Course 1: Neural networks and deep learning

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/neural-networks-deep-learning/home/welcome

****

Course 1: Neural Networks and Deep Learning

****

In this course, you will learn the foundations of deep learning. When you finish this class, you will:
- Understand the major technology trends driving Deep Learning
- Be able to build, train and apply fully connected deep neural networks
- Know how to implement efficient (vectorized) neural networks
- Understand the key parameters in a neural network's architecture

This course also teaches you how Deep Learning actually works, rather than presenting only a cursory or surface-level description. So after completing it, you will be able to apply deep learning to a your own applications. If you are looking for a job in AI, after this course you will also be able to answer basic interview questions

Week 1: Introduction to Deep Neural Networks

Learning Objectives

- Understand the major trends driving the rise of deep learning.
- Be able to explain how deep learning is applied to supervised learning.
- Understand what are the major categories of models (such as CNNs and RNNs), and when they should be applied.
- Be able to recognize the basics of when deep learning will (or will not) work well.

What is a NN?

Housing price prediction with 1 neuron

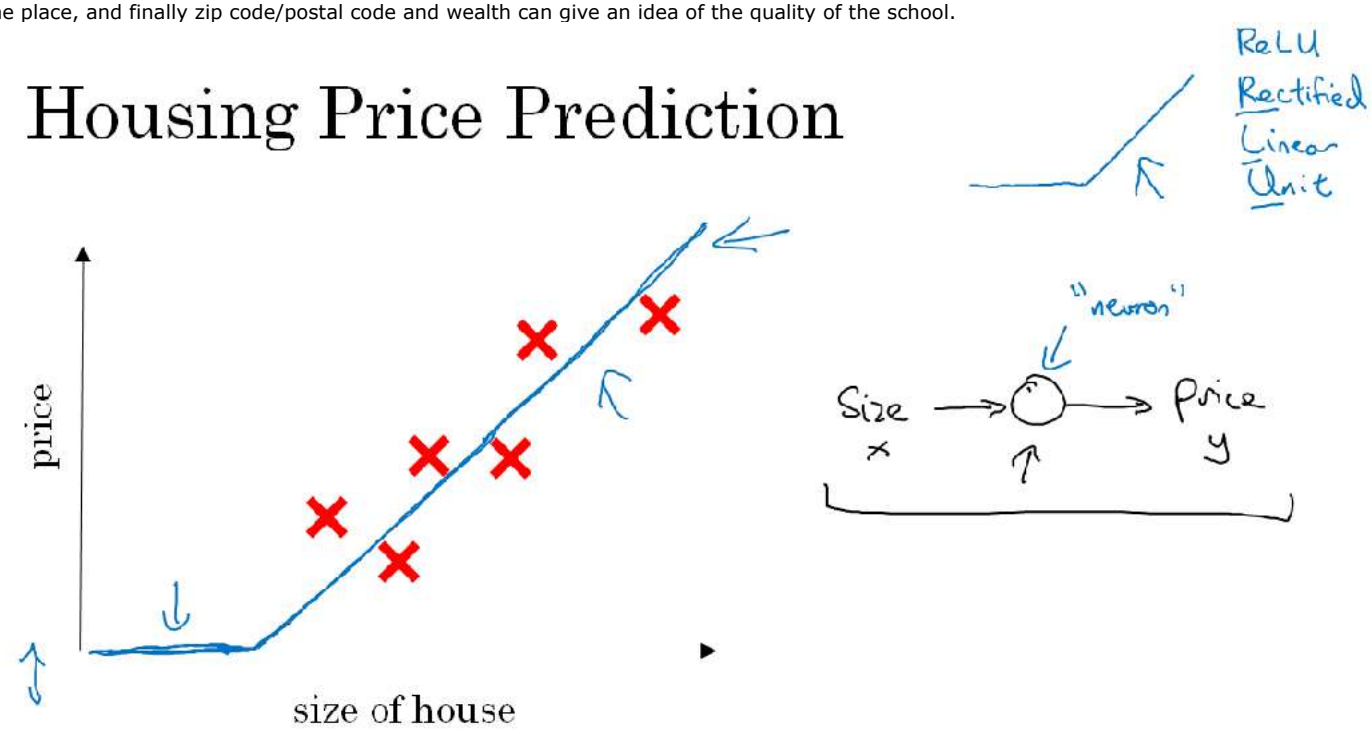
This is the simplest possible NN which can be solved by liner regression

Size of the house -> single neuron-> price -- This is the tiniest NN. We use Relu function (Recified Linear Unit) will learn more in upcoming lectures

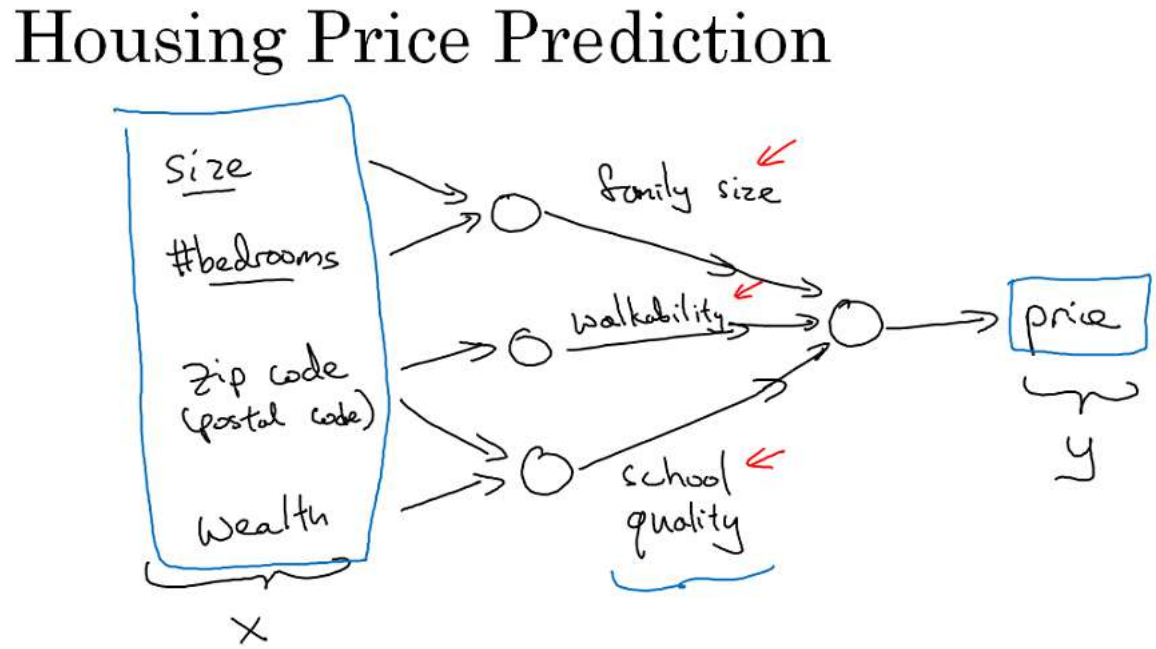
Housing price prediction advance (with more than 1 neuron)

Bigger NN can be make by stacking single NN together. Let's add more parameters like no of bedrooms, size -> family size, zip code -> walkability, zip, wealth-> school quality.

to elaborate further no of bedrooms and size of the house depends on the family size, zip code or postal code tells about location and walkability to the place, and finally zip code/postal code and wealth can give an idea of the quality of the school.



so in this scenario people might pay for the house based on **family size**, **walkability** and **school quality** and that helps to predict the **price** of the house. X is the inputs and output is y. So we have to give input which are like bedrooms, size, zip, wealth and in between parameters like family size, walkability etc will be figured by NN. So we can say that we have 4 inputs x1, x2, x3 and x4 and predicting price (y). in between we have 3 hidden NN units (as we have 3 intermediate inputs). One of the thing about these hidden units is that all takes the 4 inputs (x1, x2, x3, x4) means that any of the hidden NN unit can be used for any of the 3 functions (**family size**, **walkability** and **school quality).** NN will figure out the functions which will map x->y.



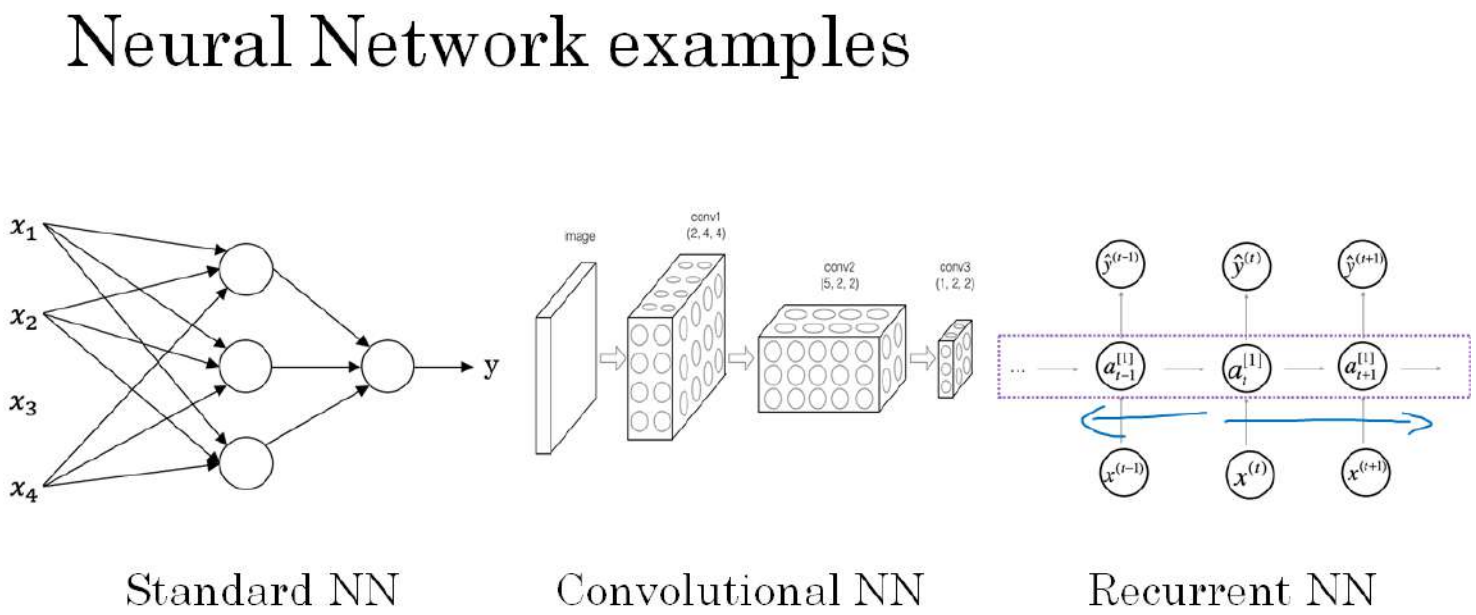
So it is found that NN are more useful in supervised learning setup where we have x input and map to y output as we saw in housing price prediction problem.

Supervised learning with NN?
Mainly supervised learning applications has taken good advantage of DL.
A simple task we do in supervised learning is to learn some function on input x which shall predict output y.

Some of the examples of **supervised learning** are as follows:

Input(x)-> Output(y) -> Application

- Home features -> Price -> Real Estate : Standard NN
- Ad, user info -> click on ad (0/1) -> Online advertising : Standard NN
- Image -> Object (1....10000) -> photo tagging : We use CNN
- Audio ->Text transcript -> speech recognition: A-D temporal data or 1-D time series data- Recurrent Neural Netword (RNN's)
- English -> Hindi -> Machine translation: More complex version of RNN
- Image, Radar info -> Position of other cars -> Autonomous driving : Custom hybrid version of CNN and RNN's



Structured Data:
Housing prediction
Ad click etc.
Unstructured data:
Audio
Images
Text

Supervised Learning

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data

Audio

Image

Four scores and seven years ago...

Text

Computers are now able to understand unstructured data in a better way due to deep learning. Techniques in this course will apply both in structured and un-structured data.

Why deep learning is taking off now this fast?
This is due to the many factors in last 10 yrs

- Digitization
- lots of data
- cell phone data
- small NN has bad performance but very large NN have very good performance so we need to train bigger NN for that we needs lots of data to get good performance
- Either bigger network or through more data to improve scale
- That is amount of labelled data
- We'll use m is the size of train set / size of training examples
- as earlier the components like Data, computation and algorithms in deep learning era a lot of emphasis has been given to algorithm. For example changing activation function from **Sigmoid->Relu** can help in solving deceasing to zero gradient problem. This means that small algorithms changes helped algorithm Gradient descent perform much faster which eventually helped in computation.
- The typical cycle of making a deep NN model **Idea->Code->Experiment REPEAT** In repeat process to implement ideas faster we need faster computation which can in making quick prototyping so inventing new algos helps.

  To conclude deep learning is taking off because of availability of lots of digital data which we are continuously creating, lots of research and availability of better hardware, GPU's, quasi networks for computation and advancements in deep learning algorithms. this all helps in implementing idea quickly and in training bigger network.

Week 2: Basic of Neural Network Programming

Learning Objectives

- Develop intuition about structure of Forward Propagation, Backward propagation and steps for algorithms and how to implement NN efficiency.
- Build a logistic regression model, structured as a shallow neural network
- Implement the main steps of an ML algorithm, including making predictions, derivative computation, and gradient descent.
- Implement computationally efficient, highly vectorized, versions of models.
- Understand how to compute derivatives for logistic regression, using a back-propagation mindset.
- Become familiar with Python and Numpy
- Work with iPython Notebooks
- Be able to implement vectorization across multiple training examples

Unit 1 - Logistic regression as a Neural Network

To implement NN as we know we can have m training examples and to go through it we'll need a for loop but in this week we'll learn how to go through m training examples without explicit for loop.
Also in this week you will how the NN is computed, a forward pass and a backward pass. We'll implement these ideas using **Logistic Regression**.

Binary classification

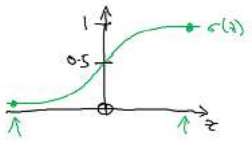
Logistic regression is the algorithm for binary classification. Ex: You have an image input, the binary classification will tell either its cat(1) or non-cat(0) this output label is called y. The input image is stored in computer in 3 separate matrices (RGB) as per size of the image, means that if you have 64x64 image of a cat then each R, G, B matrices will be 64x64 so computer will store 3 64x64 pixel matrices. So now to turn these 64x64 pixel intensities to feature vector we un-row all of them to input feature vector x. We unrow like make a col with all R then G and then B, dimension of x

will be 64x64x3 = 12288 can say nx = 12288 which represents the dimension of input feature vector x. So in binary classification we have a input feature vector x which predicts output label y which tells if this is a cat image or non-cat image.
Notations: Some of the notations which will be used throughout the course.
(x, y) -> Single training example where x belongs to nx and y belongs to {0, 1}
m training examples : $\{(x^1, y^1), (x^2, y^2), (x^3, y^3)..... (x^m, y^m)\}$ here 1, 2, 3 ... m denotes 1st, 2nd, 3rd training examples. Sometimes we do write M_{test} = No of test examples, M_{train}= No of training examples. Finally to put all the training examples in a more compact notations so we define capital X and take the training set inputs x^1 , x^2, x^3 and so on and stacking them together as a col vector. x^1 is the first col of the matrix, x^2 is the second col of the matrix and so on till x^m. So this matix X will have m col where m is the no of training examples, and the height of the matrix nx is the no of rows of the matrix. Just to recap X is a $\mathbf{n_x \times M}$ dim matrix. So while implementing in Python **X.shape** gives $\mathbf{(n_x \times M)}$. To make NN implementation more convenient we also stack y's in col vector capital Y = [y^1, y^2, y^3..... y^m] Y here will be a 1x M dim matrix and again to use the Python notation Y.shape = (1,M) means this is **1xM** dim matrix.

Logistic regression

Logistic regression is a learning algorithm used in a supervised learning problem when the output are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data. This can also be seen as a very small neural network. This is the learning algorithms which is used when output label is either 0 or 1. Given an input x in this case image, we want predict y hat which is the probability of y, **yhat = P(y=1|x)**. So it tells you that what is the chance of a image to be a cat pic for a given input x. As we have discussed in previous lecture x is a **nx** dim matrix so parameters of the logistic regression are w which is also a $\mathbf{n_x}$ dim vector together with **b** which is just an row number. So with given input x with parameters w and b, how to generate output yhat? One thing can be done is to try output yhat = $w^T x + b$ (w transpose x + b) which we do in a simple linear regression. but this isn't a very good algorithms for binary classification because we really wants yhat to be between 0|1 $w^T x + b$ can be much bigger or can be negative so doesn't make sense to use that..so in logistic regression our yhat will be the sigmoid function of the $w^T x + b$ (yhat = sigmoid($w^T x + b$))

The basic sigmoid function looks like: $\sigma(z) = \frac{1}{1+e^{-z}}$



since we are looking for a probability constraint between [0,1], the sigmoid function is used. The function is bounded between [0,1] as shown in the graph above.
Some observations from the graph:

- If is a large positive number, then sigmoid(z) = 1 / (1 + 0) = 1 (close to)
- If is small or large negative number, then sigmoid(z) = 1/(1+ very big no) = 0 (close to)
- If =0, then sigmoid(z) = 0.5

So when you implement logistic regression your job is to try to learn parameters w and b so that yhat/ˆ becomes a good estimate of the chance of being y = 1

Logistic regression cost function

To train the parameter w and b of the logistic regression model we need a cost function.
To recap from previous section we have output yhat as follows

ˆ = (w^T +), where ()= $\sigma(z) = \frac{1}{1+e^{-z}}$

coming back to notation part we use superscritpt i in brackets to denote ith training example
Given $\{(x^1, y^1), (x^2, y^2), (x^3, y^3)..... (x^m, y^m)\}$ we want $\mathbf{\hat{y}^{(i)} = y^{(i)}}$
Loss (error) function is used to tell how well our algorithm is doing

LossFn(yhat, y) = 1/2(yhat-y)^2 : This means that loss function between output label yhat and true label y is the one half of the square error. So LossFn tell that how good is our **yhat** if true label is **y**. Error of half of square root seems to be a reasonable choice except the fact that gradient descent may not work well so in logistic regression we define a different loss function. This function is similar to squared error loss function but given a convex shape which helps in optimization. We'll discuss optimization in detail in later chapters. So the loss function in logistic regression is defined:
LossFun(yhat, y) = -(y log yhat + (1-y) log (1-yhat))
Like squared error loss function we want this loss/error function to be as small as possible.
if y =1 LossFun(yhat, y) = - log yhat + (1-1)log (1-yhat)) = - log yhat, we want log yhat to be large as big as possible means yhat to be large but yhat is the sigmoid of y so it can be as big as possible but not greater than 1
if y =0 LossFun(yhat, y) = - log yhat + (1-0)log (1-yhat)) = - log (1-yhat), we want log (1-yhat) to be small means yhat to as small as possible it will try to make it close to 0
The loss function above has been defined on a single training example it measures how well you are doing on a single training example. Now we'll define a function called cost function which measures how well you are doing on a entire training set. In other words, we can say that the loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set. Cost function J is defined on parameters w and b which is the sum of loss function on each training example. see below for full formula:

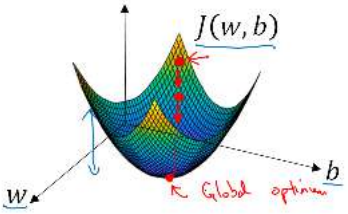
Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)})\right]$

So to conclude the understanding LossFn is applied to the single training example and the cost function is the cost of your parameter so in training the logictic regreession model we are going to try the parameter w and b that minimize the overall cost function J. So this is the setup for Logistic regression algorithm.

Gradient descent

Gradient descent algorithm is used to train the parameters w and b. While training we want to find w and b that minimize cost function J(w,b). Generally w and b are single row no values plotted in x-axis, J is a convex function a single big bow. To find the good parameters value for w and b

what we'll do is to initialize w and b to some initial value, we can initialize to zero. Random initilization can also work but people usually don't so that for logistic regression so what the gradient descent does is to start at this initial point and take a step in the steepest downhill direction or say descent as quickly as possible. This is a one iteration of the gradient descent. We can go on with the multiple iteration till it converses to the global optima / close to global optima.



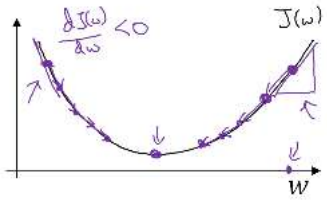
For the purpose of illustration lets say that there is some function J(w) we want to minimize. To make this easier to draw we are going to ignore b for now, just to make this a 1-D plot instead of higher dim plot.

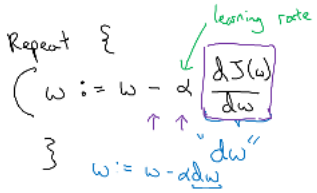
Repeat {
w := w - alpha dJ(w)/dw
}

do repeatably till the time algorithm converses. Couple of points to be noted..

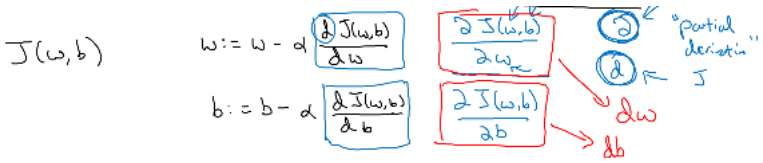
alpha here is the learning rate and controls how bigger step we take on each iteration of gradient descent. dJ(w)/dw is the derivative (slope of a function at a point), the update of the change you want to make to the parameter w. When we'll write code variable dw will be used for derivative term.

Derivative or slope of a function at a point is the height/width, there can be two cases, in one case derivative is positive then we subtract the **w - alpha dj(w)/dw**, we end up taking step to the left so gradient descent will slowly converses if you have started with the large value of w. In other example if w is small no means **dj(w)/dw < 0** then we end up taking steps to the right **w - alpha (-dj(w)/dw)** a negative number means we are increasing w bigger by bigger on successive iteration. So hopefully whether you initialize w to left (small val) or right (bigger val) gradient descent will move you to global minimum.



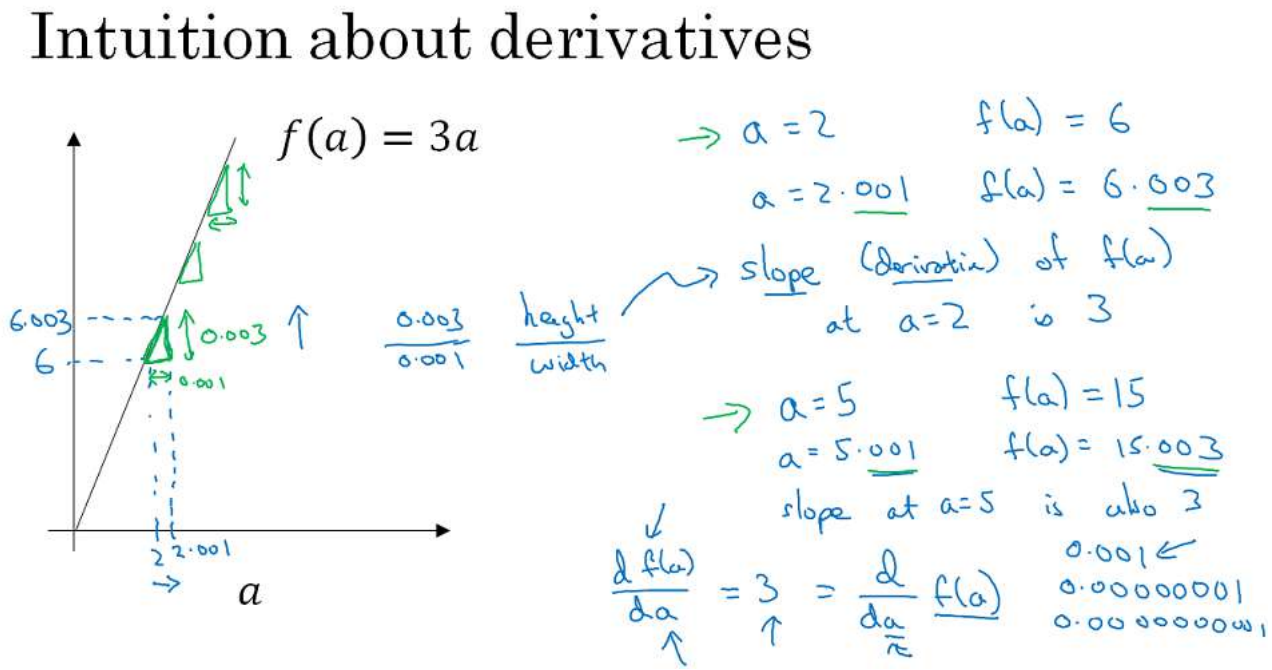


Overall intuition is now is that **dJ(w)/dw** represents the slope of the function and we want to know slope of the function at the current setting of the parameters so that we can the take the step of steepest descent. Now for both the parameter w and b **J(w,b)** we need to update the parameter b too also if J have two parameter then this derivative like **dJ(w,b)/dw** is called "partial derivative" so if J is the function of two or more than two variables then we use partial derivative symbol else we use small letter "d" but this doesn't make much difference while coding we use dw and db respectively for derivative of w and b. See below diagram for quick overview of what we have just learned.



Derivatives

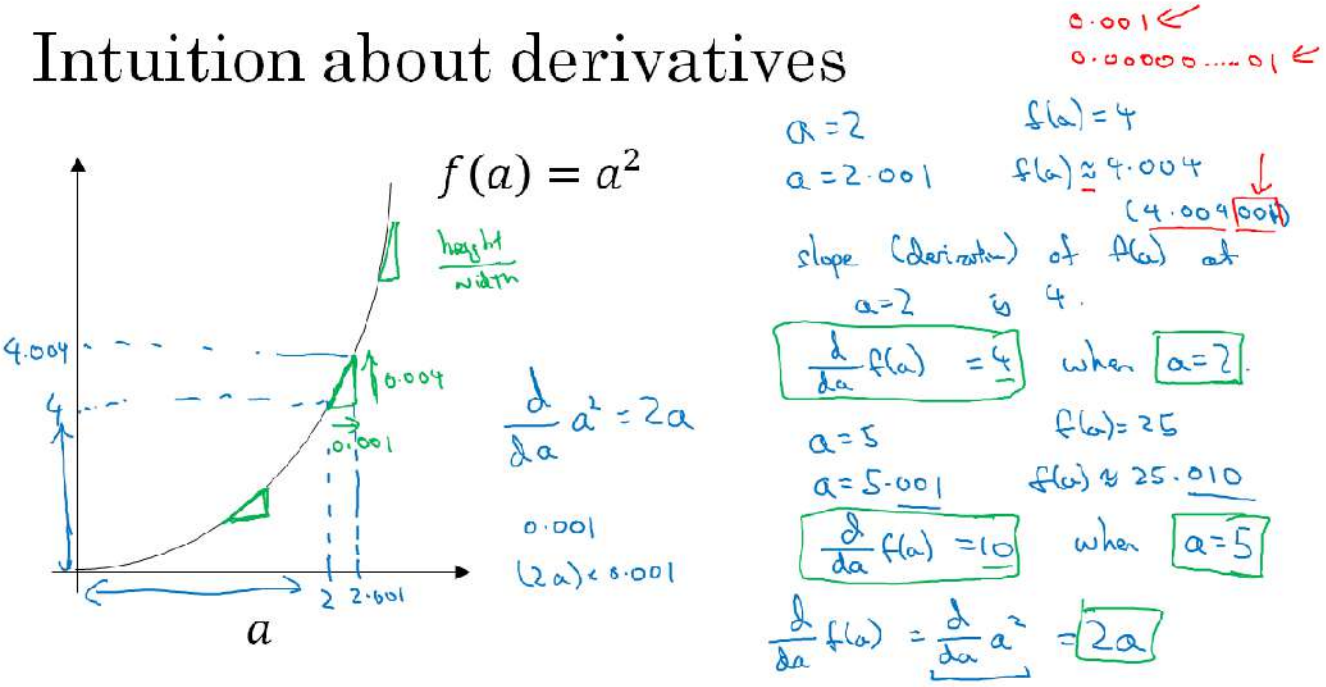
Let's first check the diagram below:



Let's go step by step and discuss the intuition about derivatives. First plot a function f(a) = 3a just a straight line, lets take a=2 in that case f(a) = 6 (check in diagram too), now give a a little bit of nudge, a = 2.001, f(a) becomes 6.003 (check in plot), now if you check the little triangle in the plot we see that if we give a tiny nudge of 0.001 to the left (x-axis) the f(a) is up by 0.003, so the amount f(a) went up is 3 times the amount nudge a to the right. So we can say that the slope (derivative) of the f(a) at 2 is 3. Formally slope is the height/width of the little triangle (check diagram) which is 0.003/0.001 = 3. Let's look at this function at a different point, lets take a=5, f(a) = 15, again give a tiny nudge a=5.001, f(a) = 15.003, again at a=5 slope is 3. We can write, slope of the function f(a) when you nudge variable a with tiny little amount is df(a)/da = 3 or d/da f(a) = 3. The value of nudge to the right can be very small but rule remains the same that we nudge a to the right with tiny tiny amount the f(a) will go up by 3 times in all the case slope will be 3. One of the property of the derivative is that no matter where you the slope of the function either a=2 or a=5 the slope of the function is 3. so wherever you draw this triangle the ratio of slope is 3:1. To conclude, in this example for a straight line slope of the function was 3 all over the places. We can say that On a straight line, the function's derivative doesn't change.

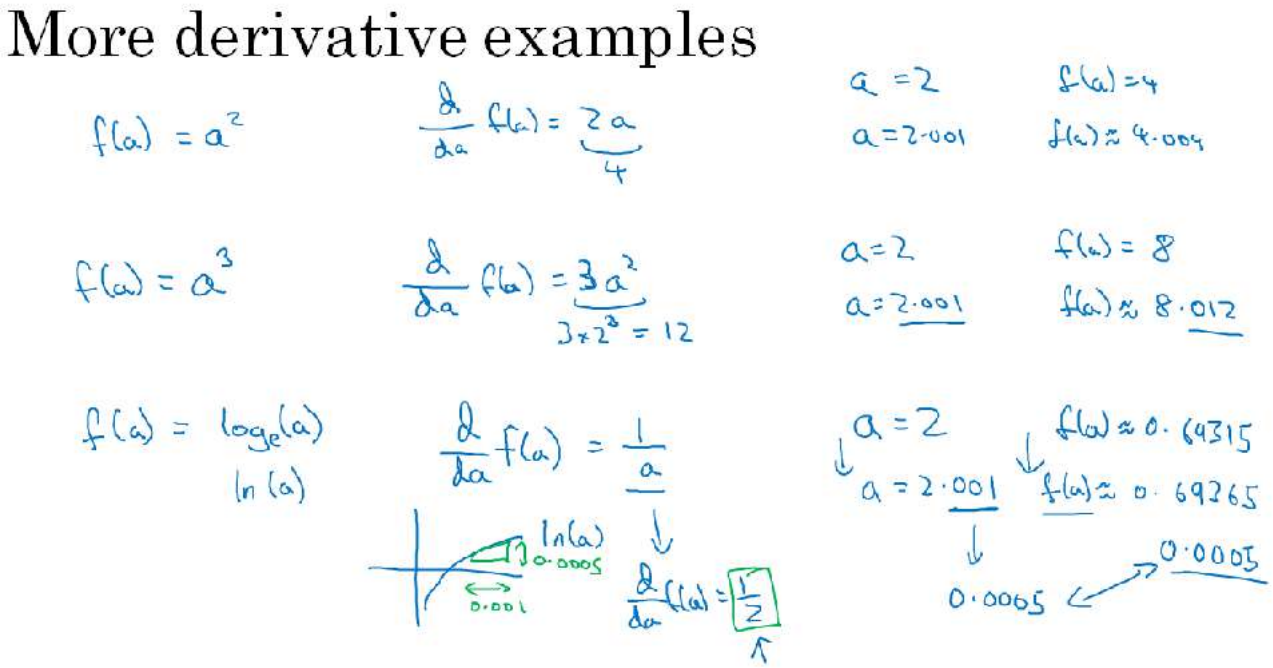
More derivative examples

Let's first check the diagram below:



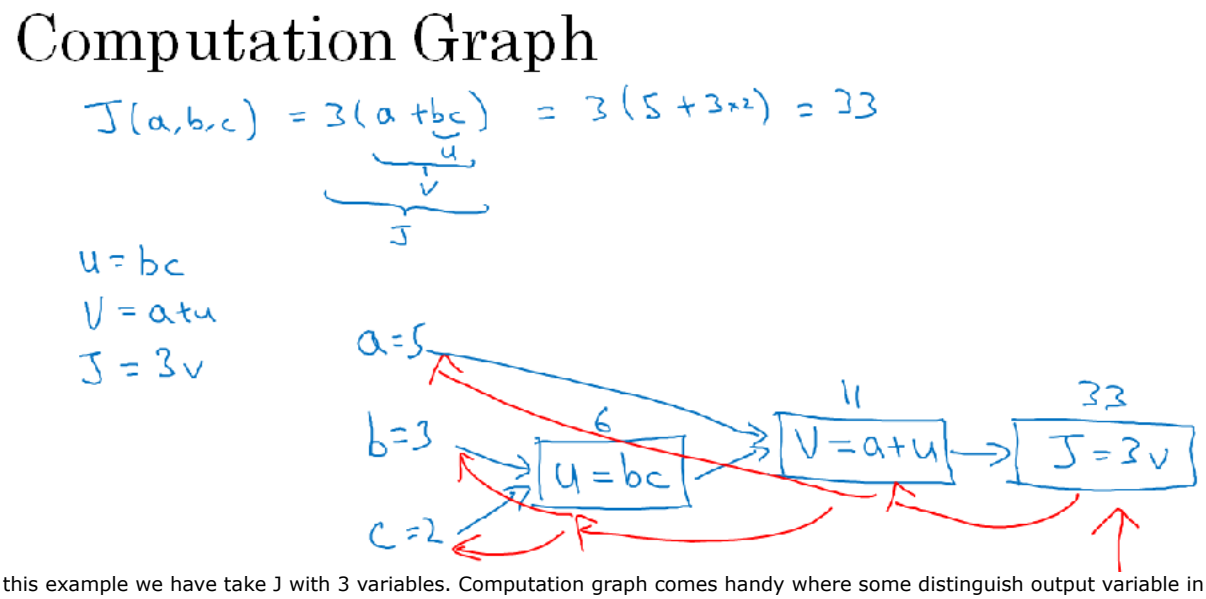
Let's go step by step and discuss the intuition about derivatives. In this example we'll see that slope of the function can be different at the different point at the function. The plotted function is f(a) = a^2 for a = 2, f(a) = 4, now slightly nudge val of a = 2.001, f(a) =4.004. If we plot this (as shown diagram above) if we nudge a by .001 to the right then f(a) went up by 0.004 which is 4 times the nudge val of a. So we can say that the slope (derivative) of f(a) at a=2 is 4 or d/da f(a) = 4, when a=2. Now lets see other val which is a=5, f(a) = 25, now nudge a by 0.001 becomes 5.001 f(a) becomes 25.010 on these val we can see that if we nudge a by 0.001 then f(a) goes up by 0.010 which is 10 times. so d/da f(a) = 10

when a=5. One way to understand why f(a) is different for different val of a, check the diagram and see that if we draw the triangle on the curve then height and width of the triangle is different at different places on the curve. If you go through the calculus text book formula table you will see d/da f(a) = d/da a^2 = 2a, this formula is satisfying the example we just saw so we can say that if you nudge a by 0.001 then f(a) will go up by 2 times a. Check the diagram below for couple of more cases mainly when f(a) = a3, d/da f(a) = $3a^2$ and f(a) = Ln(a), d/da f(a) = 1/a



Computation Graph

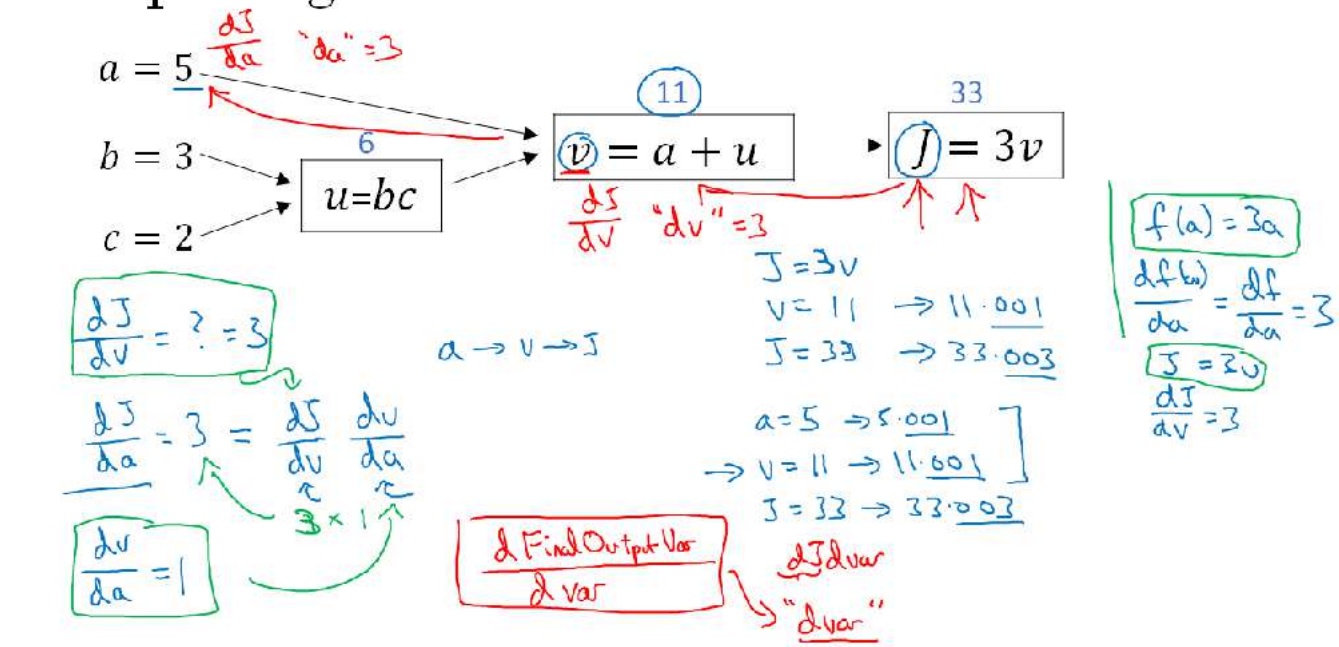
We firmly say that computation of neural network is organized in terms of two steps: a forward pass or forward propagation step which computes the output of the neural network followed by a backward pass or backward propagation which is used to compute the gradient or derivatives. The computation graph explains why it is organized this way. See below the diagram:



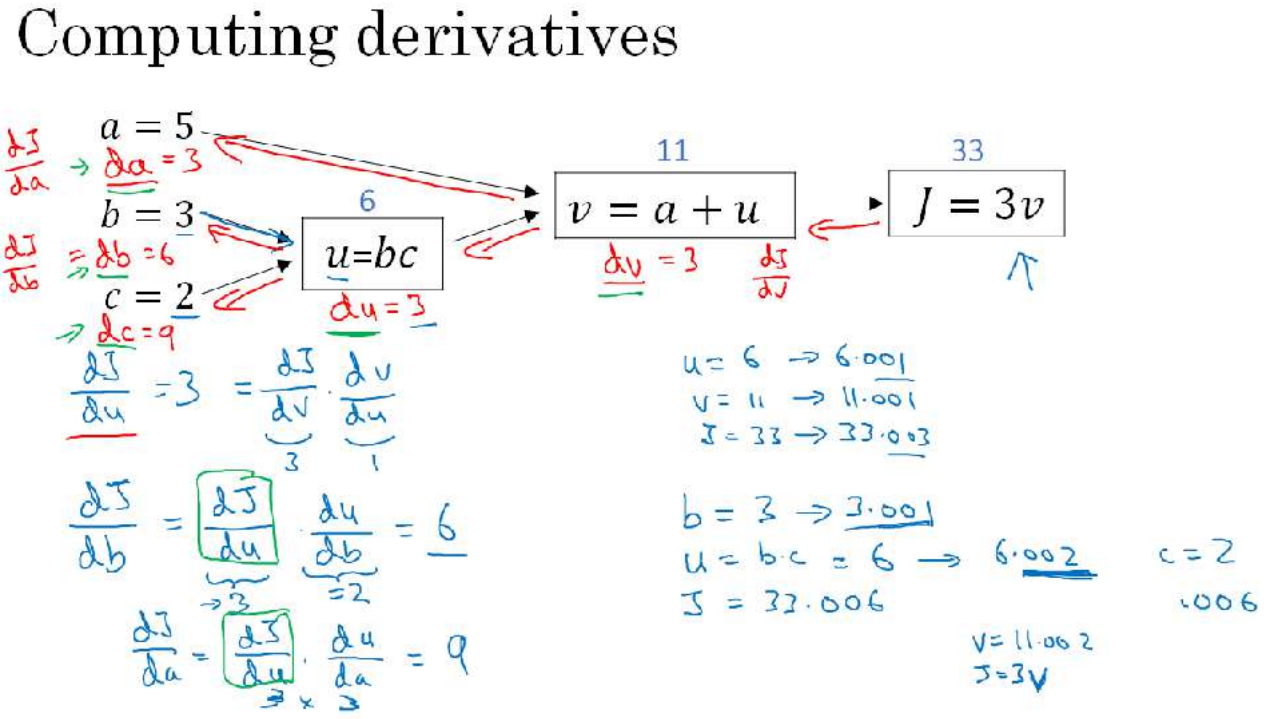
In this example we have take J with 3 variables. Computation graph comes handy where some distinguish output variable in this case to be optimized. In case of Logistic regression it is the cost function which we are trying to minimize. In this example we have also seen that with left to right pass we have computed val of J which is blue arrow in the diagram.

Derivatives with a computation graph

In last section we have made a computation graph and calculated J. In this section we'll see how this computation graph can be used to calculate derivative of function J. Let's first check the diagram below.



Looking into the computation graph at the top lets say we want to calculate dJ / dv (derivative of J with respect to v). It says that if we change the value of v little bit how much the val of J will change. Right J=3v and v=11, now if we pump v by little bit 11.001 , then J = 3v becomes -> 33.003. So J goes 3 times the val of v. So we can say that dJ / dv = 3. So in the terminology of the back propagation we have seen that if you want to compute the derivative of the final output variable J (which usually is the variable we care most about) with respect to variable v, then we have done sort of one step of back-propagation, so we came one step backward in the graph.
Now lets take another example dJ / da (derivative of J with respect to a) = 3 (check diagram for computation). so if you change a it will change v and that will change J. In other words, dv / da change in a you will end up increasing v, then change in v also make value to J also increase by dJ / dv. So the formula looks like (dJ/dv)(dv/da) which is called chain-rule in calculus. So this illustration shows us that how changing dj/dv helps in calculating dj/da, which is another step in backward propagation. A small note about nomenclature, we use a term Final Out Variable which in most of the cases is J sometimes L (loss) and we take derivatives wrt to other intermediate variables like a, b, v etc. In Python we present this term dFinalOutoutVar / dvar => dvar. See the below diagram for more examples on computing derivatives:

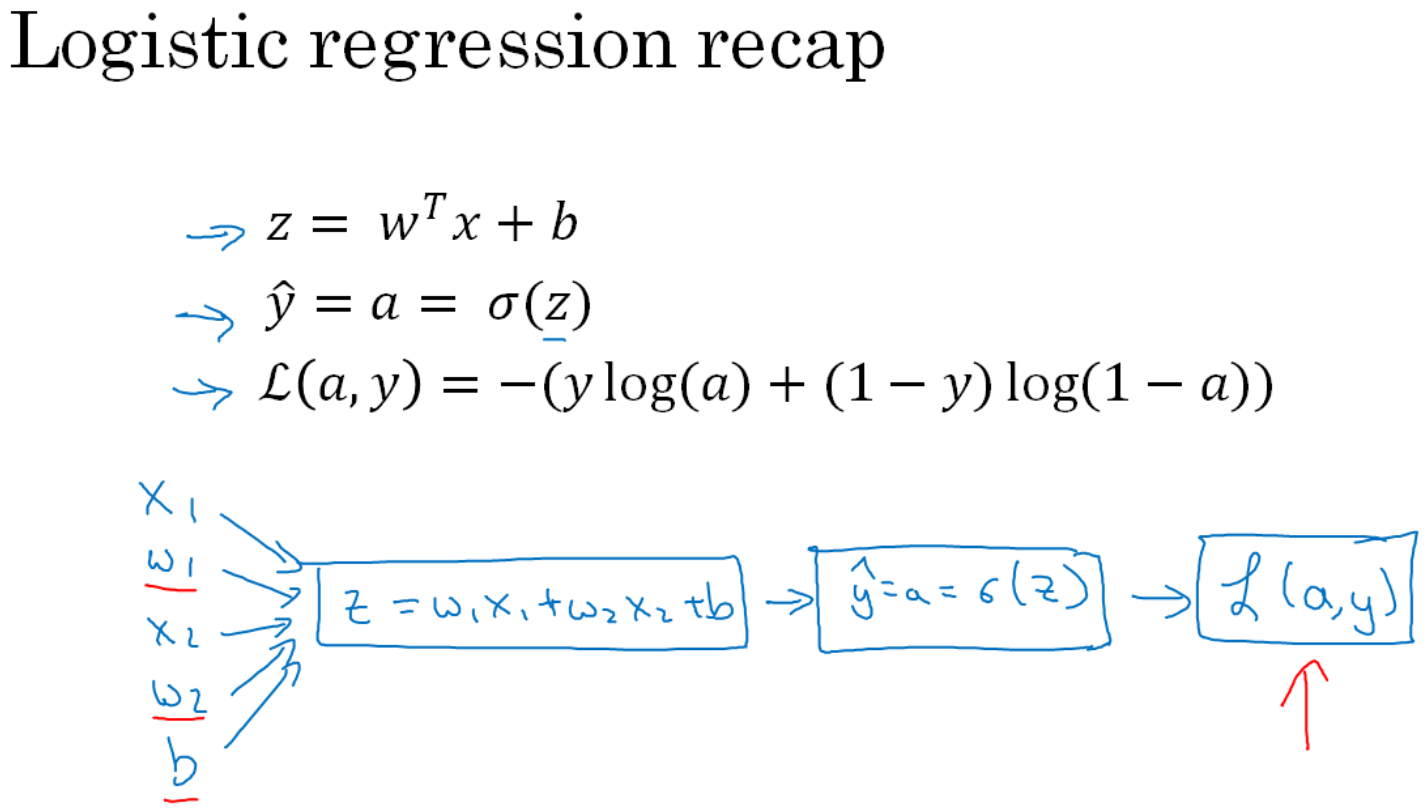


So the key take away from these example is that when computing all the derivatives, the most efficient way is to do is the the right to left computation (red arrows in graph) so first is to compute derivative with respect to v (dJ/dv or dv) and that will help in computing derivative with respect to a and u, and derivative with respect to u(du) is useful in computing derivative with respect to b and c.

Logistic regression gradient descent

In this section we'll talk about how to compute derivatives to be used in calculating gradient descent for logistic regression. The key take way will be what needs to be implement? The key equations which you need to implement gradient descent for logistic regression. We'll use computation graph to implement gradient decent, though using computation graph for computing gradient descent for logistic regression is a bit over kill because this is relatively small task but this usage of computation graph will help in understanding full fledged Neural network.

