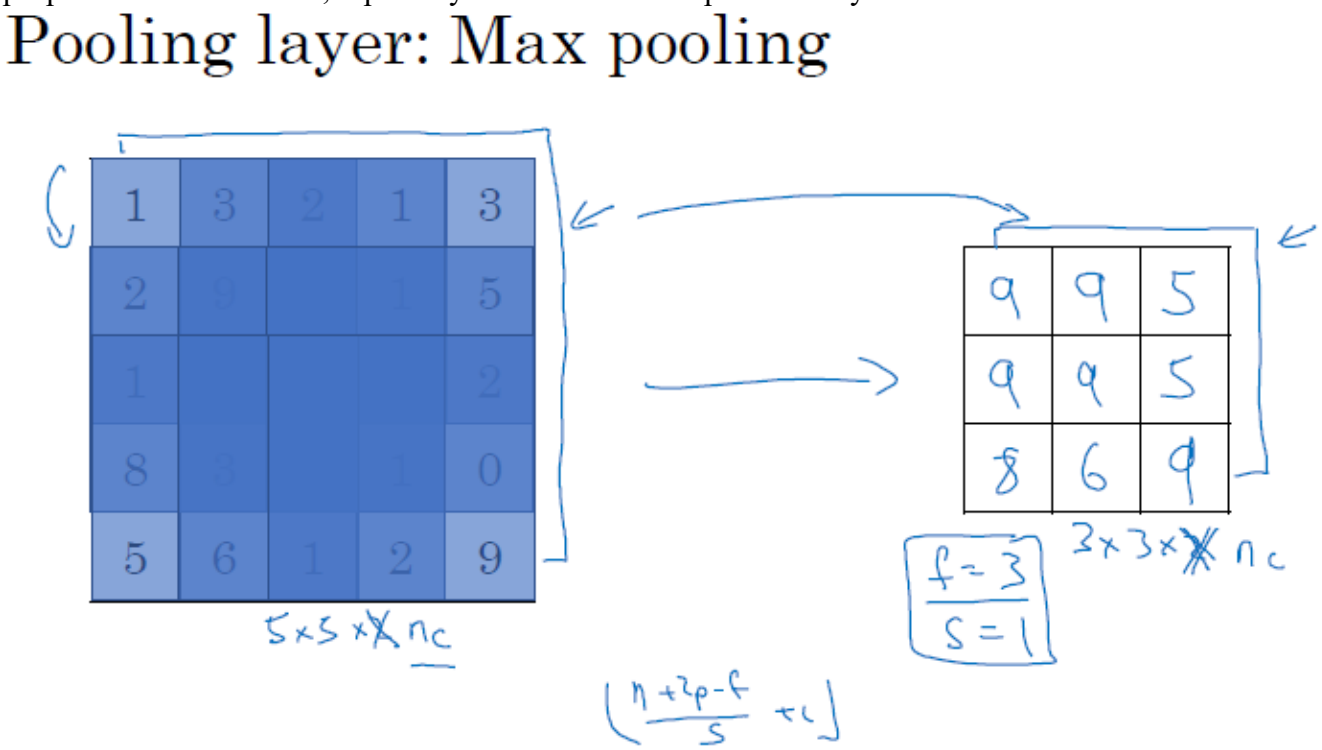
Types of layers we have in a convolution networks are:

1. Convolution
2. Pooling
3. Fully Connected (FC) network

The new ones are simpler than convolution an they are so powerful.

Pooling Layers

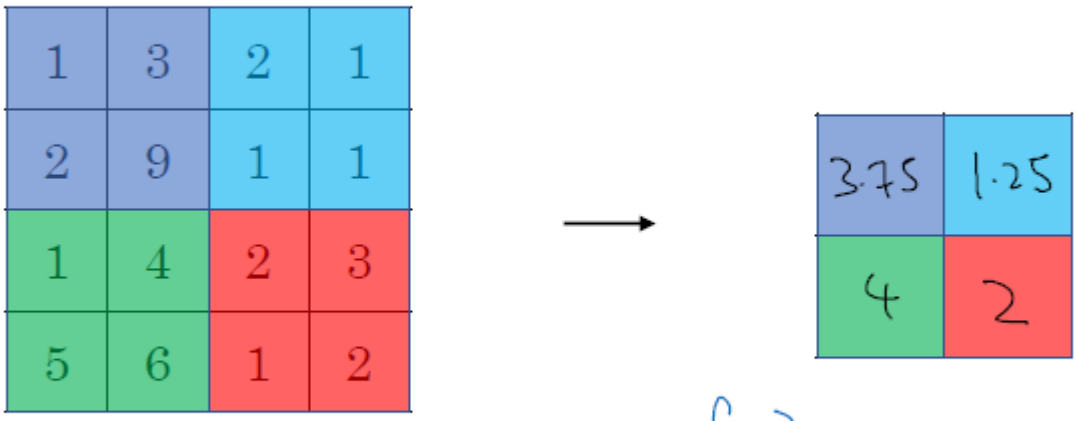
Pooling is only about getting the maximum/minimum of each block we used to convolve with filter. For the maximum case we call it max-pooling and for the other one we call it min-pooling. The intuition for using max-pooling is to maintain a specific feature within the matrix that can be an edge or a whisker or anything else. Mainly, it's been used only because it worked well. XD
Moreover, it needs no parameter to learn and can reduce the size as well. So it has many useful properties for us to use it, especially when we know experimentally it works.



"If you think of this four by four region as some set of features, the activations in some layer of the neural network, then a large number, it means that it's maybe detected a particular feature. So, the upper left-hand quadrant has this particular feature. It maybe a vertical edge or maybe a higher or whisker if you trying to detect. Clearly, that feature exists in the upper left-hand quadrant. Whereas this feature, maybe it isn't cat eye detector. Whereas this feature, it doesn't really exist in the upper right-hand quadrant. So what the max operation does is a lots of features detected anywhere, and one of these quadrants, it then remains preserved in the output of max pooling. So, what the max operates to does is really to say, 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. But I have to admit, I think the main reason people use max pooling is because it's been found in a lot of experiments to work well, and the intuition I just described, despite it being often cited, I don't know of anyone fully knows if that is the real underlying reason."

Note that pooling must be used on each channel independently.

Pooling layer: Average pooling



For pooling hyper-parameters we have filter size, stride, and mode of pooling (max, average, or min pooling). A common choice is mostly 2 for filter size and stride or maybe 3 for filter. We mostly do not use padding but we may use.

Summary of pooling

Hyperparameters:

- f : filter size (f = 2, s = 2; f = 3, s = 2)
- s : stride
- Max or average pooling

~~p: padding~~

No parameters to learn!

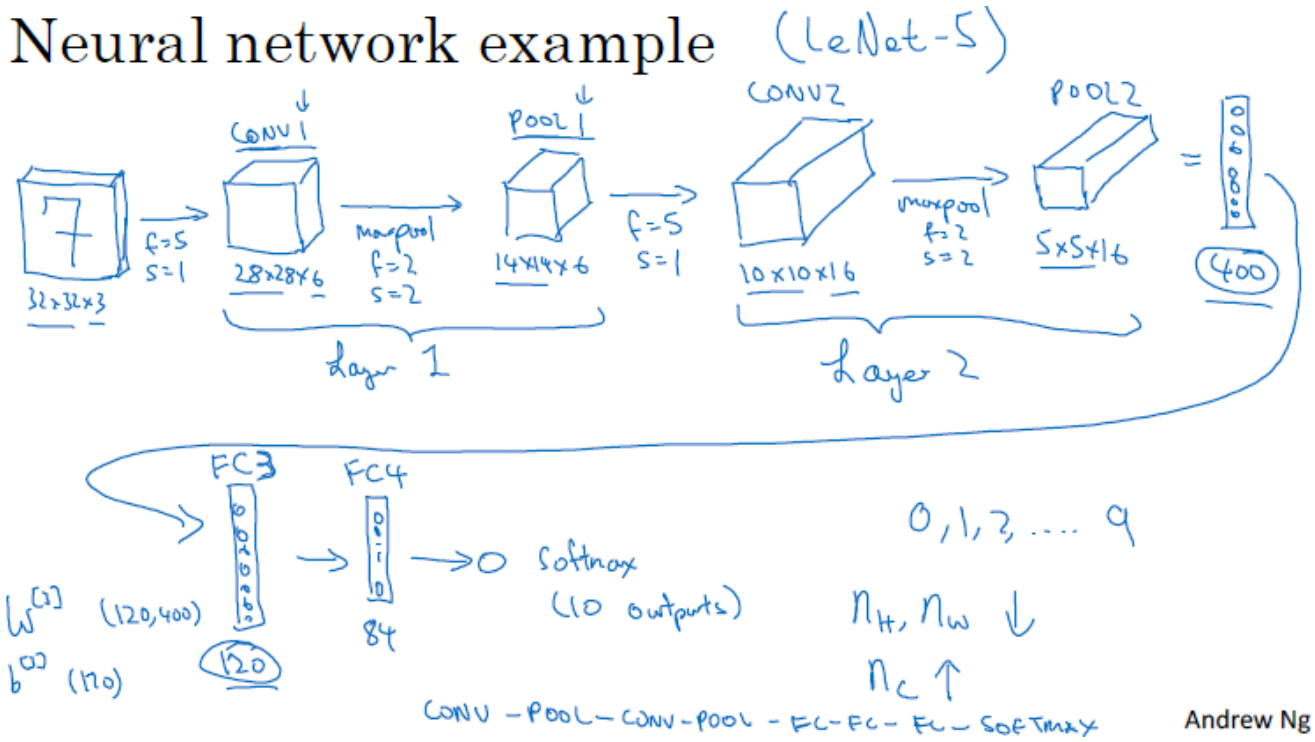
$n_H \times n_W \times n_C$

$\downarrow$

$\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor \times n_C$

CNN Example

Let's see an example of a CNN which is from "LeNet-5".
About the example: we mostly call the convolution layer and pooling layer only 1 layer. After second layer we apply a fully connected layer and pass it to another fully connected layer.



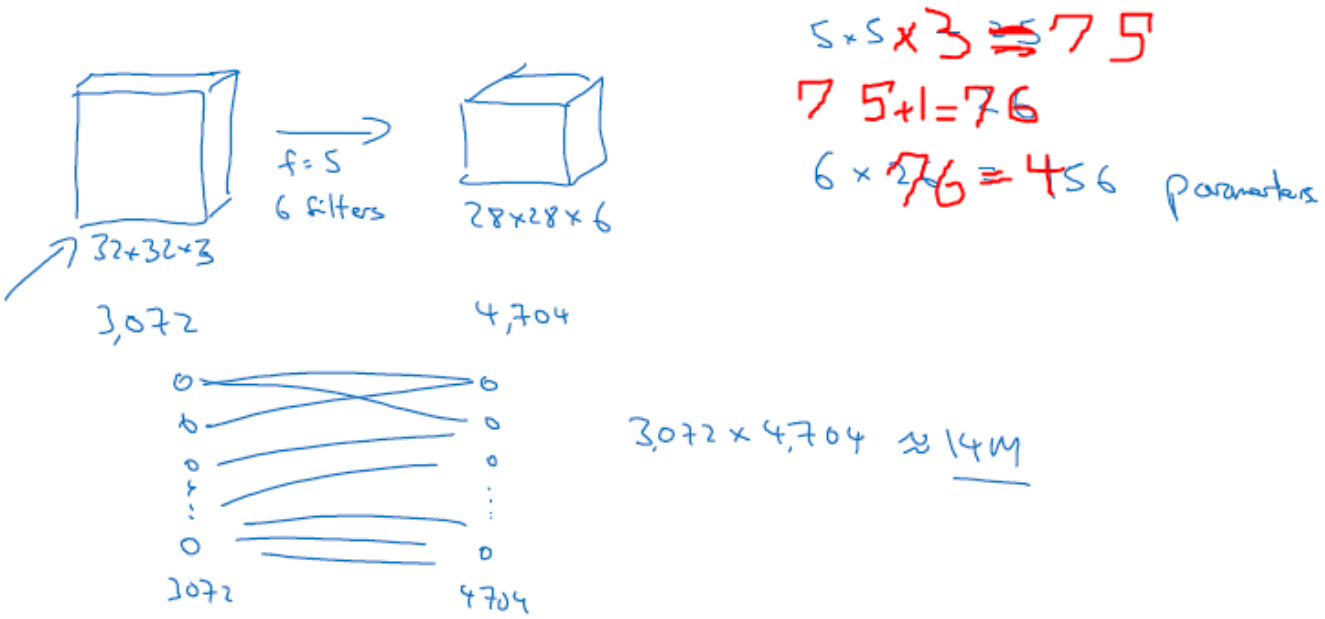
So for this example, let's see the sizes, shapes, and parameters.

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	608
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	3216
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48120
FC4	(84,1)	84	10164
Softmax	(10,1)	10	850

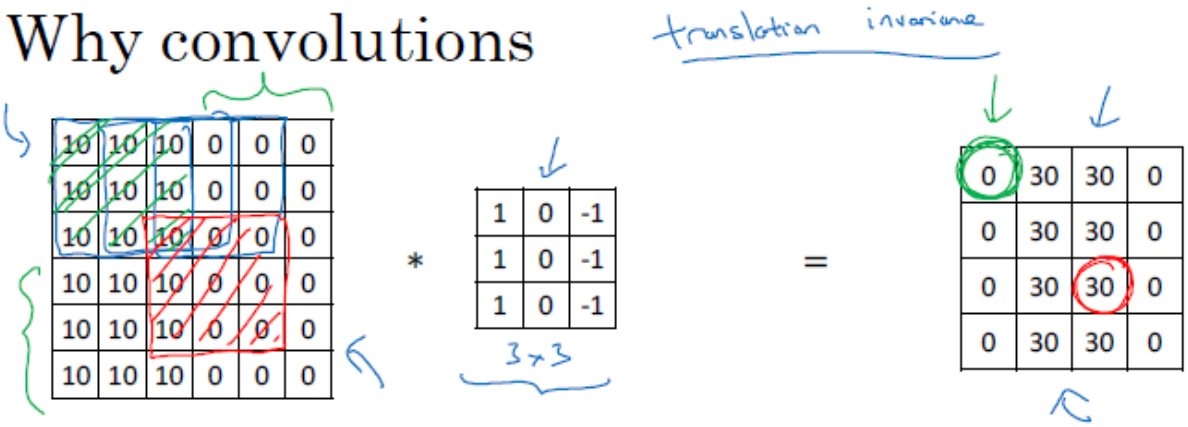
So some details about this slide: - as we can see, polling layers have parameters to learn. - Convolution layers also have very few layers. - But FC layers have many parameters to learn. - Activation size is also gradually reduces. If we drop it too quickly, it has not a great affect on performance. - The patterns will stay the same mostly.

Why Convolutions?

One of the main reasons we use convolutions is its advantages which are parameter sharing and sparsity of connections.
In the example below, if we used fully connected networks as we used in neural networks, we end up with around 14 million parameters to learn. It is too many. And as the size of the image grows it would be way more. Now we have 156 parameters.



As for the **parameter sharing** property, we apply the same filter, which its values are learned by the model, over many places of the input (image). So we are using same features over and over again.
For the **sparsity of connections**, if you take one of the corners of the output, we can see that, only the f^2 cells of input affect that while the rest has no effect on it.

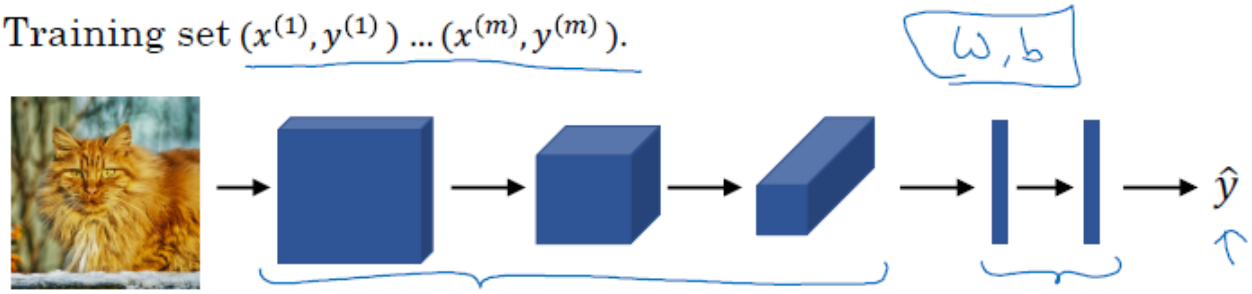


Parameter sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

→ **Sparsity of connections:** In each layer, each output value depends only on a small number of inputs.

Putting it together

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



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

Use gradient descent to optimize parameters to reduce J

Assignment tips:
The main benefits of padding are the following:

- It allows you to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.

- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.

WEEK 2:

Why look at case studies?

It turns out that a CNN working well on a task, can be applied to other tasks. So we can take some of them and apply on our works.

Classic networks are:

1. LeNet-5
2. AlexNet
3. VGG

The other more important ones are:

1. ResNet
2. Inception

Classic Networks

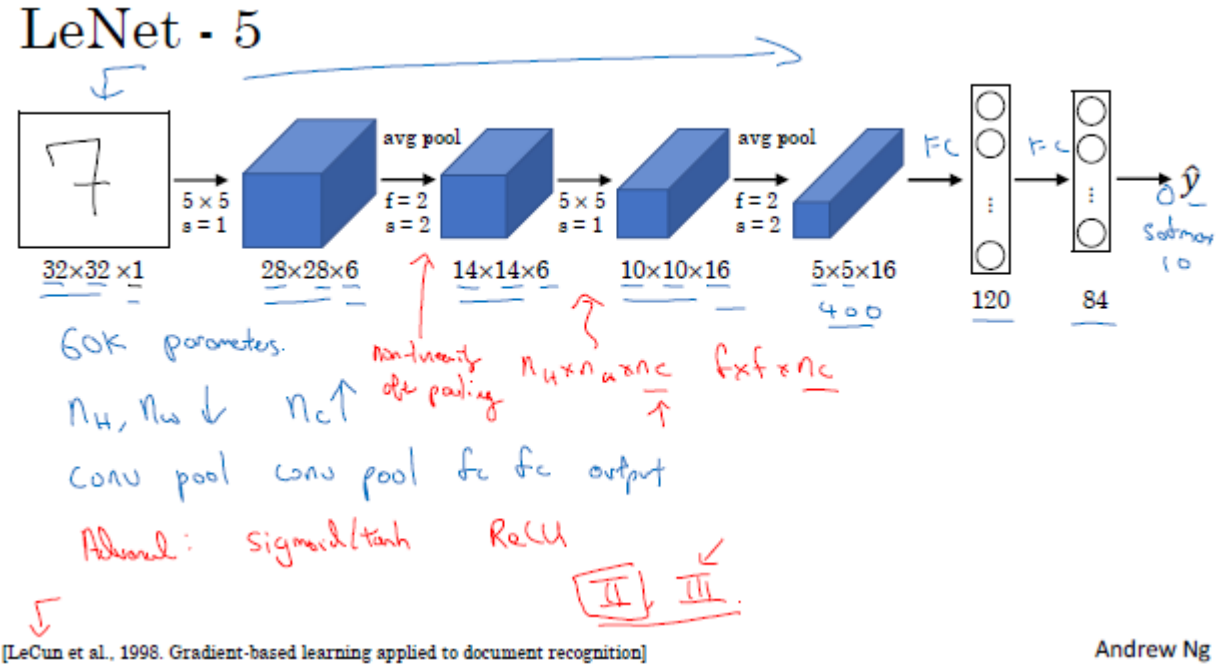
LeNet-5

Back then, 1998, researchers mostly never used padding. And for its last layer, this paper used different classifier which is useless today. So we can use Softmax instead.
This network is simple, using 60k parameters; nowadays we use networks with number of parameters between 10M to 100M.
As we go deeper, the height and width decreases and the number of channels increases.
The type of arrangements of layers in LeNet-5 is quite common:

Input → Conv → pool → Conv → pool → FC → FC → output

Additional notes:

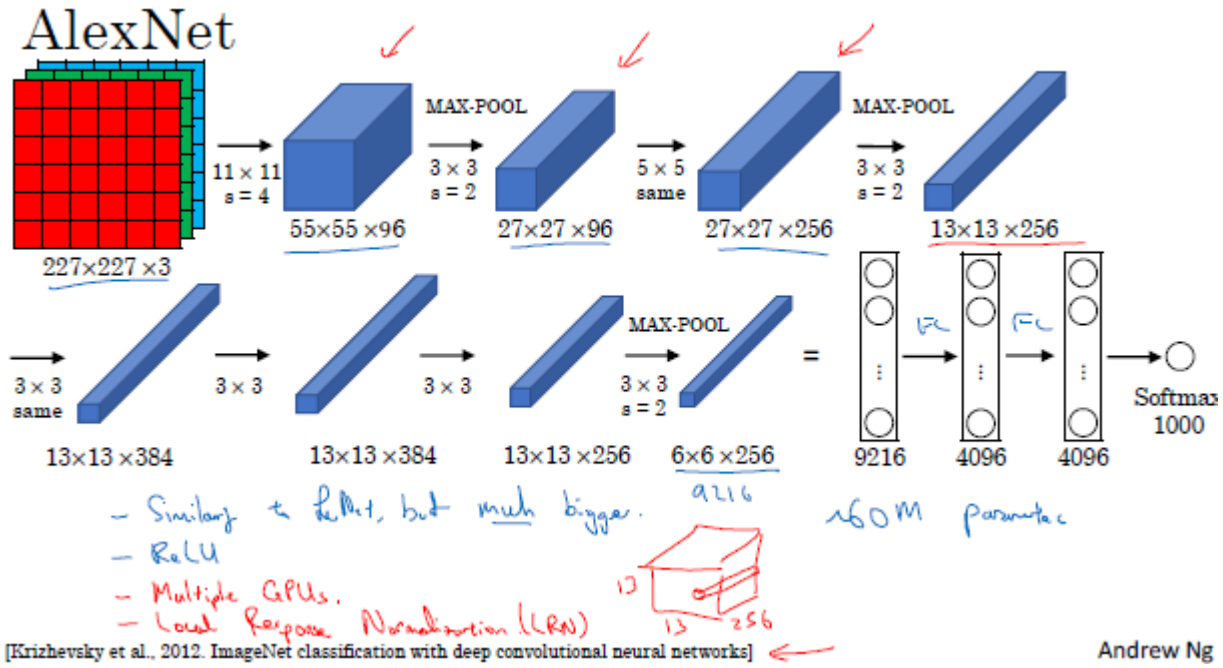
1. People tended to use Sigmoid / tanh for nonlinearity, not ReLu.
2. Due to expensive computation back then, the number of channels were not been used the same way as we use it today and it had some tricks.
3. It had Sigmoid nonlinearity after pooling which is not common anymore.



AlexNet

So many similarities to LeNet-5 however much more parameters, around 60M.
It uses ReLu
It had a complete way of training on the 2 GPUs due to GPU working complexities back then.
It also had a set of layers calling Local Response Normalization (LRN) which is not common anymore.

LRN normalizes all the values across channels for each height and width.
LRN was used to normalize neurons with a very high activation; however, found out it is not much useful.

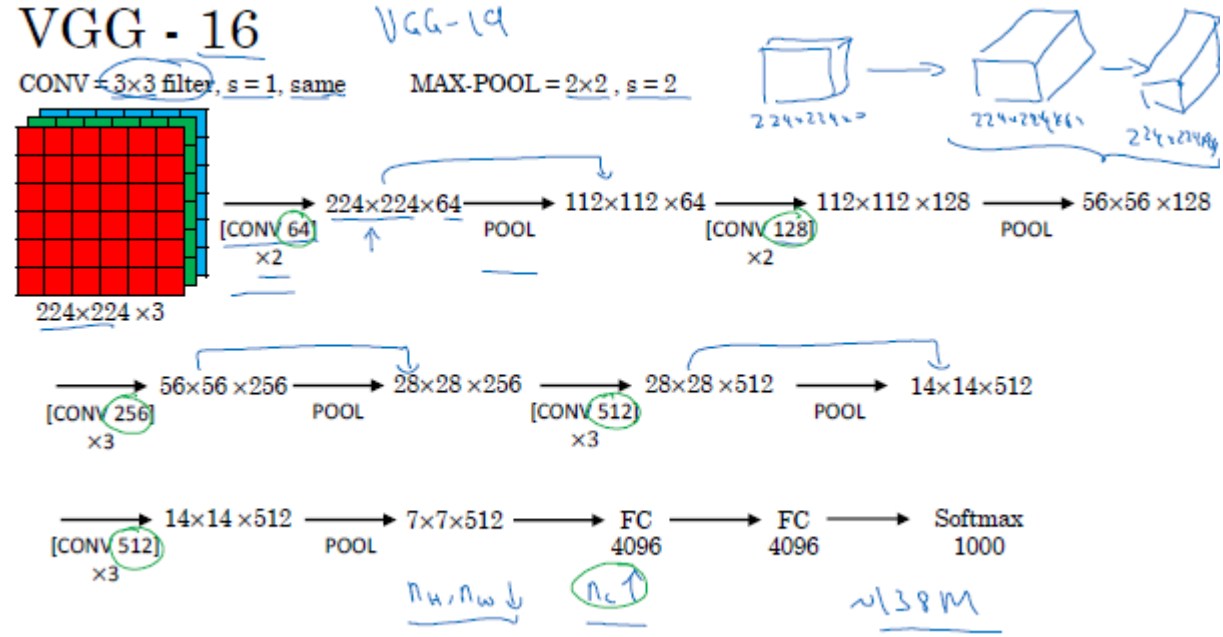


[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]

Andrew Ng

VGG-16

It has much simpler network. However, so deep it is.
The times 2 in the slide means this layer was implemented two times.
The 16 in the name points out to the 16 layers. It has 138M parameters.
It used pooling to reduce the height and width. It also doubles the number of channels on every step except the last one as 512 channels are big enough.
The downside is that, it has too many parameters to train.
We also have VGG-19 which is a bigger network but VGG-16 almost work as good as VGG-19.



[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

Andrew Ng

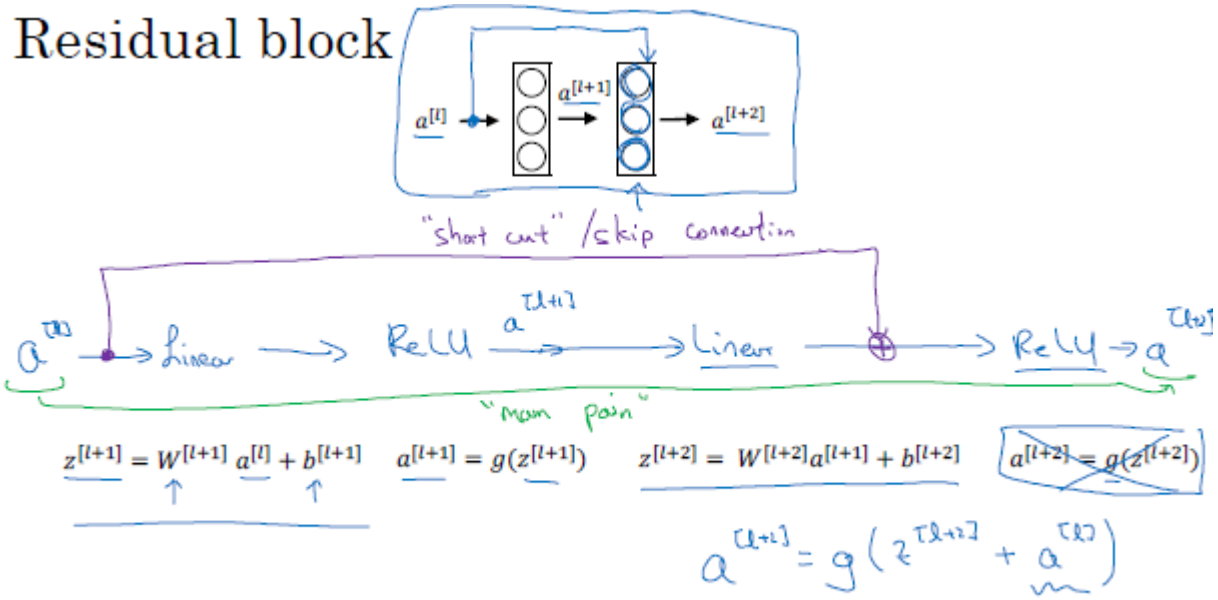
ResNets

ResNets is about skipping connections which allows you to take the activation from one layer and suddenly feed into another layer even much deeper in the neural network. It helps to train very very deep networks, something like 100 layers.

The normal steps for this kind of network is:

$$a^{[l]} \longrightarrow Linear\ step \longrightarrow ReLu \longrightarrow a^{[l+1]} \longrightarrow Linear\ step \longrightarrow ReLu \longrightarrow a^{[l+2]}$$

The linear step for example for the first one is $z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$.
It is the main path for $a^{[l]}$ to reach to $a^{[l+2]}$. However, what ResNets does, it carries the $a^{[l]}$ from the other "short cut" / "skip connection" to the linear part of its next layer. So, the $a^{[l+2]} \neq g(z^{[l+2]})$ anymore, it is $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$.

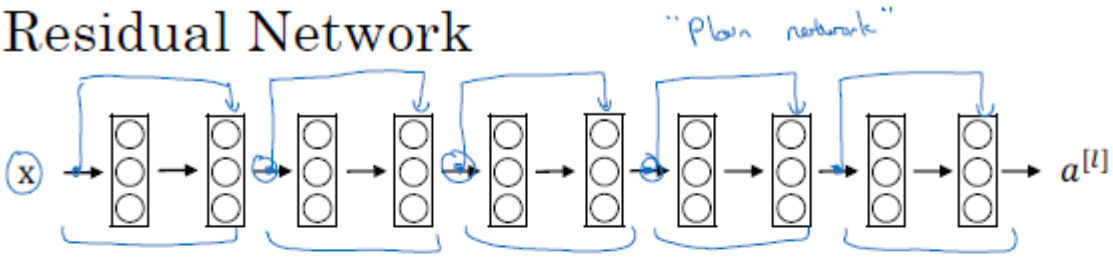


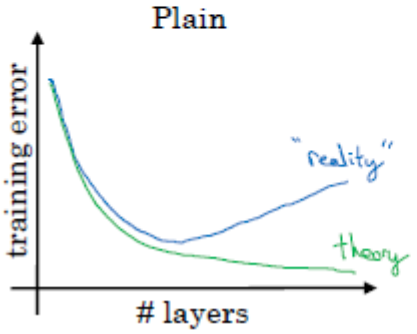
[He et al., 2015. Deep residual networks for image recognition]

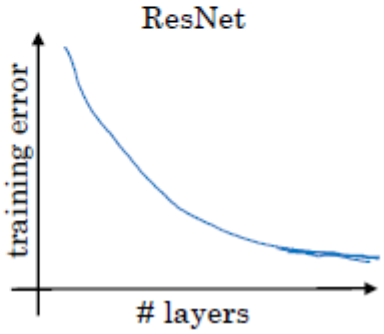
Andrew Ng

In the example below, the plain network converts into 5 stacked residual networks. ResNets helps a lot in training a very deep network as it can be seen in the diagram below. It also helps with vanishing and exploding gradient.

Residual Network



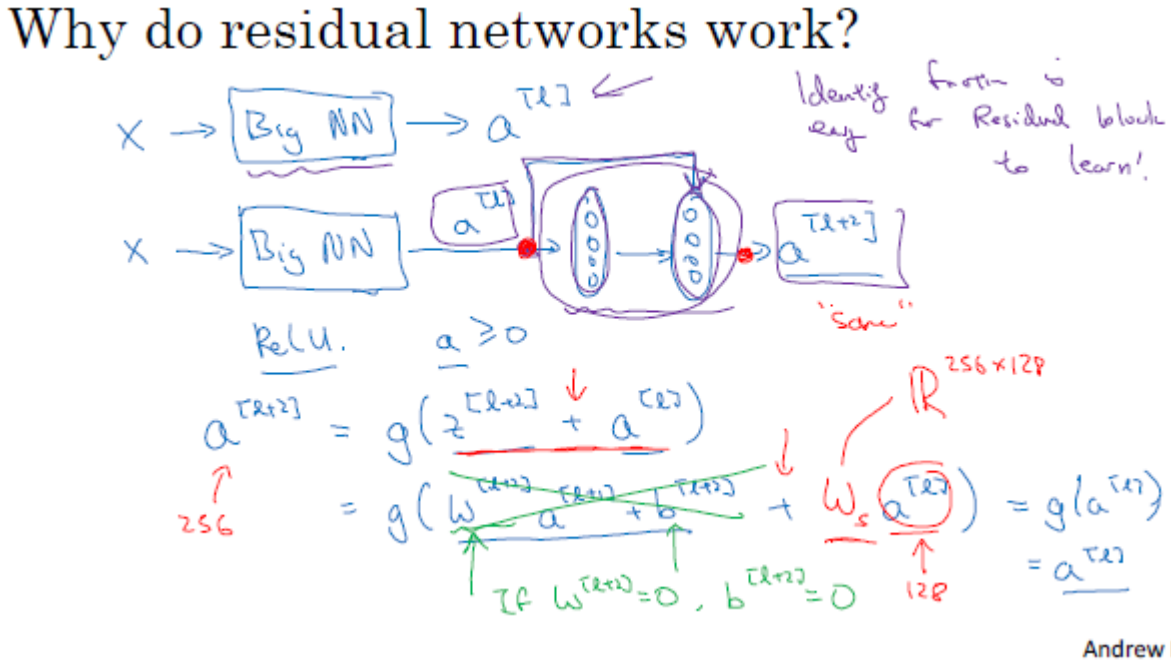




Why ResNets Work

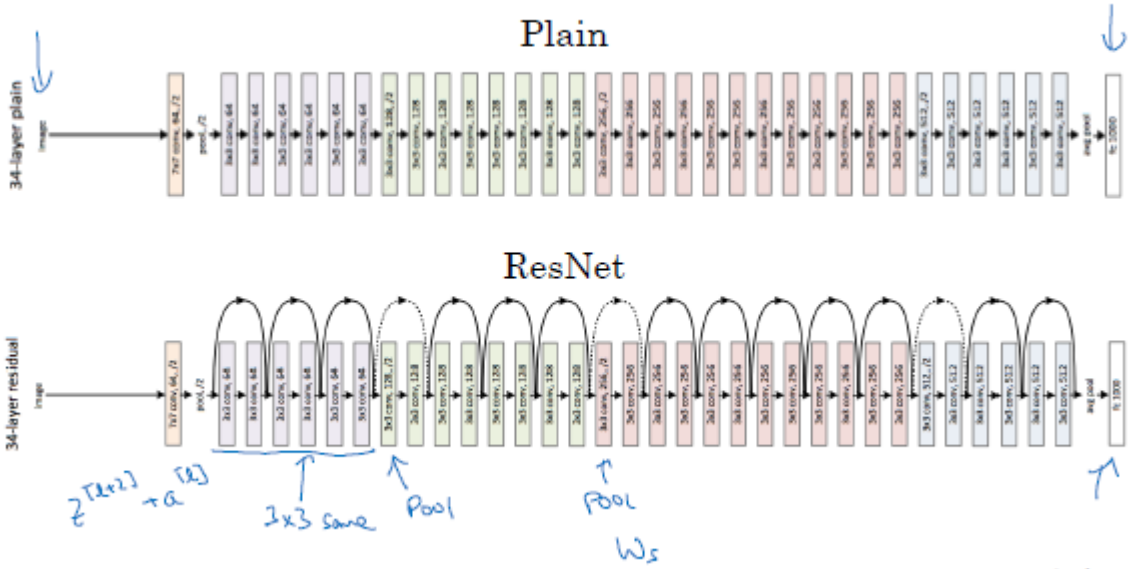
A very deep network, however, theoretically should perform better, it is not, even in training phase. ResNets made the theory work. But why?

In the worst case scenario, the added layer has no information to carry out to the next activation function. Meaning, in the formula $a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$, $z^{[l+2]}$ will be 0 if the weight is so small, very close to 0. So, $a^{[l+2]} = g\left(a^{[l]}\right)$. It shows that the identity function is easy for the ResNets block to learn. Therefore, adding a ResNets blocks, does not hurt the performance as it can be assumed that the extra layer never existed if necessary and if helpful it is there.
On the other hand, but of course our goal is to not just not hurt performance, is to help performance and so we can imagine that if all of these heading units if they actually learned something useful then maybe you can do even better than learning the identity function. And what goes wrong in very deep plain networks without this residual of the skip connections is that when you make the network 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.
"And I think the main reason the residual network works is that it's so easy for these extra layers to learn the identity function that you're kind of guaranteed that it doesn't hurt performance and then a lot the time you maybe get lucky and then even helps performance."
There is another assumption which is the $z^{[l+2]}$ and $a^{[l]}$ have equal dimensions. It is because it uses the "same" convolution. If it was not the case, we can use a W_s which can only adding padding to the $a^{[l]}$ to bring it back to the same dimension.



As an example for ResNets, we can see the example below. Most of the block filters are 3 by 3 same convolutions to preserve the connection. Sometimes it also uses pooling and uses the previous technique to brings it back to the same dimension.

ResNet

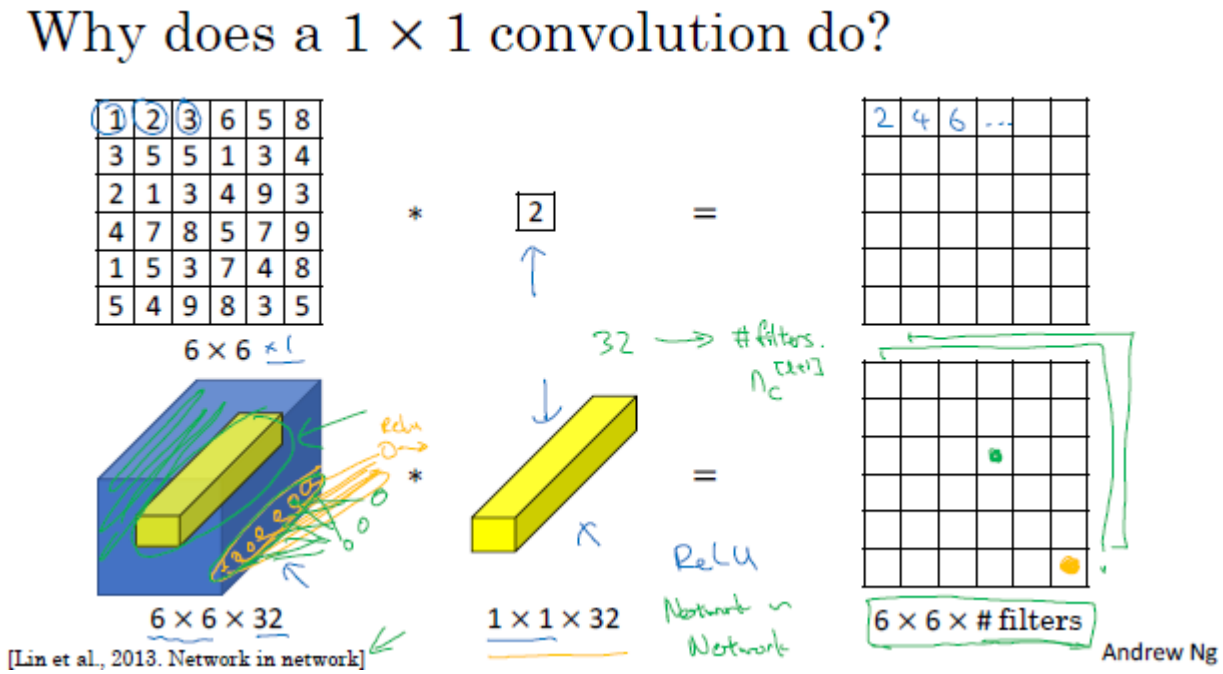


[He et al., 2015. Deep residual networks for image recognition]

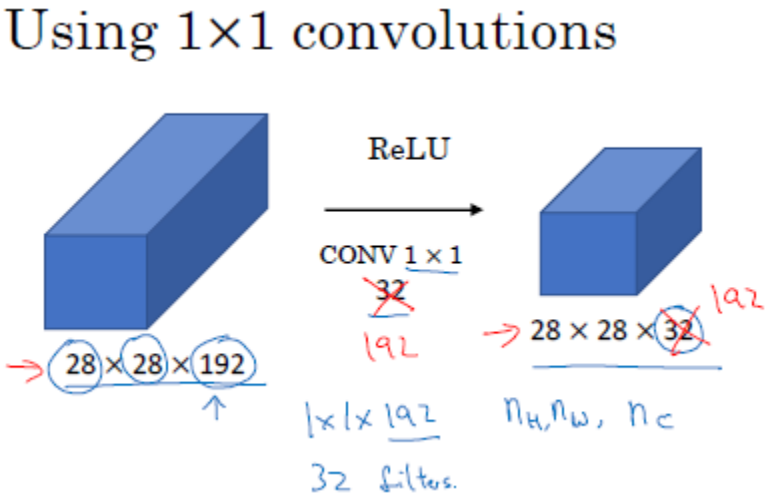
Andrew Ng

Networks in Networks and 1x1 Convolutions

So what exactly the 1by1 convolution block does is that it brings all the channels for each height and width into a single number. We can use multiple of them to stack the output for it. It is also been called "network in network".

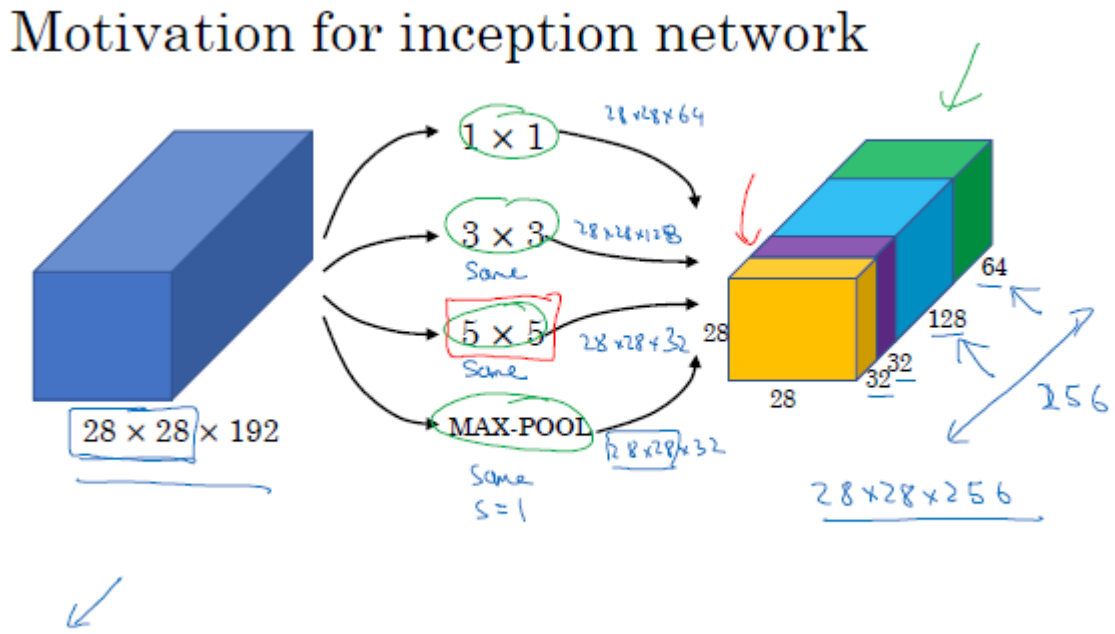


For shrining the number of channels we can use it.