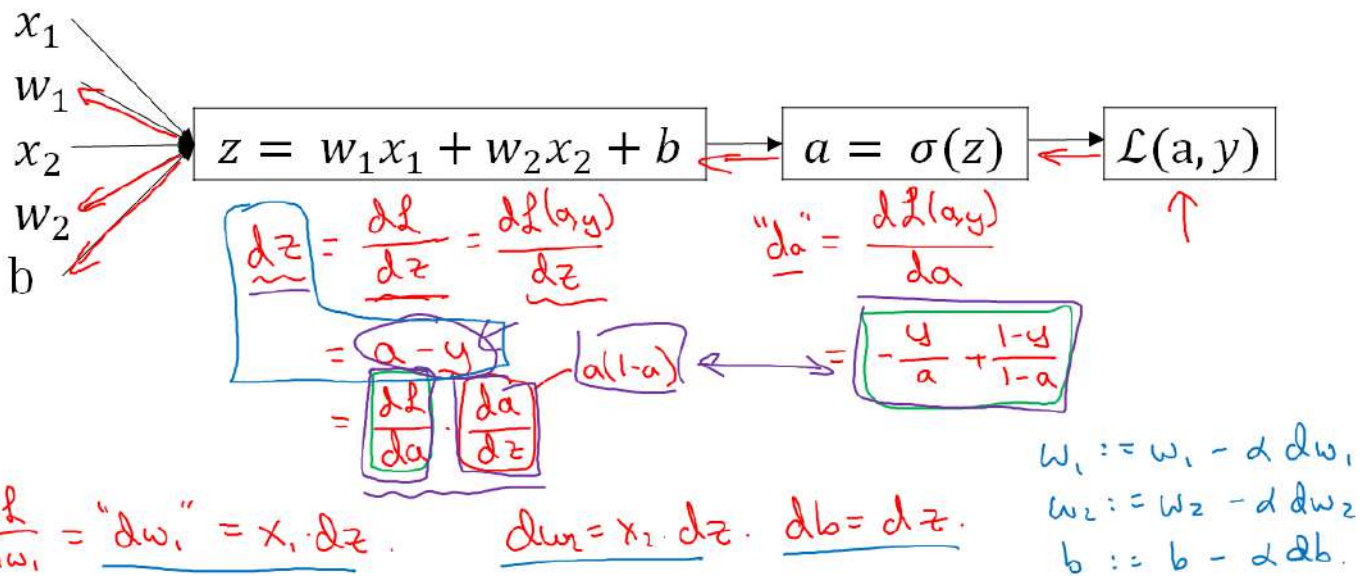
Let's check a diagram first and recap:



As per diagram, to recap, we had set up logistic regression. If we focus on just one example for now, then the loss, or respect to that one example, is defined as follows, where a is the output of logistic regression, and y is the ground truth label. Let's write this out as a computation graph and for this example, let's say we have only two features, x1 and x2. In order to compute z, we'll need to input w1, w2, and b, in addition to the feature values x1, x2. These things, in a computational graph, get used to compute z, which is w1, x1 + w2 x2 + b, rectangular box around that. Then, we compute yhat, or a = Sigmoid(z), that's the next step in the computation graph, and then, finally, we compute the loss, L(a,y). In logistic regression, what we want to do is to modify the parameters, w and b, in order to reduce this loss.

Lets take another diagram:

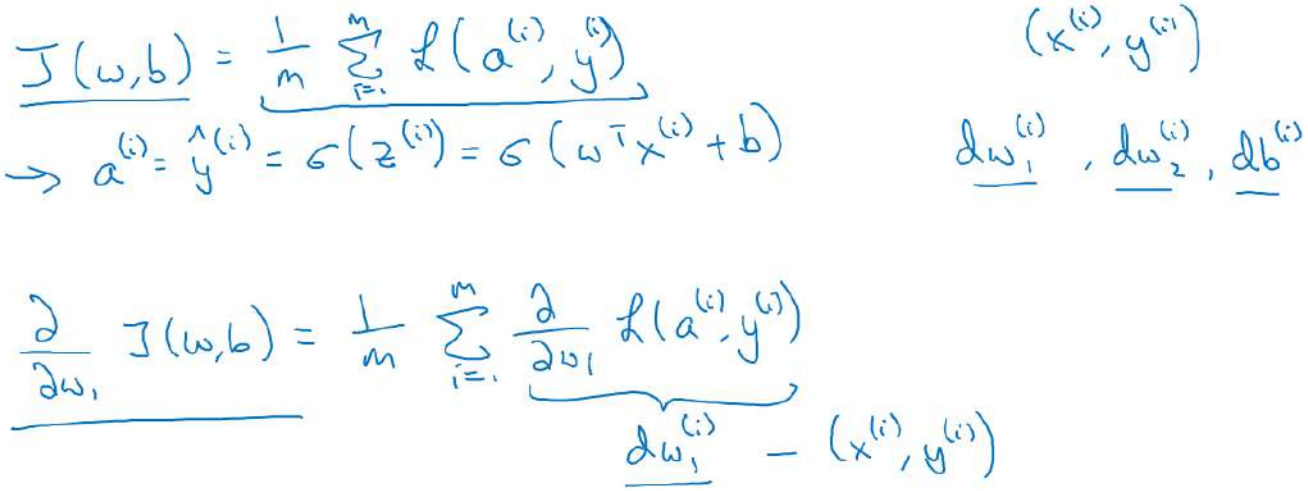
Logistic regression derivatives



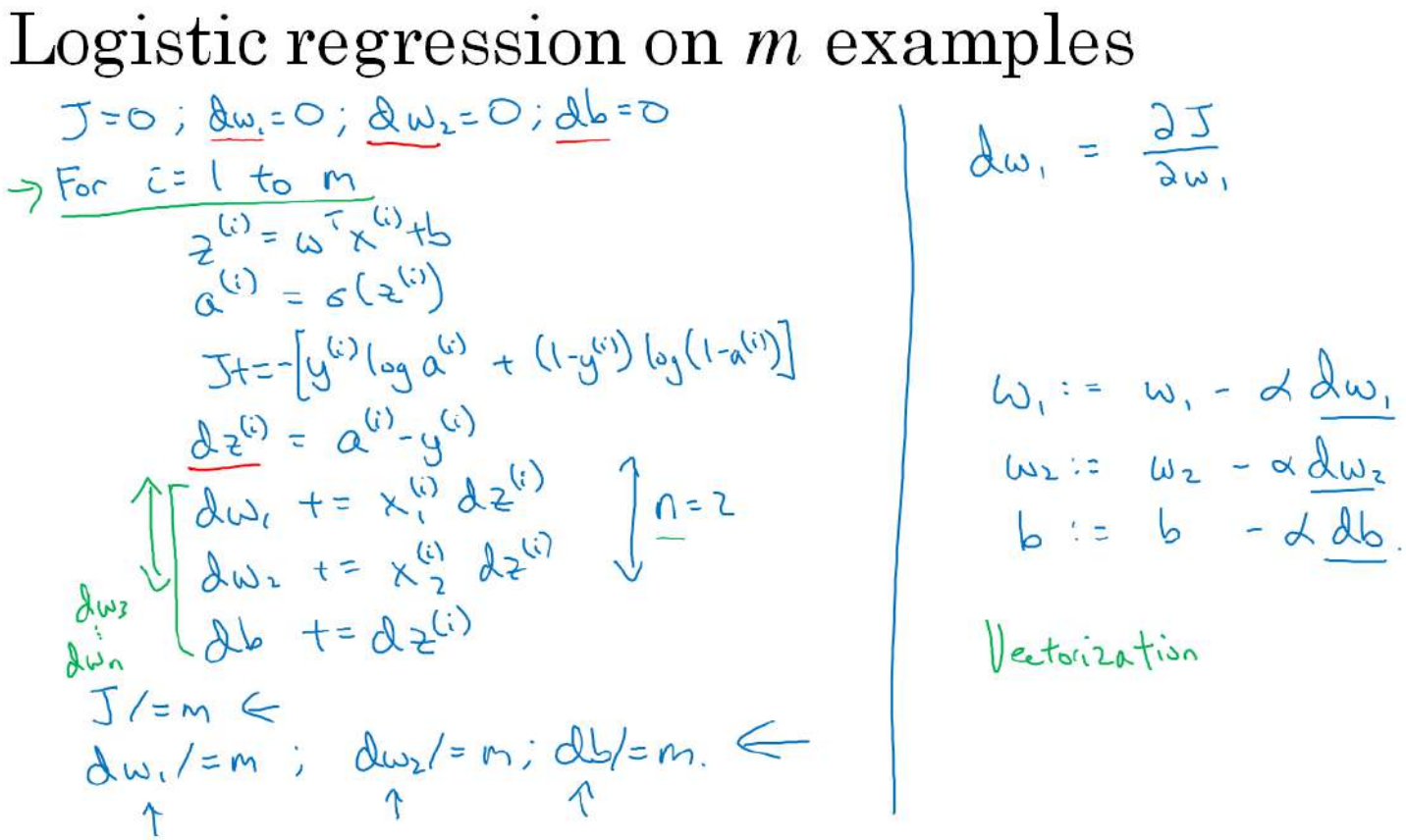
We've described the four propagation steps of how you actually compute the loss on a single training example, now let's talk about how you can go backwards to compute the derivatives. As shown in diagram, what we want to do is compute derivatives with respect to this loss, the first thing we want to do when going backwards is to compute the derivative of this loss with respect to variable a. So, in the Python code, we'll just use **da** to denote this variable. This can be further derived using calculus. I am skipping the intermediate steps which calculates **da**, **dz**. Then, the final step in that computation is to go back to compute how much you need to change w and b. In particular, you can show that the derivative with respect to **w1** and "**dw1**", this is equal to **x1.dz**. Similarly, **"dw2"**, which is how much you want to change **w2**, is **x2.dz** and finally **db** is equal to **dz**. If you want to do gradient descent with respect to just this one example, what you would do is the following; you would use this formula to compute dz, and then use these formulas to compute **dw1**, **dw2**, and db, and then you perform these updates. w1 gets updated as **w1 - learning rate (alpha). dw1** and **w2** gets updated similarly, and b gets set as **b** - **learning rate (alpha). db**. So, this is one step of gradient descent with respect to a single example.

Gradient descent on m examples
Lets see the diagram first:

Logistic regression on m examples



in a previous section we saw how to compute derivatives and implement gradient descent with respect to just one training example now we want to do it for m training examples. Just to recap definition of the cost function (check the first equation in diagram above) from the previous section we know how to compute derivative from a single training example.

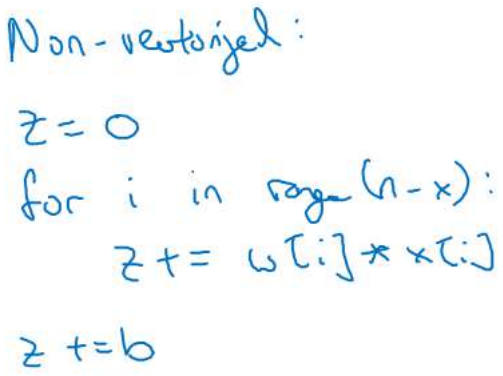


See the diagram above. Let's initialize J equals 0 on dw1=0, dw2=0, db=0 and what we're going to do is use a **for loop** over the training set and compute the derivatives to respect each training example and then add them up. Loop over 1 to M, where M is the number of training examples we compute $z^{(i)} = w^T x + b$, the prediction $a^{(i)} = \text{Sigmoid}(z^{(i)})$
similarly we'll calculate derivatives and other terms in for loop, please note that this calculation has been done just for two features i,e,; n = 2. Everything on the diagram implements just one single step of gradient descent and so you have to repeat everything on this diagram multiple times in order to take multiple steps of gradient descent.

It turns out there are two weaknesses with the calculation as with implemented here which is that to implement logistic regression this way you need to write two for loops the first for loop is a small loop over the M training examples and the second for loop is a for loop over all the features over here, so in this example we just had two features so n equal to 2 but if you have more features you end up writing your dw1, dw2 and so on ...so you need to have a for loop over all n features. So when implementing deep learning algorithms you find that having explicit for loops in your code makes your algorithm run less efficiency and so in the deep learning error would move to a bigger and bigger data sets and so being able to implement your algorithms without using explicit for loops is really important and will help you to scale to much bigger data sets so it turns out that there are set of techniques called vectorization techniques that allows you to get rid of these explicit for loops in your code I think in the pre deep learning era that's before the rise of deep learning vectorization was a nice to have you could sometimes do it to speed a vehicle and sometimes not but in the deep learning era vectorization that is getting rid of for loops like this and like this has become really important because we're more and more training on very large datasets and so you really need your code to be very efficient so in the upcoming sections we'll talk about vectorization and how to implement all this without using even a single **for** loop so of this I hope you have a sense of how to implement logistic regression or gradient descent for logistic regression.

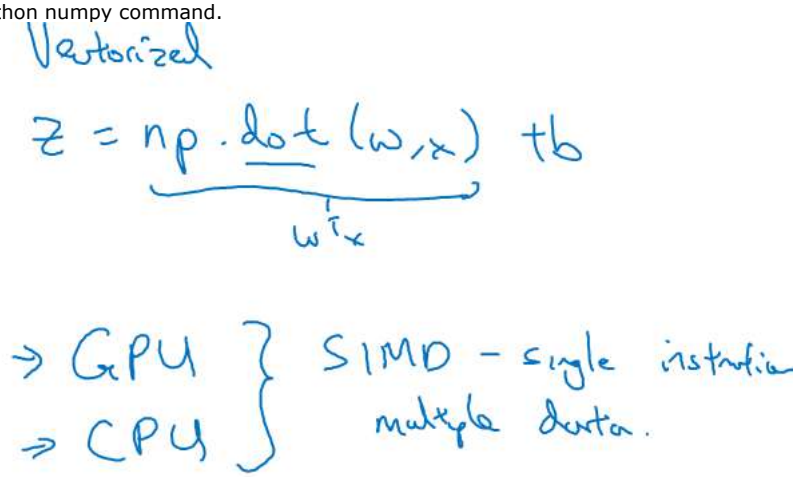
Unit 2 - Python and Vectorization
Vectorization
Vectorization is basically the art of getting rid of explicit for loop in your code. In the deep learning era, you often find yourself training on relatively large data sets, because that's when deep learning algorithms tend to shine. And so, it's important that your code run very quickly because otherwise, if it's running on a big data set, your code might take a long time to run then you just find yourself waiting a very long time to get the result. So in the deep learning era, I think the ability to perform vectorization has become a key skill.

Let's start with an example. So, what is Vectorization? In logistic regression you need to compute $\mathbf{z = w^T x + b}$, where w was this column vector and X is also a vector. These can be very large vectors if you have a lot of features. So, w and x belongs to nx dimensional vectors. So, to compute $\mathbf{w^T x}$, if you had a non-vectorized implementation, you would do something like as shown in diagram below:



So, that's a non-vectorized implementation which you find that that's going to be really slow.

In contrast, a vectorized implementation would just compute $\mathbf{w^T x + b}$ directly. See below diagram for vectorized imlementaion for the same using Python numpy command.



And you can also just add B to that directly. And you find that this is much faster.
Example code:

```
import numpy as np
import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()
print(c)
print("Vectorized version:" + str(1000*(toc-tic)) + "ms")

c = 0
tic = time.time()
for i in range(1000000)
    c += a[i]*b[i]
toc = time.time()
print(c)
print("For loop: Non-Vectorized version:" + str(1000*(toc-tic)) + "ms")
```

output:

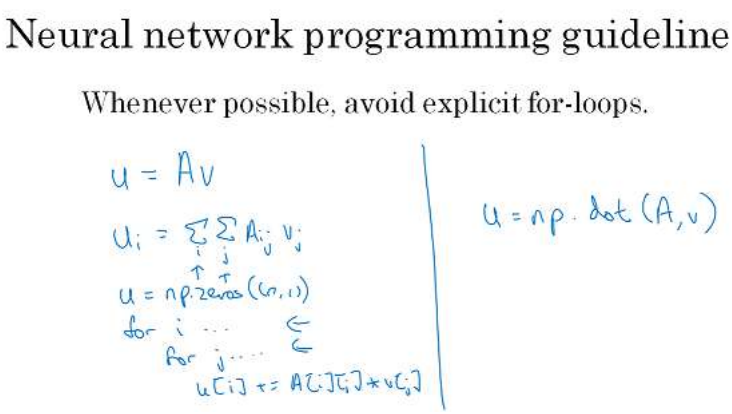
```
250286.989866
Vectorized version: 1.50275523040771484 ms
250286.989866
For loop: Non-Vectorized version: 474.29513931274414 ms
```

When we run the code found that the vectorize version took 1.5 milliseconds but the explicit for loop/non-vectorize version took about 474 milliseconds. The non-vectorize version took something like 300 times longer than the vectorize version. With this example you see that if only you remember to vectorize your code, your code actually runs over 300 times faster. And when you are implementing deep learning algorithms, you can really get a result back faster. Some of you might have heard that a lot of scaleable deep learning implementations are done on a GPU or a graphics processing unit. But all the demos we did just now in the Jupiter notebook where actually on the CPU. And it turns out that both GPU and CPU have parallelization instructions. They're sometimes called SIMD instructions. This stands for a **single instruction multiple data**. But what this basically means is that, if you use built-in functions such as this **np.dot** or other functions that don't require you explicitly implementing a for loop. It enables Python numpy to take much better advantage of parallelism to do your computations much faster. And this is true both computations on CPUs and computations on GPUs. It's just that GPUs are remarkably good at these SIMD calculations but CPU is actually also not too bad at that. Maybe just not as good as GPUs. You're seeing how vectorization can significantly speed up your code. The rule of thumb to remember is whenever possible, avoid using explicit four loops.

More Vectorization Examples

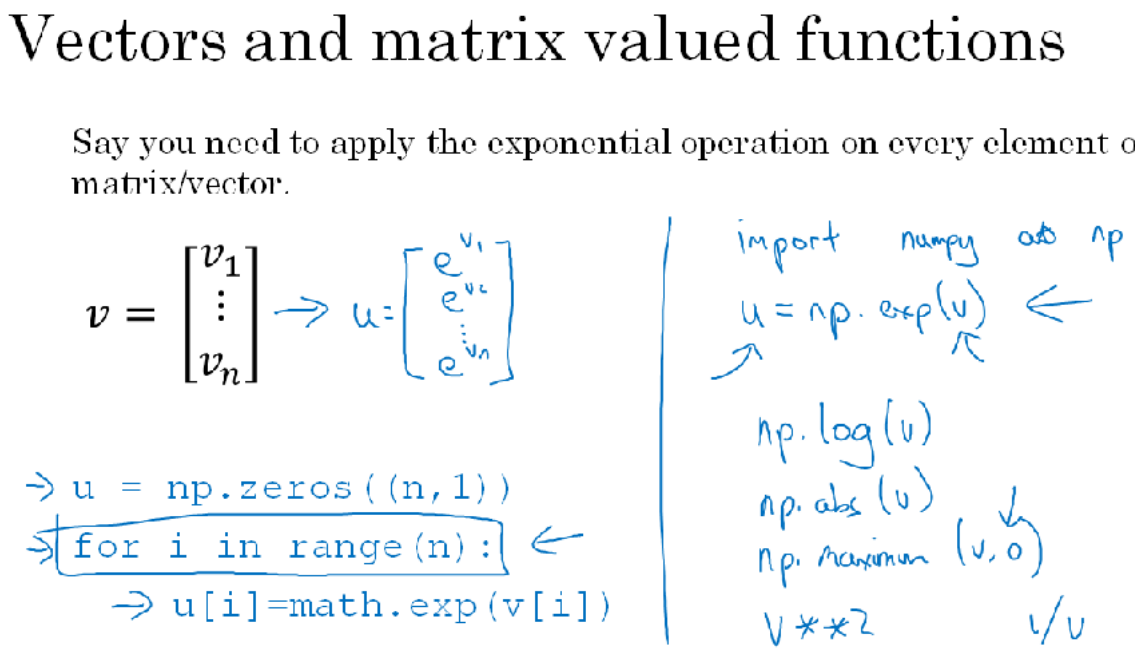
In last section we saw a example of how vectorization, by using built in functions and by avoiding explicit for loops, allows you to speed up your code significantly. Let's look at a few more examples.

The rule of thumb to keep in mind is, when you're programming your NN or when you're programming logistic regression, whenever possible avoid explicit for-loops. And it's not always possible to never use a for-loop, but when you can use a built in function or find some other way to compute whatever you need, you'll often go faster than if you have an explicit for-loop. Let's look at another example. Check the diagram below.



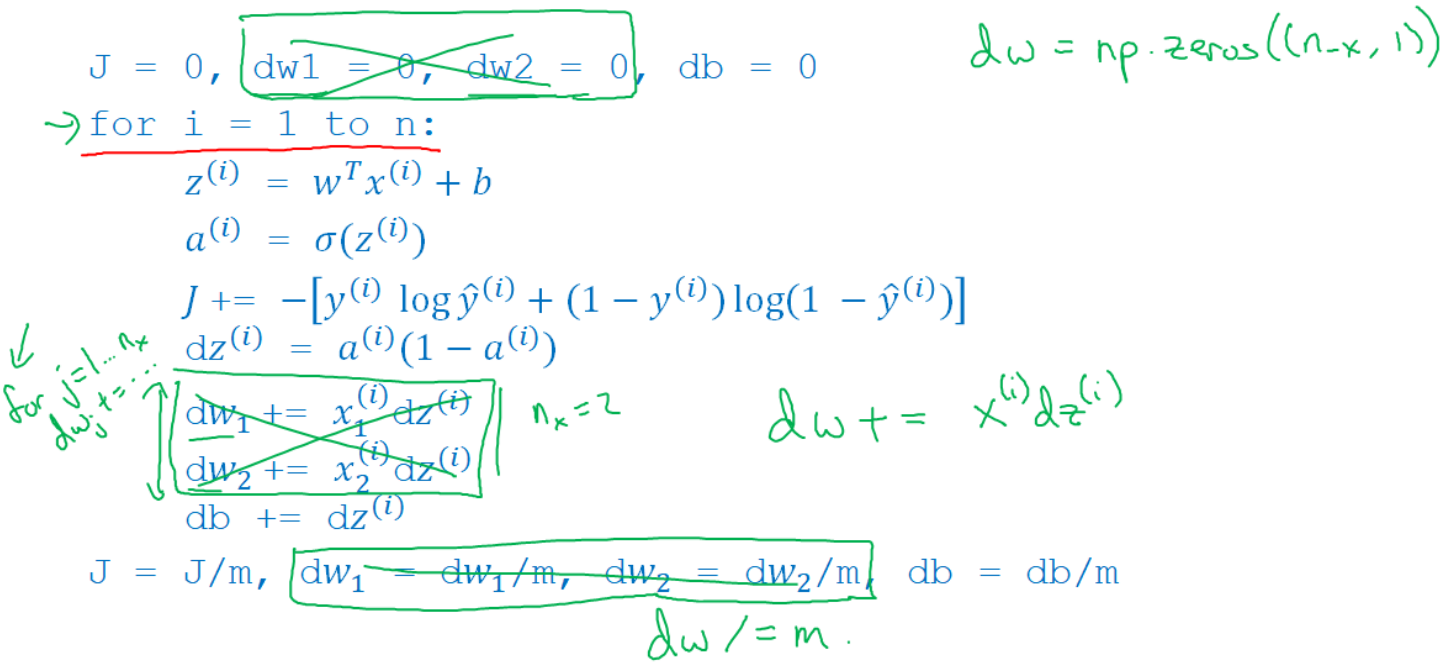
See in diagram a example which computes a vector u as the product of the matrix A, and another vector in non-vectorized implementation we have to use two for-loops, looping over both i and j (see left side of the diagram). In the vectorized implementation we just say u equals np dot (A,v) implementation on the right. So vectorized imple eliminates two different for-loops, and it's going to be way faster.

Let's go through one more example. Let's say you already have a vector, v, in memory and you want to apply the exponential operation on every element of this vector v. See below self explainable diagram of the example.



So, let's take all of these learning and apply it to our logistic regression gradient descent implementation, and see if we can at least get rid of one of the two for-loops we had. Check the diagtam below:

Logistic regression derivatives

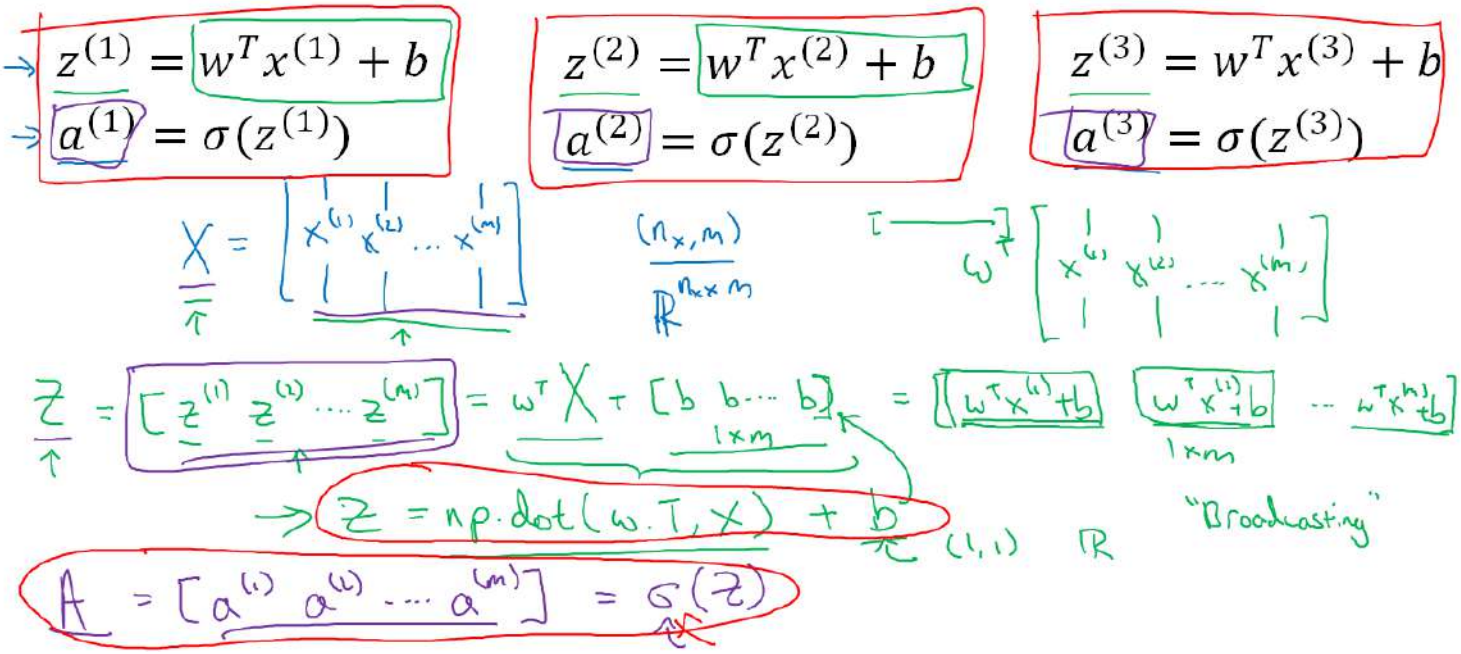


Looking into our code for computing the derivatives for logistic regression, and we had two for-loops. So in our example we had nx equals 2, but if you had more features than just 2 features then you'd need have a for-loop over dw1, dw2, dw3, and so on. Here instead of second for loop over the individual components, we'll just use this vector value operation (check diagram). We still have this one for-loop that loops over the individual training examples.

Vectorizing Logistic Regression

In this section we'll talk about how you can vectorize the implementation of logistic regression, so they can process an entire training set, that is implement a single elevation of grading descent with respect to an entire training set without using even a single explicit for loop.
First check the diagram:

Vectorizing Logistic Regression

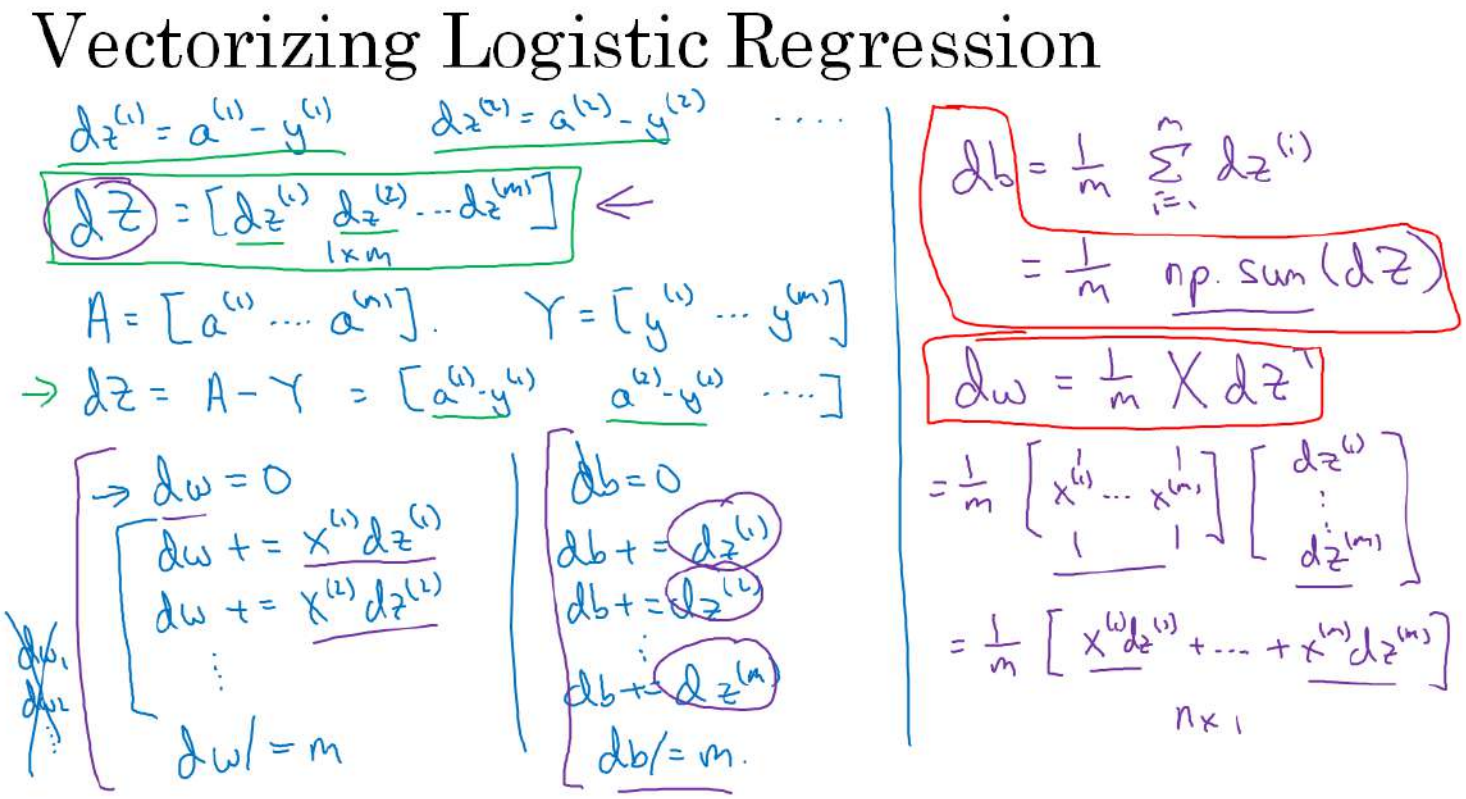


Let's first examine the forward propagation steps of logistic regression. So, if you have M training examples, then to make a prediction on the first example, you need to compute compute z, then compute the activations. Then to make a prediction on the second training example, third example and so on (check top row of the diagram), and you might need to do this M times, if you have M training examples. So, it turns out, that in order to carry out the forward propagation step, that is to compute these predictions on our M training examples, there is a way to do so, without needing an explicit for loop. So just to recap, what we've seen on this diagram is that instead of needing to loop over M training examples to compute z and a, one at a time, you can implement **Z = np.dot(w.T, x) + b** which is just one line of code to compute all these z's at the same time and then **A=**

[a(1) a(2) .... a(m)] = Sigma(z) to compute all the a's all at the same time. So this is how you implement a vectorize implementation of the forward propagation for all M training examples at the same time. So to summarize, we've just seen how you can use vectorization to very efficiently compute all of the activations, all the a's at the same time.

Vectorizing Logistic Regression's Gradient decent

In this section we'll see how we can use vectorization to perform the gradient computations for all M training examples. Again, all sort of at the same time subsequently we'll put it all together and see how one can derive a very efficient implementation of logistic regression. Check the diagram first:



In this diagram we can see that left half is the non-vectorized implementation which uses for loops and right half is the vectorized implementation without using a single for loop. So to summarize, we've just done forward propagation and back propagation, computing the predictions and computing the derivatives on all M training examples without using a for loop.This is a single iteration of gradient descent for logistic regression still a outer for loop will be needed if we want to implement multiple iterations of the gradient descent but one iteration of gradient descent logistic regression can be implemented without a single for loop which is a remarkable thing.
See the diagram below for final code:

Implementing Logistic Regression

$J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$

for i = 1 to m:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J\ += -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1\ += x_1^{(i)} dz^{(i)}$

$dw_2\ += x_2^{(i)} dz^{(i)}$

$db\ += dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m$

$db = db/m$

for iter in range(1000):

$Z = w^T X + b$

$= np.dot(w.T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$dw = \frac{1}{m} X dZ^T$

$db = \frac{1}{m} np.sum(dZ)$

$w := w - \alpha dw$

$b := b - \alpha db$

Broadcasting in Python

Broadcasting is another technique that you can use to make your Python code run faster. In this section, let's delve into how broadcasting in Python actually works. Let's try to understand broadcasting with an example. See below diagram first:

Broadcasting example

Calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

= A (3,4)

59 cal

$\frac{56}{59} \approx 94.9\%$

Calculate % of calories from Carb, Protein, Fat. Can you do this without explicit for-loop?

```
cal = A.sum(axis = 0)
percentage = 100*A/(cal.reshape(1, 4))
```

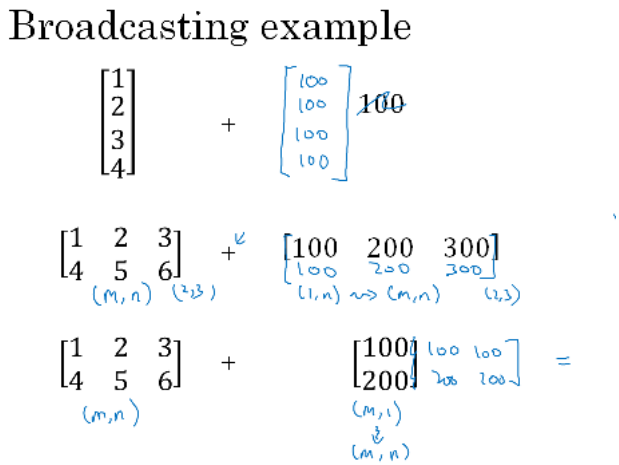
(3,4) / (1,4)

In this matrix, we've shown the number of calories from carbohydrates, proteins, and fats in 100 grams of four different foods. So for example, a 100 grams of apples turns out, has 56 calories from carbs, and much less from proteins and fats. Whereas, in contrast, a 100 grams of beef has 104 calories from protein and 135 calories from fat. Now, let's say your goal is to calculate the percentage of calories from carbs, proteins and fats for each of the four foods. So, for example, if you look at **Apple** column and add up the numbers in that column you get that 100 grams of apple has **56 + 1.2 + 1.8 = 59 calories**. And so as a percentage the percentage of calories from carbohydrates in an apple would be 56/59, that's about 94.9%. So most of the calories in an apple come from carbs, whereas in contrast, most of the calories of beef come from protein and fat and so on. So the calculation you want is really to sum up each of the four columns of this matrix to get the total number of calories in 100 grams of apples, beef,

eggs, and potatoes. And then to divide throughout the matrix, so as to get the percentage of calories from carbs, proteins and fats for each of the four foods. So the question is, can you do this without an explicit for-loop?
Let's take a look at how you could do that. What I'm going to do is show you how you can set, say this matrix A = 3x4. And then with one line of Python code we're going to sum down the columns. So we're going to get four numbers corresponding to the total number of calories in these four different types of foods, 100 grams of these four different types of foods. And I'm going to use a second line of Python code to divide each of the four columns by their corresponding sum. Here are the two lines of Python code.

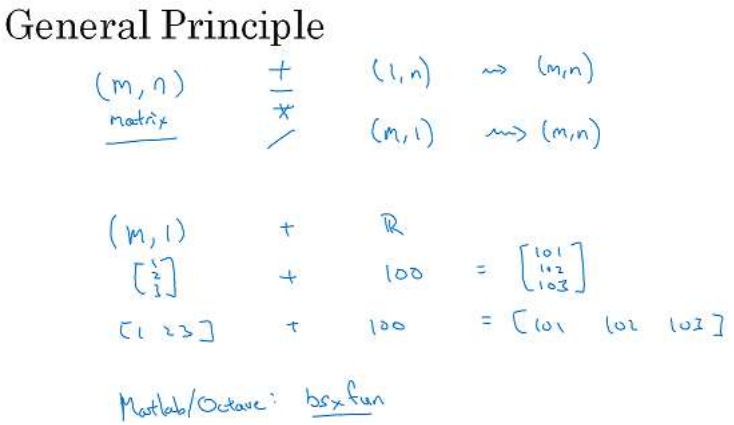
```
import numpy as np
A = np.array([56.0,0.0,4.4,68.0],
             [1.2,104.0,52.0,8.0],
             [1.8,135.0,99.0,0.9])
print (A)
cal = A.sum(axis=0)
print (cal)
percentage = 100*A/cal/reshape(1,4)
print (percentage)
```

And then let's print percentage.
Let's run that. And so that command we've taken the matrix A and divided it by 1x4 matrix. And this gives us the matrix of percentages. So as we worked out kind of by hand just now in the apple there was a first column 94.9% of the calories are from carbs. Let's go back to the slides. To add a bit of detail the parameter in diagram, (axis = 0), means that you want Python to sum vertically. So if this is axis 0 this means to sum vertically, where as the horizontal axis is axis 1. So be able to write axis 1 or sum horizontally instead of sum vertically. And then **A/cal/reshape(1,4)** command is an example of Python broadcasting where you take a matrix A. So this is a three by four matrix and you divide it by a one by four matrix. And technically, after this first line of codes cal, the variable cal, is already a one by four matrix. So technically you don't need to call reshape here again, so that's actually a little bit redundant. But when we are writing Python code and if we not entirely sure what matrix, whether the dimensions of a matrix, it's good idea to just call a reshape command to make sure that it's the a column vector or a row vector or whatever you want it to be. The reshape command is a constant time. It's a order one operation that's very cheap to call so there is no overhead so encourage to use. It will help in making sure that the matrices are the size you need it to be.
Now, let's explain in greater detail how this type of operation works? We had a 3x4 matrix and we divided it by a 1x4 matrix. So, how can you divide a 3x4 matrix by a 1x4 matrix? Or by 1x4 vector?
Let's go through a few more examples of broadcasting. If you take a 4x1 vector and add it to a number, what Python will do is take this number and auto-expand it into a 4x1 vector as well. Check the diagram:



And so the vector of the left in the first row plus the number 100 ends up a new vector which will have elements 101,102,103,104. So we've are adding a 100 to every element, and in fact we use this form of broadcasting where that constant was the parameter b. And this type of broadcasting works with both column vectors and row vectors, and in fact we use a similar form of broadcasting earlier with the constant we're adding to a vector being the parameter b in logistic regression.

Check the diagram:



So the general case would be if you have some (m,n) matrix here and you add it to a (1,n) matrix. What Python will do is copy the matrix m, times to turn this into mxn matrix. Check the above diagram for more examples.

Here's the more general principle of broadcasting in Python. If you have an (m,n) matrix and you add or subtract or multiply or divide with a (1,n) matrix, then this will copy it n times into an (m,n) matrix. And then apply the addition, subtraction, and multiplication of division element wise. The fully general version of broadcasting can do even a little bit more than this. One can read the documentation for NumPy, and look at broadcasting in that documentation. That gives an even slightly more general definition of broadcasting.

Week 3: Shallow Neural Networks

Learning Objectives

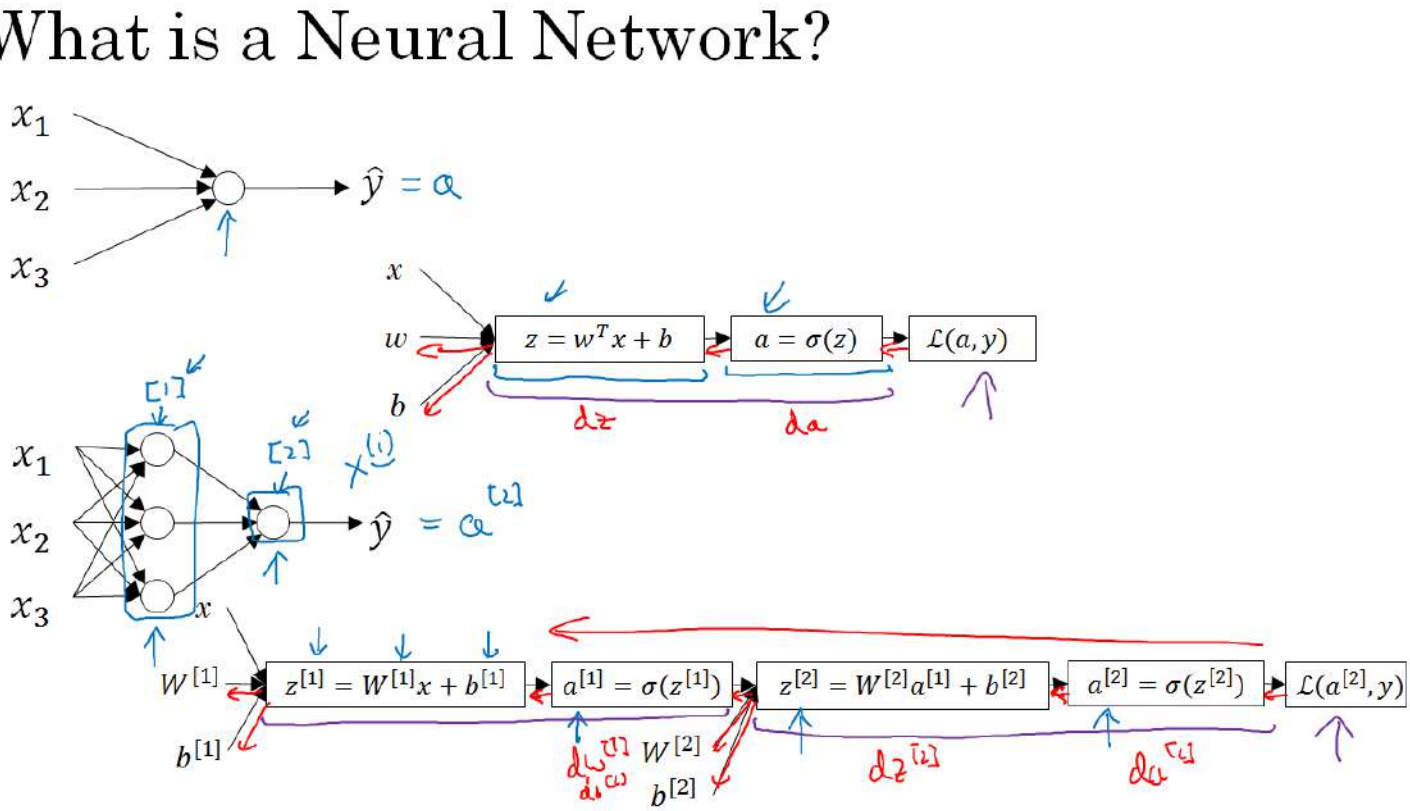
- Understand hidden units and hidden layers
- Be able to apply a variety of activation functions in a neural network.
- Build your first forward and backward propagation with a hidden layer
- Apply random initialization to your neural network
- Become fluent with Deep Learning notations and Neural Network Representations
- Build and train a neural network with one hidden layer.

The general methodology to build a Neural Network is to:

1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
- Implement forward propagation
- Compute loss
- Implement backward propagation to get the gradients
- Update parameters (gradient descent)

Neural Network Overview

Let's see the diagram first....

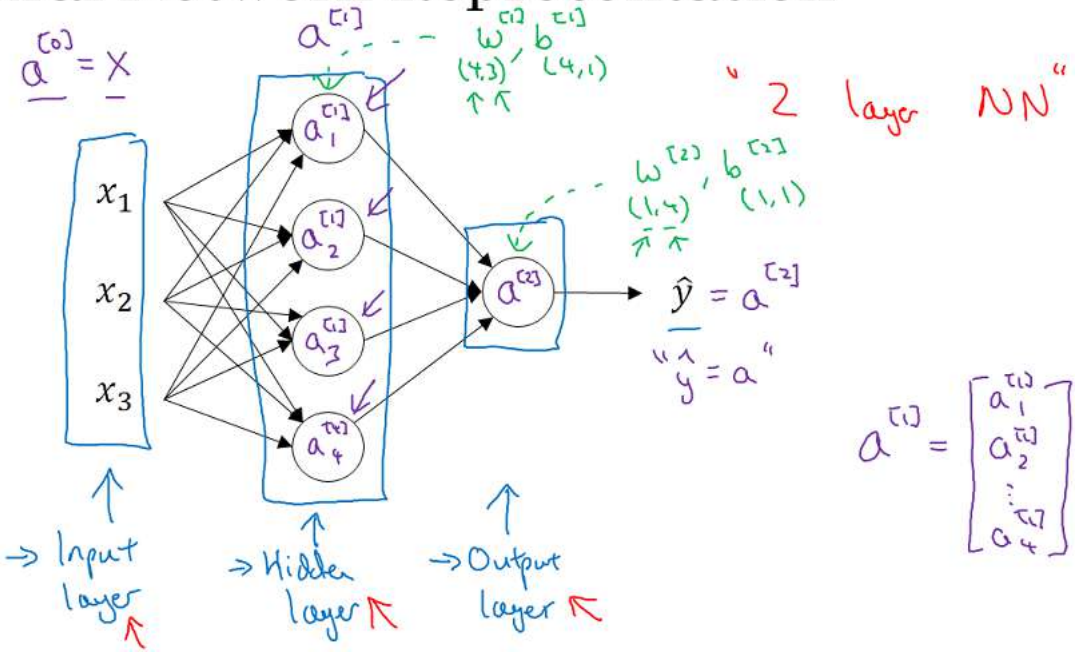


This diagram shows a quick overview of how you implement neural network. In last section we had talked about logistic regression and we saw how this model corresponds to the computation graph (as shown in diagram) where you put the input features **x** and parameters **w** and **b**, that allows you to compute **z** which is then used to compute **a** and with output **Yhat** and compute the loss function. A neural network is looks like as shown the bottom half of the above diagram which has been formed by stacking together a lot of little sigmoid units. A new notation is introduced which is a superscript square bracket e.g.; $^{[1]}$, 1 refers to quantities associated with this stack of nodes called a layer, similarly we'll use superscript square bracket 2 e.g.; $^{[2]}$ to refer to another layer of the network. One should not confuse the superscript square brackets with the superscript round brackets which we used to refer to individual training examples. The key intuition to take away is that whereas for logistic regression we had **z** followed by **a** calculation and this neural network here we just do it multiple times z followed by **a** calculation and then you finally compute the loss at the end. Just to recall that for the logistic regression we have also backward calculation in order to compute derivatives so in the same way in a neural network also we'll end up doing a backward calculation (check diagram depicted with red arrows) to compute derivatives .