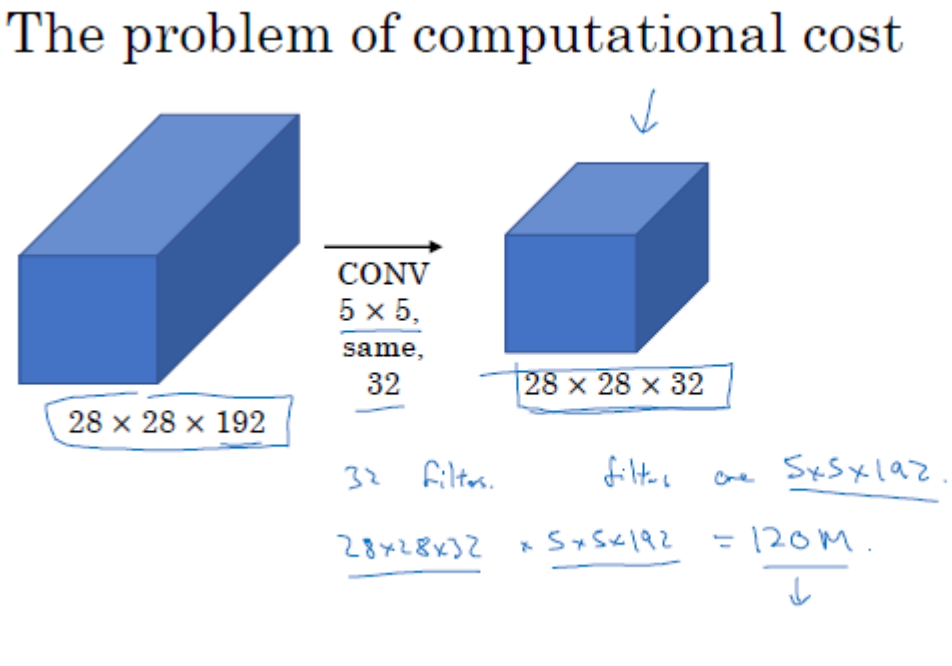


Inception Network Motivation

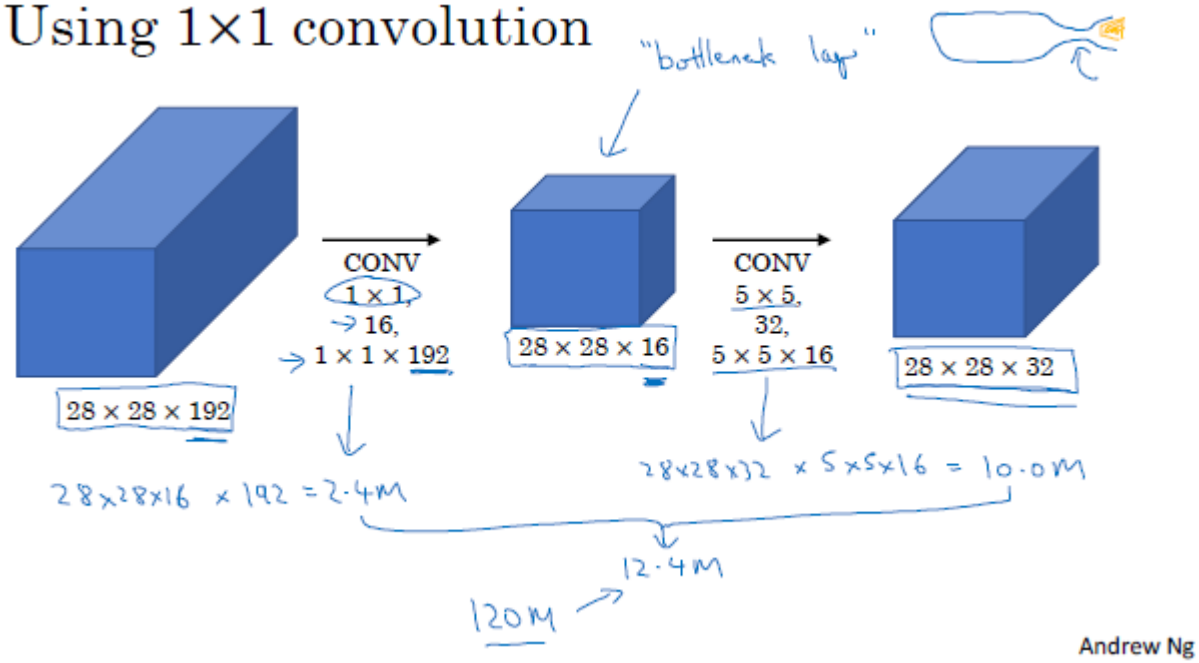
What inception network says is that instead of using only one kind of network block, let's use some of them and stack them together and let the network do all the computations and combinations it needs to get the output.
However, its downside is computational cost.



Let's see the computational cost of 5 by 5 block of the slide above. For this block, we can see the number of computations shown below. It is around 120M multiplication. So we can reduce it by factor of 10 using 1 by 1 network.



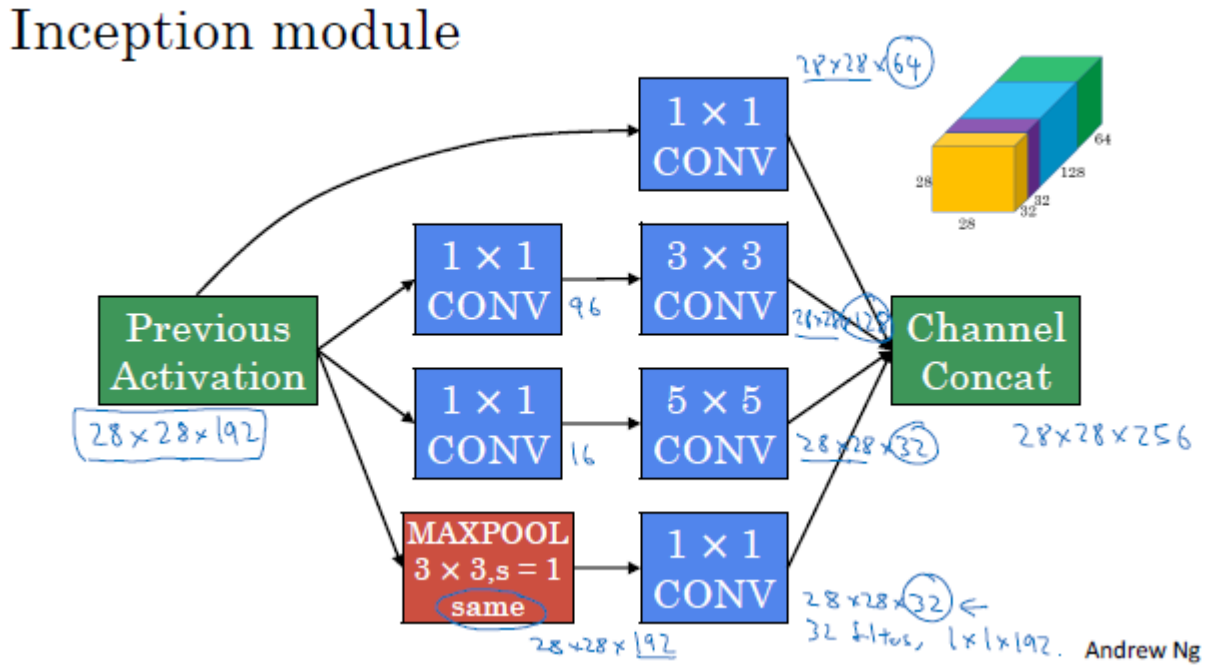
So for reducing the computational cost, we use a network in network block as “bottleneck layer” to reduce the overal size of the input and then we can apply 5 by 5 convolution and get the same dimensions.
Now its computation for the first layer will be around 2.4M multiplications and for the second one around 10M. In overall it is 12.4M which is so much less than 120M multiplications.



So if you do not want to be baffled about the type and size of a block, we can use all of them and deal with its computational cost with network in network block. In many cases the network in network block does not hurt the performance.

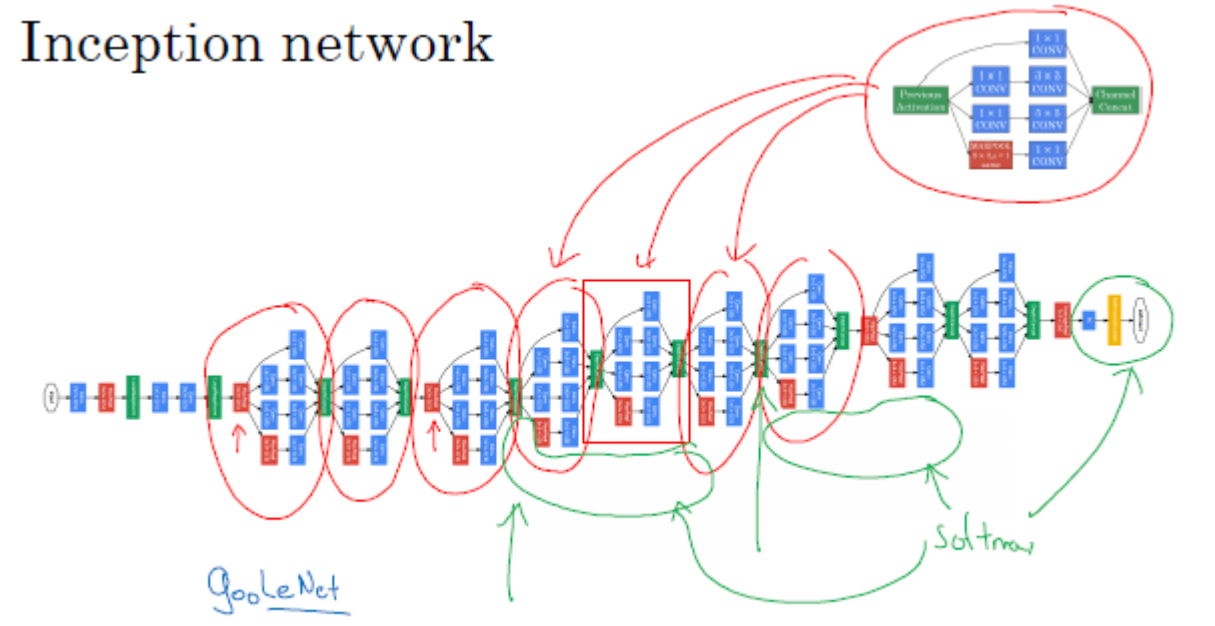
Inception Network

One example of the inception module is like this:



So in overall, the inception block is shown here which is the repetition of the inception module we saw above.

There are also additional side branches that can be seen in the slide below, inception network. The side branches tried to do some predictions in middle of the work. That's why they used Softmax there. These side branches appeared to have a regularizing effect on the inception network and prevents it from overfitting.

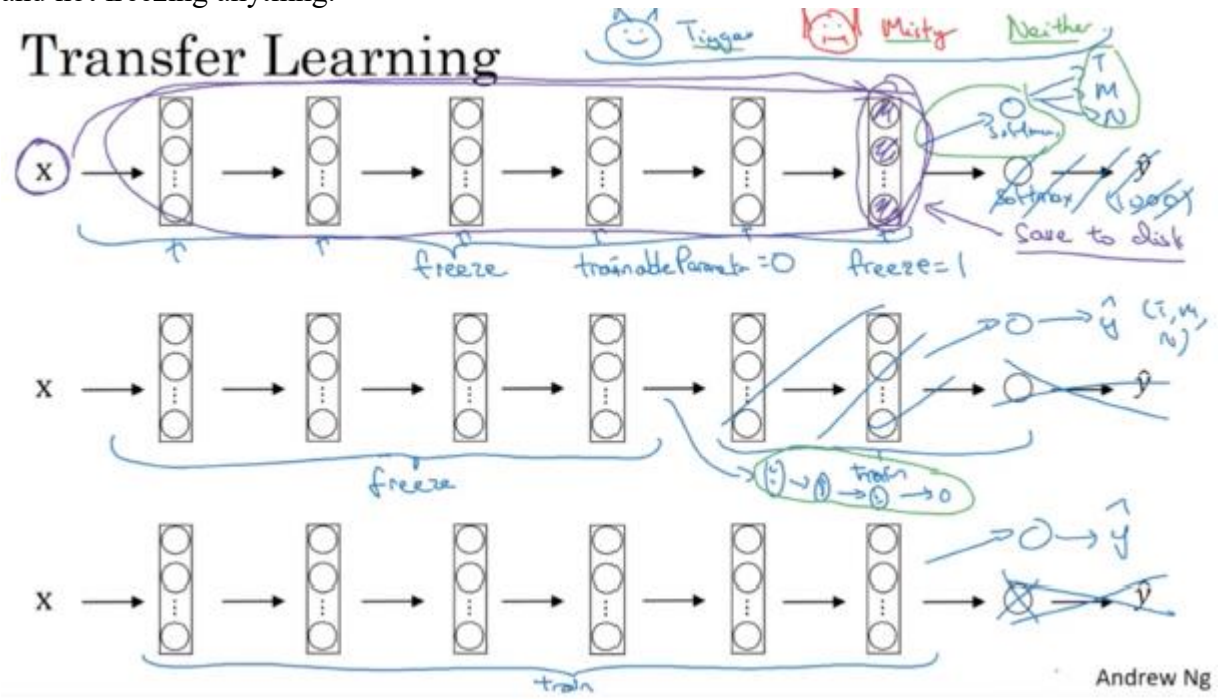


Using Open-Source Implementation

The fact that it is hard to replicate many publications and research works, the DL community routinely open source their work on GitHub.

Transfer Learning

In many cases, it is so hard and computational expensive to search for hyper-parameters in the very deep networks. We can use the weights of other works as well as their hyper-parameter choices to work on our new problems. Using those weights in our work is what we call “transfer learning”.
For example for detecting some kinds of cats which we have not much data about, we can download others people architecture and weights and freeze the part they used. Then we have to add more layers for our works and try to implement our work. So, as many layers are fixed, we can assume they are the input and just make a shallow network it’s afterward.
If we have much more data, we can freeze less layers then. It’s up to us to re-initialize the non-frozen layers or continue the operation with their current weights.
If we have enough data for the training the whole architecture, we can simply just use their architecture and not freezing anything.



Data Augmentation

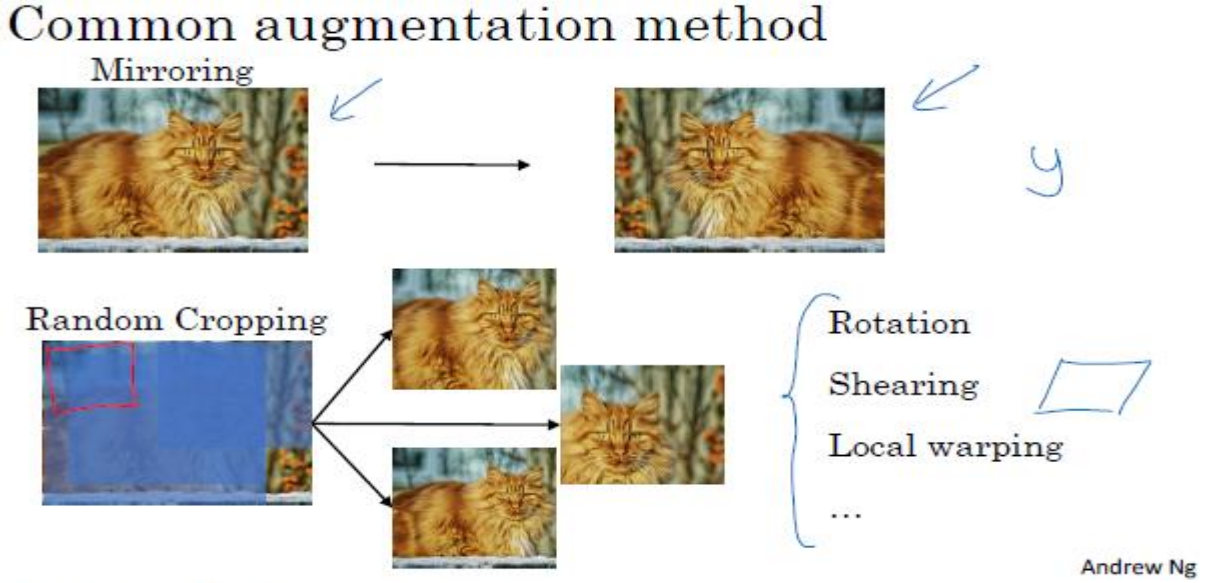
Computer vision tasks usually need much data. We feel more data we have more accurate we can be in computer vision tasks. So data augmentation can help.

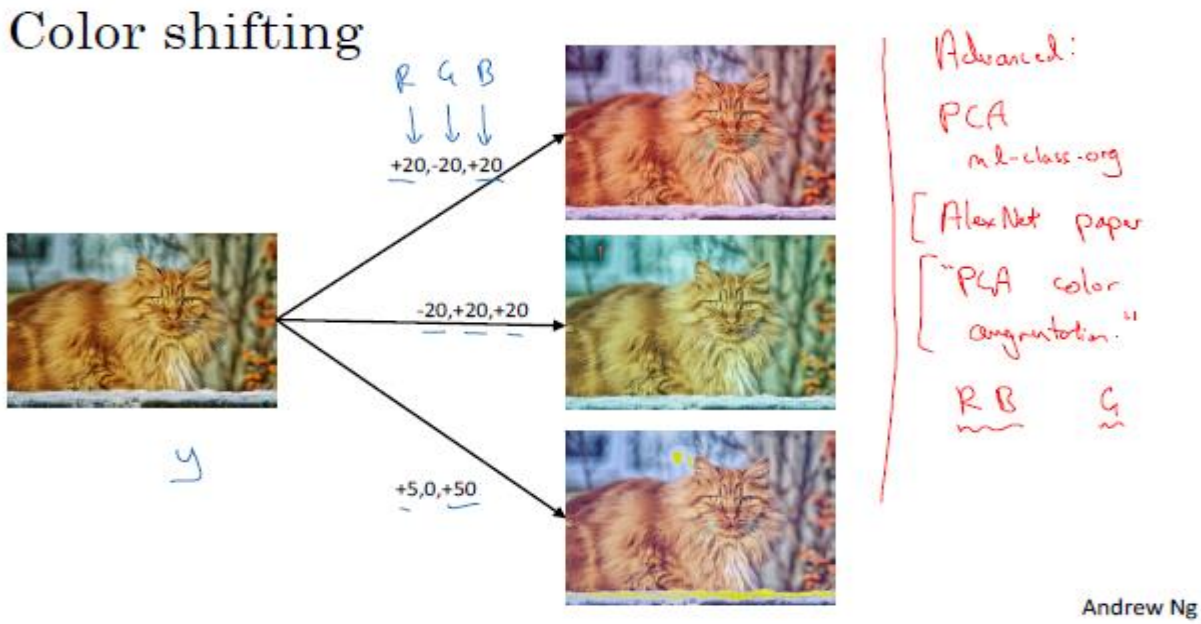
Common augmentation methods:

1. mirroring,
2. random cropping
3. rotation, ...

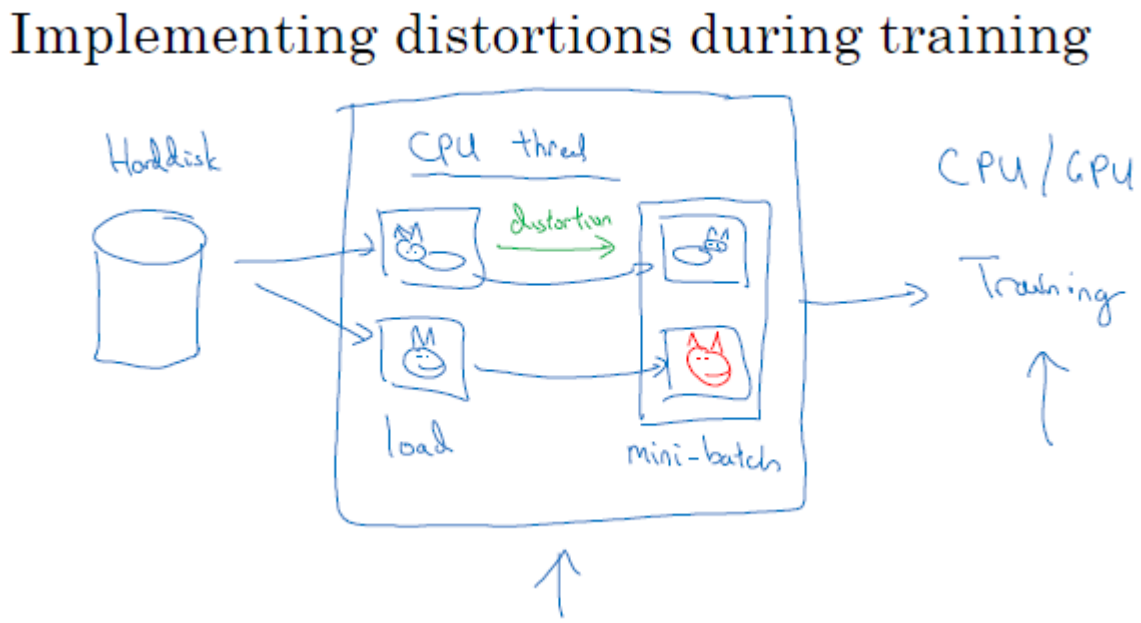
Color shifting for augmentation task: we can change the color values of RGB and feed it as a new picture. The intuition for this kind of work can be for example if there is a sunlight or darker environment, the output should still be the same and network can learn it and become more robust to changes of colors.

One way to implement color shifting is to use PCA (principles component analysis). It is mentioned in the AlexNet paper.





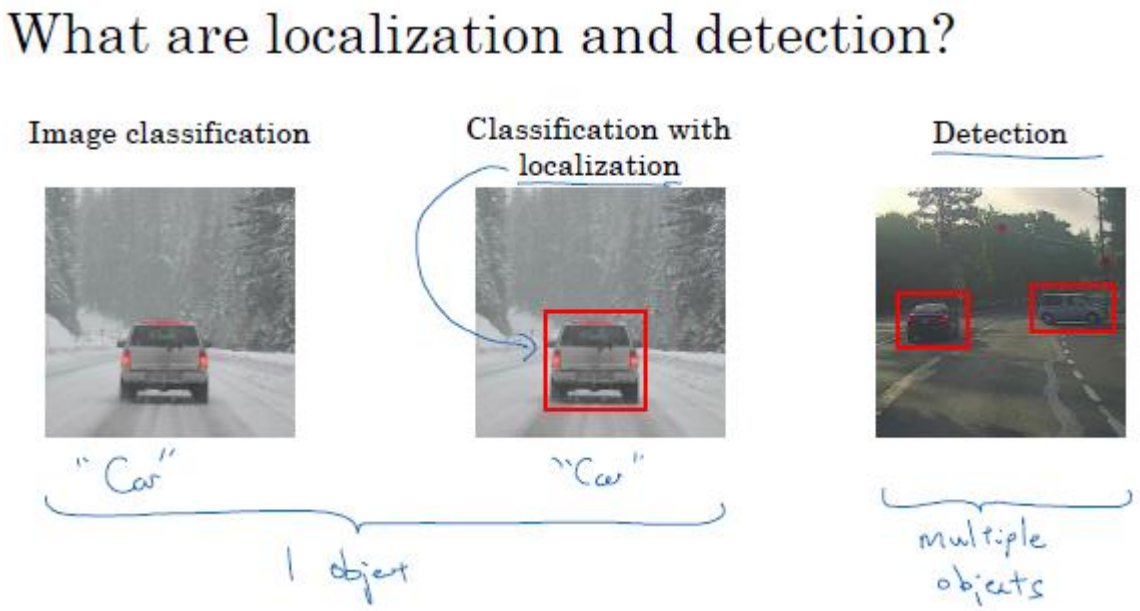
If we have a hard disk, we can use CPU threads to read images from the hard disk and distort the image or shift its colors and then pass them for training phases.
Note that the shifting and augmentations have hyper-parameters as well. So we can use other people works as well in this section.



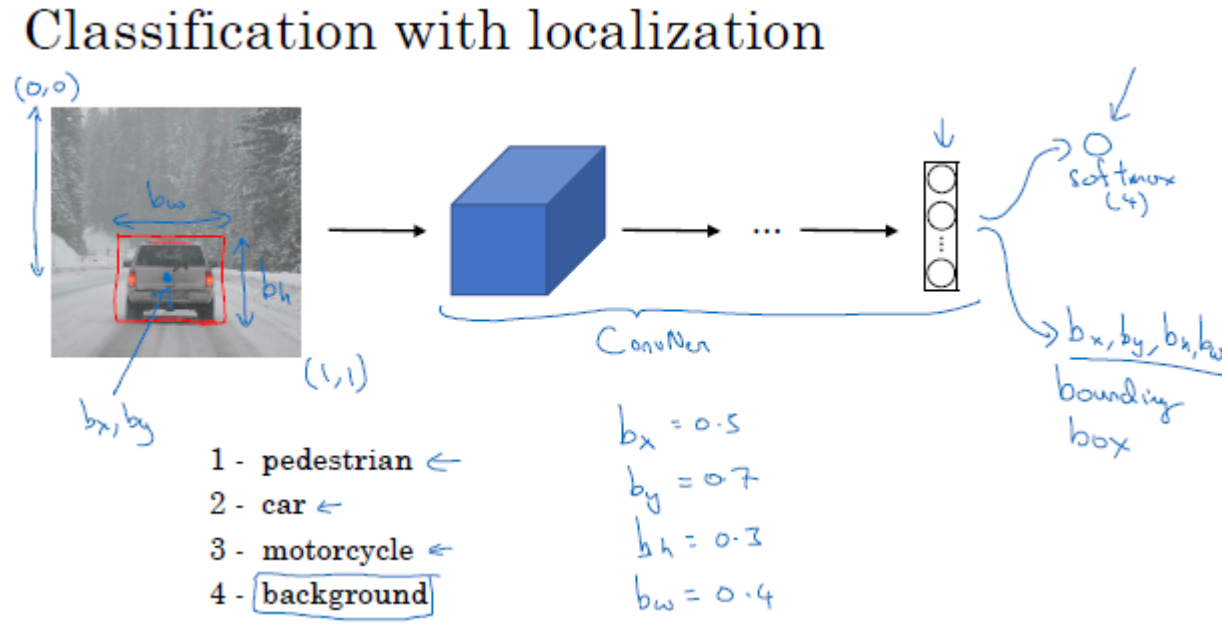
Week 3

Object Localization

Localization is about where in the picture the object is detected.
The classification of localization problems usually have one object. Usually one big object in the middle of the image that you're trying to recognize or recognize and localize. In contrast, in the detection problem there can be multiple objects. And in fact, maybe even multiple objects of different categories within a single image. So the ideas we learn about image classification will be useful for classification with localization. And that the ideas we learn for localization will then turn out to be useful for detection.

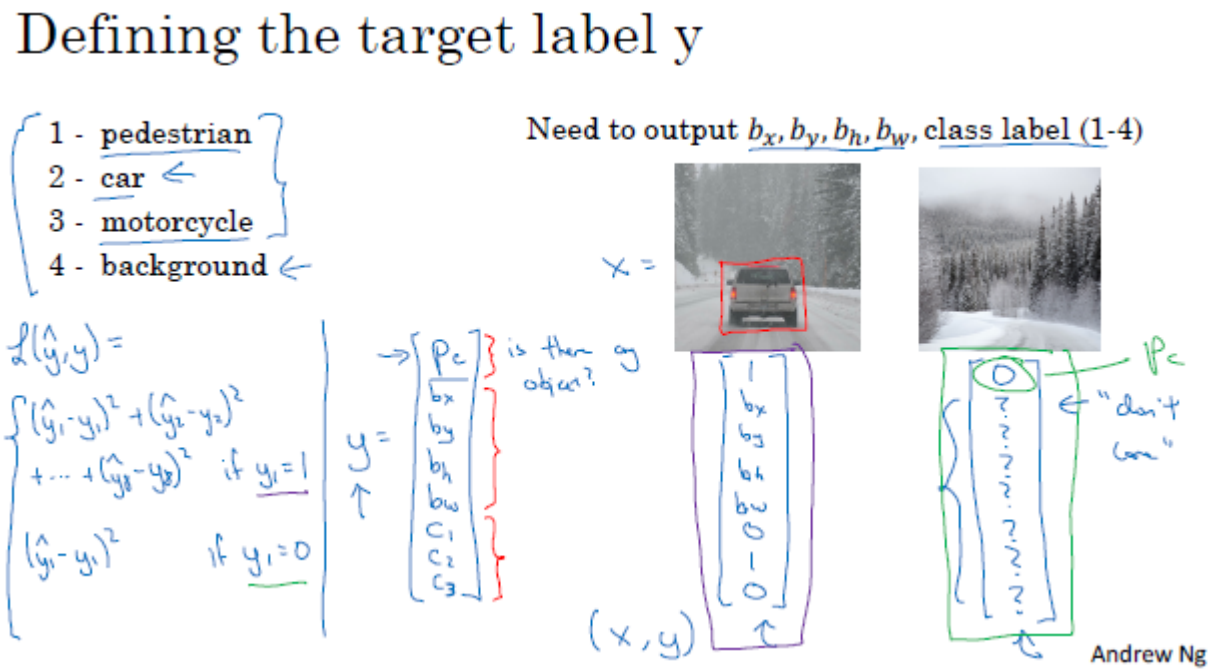


So for the object classification problem, we use a ConvNet and output classes with Softmax at the last layer. We use a notation that the (0, 0) is upper left and lower right would be (1, 1).



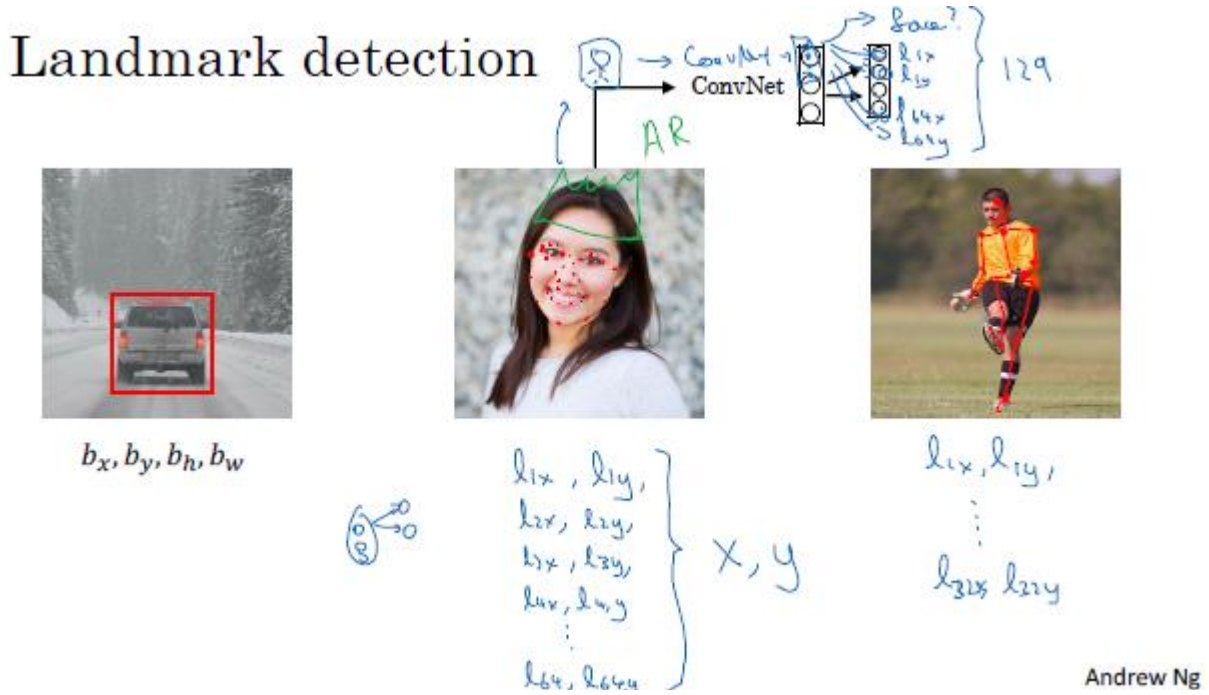
In the below picture, the P_c is the probability of having an object which is either 0 or 1; when it is 0 it means it only has background; otherwise, one of the three objects are in the picture. In the latter case

we also need to define b_x, b_y, b_h, and b_w for each object as well as the object we want to show as 1 and rest of the objects as 0. In the former case, when there is no object in the image, we use don't care sign for everything other classes.
As for the loss function, for squared error, if we see an object, we take all the differences, but if we find no object, simply we calculate the squared error of the class of object founded and the its true value. Actually we should not use squared error loss for all of the classes.



Landmark Detection

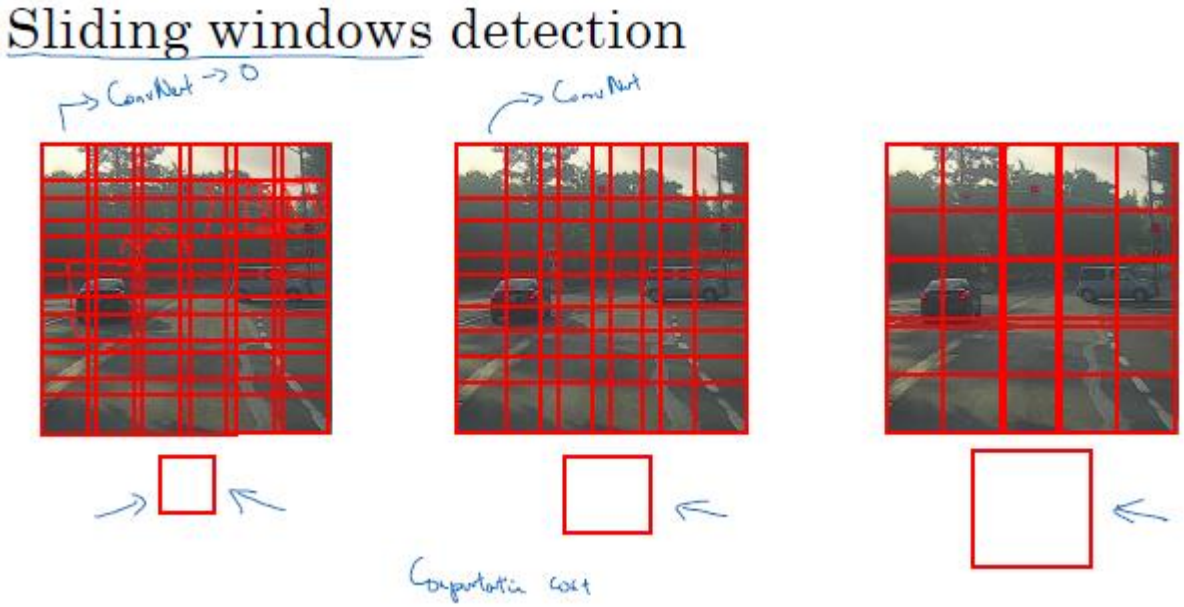
In the figure below, for the middle one we want to recognize face elements, and for the right one we want to recognize the pose of the person.



Labels must be consistent across all images. Edge of the left eye for example must always be l_1.

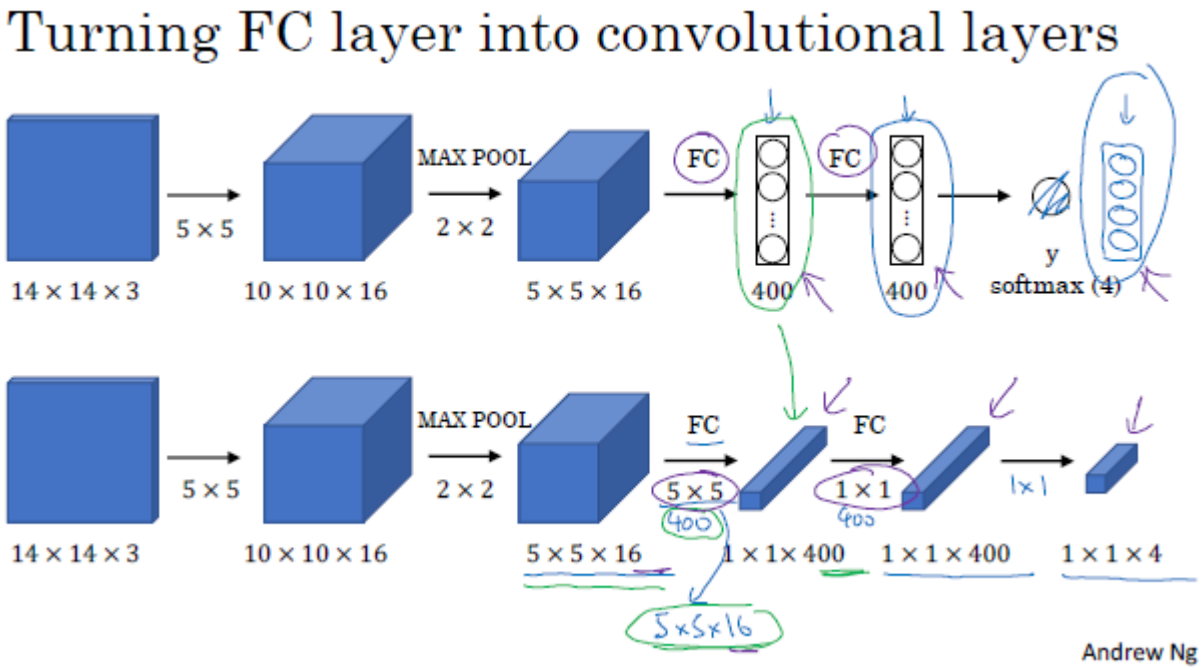
Object Detection

For the object detection, the first approach is using sliding window. We start the sliding window from the upper left of them image, and input it into a ConvNet. Then shift the window to the right and down. We can use stride here as well.
After that, we use a larger window. The hope is that we can find a window that fits the object in the image.
Disadvantage of this approach is that its computational cost is high when we use small strides. If we use higher values for strides, the cost reduces but the performance hurts. Before using a ConvNet and deep networks, it could have been OK, but with ConvNet it is not.

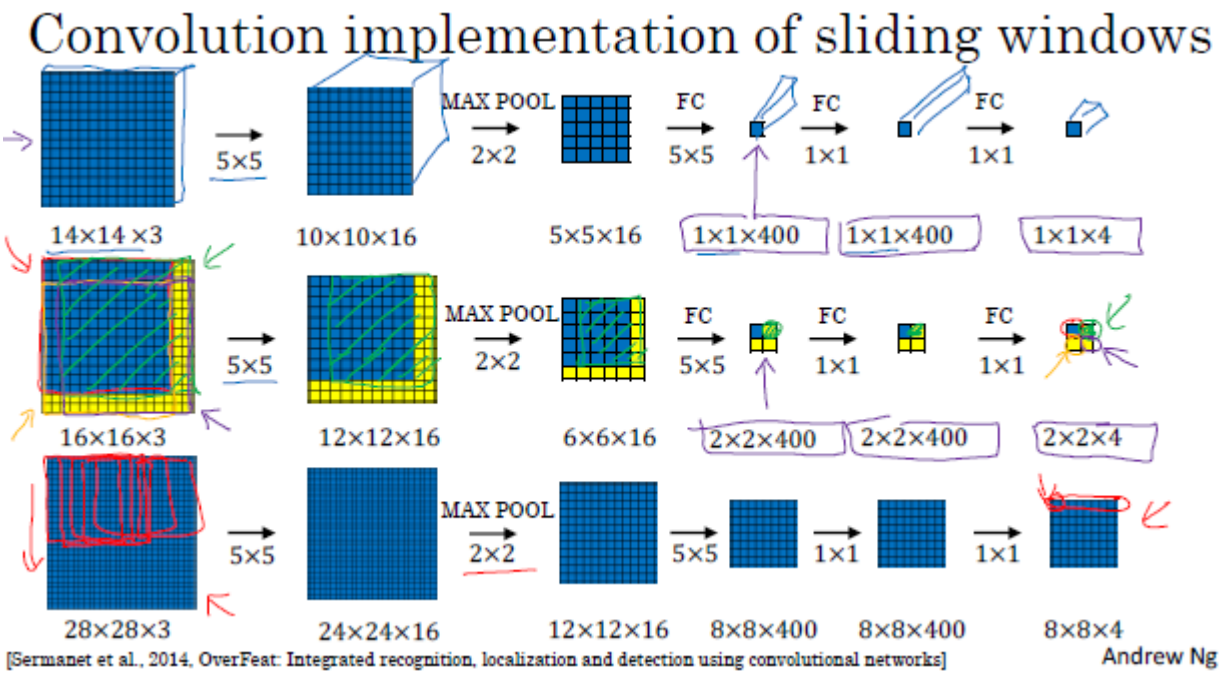


Convolutional Implementation of Sliding Windows

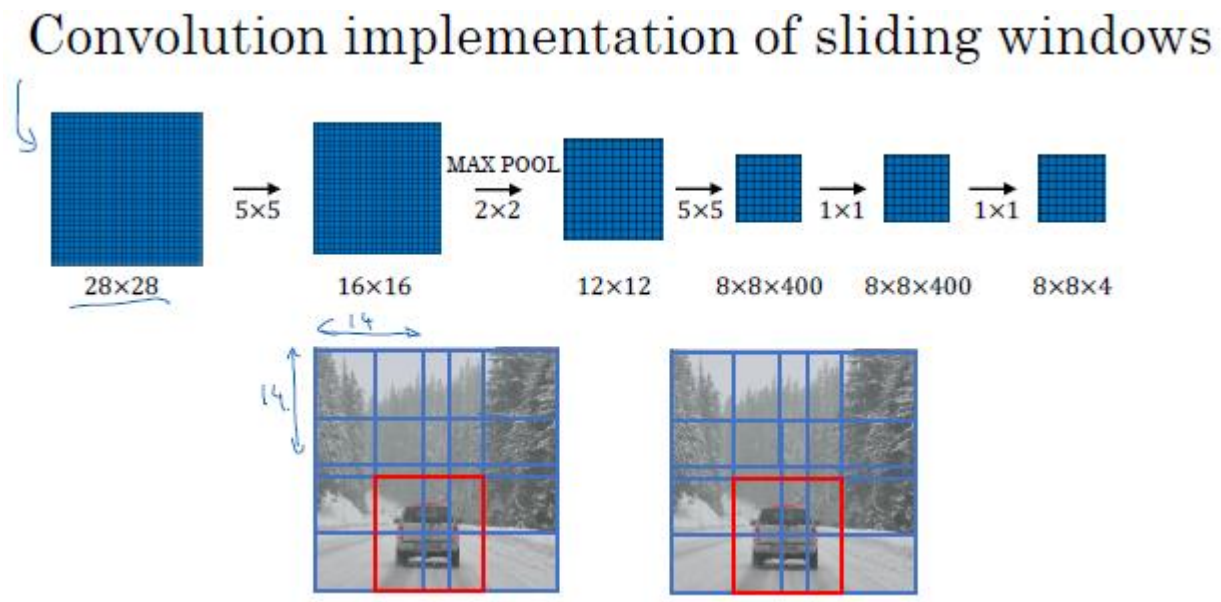
For using sliding window in ConvNets, we first need to convert the FC blocks into a blocks of ConvNet. For doing that, we use a filter size as big as the previous one but with as many as the pixels filter numbers. We can see how the conversion looks like:



The convolution implementation of the sliding windows shares a lot of computation so it works way better than the normal sliding window.
Now the trick is that, we add padding, for example in the middle ConvNet example, we go over 14 by 14 by 3 with stride of 2, 4 times. Then the end result will be 2 by 2 by 4 instead of 1 by 1 by 4. Now in the 2 by 2 image are the same as going over the initial image 4 times. It shares a lot of computation now.
The last example is as well shows the same trick.