Neural Network Representation- One hidden layer neural network

In this section, we'll first talk about a single hidden layer.

Neural Network Representation



See the diagram below:

Let's have a look into different parts of the diagram. First, we have the input features, x1, x2, x3 stacked up vertically and this is called the **input layer** of the neural network. So maybe not surprisingly, this contains the inputs to the neural network. Then there's another layer of circles and this is called a **hidden layer** of the neural network. The final layer in this case is formed by is just one node and this single-node layer is called the **output layer**, and is responsible for generating the predicted value **yhat**. In a neural network the training supervised learning, the training set contains values of the inputs **x** as well as the target outputs **y**. So the term hidden layer refers to the fact that the in the training set, the true values for this node in the middle are not observed. So actually we don't see what they should be in the training set. We see what the inputs are and we see what the output should be but the things in the hidden layer are not seen in the training set. So that kind of explains the name hidden there just because we don't see it in the training set.

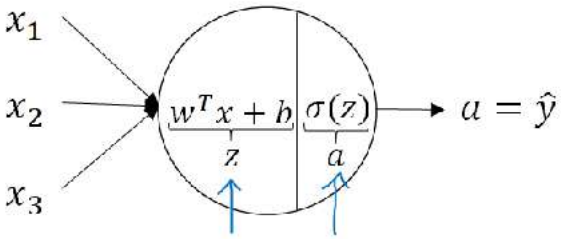
Let's introduce few more notation. Previously, we were using the vector x to denote the input features and the alternative notation for the values of the input features will be $\mathbf{a^{[0]}}$ and the term **a** also stands for activations and it refers to the values that different layers of the neural network are passing on to the subsequent layers. So the input layer passes on the value **x** to the hidden layer, so we're going to call that activations of the input layer $\mathbf{a^{[0]}}$. The next layer, the hidden layer will in turn generate some set of activations which can be written as $\mathbf{a^{[1]}}$ So in particular in this first unit /node, we generate a value $\mathbf{a1^{[1]}}$,the second node it is $\mathbf{a2^{[1]}}$ and so on. So $\mathbf{a^{[1]}}$ is a 4x1 dim vector, this is 4 dimensional because in this case we have four nodes or four units or four hidden units in this hidden layer and then finally, the output layer regenerates some value $\mathbf{a^{[2]}}$ which is just a row number. So we have **yhat = $\mathbf{a^{[2]}}$** . As we have seen in logistic regression **yhat = a** as we only had that one output layer, so we don't use the superscript square brackets **[]** on **a** but with the neural network we now going to use the superscript square bracket **[]** to explicitly indicate which layer it came from.

One funny thing about notational conventions in neural networks is that the NN we have seen in the above diagram is a **two layer neural network** this is because of the reason that in when we count layers in neural networks, we don't count the input layer. So the hidden layer is **layer 1** and the output layer is **layer 2**. In our notational convention, we're calling the input layer **layer 0**, so technically there are 3 layers in this neural network, because there's the **input layer**, the **hidden layer**, and the **output layer** but in conventional usage and also in research papers this is regarded as 2 layer neural network because we don't count the input layer as an official layer.
Finally, the **hidden layer** and the **output layers** will have parameters associated with them. So the **hidden layer** have parameters w and b as the associated parameters we can write them $w^{[1]}$ and $b^{[1]}$ to indicate that these are parameters associated with **layer 1** of the hidden layer. In this NN, **w** will be a **4x3** matrix and b will be a **4x1** vector. Where the first coordinate 4 comes from the fact that we have 4 nodes of hidden layer, and 3 comes from the fact that we have 3 input features. Similarly the output layer also have parameters $w^{[2]}$ and $b^{[1]}$ and it turns out the dimensions of these are 1 x4 (because the hidden layer has four hidden units, the output layer has just one unit) and 1 x1. So we've just seen what a two layered neural network looks like. That is a neural network with one hidden layer.

Computing a Neural Network's output

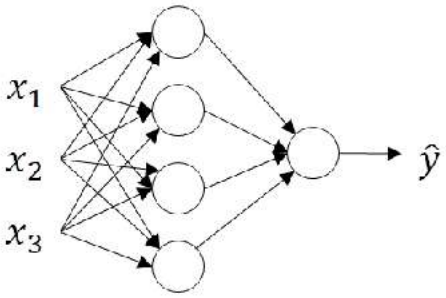
in the last section we saw what a single hidden layer neural network looks like. In this section we'll go through the details of exactly how this neural network compute outputs. This is a 2-steps of computation first you compute z and in second you compute the activation as a sigmoid function of z. Check the diagram below:

Neural Network Representation

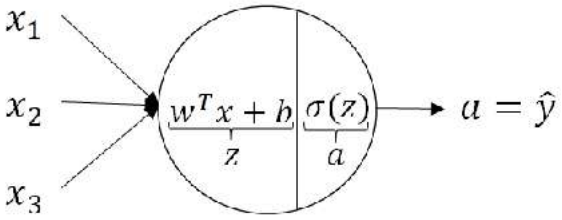


$z = w^T x + b$

$a = \sigma(z)$

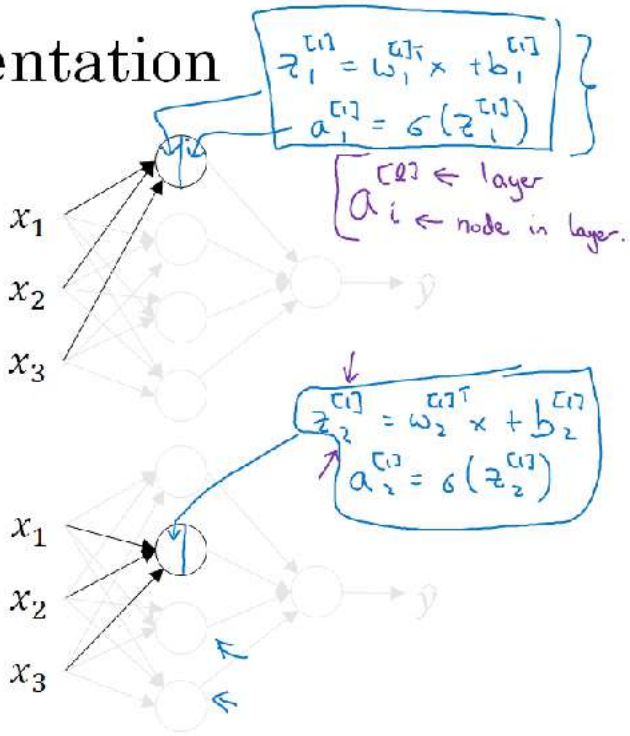


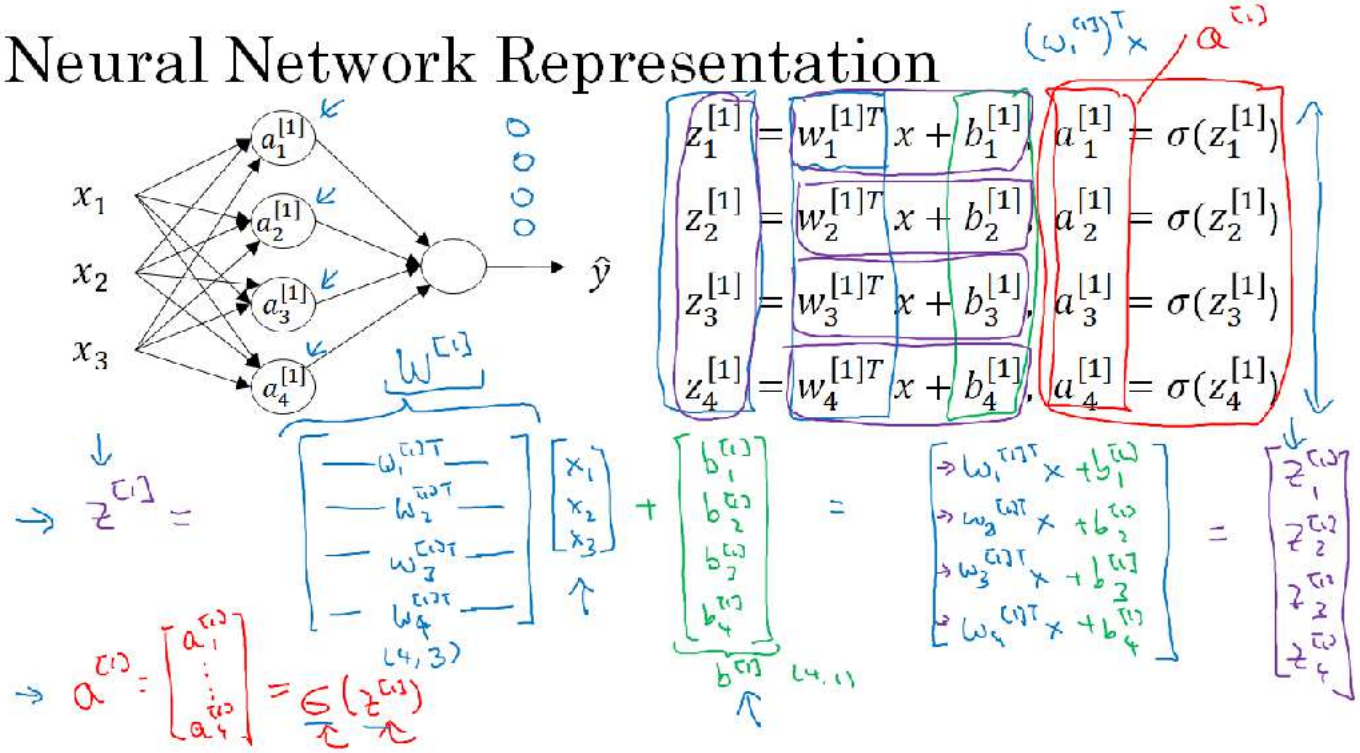
Neural Network Representation

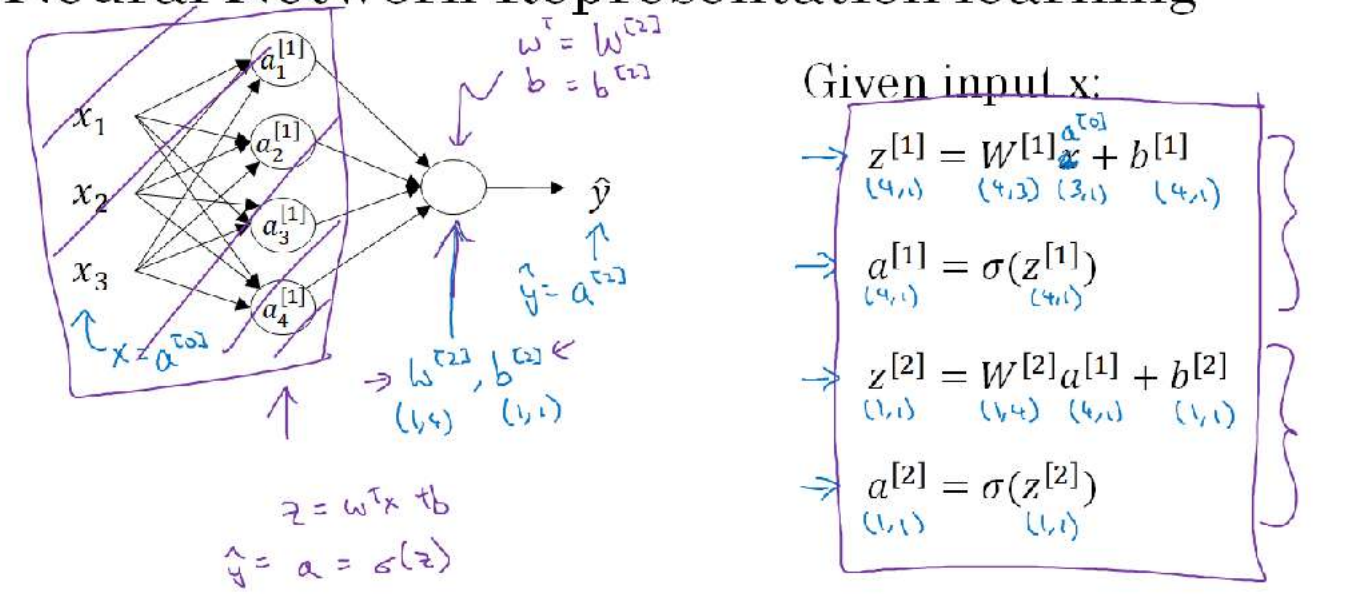


$z = w^T x + b$

$a = \sigma(z)$





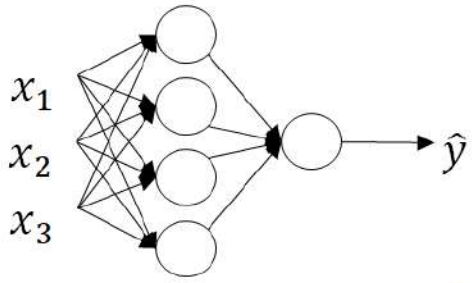


So looking into diagram, we can see that to compute the output of this neural network all you need is 4 four lines of code (right side the last image)

Vectorized across multiple examples

In the last section, we saw how to compute the prediction on a neural network, given a single training example. In this section we'll see how to vectorize across multiple training example and the outcome will be quite similar to we saw for logistic regression. See the diagram with the equations from the last section.

Vectorizing across multiple examples



$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

$X \rightarrow a^{[2]} = \hat{y}$

$x^{(1)} \rightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \rightarrow a^{[2](2)} = \hat{y}^{(2)}$

$x^{(m)} \rightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](i)}$: a — activations, $[2]$ — layer 2, (i) — example i

for $i = 1$ to m,

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Adding in notation, the round bracket i $^{(i)}$refers to training example i, and the square bracket 2 $^{[2]}$refers to layer 2
Looking into the above diagrams we see equation to implement NN with vectorization, that is vectorization across multiple examples.

Vectorizing across multiple examples

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m)

$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$

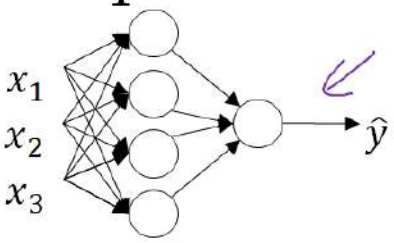
$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$

training examples

hidden units.

Looking into all the diagrams for vectorized implementation see below the recap diagram

Recap of vectorizing across multiple examples



$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

```
for i = 1 to m
```

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

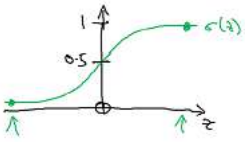
$A^{[0]} \quad X = a^{[0]} \quad x^{(i)} = a^{[0](i)}$

$$Z^{[1]} = W^{[1]} X + b^{[1]} \leftarrow W^{[1]} A^{[0]} + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

Looking into the diagram and equations it shows that that the different layers of a neural network are roughly doing the same thing or just doing the same computation over and over, here we have two-layer neural network. If you go even deeper into neural networks they are basically taking these two steps and just doing them even more times than we have observed in 2 layer NN. So that's how we can vectorize our neural network across multiple training examples.

Activation functions

when you build a neural network one of the choices you get to make is what activation functions to use in the hidden layers as well as at the output unit of your neural network so far we've just been using the **sigmoid** activation function but sometimes other choices can work much better let's take a look at some of the options in the forward propagation steps for the neural network we have steps like **$a^{[1]}$ = sigmoid($z^{[1]}$)** and **$a^{[2]}$ = sigmoid($z^{[2]}$)** these two steps where we use the **sigmoid** function that is called an activation function as we have seen before it looks like

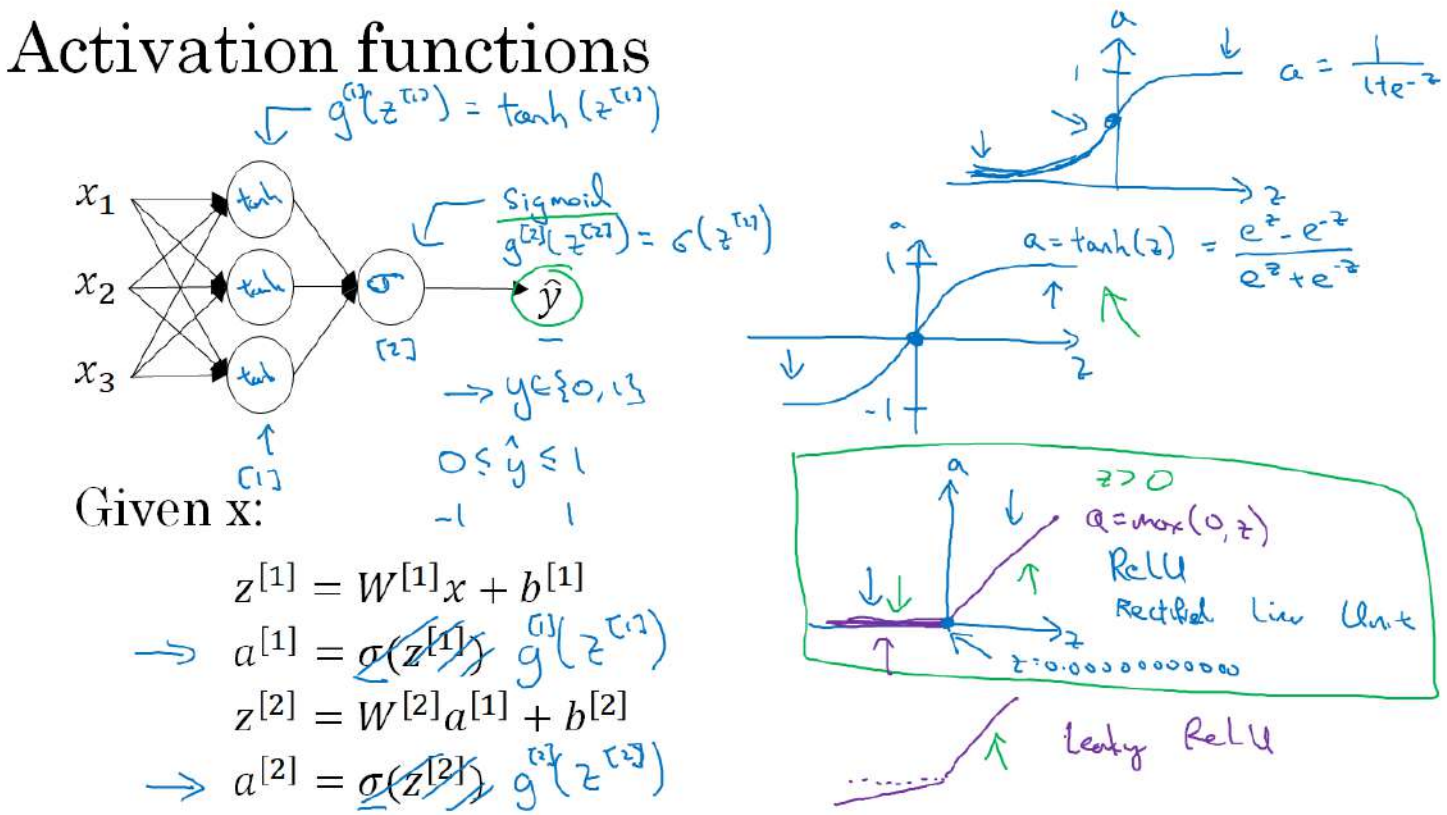


So in more general case we can have a different function **g** of **z** where **g** could be a nonlinear function that may not be the **sigmoid** function. As we have seen **sigmoid** function goes between 0 & 1, an activation function that almost always works better than the **sigmoid** function is the **tanh** function or the **hyperbolic tangent** function we can write as **a = tanh(z)** of **z** which goes between - 1 & +1. Mathematically, **tanh** is a shifted version of the **sigmoid** function which now crosses 0 and rescale so that it goes between -1 and +1. It turns out that for hidden units if you let the function **g($z^{[1]}$) = tanh($z^{[1]}$)** this almost always works better than the **sigmoid** function because with values between +1 and -1, the mean of the activations that come out of your hidden layer are closer to having a zero mean and so just as sometimes when you train a learning algorithm you might centered the data and have your data have zero mean using a **tanh** instead of a **sigmoid** function so kind of has the effect of centering your data so that the mean of the data is close to the zero rather than maybe a 0.5 and this actually makes learning for the next layer a little bit easier.

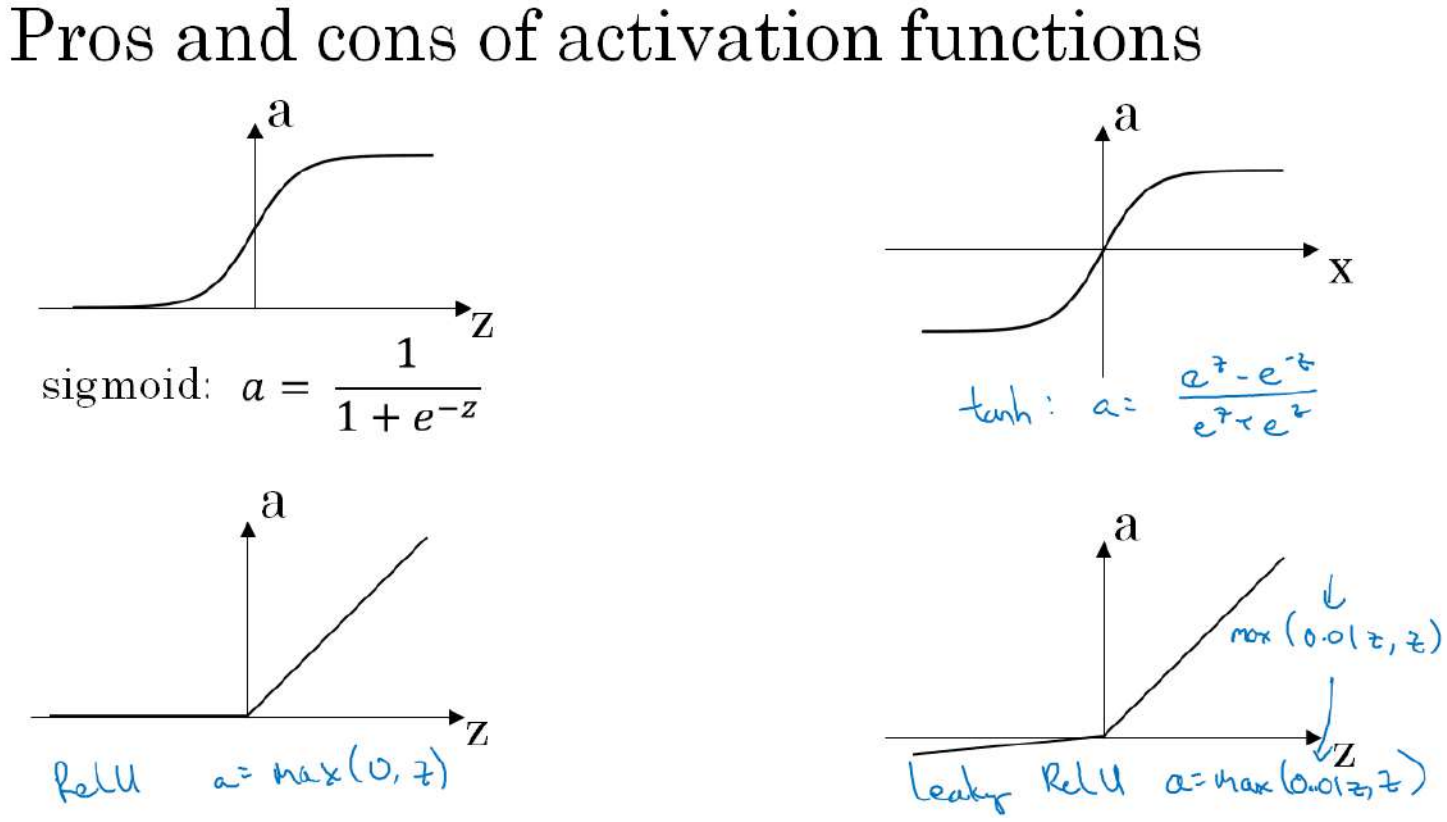
Takeaway from the current understanding is that no need to use the **sigmoid** activation as **tanh** function is almost always strictly superior the one exception is for the output layer because if **y** is either 0 or 1 then it makes sense for **yhat** to be a number that you want to output between 0 and 1 rather than between -1 and 1 so the one exception where one can use the **sigmoid** activation function is when you're using binary classification in which case you might use the **sigmoid** activation function for the output layer like **g($z^{[2]}$) = tanh($z^{[2]}$)** in the 2 layered example. So what we see in this example is where you might have a **tanh** activation function for the hidden layer and **sigmoid** for the output layer so the activation functions can be different for different layers and sometimes to denote that the activation functions are different for different layers we might use these square brackets super scripts like **$g^{[1]}$($z^{[1]}$) = tanh($z^{[1]}$)**) for hidden layer and **$g^{[2]}$($z^{[1]}$) = sigmoid($z^{[1]}$)**) for the output layer. One of the downsides of both the **sigmoid** function and the **tanh** function is that if **z** is either very large or very small then the gradient of the derivative or the slope of this function becomes very small so **z** is very large or z is very small the slope of the function ends up being close to zero and so this can slow down gradient descent. So one of the choice that is very popular in machine learning is what's called the **rectified linear unit** so the **Relu** function **a = max(0,x)** so the derivative is 1 as long as z is +ve and derivative or the slope is 0 when Z is -ve. If we implement this technically the derivative when **z** is exactly 0 is not well-defined but when you implement is in the computer the often you get exactly the equals 0.000000000 it is very small so you don't need to worry about it in practice you could pretend a derivative when **z** is equal to 0 you can pretend is either 1 or 0. So here are some rules of thumb for choosing activation functions if your output is **0/1** value if you're I'm using binary classification then the **sigmoid** activation function is very natural for the upper layer and then for all other units **Relu** or the **rectified linear unit** is increasingly the default choice of activation function so if you're not sure what to use then just use the **Relu** activation function. One disadvantage of the **Relu** is that the derivative is equal to zero when **z** is negative in practice this works just fine but there is another version of the **Relu** called the **Leaky Relu** instead of it being zero when **z** is negative it just takes a slight slope this usually works better than the **Relu** activation function although it's just not used as much in practice either one should be fine although if you had to pick **Relu** is fine, one of the advantage of both the **Relu and Leaky Relu** is that for a lot of

the space of z (between z and Relu) the derivative of the activation function or the slope of the activation function is very different from zero and so in practice using the **Relu** activation function our neural network will often learn much faster than using the **tanh** or the **sigmoid** activation function.

See the diagram below of all we have learned about different activation functions:



Let's just quickly recap there are pros and cons of different activation functions. See the diagram below:

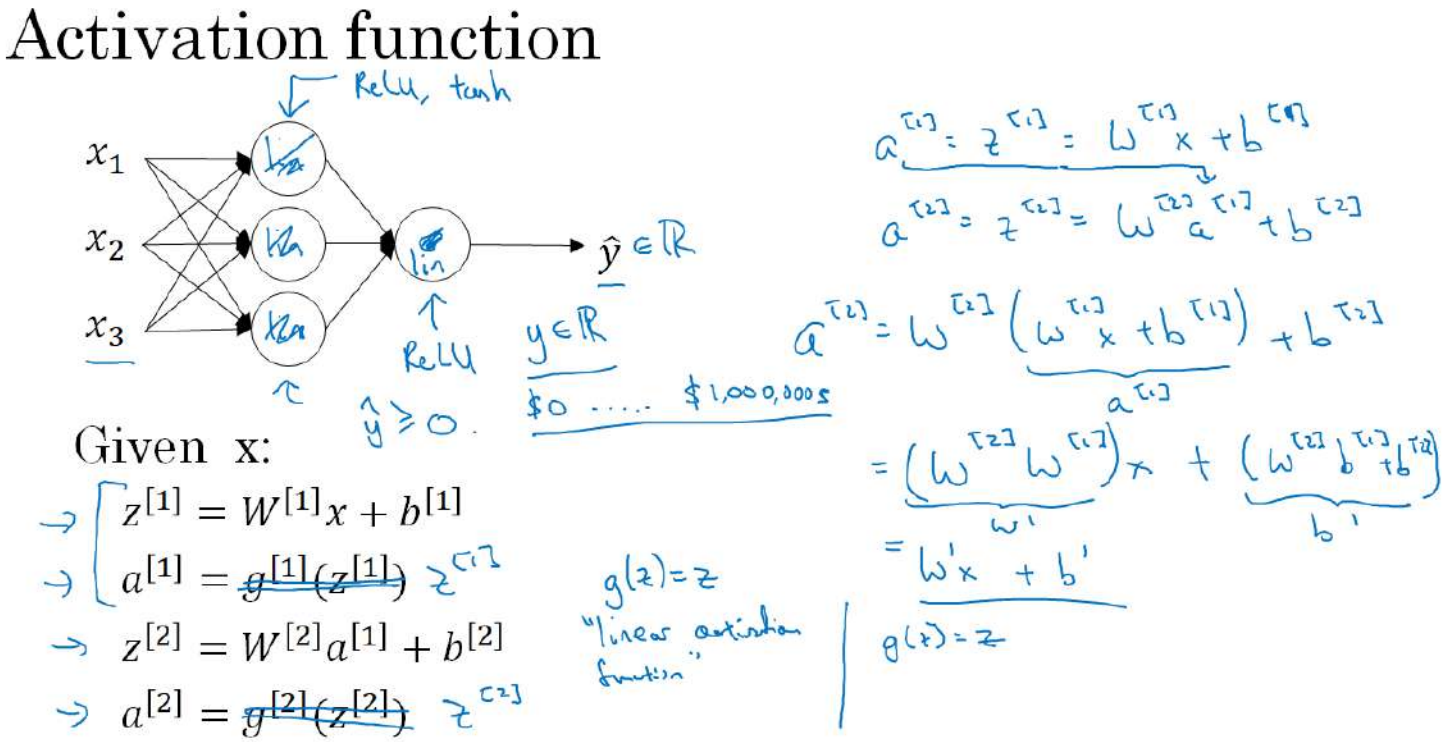


First is the **sigmoid** activation function, never use this except for the **output layer** if you are doing **binary classification** or maybe almost never use this and the reason never use this is because the **tanh** is pretty much strictly superior. The default most commonly used activation function is the **Relu** so you're not sure what activation function use then use **Relu** one can also can try **leaky Relu**.

one of the thing we'll see in deep learning is that you often have a lot of different choices in how you code your neural network ranging from number of hidden units, which activation function to be chosen and many other things so it turns out that it is sometimes difficult to get good guidelines for exactly what will work best for your problem. Advice is to try them all because it's actually very difficult to know in advance exactly what will work best

Why do you need non-linear activation functions

why does your neural network need a nonlinear activation function? It turns out that for your neural network to compute interesting functions you do need to take a nonlinear activation function.
Let's try to understand with help of a diagram.



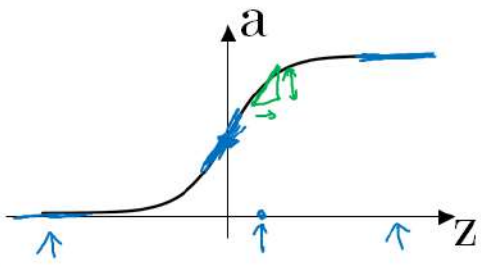
In the diagram for a given **x**, we have 4 forward propagation equations for the neural network. One of thing we can try is to get rid of the function **z** and set $\mathbf{a^{[1]} = z^{[1]}}$ or alternatively we could say that **g(z) = z** sometimes this is called the **linear activation function** or **identity activation function** because they're just outputs whatever was input for the purpose. Now if $\mathbf{a^{[2]} = z^{[2]}}$ then it turns out that model is just computing **y** or **yhat** as a **linear function** of your input features **x2.**

Please check the above diagram for more mathematical substutions for $\mathbf{a^{[1]}}$ and $\mathbf{a^{[2]}}$
So, If you were to use **linear activation functions** or **identity activation functions** then the neural network is just outputting a **linear function** of the **input** and this will be convered more in deep network unit. Neural networks with many many layers **(many many hidden layers)** and it turns out that if you use a **linear activation function** or alternatively if you don't have an activation function then no matter how many layers your neural network has it is nalways doing is just computing a **linear activation function** the take-home is that a linear hidden layer is more or less useless because on the composition of two linear functions is itself a **linear function** so unless you throw a **non-linearity** in there then you're not computing more interesting functions even as you go deeper in the network. There is just one place where you might use a linear activation function **(z) = z** that's if you are doing machine learning on a regression problem so if y is a real number so for example if you're trying to predict housing prices. In these kind of scenerion it is OK to have a **linear activation function** so that your output **yhat** is also a real number going anywhere from minus infinity to plus infinity. So the hidden units should not use linear activation functions but they can use **Relu/tanh/Leakly Relu.** So the one place where you can use linear activation functions is the output layerr but other than that using a linear activation function in hidden layer except for some very special circumstances relating to compression (we'll discuss it later) is extremely rare.
So we now understands that why having a nonlinear activation function is a critical part of neural networks.

Derivatives of activation functions

When you implement back-propagation for your neural network you need to really compute the **slope** or the **derivative** of the activation functions so let's take a look at our choices of **activation** functions and how we can compute the slope of these functions. Lets see the diagram below for familiar **sigmoid** activation function.

Sigmoid activation function



$$g(z) = \frac{1}{1+e^{-z}}$$

$a = g(z) = \frac{1}{1+e^{-z}}$

$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(z)$ at z

$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$

$= g(z)\left(1 - g(z)\right) \leftarrow$

$= a(1-a)$ $\quad | \; g'(z) = a(1-a)$

$z = 10$. $g(z) \approx 1$

$\frac{d}{dz} g(z) \approx 1\,(1-1) \approx 0$

$z = -10$ $\quad g(z) \approx 0$

$\frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$

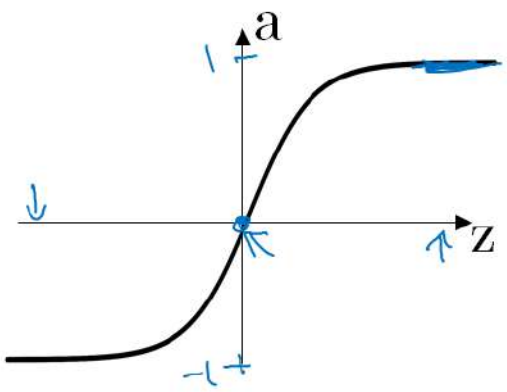
$z = 0$ $\quad g(z) = \frac{1}{2}$

$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

As per the diagram for any given value of **z** sigmoid function will have some slope or some derivative. Please go through the diagram above.

Now let's look at the **tanh** activation function. See the diagram below:

Tanh activation function



$$g(z) = \tanh(z)$$

$= \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(z)$ at z

$= 1 - (\tanh(z))^2 \leftarrow$

$a = g(z)$, $\quad g'(z) = 1 - a^2$

$z = 10$ $\quad \tanh(z) \approx 1$

$g'(z) \approx 0$

$z = -10$ $\quad \tanh(z) \approx -1$

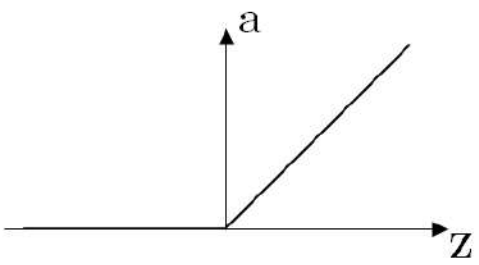
$g'(z) \approx 0$

$z = 0$ $\quad \tanh(z) = 0$

$g'(z) = 1$

Finally, lets see the derivatives for is ReLU and Leakly ReLU activation functions
See below the diagram:

ReLU and Leaky ReLU



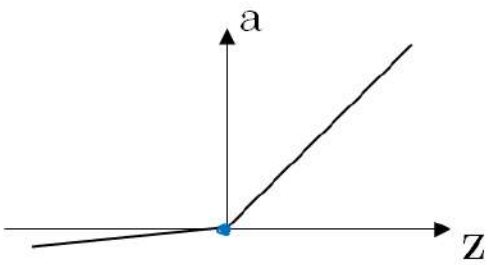
ReLU

$g(z) = \max(0, z)$

$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

~~undefined if $z = 0$~~

$z = 0.0000\ldots0$



Leaky ReLU

$g(z) = \max(0.01z, z)$

$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

So now we have all these formulas which we can use either to compute the slopes or the derivatives of our activation which are the building blocks to implemement the gradient descent for our neural network.

Gradient descent for Neural Networks

This is a exciting section as now we'll see how to implement **gradient descent** for our neural network with one hidden layer. We'll also see the equations needed to implement in order to get back- propagation of the gradient descent.
Let's check a diagram first:

Gradient descent for neural networks

Parameters: $W^{[1]}$ $(n^{[1]}, n^{[0]})$, $b^{[1]}$ $(n^{[1]}, 1)$, $W^{[2]}$ $(n^{[2]}, n^{[1]})$, $b^{[2]}$ $(n^{[2]}, 1)$

$n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

Cost function: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$ ($\hat{y} = a^{[2]}$)

Gradient descent:

Repeat {

Compute predicts $(\hat{y}^{(i)}, i = 1, \ldots, m)$

$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...

$W^{[1]} := W^{[1]} - \alpha\, dW^{[1]}$

$b^{[1]} := b^{[1]} - \alpha\, db^{[1]}$

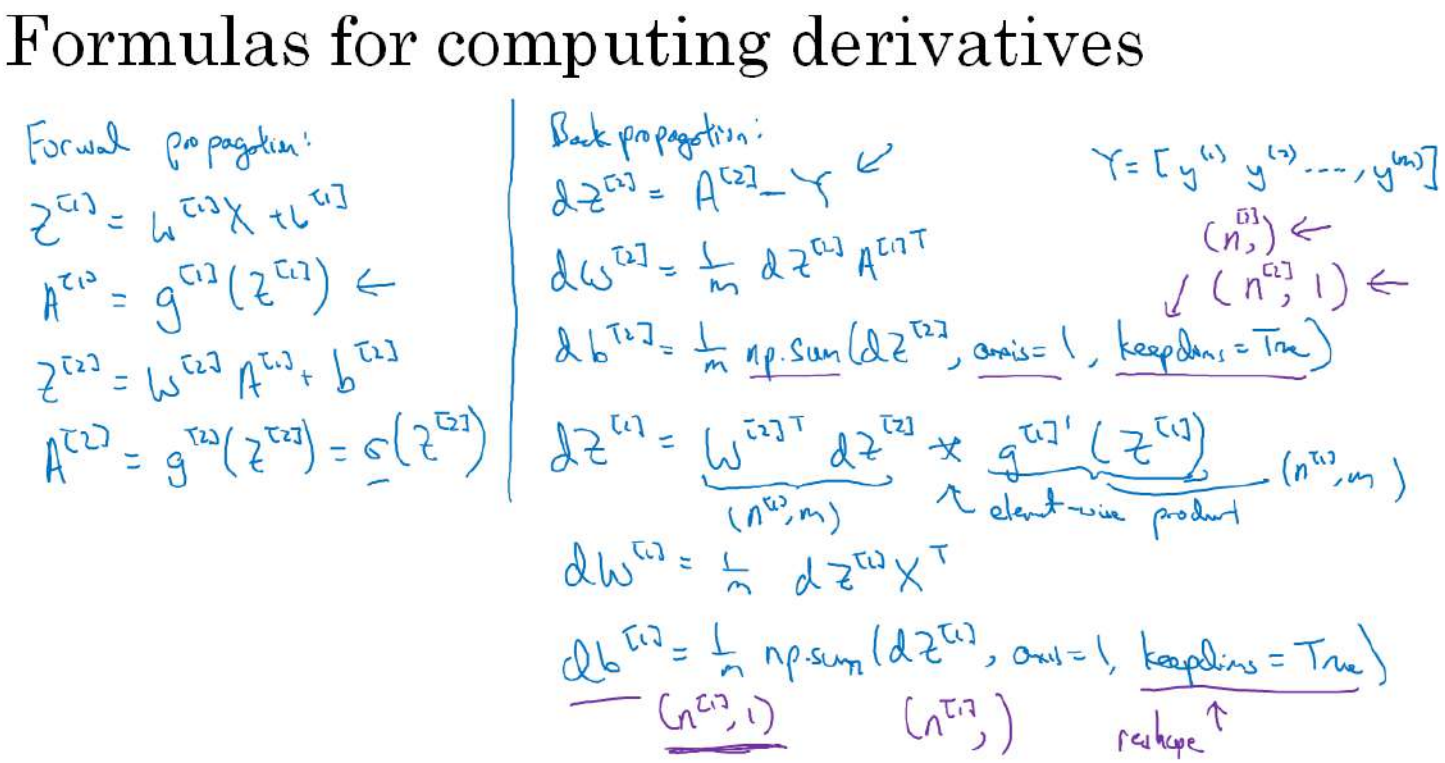
$W^{[2]} := \ldots$ $b^{[2]} := \ldots$

}

In this scenerio our neural network is with a single hidden layer where parameters are **w** and **b,** cost function is **J** and loss function is **L**. Now to perform gradient descent when training a neural network is important to initialize the parameters randomly rather than into all zeros but after initializing the parameter each loop of gradient descent would compute the predictions so you basically compute **yhat$^{[i]}$** where **i = 1...m** then you

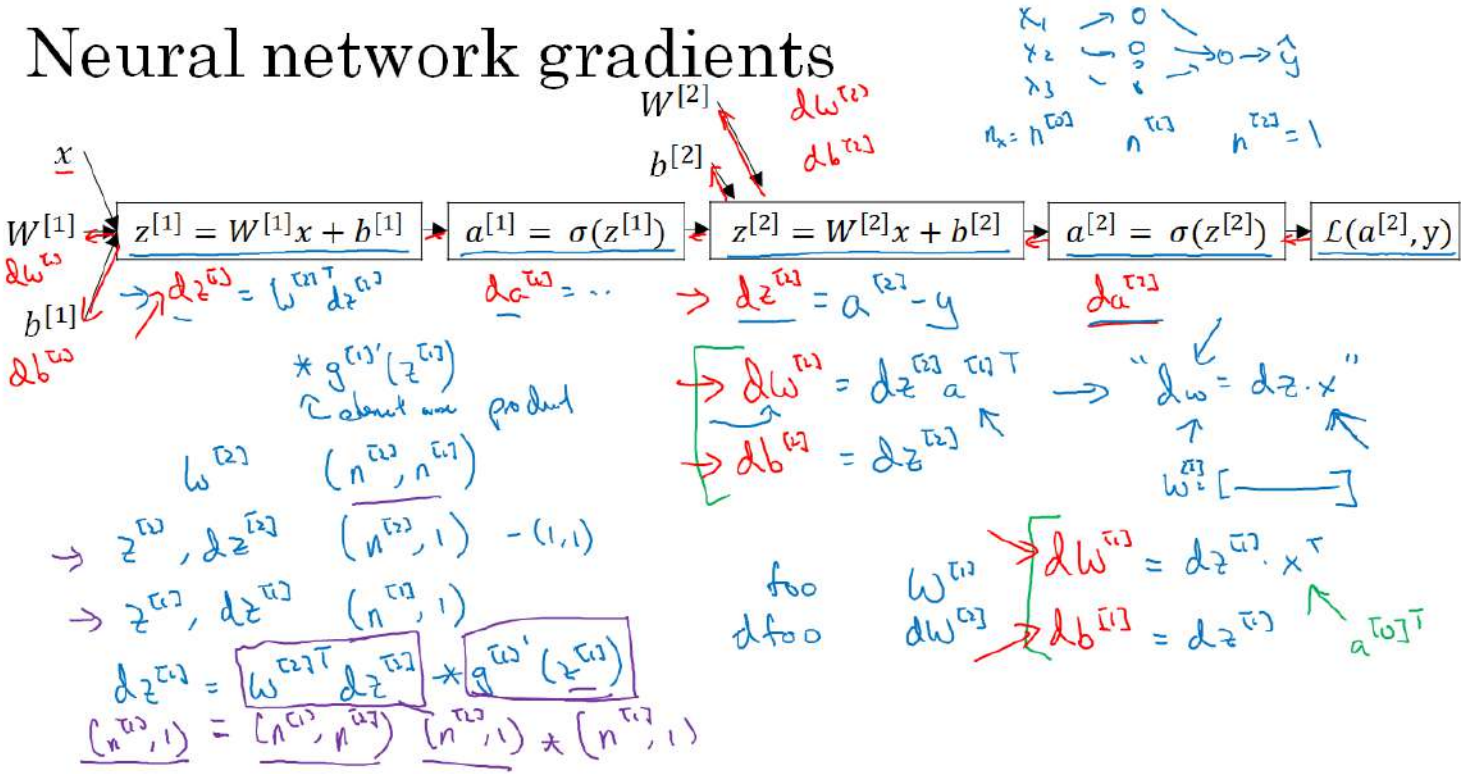
need to compute the derivative so you need to compute **Dw [1]** which is the derivative of the cost function with respect to the parameter w1, then you need to compute another variable which is going to call **Db [1]** which is the derivative or the slope of your cost function with respect to the variable **b1** and so on similarly for the other parameters **w2** and **b2** and then finally the gradient descent update would be to updated **w [1]** = **w[1] - alpha * dw [1]** and **b[1]** = **b[1] - alpha * db [1]** similarly for **w2** and **b2** so this would be one iteration of gradient descent and then your repeat this some number of times until your parameters look like they're converging

Now key is to know how to compute these partial derivative terms the **dw [1] db [1]** as well as the **dw [2] db [2]** just summarize again the equations for forward propagation. See diagram below.



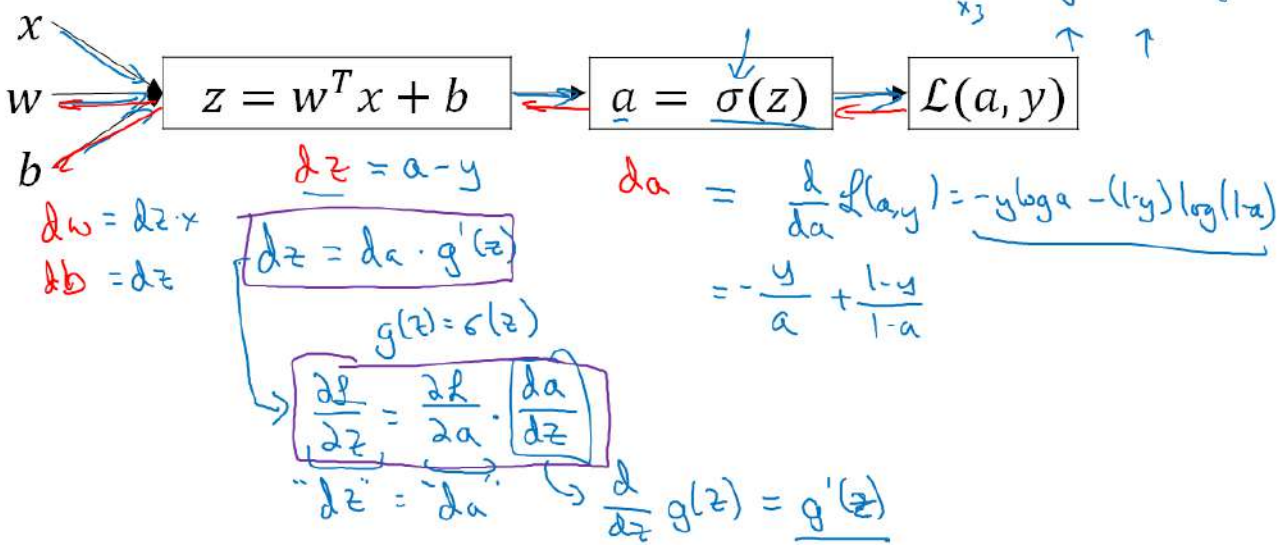
if we assume we're doing binary classification then this activation function really should be the **sigmoid** function. So that's the forward propagation or the left-to-right forward computation for our neural network, next let's compute the derivatives so this is the back propagation step it computes **dz [2]** = **a[2] - Y** and ground truth **Y = [y(1) y(2) y(3)** ..... **y(m)] .** in fact these first three equations are very similar to gradient descent for logistic regression **x**.