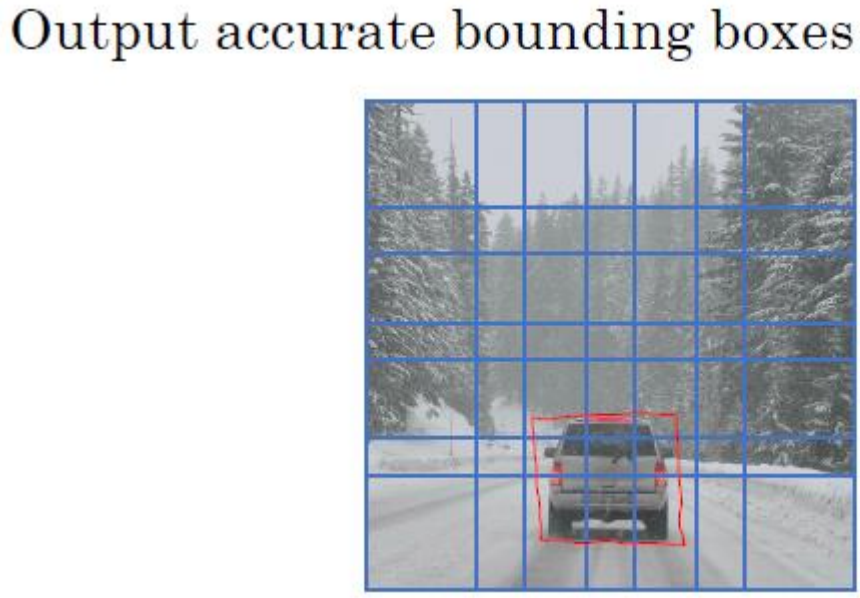


Finally, we can see through one propagation we check all of the sliding boxes and recognize all of the objects. However, it still has a weakness which is the position of the bounding boxes is not going to be too accurate for many purposes.

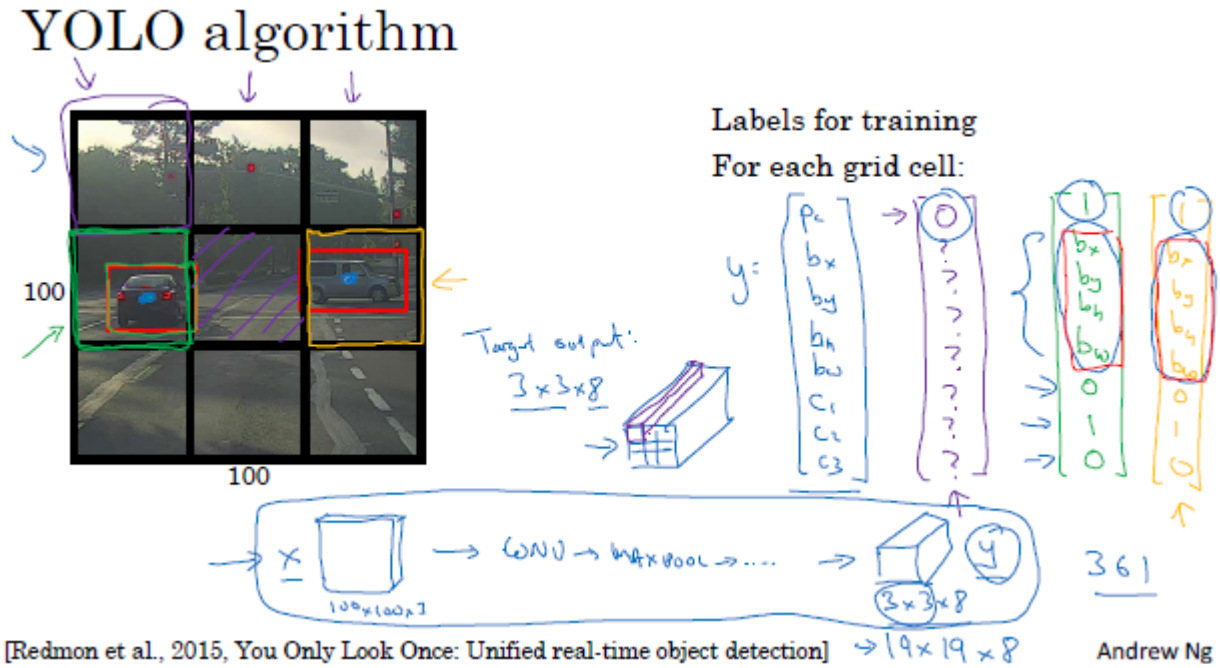


Bounding Box Predictions

So, our two main problems with sliding window version of the ConvNet is that we may not find exact location of the object in the image as well as its actual size in that one square might not be the best way of recognizing an object.

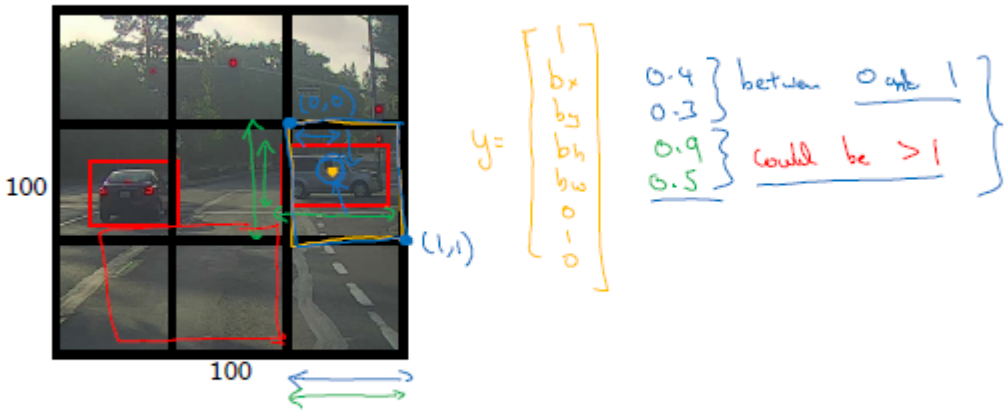


We use YOLO (you only look once) algorithm for this purpose. We apply a grid on each image. Then for each grid cell we come up with its own labels. So for the example below, we have 3 by 3 by 8 in which 3 by 3 is the number of grids and 8 is the dimension of the labels. In practice 3 by 3 is not enough and we may use 19 by 19 for example.
It is a very efficient and fast algorithm.



For specifying the bounding boxes, in YOLO, we assign (0, 0) to the upper left of a grid. So, instead of assigning the positions to the image, we assign it for every single cell. Note that some values may become larger than 1 as the height and width may lengthen more than one greed.
NOTE: this paper is hard!!!!

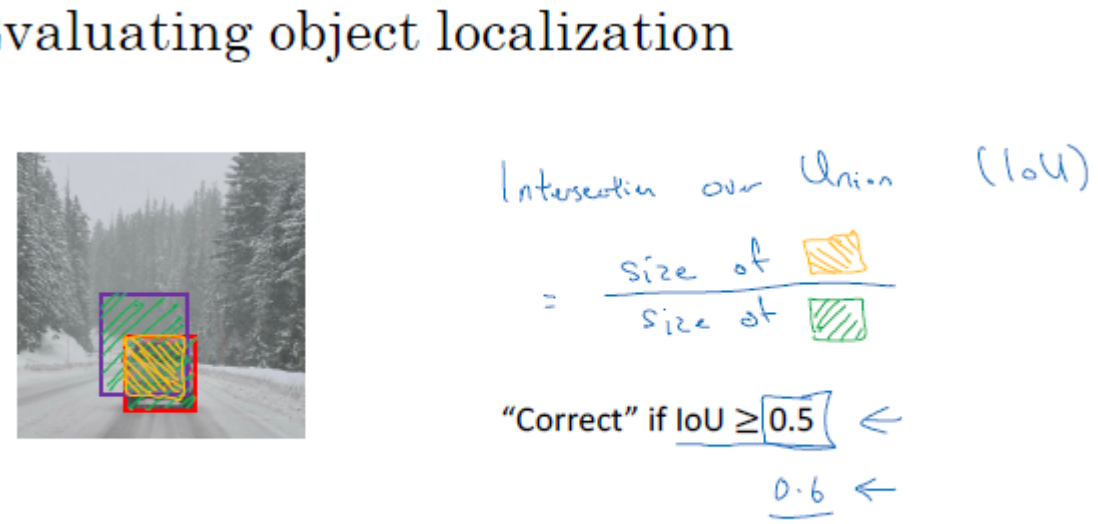
Specify the bounding boxes



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

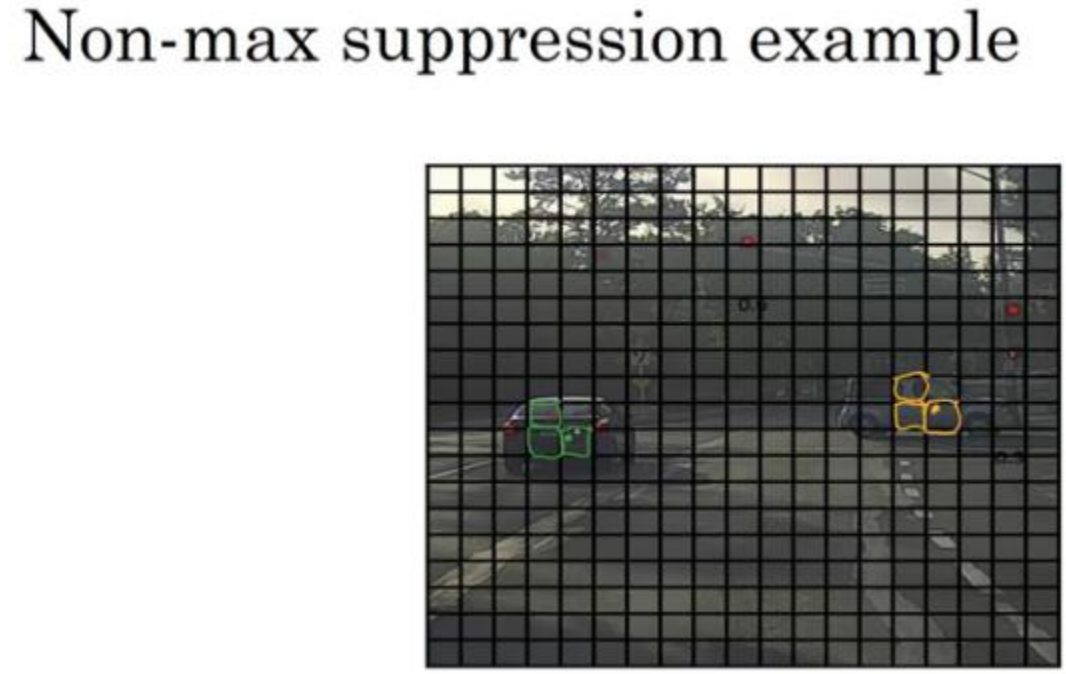
Intersection Over Union (IoU)

To map the localization into accuracy, we compute the size of intersection of two areas and divide it by union of those two areas. In most of the cases, if the IoU value is larger than 0.5 it is true. Of course, the best would be when it is 1.
0.5 is a human chosen convention, no particular reason exits for it. It is a way to find the object localization algorithm is accurate or not. In more general way, IoU is a measure of the overlap between two bounding boxes → how similar two objects are.

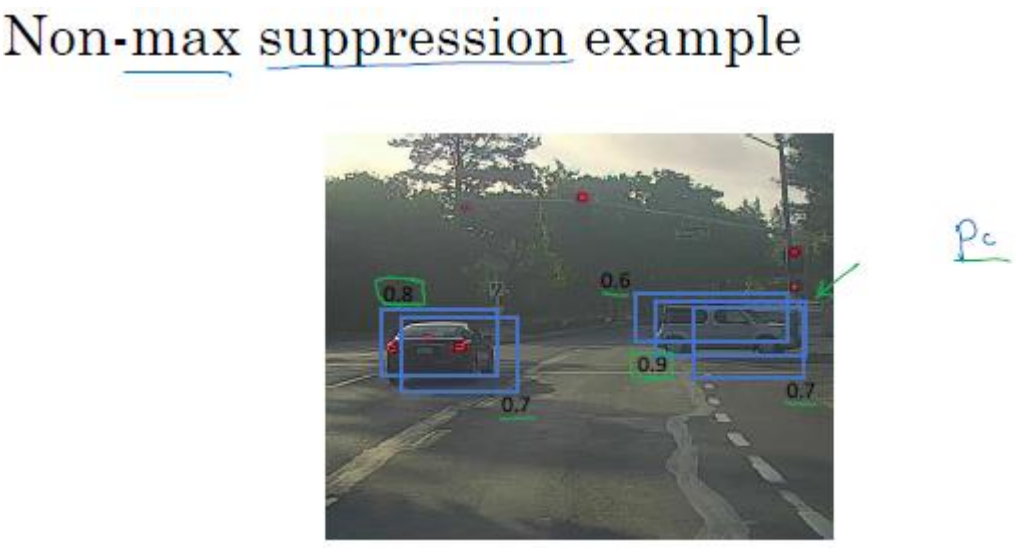


Non-max Suppression

One of the problems so far is that we might detect an object several times rather than once. We want to make it happen only once. The example is shown below.



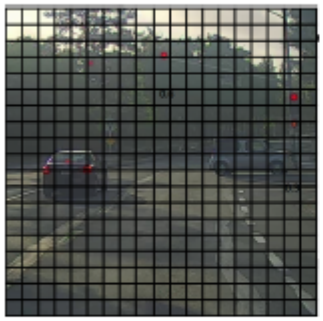
Each of the detected object will get a P_c which is the probability of detecting an object. In this way, we may end up with an object which was detected more than once. For reduce the number of detected objects of one object, we use the non-max suppression in which it will take the largest P_c for an object, then we remove those with the lower IoU values.



For example, let's assume we split the figure into 19 by 19 cell grids; then, we apply ConvNet on the 19 by 19 by 5 grids. Now, here we start our algorithm.

First, we suppress all of the boxes with $P_c \leq$ a threshold like 0.6. So, at this point, we may left off with some boxes each of which has a $P_c > 0.6$. Now, we output the box with the largest P_c as output. We still have some boxes which have not the largest P_c but they are $P_c > 0.6$. Then we discard remaining boxes with the IoU $\geq$ a threshold like 0.5 as they share a lot. Continuously, we choose one with highest P_c and discard the boxes with IoU larger than a threshold until we only get one box.
For each object we have to apply non-max suppression individually.

Non-max suppression algorithm



19×19

Each output prediction is: p_c, b_x, b_y, b_h, b_w

Discard all boxes with $p_c \leq 0.6$

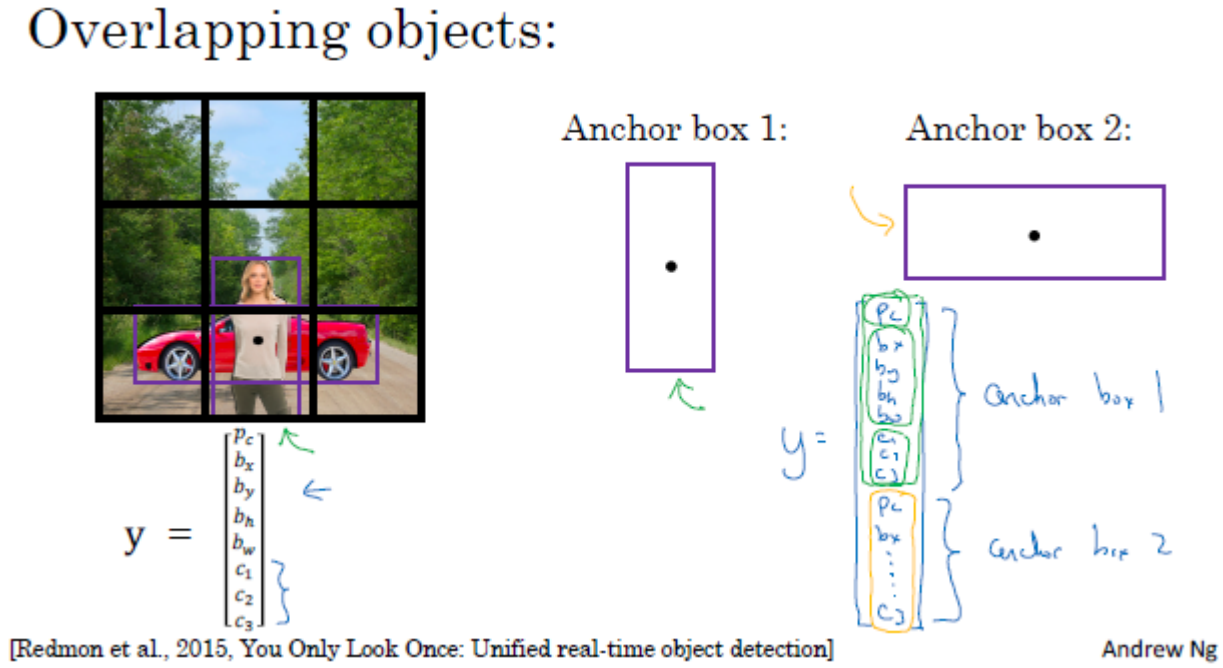
While there are any remaining boxes:

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

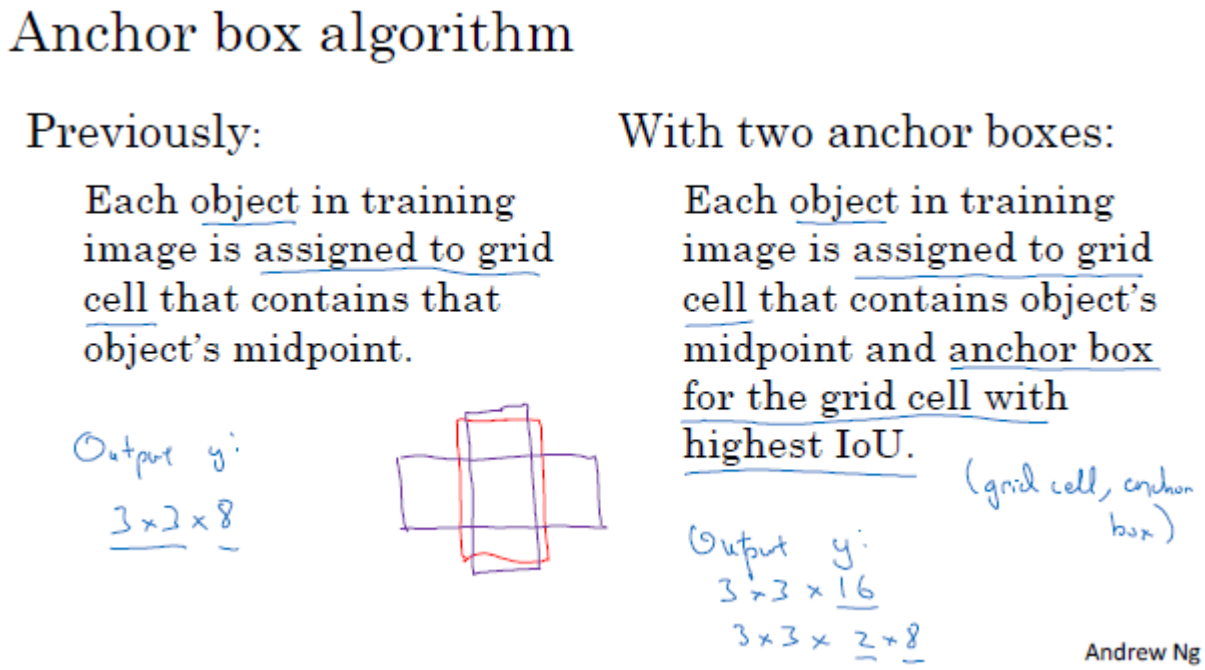
Andrew Ng

Anchor Boxes

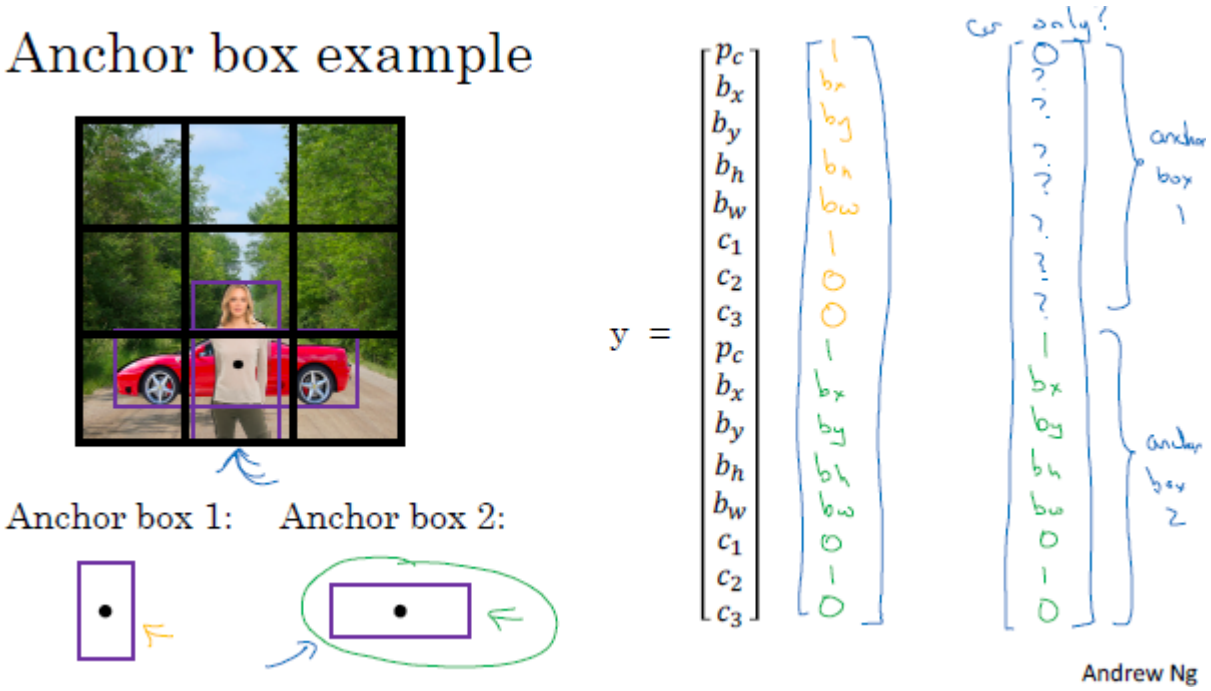
So far, we can detect object with a limitation of each cell can detect only one object, not multiple ones. So we want to understand how to detect multiple of them. The idea is to use "anchor boxes".
In this example with 3 by 3 grids, the mid point of the human and car is the same. The idea of anchor boxes is to use two predefined boxes to associate two predictions with the two anchor boxes. For this reason we stack the output values of the anchor boxes.



For summary, we have:



Let's go through an example with two objects and two anchor boxes (the middle transposed vector) and another example of having only one object which is a car (the right transposed vector). So in two cases of having anchor boxes more than objects and the same amount of objects, we see the algorithm works.

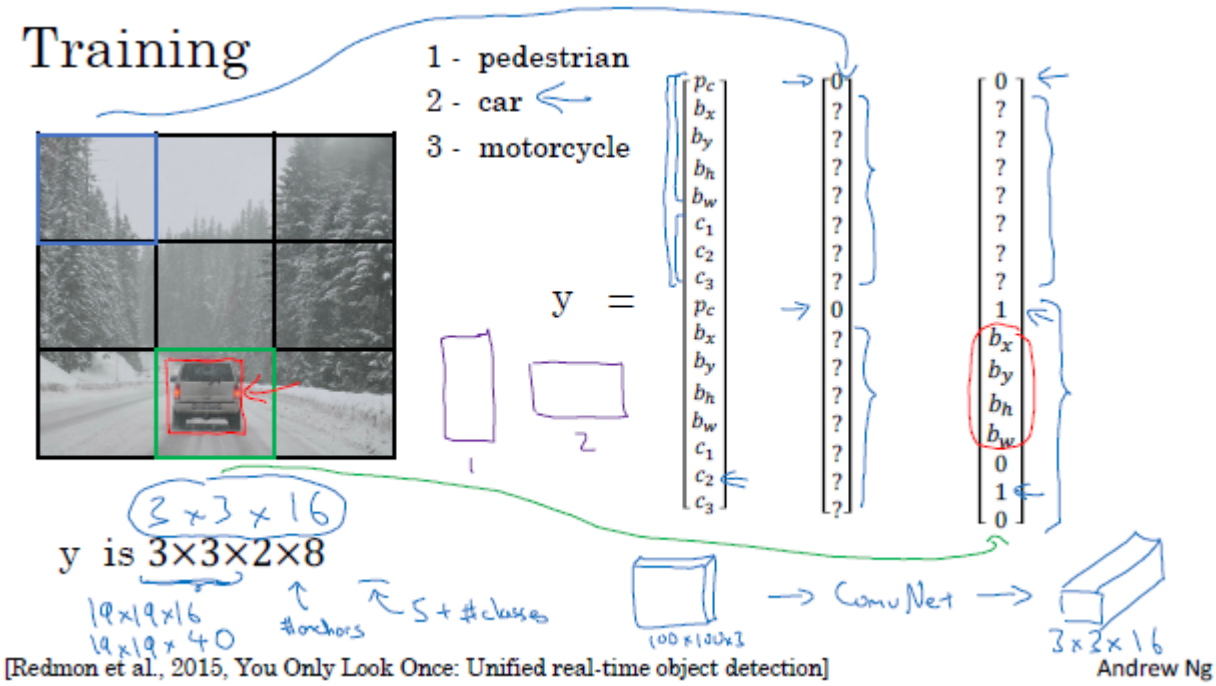


Now what if we have more object than Anchor boxes? This algorithm cannot work well in this situation. Or another case is when two objects in one image need the same shape anchor boxes. In this case as well, the algorithm does not work well. But having many grids it is not much possible to have these kinds of situations.
How to choose the anchor boxes?

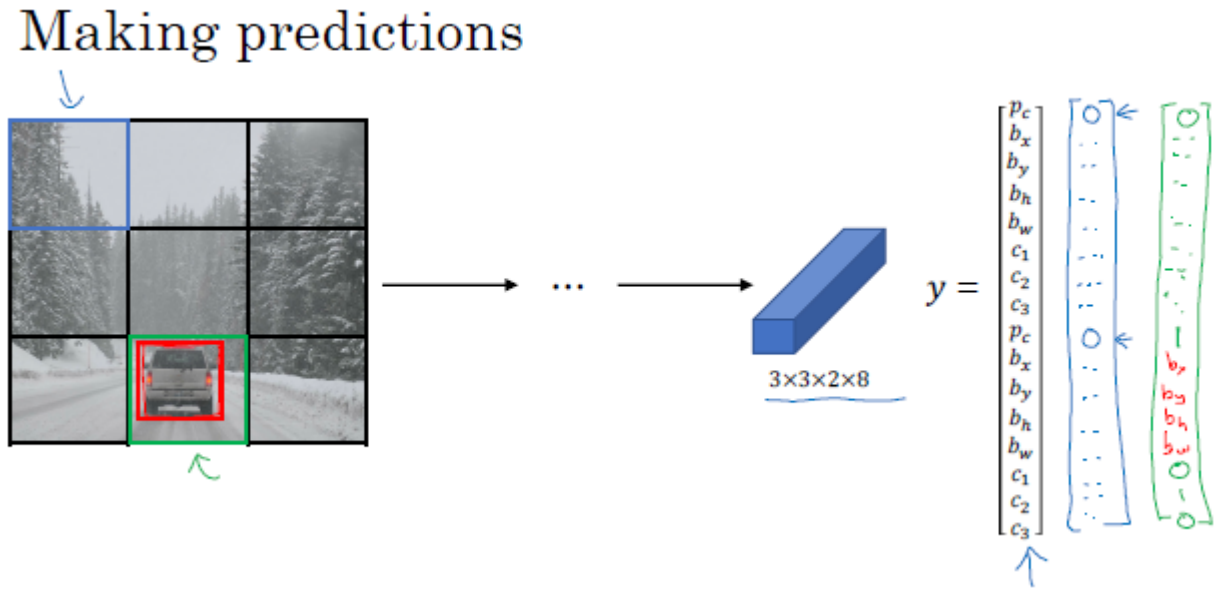
How to choose the anchor boxes? Maybe by hand or use many of them at the same time or even using k-means algorithm as the later paper of YOLO specified.

YOLO Algorithm (overall view)

In this example we are using two anchor boxes with 3 classes. When detecting car, we see its IoU is larger with the anchor box 2, so we assign the value to the anchor box 2.



For making predictions, we just feed the image to the network, and we hope to get results like we see below.



Outputting the non-max supressed outputs

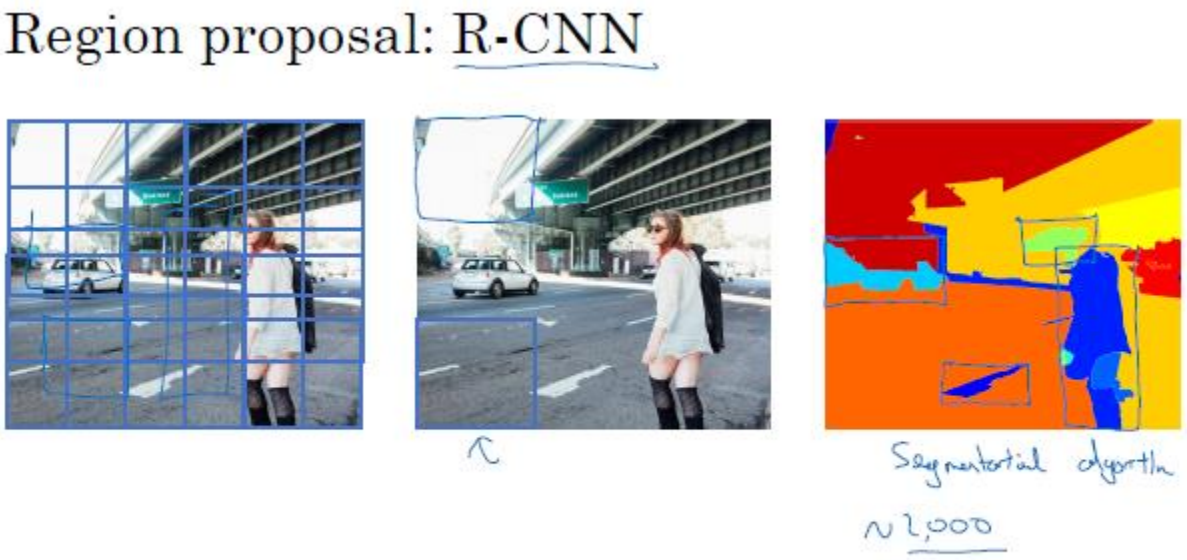
- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

Getting rid of low probability predictions will give us:

Applying non-max suppression will give us:

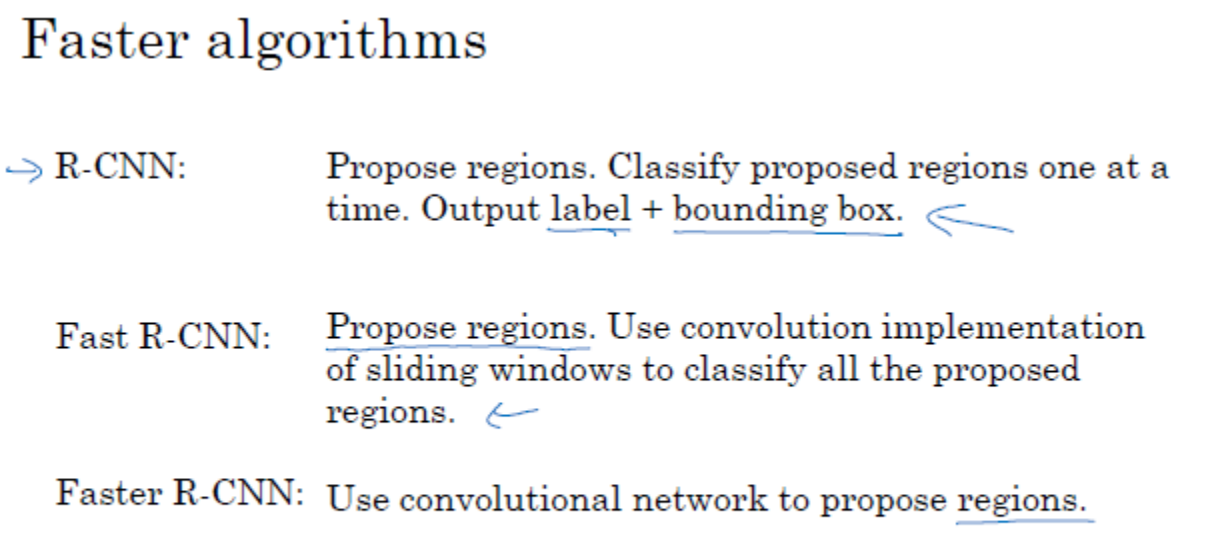
(Optional) Region Proposals

An algorithm, R-CNN which stands for Regions for CNN, tries to just pick a few regions that makes sense to run our classifier on. What could be object? We can use it while using segmentation algorithm (figure on right). In other words, a segmentation algorithm runs over an image and find around 2000 blobs and then we run CNN only on those blobs.



[Girshik et. al, 2013, Rich feature hierarchies for accurate object detection and semantic segmentation] Andrew Ng

However, R-CNN is quite slow.
Fast R-CNN uses one CNN forward propagation to go through all of the blobs.
We also have Faster R-CNN but it still not always faster than YOLO.



[Girshik et. al, 2013. Rich feature hierarchies for accurate object detection and semantic segmentation]
[Girshik, 2015. Fast R-CNN]
[Ren et. al, 2016. Faster R-CNN: Towards real-time object detection with region proposal networks] Andrew Ng

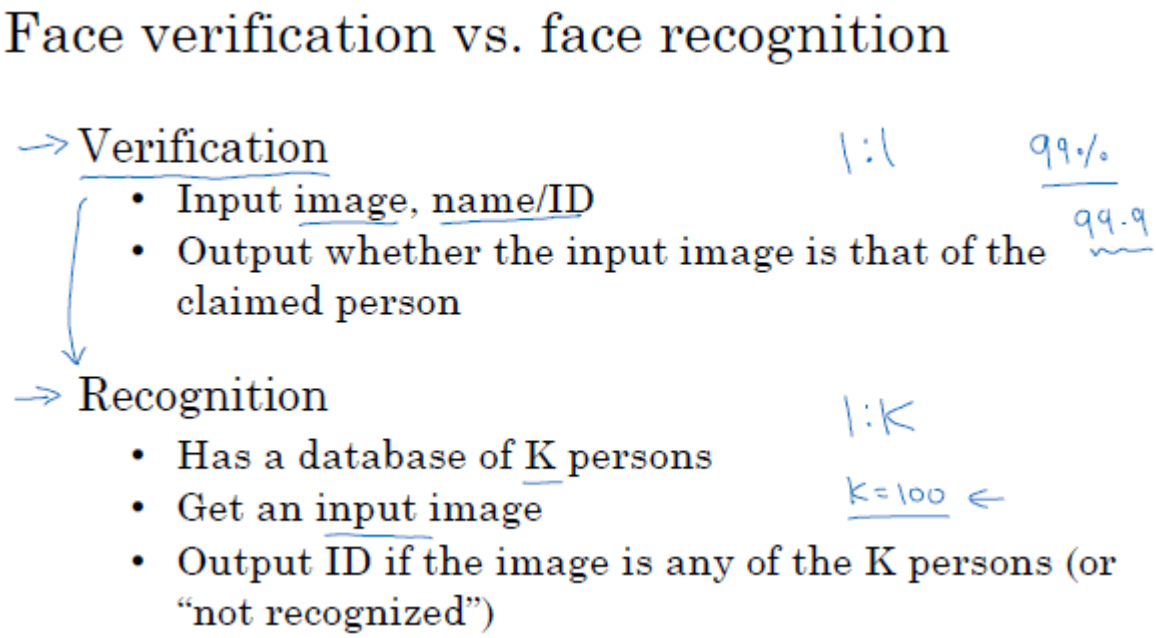
Week 4

What is face recognition?

In *face verification* system we input image and name/ID and get the output as whether the input image is that of the claimed person. → it is a 1 to 1 problem.

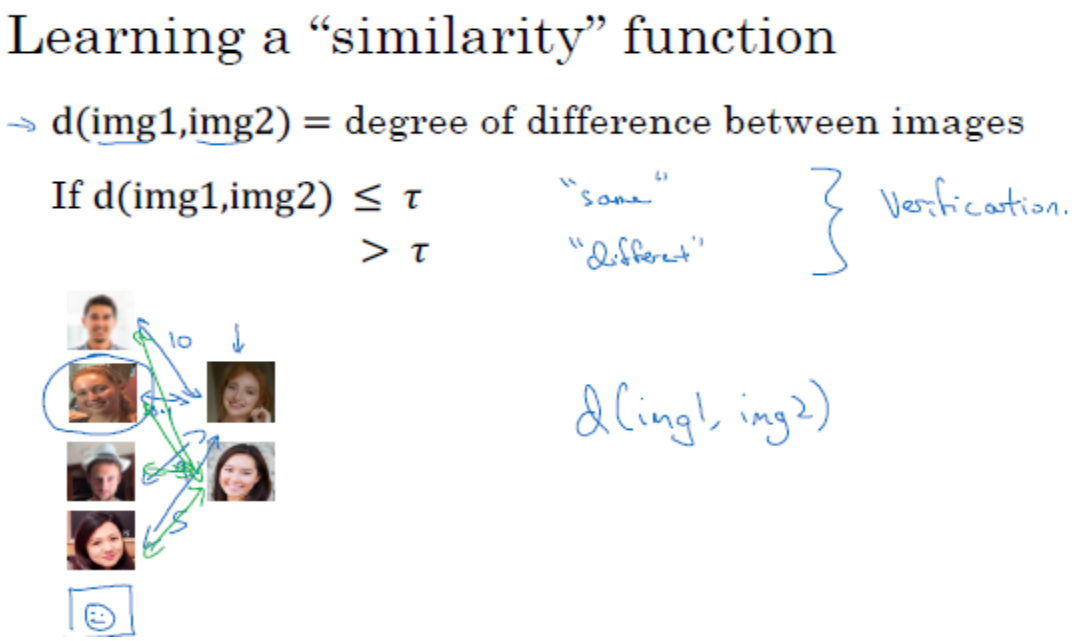
In face recognition we have a database of K persons, we then get an input image, and outputs ID if the image is any of the K persons. → it is a 1 to K problem.

For running a verification with accuracy of 99%, when we run it for recognition task, for K=100 people, as the chance of wrong verification is 1%, for 100 people it gets into a trouble. So we need a verification system with accuracy of 99.9% for example.

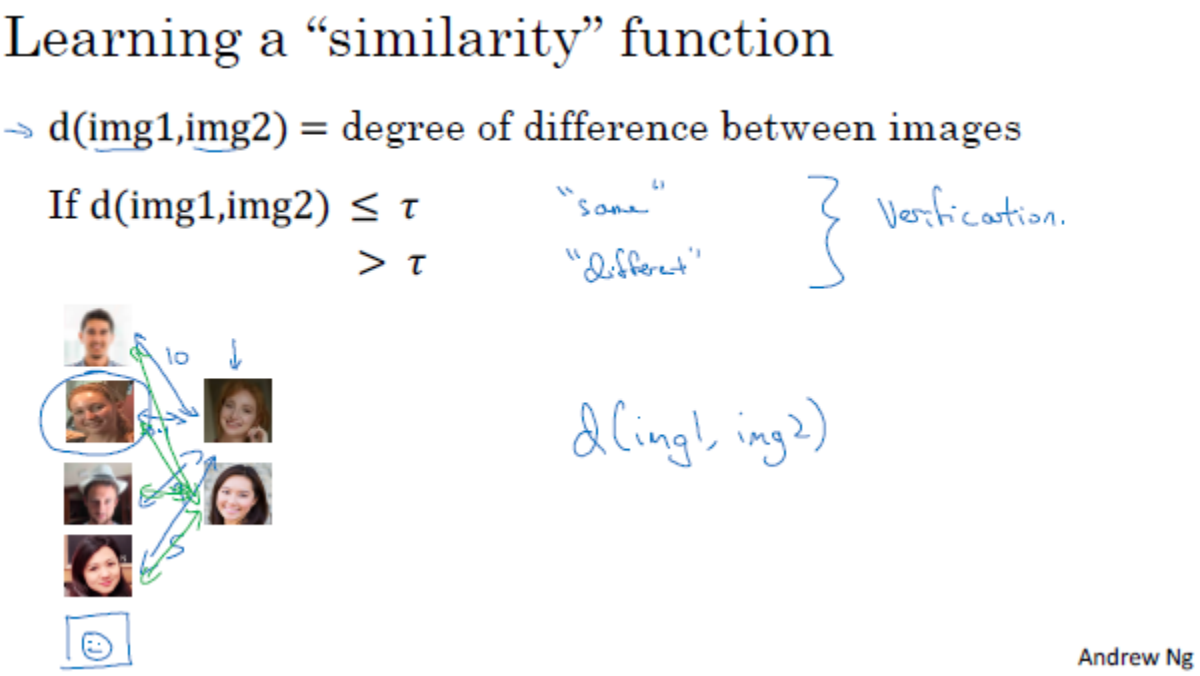


One Shot Learning

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. And, historically, deep learning algorithms don't work well if you have only one training example.
Possible problems with ConvNet: it cannot learn the model based on only one image as NN has the hunger of data. Moreover its last layer of Softmax has as many nodes as the people registered in the database. So by adding a person to database the last layer Softmax will change and will need to retrain the model again.
The outputs of CNN would be either one of the pictures in the DB or none.

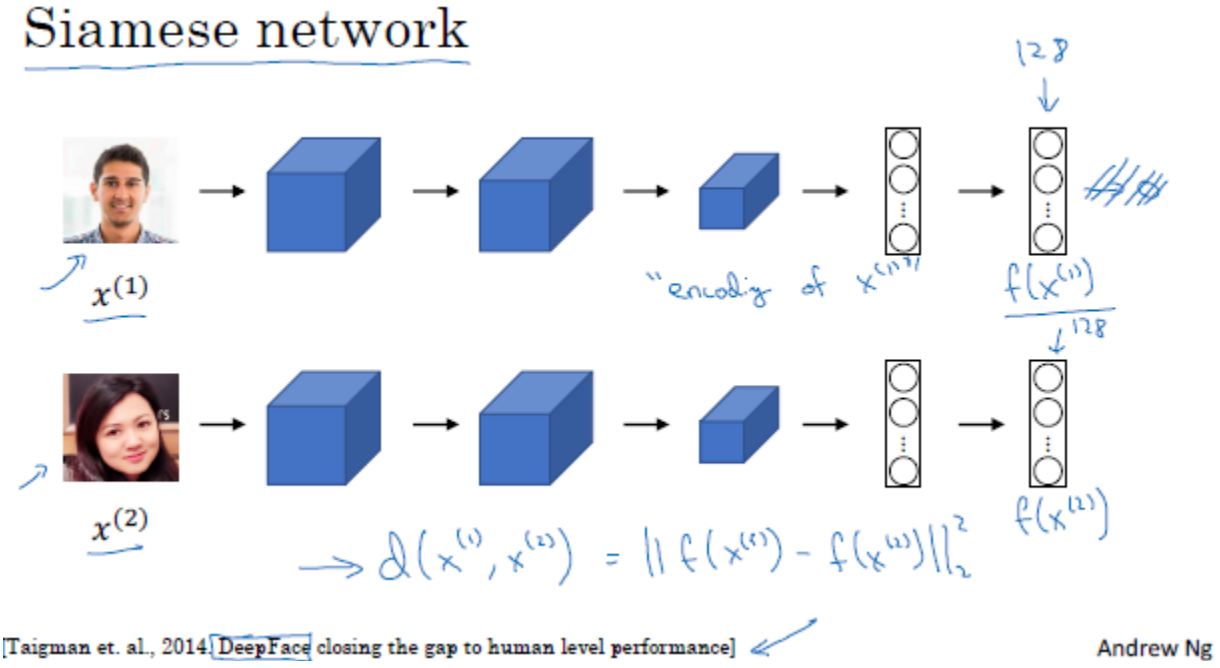


What we do instead, we learn a similarity function. So the degree of difference between two images should be smaller than a threshold which is a hyper-parameter.