Neural network gradients



Computing gradients

Logistic regression



Summary of gradient descent

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

Vectorized Implementation:

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

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

Summary of gradient descent

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$(n^{[1]}, 1)$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

$$dZ^{[2]} = A^{[2]} - Y$$

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

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

$(n^{[1]}, m)$ $(n^{[1]}, m)$ $(n^{[1]}, m)$

element-wise product

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

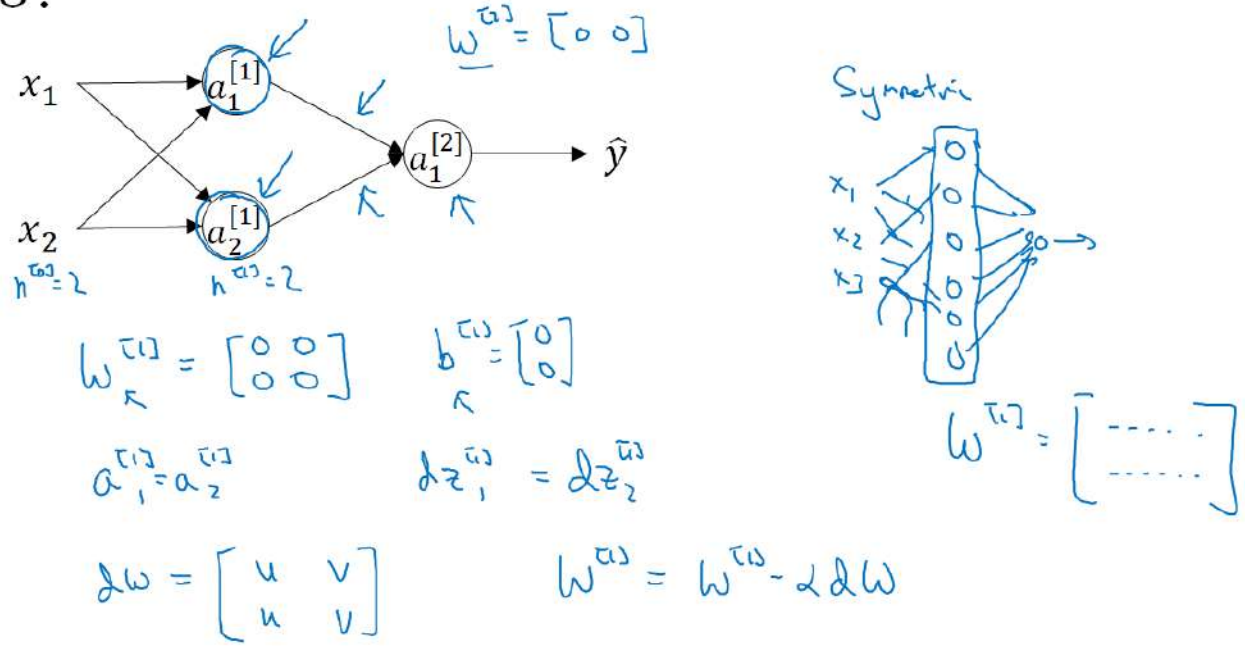
Random initialization

When you change your neural network, it's important to initialize the weights randomly. For logistic regression, it was okay to initialize the weights to zero but for a neural network if we initialize the weights and parameters to all zero and then apply gradient descent, it won't work.

Let's see why.

See the diagram first:

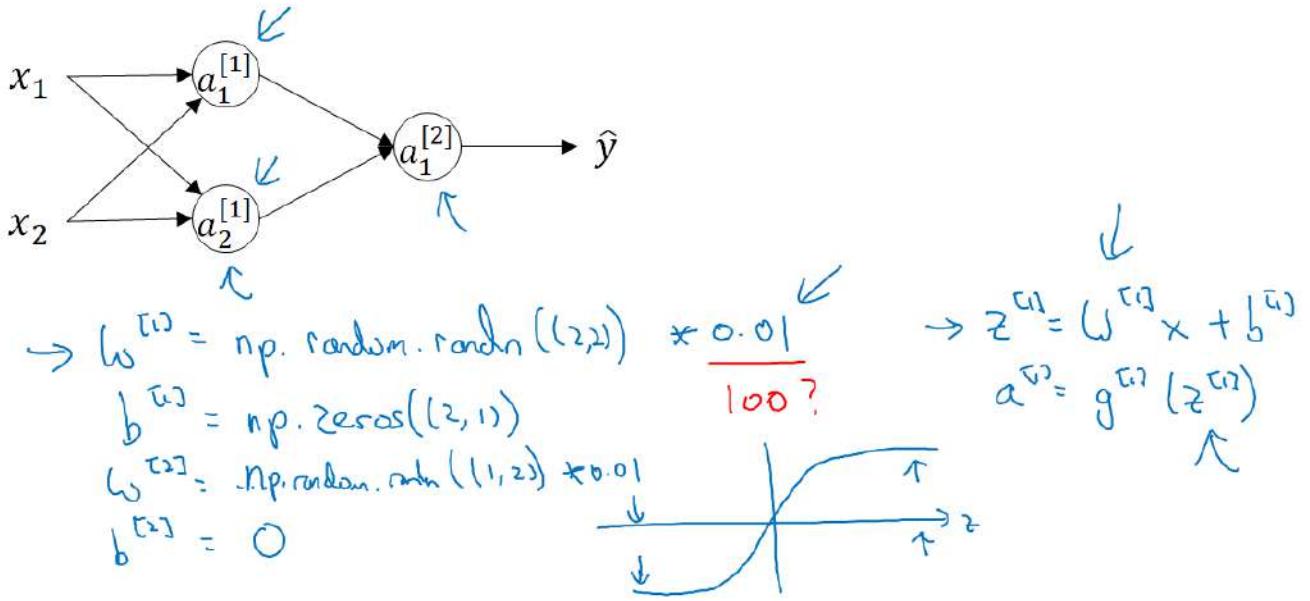
What happens if you initialize weights to zero?



So you have here two input features, so $n^{[0]}=2$, and two hidden units, so $n^{[1]}=2$ and so the matrix associated with the hidden layer, $w^{[1]}$ is going to be two-by-two. Let's say that you initialize it to all 0s, so it will be a **2x2** matrix with all zeros and let's say $b^{[1]}$ is also equal to 0 0. It turns out that initializing the bias terms **b** to 0 is actually okay, but initializing **w** to all 0's is a problem. So the problem with this formalization is that for any example you give it, you'll have $a_1^{[1]} = a_2^{[1]}$. So the $a_1^{[1]}$ activation and $a_2^{[1]}$ activation will be the same, because both of these hidden units are computing exactly the same function and then, when you compute backpropagation, it turns out that $dz_1^{[1]}$ and $dz_2^{[2]}$ will also be the same by symmetry. Both of these hidden units will initialize the same way. Technically, we are assuming that the outgoing weights are also identical. So that's w2 is equal to 0 0. But if you initialize the neural network this way, then both the hidden hidden units $a_1^{[1]}$ and $a_2^{[1]}$ completely identical. Sometimes we say they're completely symmetric, which just means that they're computing exactly the same function and by kind of a proof by induction, it turns out that after every single iteration of training your two hidden units are still computing exactly the same function. So if we check the weight update after first iteration we'll see that every row of the update matrix have same value. So it's possible to construct a proof by induction that if you initialize all the values of **w** to 0, then because both hidden units start off computing the same function and both hidden the units have the same influence on the output unit, then after one iteration, that same statement is still true, the two hidden units are still symmetric and therefore, by induction, after two iterations, three iterations and so on, no matter how long you train your neural network, both hidden units are still computing exactly the same function. And so in this case, there's really no point to having more than one hidden unit because they are all computing the same thing and of course, for larger neural networks, let's say of three features and maybe a very large number of hidden units, a similar argument works to show that with a neural network like this, if you initialize the weights to zero, then all of your hidden units are symmetric and no matter how long you run the gradient desecent it will end up in computing exactly the same function. So that's not helpful, because you want the different hidden units to compute different functions.

The solution to this is to initialize your parameters randomly.

Random initialization



So here's what we do. We can set **w [1] = np.random.randn**. This generates a gaussian random variable (2,2) and then usually, we multiply this by very small number, such as **0.01**. So we initialize it to very small random values and then **b**, it turns out that **b** does not have the symmetry problem, what's called the **symmetry breaking problem** so it's okay to initialize **b** to just zeros because as long as **w** is initialized randomly, we start off with the different hidden units computing different things and we no longer have this **symmetry breaking problem** and then similarly, for **w [2]** , we're going to initialize that randomly and **b2**, we can initialize to 0. One of the thing about the random initialization formula, where did this constant come from and why is it **0.01**? Why not put the number **100** or **1000**? TIt turns out that we usually prefer to initialize the weights to very small random values because if you are using a **tanh** or **sigmoid** activation function even just at the output layer. If the weights are too large, then when you compute the activation values, remember that **$z^{[1]}=w^{[1]}$ x + b** and then **a1** is the activation function applied to **z1**. So just a recap, if **w** is too large, we're more likely to end up even at the very start of training, with very large values of z. Which causes your **tanh** or your **sigmoid** activation function to be saturated, thus slowing down learning. If you don't have any **sigmoid** or **tanh** activation functions throughout your neural network, this is less of an issue. But if we're doing binary classification, and
our output unit is a **sigmoid function**, then you just don't want the initial parameters to be too large. So that's why multiplying by 0.01 would be something reasonable to try, or any other small number.

So turns out that sometimes they can be better constants than 0.01. When you're training a neural network with just one hidden layer, it is a relatively shallow neural network, without too many hidden layers. Set it to 0.01 will probably work okay.

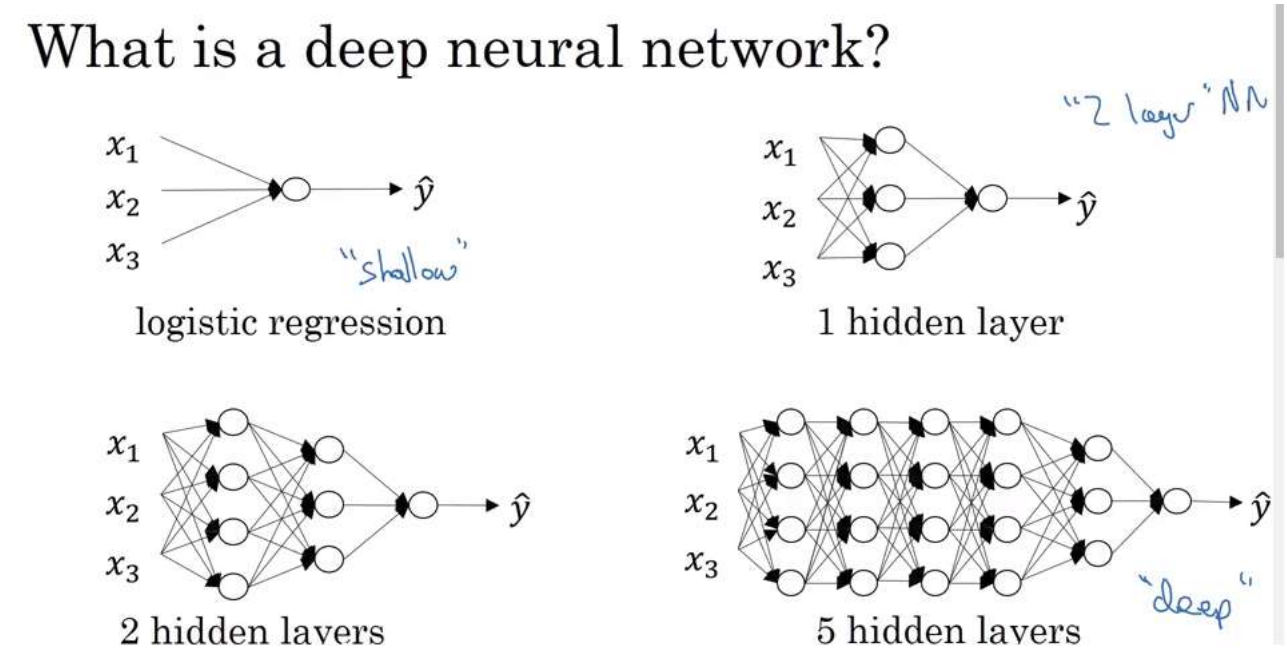
Week 3: Deep Neural Networks (Multiple layers)

Learning Objectives

- See deep neural networks as successive blocks put one after each other
- Build and train a deep L-layer Neural Network
- Analyze matrix and vector dimensions to check neural network implementations.
- Understand how to use a cache to pass information from forward propagation to back propagation.
- Understand the role of hyperparameters in deep learning

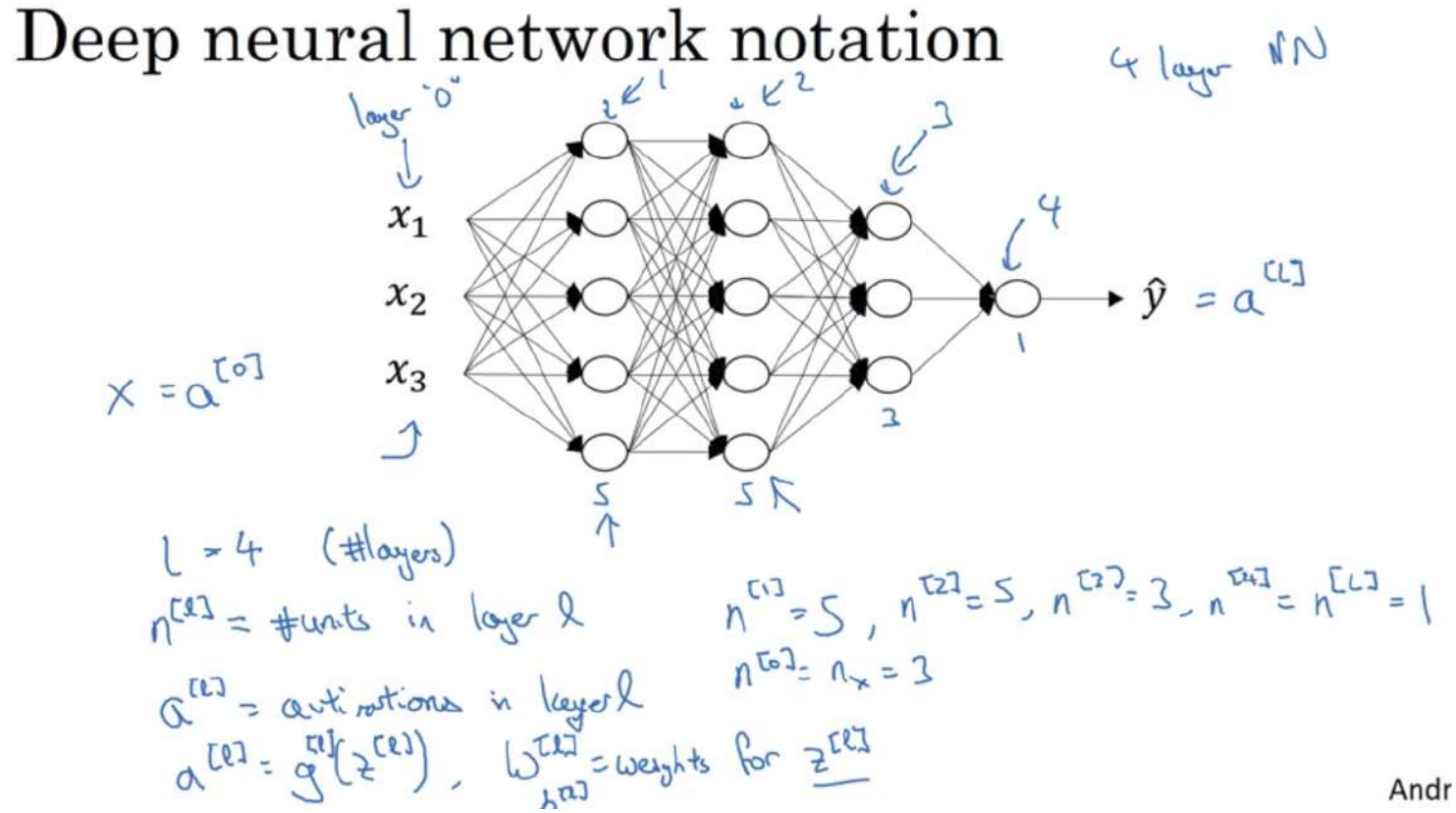
Deep L-layer Neural Network

Till now we have understood the forward propagation and backward propagation in the context of a neural network, with a single hidden layer, as well as logistic regression and we've learned about vectorization, and when it's important to initialize the ways randomly. So by now, we've actually seen most of the ideas we need to implement a deep neural network. In this section we'll take the ideas and put them together so that we'll be able to implement our own deep neural network. So what is a deep neural network?
See below diagram for neural networks with a single hidden layer, a neural network with two hidden layers and a neural network with 5 hidden layers.



We say that logistic regression is a very **"shallow" model**, whereas this model here is a much deeper model, and shallow versus depth is a matter of degree. So neural network of a single hidden layer, this would be a 2 layer neural network. Remember when we count layers in a neural network, we don't count the input layer, we just count the hidden layers as was the output layer. So, this would be a 2 layer neural network is still quite shallow, but not as shallow as logistic regression. Technically **logistic regression is a one layer neural network**, we could then, but over the last several years the AI, on the machine learning community, has realized that there are functions that very deep neural networks can learn that shallower models are often unable to. Although for any given problem, it might be hard to predict in advance exactly how deep in your network you would want. So it would be reasonable to try logistic regression, try one and then two hidden layers, and view the number of hidden layers as another hyper parameter that you could try a variety of values of, and evaluate on all that across validation data, or on your development set.

Let's now go through the notation we used to describe deep neural networks. Check the below diagram for a one, two, three, four layer neural network.



There are three hidden layers, and the number of units in these hidden layers are 5, 5, 3, and then there's one output unit. So the notation we're going to use, is going to use **L** ,to denote the number of layers in the network. So in this case **L = 4** and we're going to use $\mathbf{n^{[l]}}$ **(lowercase l)** to denote the number of nodes, or the number of units in layer.
For detailed notation check the above diagram.

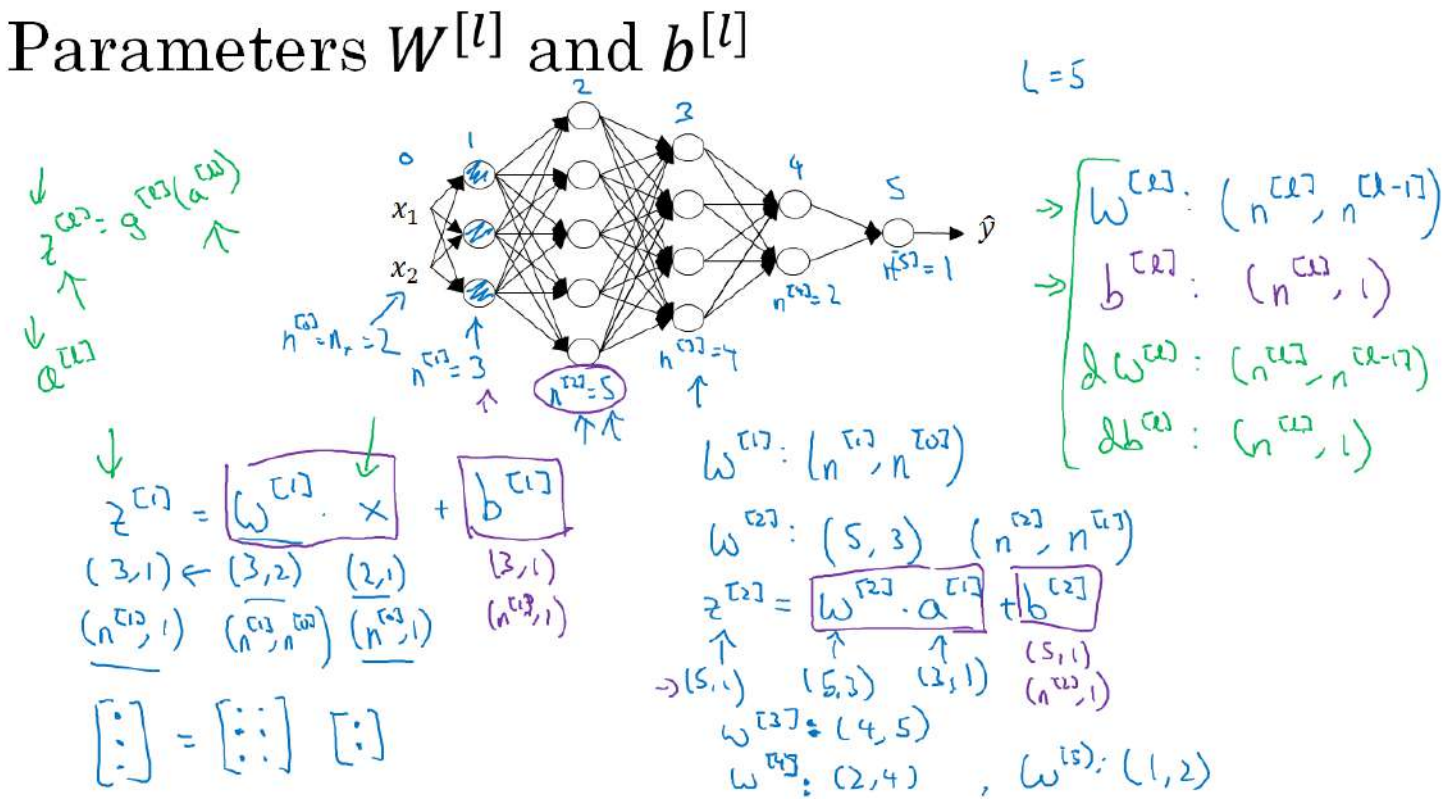
Forward Propagation in a Deep Network

in the last section we learned what is the L-layer deep neural network and also talked about the notation. I this section we'll learn how to perform forward propagation in a deep network. As usual let's first go over what forward propagation will look like for a single training example **x** and then later on we'll talk about the vectorized version where you want to carry out forward propagation on the entire training set at the same time. See below diagram to calculate the forward prpagation in L-layer network.

This diagram shows how to compute activations of the first layer and so on. For this first layer we compute $\mathbf{z^{[1]} = W^{[1]} * X + b^{[1]}}$ so $\mathbf{W^{[1]}}$ and $\mathbf{b^{[1]}}$ of parameters that affect the activations in layer 1 and then you compute the activations for that layer $\mathbf{a^{[1]} = g^{[1]}(z^{[1]})}$ the destination function g depends on what layer you're at for layer 1 this value will be 1 now let's look into layer 2, we would compute $\mathbf{z^{[2]} = W^{[2]} * a^{[1]} + b^{[2]}}$ and then so the activation of layer 2 is the weight matrix times the output of layer 1 plus the bias vector for layer 2 and then $\mathbf{a^{[2]} = g^{[2]}(z^{[2]})}$ and so on for other subsequent layers and finally we'll get to the output layer that's layer 4 and where you would see how to compute your estimated output $\mathbf{z^{[1]} = W^{[4]} * a^{[3]} + b^{[4]}}$ and $\mathbf{a^{[4]} = g^{[4]}(z^{[4]})}$ = **yhat** so just one thing to notice **X** here is also equal to $\mathbf{a^{[0]}}$ because the input feature vector **X** is also the activation of layer 0. Now we've done all this for a single training example how about for doing it in a vectorized way for the whole training set. Check the diagram for the more general formula at the top right and vectorized version at the right bottom in the diagram. When implementing neural networks we usually want to get rid of explicit for loops but this is one place where there is no way to implement this over other than explicit for loop so while implementing forward propagation it is perfectly OK to have a for loop which compute the activations for layer 1 then for layer 2 so on where **for** loop that goes from **1 to L**, from 1 through the total number of layers and in neural network. To summarize, in deep network we are just repeating (what we've seen in the neural network with a single hidden layer) more no of times.

Getting your matrix dimensions right

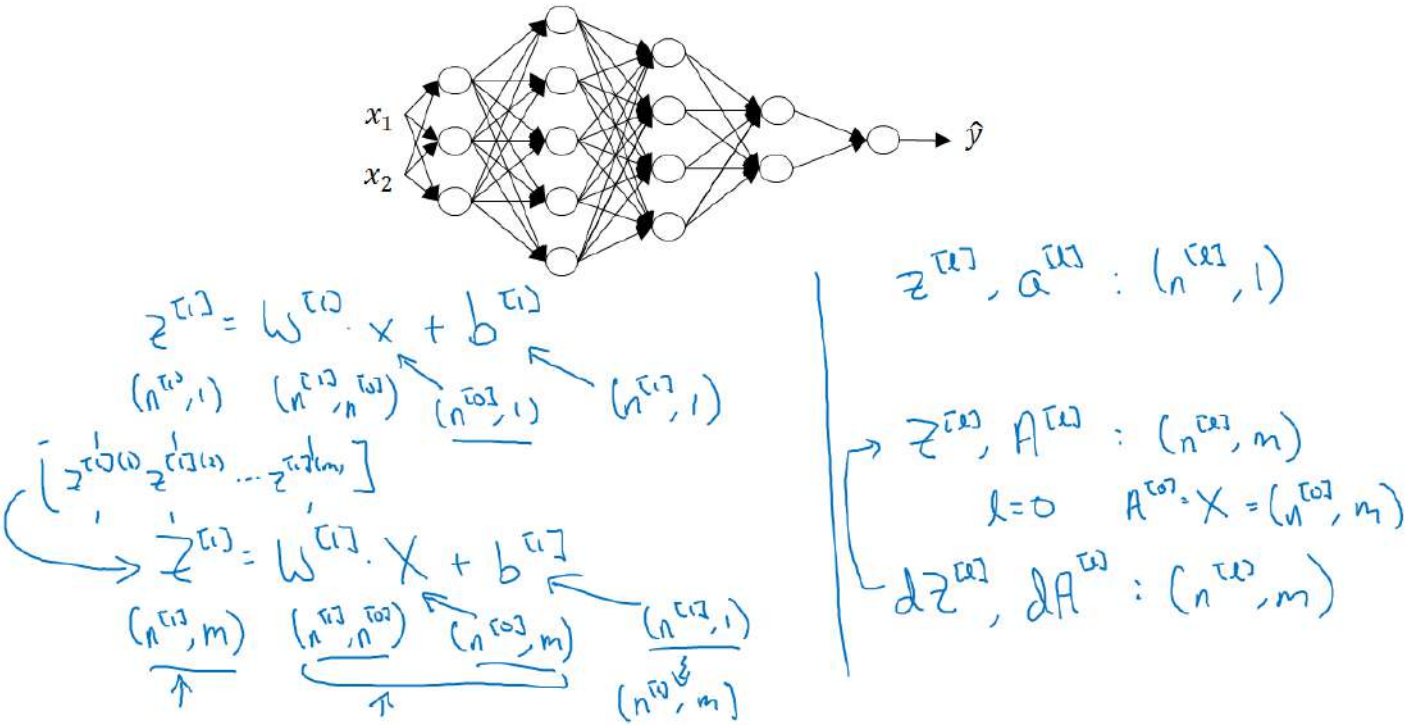
When implementing a deep neural network, one of the debugging tools people often use to check the correctness of the code.



Please check the above diagram which explains in details ways to calculate the matrix and setting them right.

When implementing a deep neural network, one of the debugging tools one should use is to check the correctness of code on a piece of paper, and just work through the dimensions and matrix we are working with. Let's see how to do this using the above diagram where L is equal to 5, not counting the input layer, there are five layers, so four hidden layers and one output layer. Now if you implement forward propagation, the first step will be $\mathbf{z^{[1]} = W^{[1]} * X + b^{[1]}}$. Let's ignore the bias terms b for now, and focus on the parameters w. Now the first hidden layer has 3 hidden units, so there are layer 0, layer 1, layer 2, layer 3, layer 4, and layer 5. So using the notation we had from the previous section, we have that $\mathbf{n^{[1]}}$, which is the number of hidden units in layer 1, is equal to 3 and here we would have the $\mathbf{n^{[2]}}$ is equal to 5, $\mathbf{n^{[3]}}$ is equal to 4, $\mathbf{n^{[4]}}$ is equal to 2, and $\mathbf{n^{[5]}}$ is equal to 1. So far we've only seen neural networks with a single output unit, but in later sections we'll learn more about neutral networks with multiple output units and finally, for the input layer, we also have $\mathbf{n^{[0]} = n^{[x]} = 2}$. So now, let's think about the dimensions of **z**, **w**, and **x**. Here **z** is the vector of activation for the first hidden layer, so **z** is going to be **3x1**, it's going to be a 3-dimensional vector/matrix. More generally we write this as $\mathbf{n^{[1]}}$ **x 1** dimensional vector/matrix in this case n=3. Now how about the input features **x**, with the diagram we have two input features. So x is in this diagram is **2x1**, but more generally, it would be $\mathbf{n^{[0]}}$ **x 1**. So what we need is for the matrix **w1** to be something that when we multiply an $\mathbf{n^{[0]}}$ **x 1** vector to it, we get an $n^{[1]}$ x 1 vector. So you have sort of a three dimensional vector equals something times a two dimensional vector and so by the rules of matrix multiplication, this has got be a **3x2** matrix because a **3x2** matrix times a **2x1** matrix that gives you a **3x1** vector and more generally, this is going to be an $\mathbf{n^{[1]}}$ **x** $\mathbf{n^{[0]}}$ dimensional matrix. So what we figured out here is that the dimensions of w1 has to be $\mathbf{n^{[1]}}$ **x** $\mathbf{n^{[0]}}$ and more generally, the dimensions of $\mathbf{w^{[l]}}$ must be $\mathbf{n^{[l]}}$ **x** $\mathbf{n^{[l-1]}}$. This is to be done for $\mathbf{w^{[2]}}$, $\mathbf{w^{[3]}}$ so on..Similarly general formula for dimension for $\mathbf{b^{[l]}}$ must be $\mathbf{n^{[l]}}$ **x 1.** Check the diagram above for full explanation.

Vectorized implementation



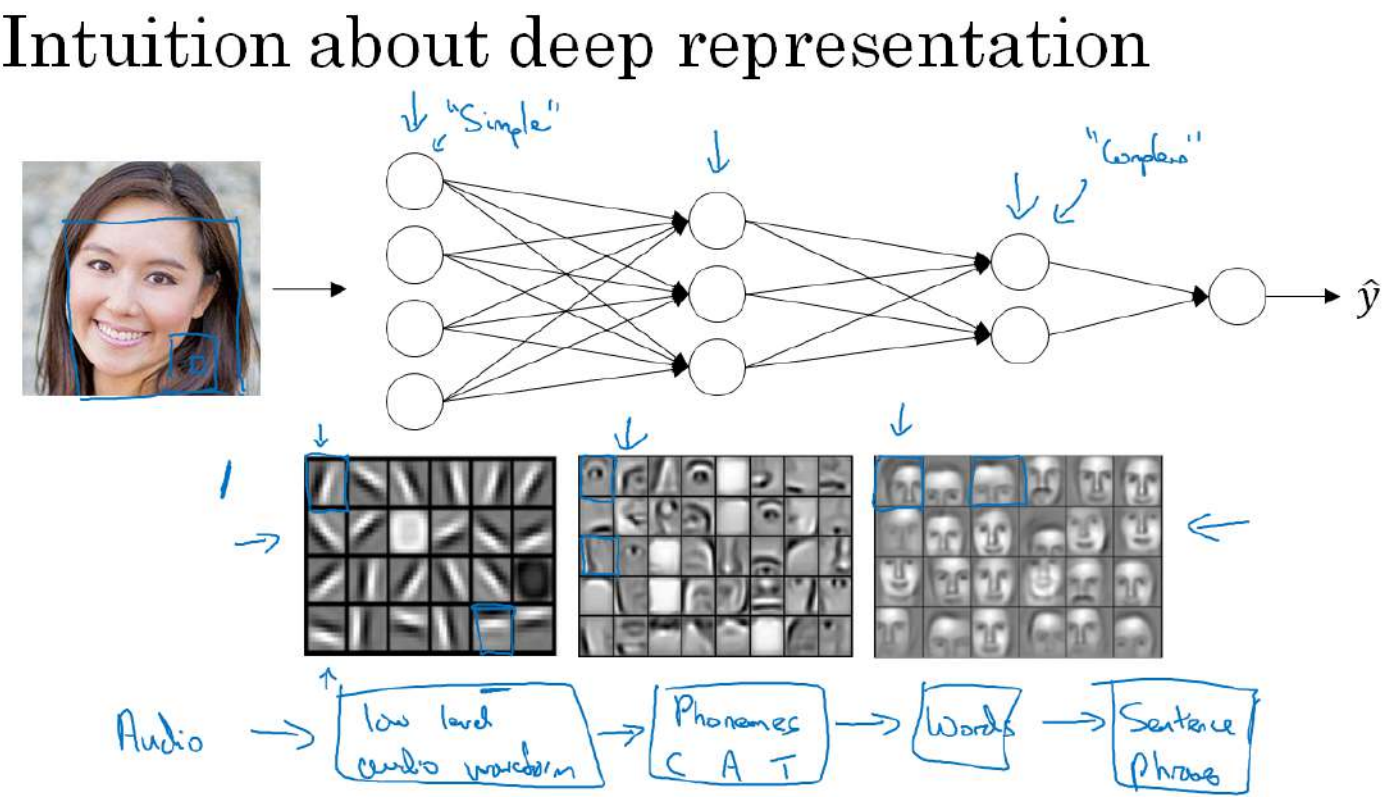
Check the above diagram. Now if you're implementing backpropagation then the dimensions of **dw** should be the same as dimension of **w** similarly **db** should be the same dimension as **b**. Now the other key set of quantities whose dimensions to check are these **z**, **x**, as well as $\mathbf{a^{[l]}}$ we know the equation $\mathbf{z^{[l]} = g^{[l]}(a^{[l]})}$ if multipy element wise then **z** and **a** should have the same dimension in these types of networks. Now let's see what happens when you have a vectorized implementation that looks at multiple examples at a time. Even for a vectorized implementation, the dimensions of **w**, **b**, **dw**, and **db** will stay the same but the dimensions of **z**, **a**, as well as **x** will change.

So previously, we had $\mathbf{z^{[1]} = w^{[1]} * x + b^{[1]}}$ where $\mathbf{z^{[1]}}$ was $\mathbf{n^{[1]} \times 1}$, $\mathbf{w^{[1]}}$ was $\mathbf{n^{[1]} \times n^{[0]}}$, **x** was $\mathbf{n^{[0]} \times 1}$, and $\mathbf{b^{[1]}}$ was $\mathbf{n^{[1]} \times 1}$. Now, in a vectorized implementation, we would have $\mathbf{Z^{[1]} = W^{[1]} * X + b^{[1]}}$. Where now $\mathbf{Z^{[1]}}$ (Capital **Z**) is obtained by taking the $\mathbf{z^{[1]}}$ for the individual examples, so there's $\mathbf{z^{[11]}}$, $\mathbf{z^{[12]}}$, up to $\mathbf{z^{[1m]}}$, and stacking them which gives us $\mathbf{Z^{[1]}}$. So the dimension of $\mathbf{Z^{[1]}}$ is instead of being $\mathbf{n^{[1]} \times 1}$, it ends up being $\mathbf{n^{[1]} \times m}$, and **m** is the size we're trying to set. The dimensions of $\mathbf{W^{[1]}}$ stays the same, so it remains $\mathbf{n^{[1]} \times n^{[0]}}$ and x, instead of being $\mathbf{n^{[1]} \times 1}$ is now as all our training examples stacked horizontally so it's now $\mathbf{n^{[1]} \times m}$, so we have notice that when you take a $\mathbf{n^{[1]} \times n^{[0]}}$ matrix and multiply that by an $\mathbf{n^{[1]} \times m}$ matrix. That together actually give you an $\mathbf{n^{[1]} \times m}$ dimensional matrix as expected. Now, the final detail is that $\mathbf{b^{[1]}}$ is still $\mathbf{n^{[1]} \times 1}$, but when you take first part of the equation $\mathbf{W^{[1]} * X}$ and add it to $b^{[1]}$, then through Python broadcasting, this will get duplicated and will return a $\mathbf{n^{[1]} \times m}$. When you implement backpropagation for a deep neural network, so long as you work through your code and make sure that all the matrices' dimensions are consistent. That will usually help, it'll go some ways toward eliminating some cause of possible bugs. Please check the above diagram for final equations.

Why deep representation?

We've all been hearing that deep neural networks work really well for a lot of problems, and it's not just that they need to be big neural networks, is that specifically, they need to be deep or to have a lot of hidden layers. So why is that? Let's go through a couple examples and try to gain some intuition for why deep networks might work well. So first, what is a deep network computing? If you're building a system for face recognition or face detection, here's what a deep neural network could be doing (Check the diagram below). Perhaps you input a picture of a face then the first layer of the neural network you can think of as maybe being a feature detector or an edge detector. In this example, we're plotting what a neural network with maybe 20 hidden units, might be kind of compute on this image. So the 20 hidden units visualize by these little square boxes. So for this example, this little visualization represents a hidden unit is trying to figure out if where the edges of that orientation and maybe this hidden unit is tying to figure out where are the horizontal edges in this image and when we talk about convolutional networks in a later course, this particular visualization will make a bit more sense. But the form, you can think of the first layer of the neural network as look at the picture and try to figure out where are the edges in this picture. Now, let's think about where the edges in this picture by grouping together pixels to form edges. It can then de-detect the edges and group edges together to form parts of faces. So for example, you might have a low neuron trying to see if it's finding an eye, or a different neuron trying to find that part of the nose and so by putting together lots of edges, it can start to detect different parts of faces and then, finally, by putting together different parts of faces, like an eye or a nose or an ear or a chin, it can then try to recognize or detect different types of faces. So intuitively, you can think of the **earlier layers of the neural network as detecting simple functions, like edges and then composing them together in the later layers of a neural network so that it can learn more and more complex functions**. These visualizations will make more sense when we talk about convolutional nets and one technical detail of this visualization, the edge detectors are looking in relatively small areas of an image, maybe very small regions and then the facial detectors you can look at maybe much larger areas of image but the main addition while you take away from this is just finding simple things like edges and then building them up composing them together to detect more complex things and this type of simple to complex hierarchical representation, or compositional representation, applies in

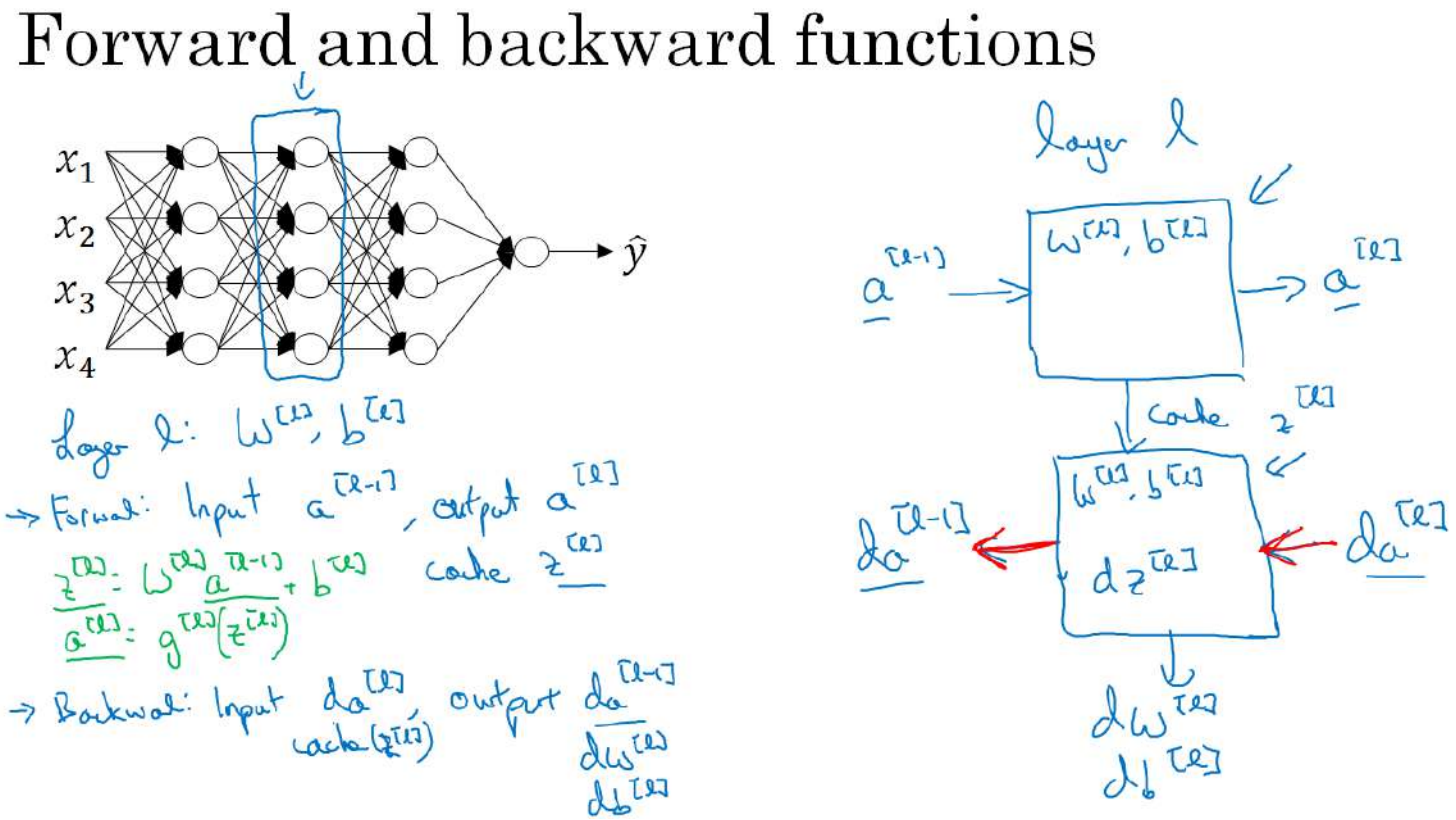
other types of data than images and face recognition as well.



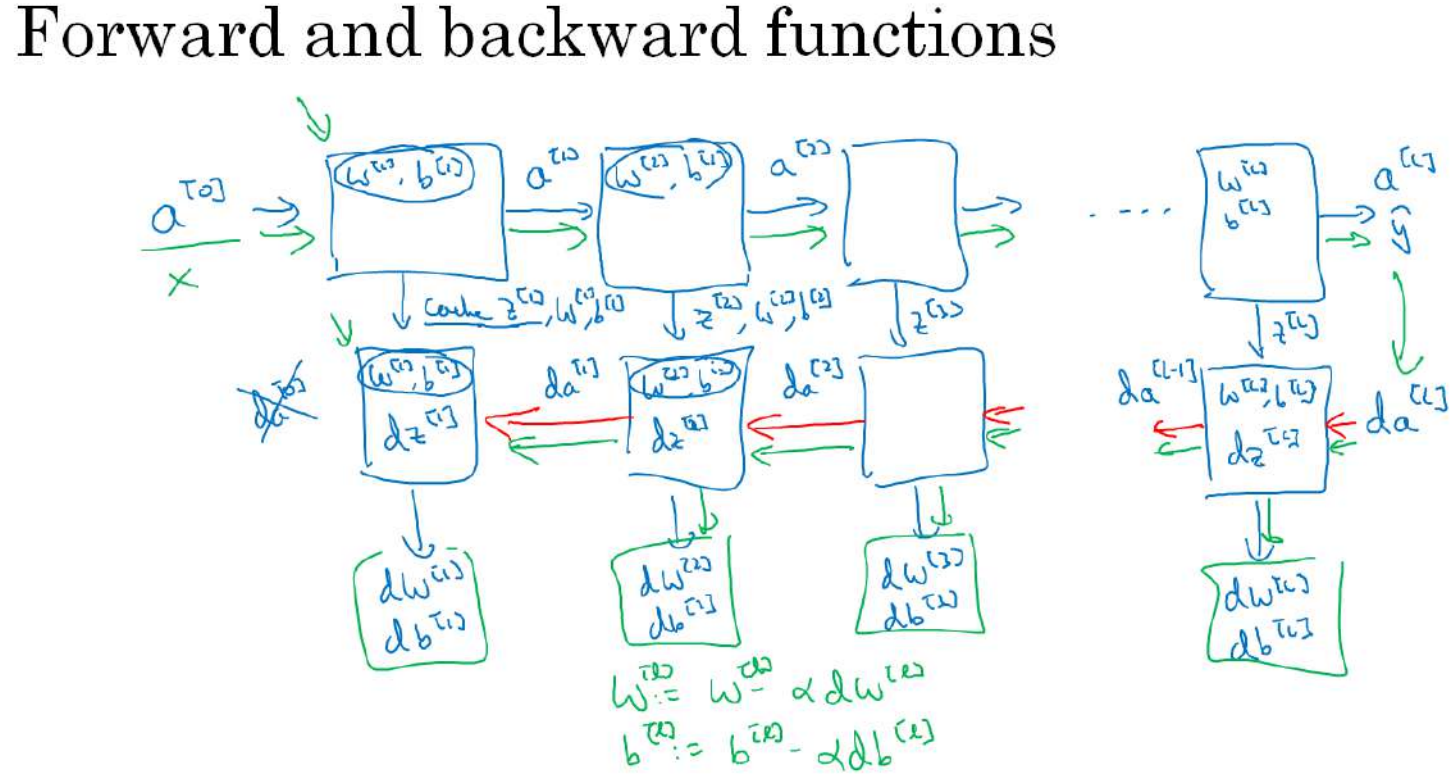
Another example, if you're trying to build a **speech recognition system**, it's hard to revisualize speech but if you input an audio clip then maybe the first level of a neural network might learnt to detect low level audio wave form features, such as is this tone is going up? Is it going down? Is it white noise or slithering sound and what is the pitch? When it comes to that, detect low level wave form features like that and then by composing low level wave forms, maybe you'll learn to detect basic units of sound. In linguistics they call phonemes. But, for example, in the word cat, the C is a phoneme, the A is a phoneme, the T is another phoneme. But learns to find maybe the basic units of sound and then composing that together maybe learn to recognize words in the audio. And then maybe compose those together, in order to recognize entire phrases or sentences. So deep Internet work with multiple hidden layers might be able to have the earlier layers learn these lower level simple features and then have the later deeper layers then put together the simpler things it's detected in order to detect more complex things like recognize specific words or even phrases or sentences. The uttering in order to carry out speech recognition and what we see is that whereas the other layers are computing, what seems like relatively simple functions of the input such as right at the edges, by the time you get deep in the network you can actually do surprisingly complex things. Such as detect faces or detect words or phrases or sentences. Some people like to make an analogy between deep neural networks and the human brain, where we believe, or neuroscientists believe, that the human brain also starts off detecting simple things like edges in what your eyes see then builds those up to detect more complex things like the faces that you see. I think analogies between deep learning and the human brain are sometimes a little bit dangerous. But there is a lot of truth to, this being how we think that human brain works and that the human brain probably detects simple things like edges and then put them together to from more and more complex objects and so that has served as a loose form of inspiration for some people learning as well. The other piece of intuition about why deep networks seem to work well is the following. So this result comes from **circuit theory** of which pertains the thinking about what types of functions you can compute with different logic case. So informally, their functions compute with a relatively small but deep neural network and by small I mean the number of hidden units is relatively small. But if you try to compute the same function with a shallow network, no hidden layers, then you might require exponentially more hidden units to compute. Now, in addition to this reasons for preferring deep neural networks to be roughly on, is I think the other reasons the term deep learning has taken off is just branding. This things just we call neural networks belong to hidden layers, but the phrase deep learning is just a great brand, it's just so deep. So I think that once that term caught on that really neural networks rebranded or neural networks with many hidden layers rebranded, help to capture the popular imagination as well. They regard as the PR branding deep networks do work well. Sometimes people go overboard and insist on using tons of hidden layers. But when I'm starting out a new problem, I'll often really start out with logistic regression then try something with one or two hidden layers and use that as a hyper parameter. Use that as a parameter or hyper parameter that you tune in order to try to find the right depth for your neural network. But over the last several years there has been a trend toward people finding that for some applications, very, very deep neural networks here with maybe many dozens of layers sometimes, can sometimes be the best model for a problem.

Building blocks of deep neural networks

In the earlier sections we've already seen the basic building blocks of forward propagation and back propagation which are actually the key components to implement a deep neural network. Let's see how we can put these components together to build a deep net. Check the diagram below

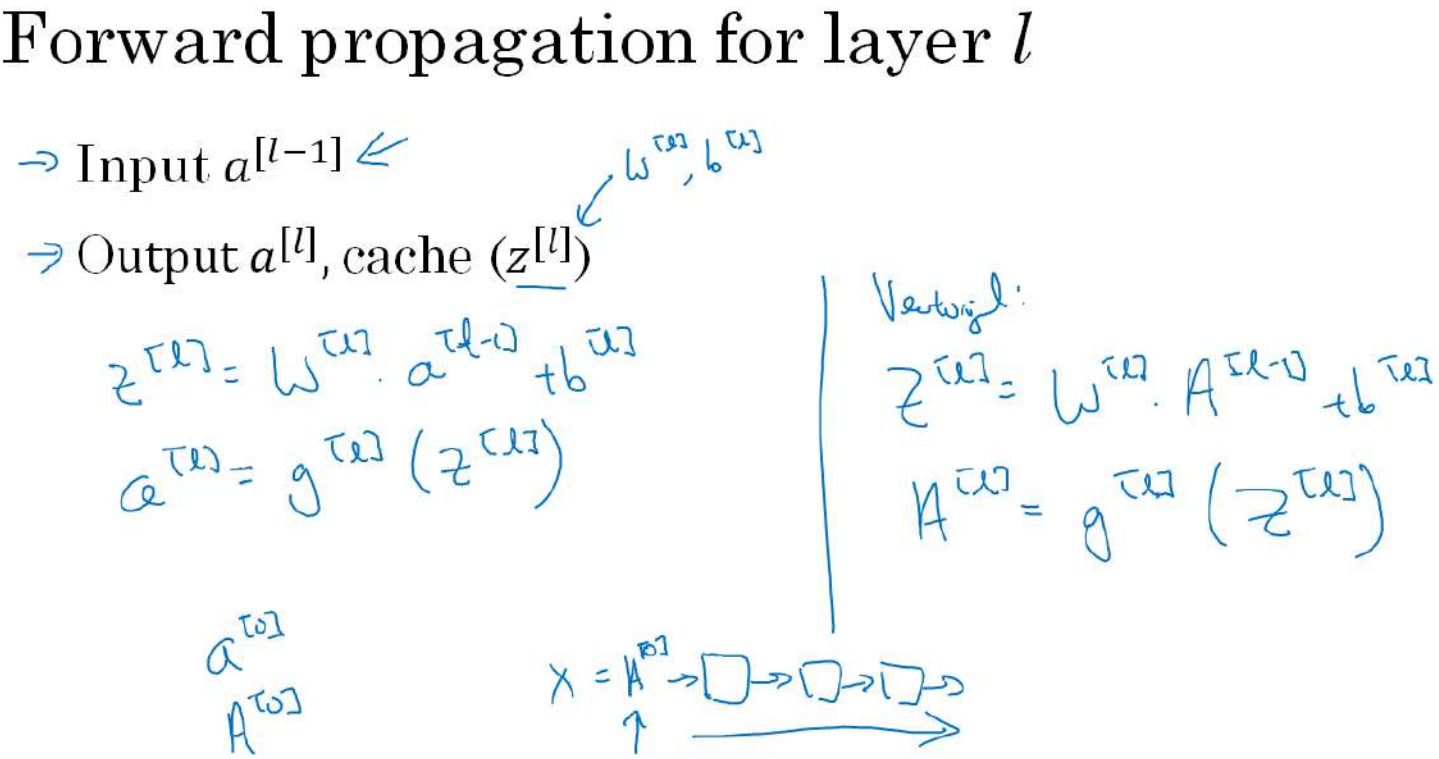


We have used NN with a few layers let's pick one layer and look at the computations focusing on just that layer for now check diagram for full flow of forward propagation nad backward propagation for that layer. This also show how we cache intermediate values for forward propagation and backward propagation.



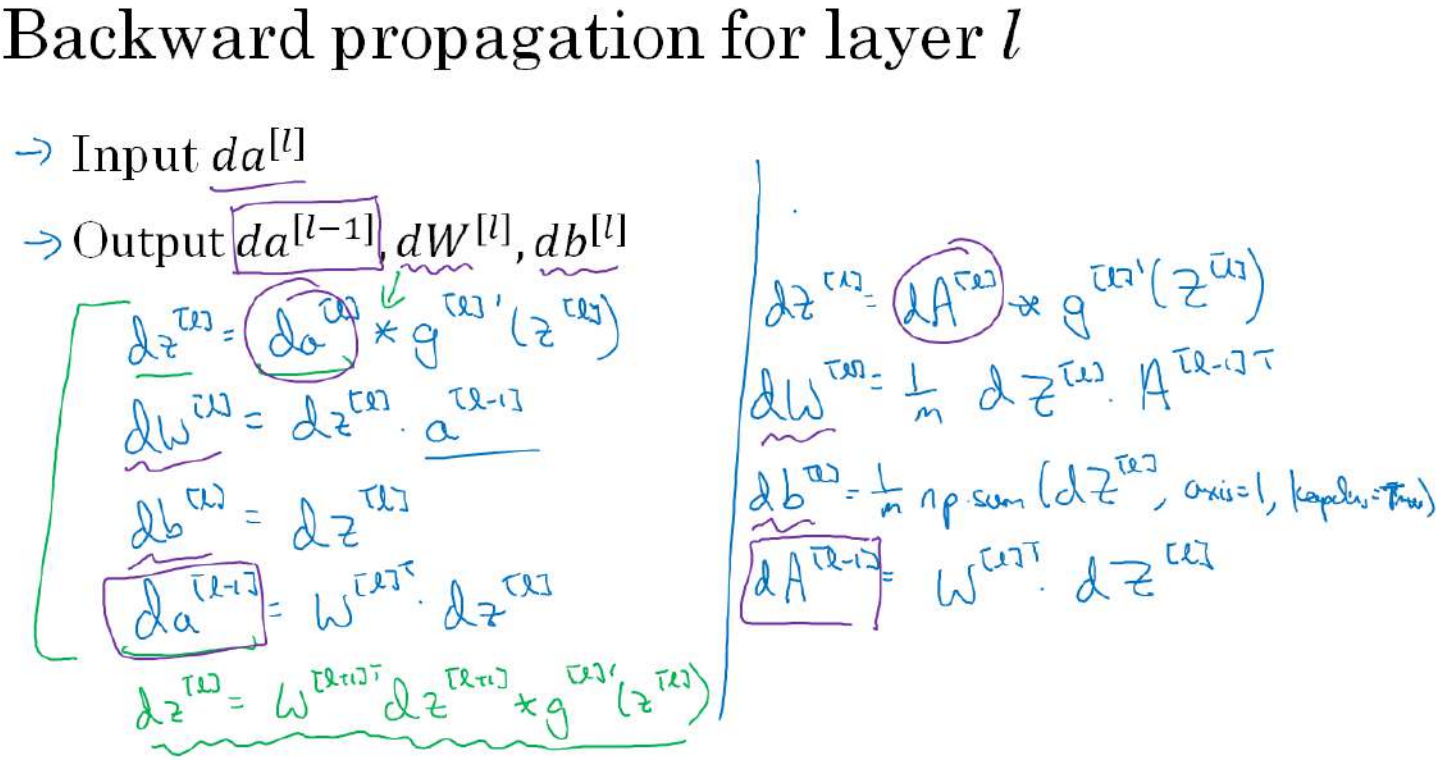
Forward and backward propagation

In last section we saw the basic blocks of implementing a deep neural network, a forward propagation step for each layer and a corresponding backward propagation step now let's see how we can actually implement these steps will start forward propagation. Recall from the last section (check the diagram)



we have input **a[l-1]**, output **a[l]** and cache **z[l]** and as we discussed before from implementational point of view maybe we'll cache **w[l]** and **b[l]** as well. Right side of the diagram is vectorized implementation of the same.

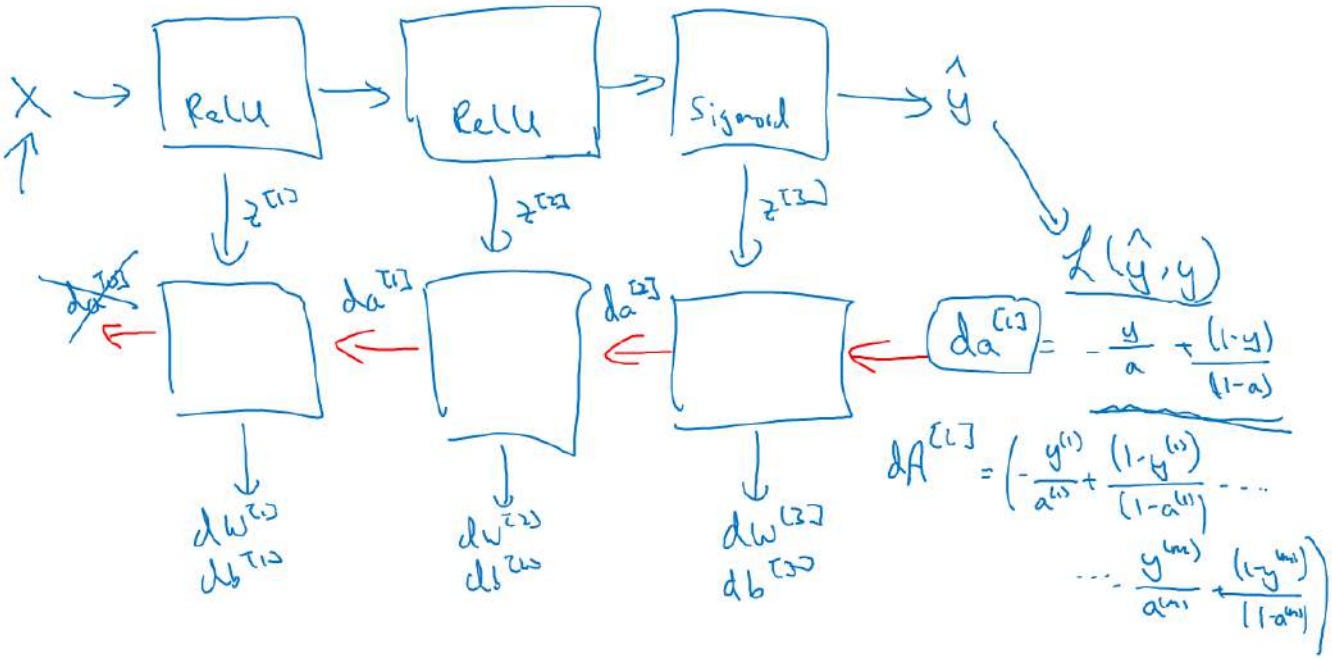
Now, Let's talk about the backward propagation step. See the diagram below.



here we have input **da[l]** and output **da[l-1]**, **dW[l]**, **db[l]** Check the left part of the diagram those four equations you actually need to implement backward function and for the vectorized version equations check the right side of the diagram.

So just to summarize you take the input X you might have the first layer maybe has a **ReLU** activation function then go to the second layer maybe uses another **ReLU** activation function goes to the third layer maybe has a **sigmoid** activation function (if you're doing binary classification) and this outputs **yhat** and then using **yhat** we can compute the **loss**. Check the diagram below:

Summary



and this allows us to start back propagation (check the diagram woth RED arrows) it shows how we calculate derivates along the way back and also use cache.

**********END OF COURSE **********

Course 2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Author: Pradeep K. Pant
URL: https://www.coursera.org/learn/deep-neural-network/home/welcome

****

Course 2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

****

This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow.
After 3 weeks, you will:
- Understand industry best-practices for building deep learning applications.
- Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking,
- Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.
- Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance
- Be able to implement a neural network in TensorFlow.

Week 1: Introduction to Deep Neural Networks

Learning Objectives

- Recall that different types of initializations lead to different results
- Recognize the importance of initialization in complex neural networks.
- Recognize the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Learn when and how to use regularization methods such as dropout or L2 regularization.
- Understand experimental issues in deep learning such as Vanishing or Exploding gradients and learn how to deal with them
- Use gradient checking to verify the correctness of your backpropagation implementation

Setting up your Machine Learning Application

Train/Dev/Test sets

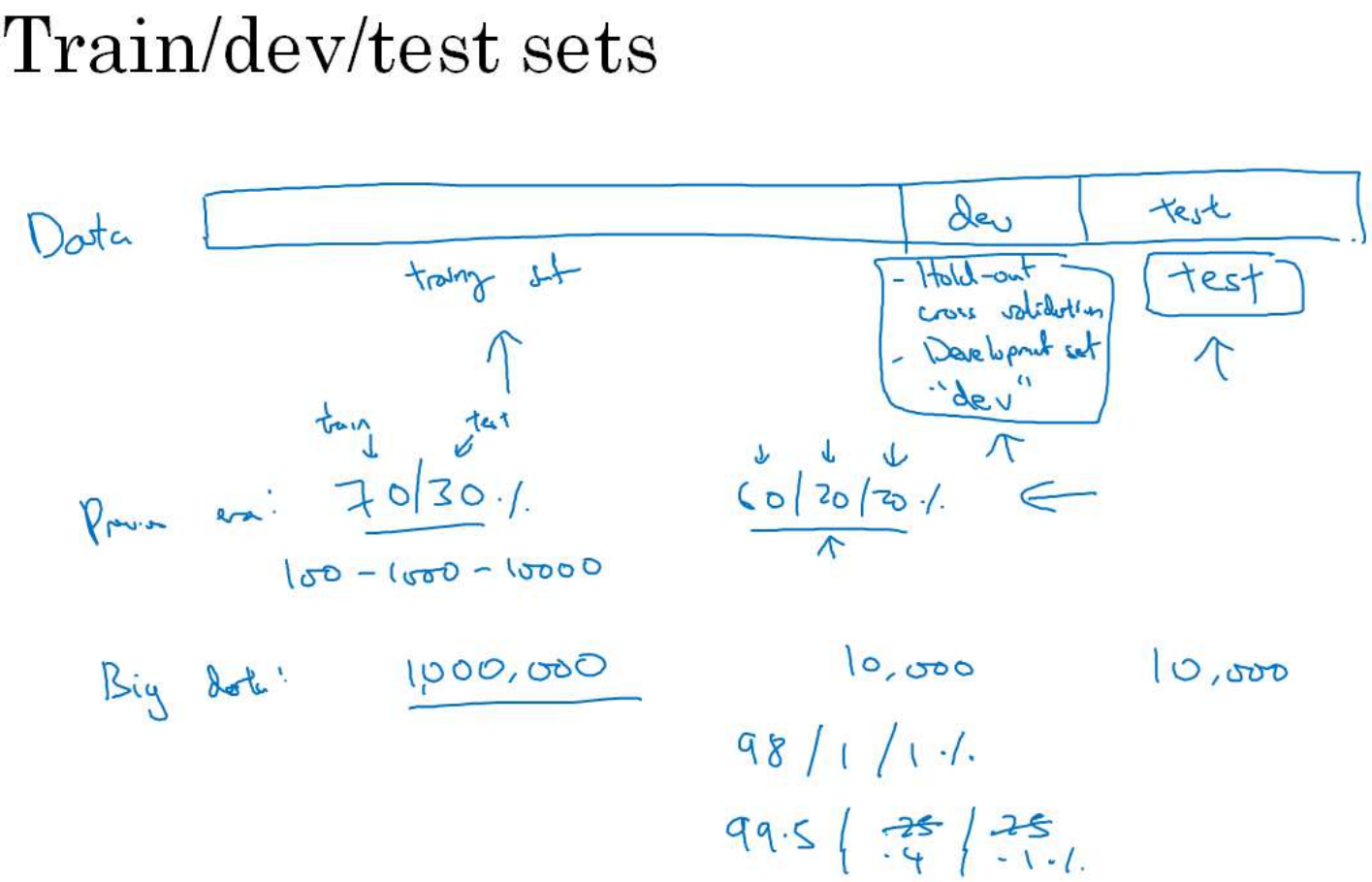
This course teaches about the practical aspects of deep learning. We just learned in last course how to implement a neural network. In this section we'll learn the practical aspects of how to make your neural network work well. Ranging from things like hyperparameter tuning to, how to set up your data, to how to make sure your optimization algorithm runs quickly so that you get your learning algorithm to learn in a reasonable time. In this section we'll first talk about the how to set a machine learning problem, then we'll talk about randomization. And we'll talk about some tricks for making sure your neural network implementation is correct.

Making good choices in how you set up your training, development, and test sets can make a huge difference in helping you quickly find a good high performance neural network. When training a neural network you have to make a lot of decisions, such as how many layers will your neural network have? And how many hidden units do you want each layer to have? And what's the learning rates? And what are the activation functions you want to use for the different layers? When you're starting on a new application, it's almost impossible to correctly guess the right values for all of these, and for other hyperparameter choices, on your first attempt. So in practice applied machine learning is a highly iterative process, in which you often start with an idea, such as you want to build a neural network of a certain number of layers, certain number of hidden units, maybe on certain data sets and so on. And then you just have to code it up and try it by running your code. You run and experiment and you get back a result that tells you how well this particular network, or this particular configuration works. And based on the outcome, you might then refine your ideas and change your choices and maybe keep iterating in order to try to find a better and a better neural network.

Today, deep learning has found great success in a lot of areas. Ranging from natural language processing to computer vision to speech recognition to a lot of applications on also structured data. And structured data includes everything from advertisements to web search, which isn't just Internet search engines it's also, for example, shopping websites. Already any websites that wants deliver great search results when you enter terms into a search bar. To computer security, to logistics, such as figuring out where to send drivers to pick up and drop off things, to many more. So what I'm seeing is that sometimes a researcher with a lot of experience in NLP might try to do something in computer vision. Or maybe a researcher with a lot of experience in speech

recognition might jump in and try to do something on advertising. Or someone from security might want to jump in and do something on logistics. And what I've seen is that intuitions from one domain or from one application area often do not transfer to other application areas. And the best choices may depend on the amount of data you have, the number of input features you have through your computer configuration and whether you're training on GPUs or CPUs. And if so, exactly what configuration of GPUs and CPUs, and many other things. So for a lot of applications I think it's almost impossible. Even very experienced deep learning people find it almost impossible to correctly guess the best choice of hyperparameters the very first time. And so today, applied deep learning is a very iterative process where you just have to go around this cycle many times to hopefully find a good choice of network for your application. So one of the things that determine how quickly you can make progress is how efficiently you can go around this cycle. And setting up your data sets well in terms of your train, development and test sets can make you much more efficient at that.

Let's see the below diagram:



So if this is your training data, then traditionally we might take all the data you have and carve off some portion of it to be your **training set**. Some portion of it to be your hold-out cross validation set, and this is sometimes also called the development set. And for brevity I'm just going to call this the **dev set**, but all of these terms mean roughly the same thing. And then you might carve out some final portion of it to be your **test set**. And so the workflow is that you keep on training algorithms on your training sets. And use your dev set or your hold-out **cross validation set** to see which of many different models performs best on your dev set. And then after having done this long enough, when you have a final model that you want to evaluate, you can take the best model you have found and evaluate it on your test set. In order to get an unbiased estimate of how well your algorithm is doing. So in the previous era of machine learning, it was common practice to take all your data and split it according to maybe a 70/30% in terms of a people often talk about the **70/30 train test splits**. If you don't have an explicit dev set or maybe a 60/20/20% split in terms of 60% train, 20% dev and 20% test. And several years ago this was widely considered best practice in machine learning. If you have maybe 100 examples in total, maybe 1000 examples in total, maybe after 10,000 examples. These sorts of ratios were perfectly reasonable rules of thumb. But in the modern big data era, where, for example, you might have a million examples in total, then the trend is that your dev and test sets have been becoming a much smaller percentage of the total. Because remember, the goal of the dev set or the development set is that you're going to test different algorithms on it and see which algorithm works better. So the dev set just needs to be big enough for you to evaluate, say, two different algorithm choices or ten different algorithm choices and quickly decide which one is doing better. And you might not need a whole 20% of your data for that. So, for example, if

you have a million training examples you might decide that just having 10,000 examples in your dev set is more than enough to evaluate which one or two algorithms does better. And in a similar vein, the main goal of your test set is, given your final classifier, to give you a pretty confident estimate of how well it's doing. And again, if you have a million examples maybe you might decide that 10,000 examples is more than enough in order to evaluate a single classifier and give you a good estimate of how well it's doing. So in this example where you have a million examples, if you need just 10,000 for your dev and 10,000 for your test, your ratio will be more like this 10,000 is 1% of 1 million so you'll have 98% train, 1% dev, 1% test. And I've also seen applications where, if you have even more than a million examples, you might end up with 99.5% train and 0.25% dev, 0.25% test. Or maybe a 0.4% dev, 0.1% test.

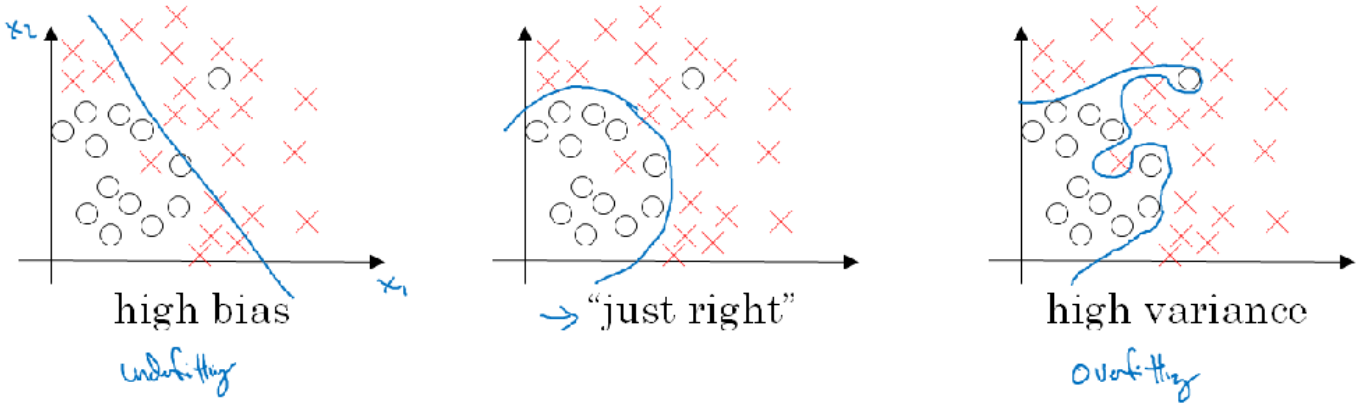
So just to recap, when setting up your machine learning problem, We'll often set it up into a train, dev and test sets, and if you have a relatively small dataset, these traditional ratios might be okay. But if you have a much larger data set, it's also fine to set your dev and test sets to be much smaller than your 20% or even 10% of your data. We'll give more specific guidelines on the sizes of dev and test sets later in this specialization. One other trend we're seeing in the era of modern deep learning is that more and more people train on mismatched train and test distributions. Let's say you're building an app that lets users upload a lot of pictures and your goal is to find pictures of cats in order to show your users. Maybe all your users are cat lovers. Maybe your training set comes from cat pictures downloaded off the Internet, but your dev and test sets might comprise cat pictures from users using our app. So maybe your training set has a lot of pictures crawled off the Internet but the dev and test sets are pictures uploaded by users. Turns out a lot of webpages have very high resolution, very professional, very nicely framed pictures of cats. But maybe your users are uploading blurrier, lower res images just taken with a cell phone camera in a more casual condition. And so these two distributions of data may be different. The rule of thumb I'd encourage you to follow in this case is to make sure that the **dev and test sets come from the same distribution**. We'll say more about this particular guideline as well, but because you will be using the dev set to evaluate a lot of different models and trying really hard to improve performance on the dev set. It's nice if your dev set comes from the same distribution as your test set. But because deep learning algorithms have such a huge hunger for training data, one trend I'm seeing is that you might use all sorts of creative tactics, such as crawling webpages, in order to acquire a much bigger training set than you would otherwise have. Even if part of the cost of that is then that your training set data might not come from the same distribution as your dev and test sets. But you find that so long as you follow this rule of thumb, that progress in your machine learning algorithm will be faster.

Finally, it might be okay to not have a test set. Remember the goal of the test set is to give you a unbiased estimate of the performance of your final network, of the network that you selected. But if you don't need that unbiased estimate, then it might be okay to not have a test set. So what you do, if you have only a dev set but not a test set, is you train on the training set and then you try different model architectures. Evaluate them on the dev set, and then use that to iterate and try to get to a good model. Because you've fit your data to the dev set, this no longer gives you an unbiased estimate of performance. But if you don't need one, that might be perfectly fine. In the machine learning world, when you have just a train and a dev set but no separate test set. Most people will call this a training set and they will call the dev set the test set. But what they actually end up doing is using the test set as a hold-out cross validation set. Which maybe isn't completely a great use of terminology, because they're then overfitting to the test set. So when the team tells you that they have only a train and a test set, I would just be cautious and think, do they really have a train dev set? Because they're overfitting to the test set. Culturally, it might be difficult to change some of these team's terminology and get them to call it a trained dev set rather than a trained test set. Even though I think calling it a train and development set would be more correct terminology. And this is actually okay practice if you don't need a completely unbiased estimate of the performance of your algorithm. So having set up a train dev and test set will allow you to integrate more quickly. It will also allow you to more efficiently measure the bias and variance of your algorithm so you can more efficiently select ways to improve your algorithm.

Bias/Variance

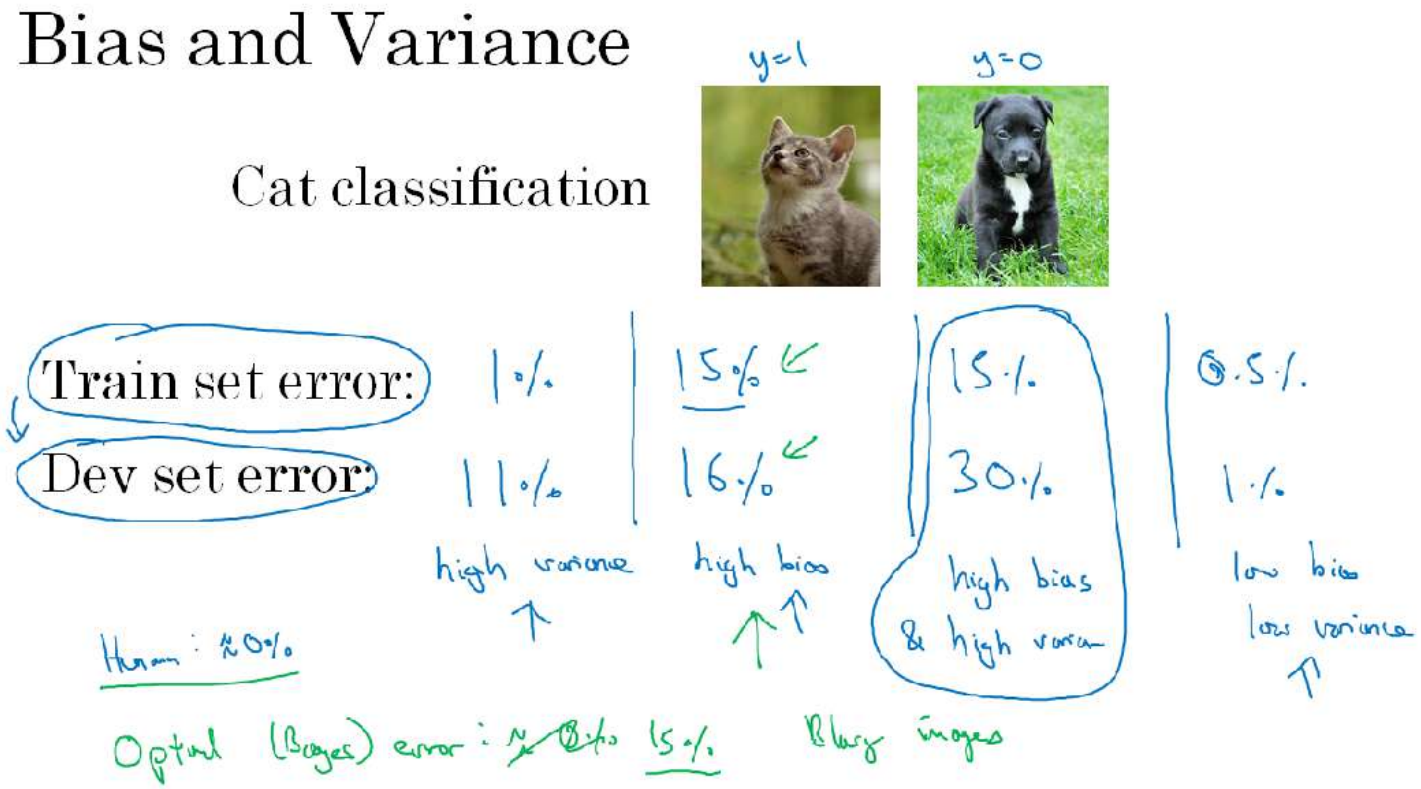
Bias and Variance is one of those concepts that's easily learned but difficult to master. Even if you think you've seen the basic concepts of Bias and Variance, there's often more new ones to it than you'd expect. In the Deep Learning era, another trend is that there's been less discussion of what's called the bias-variance trade-off. You might have heard this thing called the **bias-variance trade-off** but in Deep Learning era there's less of a trade-off, so we talk about the bias, we also talk about the variance but we just talk less about the **bias-variance trade-off**. Let's see what this means. Check the below diagram:

Bias and Variance



In first diagram, if you fit a straight line to the data, maybe you get a logistic regression fit to that. This is not a very good fit to the data and so this is class of a **high bias**, what we say that this is **underfitting** the data. On the opposite end (see 3rd diagram), if you fit an incredibly complex classifier, maybe a deep neural network, or neural network with all the hidden units, maybe you can fit the data perfectly, but that doesn't look like a great fit either. So there's a classifier of **high variance** and this is **overfitting** the data and there might be some classifier in between, with a medium level of complexity, that maybe fits it correctly like (see diagram 2). That looks like a much more reasonable fit to the data, so we call that just right. It's somewhere in between. So in a 2D example like this, with just two features, x1 and x2, we can plot the data and visualize bias and variance. In high dimensional problems, we can't plot the data and visualize decision boundary. Instead, there are couple of different metrics, which helps in understanding bias and variance.

So continuing our example of cat picture classification (check below diagram)



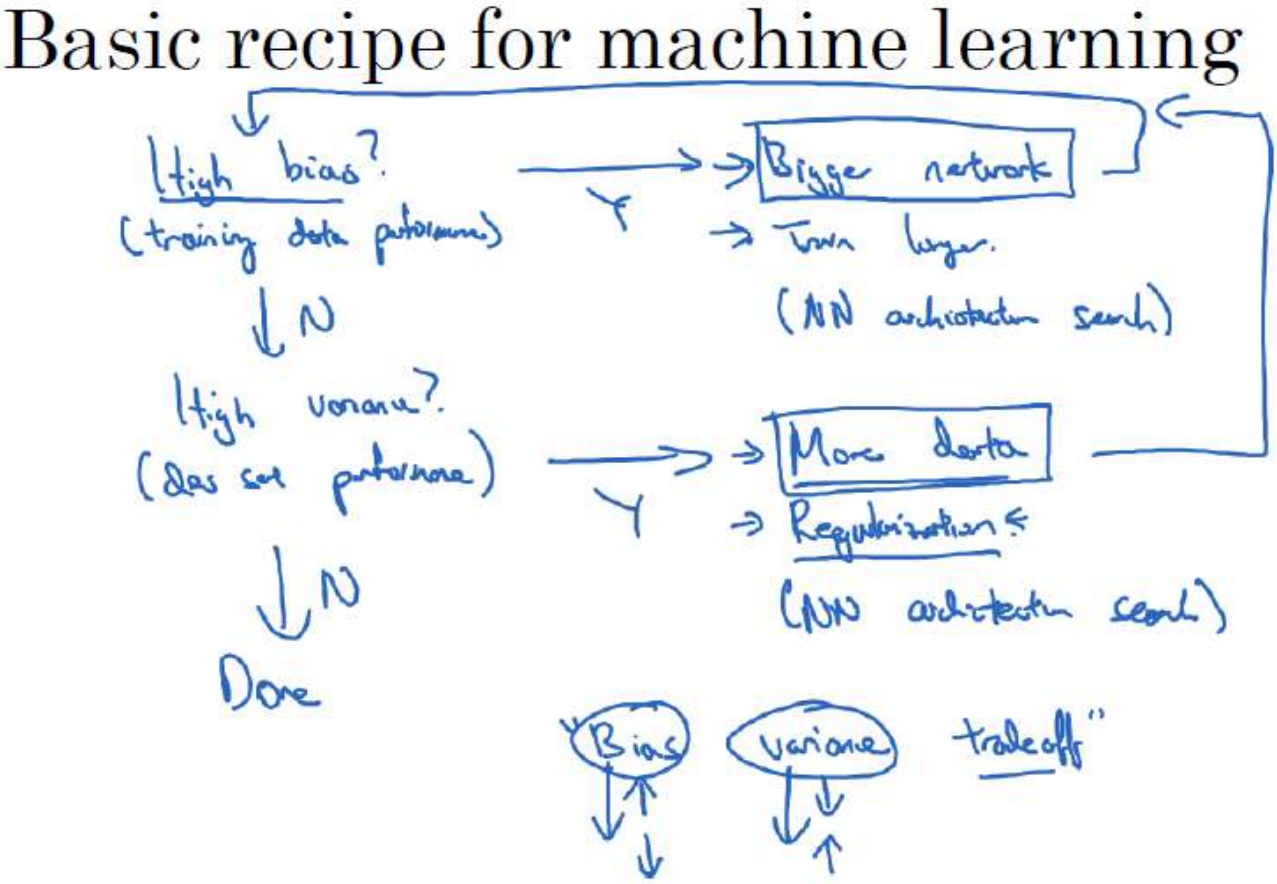
where found a cat is an positive example and found dog is a negative example, the two key numbers to look at to understand bias and variance will be the **train set error** and the **dev set** or the **development set error**. So for the sake of argument, let's say that you're recognizing cats in pictures, is something that people can do

nearly perfectly, right? So let's say, your training set error is 1% and your **dev set error** is, for the sake of argument, let's say is 11%. So in this example, you're doing very well on the training set, but you're doing relatively poorly on the development set. So this looks like you might have **overfit** the training set, that somehow you're not generalizing well, to this whole cross-validation set which is a development set. And so if you have an example like this, we would say this has **high variance**. So by looking at the training set error and the development set error, you would be able to render a diagnosis of your algorithm having high variance. Now, let's say, that you measure your training set and your dev set error, and you get a different result. Let's say, that your training set error is 15%. I'm writing your training set error in the top row, and your dev set error is 16%. In this case, assuming that humans achieve roughly 0% error, that humans can look at these pictures and just tell if it's cat or not, then it looks like the algorithm is not even doing very well on the training set. So if it's not even fitting the training data seam that well, then this is **underfitting** the data so this algorithm has high bias. But in contrast, this actually generalizing at a reasonable level to the dev set, whereas performance in the dev set is only 1% worse than performance in the training set. So this algorithm has a problem of high bias, because it was not even fitting the training set. Well, this is similar to the leftmost plots we had on the previous diagram.

Now, here's another example. Let's say that you have **15%** training set error, so that's pretty high bias, but when you evaluate to the dev set it does even worse, maybe it does **30%**. In this case, I would diagnose this algorithm as having high bias, because it's not doing that well on the training set, and high variance. So this has really the worst of both worlds. And one last example, if you have **0.5% training set error**, and **1% dev set error**, then maybe our users are quite happy, that you have a cat classifier with only 1% error, than just we have low bias and low variance. One subtle thing to mention is that this analysis is predicated on the assumption, that human level performance gets nearly **0%** error or, more generally, that the optimal error, sometimes called **bayes error**, so the base in optimal error is nearly 0%. It turns out that if the **optimal error** or the **bayes error** were much higher, say, it were **15%**, then if you look at this classifier, **15%** is actually perfectly reasonable for training set and you wouldn't see it as high bias and also a pretty low variance. So the case of how to analyze bias and variance, when no classifier can do very well, for example, if you have really blurry images, so that even a human or just no system could possibly do very well, then maybe bayes error is much higher, and then there are some details of how this analysis will change. But leaving aside this subtlety for now, the takeaway is that by looking at your training set error you can get a sense of how well you are fitting, at least the training data, and so that tells you if you have a bias problem. And then looking at how much higher your error goes, when you go from the training set to the dev set, that should give you a sense of how bad is the variance problem, so you'll be doing a good job generalizing from a training set to the dev set, that gives you sense of your variance. All this is under the assumption that the bayes error is quite small and that your training and your dev sets are drawn from the same distribution. So to summarize, we've seen how by looking at your algorithm's error on the training set and your algorithm's error on the dev set you can try to diagnose, whether it has problems of high bias or high variance, or maybe both, or maybe neither. And depending on whether your algorithm suffers from bias or variance, it turns out that there are different things you could try.

Basic recipe for machine learning

In the previous section, we saw how looking at training error and dev error can help you diagnose whether your algorithm has a bias or a variance problem, or maybe both. It turns out that this information that lets you much more systematically using what they call a **basic recipe for machine learning** and lets you much more systematically go about improving your algorithms' performance. Let's take a look.



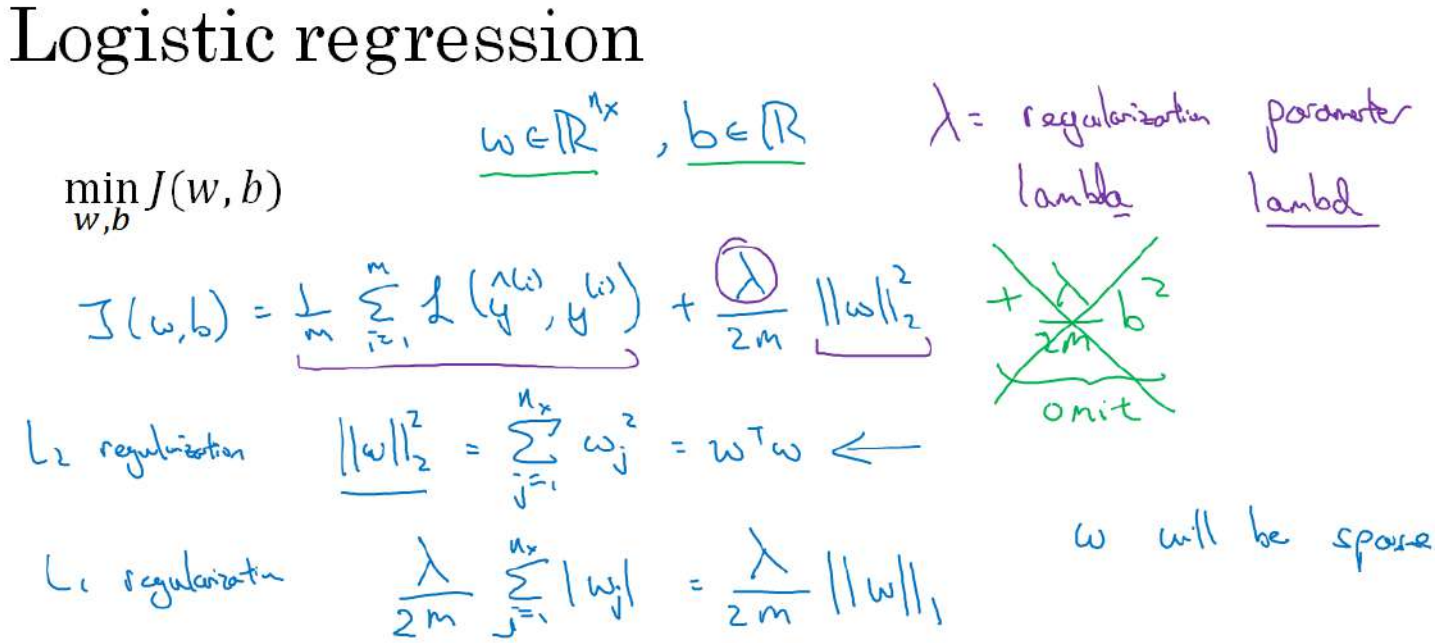
When training a neural network, here's a basic recipe we will use. After having trained an initial model, we will first ask, does our algorithm have high bias? And so to try and evaluate if there is high bias, you should look at, really, the training set or the training data performance and so, if it does have high bias, does not even fit in the training set that well, some things you could try would be to try pick a **network, such as more hidden layers or more hidden units, or you could train it longer. Maybe run trains longer or try some more advanced optimization algorithms,** we'll talk about optimization in upcoming sections or you can also try, not sure if works or not try but we can try a lot of different neural network architectures and maybe you can find a new network architecture that's better suited for this problem but there is no gurantee that this step will work but can be tried out. **Whereas getting a bigger network almost always helps and training longer doesn't always help, but it certainly never hurts**. So when training a learning algorithm, we should try these things until we can at least get rid of the bias problems, as in go back after we've tried this and keep doing that until we can fit, at least, fit the training set pretty well. And usually if you have a big enough network, you should usually be able to fit the training data well so long as it's a problem that is possible for someone to do. If the image is very blurry, it may be impossible to fit it. But if at least a human can do well on the task, if you think base error is not too high, then by training a big enough network you should be able to, hopefully, do well, at least on the training set. To at least fit or overfit the training set. Once you reduce bias to acceptable amounts then ask, do you have a variance problem? And so to evaluate that we would look at **dev set performance.** Are you able to generalize from a pretty good training set performance to having a pretty good dev set performance? And if you have high variance, well, best way to solve a **high variance problem is to get more data but** sometimes you can't get more data Or we can try regularization, which we'll talk about in the next section, to try to **reduce overfitting** but if you can find a more appropriate neural network architecture, sometimes that can reduce your variance problem as well, as well as reduce your bias problem. But how to do that? It's harder to be totally systematic how you do that. But so we can try these things and I kind of keep going back, until hopefully you find something with both low bias and low variance, where upon you would be done. So a couple of points to notice. First is that, depending on whether you have high bias or high variance, the set of things you should try could be quite different. So I'll usually use the training dev set to try to diagnose if you have a bias or variance problem, and then use that to select the appropriate subset of things to try. So for example, **if you actually have a high bias problem, getting more training data is actually not going to help.** Or at least it's not the most efficient thing to do. So being clear on how much of a bias problem or variance problem or both can help you focus on selecting the most useful things to try. Second, in the earlier era of machine learning, there used to be a lot of discussion on what is called the **bias variance**

tradeoff and the reason for that was that, for a lot of the things you could try, we can increase bias and reduce variance, or reduce bias and increase variance but back in the pre-deep learning era, we didn't have many tools, we didn't have as many tools that just reduce bias or that just reduce variance without hurting the other one. but in the modern deep learning, big data era, so long as you can keep training a bigger network, and so long as you can keep getting more data, which isn't always the case for either of these but if that's the case, then **getting a bigger network almost always just reduces your bias without necessarily hurting your variance, so long as you regularize appropriately and getting more data pretty much always reduces your variance and doesn't hurt your bias much**. So what's really happened is that, with these two steps, the ability to train, pick a network, or get more data, we now have tools to drive down bias and just drive down bias, or drive down variance and just drive down variance, without really hurting the other thing that much and I think this has been one of the big reasons that deep learning has been so useful for supervised learning, that there's much less of this tradeoff where you have to carefully balance bias and variance, but sometimes you just have more options for reducing bias or reducing variance without necessarily increasing the other one. And, in fact you have a well regularized network. We'll talk about regularization starting from the next section. Training a bigger network almost never hurts. And the main cost of training a neural network that's too big is just computational time, so long as you're regularizing. So I hope this gives you a sense of the basic structure of how to organize your machine learning problem to diagnose bias and variance, and then try to select the right operation for you to make progress on your problem. One of the things I mentioned several times in the section is **regularization**, is a **very useful technique for reducing variance**. There is a little bit of a bias variance tradeoff when you use regularization. It might increase the bias a little bit, although often not too much if you have a huge enough network.

Regularizing your neural network

Regularization

If you suspect your neural network is **over-fitting** your data this means that you have a high variance problem, one of the first things you should try is regularization. The other way to address high variance, is to get **more training data** that's also quite reliable. But you can't always get more training data, or it could be expensive to get more data. But adding regularization will often help to prevent overfitting, or to reduce the errors in your network. So let's see how regularization works. Let's develop these ideas using **logistic regression**.