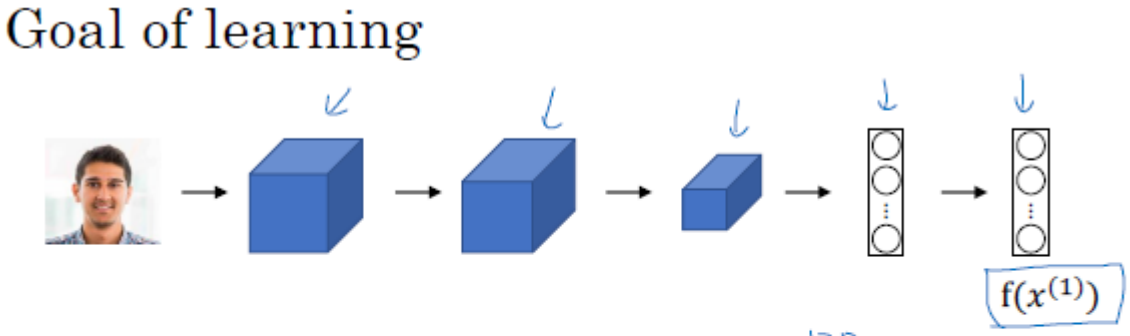


Siamese Network

One way of learning the similarity of two images which is a one shot learning is to decode the image into values by running a set of ConvNet and neural network and getting some values. For example for the two images below, we feed them into the network and get 128 values (image encoded values). By comparing these two encodings we can find out their identity and similarity.



So the loss function would be the one which given two pictures of same person returns a very small value, while giving two pictures of two different persons, it returns a large value.



Triplet Loss

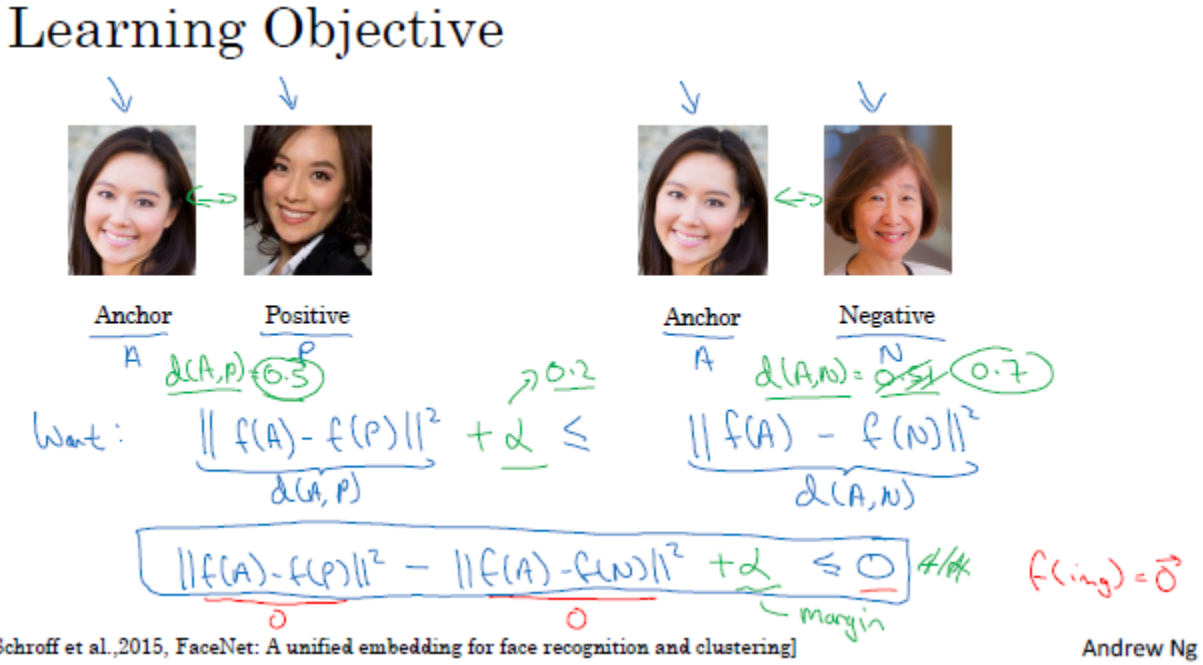
It uses the idea of looking at an image as anchor, and a positive and negative image. So distance of the positive image and anchor must be low and its distance with negative image must be high. So we get:

$$\| f(A) - f(P) \|^2 \leq \| f(A) - f(N) \|^2$$

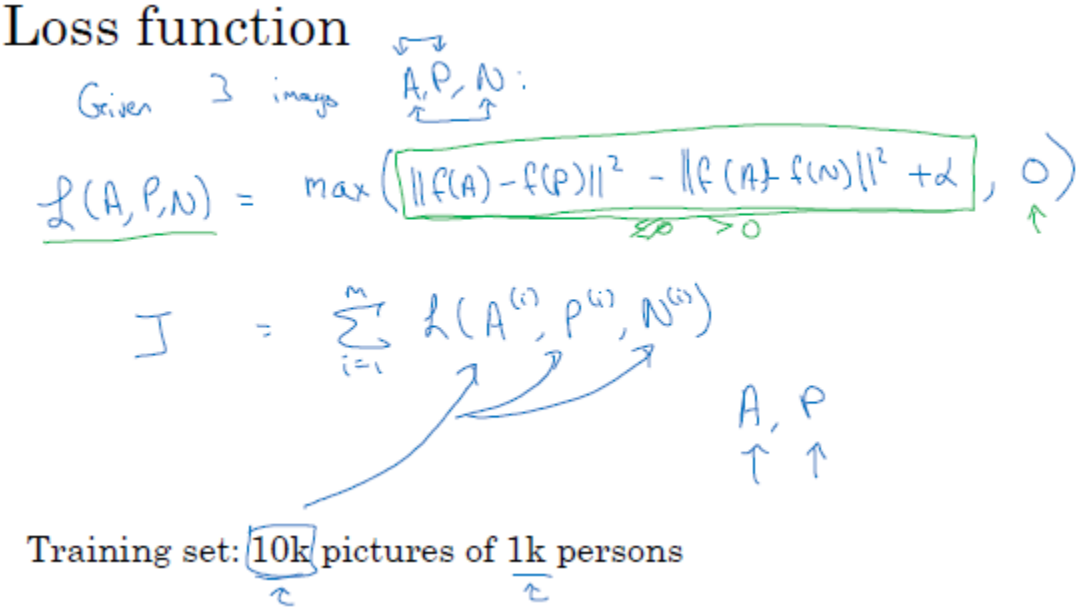
$$\rightarrow \| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2 \leq 0$$

Now if trivially we use an encoder that always outputs same values, to prevent it to happen we add another hyper-parameter to gives a value larger than 0 so that the trivial answer of the encoding don't trick the loss function that 0 would be a solution to the problem. We call the α hyper-parameter margin.

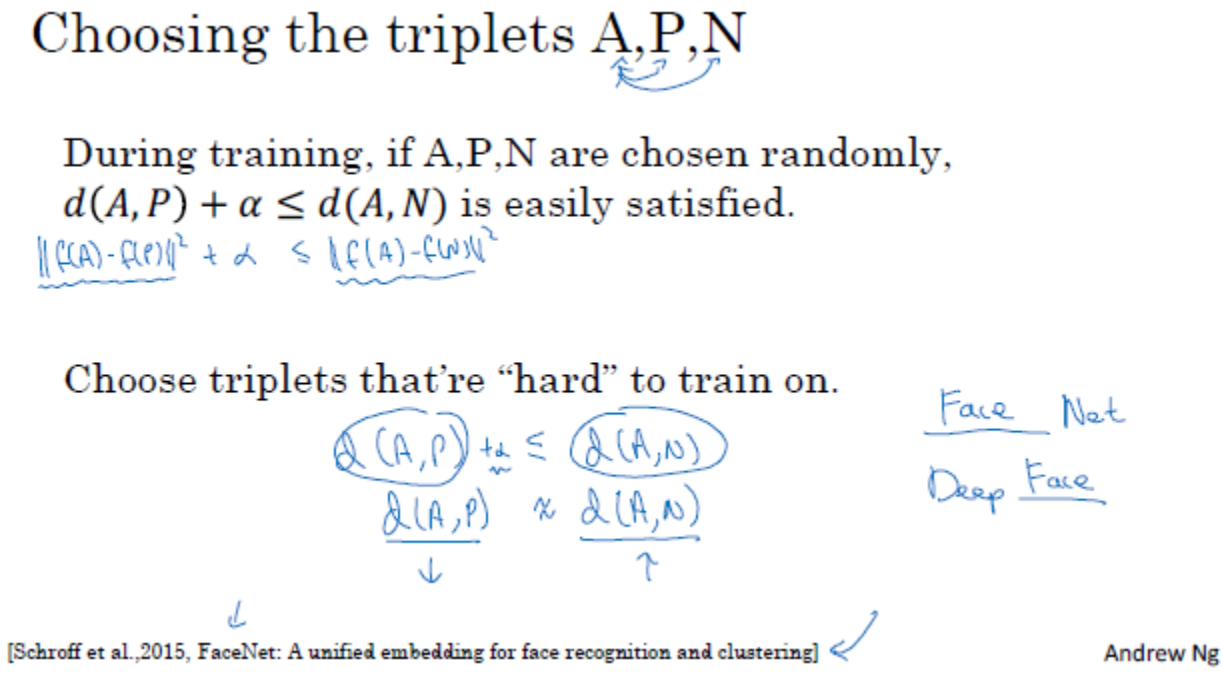
$$\rightarrow \| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2 + \alpha \leq 0$$



First of all, the loss function below uses the max argument as we may get negative answers which correct. Other thing is that, we need to have pairs of anchors and positives for training, at least 10 for each person. After training, we can use the one-shot learning problem.



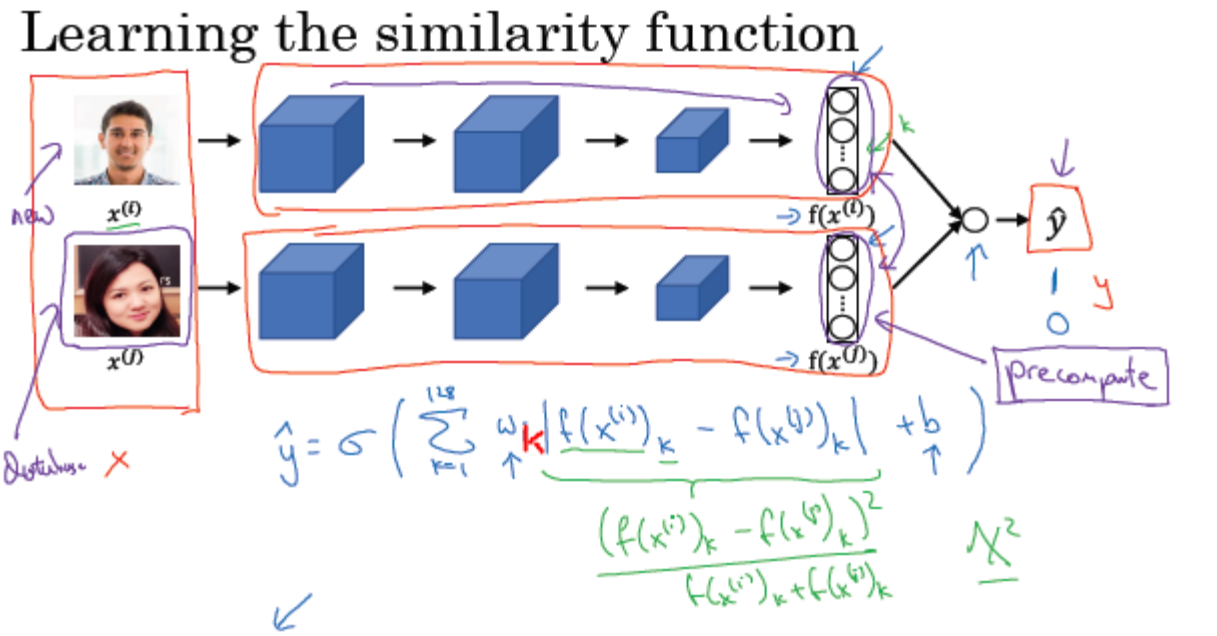
If we give an anchor and positive image of a person randomly along with negative example, the loss easily get satisfied. So we better try to give triplets which are hard to train on since the loss has a harder time to satisfy the loss function.



Face Verification and Binary Classification

We can learn the similarity function by giving two images to a CNN and then gets the output of 1 or 0 from that. It is an alternative to triplet loss function. So we use a logistic regression at the end of the CNN. So for the LR, we take element-wise differences of two encoded values.

So for decrease the computation, when a new user comes in, we compute its encoded values but use the precomputed values of the previous users and images we had. So we do not need to compute the encoded values for every single image over and over again.

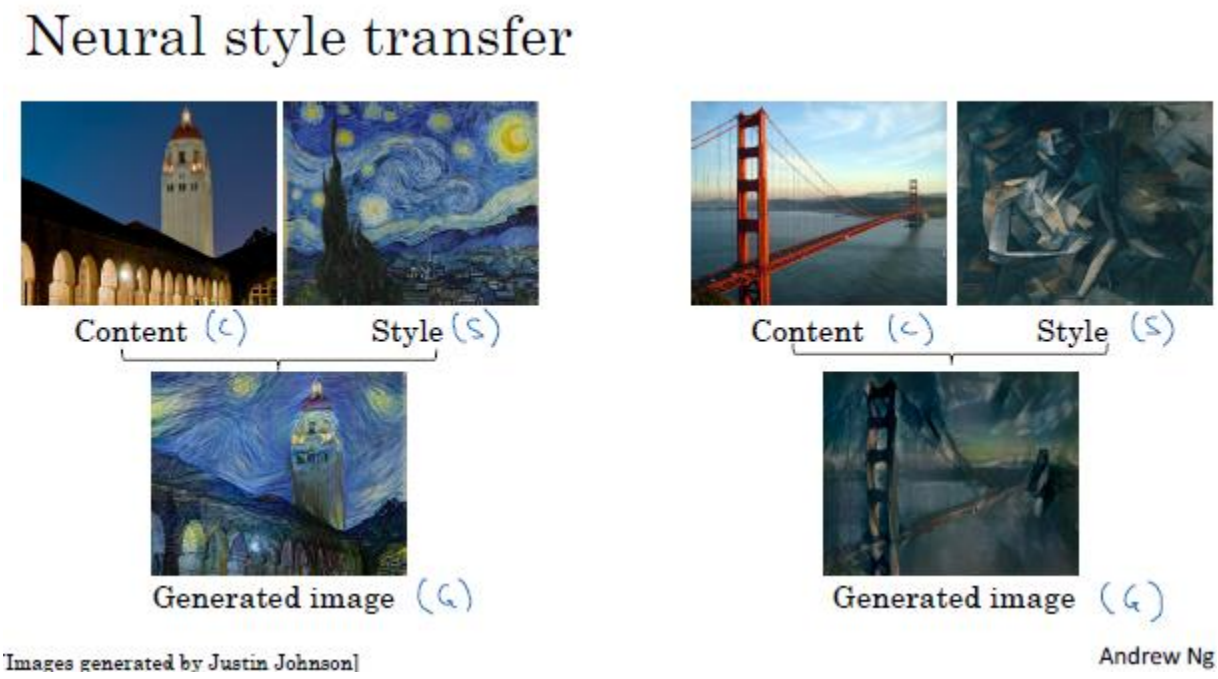


[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

Andrew Ng

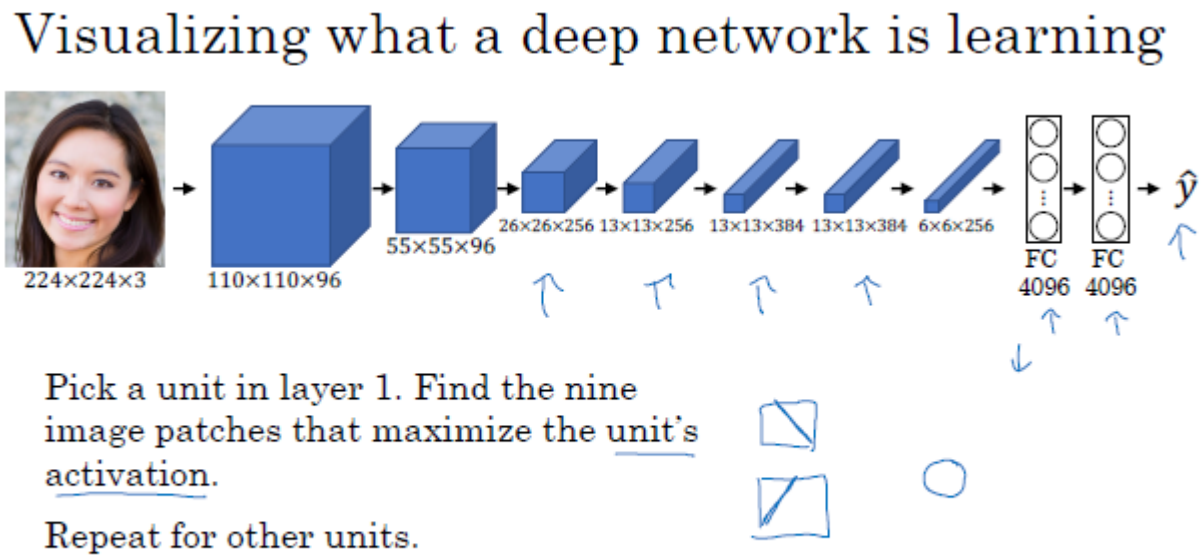
What is neural style transfer?

We have three images, content image (actual image), style image (like a Picasso painting), and get the generated image which is the output of the previous given images.



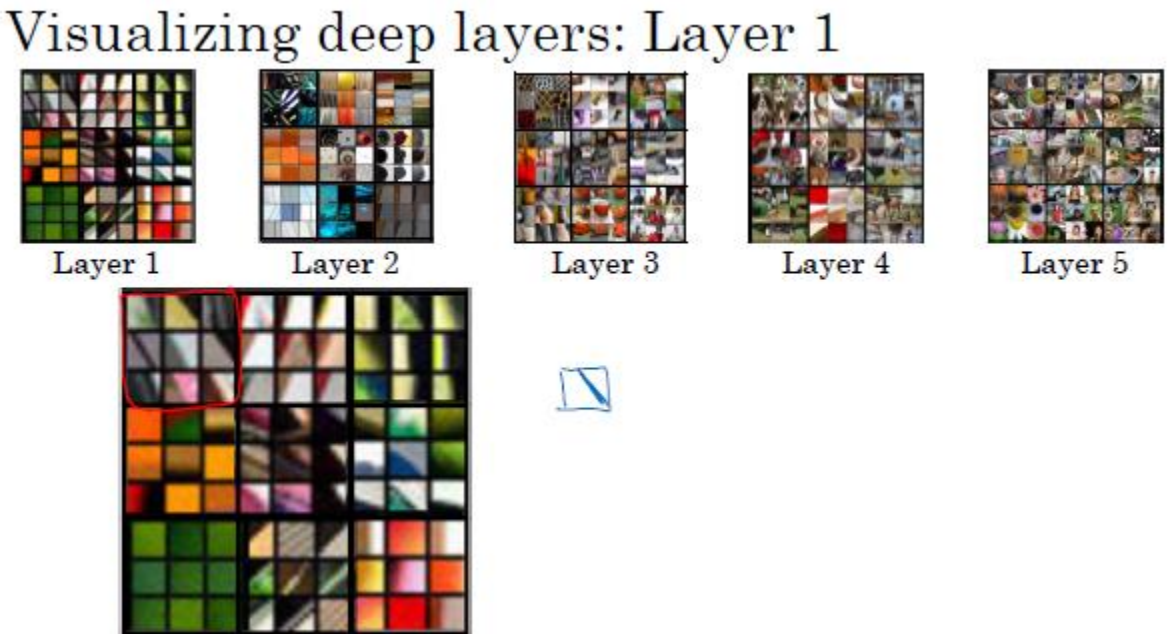
What are deep ConvNets learning?

We want to see what the hidden layers of different layers are generating and doing.



[Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks] Andrew Ng

Let's go deep and see what they show:



Visualizing deep layers: Layer 2


Layer 1


Layer 2


Layer 3


Layer 4


Layer 5

Andrew Ng

Visualizing deep layers: Layer 3


Layer 1


L 4


Layer 5

Visualizing deep layers: Layer 4

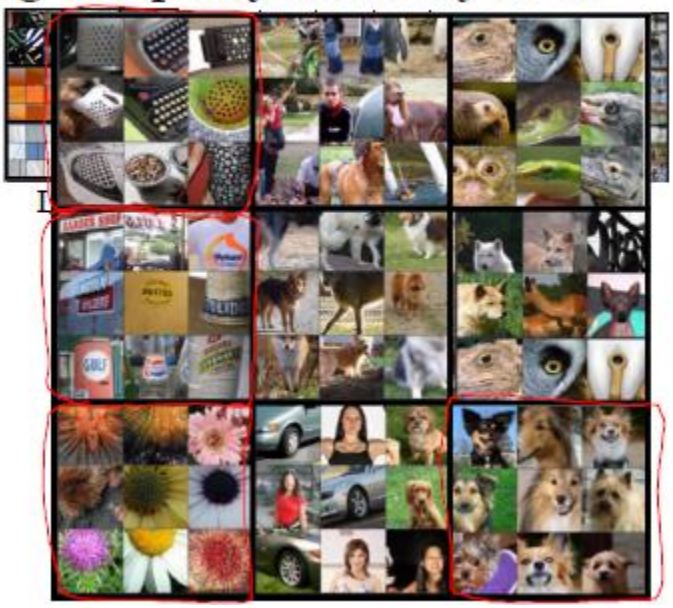
Lay

Layer 4

Layer 5

Visualizing deep layers: Layer 5

Layer 1

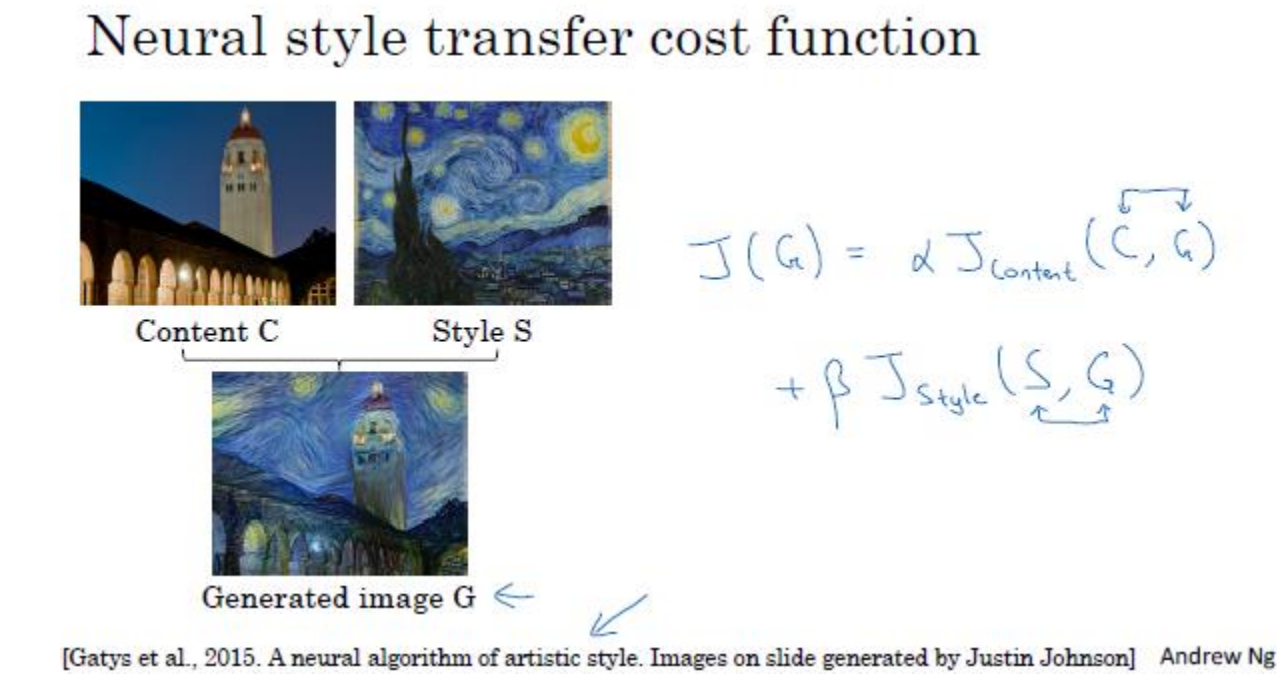


Layer 5

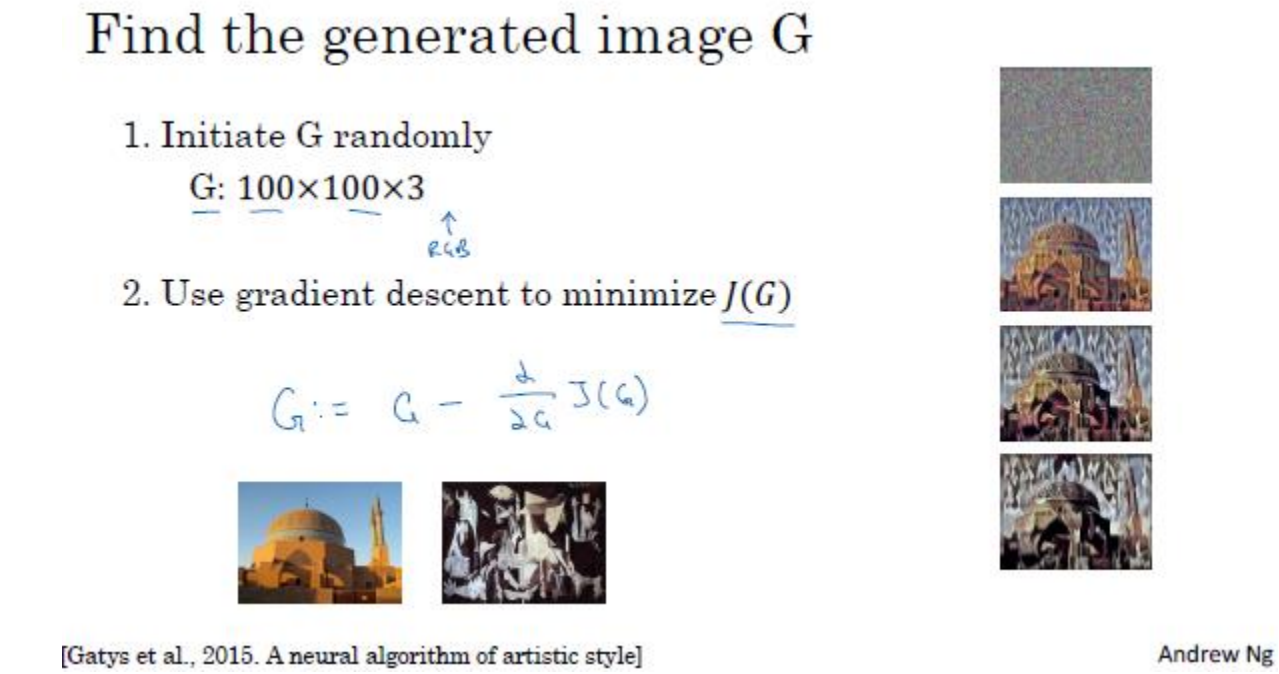
Andrew Ng

Cost Function

The loss function for the neural style transfer is based on similarity of the content image and generated image in addition to similarity of the style image and generated image.



So we start with a random valued image with 3 channels as RGB, then feed it to the network to minimize its loss. The image improvement is shown on the right hand side of the slide below.



Content Cost Function

We can see all the details of the loss function below. The one thing is that the coefficient of the loss function does not matter much as α hyper-parameter exists.

Content cost function

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

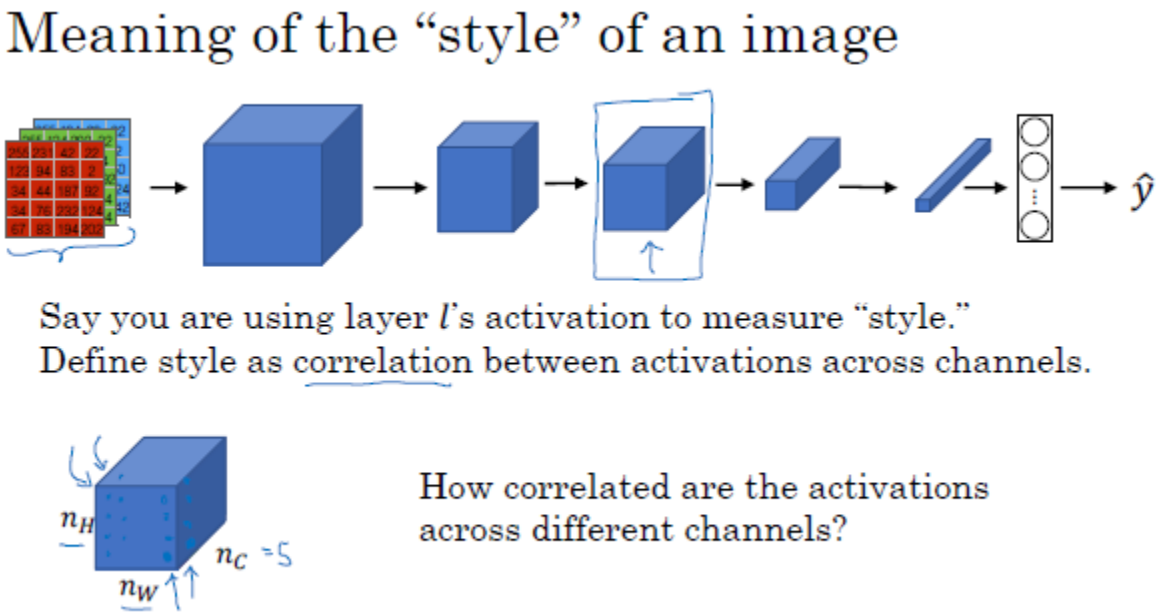
- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

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

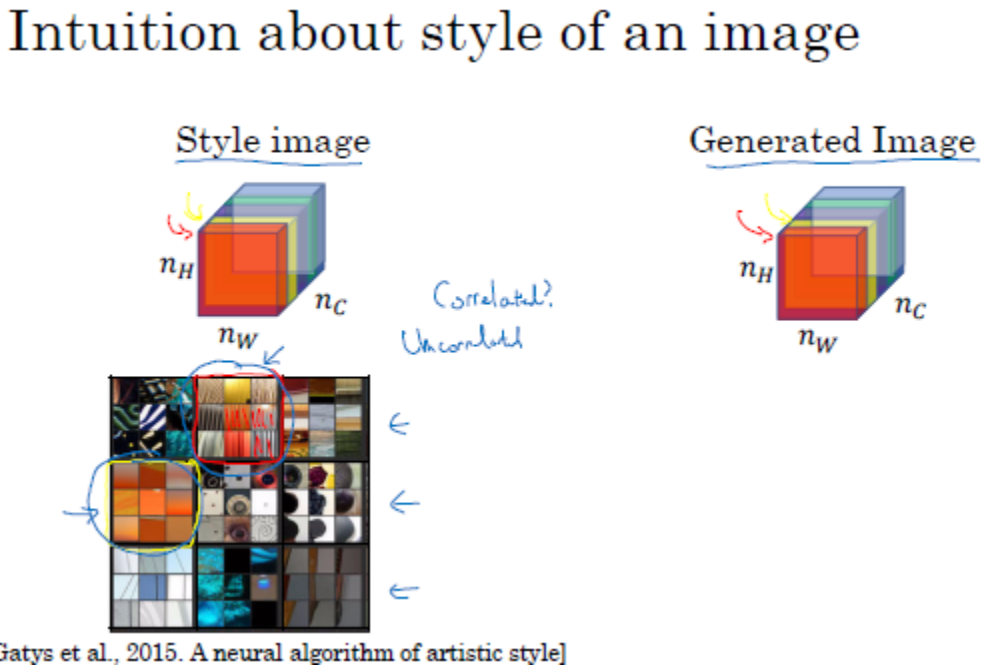
[Gatys et al., 2015. A neural algorithm of artistic style]

Andrew Ng

Style Cost Function



The correlation between channels means that if for example on the red channel we see vertical lines, on the yellow channel we see orange-ish shapes. The uncorrelated one would be when there is no definite pattern for the yellow channel when we see vertical lines on the red channel. So we take the distance of the correlations between channels as a way to understand how well a generated image and style image are similar.



So for taking the correlation of two matrices, what we do is that for all of the heights and widths, we sum the multiplications of activations of those matrices for one specific layer. Again the coefficient does not matter due to existence of a hyper-parameter.

Style matrix

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

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

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

"Gram matrix"

$$J^{[l]}_{style}(S, G) = \frac{1}{(\dots)} \left\| G^{[l](S)} - G^{[l](G)} \right\|_F^2$$

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

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

Andrew Ng

Style cost function

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

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

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

$$\min_G J(G)$$

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

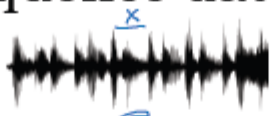
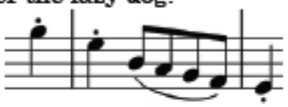
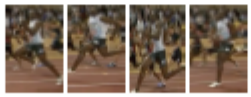
Andrew Ng

Course 5

Week 1: Recurrent Neural Networks

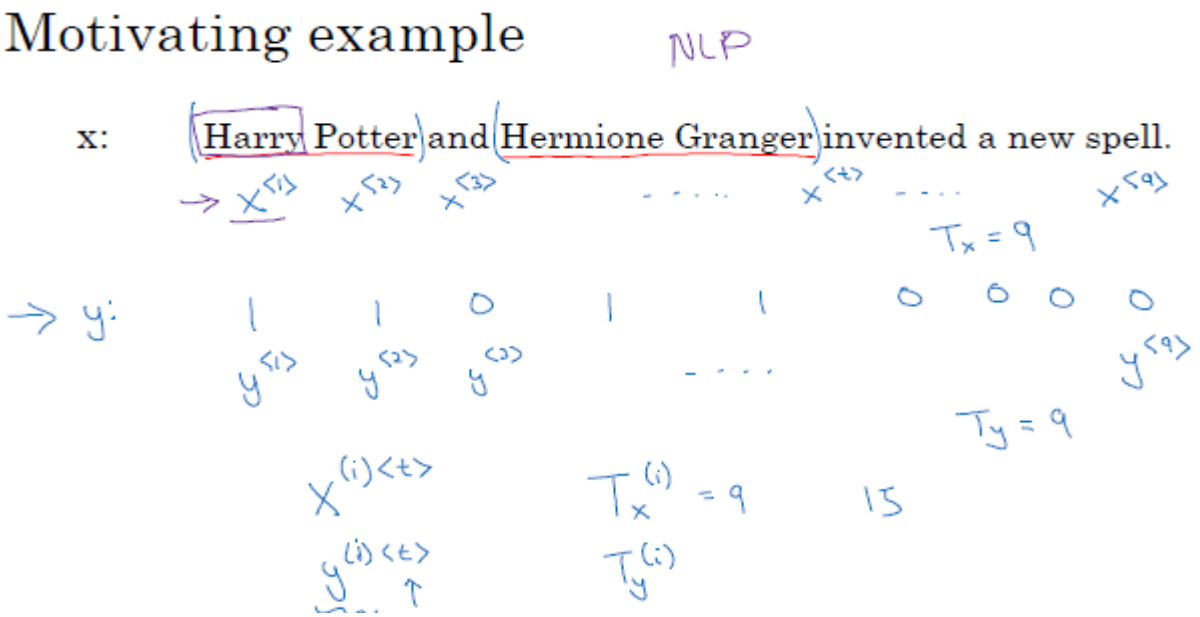
Why sequence models?

Examples of sequence data

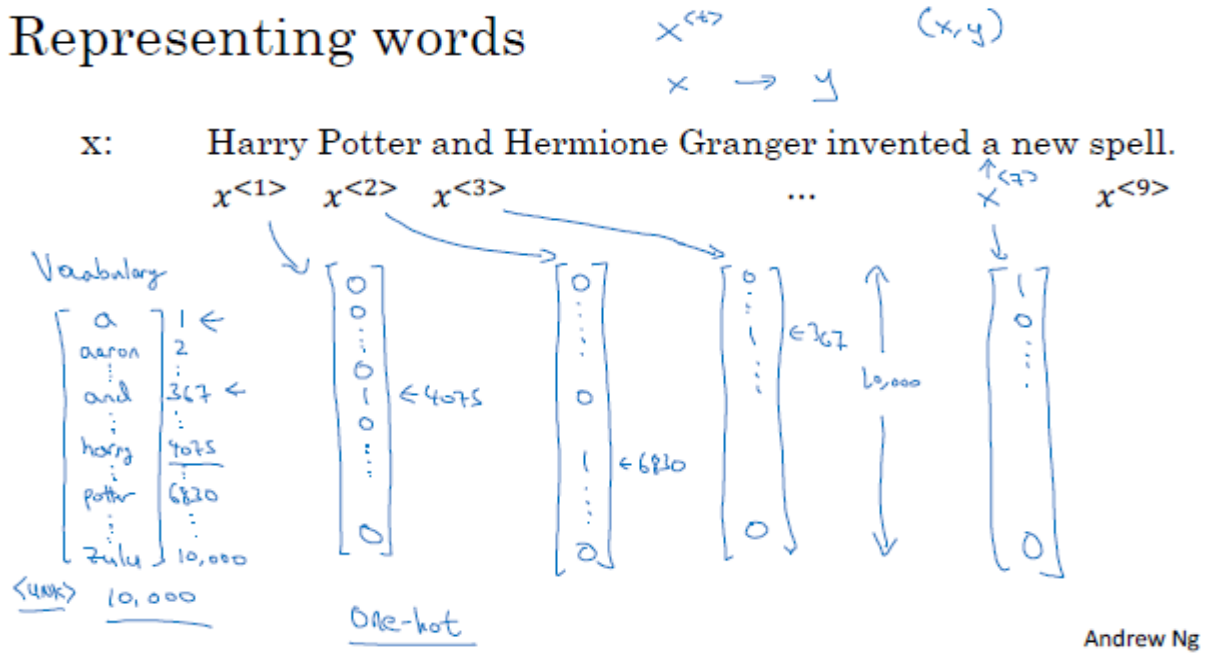
Speech recognition		"The quick brown fox jumped over the lazy dog."
Music generation		
Sentiment classification	"There is nothing to like in this movie."	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	Do you want to sing with me?
Video activity recognition		Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	Yesterday, Harry Potter met Hermione Granger.

Notation

Given a sequence model, like a sentence, we want to implement "name entity identification", used in search engines mainly. The target output will be 0 for non-names and 1 for names. So it needs a start and end place but for this example we omit this assumption. Look at the slide below to see how the notation works for this example. For the sake of this course, we use "(i)" for i_{th} training example and we use "$< i >$" for the i_{th} word in the sentence.



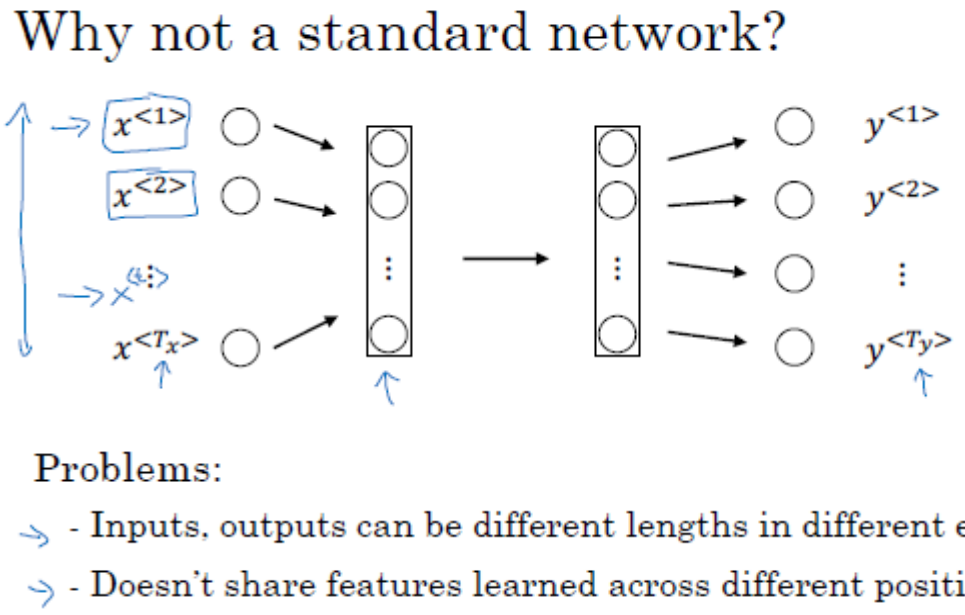
Now we need to know how to represent a word. We have a vocabulary list. Each word has an index in the dictionary. Larger the dictionary size it is better. Something like 50000 words. Then we use one-hot representation of each word in the dictionary. The word which we want to choose has the value of 1 and the rest are 0s. We also have a word as "UNK" for a word which is not in the dictionary.



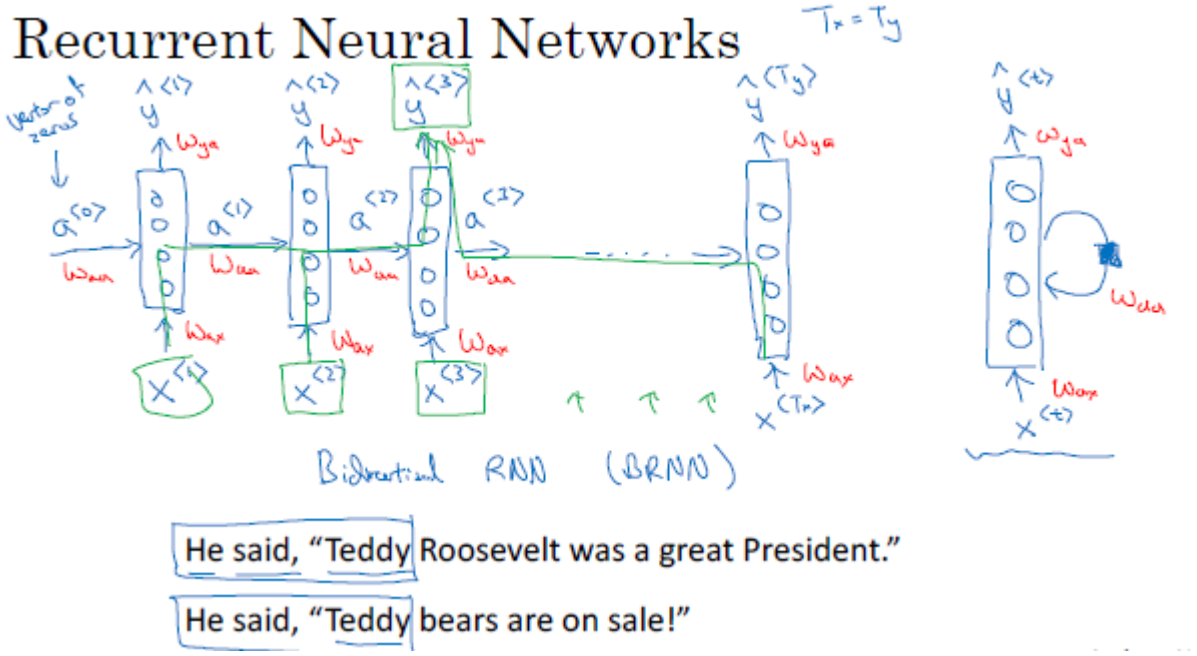
Recurrent Neural Network Model

We can input these values into neural network and get its output as 0 or 1 for each word. It doesn't work however. The problems are:

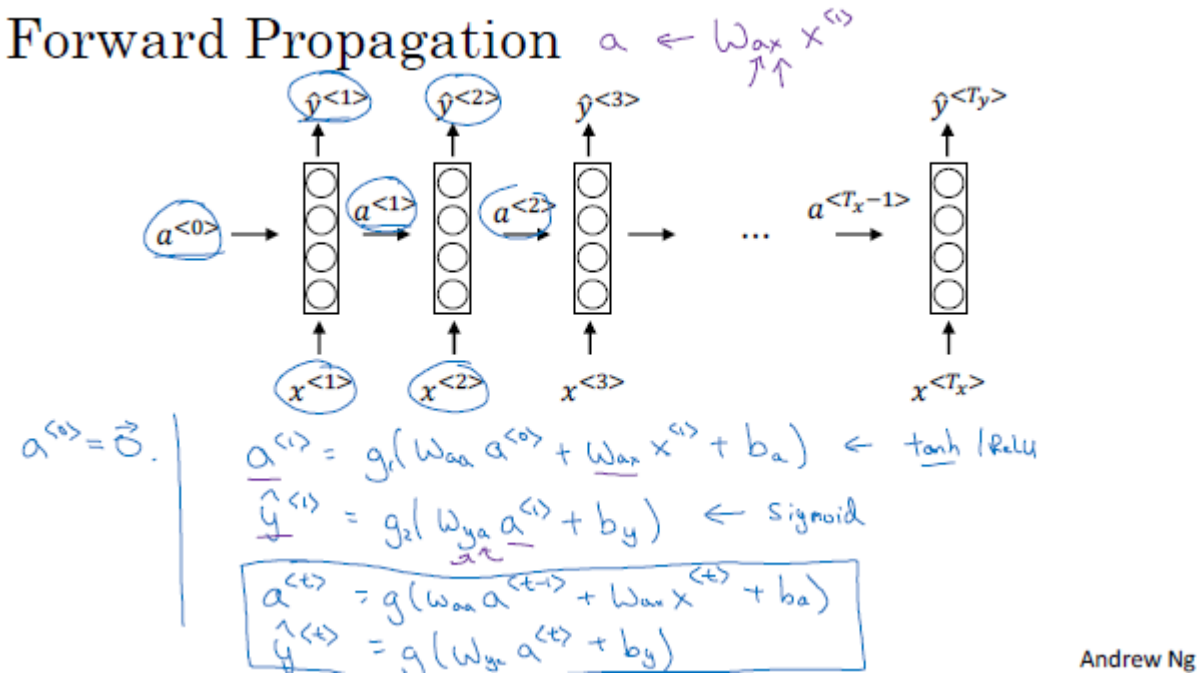
1. Input and outputs can be different lengths in different examples. What we can do is to take the longest input or output and use padding for the rest. It still is not much helpful.
2. This model doesn't share features learned across the different positions in the text. It is similar to what we had in mind in CNN.
3. Other problem is that the input size is huge as for each word we want to represent a one-hot index in a dictionary with size of for example 10000. In this case, the number of variables are too much.



In RNN we input an input into a neural network and try to predict its label instantly. For the other words we make the same kind of blocks. The crucial idea is that we pass the information, activation of the previous block, to the next one. In the example below, $T_x = T_y$, the length of input and output are equal. For other cases, we change the architecture a little bit. The right hand side architecture is another way to depict the RNN. One weakness of the RNN is that, it relies on the information from past. But sometimes we need some information from future words which affect our decision regarding the earlier ones. For examples we see beneath the slide below, Teddy in the first one is a president while in the second one it is a bear. Bidirectional RNN (BRNN) can easily fix this problem. We will see it later.



So, for the RNN what we have two activation functions, one for the output, and one for passing information. As for the output, we mainly use Sigmoid or Softmax activation function and for the latter we use either tanh or ReLu; although tanh is a better choice for this problem. As for the $a^{<t>}$ we incorporate $a^{<t-1>}$ and input vector with the weights assigned specifically to them. As for the output, we only incorporate $a^{<t>}$ with its weight.



We want to simplify the notation, so we stack vertically two weights of $[w_{aa}: w_{ax}]$ and call it w_a. For the same reason we stack horizontally the $[a^{<t-1>}: x^{<t>}]$. Then, we can rewrite it as follows:

Simplified RNN notation

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

(100,100) 100 (100,10,000) 10,000

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

$$a^{<t>} = g(W_a [a^{<t-1>}, x^{<t>}] + b_a)$$

$$[W_{aa}; W_{ax}] = W_a \quad (100, 10100)$$

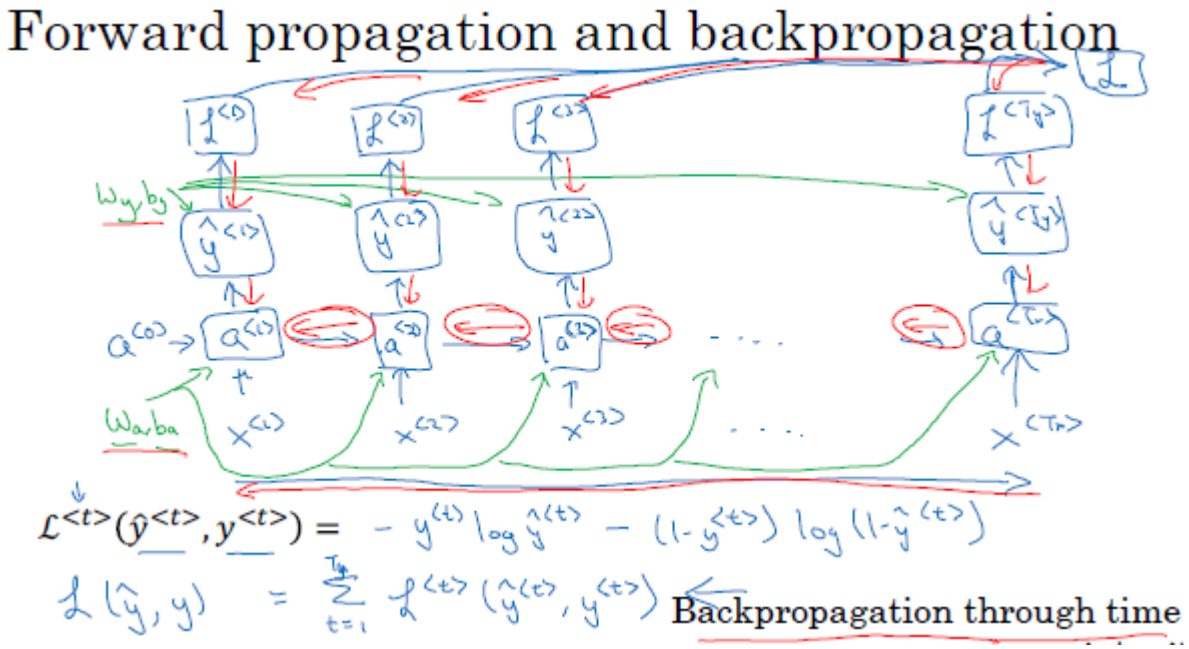
100 × 100 | 100 × 10000

$$[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} \quad 100 + 10000 = 10100$$

$$[W_{aa}; W_{ax}] \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} = W_{aa}a^{<t-1>} + W_{ax}x^{<t>}$$

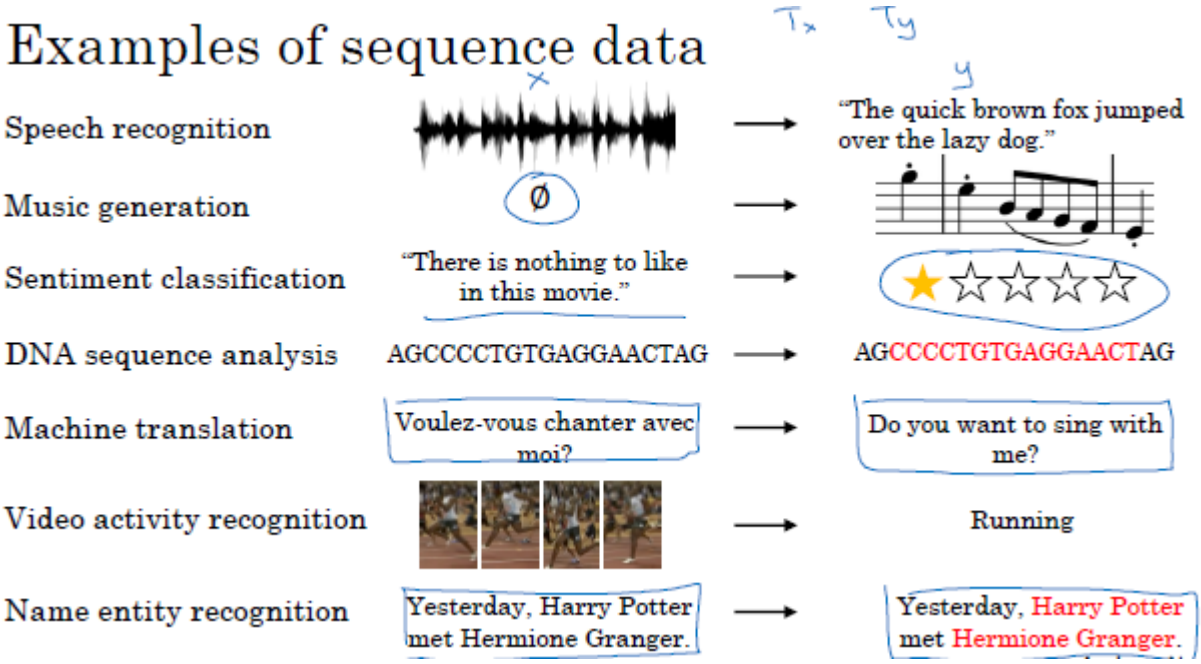
Backpropagation through time

So, we talked about the forward propagation formulas. It is shown again on the slide below. Note that the bias and weights, w_a, b_a and w_y, b_y are shared in all time-steps. For its loss function, we use the usual cross entropy. Now, each block has its own loss and for computing the overall loss we add them up. Then, the red lines show in the slide, is the way gradient descent updates the weights and biases through backpropagation. When backpropagation goes all the way back to the first words, we call it backpropagation through time while for the forward propagation, we only go from the first word to the last one.

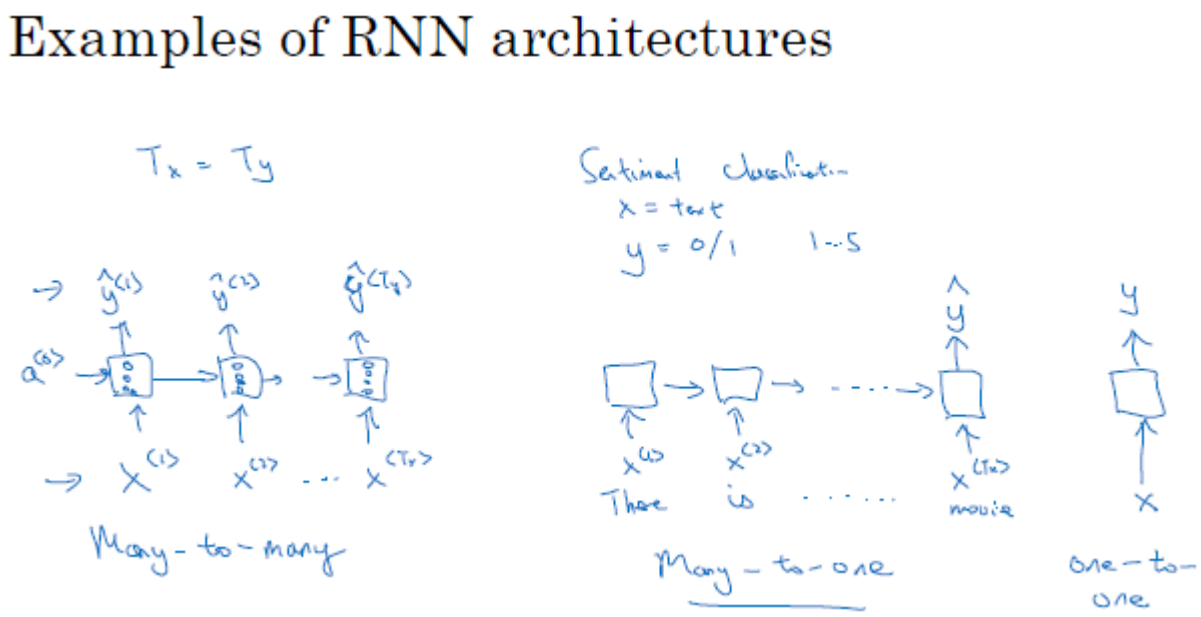


Different types of RNNs

Let's get back to the very first examples of this course. In some cases the length of the input is the same as the output like the examples we saw so far. In some cases like sentiment analysis, the input size can be huge or small while the output size is predefined, like 5 integer values. In some cases like machine translation, the input and output size may change and different.

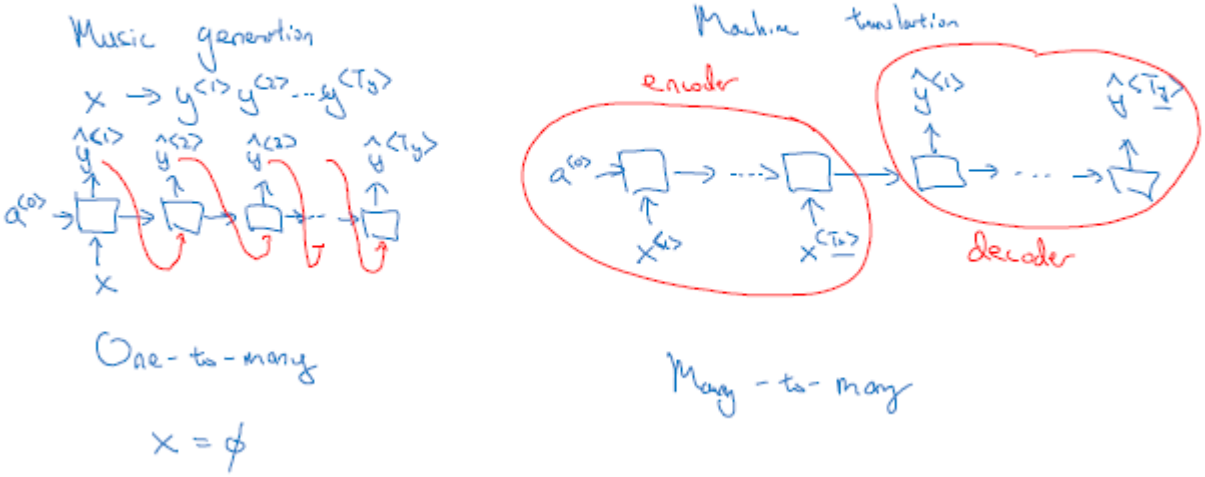


The models we saw so far was many-to-many. For sentiment analysis it is many-to-one. We also have one-to-one but it is a normal neural network. For the case of one-to-many is the music generation problem. In this case and many other cases we may want to feed the result of the previous block to the next one.



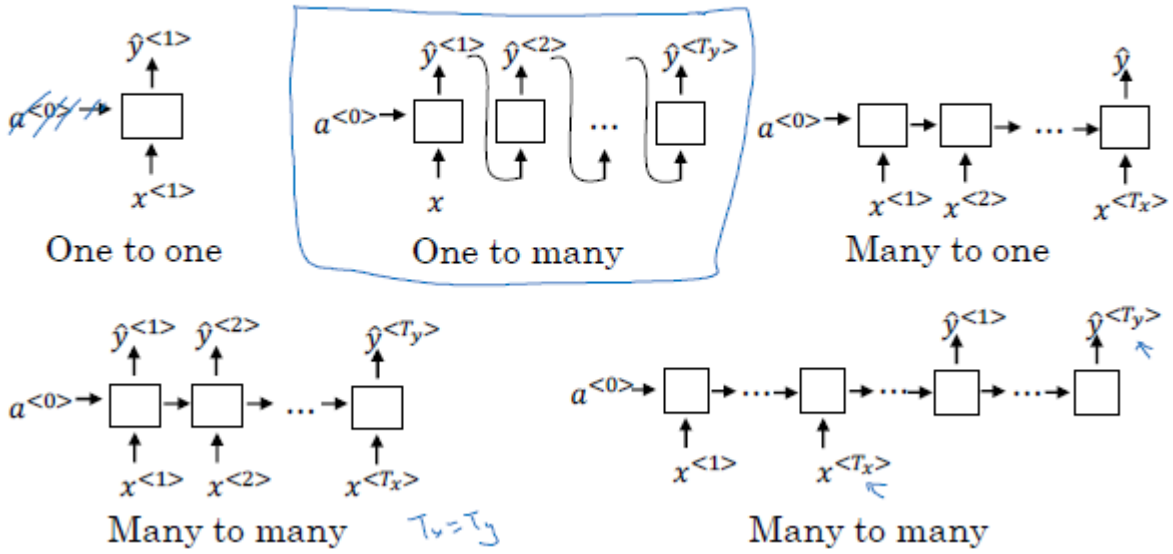
There is also an application of many-to-many, like machine translation, which is interesting. The architecture could be something like the one we see on the right hand side of the slide below.

Examples of RNN architectures



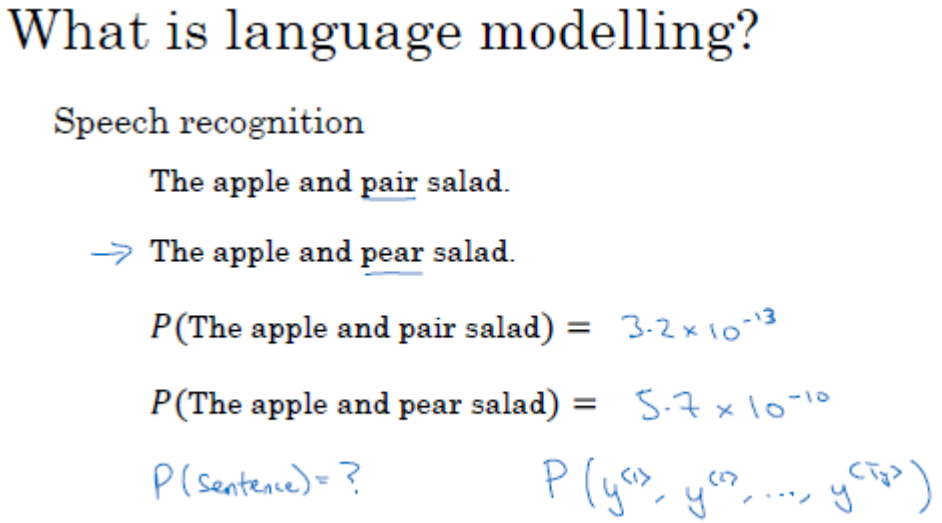
Let's to recap architectures we talked about:

Summary of RNN types

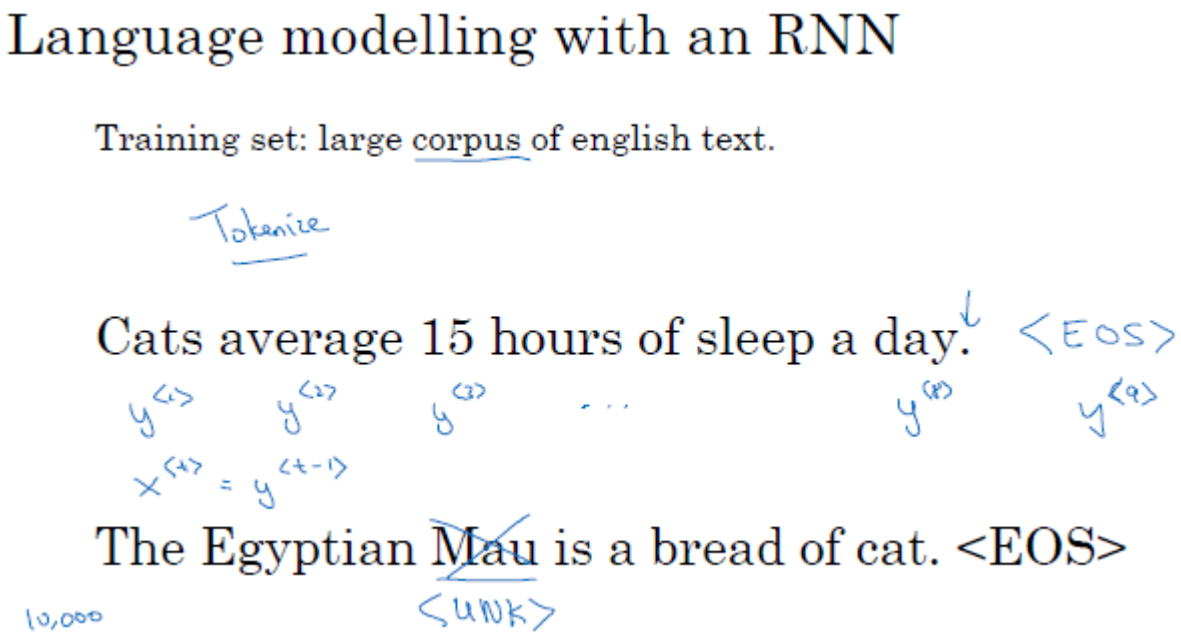


Language model and sequence generation

In the task of language modelling, like the one in the speech recognition, we may say two words with the same pronunciation. So the language model counts their probabilities and choose the one which is more likely to be true. It is the same problem as translation system.

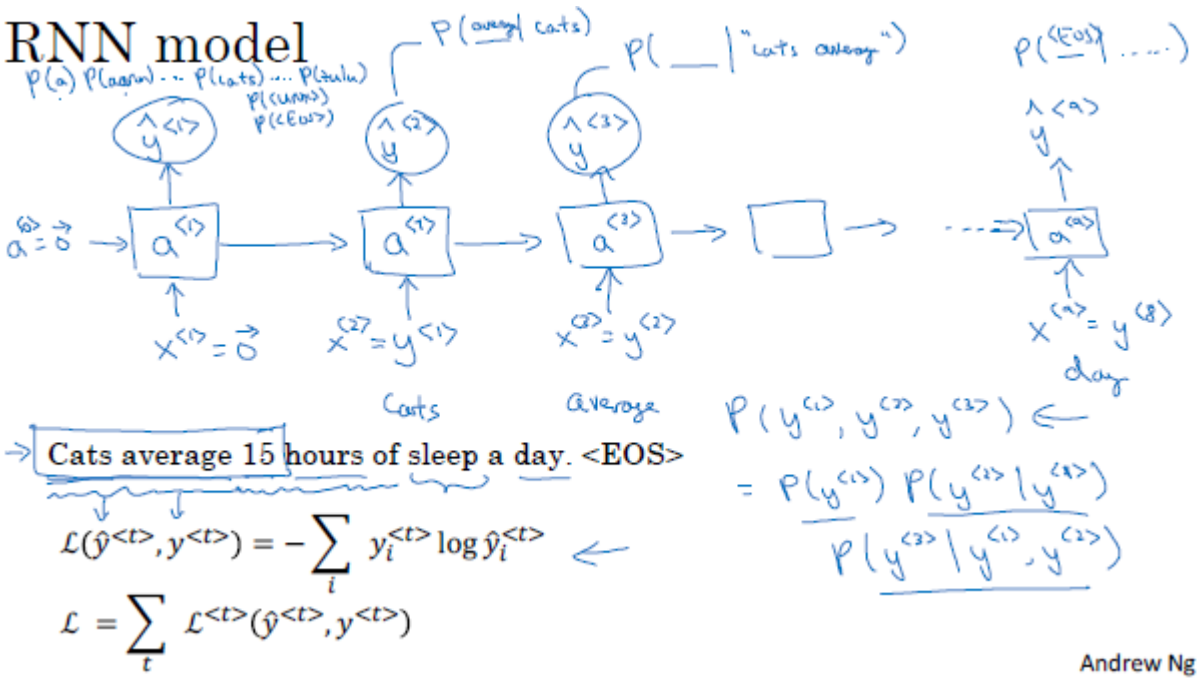


For making the language model, we use a large corpus of English text for training the model. The first thing we do with each sentence is tokenizing. Here, the tokens are vocabularies in the sentence and we map it to a one-hot dictionary format. We also use a <EOS> for end of the sentence token as well in many places. For words not in the dictionary as <UNK> word in the dictionary.



The RNN model for this problem is to first feed the first block with a 0s vector and we ask the NN in this block to predict the most probable word using Softmax. For example, if the dictionary size is 10000, each word has a probability and Softmax will choose the one with the highest probability. Then, we feed the correct word to the next layer and ask its NN to predict the most probable word given the previous one(s). We use the usual cross-entropy for calculating loss of each block and we sum them up to calculate the overall loss.

RNN model



Cats average 15 hours of sleep a day. <EOS>

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

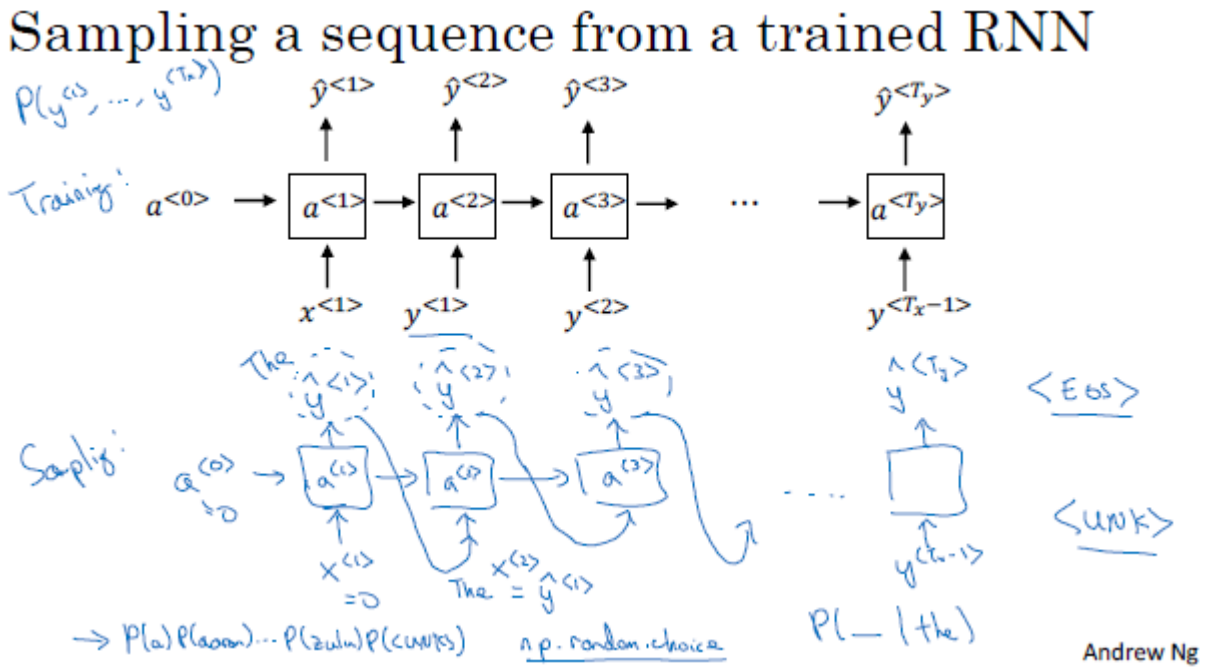
$$\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>})\, P(y^{<2>} \mid y^{<1>})\, P(y^{<3>} \mid y^{<1>}, y^{<2>})$$

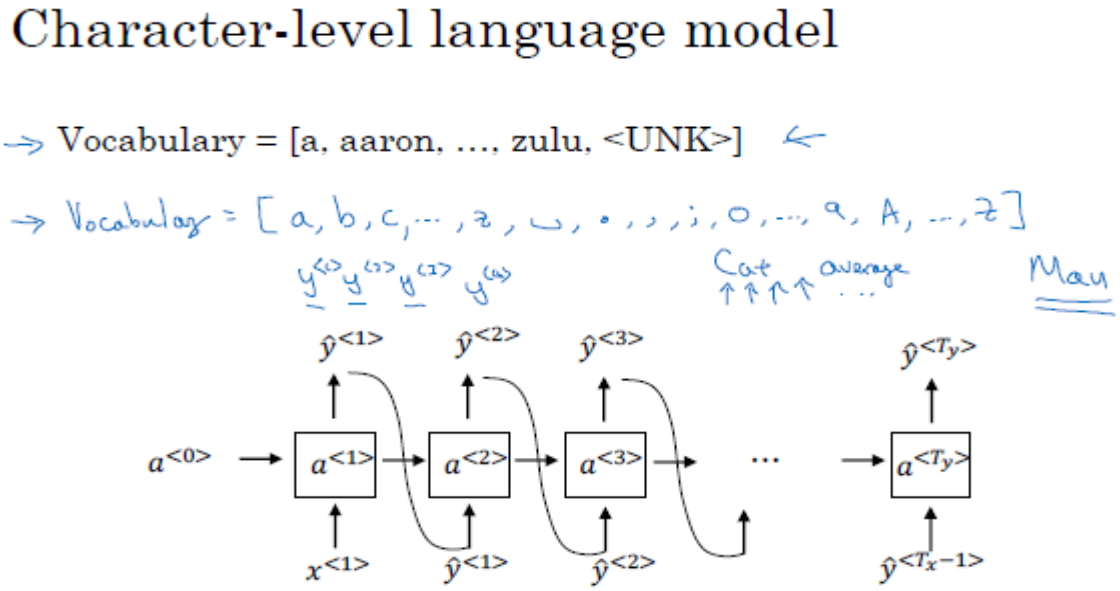
Andrew Ng

Sampling novel sequences

The sampling procedure is coming after training step. For the first time-step, we ask the Softmax to give us probabilities on all the words in the dictionary. Then we randomly choose one word based on the word distribution. Then we feed that chosen word to the next block and continue to do the same. We also have control over when to end by specifying an <EOS> token to the dictionary or maybe by specifying the length of the sentence. Moreover, if we do not like the <UNK> words, we can just resample again when the output was this token.



We can also make a character-level language model as well. The advantage of this model is that, there is always a probability for the words we've never met before and is not in our dictionary. The main disadvantage of this model is that we end up with dozens of characters for generating a sentence which is too much. And it also needs to find the effect of the earlier parts of the sentence on the later parts of it. So its computation is highly expensive.



The slide below is an example of this character-level language model for Shakespeare.

Sequence generation

News

President enrique peña nieto, announced sench's sulk former coming football langston paring.

"I was not at all surprised," said hich langston.

"Concussion epidemic", to be examined.

The gray football the told some and this has on the uefa icon, should money as.

Shakespeare

The mortal moon hath her eclipse in love.

And subject of this thou art another this fold.

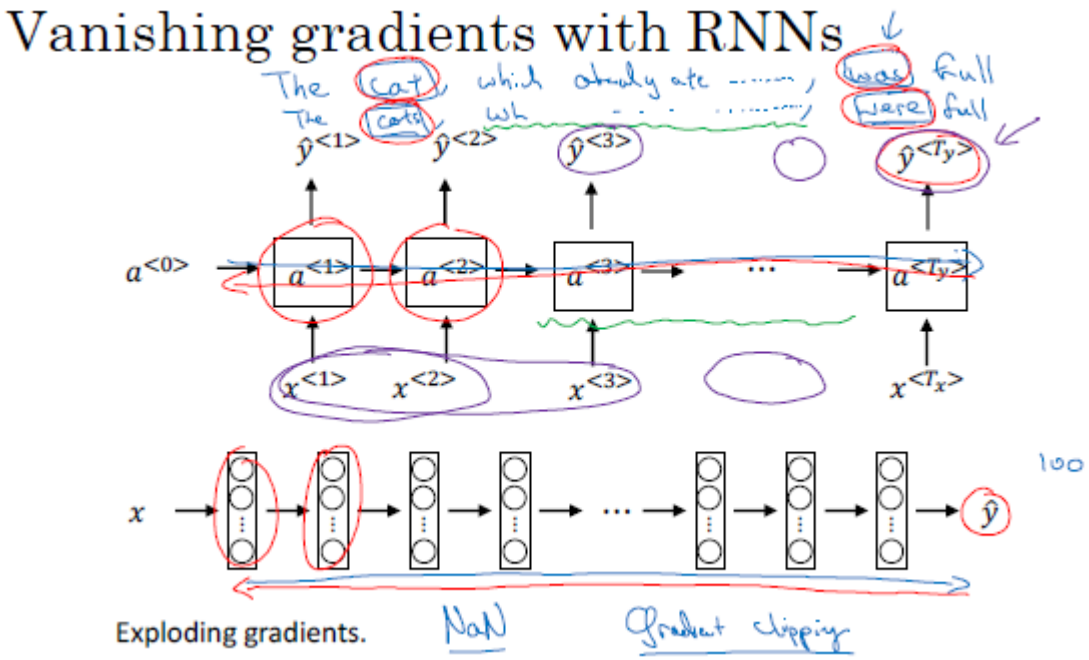
When besser be my love to me see sabl's.

For whose are ruse of mine eyes heaves.

Vanishing gradients with RNNs

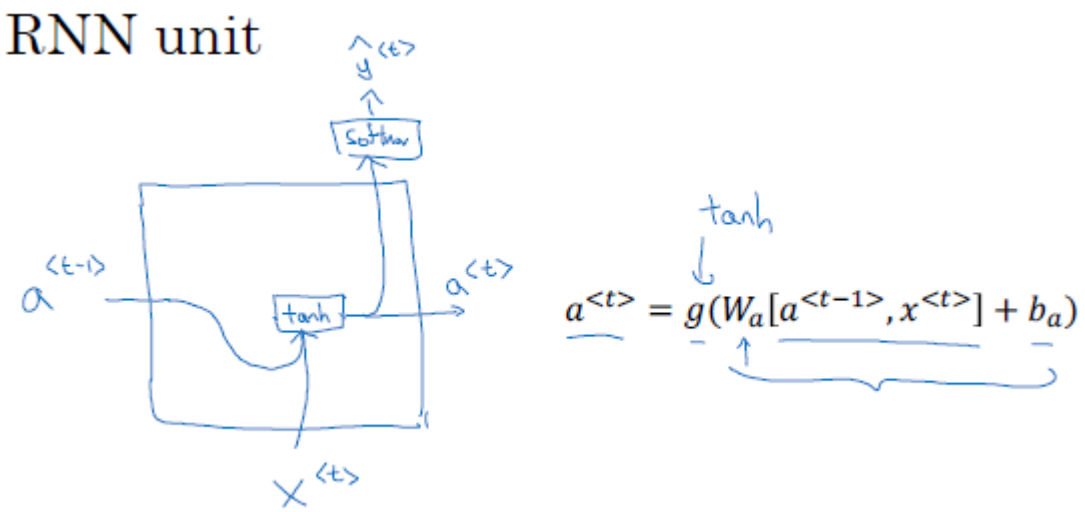
One of the problems of RNNs is vanishing gradients. There is an effect of the language which comes to scene when working with RNNs is that words may have long term dependencies like their singular or plural forms of them. As we talked about it earlier in NN, when number of layers increases, the backpropagation has a hard time to update the earlier layers. RNN has the similar problem in backpropagation in time. So it is hard for the RNN to remember that the noun came earlier was plural or singular, as in some cases it has to remember it for a long time. So the output is always much under the affect of the recent earlier blocks rather than the very first ones.

As in very deep NNs we discussed before, exploding gradients might also happens; however, the vanishing gradient is more likely to happen in the RNNs. Moreover, the exploding gradient is easier to spot as the weights will become NaNs. For this case we can do gradient clipping which when the value passes a threshold, we re-scale it so that it doesn't become very big. But vanishing gradient is so much harder to address.



Gated Recurrent Unit (GRU)

GRU is one of the biggest modifications on the RNNs which helps a lot when we have vanishing gradient problem. So for introducing GRU, first let's look at the RNN visualization.

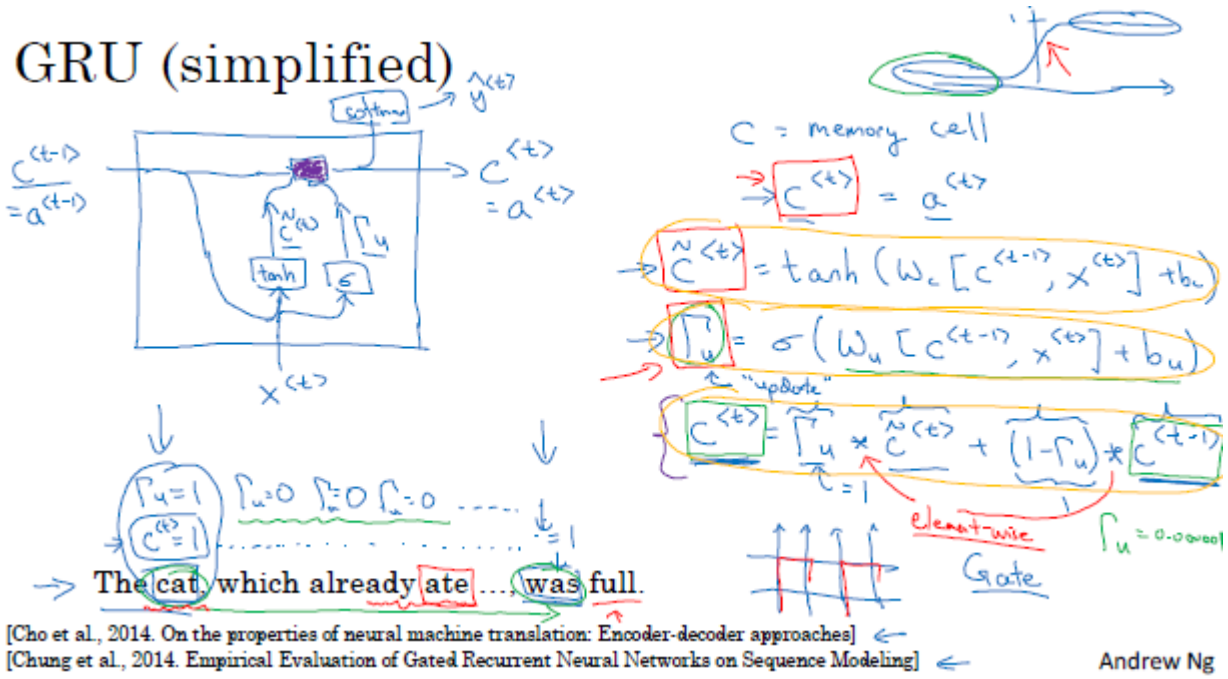


In the GRU, as we go from left to right, we have another variable, called "c" which comes from "memory **cell**". It provides a memory to remember whether the noun was singular or plural. In the RNN, the $c^{<t>} = a^{<t>}$ as the memory cell affects the activation function. However, later we will see that it is not the case for the LSTM architecture. So we use two different symbols for them. We also come up with $\tilde{c}^{<t>}$ as a candidate for replacing $c^{<t>}$. The $\tilde{c}^{<t>}$ formula is the same as the $a^{<t>}$ with one difference which is using $c^{<t>}$ instead of $a^{<t>}$. A way to think about $c^{<t>}$ is that it is going to be 0 or 1 depending on whether a noun is plural or singular. Then when it memorizes the assigned value until it reaches the verb of the sentence.

We also have an update gate as Γ_u, uses Sigmoid activation function and the function inside is the same as before. The job of this update gate is to update the $c^{<t>}$ for when it does not need to memorize the singular or plural status of the noun or subject of the sentence. When gate is 1 the $c^{<t>} = \tilde{c}^{<t>}$ otherwise it would be $c^{<t>} = c^{<t-1>}$.

The effect of the memory cell and the update gate in preventing vanishing gradient comes to scene when we notice a very small value for the update gate forces $c^{<t>} = c^{<t-1>}$.

The memory cells and update gates all are a vector of the same dimensions. So when we want to apply the update gate on the memory cells, we only perform an element-wise multiplication. In this way, we set some bits to 0 and some bits that we want to remember as 1.



So far we went through a simple version of GRU. Now for the full version of GRU we also have Γ_r which tells how relevant the $c^{<t-1>}$ is to its candidate $\tilde{c}^{<t>}$. The other version is LSTM which most commonly used.

Full GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

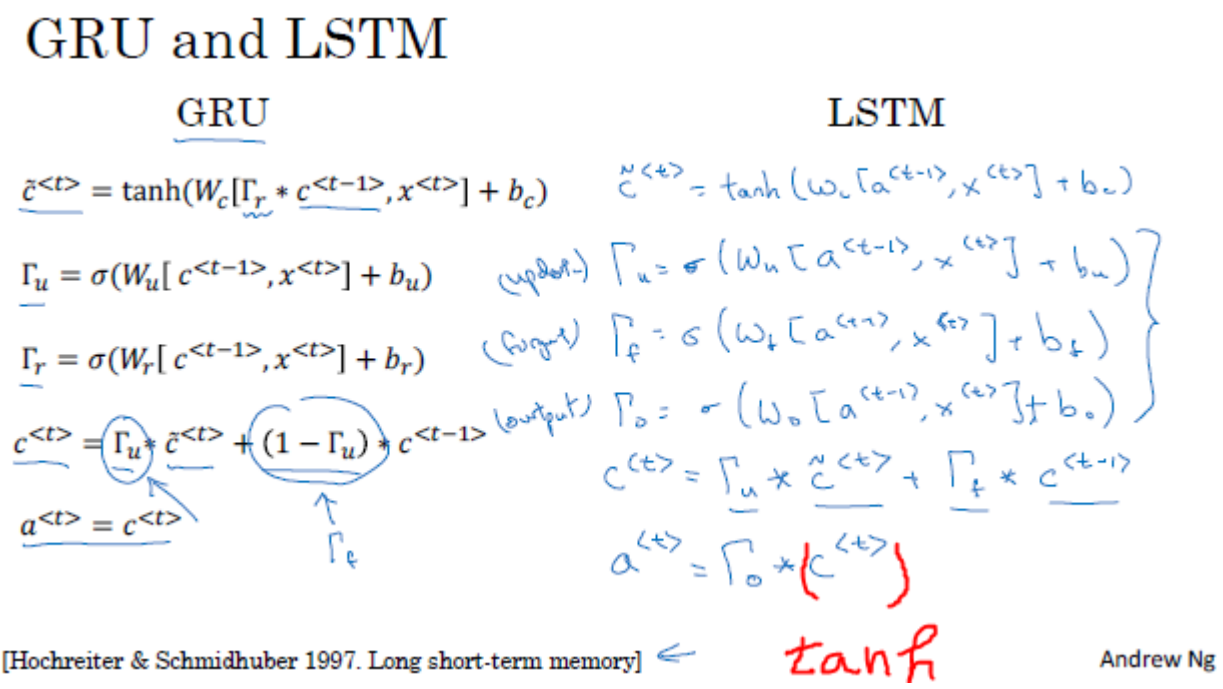
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

LSTM

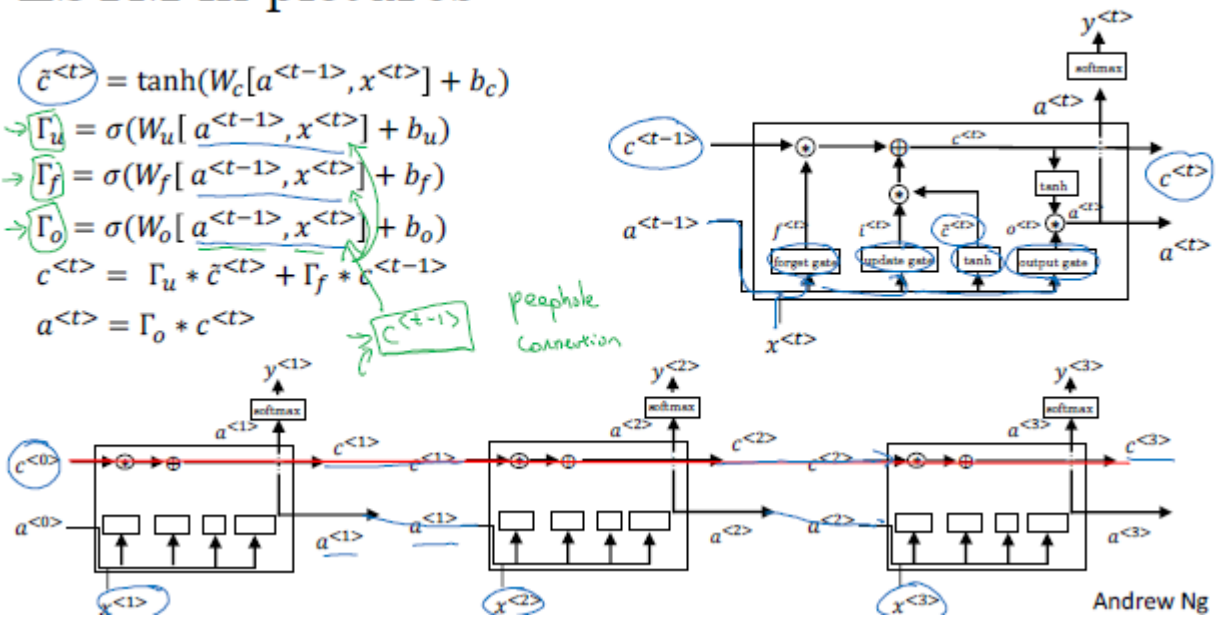
The cat, which ate already, was full.