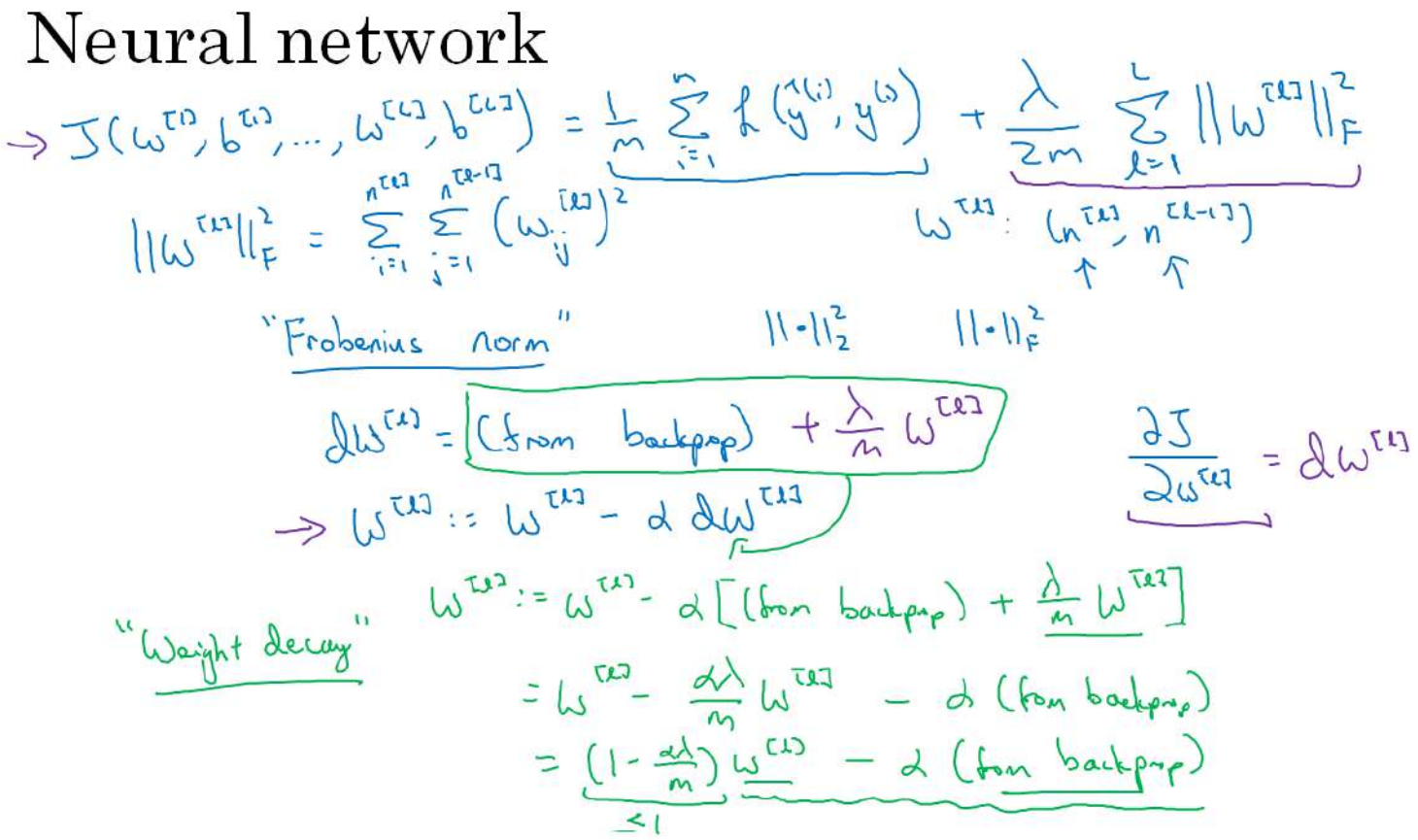


Recall that for logistic regression, we were trying to minimize the **cost function J**. Just to recall w and b in the logistic regression, are the parameters. So w is an x-dimensional parameter vector, and b is a real number. And so to add regularization to the logistic regression, what you do is add a thing called, **lambda**, which is called the **regularization parameter**. Check diagram for formula of regularization which is also called **L2 regularization**.

Now, why do you regularize just the parameter w? Why don't we add something here about b as well? In practice, you could do this, but I usually just omit this. Because if you look at your parameters, **w is usually a pretty high dimensional parameter vector**, **especially with a high variance problem**. Maybe w just has a lot of parameters, so you aren't fitting all the parameters well, whereas **b is just a single number**. So almost all the parameters are in w rather b. And if you add this last term, in practice, it won't make much of a difference, because b is just one parameter over a very large number of parameters. In practice, I usually just

don't bother to include it. But you can if you want. So **L2 regularization is the most common type of regularization**. You might have also heard of some people talk about **L1 regularization**. And that's when you add, instead of this L2 norm, you instead add a term that is lambda/m of sum over of this. And this is also called the L1 norm of the parameter vector w, I guess whether you put m or 2m in the denominator, is just a scaling constant. **If you use L1 regularization, then w will end up being sparse.** And what that means is that the **w vector will have a lot of zeros in it.** And some people say that this can help with compressing the model, because the set of parameters are zero, and you need less memory to store the model. Although, I find that, in practice, L1 regularization to make your model sparse, helps only a little bit. So I don't think it's used that much, at least not for the purpose of compressing your model. And when people train your networks, L2 regularization is just used much much more often. So just to add to notation ,**Lambda here is called the regularization, Parameter** and usually, you set this using **your development set, or using cross validation**. When you a variety of values and see what does the best, in terms of trading off between doing well in your training set versus also setting that two normal of your parameters to be small. Which helps prevent over fitting. So **lambda is another hyper parameter** that you might have to tune.

So this is how you implement L2 regularization for logistic regression. How about a neural network?



In a neural network, we have a cost function that's a function of all of our parameters, $w^{[1]}$, $b^{[1]}$ through $w^{[L]}$, $b^{[L]}$, where capital L is the number of layers in your neural network. Check above diagram for formula. So **the cost function, sum of the losses, summed over your m training examples. And says at regularization, you add lambda over 2m of sum over all of your parameters W, your parameter matrix is w, of their, that's called the squared norm**. Where this norm of a matrix, meaning the squared norm is defined as the sum of the i sum of j, of each of the elements of that matrix, squared.? Previously, we would complete **dw using backprop**, where backprop would give us the partial derivative of J with respect to w, or really w for any given $^{[l]}$. And then you update $w^{[l]}$, as $w^{[1]}$ - the learning rate times d. Check the diagram above. **L2 regularization** is sometimes also called **weight decay** because it's just like the ordinally gradient descent, where you update w by subtracting alpha times the original gradient you got from **backprop**.

Why regularization reduces overfitting?

Why does regularization help with overfitting? Why does it help with reducing variance problems? Few methods we can try:

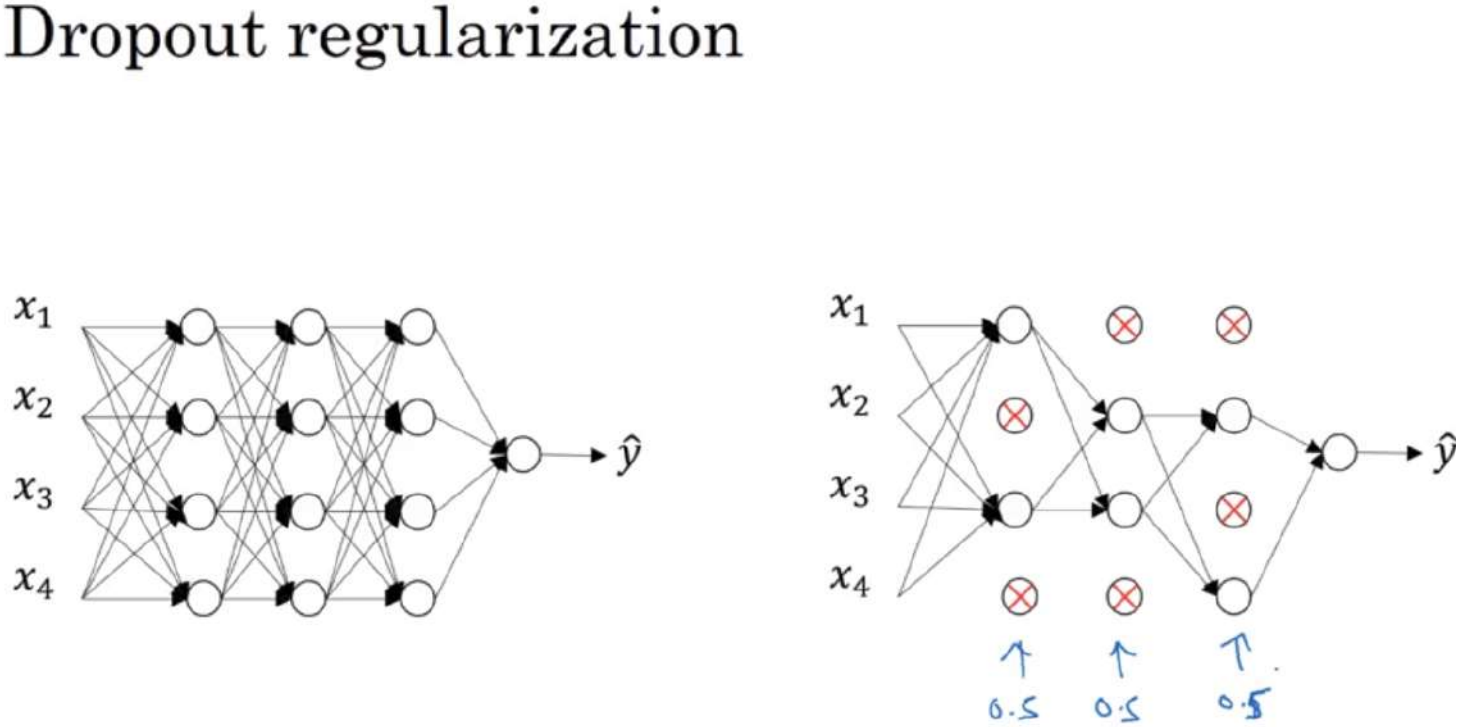
- Set the weight matrices W to be **reasonably close to zero**. So one piece of intuition is maybe it set the weight to be so close to zero for a lot of hidden units that's basically zeroing out a lot of the impact of these hidden units. And if that's the case, then this much simplified neural network becomes a much

smaller neural network. In fact, it is almost like a logistic regression unit, but stacked most probably as deep. And so that will take you from this overfitting case much closer to the left to other high bias case.

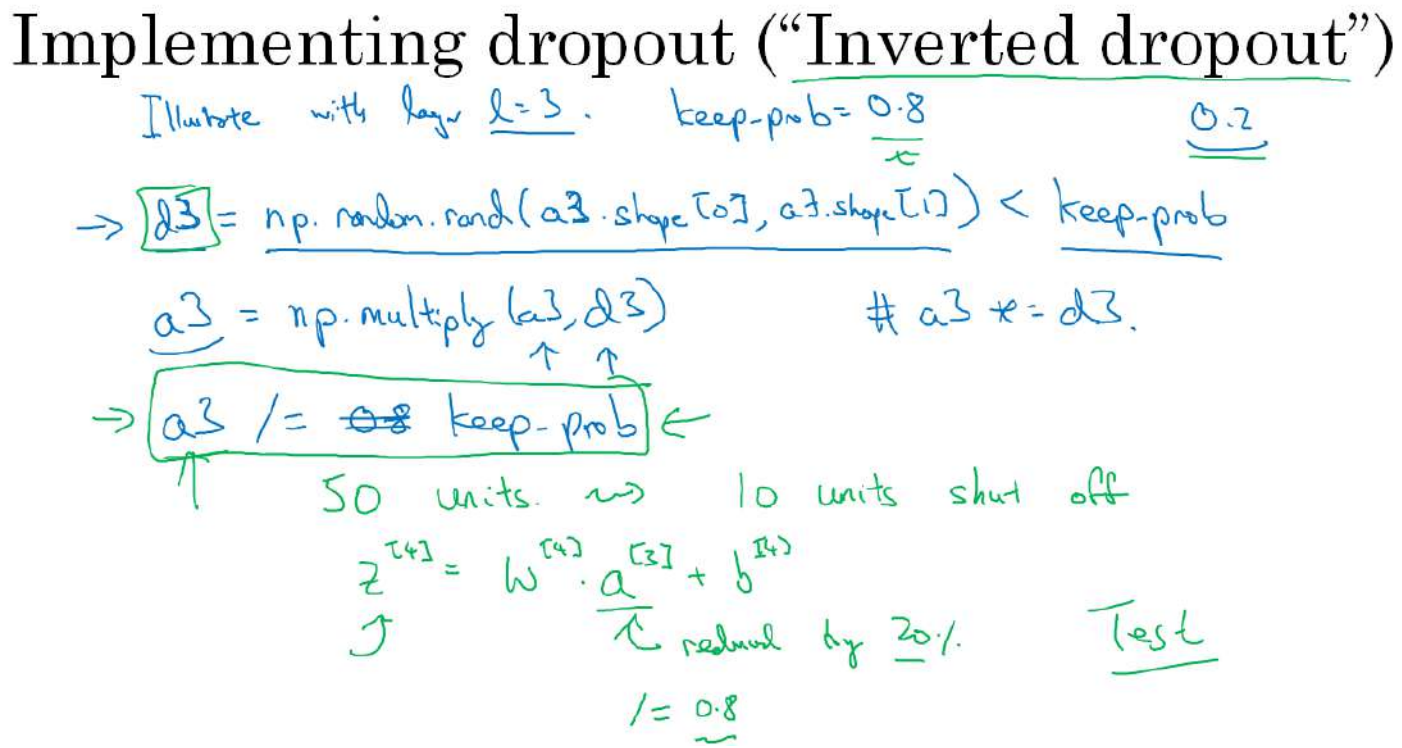
- Another trial we can do by using the **tanh**. If the regularization becomes very large, the parameters W very small, so Z will be relatively small, kind of ignoring the effects of b for now, so Z will be relatively small or, really it takes on a small range of values. And so the activation function if is **tanh**, say, will be relatively linear. And so your whole neural network will be computing something not too far from a **big linear function** which is therefore pretty simple function rather than a very complex highly non-linear function. And so is also much less able to **overfit**.

Dropout regularization

In addition to L2 regularization, another very powerful regularization techniques is called "**dropout**." Let's see how that works. Let's say you train a neural network like the one on the left and there's over-fitting. (Check the diagram below)



Here's what you do with dropout. Let me make a copy of the neural network. **With dropout, what we're going to do is go through each of the layers of the network and set some probability of eliminating a node in neural network**. Let's say that for each of these layers, we're going to- for each node, toss a coin and have a 0.5 chance of keeping each node and 0.5 chance of removing each node. So, after the coin tosses, maybe we'll decide to eliminate those nodes, then what you do is actually remove all the outgoing things from that no as well. So you end up with a much smaller, really much diminished network. And then you do **back propagation training**. There's one example on this much diminished network. And then on different examples, you would toss a set of coins again and keep a different set of nodes and then dropout or eliminate different than nodes. And so for each training example, you would train it using one of these neural based networks. So, maybe it seems like a slightly crazy technique. They just go around coding those are random, but this **actually works**. But you can imagine that because you're training a much smaller network on each example or maybe just give a sense for why you end up able to regularize the network, because these much smaller networks are being trained. Let's look at how you implement dropout. There are a few ways of implementing dropout. the most common one, which is technique called **inverted dropout**. Check diagram below:

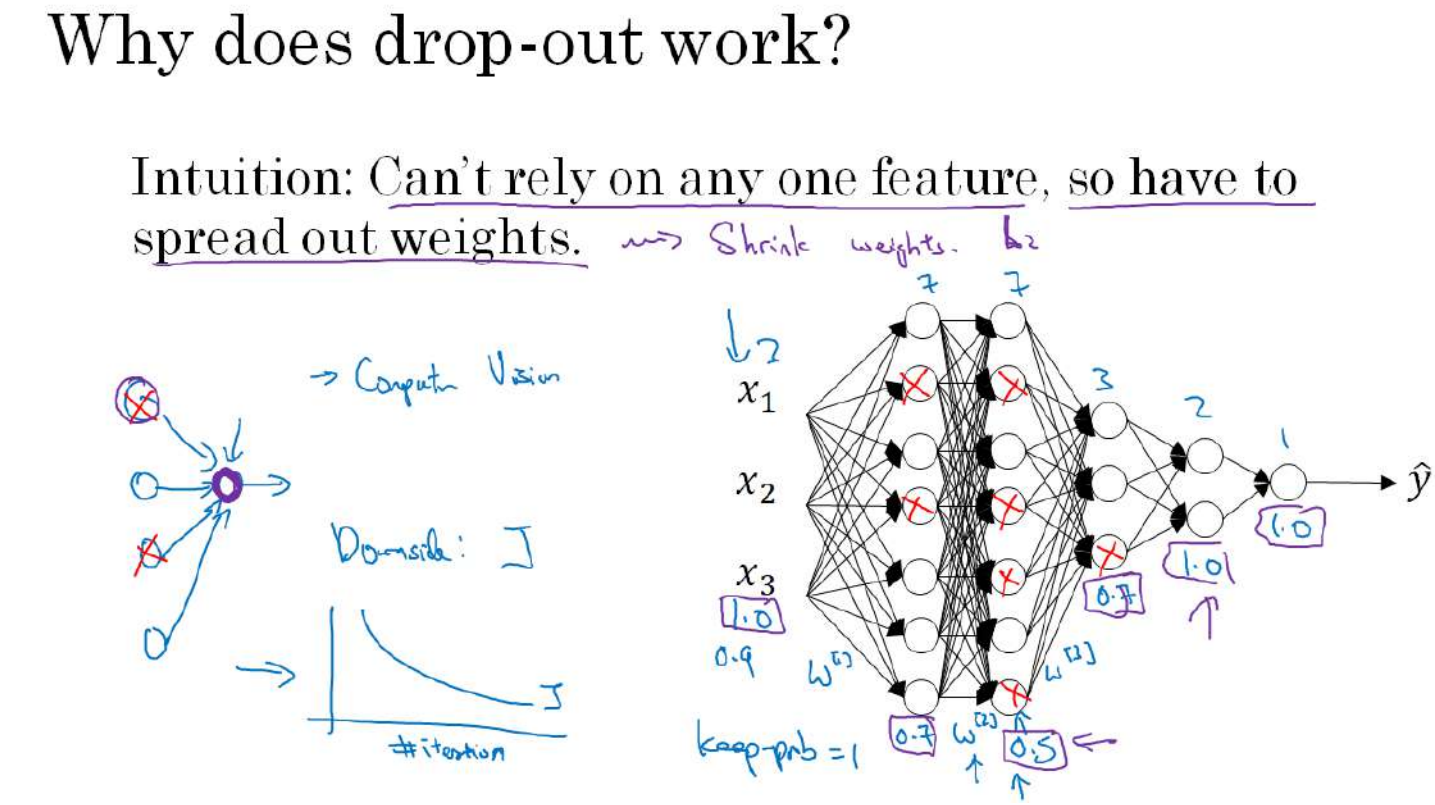


The last line is what's called the **inverted dropout technique** and its effect is that, no matter what you set to keep.prob to, whether it's 0.8 or 0.9 or even one, if it's set to one (1) then there's no dropout, because it's keeping everything or 0.5 or whatever, this **inverted dropout technique by dividing by the keep.prob**, it ensures that the expected value of a3 remains the same. And it turns out that at test time, when you trying to evaluate a neural network, this inverted dropout technique makes test time easier because you have less of a scaling problem. By far the most common implementation of dropouts today as far as I know is inverted dropouts.

Understanding Dropout

Drop out does this seemingly crazy thing of randomly knocking out units on your network. Why does it work so well with a regularizer? Let's gain some better intuition. In the previous section, we saw the intuition that drop-out randomly knocks out units in your network. So it's as if on every iteration you're working with a

smaller neural network, and so using a smaller neural network seems like it should have a **regularizing effect**.



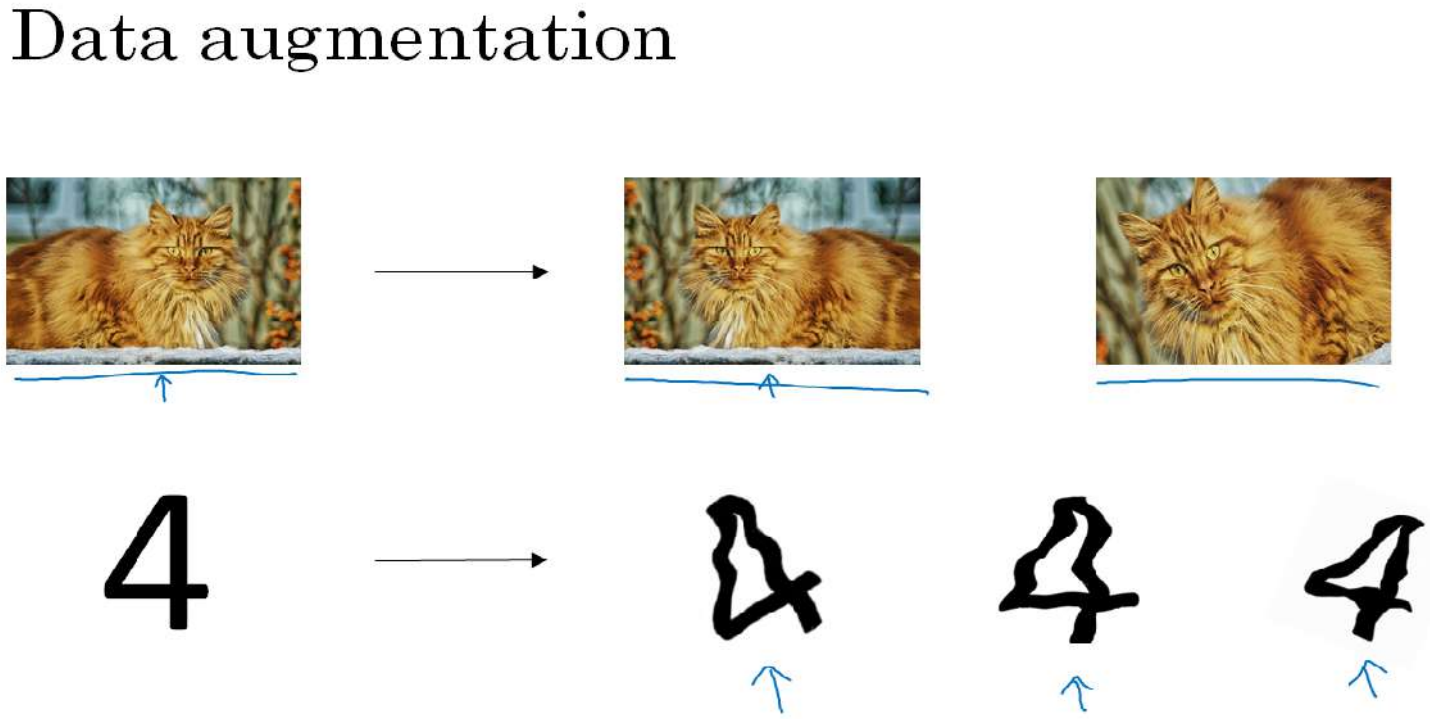
Here's a **Second intuition** which is, let's look at it from the perspective of a single unit. Let's take one unit, now, for this unit to do his job as for inputs and it needs to generate some meaningful output. Now with drop out, the inputs can get randomly eliminated. Sometimes other two units will get eliminated, sometimes a different unit will get eliminated. So, if we take example of a unit in diagram, it can't rely on any one feature because any one feature could go away at random or any one of its own inputs could go away at random. Some one would be reluctant to put all of its bets on, say, just one input, right? The weights, we're reluctant to put too much weight on any one input because it can go away. So this unit will be more motivated to spread out this way and give you a little bit of weight to each of the four inputs to this unit. And by spreading all the weights, this will tend to have an effect of shrinking the squared norm of the weights. And so, similar to what we saw with **L2 regularization**, the effect of implementing **drop out is that it shrinks the weights and does some of those outer regularization that helps prevent over-fitting.** To summarize, it is possible to show that drop out has a similar effect to L2 regularization. Only to L2 regularization applied to different ways can be a little bit different and even more adaptive to the scale of different inputs.

One more detail for when you're implementing drop out. See the diagram where we have three input features. This is seven hidden units here, seven, three, two, one. So, one of the parameters we had to choose was the **keep-prob** which has a chance of keeping a unit in each layer. So, it is also feasible to vary **keep-prob** by layer so there could be different **keep-prob** for different layers. Notice that the **keep-prob** of 1.0 means that you're keeping every unit and so, you're really not using drop out for that layer. But for layers where you're more worried about over-fitting, really the layers with a lot of parameters, you can set the **keep-prob** to be smaller to apply a more powerful form of drop out. It's kind of like cranking up the regularization parameter **lambda of L2 regularization** where you try to regularize some layers more than others. And technically, you can also apply drop out to the input layer, where you can have some chance of just maxing out one or more of the input features. Although in practice, usually don't do that that often. And so, a key prop of 1.0 was quite common for the input there. You can also use a very high value, maybe 0.9, but it's much less likely that you want to eliminate half of the input features. **So just to summarize**, if you're more worried about some layers overfitting than others, you can set a lower key prop for some layers than others. The downside is, this gives you even more hyper parameters to search for using cross-validation. One other alternative might be to have some layers where you apply drop out and some layers where you don't apply drop out and then just have one hyper parameter, which is a **keep-prob** for the layers for which you do apply drop outs. Some implementational tips-> Many of the first successful implementations of drop outs were to **computer vision**. So in computer vision, the input size is so big, inputting all these pixels that you almost never have enough data. And so drop out is very frequently used by computer vision. And there's some **computer vision researchers that pretty much always use it**, almost as a default. But really the thing to remember is that **drop out is a regularization technique**, **it helps prevent over-fitting**. And so, unless my algorithm is over-fitting, I

wouldn't actually bother to use drop out. So it's used somewhat less often than other application areas. There's just with computer vision, you usually just don't have enough data, so you're almost always overfitting, which is why there tends to be some computer vision researchers who swear by drop out. But their intuition doesn't always generalize I think to other disciplines. **One big downside of drop out is that the cost function J is no longer well-defined**. On every iteration, you are randomly killing off a bunch of nodes. And so, if you are double checking the performance of grade descent, it's actually harder to double check that you have a well defined **cost function J** that is going downhill on every iteration. Because the cost function J that you're optimizing is actually less. Less well defined, or is certainly hard to calculate. So what we can try usually is to turn off drop out, means **keep-prob = 1.0** and then run the code code and make sure that it is **monotonically decreasing J**, and then turn on drop out and hope that I didn't introduce bugs into my code during drop out.

Other regularization methods

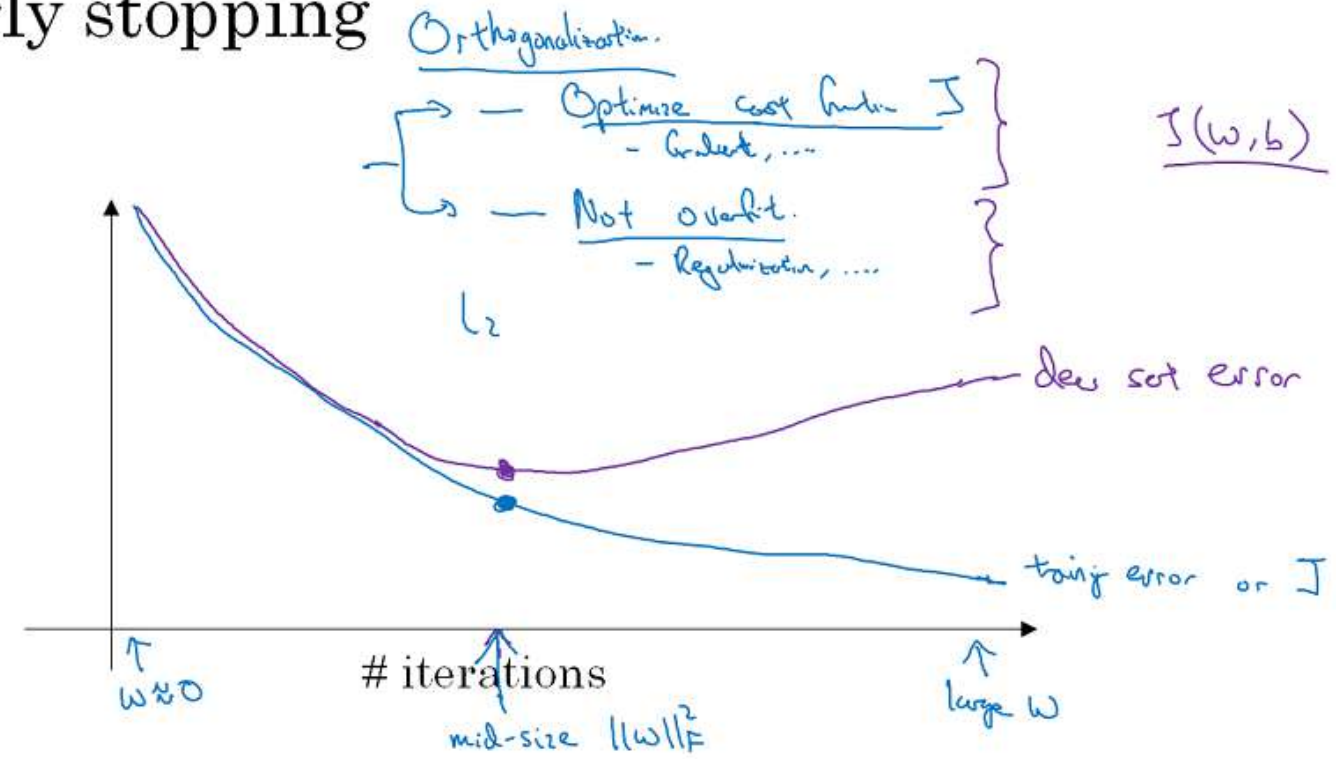
In addition to **L2 regularization** and **drop out regularization** there are few other techniques to reducing over fitting in your neural network. Let's take a look. Let's say you fitting a cat classifier, If you are **over fitting getting more training data can help**, but getting more training data can be expensive and sometimes you just can't get more data. But what you can do is **augment your training set** (Check the image below).



And for example, flipping it horizontally and adding that also with your training set. So now instead of just this one example in your training set, you can add this to your training example. So by flipping the images horizontally, you could double the size of your training set. Because you're training set is now a bit redundant this isn't as good as if you had collected an additional set of brand new independent examples. But you could do this Without needing to pay the expense of going out to take more pictures of cats. And then other than flipping horizontally, you can also take random crops of the image. So here we're rotated and sort of randomly zoom into the image and this still looks like a cat. So by taking random distortions and translations of the image you could augment your data set and make additional fake training examples. Again, these extra fake training examples they don't add as much information as they were to call they get a brand new independent example of a cat. But because you can do this, almost for free, other than for some confrontational costs. This can be an **inexpensive way to give your algorithm more data and therefore sort of regularize it and reduce over fitting**. And by synthesizing examples like this what you're really telling your algorithm is that If something is a **cat then flipping it horizontally is still a cat**. Notice we didn't flip it vertically, because maybe we don't want upside down cats, right? And then also maybe randomly zooming in to part of the image it's probably still a cat. For optical character recognition you can also bring your data set by taking digits and imposing random rotations and distortions to it. So If you add these things to your training set, these are also still digit force. So data augmentation can be used as a **regularization technique**, in fact similar to regularization.

There's one other technique that is often used called **early stopping**. So what you're going to do is as you run gradient descent you're going to plot your, either the training error, you'll use 0/1 classification error on the training set. Or just plot the cost function J optimizing, and that should decrease monotonically, so with **early stopping**, we plot like shown below:

Early stopping

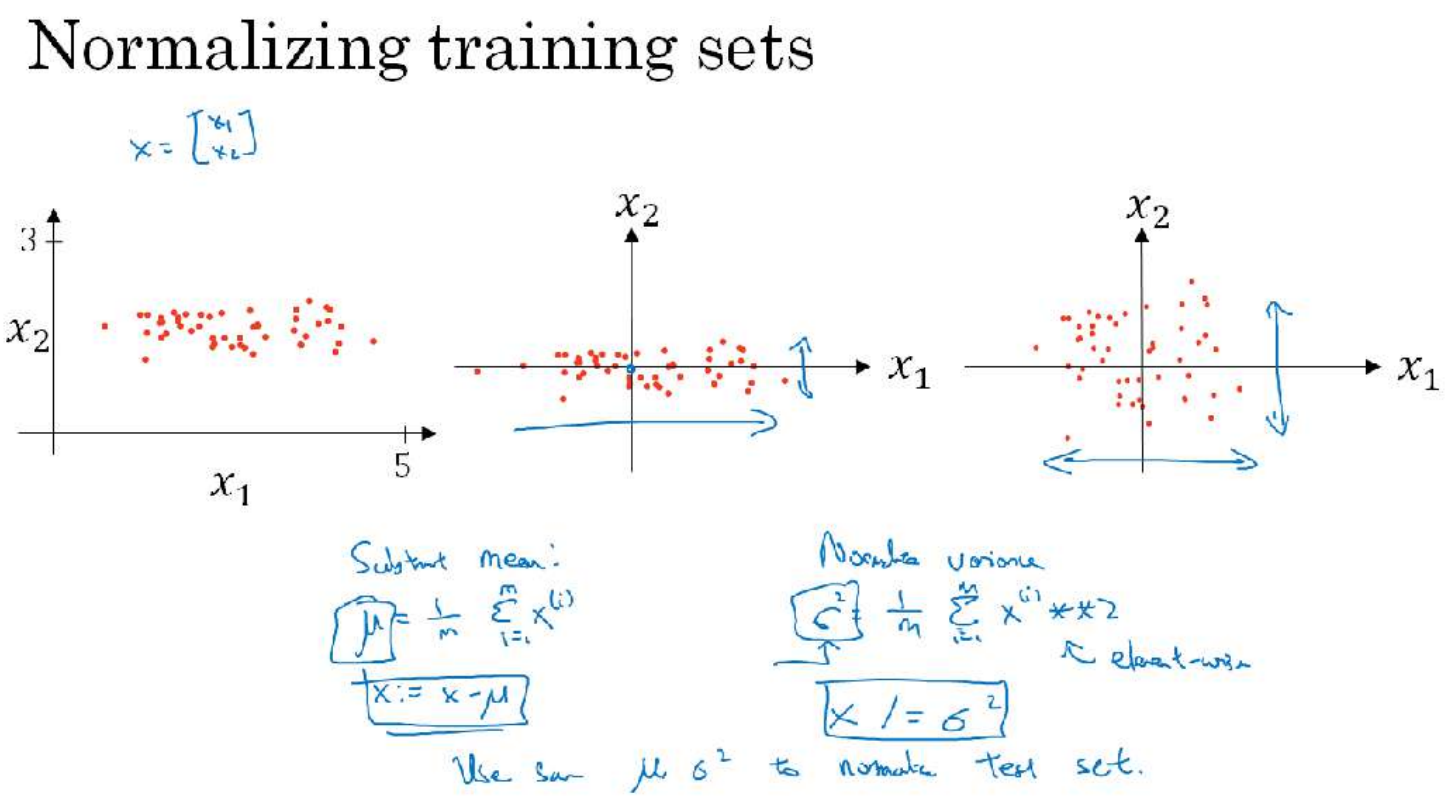


And again, this could be a classification error in a development sense, or something like the cost function, like the logistic loss or the log loss of the dev set. Now what you find is that your dev set error will usually go down for a while, and then it will increase from there.

So what early stopping does is that that look for the iteration where NN is doing best and stop trading on neural network halfway and take whatever value achieved this dev set error. So why does this work? Well when we've haven't run many iterations for our neural network yet our parameters **w** will be close to zero. Because with random initialization you probably initialize w to small random values so before you train for a long time, w is still quite small. And as you iterate, as you train, w will get bigger and bigger and bigger until here maybe you have a much larger value of the parameters w for your neural network. **So what early stopping does is by stopping halfway you have only a mid-size rate w**. And so similar to L2 regularization by picking a neural network with smaller norm for your parameters w, hopefully your neural network is over fitting less. And the term **early stopping refers to the fact that you're just stopping the training of your neural network earlier.** There is a downside of early stopping too, machine learning process as comprising several different steps. One, is that you want an algorithm to optimize the cost function J and we have various tools to do that, such as **gradient descent** and then also other algorithms, like **momentum** and **RMS prop** and **Atom** and so on. But after optimizing the cost function J, you also wanted to **not over-fit**. And we have some tools to do that such as your **regularization, getting more data** and so on. Now in machine learning, we already have so many hyper-parameters it surge over. It's already very complicated to choose among the space of possible algorithms. And so I find in machine learning easier to think about when you have one set of tools for **optimizing the cost function J,** and when you're focusing on optimizing the cost function J. All you care about is finding w and b, so that **J(w,b) is as small as possible**. You just don't think about anything else other than reducing this. And then it's completely separate task to not over fit, in other words, to **reduce variance**. And when you're doing that, you have a separate set of tools for doing it. And this principle is sometimes called **orthogonalization**. And there's this idea, that **you want to be able to think about one task at a time**. I'll say more about orthorganization in a later sections, so if you don't fully get the concept yet, don't worry about it. But, to me the main downside of early stopping is that this couples these two tasks. So you no longer can work on these two problems independently, because by stopping gradient decent early, you're sort of breaking whatever you're doing to optimize cost function J, because now you're not doing a great job reducing the cost function J. You've sort of not done that that well. And then you also simultaneously trying to not over fit. So instead of using different tools to solve the two problems, you're using one that kind of mixes the two. And this just makes the set of things you could try are more complicated to think about. **Rather than using early stopping, one alternative is just use L2 regularization** then you can just train the neural network as long as possible. I find that this makes the search space of hyper parameters easier to decompose, and easier to search over. But the downside of this though is that you might have to try a lot of values of the regularization parameter lambda. And so this makes searching over many values of lambda more computationally expensive. And the advantage of **early stopping is that running the gradient descent**

process just once, you get to try out values of small w, mid-size w, and large w, without needing to try a lot of values of the L2 regularization hyperparameter lambda.

Setting up your optimization problem
Normalizing inputs

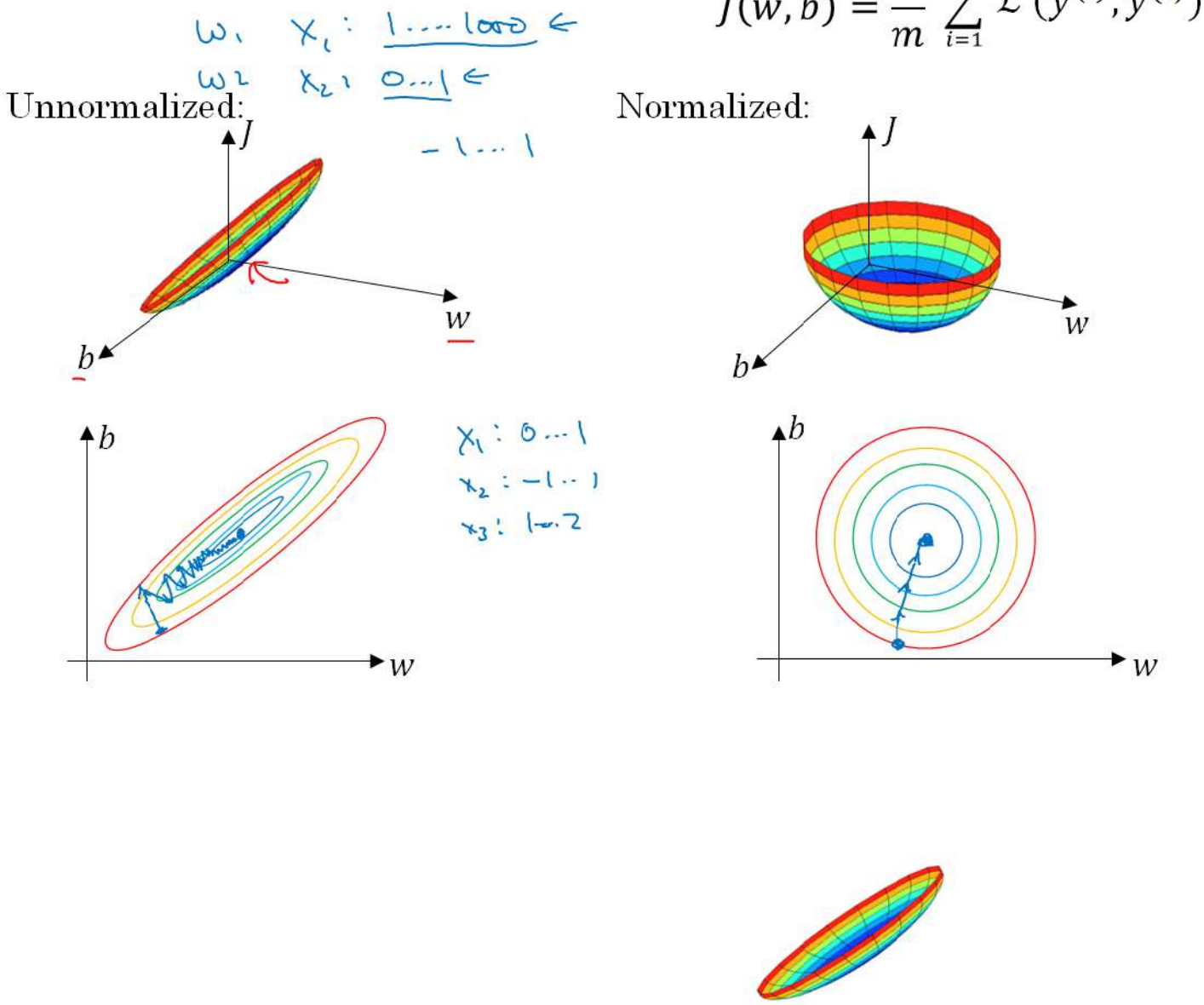


When training a neural network, one of the techniques that will speed up your training is if you normalize your inputs. Let's see what that means. Let's see if a training sets with two input features. So the input features x are two dimensional, and check the diagram for a scatter plot of our training set. Normalizing your inputs corresponds to two steps. The first is to subtract out or to zero out the mean. So you set mu = 1 over M sum over I of $X^{[i]}$. So this is a vector, and then X gets set as X- mu for every training example, so this means you just move the training set until it has 0 mean. And then the second step is to normalize the variances. So notice here that the feature X1 has a much larger variance than the feature X2 here. So what we do is set sigma = 1 over m sum of Xi**2. I guess this is a element y squaring. And so now sigma squared is a vector with the variances of each of the features, and notice we've already subtracted out the mean, so Xi squared, element y squared is just the variances. And you take each example and divide it by this vector sigma squared. And so in pictures, you end up with this. Where now the variance of X1 and X2 are both equal to one.
And one tip, if you use this to scale your training data, then use the same mu and sigma squared to normalize your test set, right? In particular, you don't want to normalize the training set and the test set differently. Whatever this value is and whatever this value is, use them in these two formulas so that you scale your test set in exactly the same way, rather than estimating mu and sigma squared separately on your training set and test set. Because you want your data, both training and test examples, to go through the same transformation defined by the same mu and sigma squared calculated on your training data. So, why do we do this? **Why do**

we want to normalize the input features?

Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

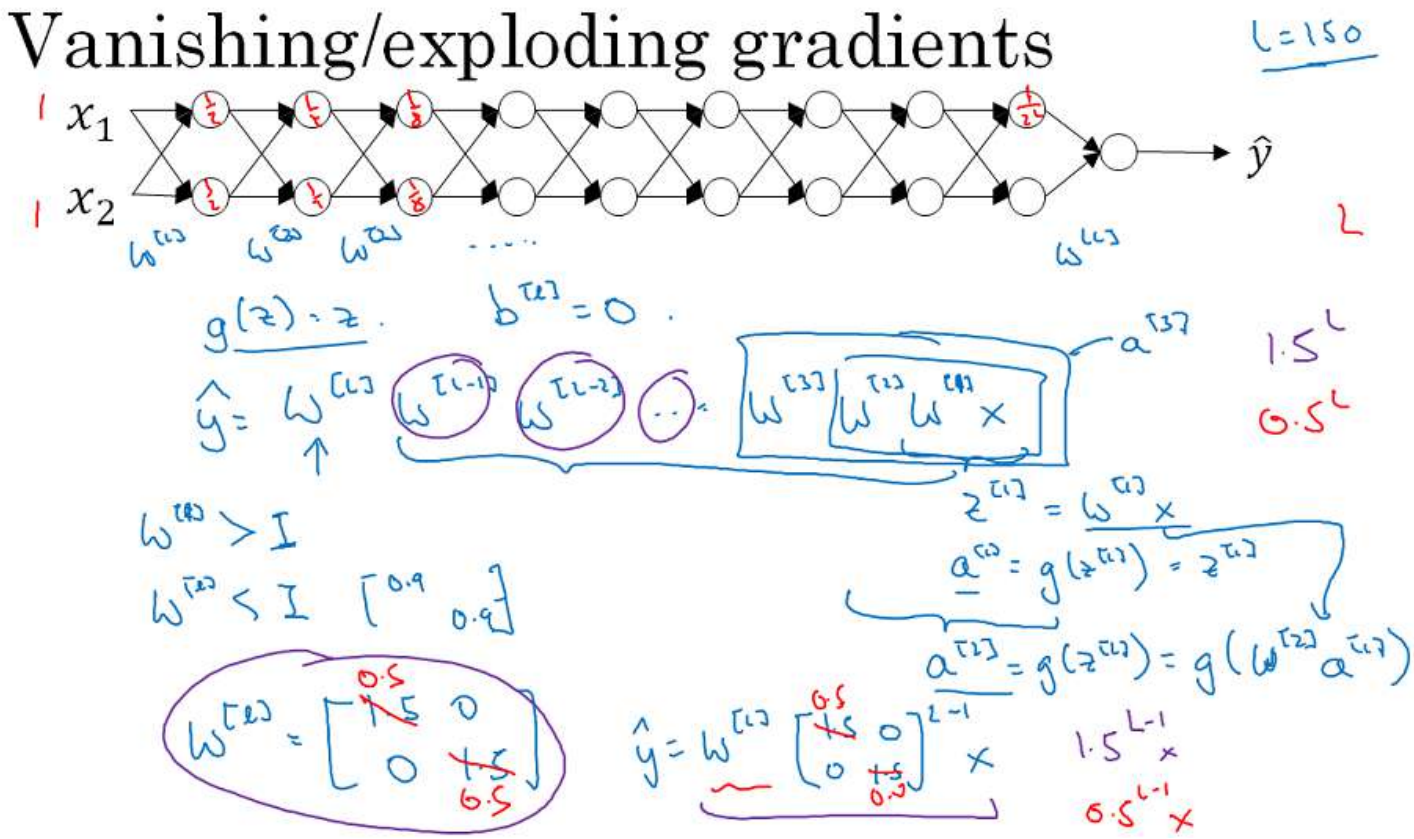


Recall that a cost function is defined as written on the top right. It turns out that if you use unnormalized input features, it's more likely that your cost function will look like this, it's a very squished out bowl, very elongated cost function, where the minimum you're trying to find is maybe over there. But if your features are on very different scales, say the feature X1 ranges from 1 to 1,000, and the feature X2 ranges from 0 to 1, then it turns out that the ratio or the range of values for the parameters w1 and w2 will end up taking on very different values. And so maybe these axes should be w1 and w2, but I'll plot w and b, then your cost function can be a very elongated bowl like that. So if you part the contours of this function, you can have a very elongated function like that. Whereas if you normalize the features, then your cost function will on average look more symmetric. And if you're running gradient descent on the cost function like the one on the left, then you might have to use a very small learning rate because if you're here that gradient descent might need a lot of steps to oscillate back and forth before it finally finds its way to the minimum. Whereas if you have a more spherical contours, then wherever you start gradient descent can pretty much go straight to the minimum. You can take much larger steps with gradient descent rather than needing to oscillate around like like the picture on the left. Of course in practice w is a high-dimensional vector, and so trying to plot this in 2D doesn't convey all the intuitions correctly. But the rough intuition that your cost function will be more round and easier to optimize when your features are all on similar scales. Not from one to 1000, zero to one, but mostly from minus one to one or of about similar variances of each other. **That just makes your cost function J easier and faster to optimize.** In practice if one feature, say X1, ranges from zero to one, and X2 ranges from minus one to one, and X3 ranges from one to two, these are fairly similar ranges, so this will work just fine. It's when they're on dramatically different ranges like ones from 1 to a 1000, and the another from 0 to 1, that that really hurts your optimization algorithm. But by just setting all of them to a 0 mean and say, variance 1, like we did in the last

part of --, that just guarantees that all your features on a similar scale and will usually help your learning algorithm run faster. So, if your input features came from very different scales, maybe some features are from 0 to 1, some from 1 to 1,000, then it's important to normalize your features. If your features came in on similar scales, then this step is less important. Although performing this type of normalization pretty much never does any harm, so I'll often do it anyway if I'm not sure whether or not it will help with speeding up training for your algebra.

Vanishing/exploding gradients

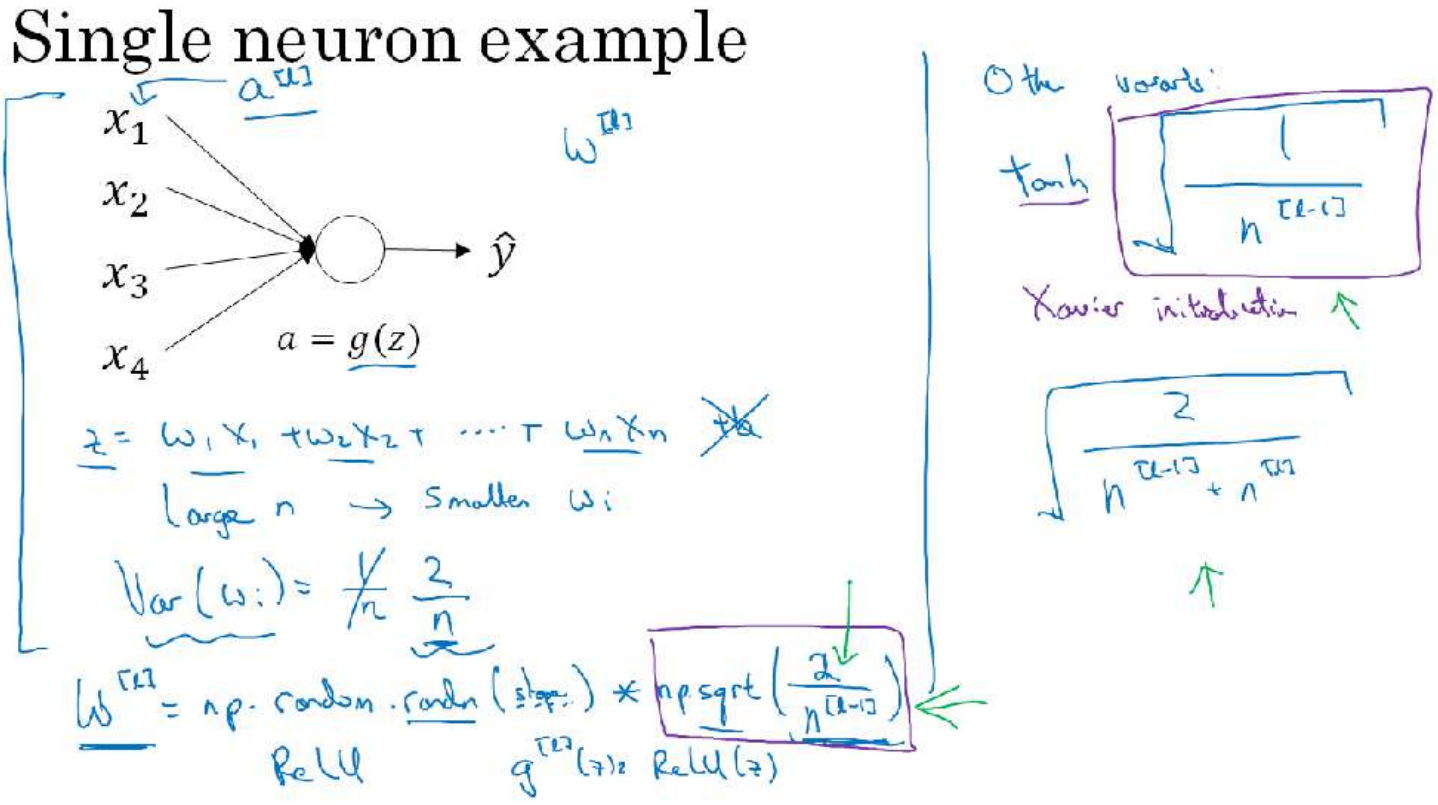
One of the problems of training neural network, especially very deep neural networks, is data **vanishing and exploding gradients**. What that means is that when you're training a very deep network your derivatives or your slopes can sometimes get either very, very big or very, very small, maybe even exponentially small, and this makes training difficult. In this section you will see what this problem of exploding and vanishing gradients really means, as well as how you can use careful choices of the **random weight initializatio**n to significantly reduce this problem. Check the diagram:



So the intuition you can take away from this is that at the weights W, if they're all just a **little bit bigger** than one or just a little bit bigger than the identity matrix, then with a very deep network the activations can explode. And if **W is just a little bit less** than identity. So this maybe here's 0.9, 0.9, then you have a very deep network, the activations will **decrease exponentially**.

Weight initialization for deep networks

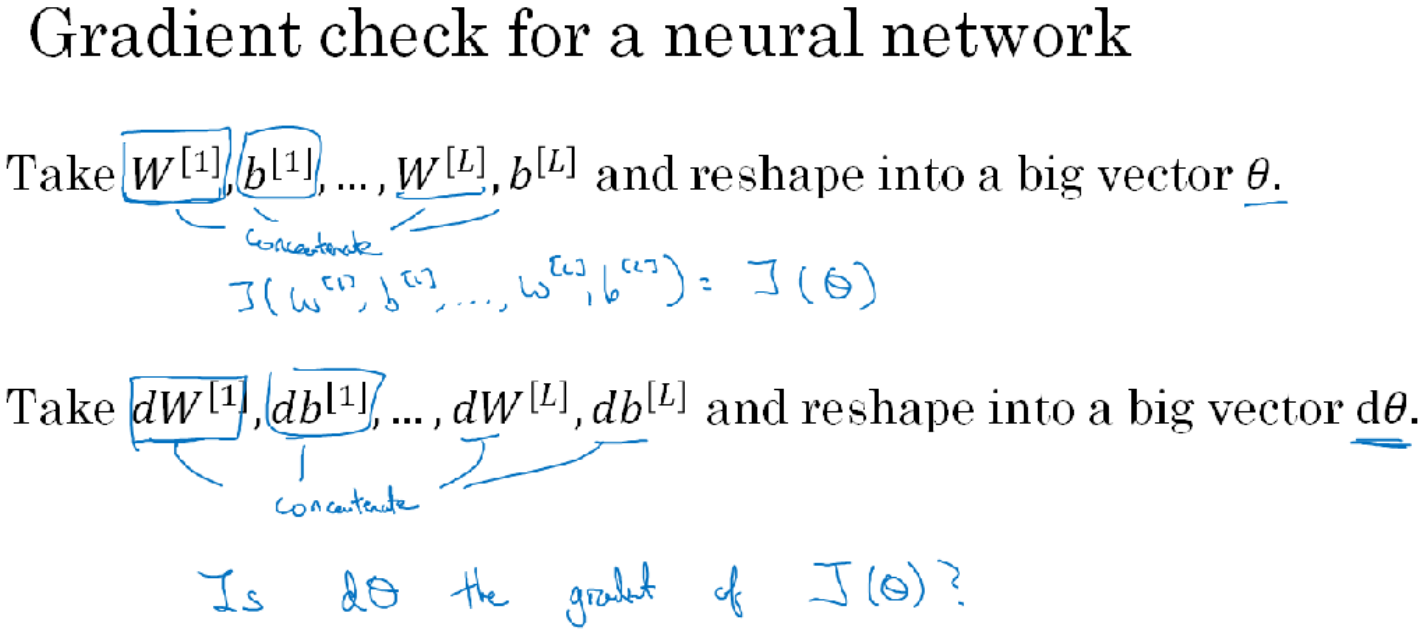
In the last section, we saw how very deep neural networks can have the problems of vanishing and exploding gradients. It turns out that a partial solution to this, doesn't solve it entirely but helps a lot, is better or more careful choice of the random initialization for your neural network. To understand this lets start with the example of initializing the ways for a **single neuron** and then we're going to generalize this to a deep network.



Looking into diagram we get some intuition about the problem of vanishing or exploding gradients as well as how choosing a **reasonable scaling for how you initialize the weights**. Hopefully that makes your weights not explode too quickly and not decay to zero too quickly so you can train a reasonably deep network without the weights or the gradients exploding or vanishing too much. When you train deep networks this is another trick that will help you make your neural networks trained much.

Gradient checking

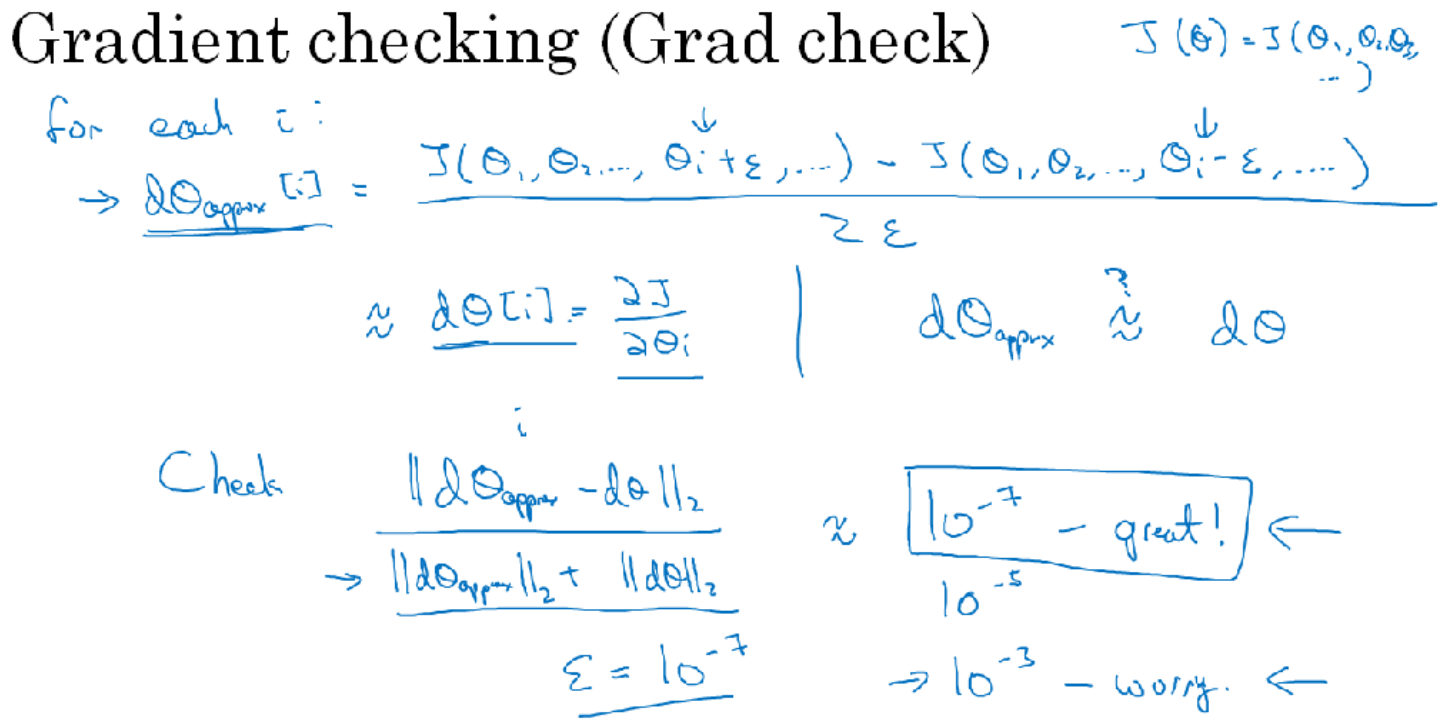
Gradient checking is a technique that can help in saving lots of time to find bugs in implementations of back propagation. In this section we'll see that how could we use it too to debug, or to verify that the implementation of back-prop is correct. Check the diagram below:



So your new network will have some sort of parameters, $W^{[1]}$, $b^{[1]}$ and so on up to $W^{[L]}$,$b^{[1]}$. So to implement gradient checking, the first thing you should do is take all your parameters and reshape them into a giant vector data. So what you should do is take **W which is a matrix, and reshape it into a vector**. You gotta take all of

these W's and reshape them into vectors, and then concatenate all of these things, so that you have a giant **vector theta**. Next, with W and b ordered the same way, you can also take $dW^{[1]}$, $db^{[1]}$ and so on, and initiate them into big, giant vector **d_theta** of the same dimension as **theta**. Remember, $dW^{[1]}$ has the same dimension as $W^{[1]}$ and $db^{[1]}$ has the same dimension as $b^{[1]}$. So the same sort of reshaping and concatenation operation, you can then reshape all of these derivatives into a giant vector **d_theta**. Which has the same dimension as theta.

So to implement grad check, what you're going to do is implements a loop so that for each i, so for each component of theta, let's compute **d_theta_approx**. Check the diagram below:

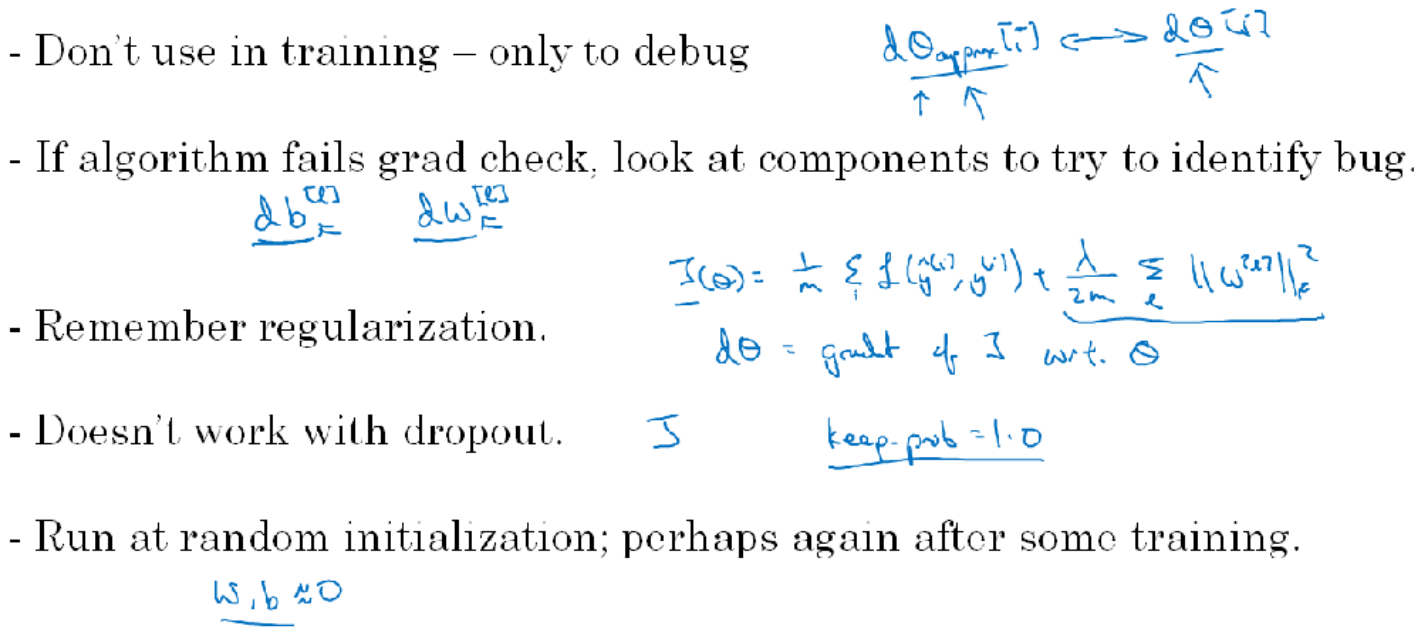


If the grad check has a relatively big value then there are great chances of having a bug, so go in debug, debug, debug and after debugging for a while, If we find that it passes grad check with a small value, then you can be much more confident that NN is correct.

Gradient checking implementation notes

This section we'll learn about some practical tips or notes on how to actually go about implementing grad-check for your neural network.

Gradient checking implementation notes



First, don't use grad check in training, only to debug. So what I mean is that, computing **d_theta_approx** i, for all the values of i, this is a very slow computation. So to implement gradient descent, you'd use backprop to compute d theta and just use backprop to compute the derivative. And it's only when you're debugging that you would compute this to make sure it's close to d theta. But once you've done that, then you would turn off the grad check, and don't run this during every iteration of gradient descent, because that's just much too slow. Second, if an algorithm fails grad check, look at the components, look at the individual components, and try to identify the bug. So what I mean by that is if d theta approx is very far from d theta, what I would do is look at the different values of i to see which are the values of d theta approx that are really very different than the values of d theta. So for example, if you find that the values of theta or **d_theta**, they're very far off, all correspond to dbl for some layer or for some layers, but the components for dw are quite close, right? Remember, different components of theta correspond to different components of b and w. When you find this is the case, then maybe you find that the bug is in how you're computing db, the derivative with respect to parameters b. And similarly, vice versa, if you find that the values that are very far, the values from d theta approx that are very far from d theta, you find all those components came from dw or from dw in a certain layer, then that might help you hone in on the location of the bug. This doesn't always let you identify the bug right away, but sometimes it helps you give you some guesses about where to track down the bug.

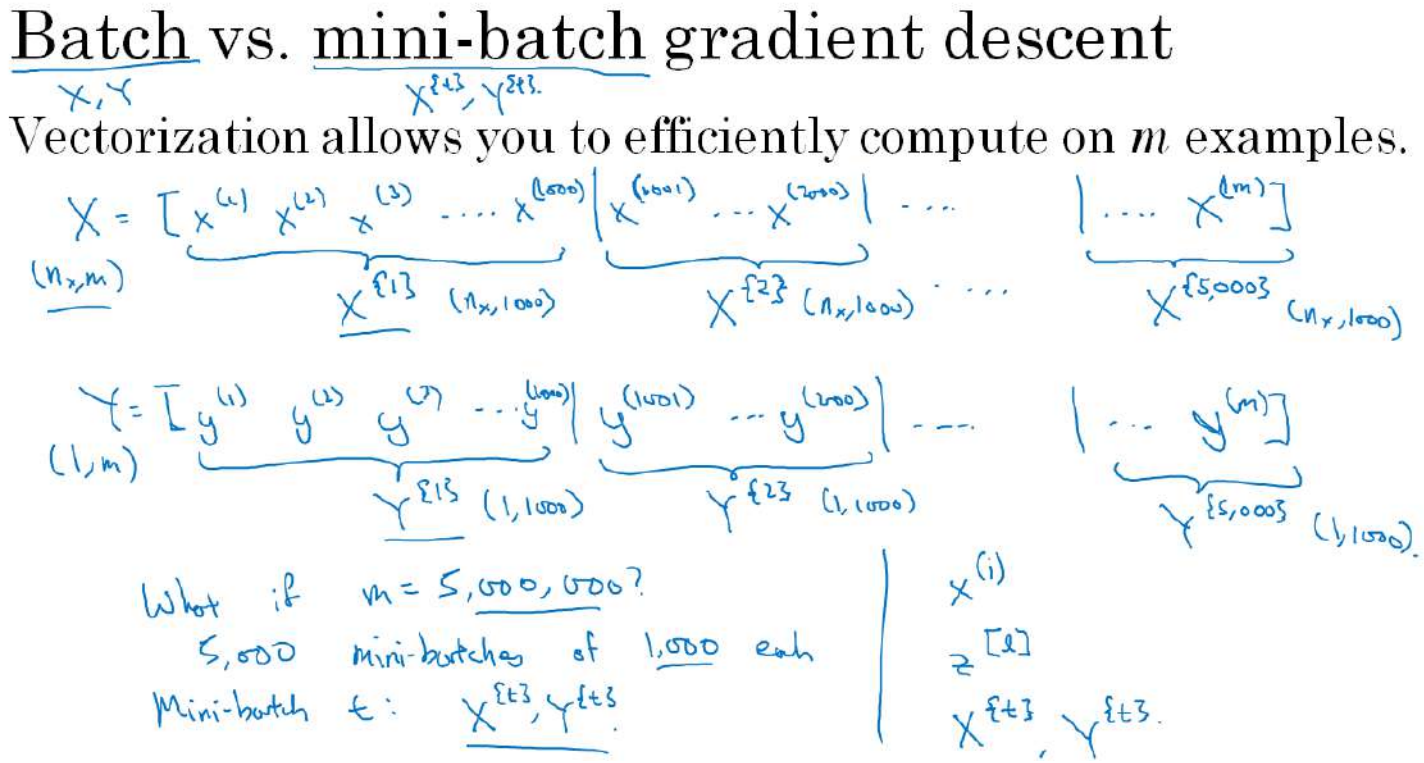
Next, when doing grad check, **remember your regularization** term if you're using regularization. Next, **grad check doesn't work with dropout, because in every iteration, dropout** is randomly eliminating different subsets of the hidden units. There isn't an easy to compute cost function J that dropout is doing gradient descent on. It turns out that dropout can be viewed as optimizing some cost function J, but it's cost function J defined by summing over all exponentially large subsets of nodes they could eliminate in any iteration. So the cost function J is very difficult to compute, and you're just sampling the cost function every time you eliminate different random subsets in those we use dropout. So it's difficult to use **grad check to double check your computation with dropouts**. So what we can usually do is implement grad check without dropout. So if you want, you can set **keep-prob and dropout to be equal to 1.0**. And then turn on dropout and hope that my implementation of dropout was correct. So the recommended setting is to turn off dropout, use grad check to double check that your algorithm is at least correct without dropout, and then turn on dropout.

Week 2: Optimization algorithms

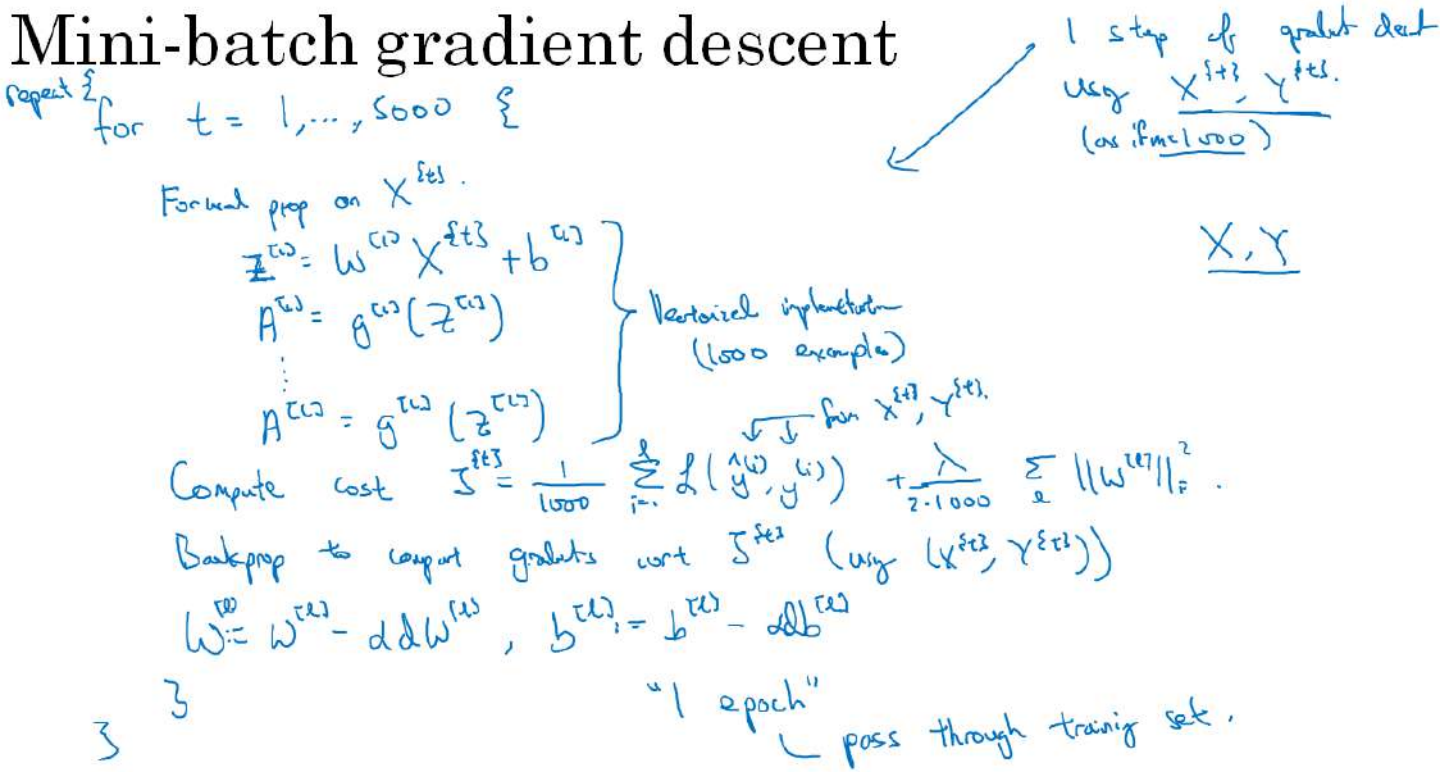
Learning Objectives

- Remember different optimization method such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate the convergence and improve the optimization
- Know the benefit of learning rate decay and apply it on your optimization

Optimization algorithms
Mini-batch gradient descent
In this section we'll learn about optimization algorithms that will enable user to train your neural network much faster. As mentioned before applying machine learning is a highly empirical process, is highly iterative process. In which you just had to train a lot of models to find one that works really well. So, it really helps to really train models quickly. One thing that makes it more difficult is that Deep Learning does not work best in a regime of big data. We are able to train neural networks on a huge data set and training on a large data set is just slow. So, what you find is that having fast optimization algorithms, having good optimization algorithms can really speed up the efficiency of you and your team. So, let's get started by talking about mini-batch gradient descent. We've learned previously that vectorization allows you to efficiently compute on all m examples, that allows you to process your whole training set without an explicit formula. That's why we would take our training examples and stack them into these huge matrix capsul. Check diagram below:

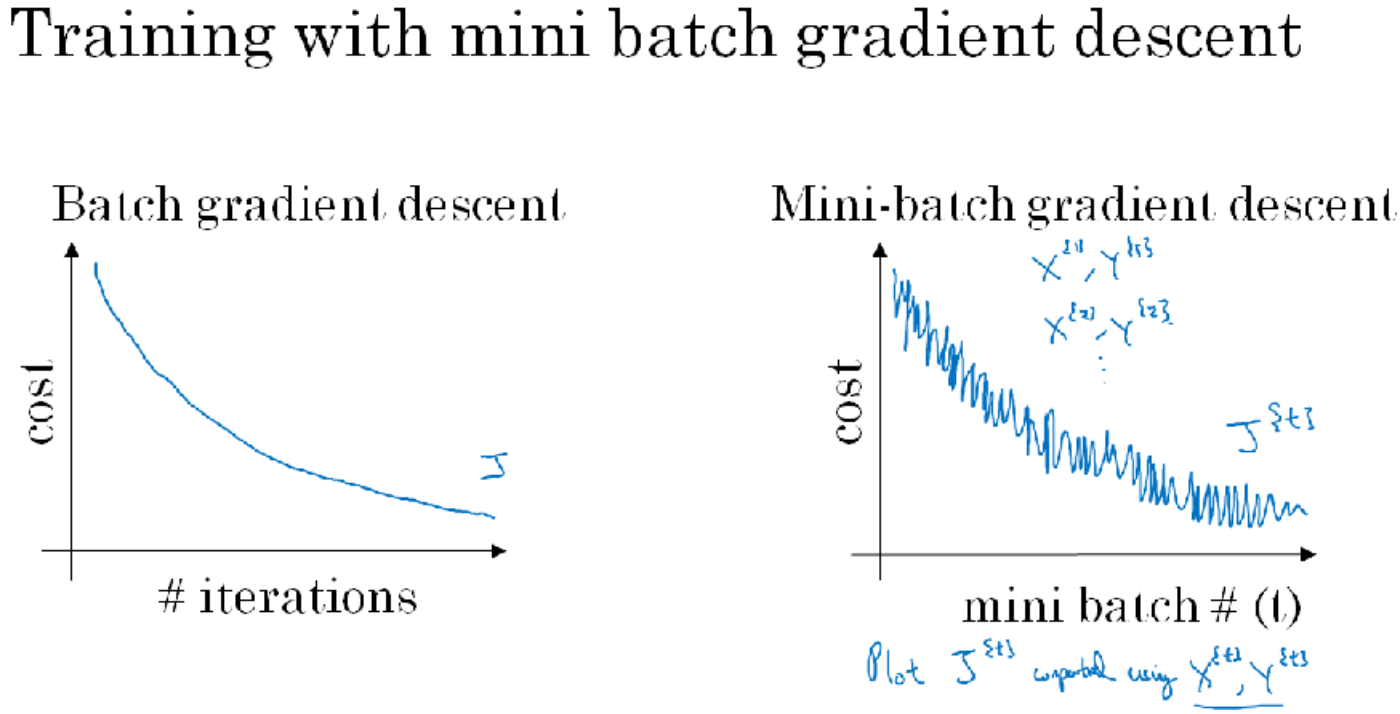


Vectorization allows you to process all M examples relatively quickly if M is very large then it can still be slow. For example what if M was 5 million or 50 million or even bigger. With the implementation of gradient descent on your whole training set, what you have to do is, you have to process your entire training set before you take one little step of gradient descent. And then you have to process your entire training sets of five million training samples again before you take another little step of gradient descent. So, it turns out that you can get a faster algorithm if you let **gradient descent start to make some progress even before you finish processing your entire, your giant training sets of 5 million examples**. In particular, here's what you can do. Let's say that you split up your training set into smaller, little baby training sets and these baby training sets are called **mini-batches**. Check diagram for detail explanation. To explain the name of this algorithm, **batch gradient descent**, refers to the gradient descent algorithm we have been talking about previously. Where you process your entire training set all at the same time. And the name comes from viewing that as processing your entire batch of training samples all at the same time. **Mini-batch gradient** descent in contrast, refers to algorithm which we'll talk about on the next section and which you process is single mini batch with mini-batch gradient descent, a single pass through the training set, that is one epoch, allows you to take 5,000 gradient descent steps. Now of course you want to take multiple passes through the training set which you usually want to, you might want another for loop for another while loop out there. So you keep taking passes through the training set until hopefully you converge with approximately converge. When you have a lost training set, mini-batch gradient descent runs much faster than batch gradient descent and that's pretty much what everyone in Deep Learning will use when you're training on a large data set.
Check this diagram too:



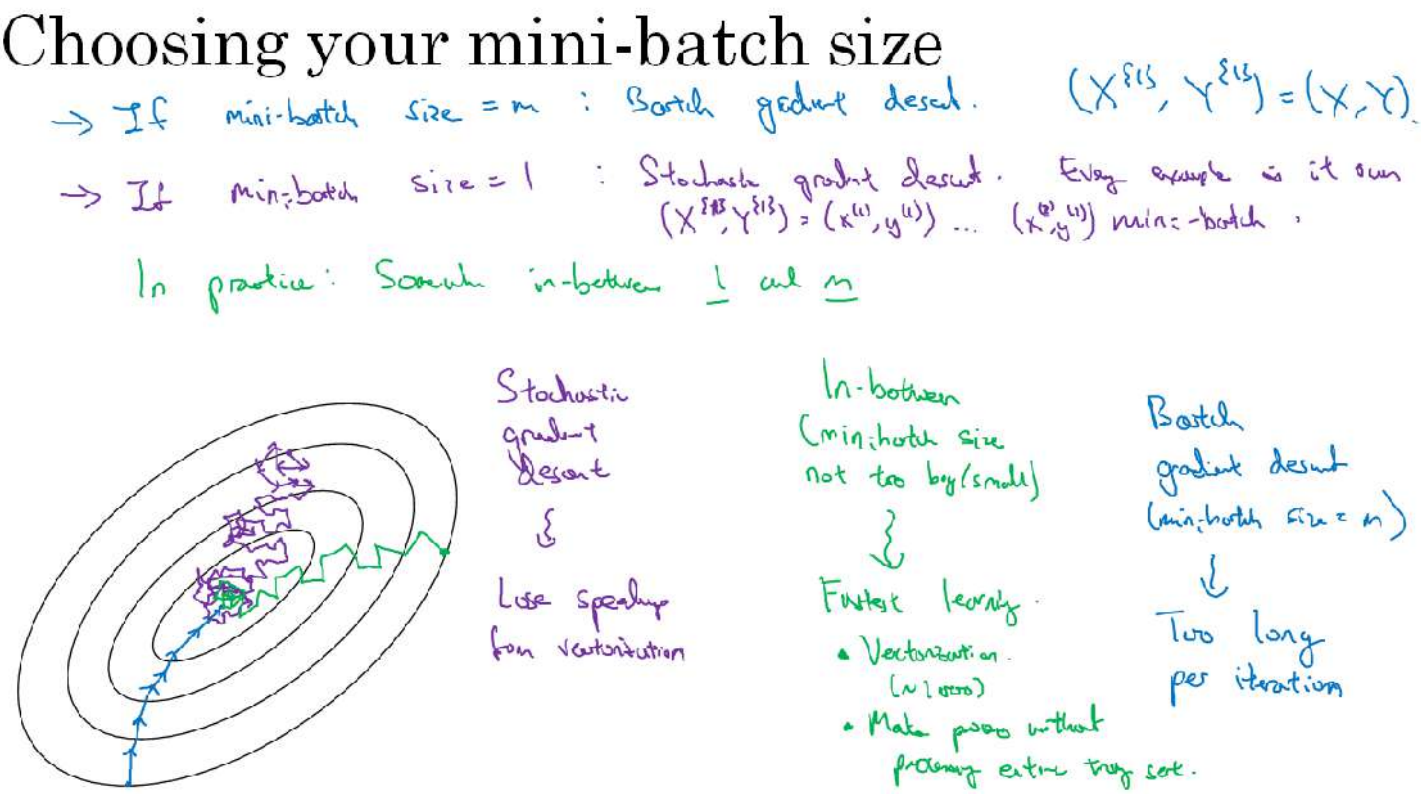
Understanding Mini-batch gradient descent

In this section, you learn more details of how to implement gradient descent and gain a better understanding of what it's doing and why it works. With batch gradient descent on every iteration you go through the entire training set and you'd expect the cost to go down on every single iteration. Check the diagram below:

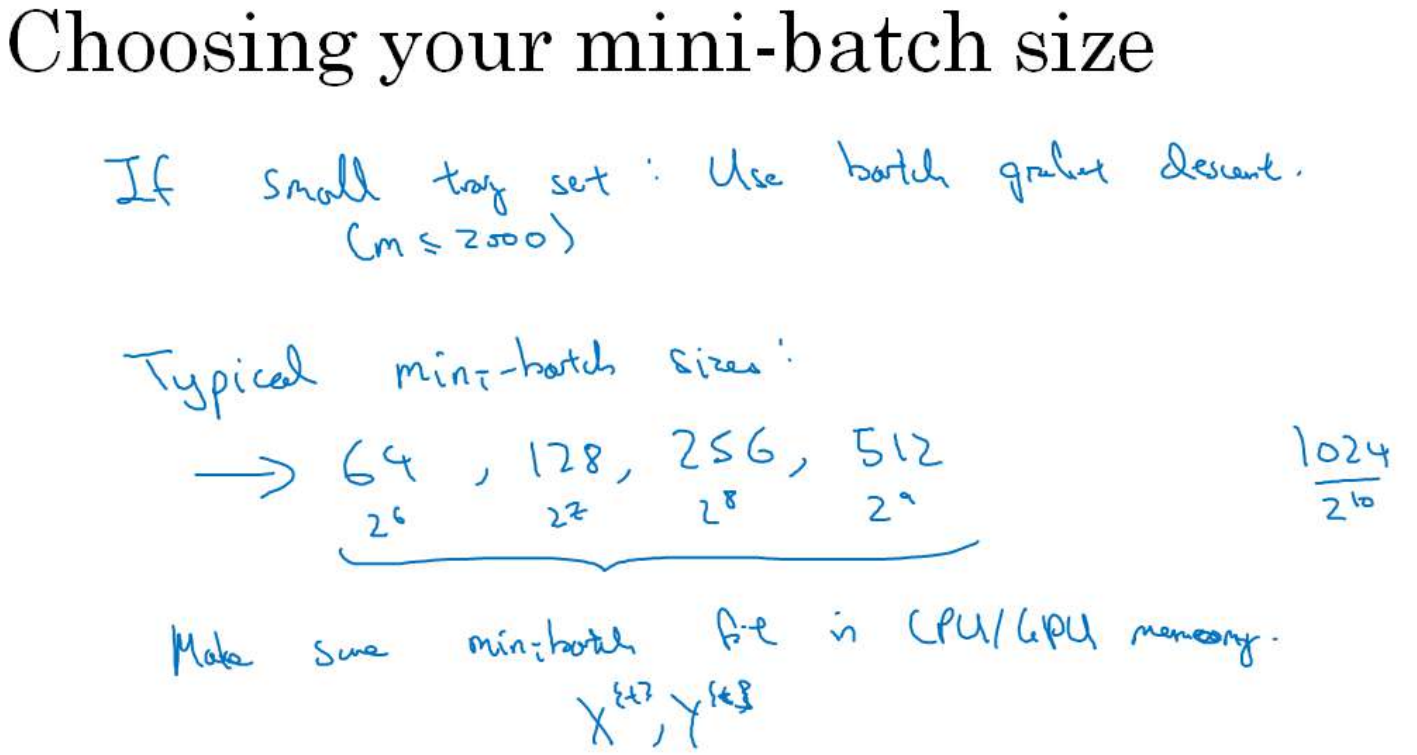


So if we've had the cost function j as a function of different iterations it should decrease on every single iteration. And if it ever goes up even on iteration then something is wrong. Maybe you're running ways to big. On mini batch gradient descent though, if you plot progress on your cost function, then it may not decrease on every iteration. In particular, on every iteration you're processing some $X^{\{t\}}$, $Y^{\{t\}}$ and so if you plot the cost function $J^{\{t\}}$, which is computer using just $X^{\{t\}}$, $Y^{\{t\}}$. Then it's as if on every iteration you're training on a

different training set or really training on a different mini batch. So you plot the cross function J, you're more likely to see something that looks like this. It should trend downwards, but it's also going to be a little bit noisier.



Some guidelines to choose mini-batch algorithm and size. First, if you have a small training set, Just use **batch gradient descent.** If you have a small training set then no point using **mini-batch gradient descent** you can process a whole training set quite fast. So you might as well use batch gradient descent. What a small ~ 2000 training set means. Otherwise, if you have a bigger training set, typical mini batch sizes would be anything from 64 up to maybe 512 are quite typical. And because of the way computer memory is laid out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2.

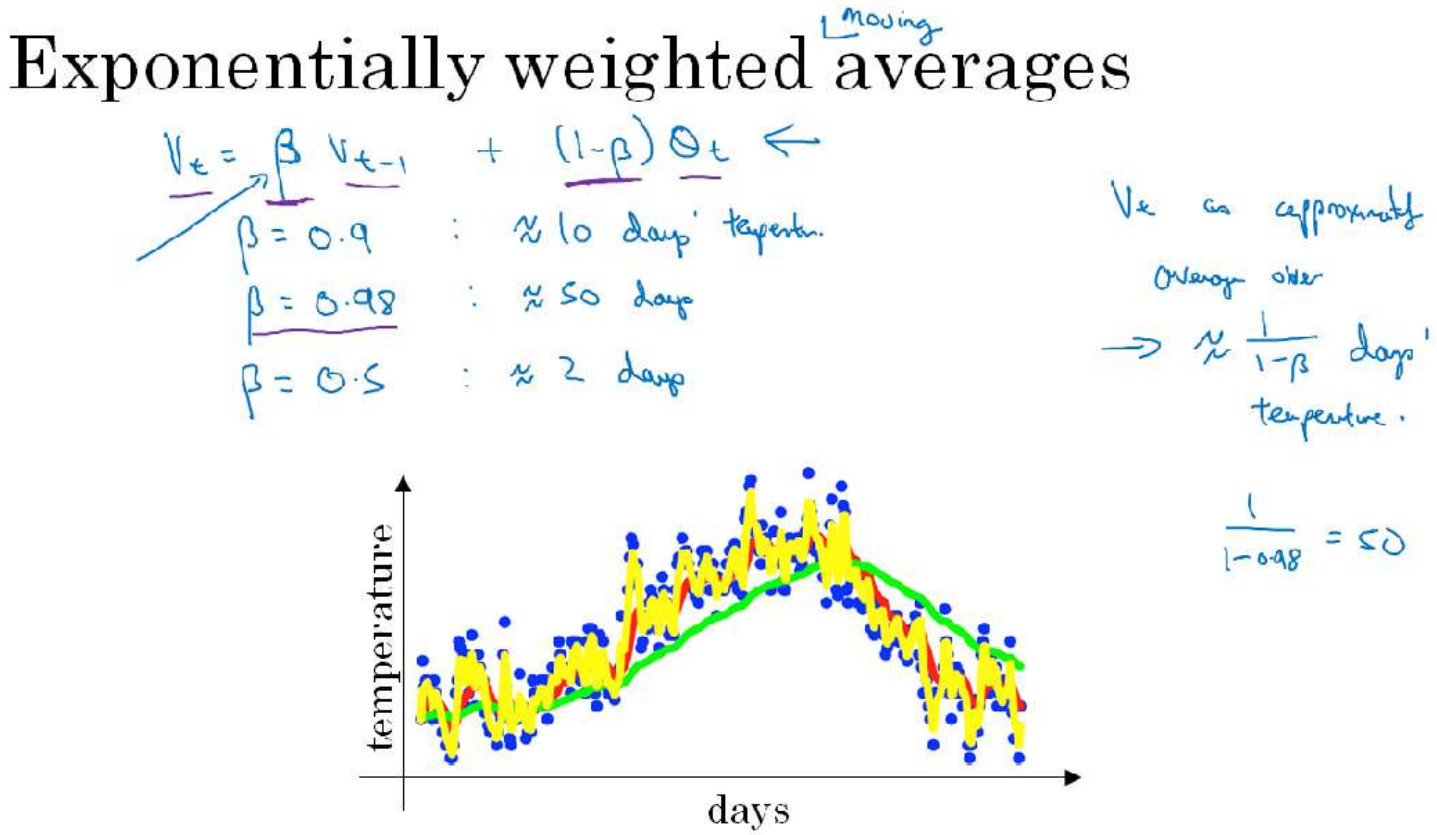


One more point, all of your $X^{\{t\}}$, $Y^{\{t\}}$ that that fits in CPU/GPU memory and this really depends on your application and how large a single training sample is. But if you ever process a mini-batch that doesn't actually

fit in CPU, GPU memory, whether you're using the process, the data. Then you find that the performance suddenly falls of a cliff and is suddenly much worse. So this gives a sense of the typical range of mini batch sizes that people use. In practice of course the mini batch size is another hyper parameter that you might do a quick search over to try to figure out which one is most sufficient of reducing the cost function J.

Exponentially weighted averages

There are some optimization algorithms which are faster than gradient descent. In order to understand those algorithms, you need to be able they use something called **exponentially weighted averages**. Also called exponentially weighted moving averages in statistics.



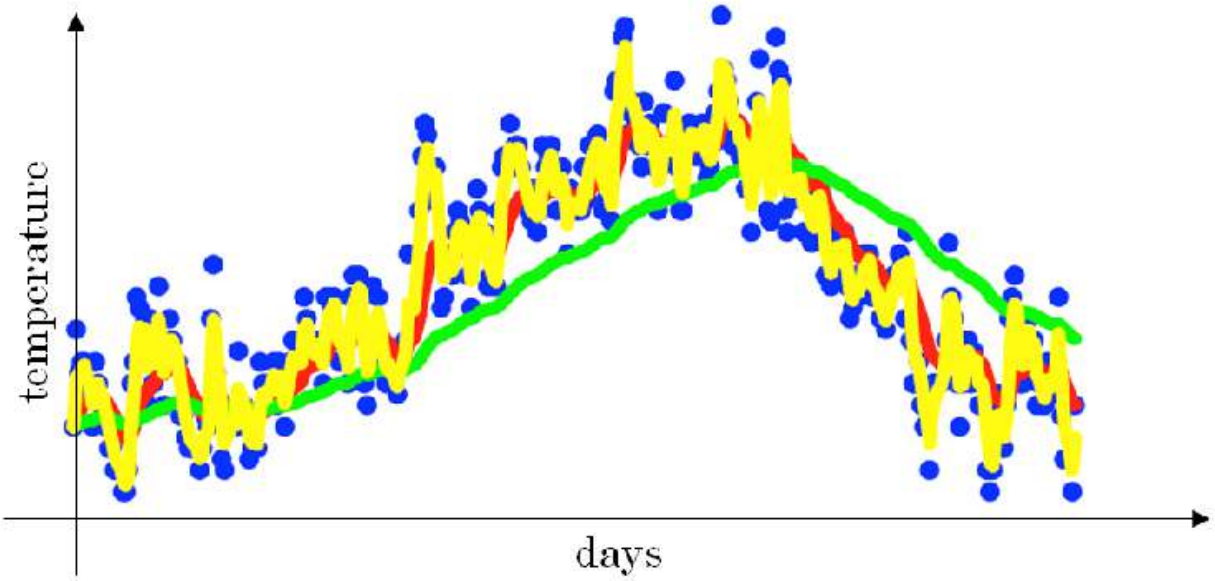
Understanding exponentially weighted averages

In the last section, we talked about **exponentially weighted averages**. This will turn out to be a key component of several **optimization algorithms** that you used to train your neural networks. So, in this section, we'll delve a little bit deeper into intuitions for what this algorithm is really doing.

Exponentially weighted averages

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9 \quad 0.98 \quad 0.5$



Implementing exponentially weighted averages

$v_0 = 0$

$v_1 = \beta v_0 + (1-\beta)\,\theta_1$

$v_2 = \beta v_1 + (1-\beta)\,\theta_2$

$v_3 = \beta v_2 + (1-\beta)\,\theta_3$

...

$V_\theta := 0$

$V_\theta := \beta V + (1-\beta)\theta_1$

$V_\theta := \beta V + (1-\beta)\theta_2$

$\vdots$

$\rightarrow V_\theta = 0$

Repeat {

Get next θ_t

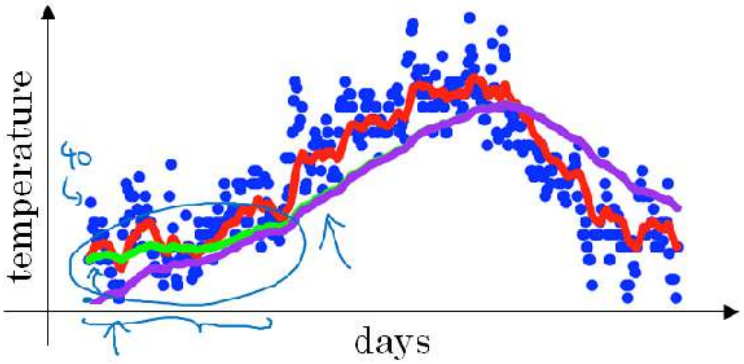
$V_\theta := \beta V_\theta + (1-\beta)\theta_t$

}

Bias correction in exponentially weighted averages

We've learned how to implement exponentially weighted averages. There's one technical detail called **bias correction** that can make you computation of these averages more accurately.