Long Short Term Memory (LSTM)

The idea of in the LSTM is similar to the GRU but more complicated. First of all, the equation $c^{<t>} = a^{<t>}$ is no longer valid in LSTM. Second, we have a forget gate Γ_f and output gate Γ_o as well as update gate Γ_u we had before. The forget gate Γ_f replaces $(1 - \Gamma_u)$ in the memory cell equation. The reason for doing so is that the new forget gate gives the option of keeping previous value of memory cell and add it to the $\tilde{c}^{<t>}$.



In the slide below, we see multiple LSTM units are connected to each other. The cool thing about this picture is that we can see a straight line from $c^{<0>}$ to $c^{<3>}$ which means that it is so easy for the LSTM to memorize certain values for a very long time. There are also some variations of the LSTM. In one of them, they used "peephole connection" for when $c^{<t-1>}$ affects $x^{<t>}$.

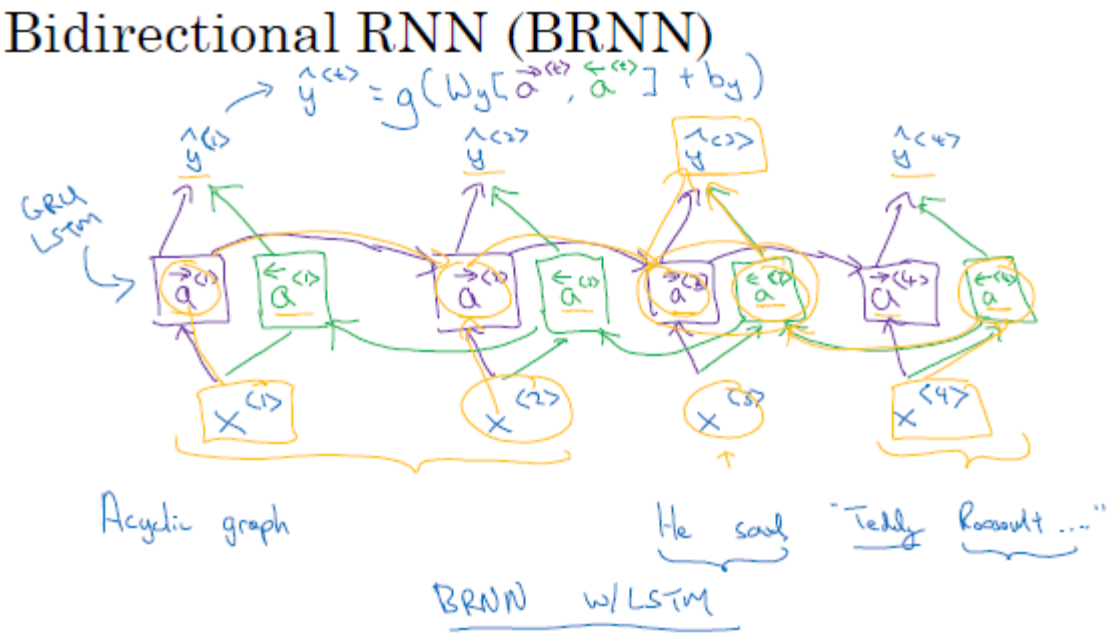


The advantage of GRU is that it is much simpler model and can be used to build a much bigger network since it only has 2 gates that makes the GRU faster and computationally less expensive. However, LSTM is more powerful and more effective and flexible as it is using 3 gates instead of 2.

Bidirectional RNN

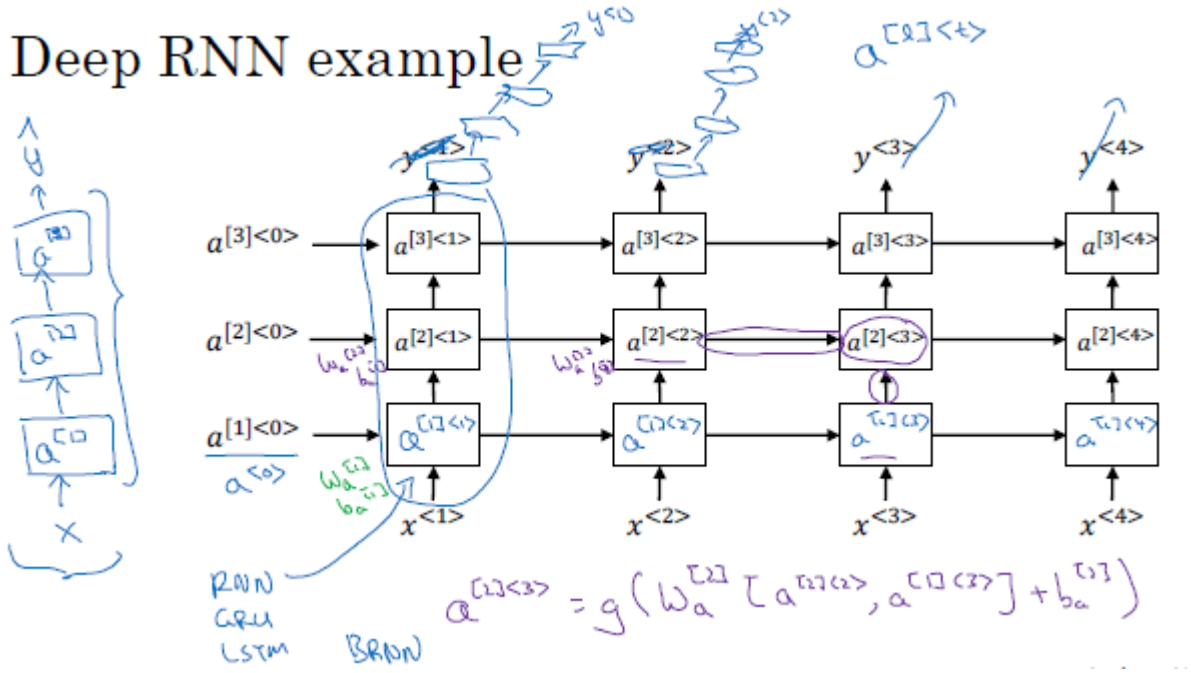
So we had a problem that when we saw the name “Teddy” we couldn’t understand it is for president or the bear. For using the later information to decode this problem we use BRNN.

In the BRNN, in the feedforward we go from the very first time-step to the very last one and we also go from the very last one to the very first one. It is like an acyclic graph. This approach is most commonly used in NLP tasks. One of the obvious problems of the BRNN is that it needs to have the full sequence of words to perform its job in for example predicting a word.



Deep RNNs

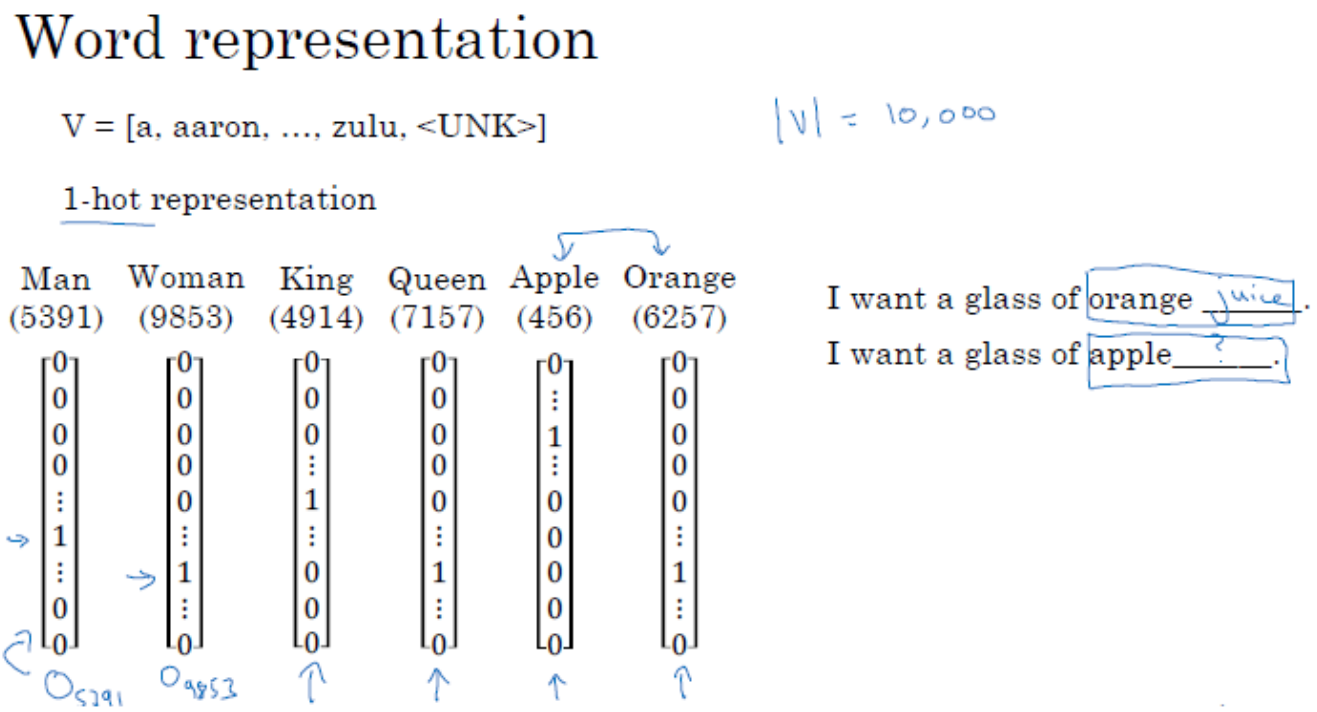
Let's see a Deep RNN example:



Week 2

Word Representation

In the word representation, we first use a dictionary and assign 1 to the word we are using and 0 to rest of them. For the dictionary size we can assume it is 10000 as before. The disadvantage of this model is that it is hard for the learning algorithm to generalize. For example, we have "a glass of orange *juice*" as well as "a glass of apple *juice*". But the model cannot find the relation between apple and orange in the sentence since it saw *orange juice* more than *apple juice*. The problem comes from the point that the inner product of each two pair in the dictionary is 0 so it cannot generalize the relations.



For generalizing better across the words, we use some features. These features gives some weights to each word. So by having these features, the model can better generalize the types of word it encounters and the inner product wouldn't be 0 anymore. So each word has some weights for each feature. For the same example now, we see *apple* and *orange* are so similar in case of features. However, we cannot always come up with some set of features for each specific word we see. Consequently, this approach is not always going to work.

Featurized representation: word embedding

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97
⋮	⋮	⋮				
size						
cost						
alive						
verb						

300

e_{5391} e_{9853}

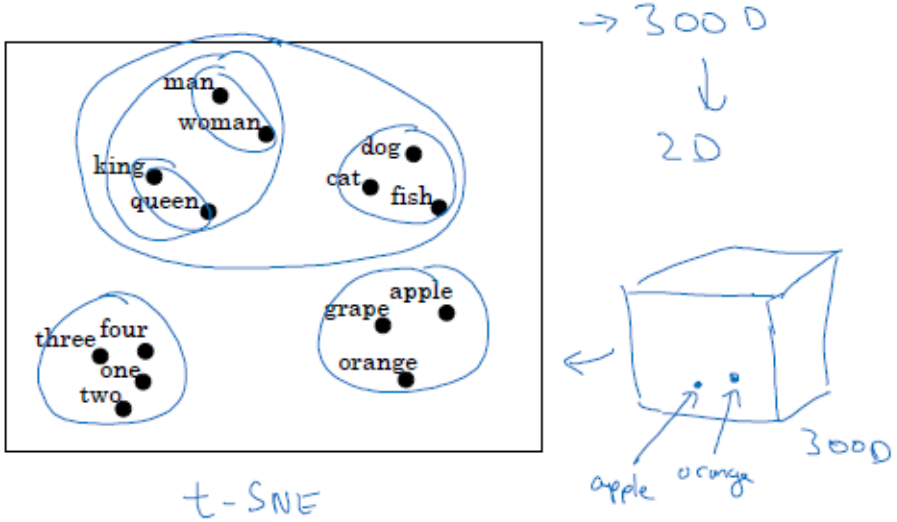
I want a glass of orange juice.

I want a glass of apple juice.

Andrew Ng

One nice thing we could do is to convert for example 300 dimensional features into 2 dimensional so that we can plot their similarity. The algorithm for doing so is t-SNE. For example we can see man/woman and king/queen or numbers group up together. Even human can be seen on only one side of the diagram. The "embedding" is used because each word is getting mapped in one point in the 300 dimensional space.

Visualizing word embeddings



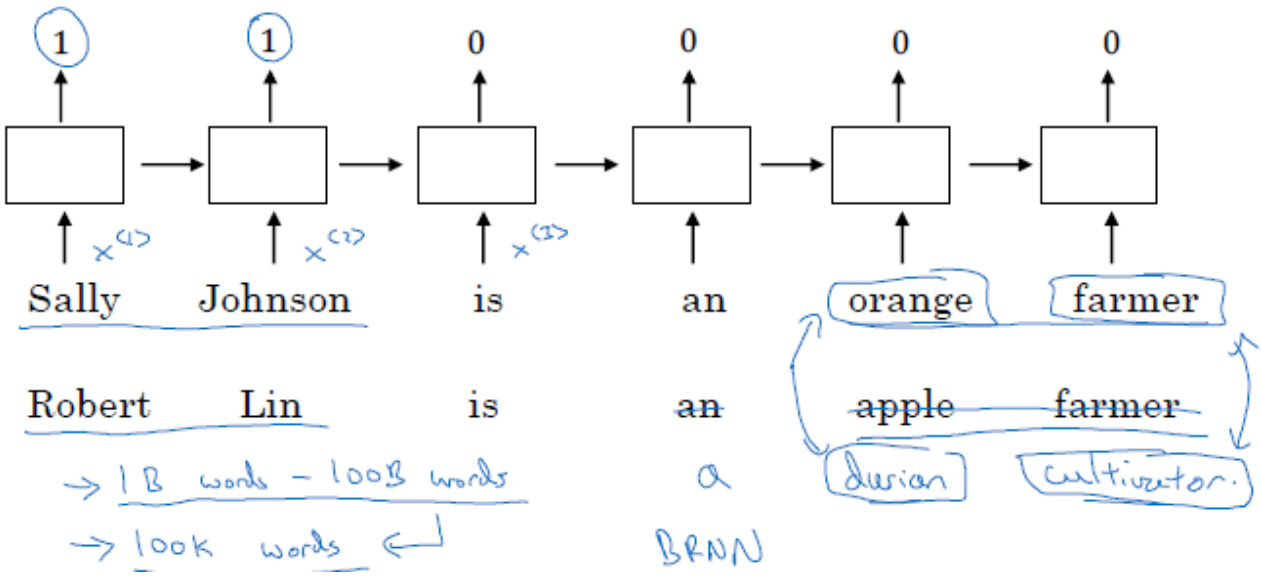
[van der Maaten and Hinton., 2008. Visualizing data using t-SNE]

Andrew Ng

Using word embeddings

So having word embedding, now we can give it to an RNN model, and it can find the similarities between words. Now what if we never saw a word in our training set? In this case generalizing is not possible anymore. For this task, as we do not need labeled data, we can use many texts on the internet and learn their features. Then we can understand orange / durian and farmer /cultivator are kind of similar. Finally we can transfer that knowledge to a task such as named entity recognition.
Note: this unidirectional RNN is just for simplicity, for some task it might have been bidirectional.

Named entity recognition example



Note that in the slide below, if we have so much data from transfer embedding while very small dataset for training, then fine-tuning does not going to work very well and we can skip this step. Also, note that, this technique is much less useful for some NLP task such as language modeling or machine translation.

Transfer learning and word embeddings

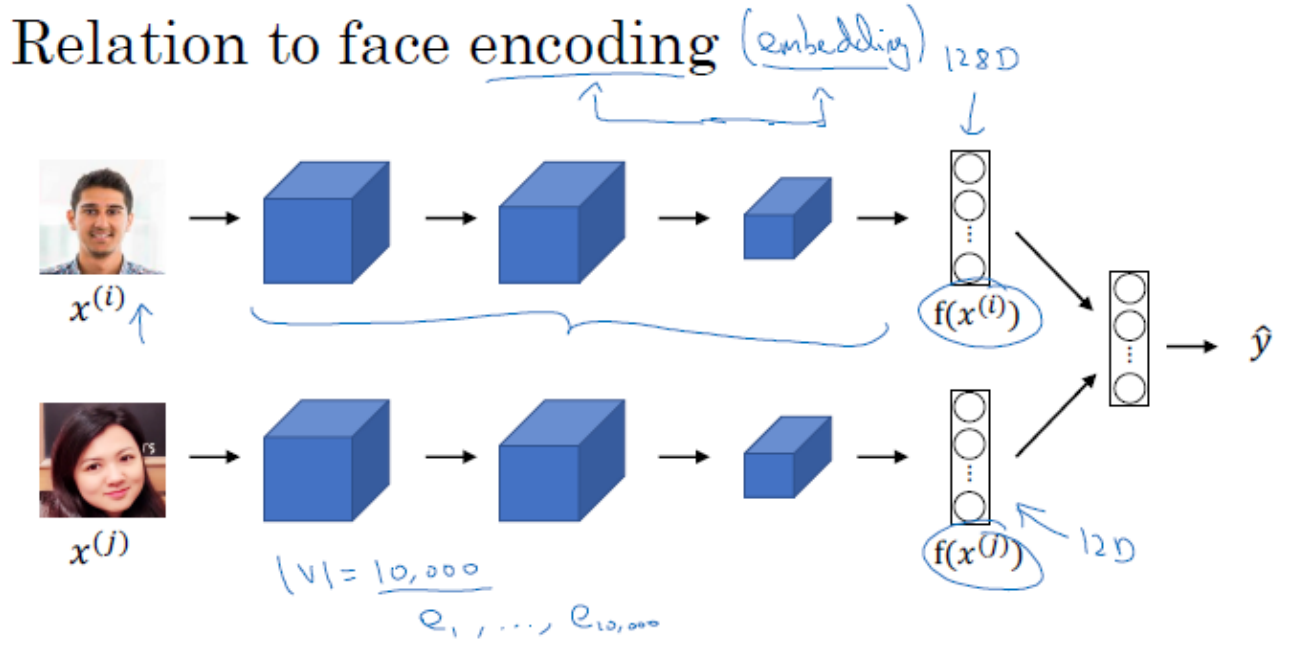
A
1. Learn word embeddings from large text corpus. (1-100B words)

   (Or download pre-trained embedding online.)

B
2. Transfer embedding to new task with smaller training set. (say, 100k words) → 10,000 → 300

3. Optional: Continue to finetune the word embeddings with new data.

In the word embedding we have a fixed size dictionary (say with 10,000 words) and we learn fixed embedding for each word. While in picture encoding (embedding) we want to learn, how to encode a picture that will change.

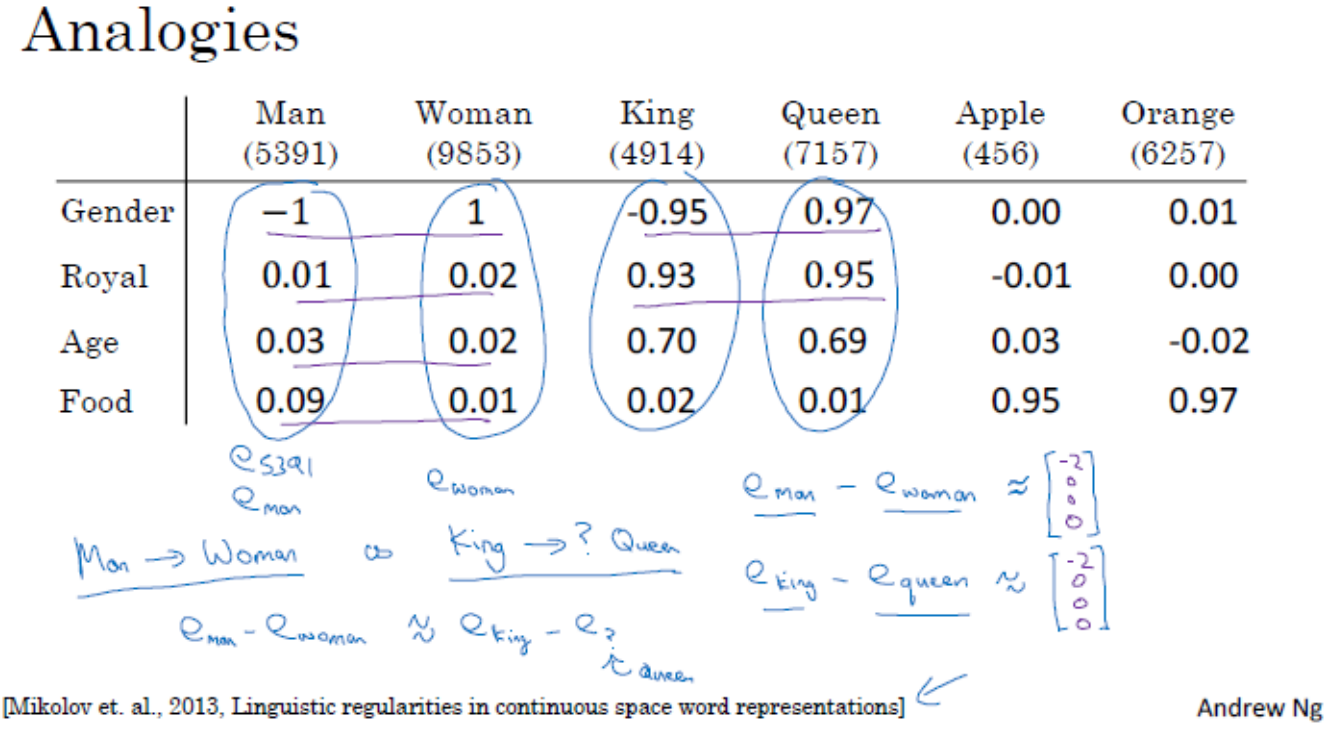


[Taigman et. al., 2014. DeepFace: Closing the gap to human level performance]

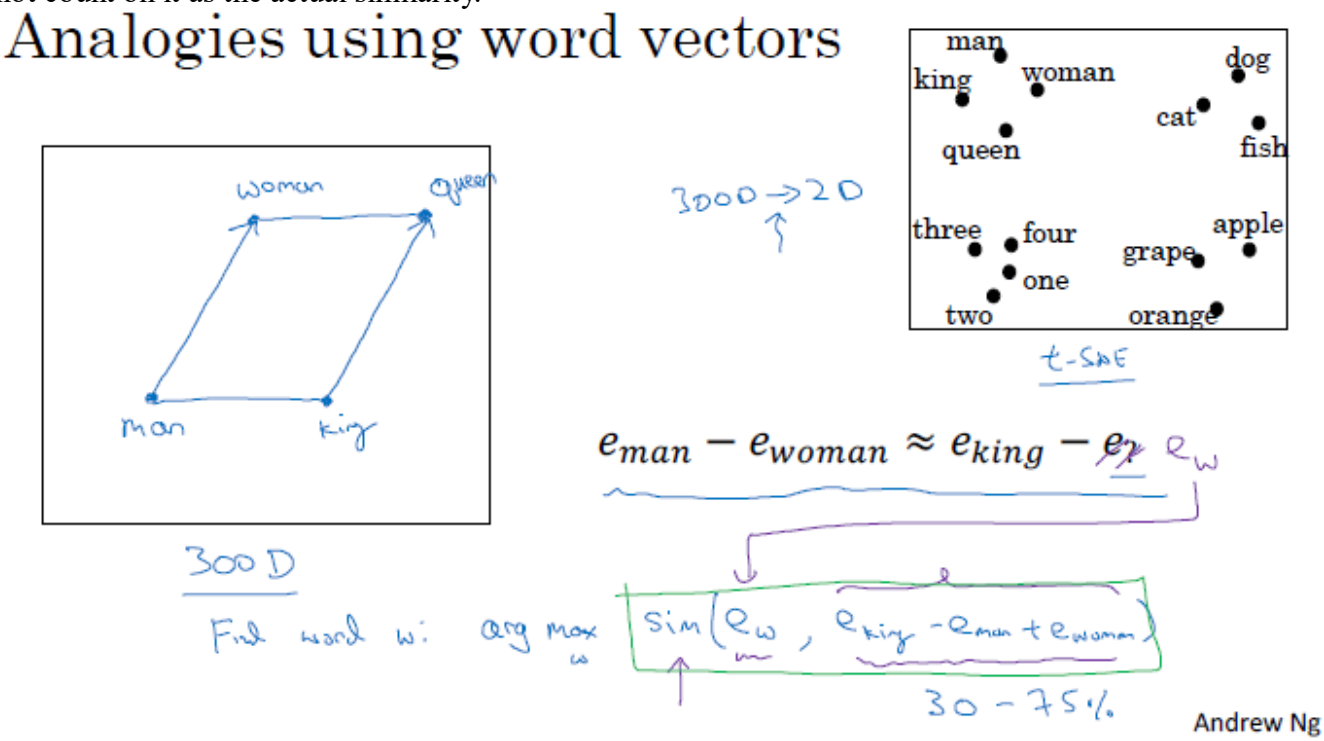
Andrew Ng

Properties of word embeddings

One interesting property of word embedding using feature is that the difference of two analogies are very close in vector space.



The left figure of the below slide, is the same technique, called t-SNE, in which the 300D word vectors are converted to a 2D diagram. It is not a good practice to do since we better work with 300D, but it is useful for visualization. For similarity task cosine similarity is most widely used but sometimes the squared distance (Euclidian distance) is also been used. If we see some similarities in t-SNE, we should not count on it as the actual similarity.

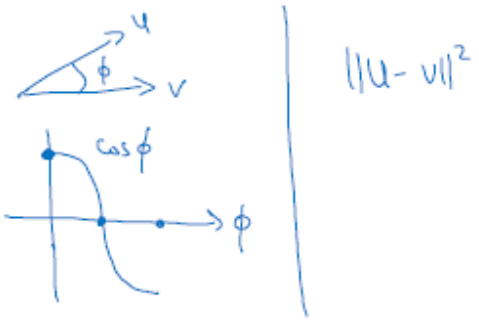


One way to compare two vector is to use cosine similarity which is the angle between two vectors in space.
One of the interesting features of word embedding is its generalization feature.

Cosine similarity

$$sim(e_w, e_{king} - e_{man} + e_{woman})$$

$$Sim(u, v) = \frac{u^T v}{\|u\|_2 \|v\|}$$



$$\|u - v\|^2$$

Man:Woman as Boy:Girl

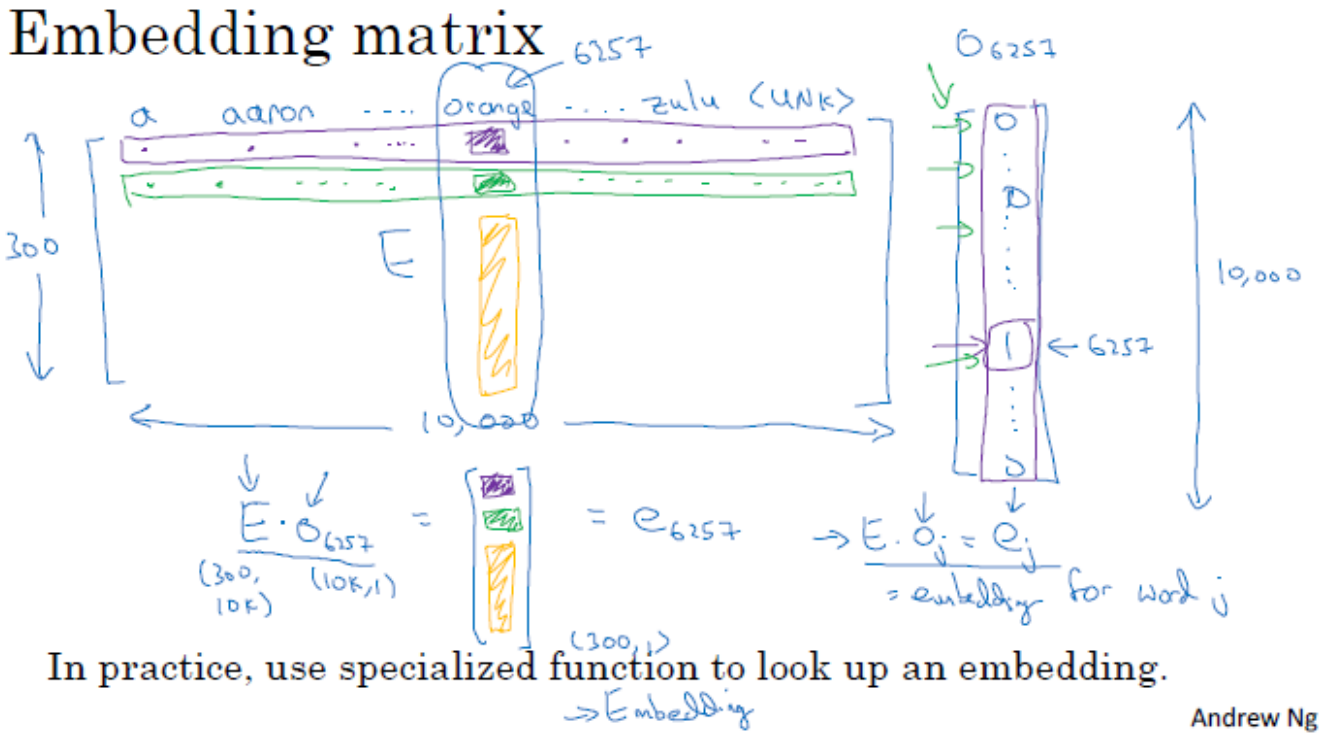
Ottawa:Canada as Nairobi:Kenya

Big:Bigger as Tall:Taller

Yen:Japan as Ruble:Russia

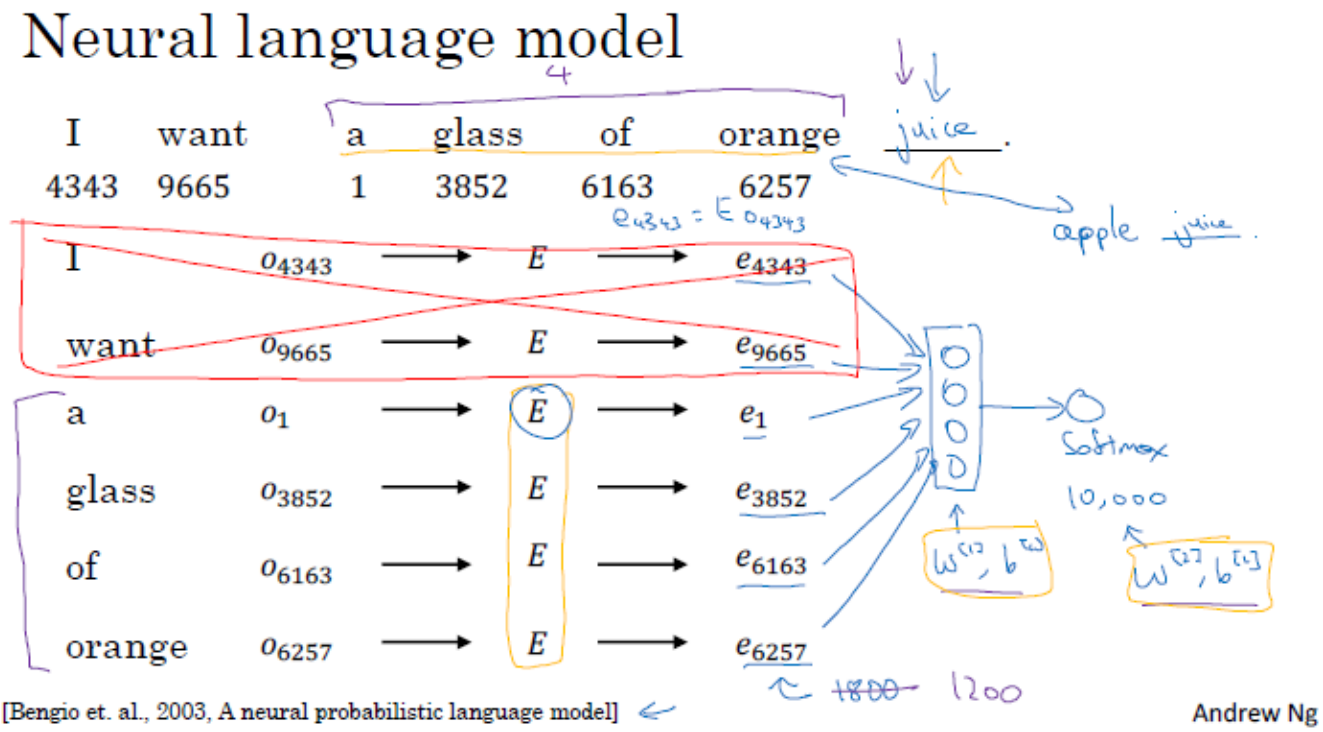
Embedding matrix

So in the word embeddings, actually what we learn is an embedding matrix. So multiplying an embedding matrix with a one-hot vector of a word we can extract all its features. The matrix multiplication is not efficient as there are many 0s in the word one-hot vector. So it is better to take out the word index and by that extract one column of the embedding matrix.



Learning word embeddings

In the neural language model, we take all the words embeddings, and stack them up all together. Then we will feed it into a Softmax activation function to choose a word out of possible words in the dictionary. What mostly been used is to a history window, a value which is a hyper-parameter, then to the same for predicting next word. This model easily learn the word similarities.



We can use it in other forms. For example we can use 4 words behind and after the word and use word embedding. But it is not much possible with 1 word. It is when we use a simpler model “skip gram”. So what researchers found was that if you really want to build a language model, it's natural to use the last few words as the context. But, if your main goal is really to learn a word embedding, then you can use all of these other contexts and they will result in very meaningful word embeddings as well.

Other context/target pairs

I want a glass of orange juice to go along with my cereal.

context target

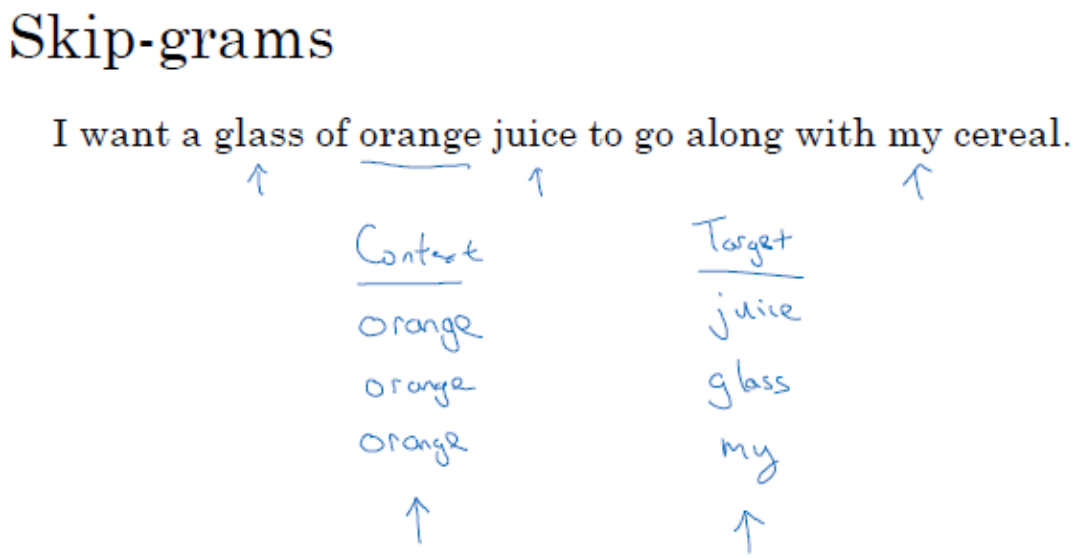
Context: Last 4 words.

- 4 words on left & right — a glass of orange ? to go alg with
- Last 1 word — orange ?
- Nearby 1 word — glass ?

skip gram

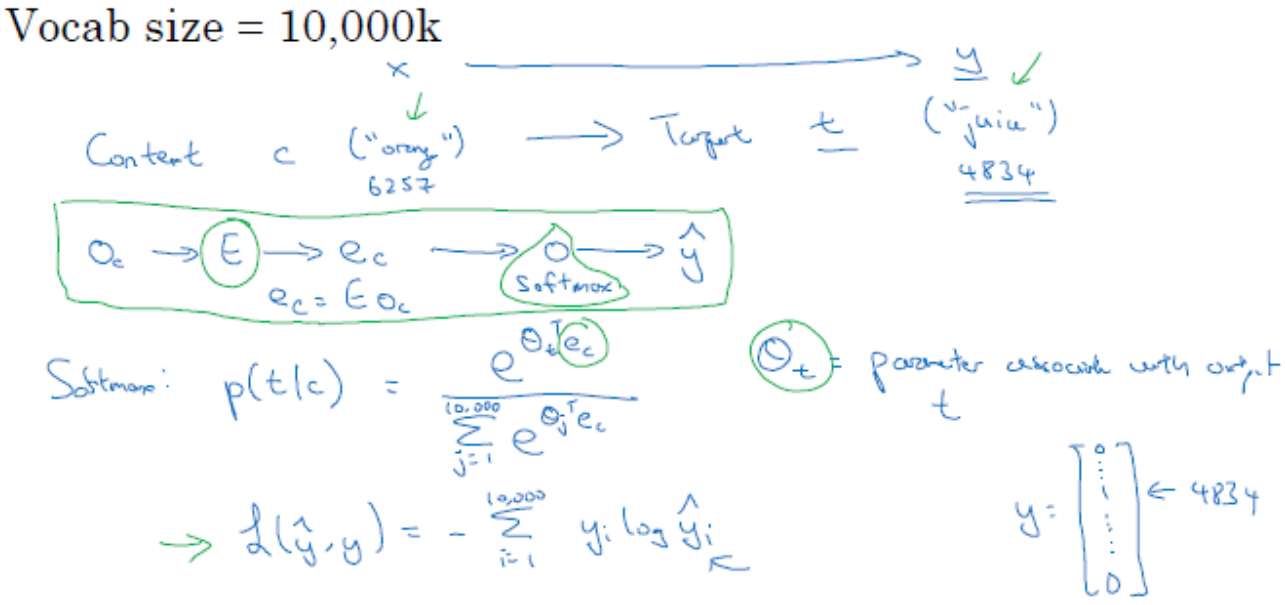
Word2Vec

In the skip-grams model we have context word and a target. So we randomly choose orange for example as the context. Then we use a word in a certain distance as target.



[Mikolov et. al., 2013. Efficient estimation of word representations in vector space.] Andrew Ng

So, what algorithm does is by taking a word as context, multiplying by embedding matrix, we get features associated with that word. Then by feeding it into a Softmax we try to predict probability of each word in the dictionary as target, given the context. Running this model using gradient descent turns out to be very good. The loss is the usual cross-entropy.



One of the problems of this model is computation. It needs to sum over all words in the dictionary which is at least 10000. Instead we can use a hierarchal Softmax tree in which for each layer, a word will be in one of the binary sub-trees. Then we go down until we find the word. Now choosing one of the two subtrees is the same as sigmoid function. As we are going down we calculate the probability mass function. At the end it acts as a Softmax activation function. Of course, there are some heuristics to traverse the tree so much easier, like putting the more common words at the top. But in the worst case this algorithm has O(log(size_of_dic)) in contrast to the actual Softmax which is O(size_of_dic).

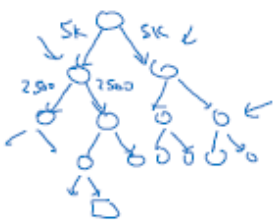
For sampling the context "c", we use some heuristics to sample some from common words and sample also from uncommon words otherwise randomly it most likely train on common words and after so much effort our model will learn nothing.

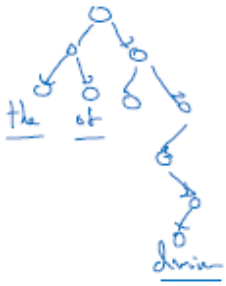
Problems with softmax classification

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

Hierarchical softmax.

$\log |V|$





How to sample the context c?

t

$c \rightarrow t$

→ the, of, a, and, to, ...

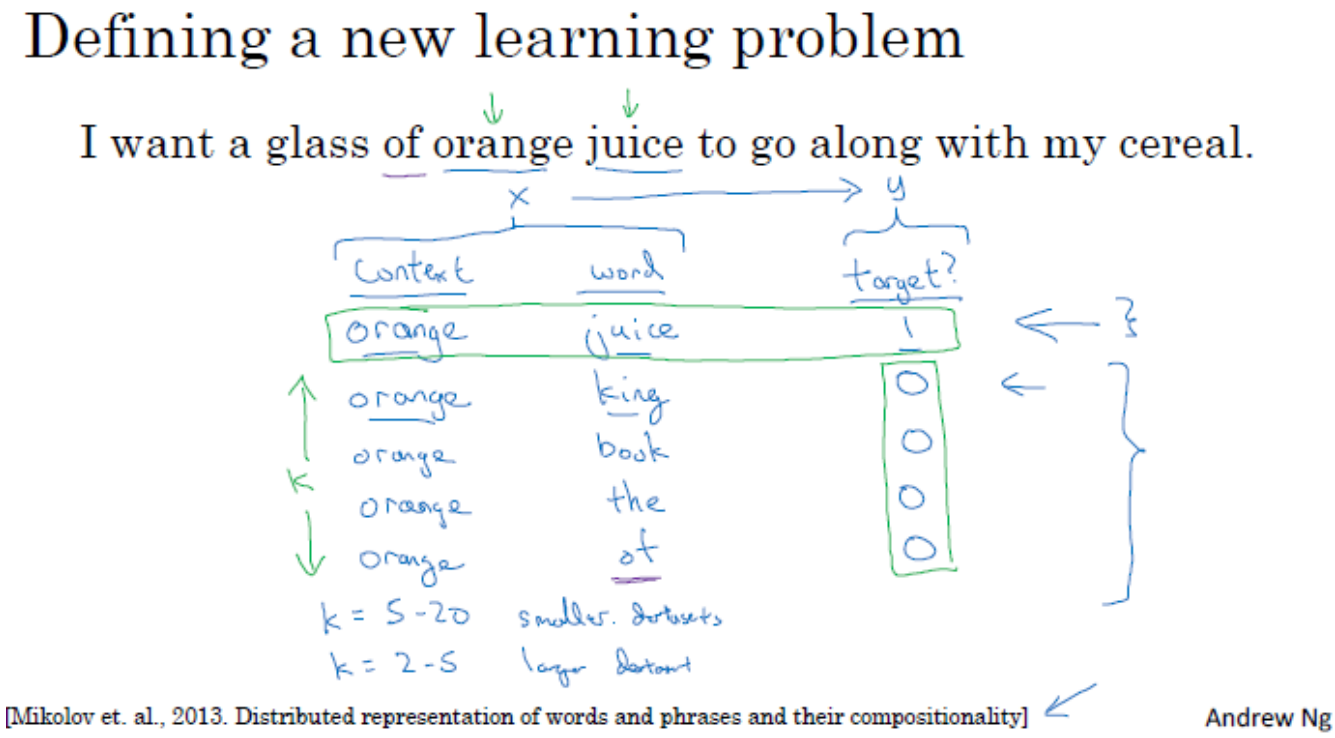
→ orange, apple, durian

$P(c)$

$\frac{3}{4}$ $P(c)$ $+$ $\frac{1}{4}$ durian

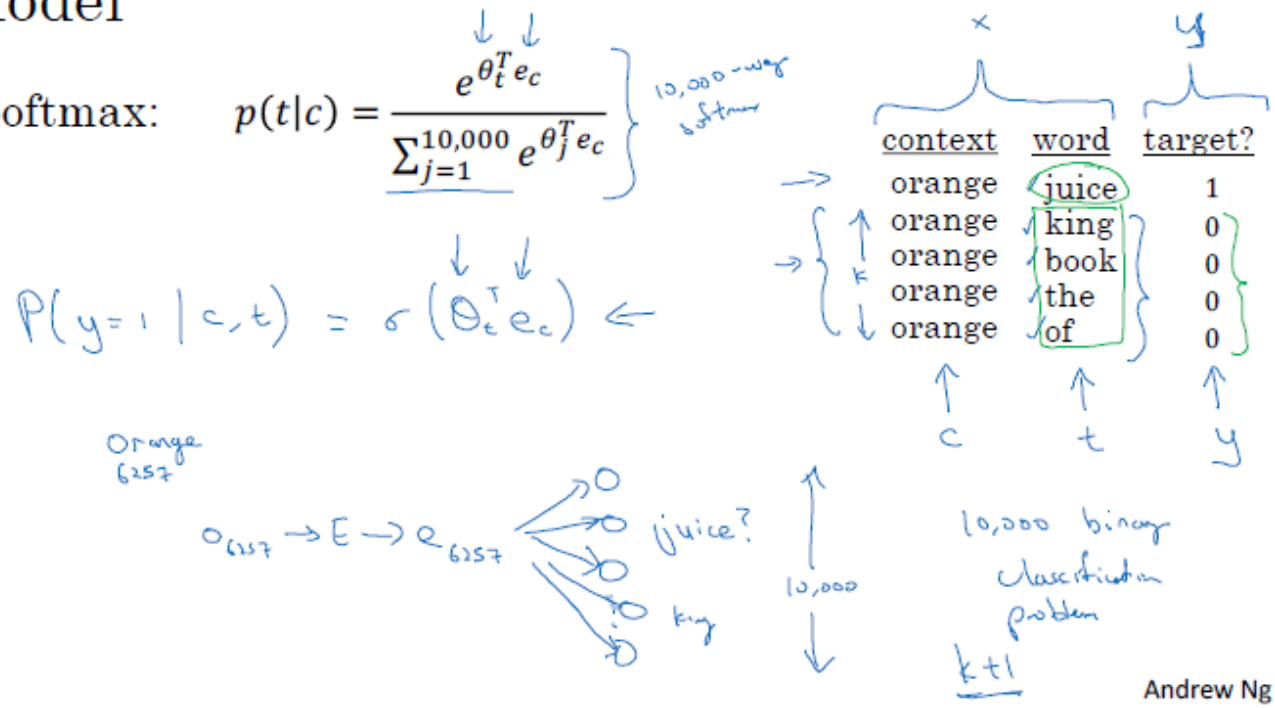
Negative Sampling

The problem about the skip-gram was Softmax was very slow as it sums up over all dictionary. This algorithm tries to solve this problem. So we restate the problem. If we have a context word and word was as it indented to be, then, target would be either 1 or 0. The words we choose for this task are random words from dictionary which might be in the sentence or might not. Number of words, "k", is a hyper-parameter and its size differ depend on size of dataset. For smaller datasets we set "k" as 5-20 as we do not have much content word to asses them, and for larger datasets "k" would be 2-5.



We restate the Softmax in this problem as probability of target value being 1 given the word and context. For each context word, we have "k" negative examples which means their target is 0. So the probability is the same as applying Sigmoid function. In this problem stating, we only going to train the true target word and "k" negative ones. So instead of having 10000 comparison which we had previously, we only have "k+1" samples. Now it is obvious why we call it negative sampling.

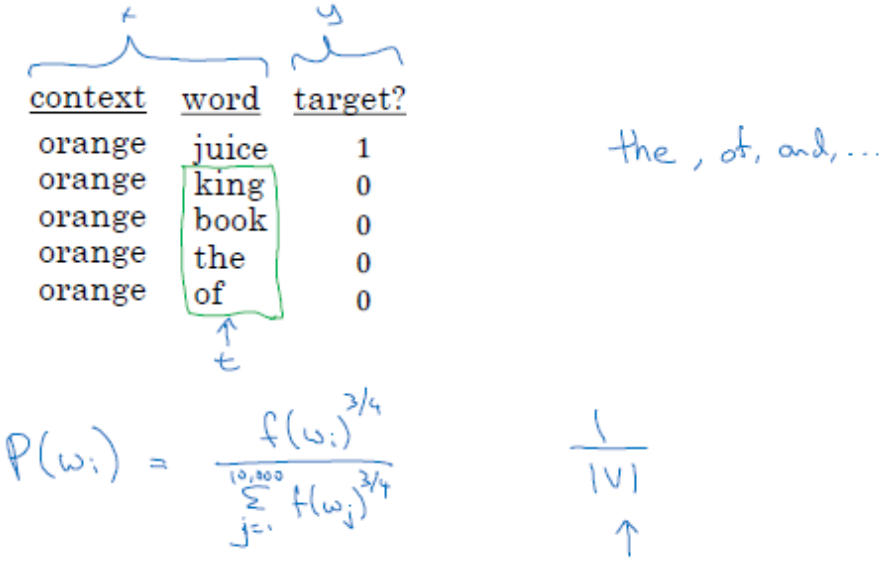
Model



Andrew Ng

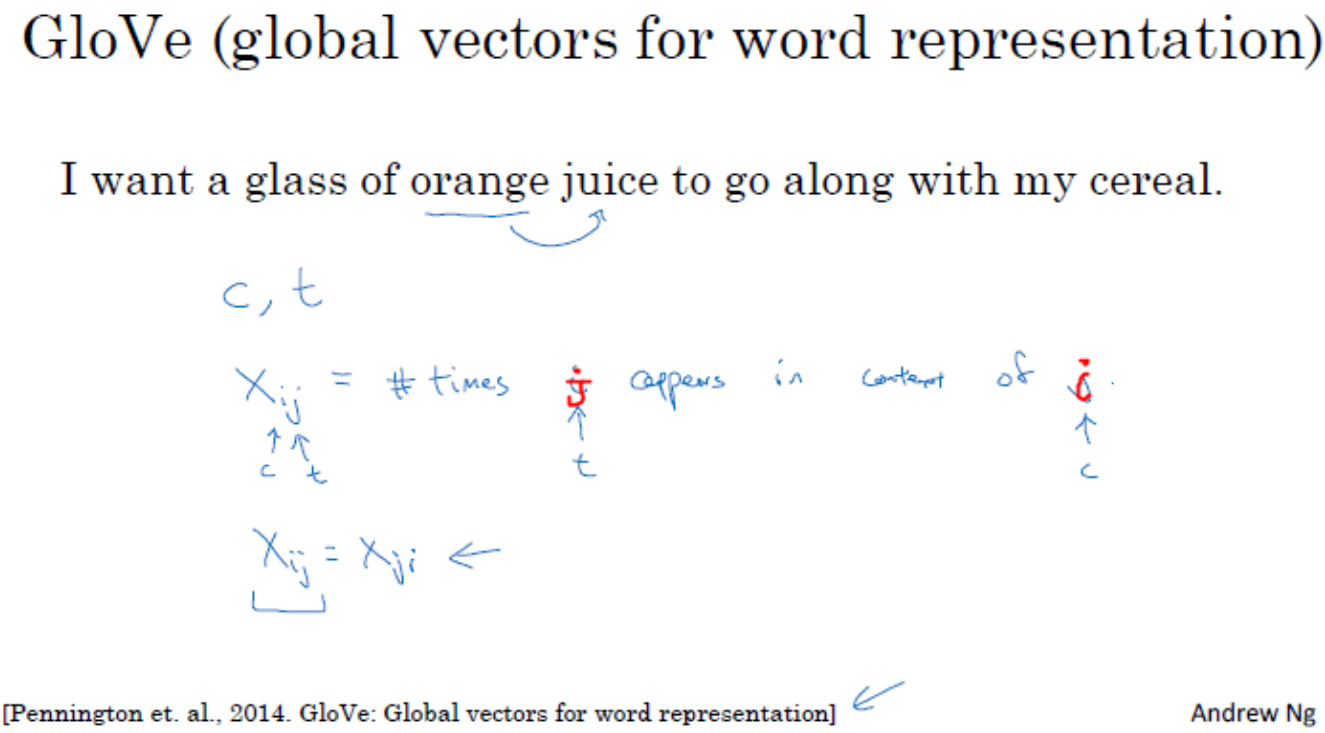
Now the question is, how to choose negative samples? We can sample based on their frequency of their appearance. But in this case, we encounter some break words, like the, of, and, … . The other approach would be choosing a word uniformly based on 1 over size of dictionary. Again the same problem happens. So the heuristic which worked the best. Its formula can be seen in the slide below. So this is in between of taking every observation and choosing randomly.

Selecting negative examples



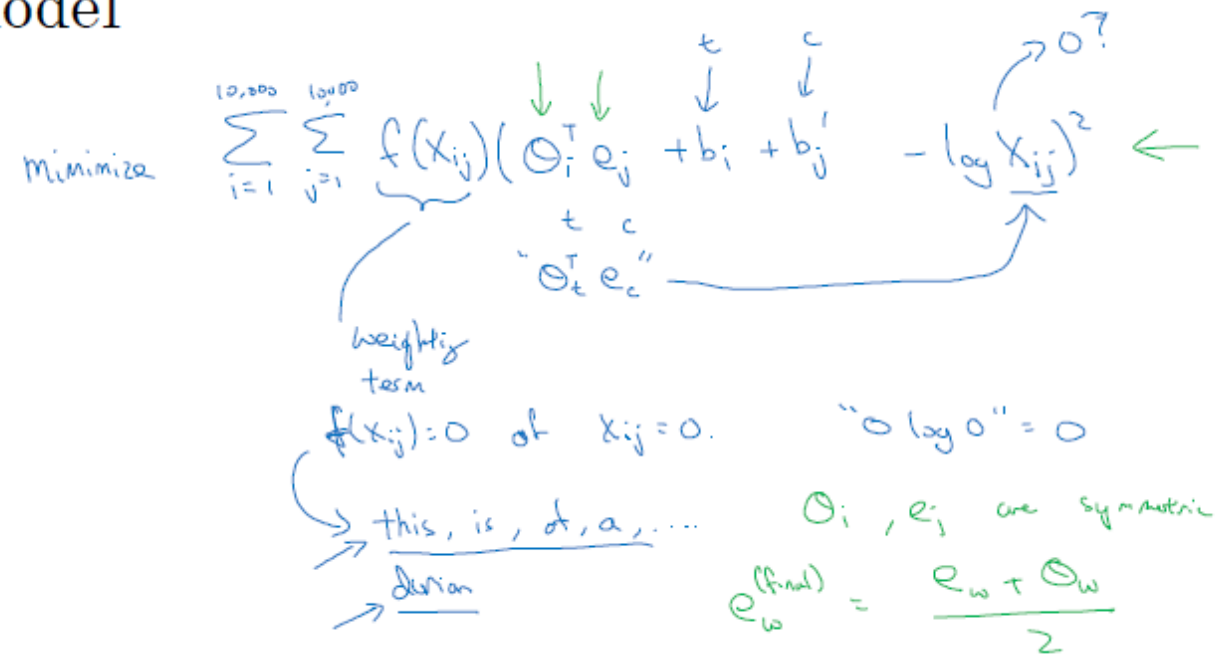
GloVe word vectors

Although this algorithm is not commonly used, but its simplicity gives a momentum to it. In this formulation, we count number of times a target word appears in the context of context word. In case of having a window we claim that the context the target word should be appear in must be in the window size back and forth of the word. In this case if we swap the context and target words, still the count would be the same. It is symmetric only in this case, but using something else than window, it is asymmetric.



The algorithm first tries to minimize the same weight and embedding matrix multiplication as we had before, by comparing its output with logarithm of word counting. As we may encounter a problem in which the counting is 0 and logarithm of 0 is not defined, we add a weight term in which outputs 0 for when the counting is 0. The weighting factor also balances the weights of stop words which are very common and very rare words.
Now, in this formulation and restated problem, we have symmetric weight and embedding matrix. To this end, we need to take their average which is their sum over 2.

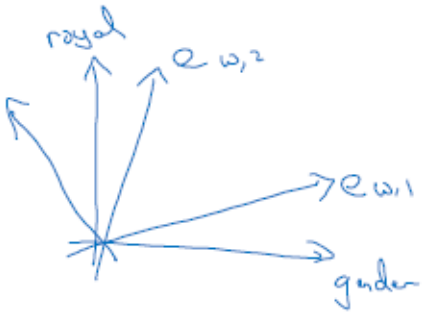
Model



One thing about this model is that we cannot guarantee that the individual components of the embeddings are interpretable. It is unlike what we started our journey with that was using features for words to find their relation. So each value for a feature can be combination of different features for each word. The reason is that we can define a matrix "A", in which it has a relation with the weight values and embedding matrix. When we expand the formula, "A" matrices cancel each other. But, existence of "A" matrix proves that some other variables are getting involved that takes the dependency between features and words away to some extent.

A note on the featurization view of word embeddings

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	
Gender	−1	1	-0.95	0.97	←
Royal	0.01	0.02	0.93	0.95	←
Age	0.03	0.02	0.70	0.69	←
Food	0.09	0.01	0.02	0.01	←



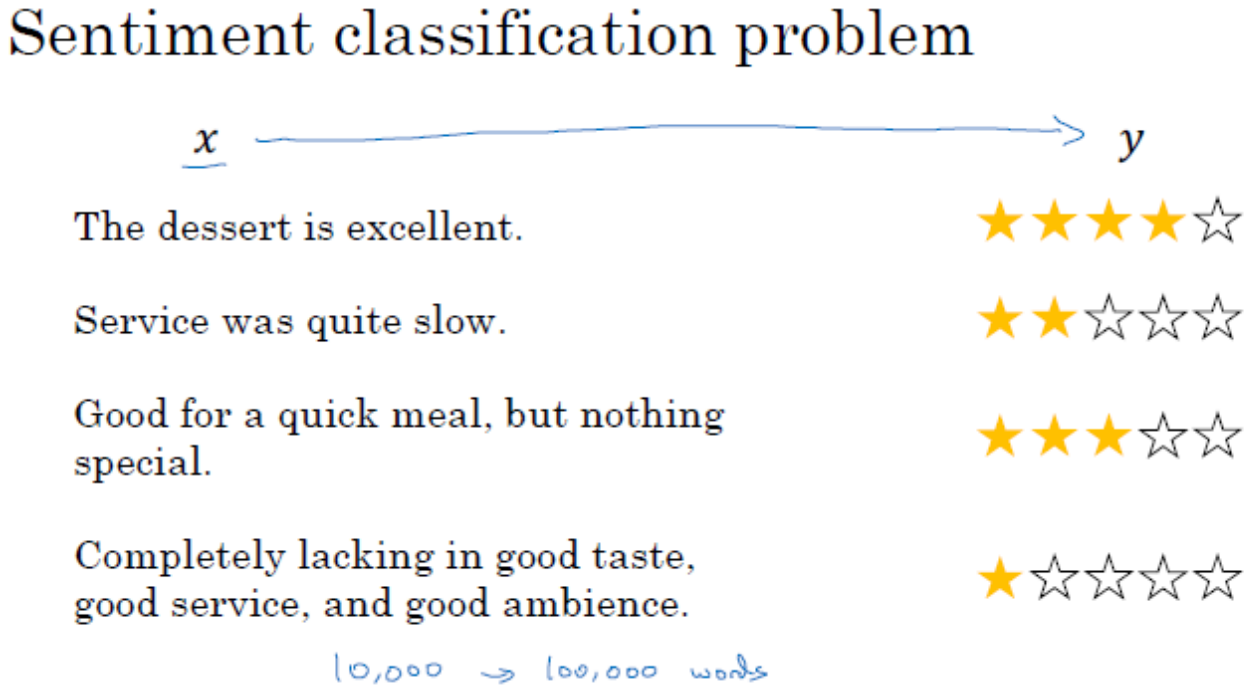
minimize $\sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$

$(A\theta_i)^T (A^{-T} e_j) = \theta_i^T A^T A^{-T} e_j$

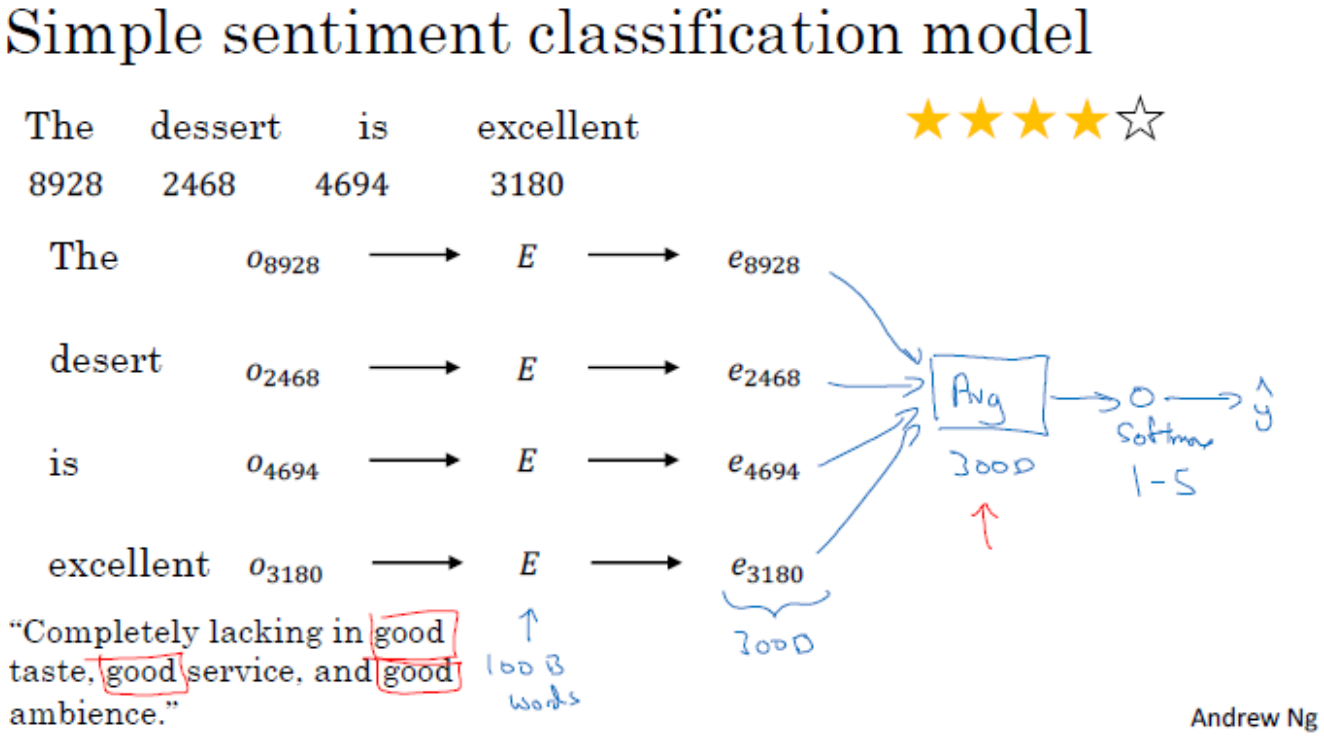
Andrew Ng

Sentiment Classification

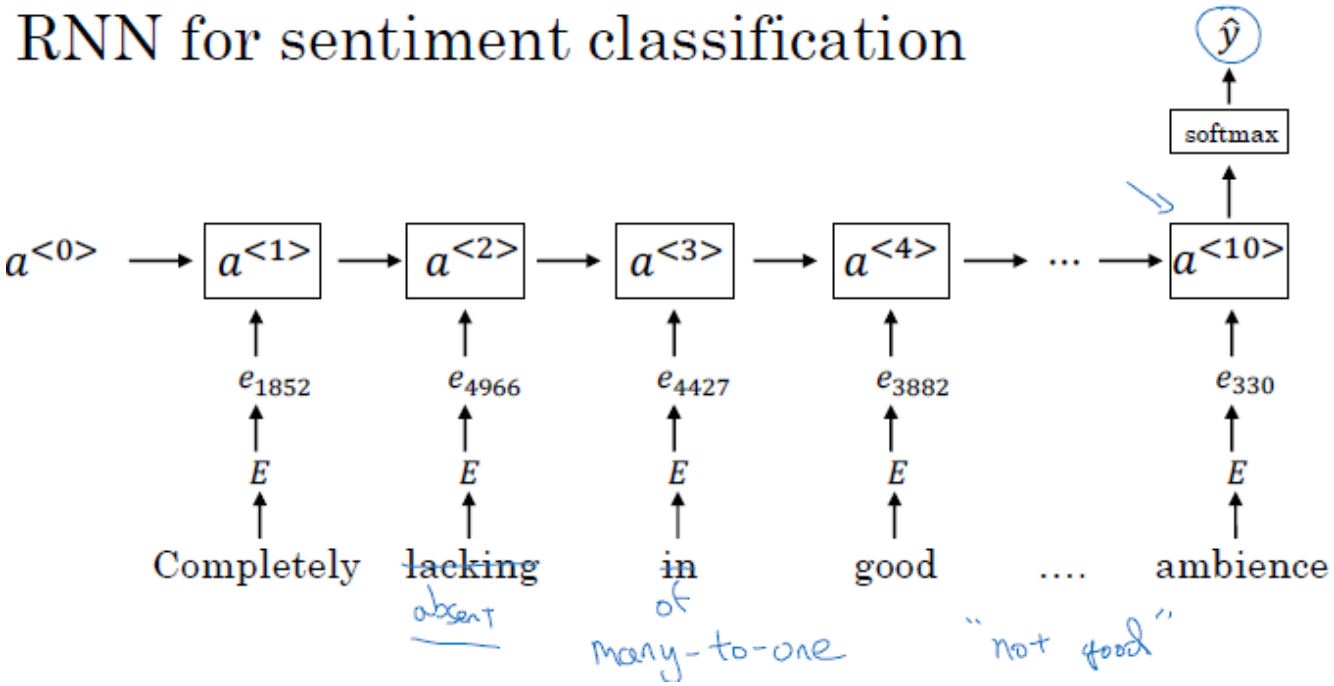
One of the challenges of sentiment classification is lack of enough training set. But with word embedding we can overcome this problem even with moderate dataset.



At very first, we start with the same approach of multiplying the word one-hot vector by the embedding matrix. So now we get much knowledge. Then we sum or average them and use Softmax to output a value between 1 and 5. This model works well. However, it has the disadvantage of losing the word order. For example, using one negative word like "lack" reverts all the positive words like "good". Hence, the problem is again about the order and RNN helps us with that.

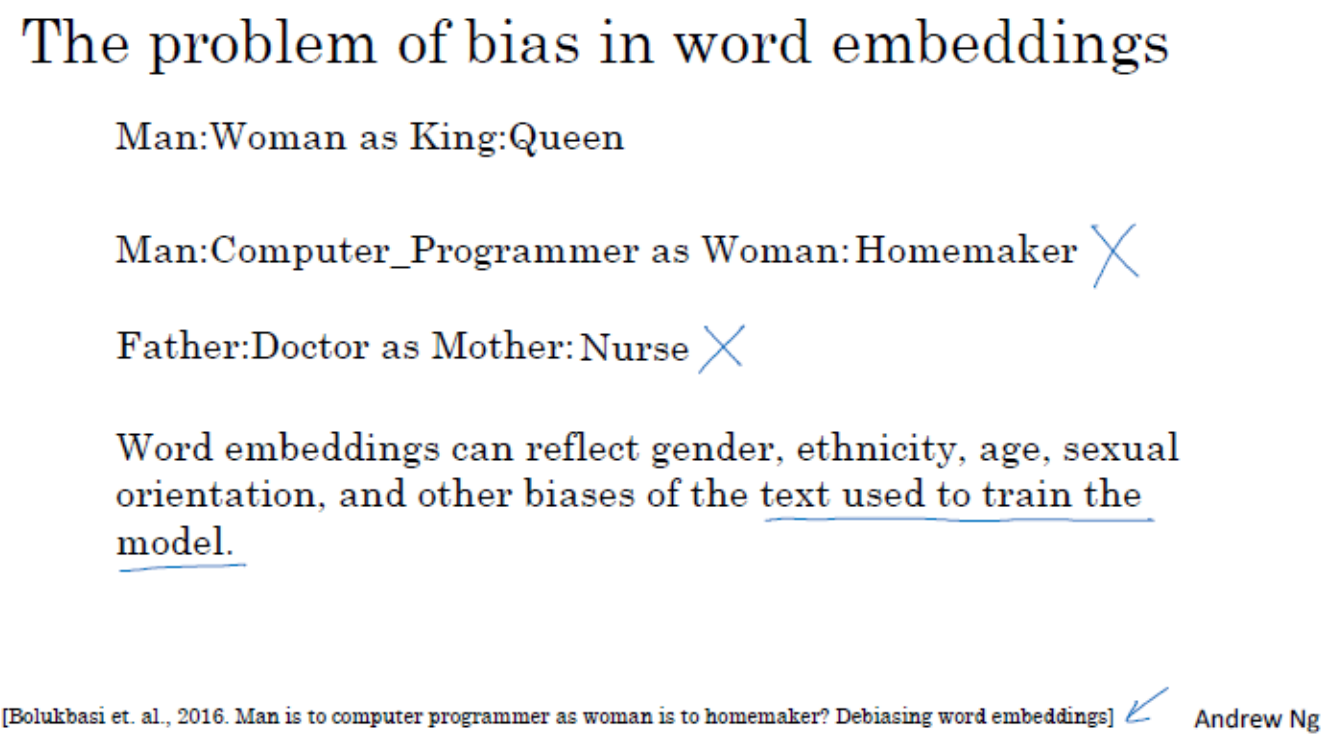


Now we can use a many-to-one RNN architecture. Even for the case of not seeing a word in the training set, the model can decently generalize.



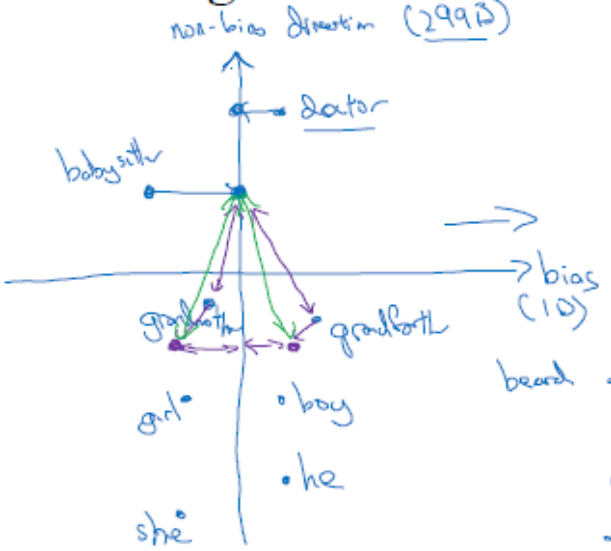
Debiasing word embeddings

In this context, we use the word bias not as bias and variance trade-off, but as gender or ethnicity bias. It may come up with horrifying outputs like man is programmer and woman is homemaker or man is doctor and woman is nurse.



For solving this problem, we can average over embeddings of biased words. In this example we talk about gender bias. This idea is similar to SVD. Next step is to project the neutral words so they are not related to specific gender anymore. Then we equalize the pairs. This means that we do not want the word babysitting be closer to grandmother than grandfather.
Mathematically, we project the neutral words into the main axis. Then make the distance of gender based words to the shared axis the same. To find the neutral words, as most of the words are neutral, we can use a linear classifier to find them.

Addressing bias in word embeddings



1. Identify bias direction.

$e_{he} - e_{she}$

$e_{male} - e_{female}$

average

2. Neutralize: For every word that is not definitional, project to get rid of bias.

3. Equalize pairs.

grandmother − grandfather

girl boy

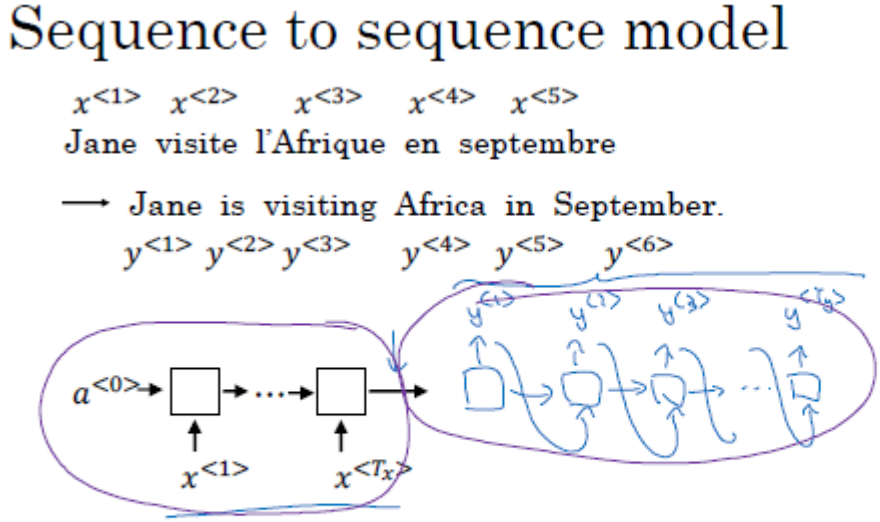
[Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings]

Andrew Ng

Week 3

Basic Models

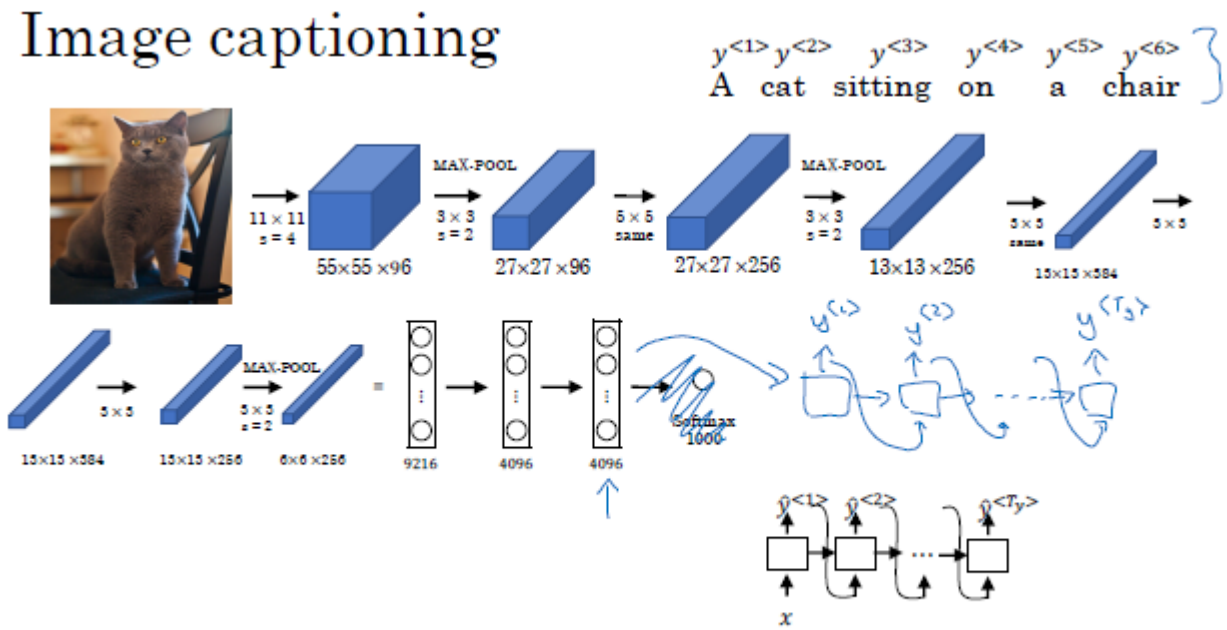
A basic model for translation is to use an RNN to encode the French sentence, then connecting it to an RNN to decode the translation while the sentence should be meaningful. This model actually works very well.



[Sutskever et al., 2014. Sequence to sequence learning with neural networks]
[Cho et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation]

Andrew Ng

It also works for image captioning. What we do is that we use an image CNN model like AlexNet and then instead of giving its output as a Softmax, we use at as input of an RNN.

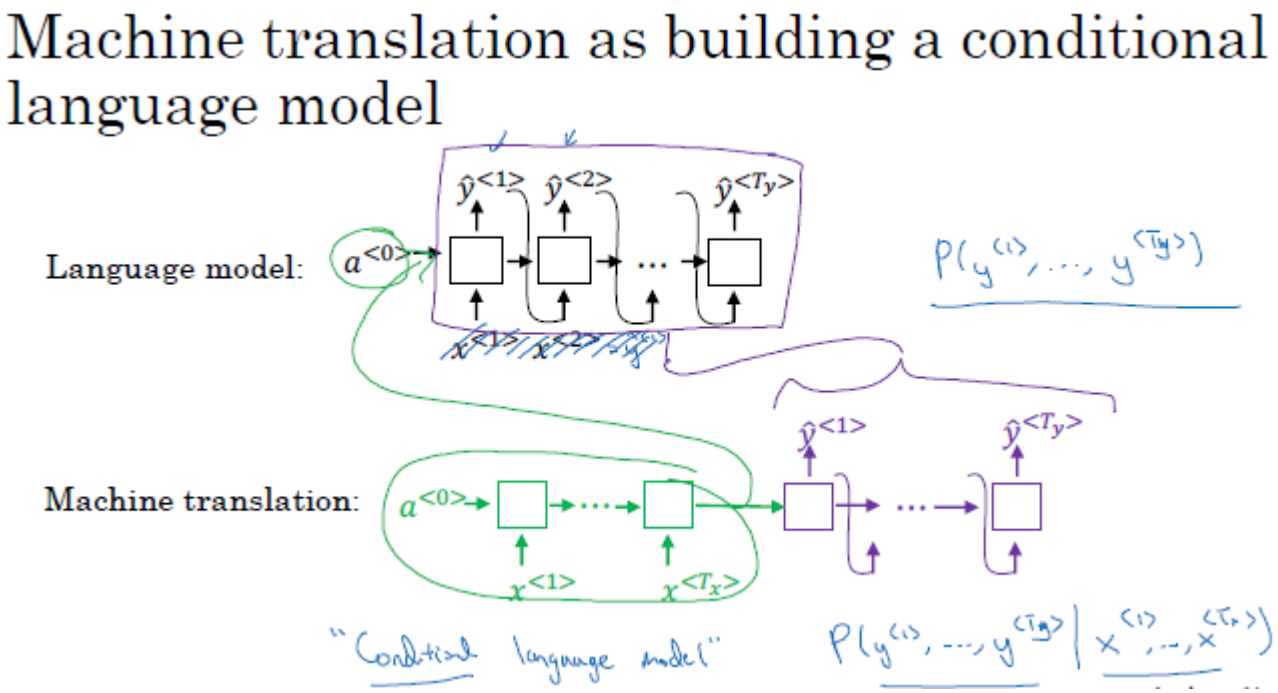


[Mao et. al., 2014. Deep captioning with multimodal recurrent neural networks]
[Vinyals et. al., 2014. Show and tell: Neural image caption generator]
[Karpathy and Li, 2015. Deep visual-semantic alignments for generating image descriptions]

Andrew Ng

Picking the most likely sentence

In the previous model, the caption or the translation could have been many similar sentences; however, we want the most likely correct and accurate sentence.
Language model has some differences with language model. In the language model we calculate the probability of having a specific output. However, in the machine translation model, we have two models combined, the encoder model and the decoder model. The decoder model is the same as the language model. Now instead of maximizing the probability of the sentence, we aim to maximize the probability of the sentence given the encoder model. To put it differently, we calculate the probability of an English sentence, given the French sentence. So it is a conditional language model.



In this model, instead of sampling randomly from the distribution, we want to maximize the conditional probability. The reason is that sampling randomly generates different translations for a sentence most of which are not good and acceptable for us as we do not use them in daily basis.

Finding the most likely translation

Jane visite l'Afrique en septembre.

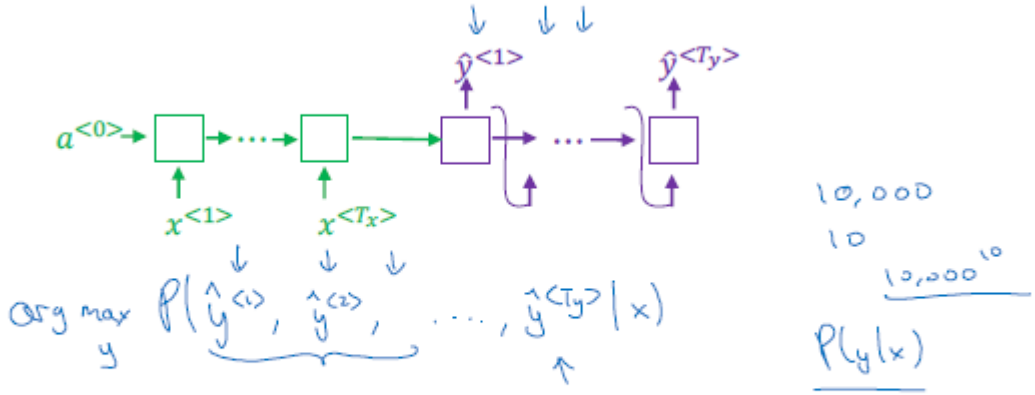
$P(y^{<1>}, \ldots, y^{<T_y>} | x)$

- Jane is visiting Africa in September.
- Jane is going to be visiting Africa in September.
- In September, Jane will visit Africa.
- Her African friend welcomed Jane in September.

$$\arg\max_{y^{<1>},\ldots,y^{<T_y>}} P(y^{<1>}, \ldots, y^{<T_y>} | x)$$

So what greedy search does is choosing the most likely word from the possibilities of output at each block. However, we want to maximize the whole conditional probaiblity and greedy search only maximizes each step in choosing a word. Consequently, in most of cases the greedy search does not end up maximizing the whole conditional probability. As the example in the slide, the second sentence is too wordy however the probaibity of "going" is higher than "visiting". We also cannot check every possible seqeunces as it cost is very high. So we use an approximate search algorithm. Although this one also may fail in maximizing the conditional probability.

Why not a greedy search?



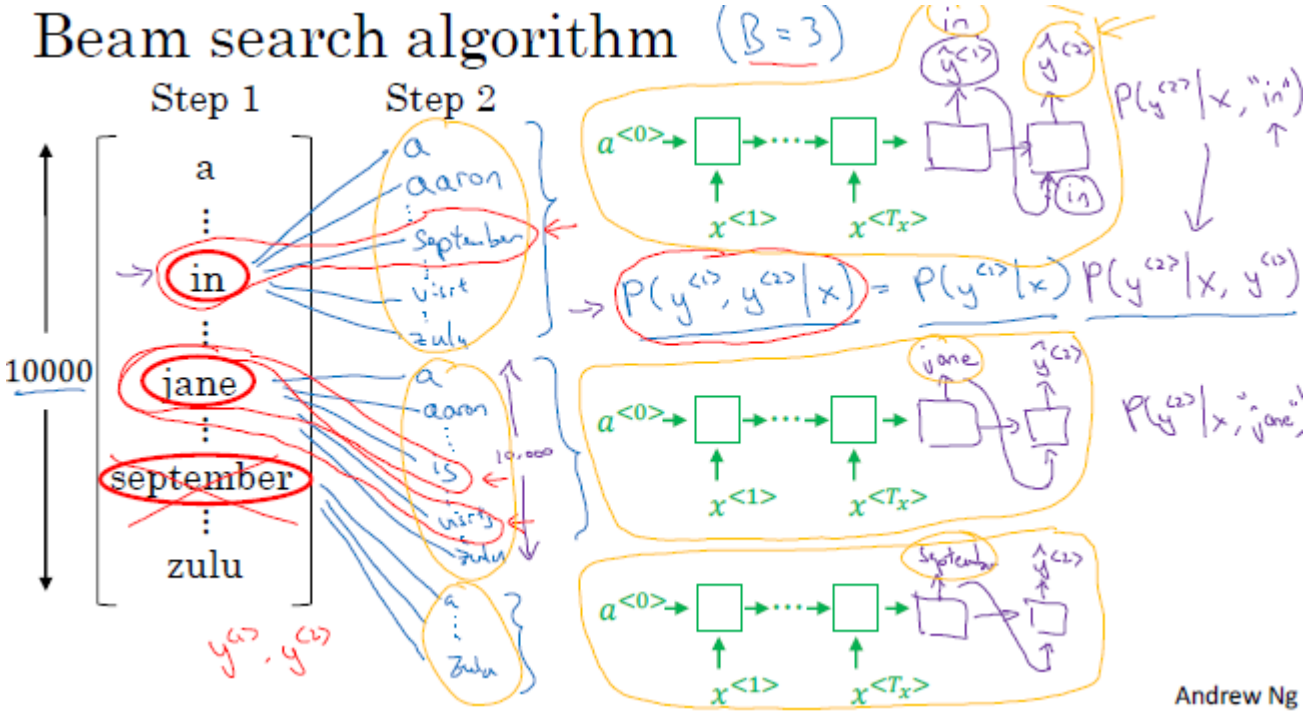
- Jane is visiting Africa in September.
- Jane is going to be visiting Africa in September.

$P(\text{Jane is going} | x) > P(\text{Jane is visiting} | x)$

Beam Search

So the approximate search we talked about is beam search. Beam search has a hyper-parameter, "B", which instead of taking the most probable word at each step, it takes "B" most probable words. So it tries all of them for the next steps. For example, the second step, it tries to maximize the probability of the output given the encoder and the first word. Its reason is almost obvious. We want to maximize the whole probability of sentence given encoder. As the each word is independent given the encoder, so we rewrite it as maximizing probability of first word given encoder multiplied by probability of second word given encoder and first word. As we know we maximized the probability of first word given encoder, we can only focus on maximizing probability of second word given encoder and first word only.

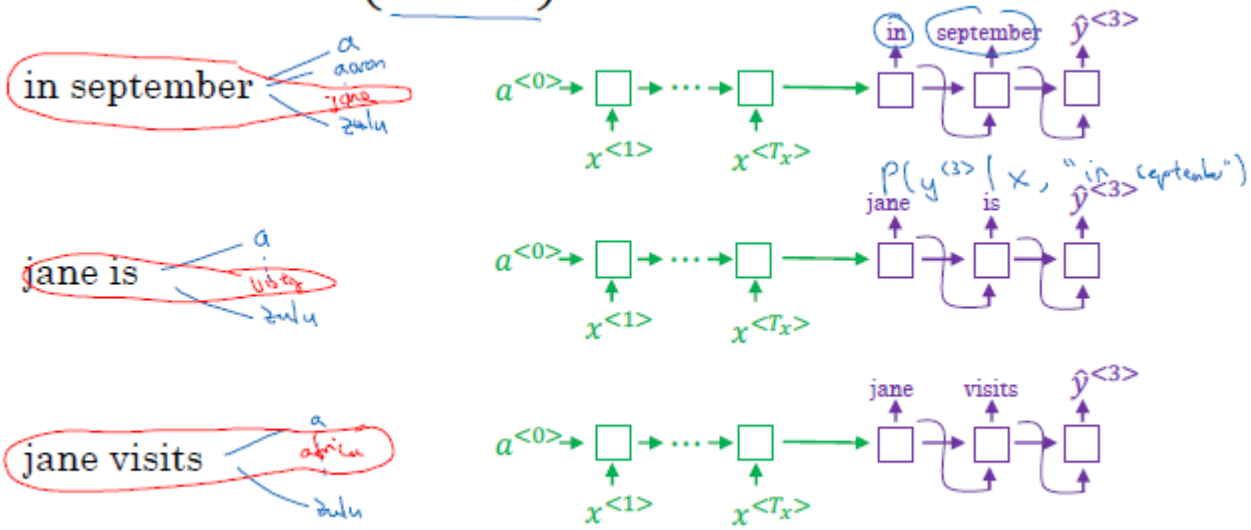
Now at each step, if we assume we have a dictionary with size of 10000, for the second step we choose again "B" word among $B \times 10000$ words. Now, the beam search may find the most probable words as "in September", "Jane is", and "Jane visits". So it rejects the September as the first word candidate, the word it found one of the most probable ones at the very first step. Note that we have "B" copies of the same model.



So we can see the next steps in the slide below. Note that B=1 it becomes the greedy search.

Beam search ($B = 3$)

$B = 1$ ⇝ greedy search

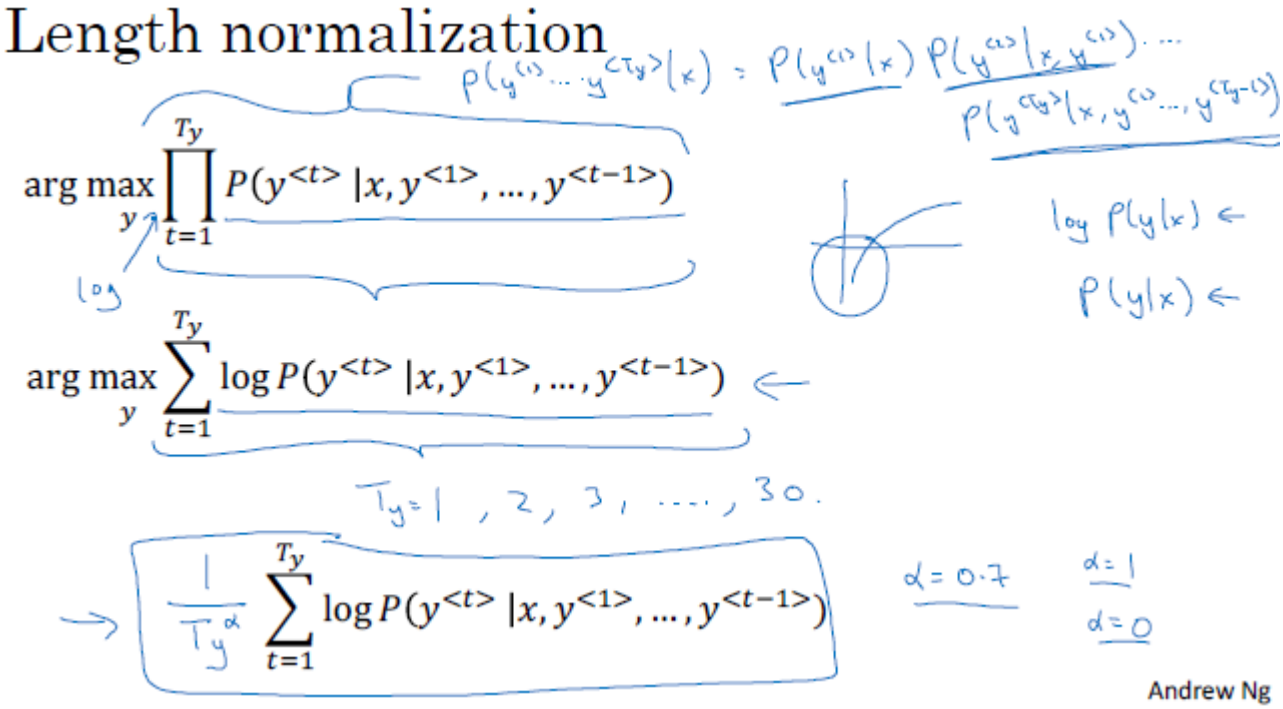


$P(y^{<1>}, y^{<2>} | x)$

jane visits africa in september. <EOS>

Refinements to Beam Search

Now that we are familiar with beam search, we are looking for a way to improve it. When we expand the probability, we see there are many probability values, between 0 and 1, which are getting multiplied. It results in numerical underflow. Consequently, we take its logarithm like maximum likelihood, we end up with a more numerical stable model.
It also has one undesirable effect. The more we multiply these probabilities, its value gets smaller and smaller. To this end, the model may tend to choose the shorter sentence. To deal with this problem, we can divide it by the size of the sentence, meaning number of words in the sentence. We can further use an α as a hyper-parameter which heuristically should be chosen and researchers found out it aids more in solving the problem.



Now the question is how to choose B? The larger it be, the probability of finding a better sentence is higher but computationally costs more. Researchers for publishing papers tend to use very large values for "B", however, in production B can be as small as 10 and as large as 100.
One more thing is that, for comparing the beam search with BFS or DFS, unlike the latter algorithms, beam search never guarantees to find the exact maximum for the conditional probability; however, runs much faster.

Beam search discussion

large B: better result, slower
small B: worse result, faster

Beam width B?

1 → 3 → 10, 100, 1000 → 3000

Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find exact maximum for $\arg\max_y P(y|x)$.

Error analysis in beam search

In the slide below, we have an example of the human and computer translation. So the problem is either from the RNN algorithm we used or the beam search. We can collect more data or increase the "B"; however, they are time-consuming and may not lead us to the correct answer. So the very first idea would be comparing the probability of the human sentence and computer one.

Example

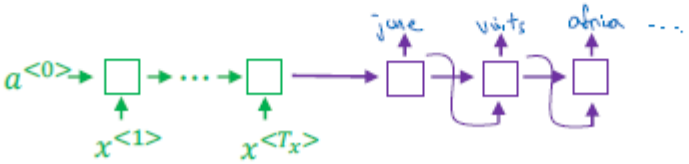
→ RNN
→ Beam Search
B↑

Jane visite l'Afrique en septembre.

Human: Jane visits Africa in September. (y^*)

Algorithm: Jane visited Africa last September. $(\hat{y})$ ⇐

RNN computes $P(y^*|x) \gtrless P(\hat{y}|x)$



How does it help? So if the human sentence makes more sense to the model, the blame goes to the beam search that couldn't find the correct set of words to maximize the probability of the sentence. If the computer sentence has higher probability of the human one, the beam search found the best words to maximize the probability; however, it was the RNN model fault that a sentence that makes less sense got higher probability of incident. Also, the normalization term is involved in the end result. So we may want to play with that as well.

Error analysis on beam search

$P(y^*|x)$
$P(\hat{y}|x)$

Human: Jane visits Africa in September. (y^*)

Algorithm: Jane visited Africa last September. $(\hat{y})$

Case 1: $P(y^*|x) > P(\hat{y}|x)$ ⇐

$\arg\max_y P(y|x)$

Beam search chose $\hat{y}$. But y^* attains higher $P(y|x)$.

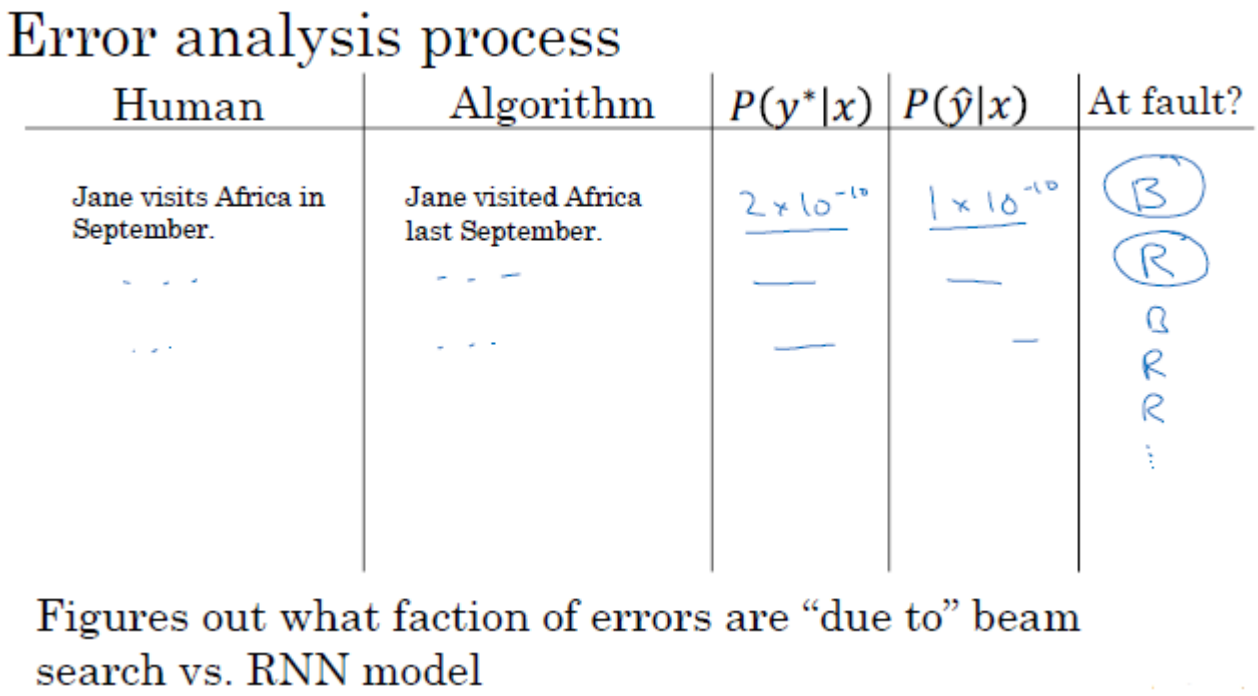
Conclusion: Beam search is at fault.

Case 2: $P(y^*|x) \leq P(\hat{y}|x)$ ⇐

y^* is a better translation than $\hat{y}$. But RNN predicted $P(y^*|x) < P(\hat{y}|x)$.

Conclusion: RNN model is at fault.

As before, we can use a error analysis table. In the table we find out each one of the wrong samples fault source and try to solve them. Everything we discussed in the third course is now applicable here as well.

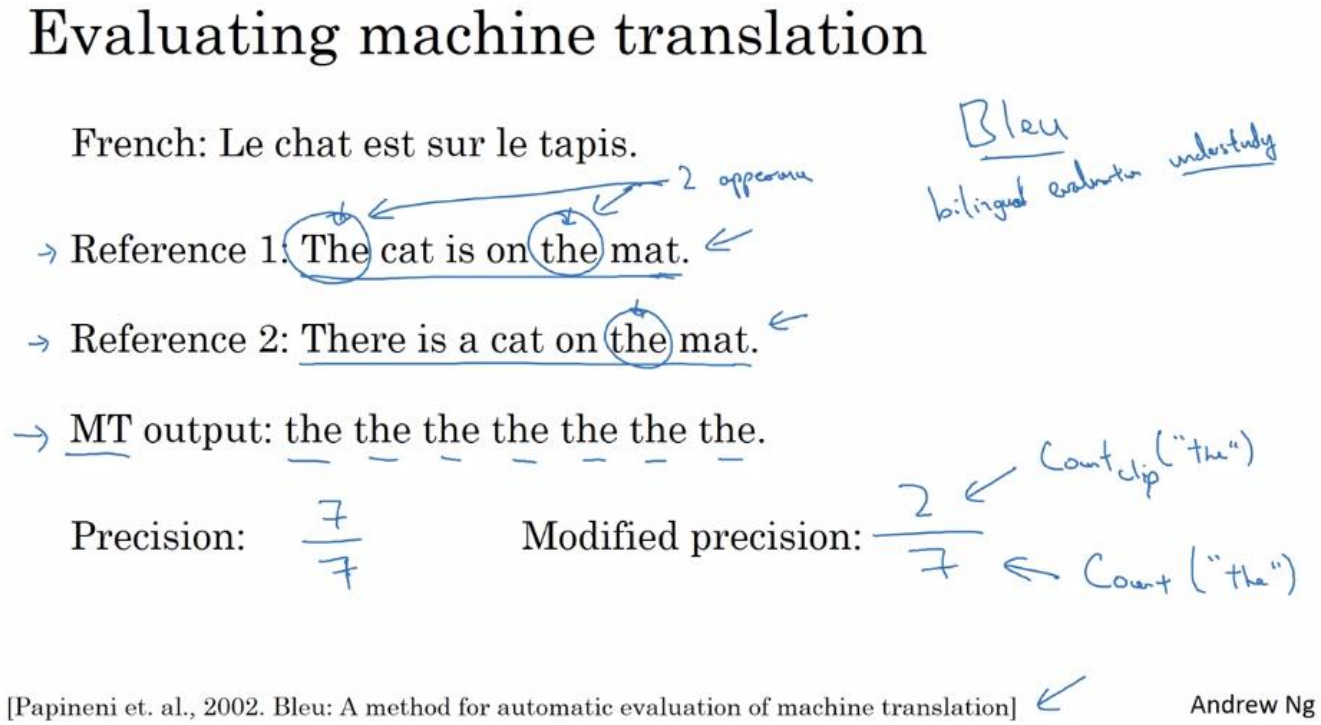


Figures out what faction of errors are "due to" beam search vs. RNN model

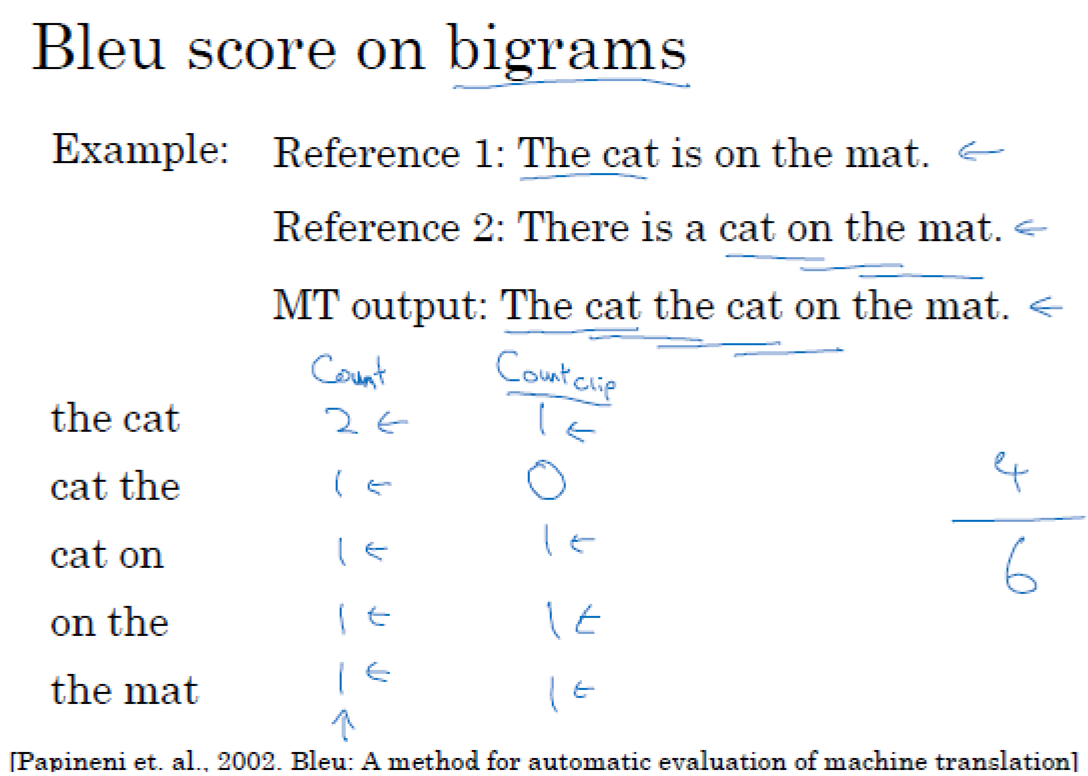
Bleu Score (optional)

In machine translation, we may have some equally good answers. So the question which arises is that which one to pick? We use Bleu Score for this purpose.

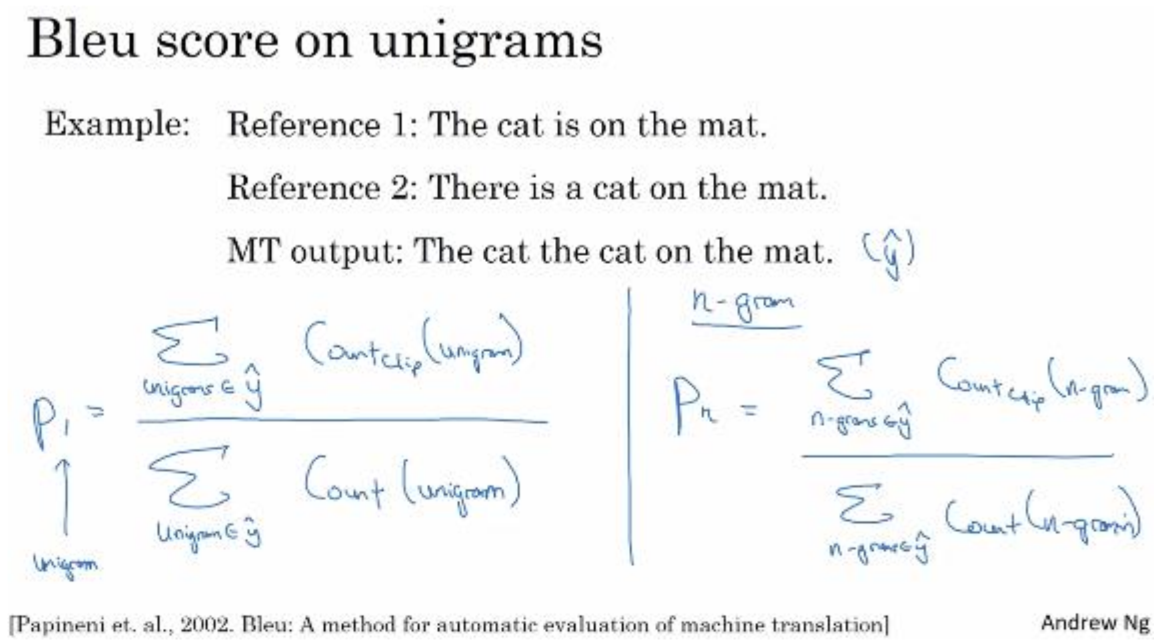
So with the Bleu (bilingual evaluation understudy) Score we calculate the score of each sentence. So the measure could be precision of a word translated with the machine, against reference translations we have. Since the word the in the example below, exists in both references, it takes 7/7 score. It is not good though. So for the modified version, we have in maximum, 2 “the” words. Then it means when we count the valid words, it becomes 2/7.



Now when we want to use it for bigrams, we extract the bigrams from the machine translated output. Then we compare it with references and we get 4/6.



So when we formulize it, we get:



As we want to calculate the Bleu score for the whole sentence, we have a set of n-grams and for each one we count the MT output correlation with reference. Then combine them. Moreover, we have a penalty, called brevity penalty (BP), which assigns weights to the MT output as it could be very short and get a very high score for that. So it is penalizing the very short sentences.

Bleu details

p_n = Bleu score on n-grams only

Combined Bleu score:

$$BP \exp\left(\frac{1}{4}\sum_{n=1}^{4} p_n\right)$$

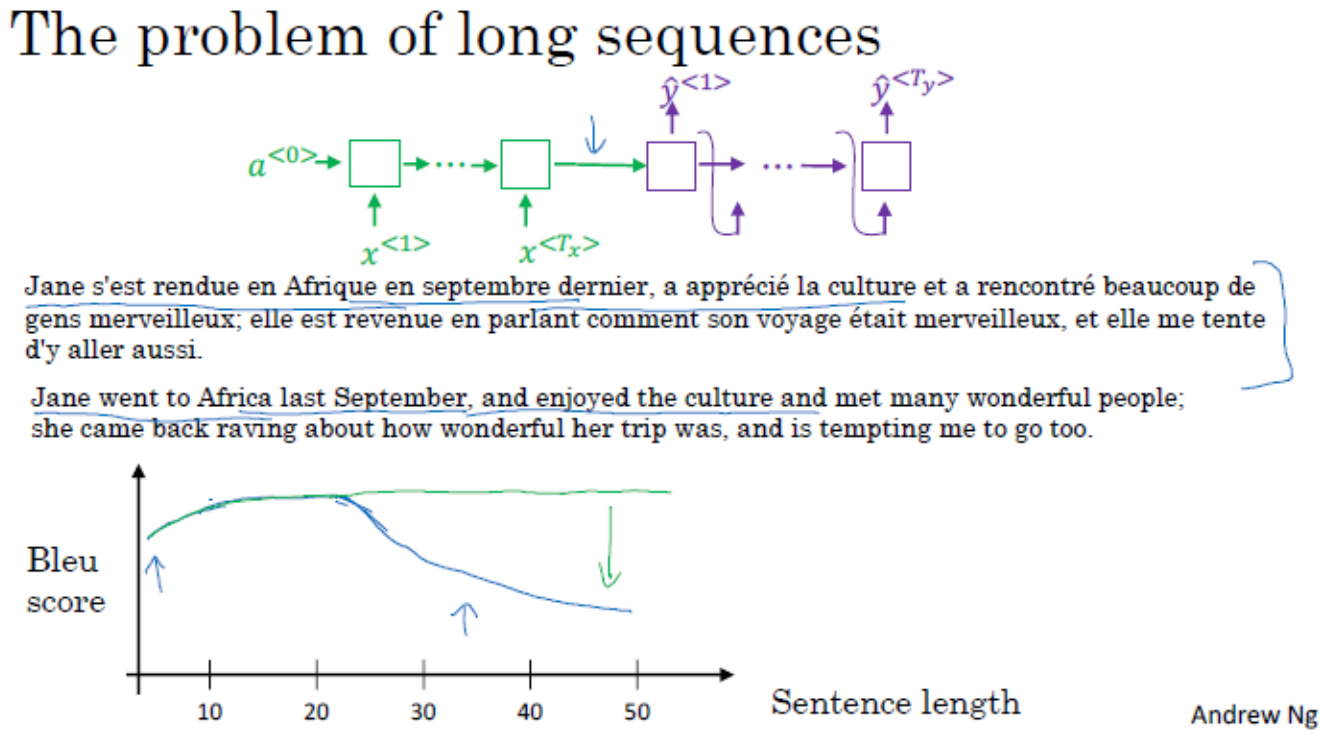
BP = brevity penalty

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

[Papineni et. al., 2002. Bleu: A method for automatic evaluation of machine translation]

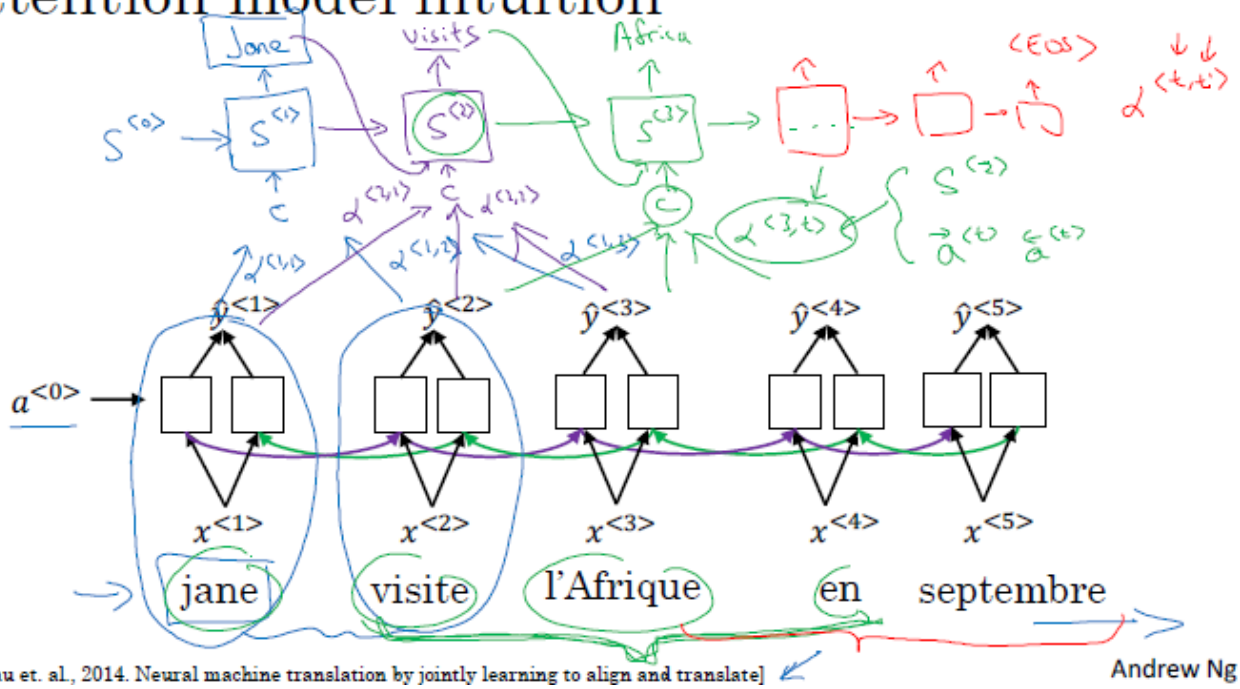
Attention Model Intuition

There is a problem with RNN models with encoder and decoder we had so far. Given a long sentence, how the RNN models behave is to first encode the whole long sentence or paragraph, then gives it to decoder to translate and output it. However, for human, it is not the case. We read one chunk of it, we output it, then, we read the next chunk and so forth. So what we expect to see from RNNs is:



So here, the attention model comes to scene. The very first step is for translating the first word, we check how much attention we got from the first word, second word, and so forth. Then, we give it to the decoder to generate word. For the next word as well, we have the attentions and the previous word as input to decoder for the second word output. Note we use "S" as activation of the decoder instead of "a". The $\alpha^{<t,t'>}$ is the amount of attention of the t' word on the t word.

Attention model intuition

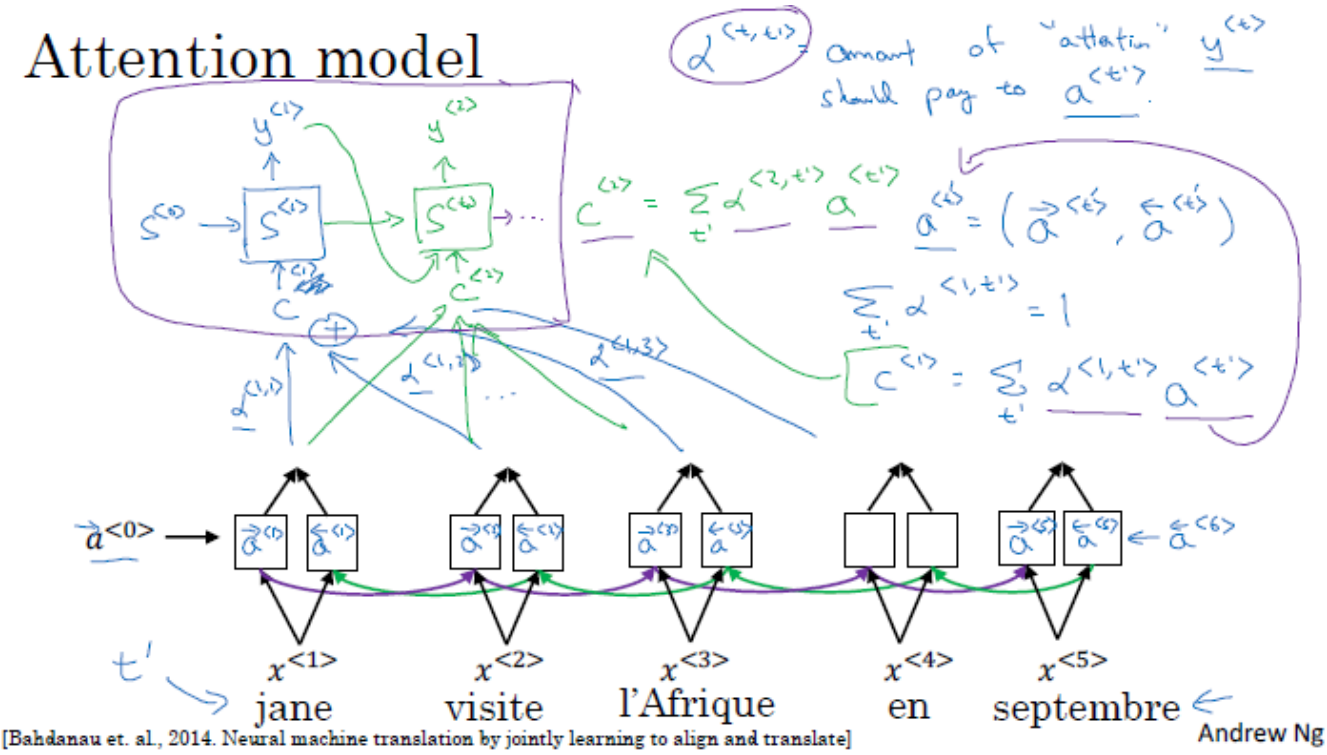


[Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate]

Andrew Ng

Attention Model

Let's go through details of this model. Let's recap what we saw before.



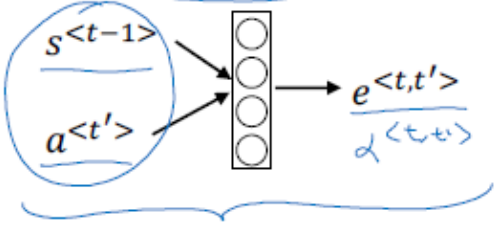
[Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate]

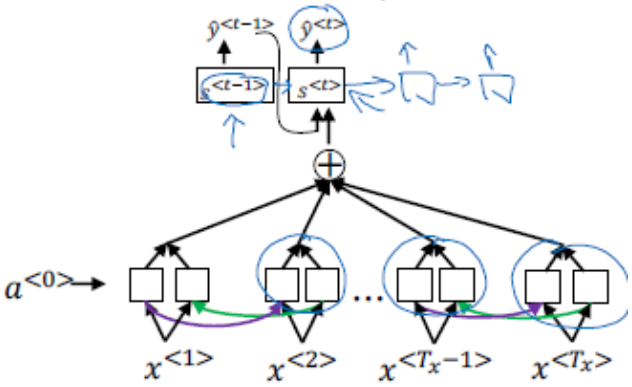
One point is that, we need the attention factors for each word sum up to 1. To this end, we use a Softmax to ensure they sum up to 1. The only way we can find the correct value to feed the decoder is to use a neural network and feed the activation of the previous decoded word and activations of the attention words. In this way, the NN can come up with a function to find the relation between them and feed it to the next decoder block. However, this model suffers from very high time complexity (quadratic). But if the sentence is not too long, it is not that bad.

Computing attention $\alpha^{<t,t'>}$

$\alpha^{<t,t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$





[Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate]
[Xu et. al., 2015. Show, attend and tell: Neural image caption generation with visual attention]

Andrew Ng

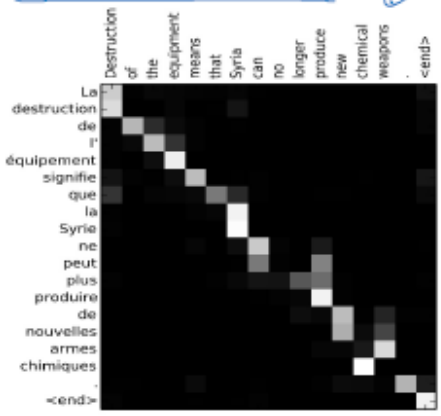
Let's see the examples and visualization. The visualization provides us with the fact that the attention model gives the higher weight to the correct words.

Attention examples

July 20th 1969 ⟶ 1969 – 07 – 20

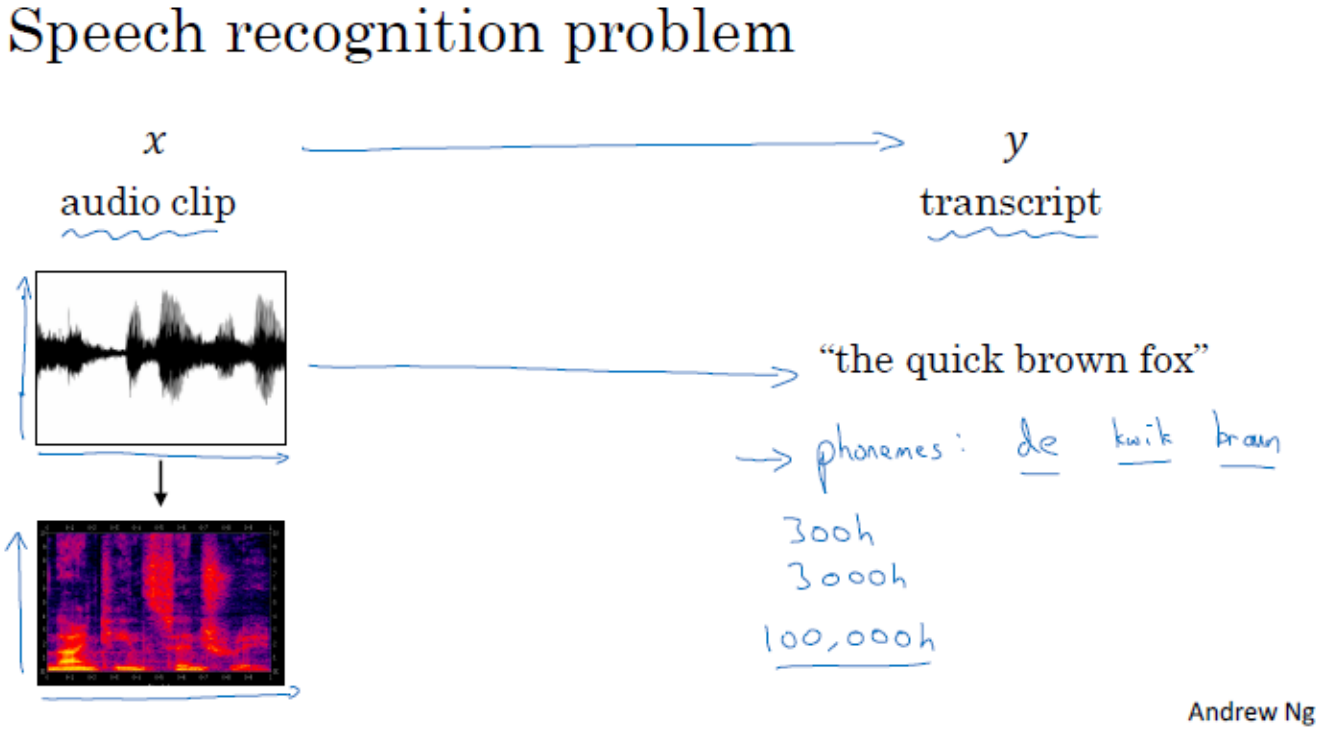
23 April, 1564 ⟶ 1564 – 04 – 23

Visualization of $\alpha^{<t,t'>}$:

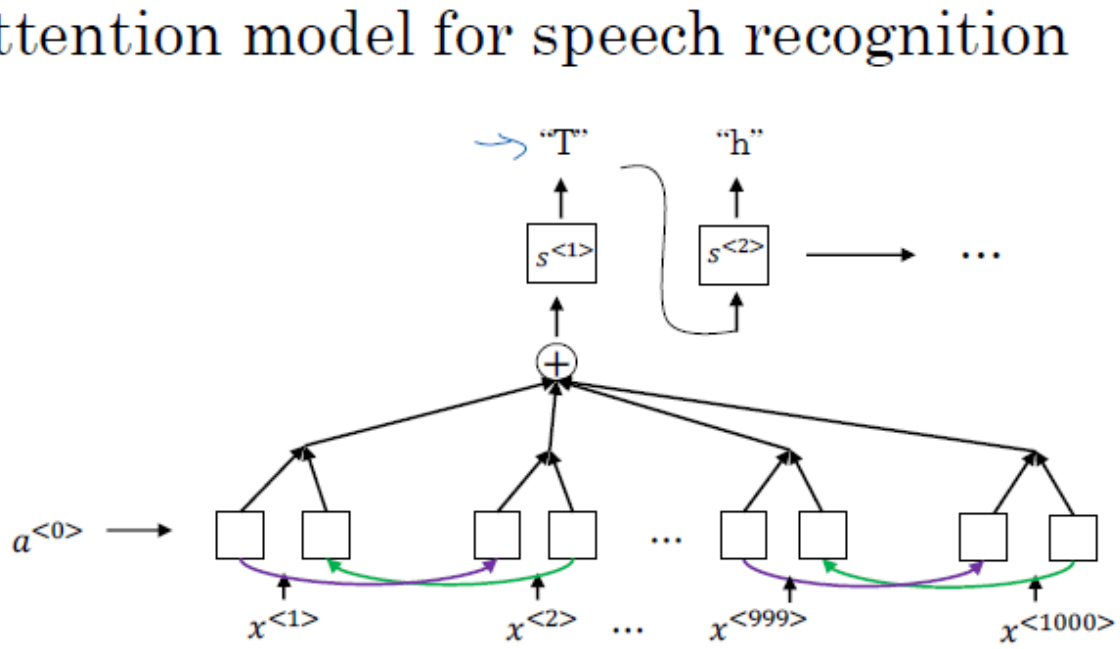


Speech recognition

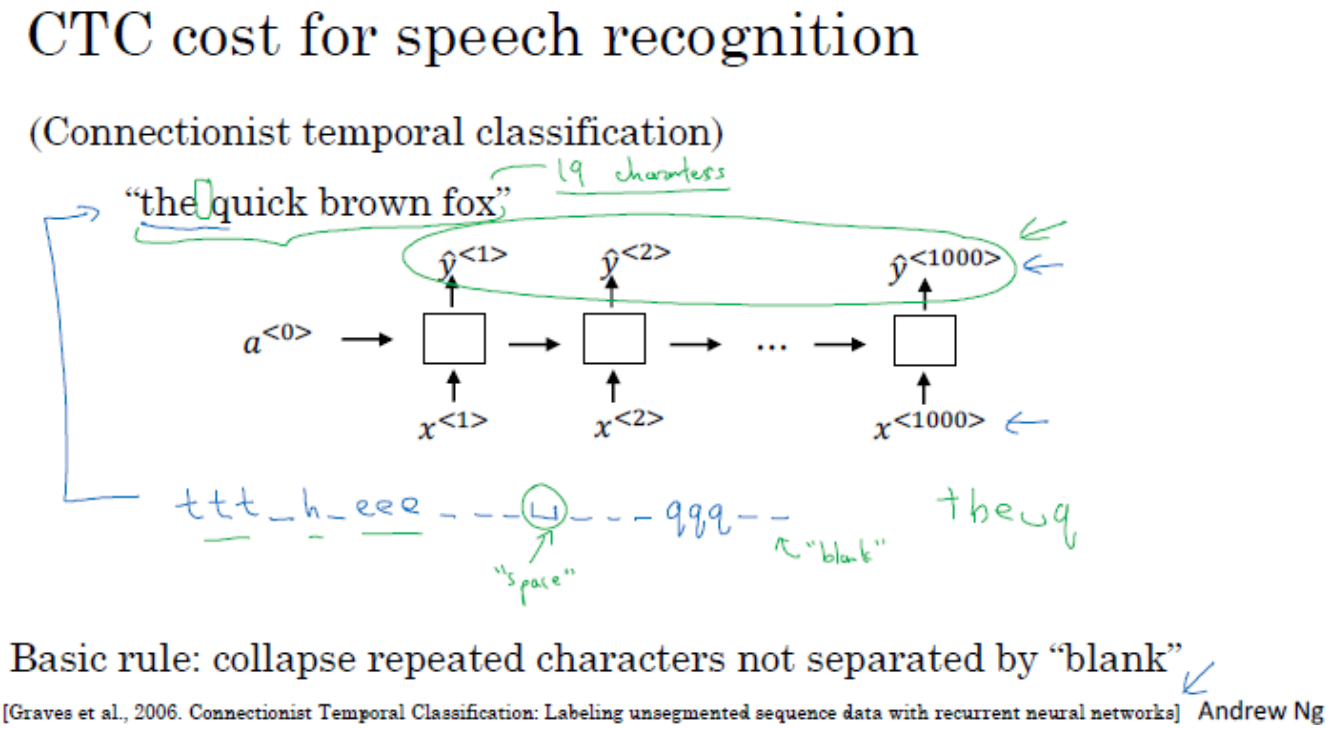
So, what we have in a speech recognition system, are time, frequency, and its intensity. What traditionally scientists were doing was coming up with phonemes for different words, like “the” as “de” and “quick” as “kwik”. They then tried to extract the phonemes out of the speech. But in the end-to-end architecture, researchers found phonemes are not necessary.



Now for the speech recognition model, we can use attention model.



In many cases of speech recognition, the number of input time-steps are highly larger than number of output time-steps. Now, if we get a very short sentence, while a very large RNN model, what we can do is to use CTC (connectionist temporal classification).
So for the case of a sentence like “the quick brown fox”, we have “ttt_h_eee_ _ _ _ _ _qqq_ _ …”. So in this way, by repeating some characters multiple times, we can output a 19 characters with 1000 RNN blocks for example.



Trigger Word Detection

What is trigger word detection?

Amazon Echo
(Alexa)

Baidu DuerOS
(xiaodunihao)

Apple Siri
(Hey Siri)

Google Home
(Okay Google)

For the trigger word detection application, researchers still didn't reach a consensus. So in below, we can see one of the solutions. So one possible architecture is for the model to hear everything and returns 0 except for the trigger word. As the number of 0s are highly much more than the trigger words, we can set several 1s for the trigger word to get outputted from the RNN. It a hack actually.

Trigger word detection algorithm