Bias correction



$\beta = 0.98$

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$v_0 = 0$

$v_1 = 0.98 v_0 + 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

$\frac{v_t}{1-\beta^t}$

$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

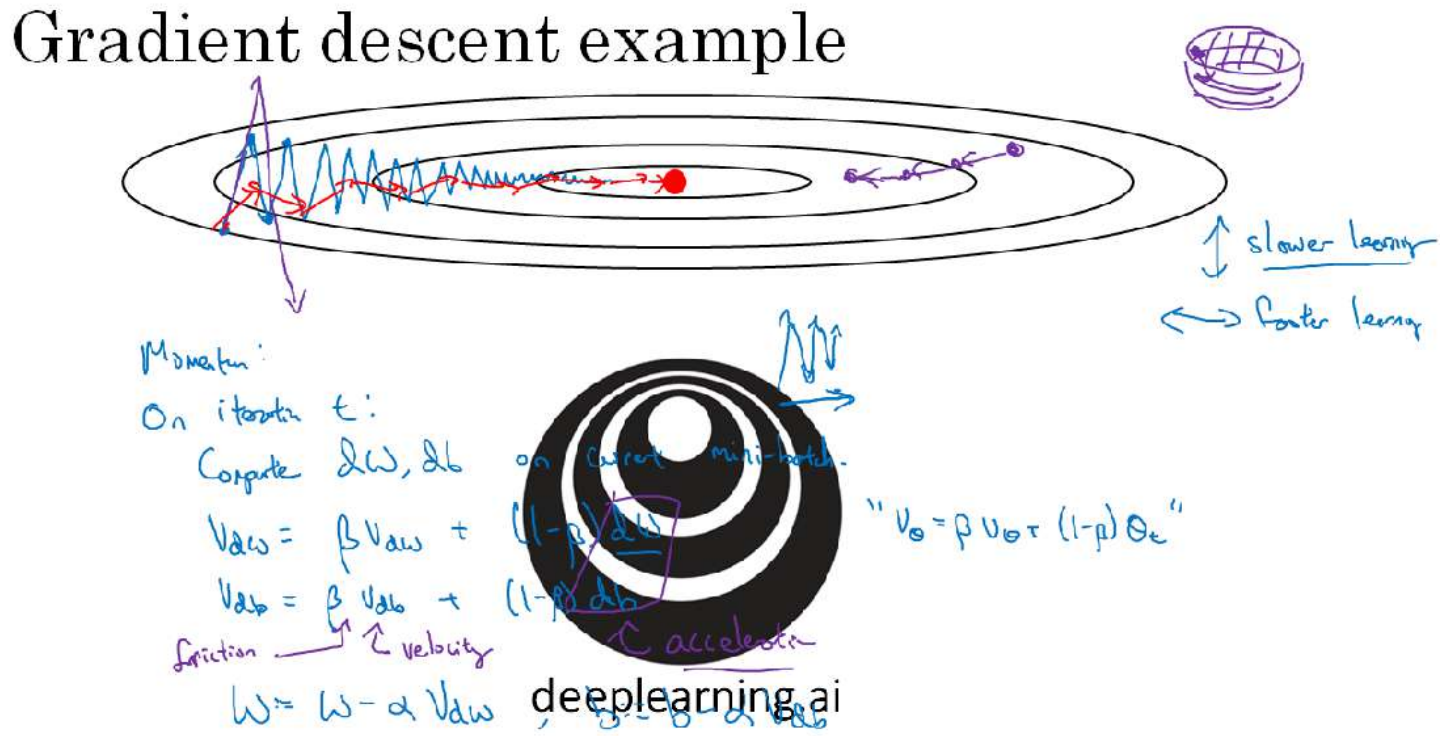
$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

In machine learning, for most implementations of the **exponential weighted average**, people don't often bother to implement **bias corrections** because most people would rather just wait that initial period and have a slightly more biased estimate and go from there but if you are concerned about the bias during this initial phase, while your exponentially weighted moving average is still warming up. Then bias correction can help you get a better estimate early on.

Gradient descent with momentum

There's an algorithm called **momentum**, or **gradient descent with momentum that almost always works faster than the standard gradient descent algorithm**. In one sentence, the basic idea is to compute an **exponentially weighted average of your gradients, and then use that gradient to update your weights instead**. In this section, let's unpack that one sentence description and see how you can actually implement this. As a example let's say that you're trying to optimize a cost function which has contours like this.

So the red dot denotes the position of the minimum.



Looking at diagram we can see that the up and down **oscillations slows down gradient descent** and prevents us from using a much **larger learning rate**. In particular, if you were to use a much larger learning rate you might end up over shooting and end up diverging and so the need to prevent the oscillations from getting too big forces you to use a learning rate that's not itself too large. Another way of viewing this problem is that on the vertical axis you want your learning to be a bit slower, because you don't want those oscillations. But on the horizontal axis, you want **faster learning**. As we want to aggressively move from left to right, toward that minimum, toward that red dot (in diagram).To do that we can implement gradient descent with momentum.

On each iteration, or more specifically, during iteration you would compute the usual derivatives dw, db, and if you're using batch gradient descent, then the current mini-batch would be just your whole batch and this works as well off a batch gradient descent. So if your current mini-batch is your entire training set, this works fine as well.

Implementation details

$V_{dW} = 0$, $V_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta) dW$$
$$v_{db} = \beta v_{db} + (1-\beta) db$$

$V_{dW} = \beta V_{dW} + dW$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β

$\beta = 0.9$

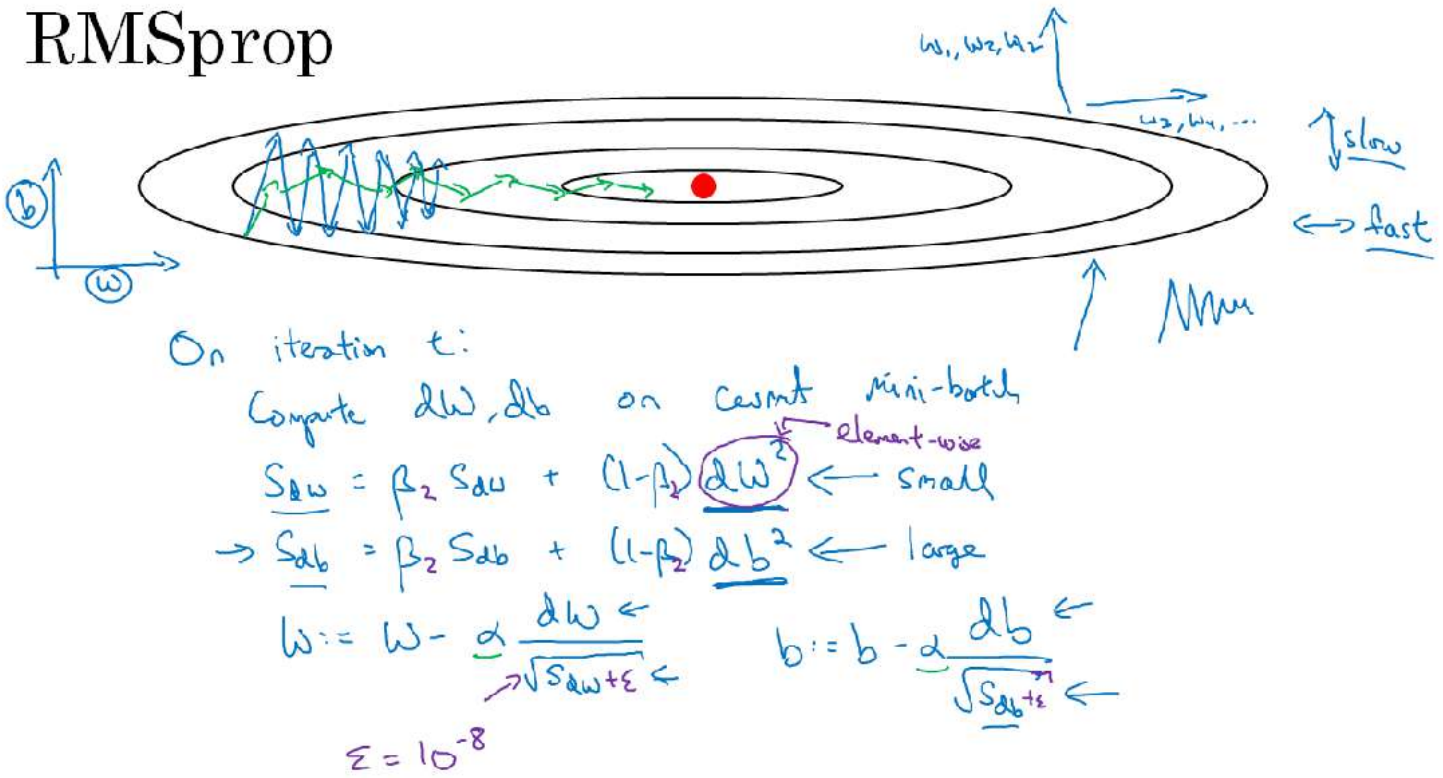
average over last ≈ 10 gradients

(Check the diagram for explanation)
If you average out these gradients, you find that the oscillations in the vertical direction will tend to average out to something closer to zero. So, in the vertical direction, where you want to slow things down, this will average out positive and negative numbers, so the average will be close to zero. Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big. So that's why with this algorithm, with a few iterations you find that the gradient descent with momentum ends up eventually just taking steps that are much smaller oscillations in the vertical direction, but are more directed to just moving quickly in the horizontal direction. And so this allows your algorithm to take a more straightforward path, or to damp out the oscillations in this path to the minimum.

RMSProp

There's another algorithm called **RMSprop**, which stands for **root mean square prop**, that can also speed up gradient descent.
Let's see how it works. Recall our example from before, that if you implement gradient descent, you can end up with huge oscillations in the vertical direction, even while it's trying to make progress in the horizontal direction. In order to provide intuition for this example, let's say that the vertical axis is the parameter b and horizontal axis is the parameter w. It could be w1 and w2 where some of the center parameters was named as b and w for the sake of intuition. And so, you want to slow down the learning in the b direction, or in the vertical direction. And speed up learning, or at least not slow it down in the horizontal direction. So this is what the RMSprop algorithm does to accomplish this. On iteration t, it will compute as usual the derivative dW, db on the current

mini-batch.



Check the diagram for formulas.

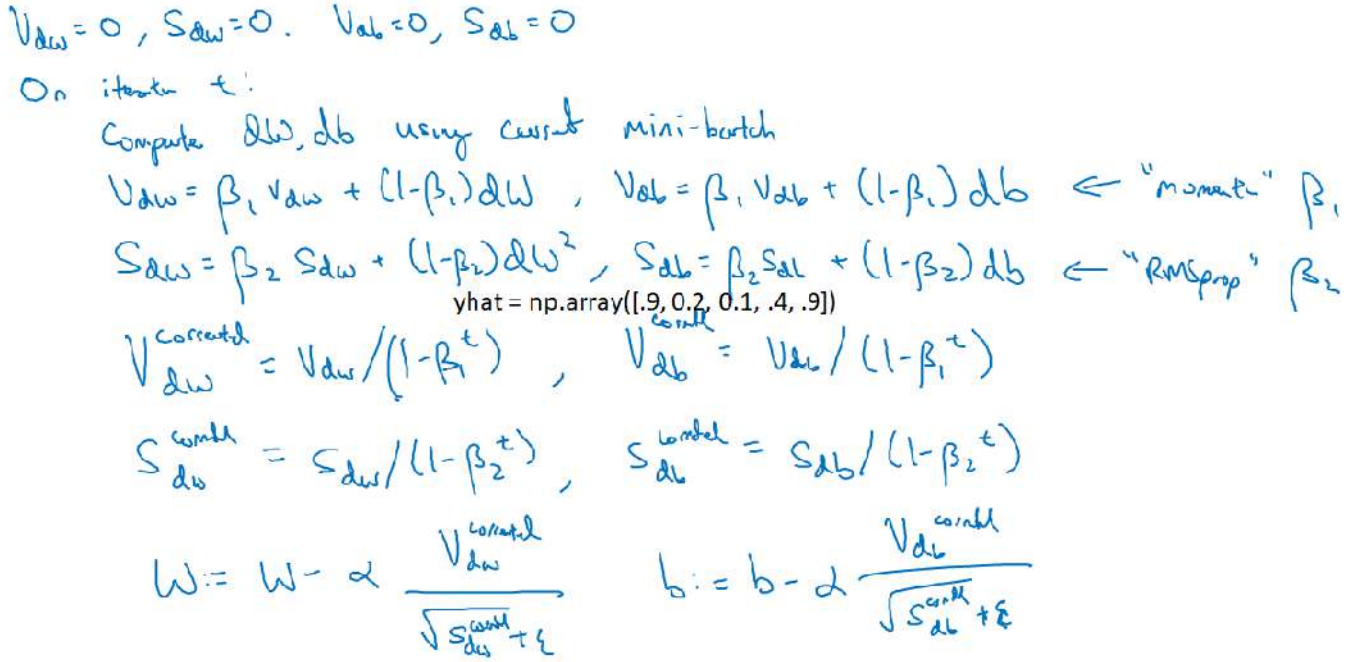
Now let's gain some intuition about how this works. Recall that in the horizontal direction or in this example, in the W direction we want learning to go pretty fast. Whereas in the vertical direction or in this example in the b direction, we want to slow down all the oscillations into the vertical direction. So with this terms $\mathbf{S_{dW}}$ and $\mathbf{S_{db}}$, what we're hoping is that $\mathbf{S_{dW}}$ will be relatively small, so that here we're dividing by relatively small number. Whereas $\mathbf{S_{db}}$ will be relatively large, so that here we're dividing a relatively large number in order to slow down the updates on a vertical dimension and indeed if you look at the derivatives, these derivatives are much larger in the vertical direction than in the horizontal direction. So the slope is very large in the b direction, so with derivatives like this, this is a very large **db** and a relatively small **dw**. Because the function is sloped much more steeply in the vertical direction than as in the b direction, than in the w direction, than in horizontal direction and so, **db squared** will be relatively large. So $\mathbf{S_{db}}$ will relatively large, whereas compared to that **dW** will be smaller, or **dW squared** will be smaller, and so $\mathbf{S_{dw}}$ will be smaller. The one effect of this is also that you can therefore use a **larger learning rate** alpha, and get faster learning without diverging in the vertical direction. So that's RMSprop, and it stands for root mean squared prop, because here you're squaring the derivatives, and then you take the square root.

So finally, RMSprop is a another way for you to speed up your learning algorithm. One fun fact about RMSprop, it was actually first proposed not in an academic research paper, but in a Coursera course that Jeff Hinton had taught on Coursera many years ago.

Adam Optimization Algorithm

During the history of deep learning, many researchers including some very well-known researchers, sometimes proposed optimization algorithms and showed that they worked well in a few problems. But those optimization algorithms subsequently were shown not to really generalize that well to the wide range of neural networks you might want to train. So over time, I think the deep learning community actually developed some amount of skepticism about new optimization algorithms. And a lot of people felt that **gradient descent with momentum** really works well, was difficult to propose things that work much better. So, **rms prop** and the **Adam optimization algorithm**, which we'll talk about in this section, is one of those rare algorithms that has really stood up, and has been shown to work well across a wide range of deep learning architectures So, this is one of the algorithms that I wouldn't hesitate to recommend you try because many people have tried it and seen it work well on many problems and the **Adam optimization algorithm** is basically taking **momentum and rms prop and putting them together**.

Adam optimization algorithm



Looking into equations in diagram we cab see that this algorithm combines the effect of **gradient descent with momentum** together with **gradient descent with rms prop** and this is a commonly used learning algorithm that is proven to be very effective for many different neural networks of a very wide variety of architectures. So, this algorithm has a number of hyper parameters. The learning with hyper parameter alpha is still important and usually needs to be tuned. So you just have to try a range of values and see what works. A common choice really the default choice for ß1 is 0.9, this is the moving average. The hyper parameter for ß2, the authors of the Adam paper, inventors of the Adam algorithm recommend 0.999. Again this is computing the moving weighted average of dw2 as well as db squares. And then Epsilon, the choice of epsilon doesn't matter very much. But the authors of the Adam paper recommended it 10 to the -8. But this parameter you really don't need to set it and it doesn't affect performance much at all. But when implementing Adam, what people usually do is just use the default value. So, ß1 and ß2 as well as epsilon. I don't think anyone ever really tunes Epsilon. And then, try a range of values of Alpha to see what works best. You could also tune ß1 and ß2 but it's not done that often among the practitioners I know.

Hyperparameters choice:

→ α : needs to be tune

→ β_1 : 0.9 → (dw)

→ β_2 : 0.999 → (dw^2)

→ ε : 10^{-8}

Adam : Adaptive moment estimation

Adam Coates

So, where does the term 'Adam' come from? Adam stands for **Adaptive Moment Estimation**. So ß1 is computing the mean of the derivatives. This is called the **first moment**. And ß2 is used to compute exponentially weighted average of the dw² and that's called the **second moment**. So that gives rise to the name adaptive moment estimation. But everyone just calls it the **Adam optimization algorithm**.

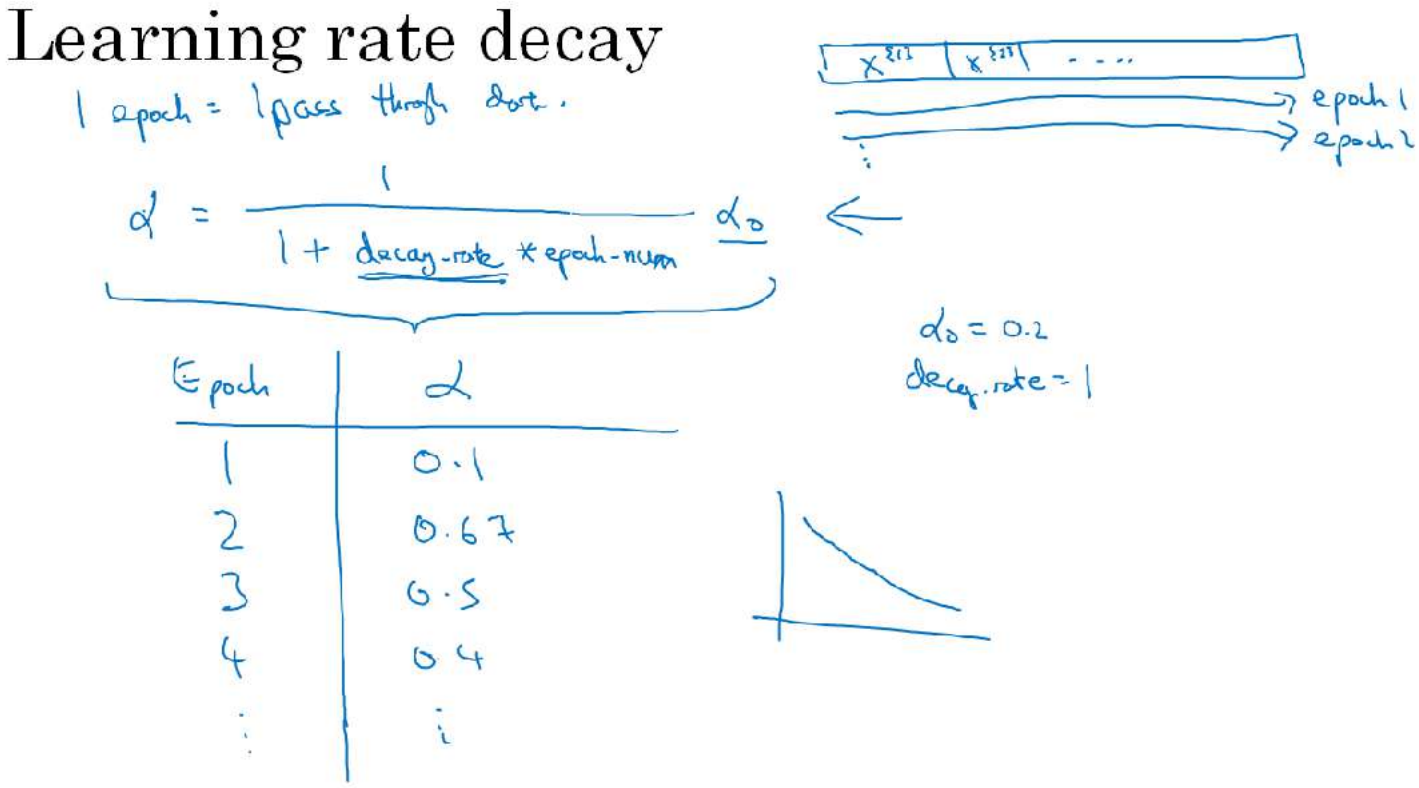
Learning rate decay

One of the things that might help speed up your learning algorithm, is to slowly reduce your learning rate over time. We call this **learning rate decay**. Let's see how you can implement this. Let's start with an example of why you might want to implement learning rate decay. Suppose you're implementing mini-batch gradient descent, with a reasonably small mini-batch. Maybe a mini-batch has just 64, 128 examples. Then as you iterate, your steps will be a little bit noisy. And it will tend towards this minimum (see the diagram)

Learning rate decay

Slowly reduce α

but it won't exactly converge but our algorithm might just end up wandering around, and never really converge, because we're using some fixed value for alpha and there's just some noise in your different mini-batches. But if you were to slowly reduce your learning rate alpha, then during the initial phases, while your learning rate alpha is still large, you can still have relatively fast learning but then as alpha gets smaller, your steps you take will be slower and smaller and so you end up oscillating in a tighter region (check diagram) rather than wandering far away, even as training goes on and on. So the intuition behind slowly reducing alpha, is that maybe during the initial steps of learning, you could afford to take much bigger steps but then as learning approaches converges, then having a slower learning rate allows you to take smaller steps. Check the diagram for steps to implement learning rate decay.

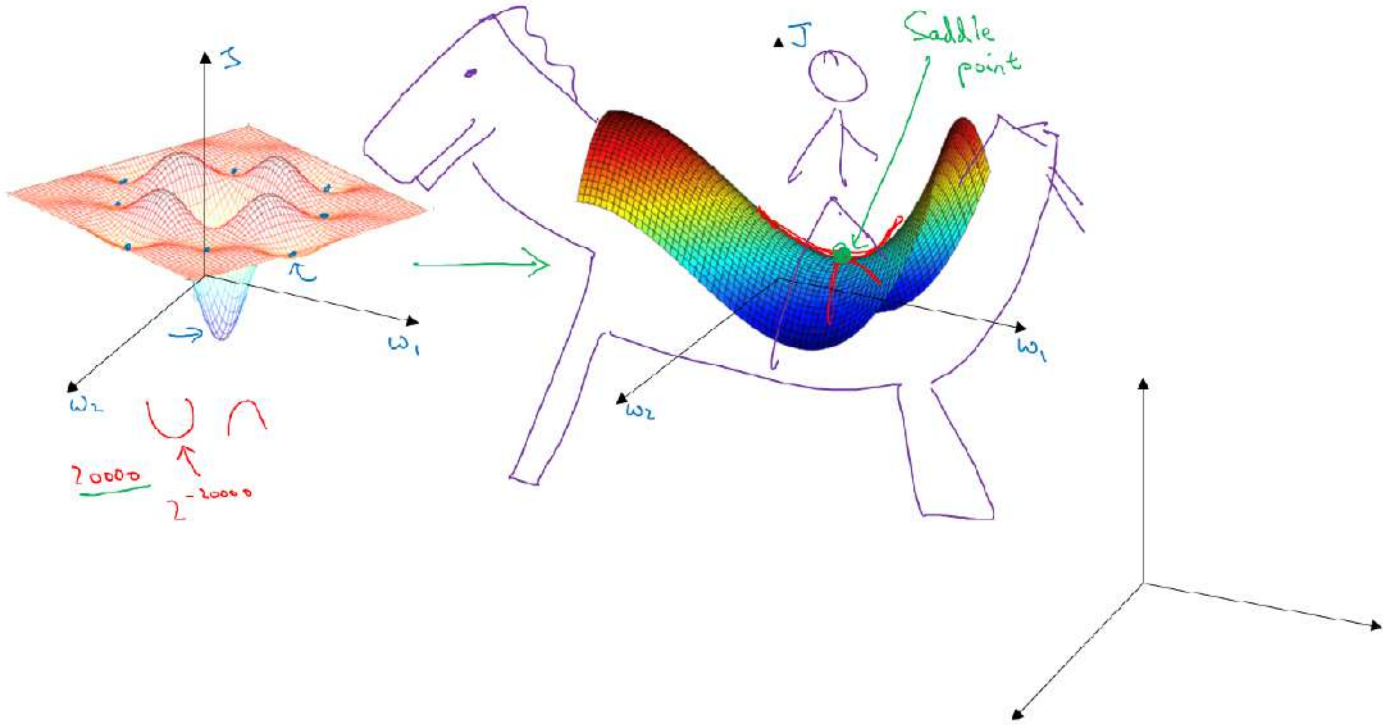


So so far, we've talked about using some formula to govern how alpha, the learning rate, changes over time. One other thing that people sometimes do, is **manual decay** and so if you're training just one model at a time, and if your model takes many hours, or even many days to train. What some people will do, is just watch your model as it's training over a large number of days and then manually say, it looks like the learning rate slowed down, I'm going to decrease alpha a little bit. Of course this works, this manually controlling alpha, really tuning alpha by hand, hour by hour, or day by day. This works only if you're training only a small number of models, but sometimes people do that as well. So now you have a few more options for how to control the learning rate alpha. Finally, **learning rate decay** is usually lower down on the list of things I try. **Setting alpha**, just a fixed value of alpha, and getting that to be well tuned, has a huge impact.

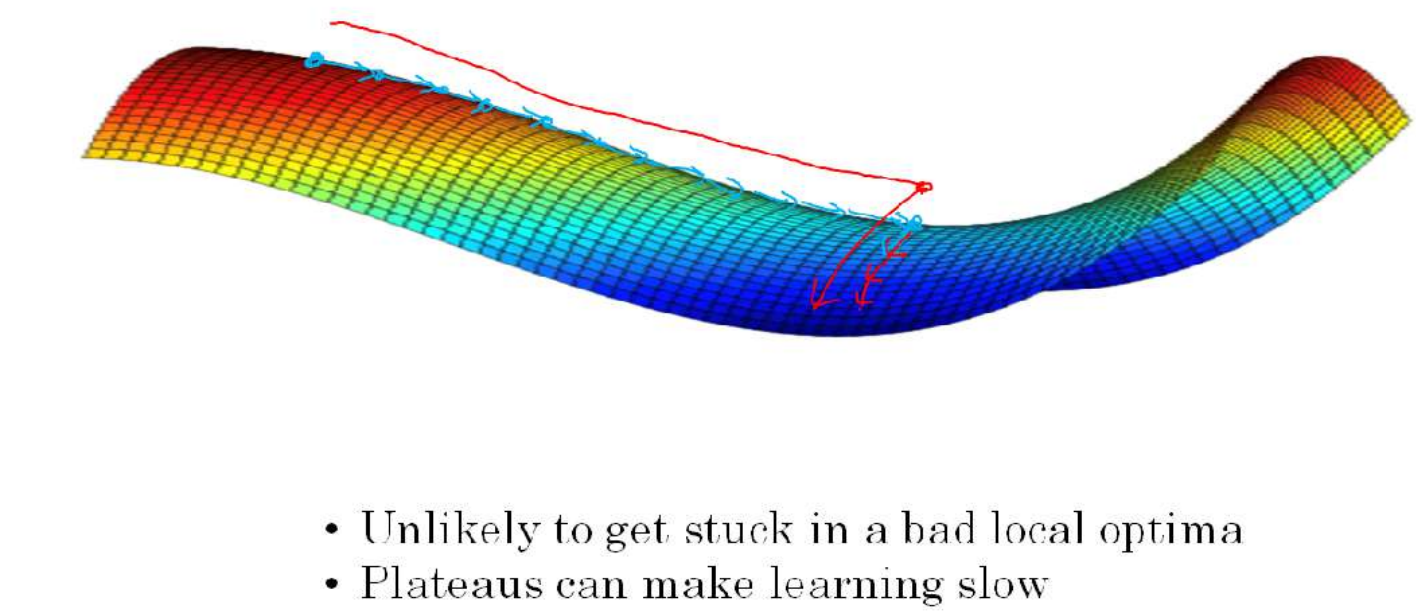
The problem of local optima

In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima but as this theory of deep learning has advanced, our understanding of **local optima** is also changing. Let me show you how we now think about local optima and problems in the optimization problem in deep learning. This was a picture people used to have in mind when they worried about local optima.

Local optima in neural networks



Maybe you are trying to optimize some set of parameters, we call them W1 and W2, and the height in the surface is the cost function. In this picture, it looks like there are a lot of local optima in all those places. And it'd be easy for grading the sense, or one of the other algorithms to get stuck in a local optimum rather than find its way to a global optimum. It turns out that if you are plotting a figure like this in two dimensions, then it's easy to create plots like this with a lot of different local optima. And these very low dimensional plots used to guide their intuition. But this intuition isn't actually correct. It turns out if you create a neural network, most points of zero gradients are not local optima like points like this. Instead most points of **zero gradient in a cost function are saddle points**. So, that's a point where the zero gradient, again, just is maybe W1, W2, and the height is the value of the cost function J. But informally, a function of very high dimensional space, if the gradient is zero, then in each direction it can either be a **convex like** function or a **concave like** function. And if you are in, say, a 20,000 dimensional space, then for it to be a local optima, all 20,000 directions need to look like this. And so the chance of that happening is maybe very small, maybe two to the minus 20,000. Instead you're much more likely to get some directions where the curve bends up like so, as well as some directions where the curve function is bending down rather than have them all bend upwards. **So that's why in very high-dimensional spaces you're actually much more likely to run into a saddle point like that shown on the right, then the local optimum**. As for why the surface is called a saddle point, if you can picture, maybe this is a sort of saddle you put on a horse, right? Maybe this is a horse. This is a head of a horse, this is the eye of a horse. Well, not a good drawing of a horse but you get the idea. Then you, the rider, will sit here in the saddle. That's why this point here, where the derivative is zero, that point is called a saddle point. There's really the point on this saddle where you would sit, I guess, and that happens to have derivative zero and so, one of the lessons we learned in history of deep learning is that a lot of our intuitions about low-dimensional spaces, like what you can plot on the left, they really don't transfer to the very high-dimensional spaces that any other algorithms are operating over. Because if you have 20,000 parameters, then J as your function over 20,000 dimensional vector, then you're much more likely to see saddle points than local optimum. If local optima aren't a problem, then what is a problem?



It turns out that plateaus can really slow down learning and a plateau is a region where the derivative is close to zero for a long time. So if you're here, then gradient descents will move down the surface, and because the gradient is zero or near zero, the surface is quite flat. You can actually take a very long time, you know, to slowly find your way to maybe this point on the plateau. Let it take this very long slope off before it's found its way here and they could get off this plateau. So the takeaways from this section are, first, **you're actually pretty unlikely to get stuck in bad local optima so long as you're training a reasonably large neural network, save a lot of parameters, and the cost function J is defined over a relatively high dimensional space**. But second, **that plateaus are a problem and you can actually make learning pretty slow. And this is where algorithms like momentum or RmsProp or Adam can really help your learning algorithm as well**. And these are scenarios where more sophisticated optimization algorithms, such as Adam, can actually speed up the rate at which you could move down the plateau and then get off the plateau.

Week 3: Hyperparameter tuning

Learning Objectives

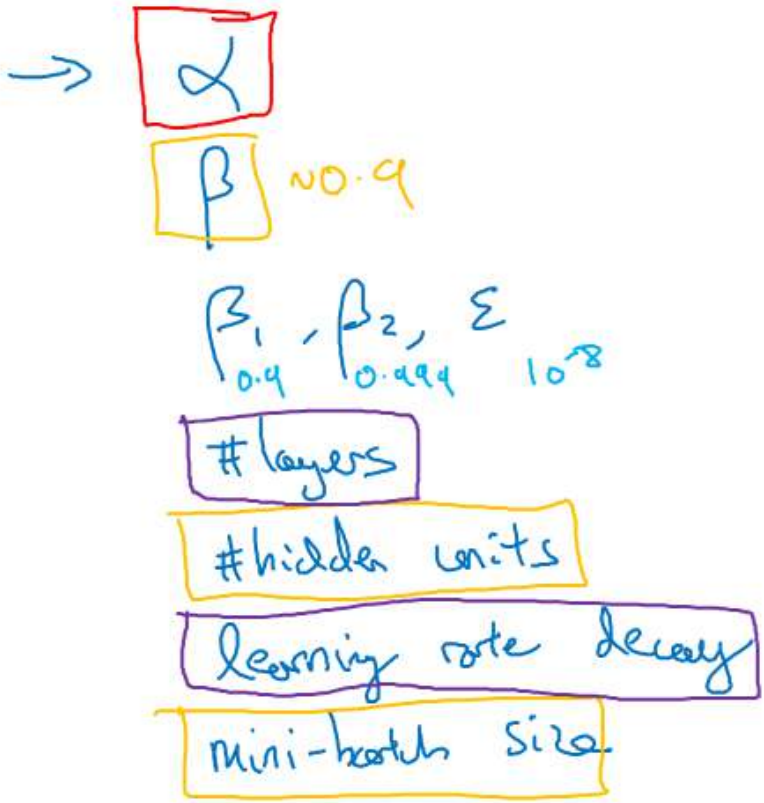
Master the process of hyperparameter tuning

Hyperparameter tuning

Tuning process

We've seen by now that changing neural nets can involve setting a lot of different hyperparameters. Now, how do you go about finding a good setting for these hyperparameters? In this section, we'll discuss some guidelines, some tips for how to systematically organize your hyperparameter tuning process, which hopefully will make it more efficient for you to converge on a good setting of the hyperparameters. One of the painful things about training deep nets is the sheer number of hyperparameters you have to deal with, ranging from the **learning rate alpha** to the **momentum term beta** (if using momentum), or the hyperparameters for the **Adam Optimization Algorithm** which are beta 1, beta 2, and epsilon. Maybe you have to pick the **number of layers**, maybe you have to pick the **number of hidden units** for the different layers, and maybe you want to use **learning rate decay**, so you don't just use a single learning rate alpha. And then of course, you might need to choose the **mini-batch size**. So it turns out, some of these hyperparameters are more important than others. The most learning applications I would say, **alpha, the learning rate is the most important hyperparameter to tune**. Other than alpha, a few other hyperparameters which we tune next, would be the **momentum term**, say, 0.9 is a good default. I'd also tune the **mini-batch size** to make sure that the optimization algorithm is running efficiently. Often we can fiddle around with the **hidden units**. Check the diagram below:

Hyperparameters



Of the ones we've circled in orange, these are really the three that we would consider second in importance to the learning rate alpha and then third in importance after fiddling around with the others, the number of layers can sometimes make a huge difference, and so can learning rate decay. So hopefully it does give us some rough sense of what hyperparameters might be more important than others, **alpha**, most important, for sure, followed maybe by the ones we've circle in **orange**, followed maybe by the ones we circled in **purple**.

Now, if you're trying to tune some set of hyperparameters, how do you select a set of values to explore? The two key takeaways are, use random sampling and adequate search and optionally consider implementing a coarse to fine search process.

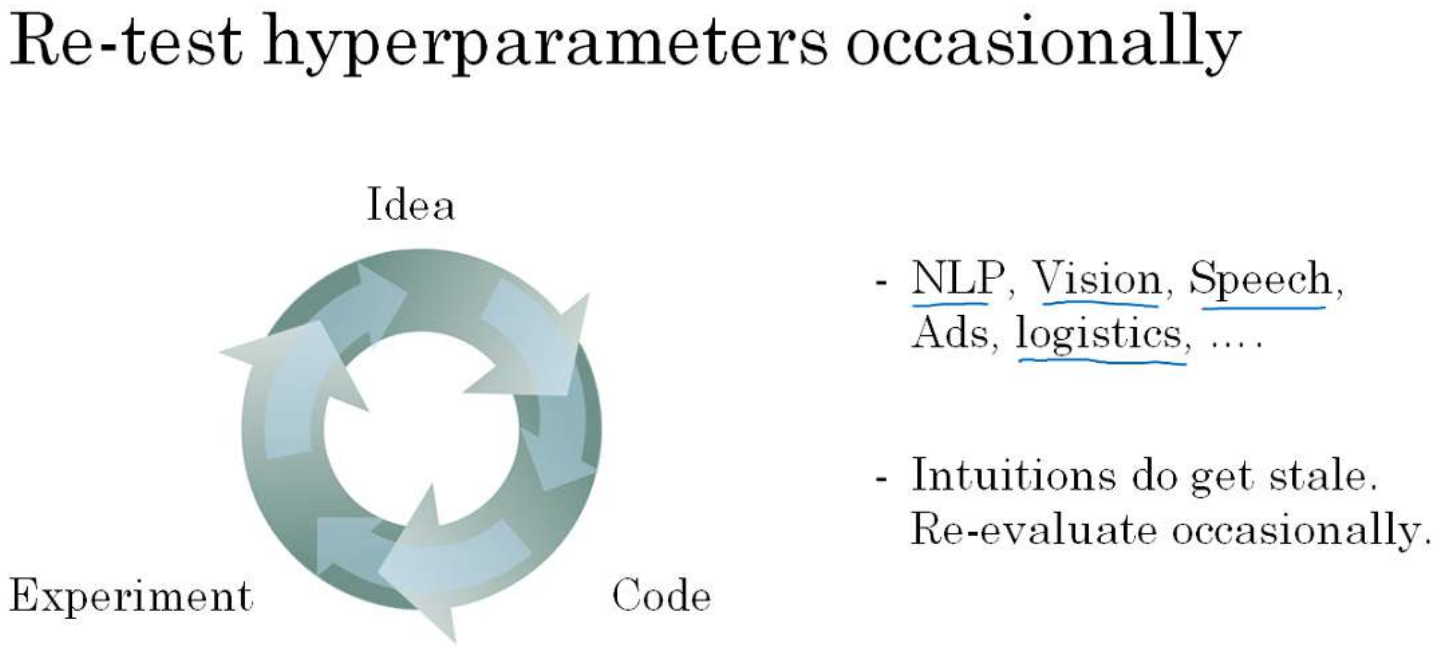
Using an appropriate scale to pick hyperparameters

In the last section, we saw how sampling at random, over the range of hyperparameters, can allow us to search over the space of hyperparameters more efficiently. But it turns out that sampling at random doesn't mean sampling uniformly at random, over the range of valid values. Instead, it's important to pick the appropriate scale on which to explore the hyperparamaters. Sampling uniformly at random over the range is not possible for all hyperparameters for example **learning rate alpha** where instead of linear scale, log scale is better choice. Finally, one other tricky case is sampling the hyperparameter beta, used for computing exponentially weighted averages.

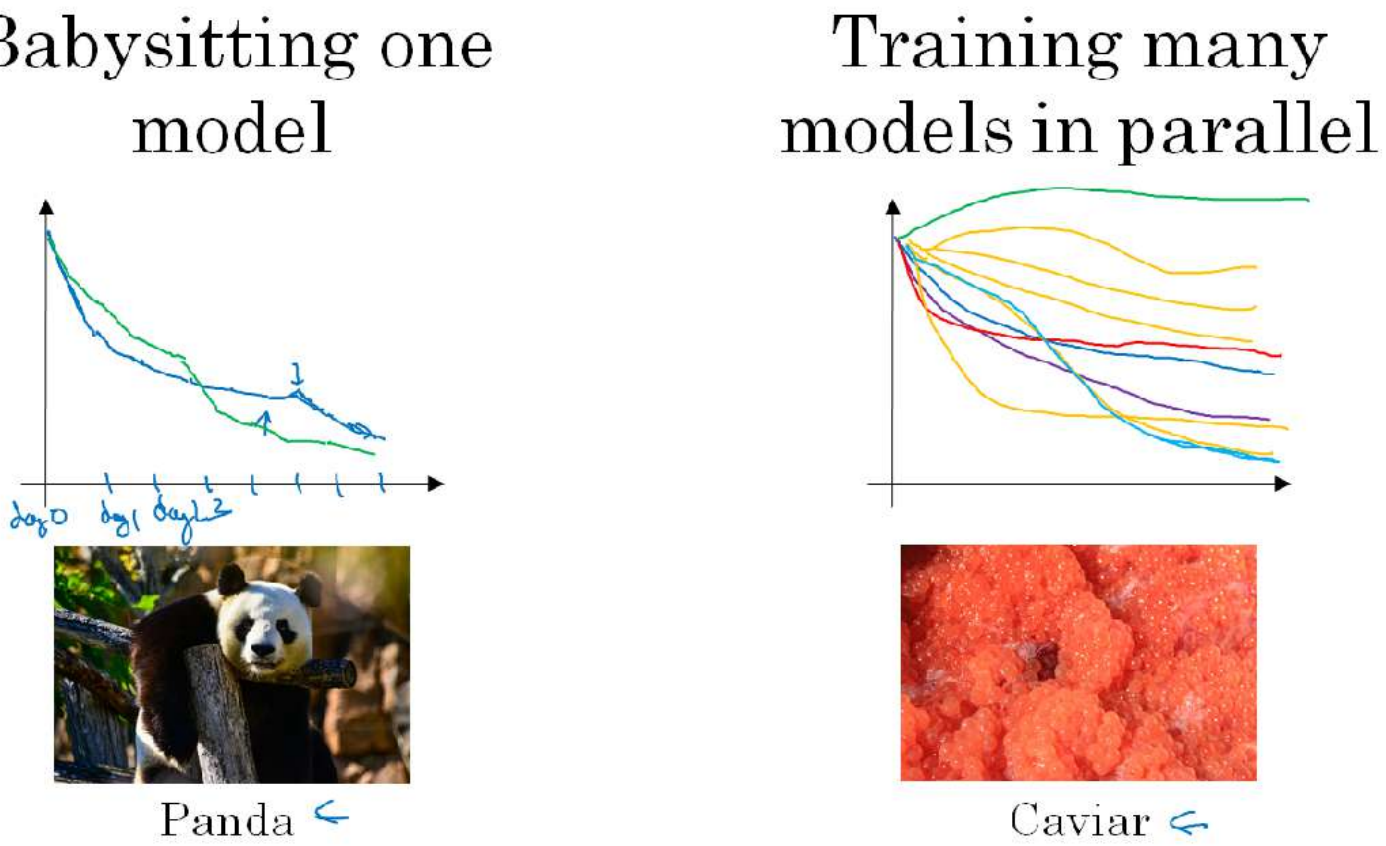
Hyperparameters tuning in practice: Pandas Vs. Caviar

Before wrapping up our discussion on hyperparameter search, here are just a couple of final tips and tricks for how to organize your hyperparameter search process. Deep learning today is applied to many different application areas and that intuitions about hyperparameter settings from one application area may or may not transfer to a different one. There is a lot of cross-fertilization among different applications' domains, so for example, we've seen ideas developed in the computer vision community, such as **Conv nets** or **ResNets**, (which we'll talk about later course), successfully applied to speech. We've seen ideas that were first developed in speech successfully applied in NLP, and so on. So one nice development in deep learning is that people from different application domains do read increasingly research papers from other application domains to look for

inspiration for cross-fertilization. In terms of your settings for the hyperparameters, though, we've seen that intuitions do get stale. So even if you work on just one problem, say logistics, you might have found a good setting for the hyperparameters and kept on developing your algorithm, or maybe seen your data gradually change over the course of several months, or maybe just upgraded servers in your data center. And because of those changes, the best setting of your hyperparameters can get stale. So recommendation is to just retesting or reevaluating your hyperparameters at least once every several months to make sure that you're still happy with the values you have.



Finally, in terms of how people go about searching for hyperparameters, I see maybe two major schools of thought, or maybe two major different ways in which people go about it. One way is if you **babysit one model**. And usually you do this if you have maybe a huge data set but **not a lot of computational resources**, not a lot of CPUs and GPUs, so you can basically afford to train only one model or a very small number of models at a time. In that case you might gradually babysit that model even as it's training. So, for example, on Day 0 you might initialize your parameter as random and then start training. And you gradually watch your learning curve, maybe the cost function J or your dataset error or something else, gradually decrease over the first day. Then at the end of day one you feel confident with the learning progress and try increasing the learning rate a little bit and see how it does. And then maybe it does better. And then that's your Day 2 performance. And after two days you say, okay, it's still doing quite well. Maybe I'll fill the momentum term a bit or decrease the learning variable a bit now, and then you're now into Day 3. And every day you kind of look at it and try nudging up and down your parameters. And maybe on one day you found your learning rate was too big. So you might go back to the previous day's model, and so on. But you're kind of babysitting the model one day at a time even as it's training over a course of many days or over the course of several different weeks. So that's one approach, and people that babysit one model, that is **watching performance and patiently nudging the learning rate up or down**. But that's usually what happens if you don't have enough computational capacity to train a lot of models at the same time.



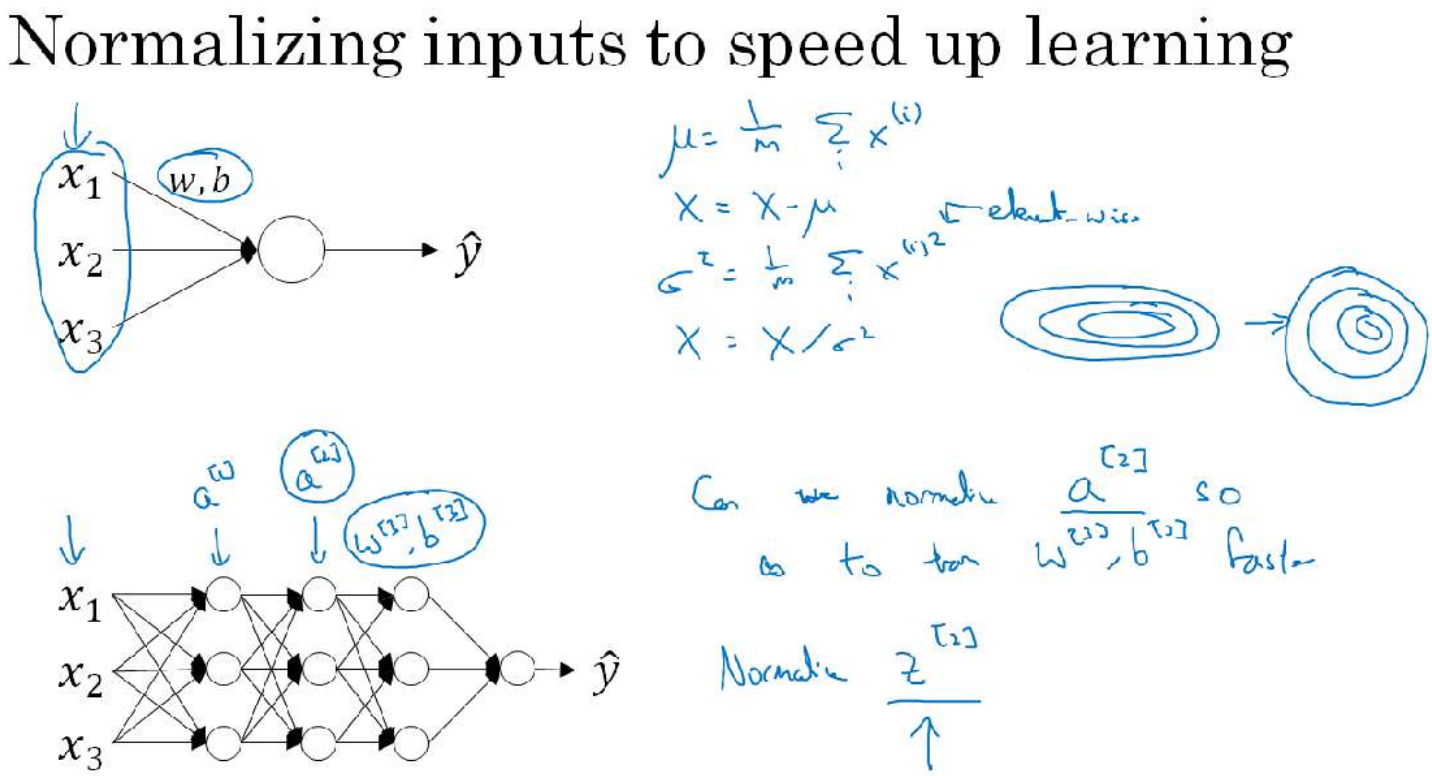
The other approach would be if you train many models in parallel. So you might have some setting of the hyperparameters and just let it run by itself ,either for a day or even for multiple days, and then you get some learning curve like (check right part of the diagram) and this could be a plot of the cost function J or cost of your training error or cost of your dataset error and then at the same time you might start up a different model with a different setting of the hyperparameters. And so, your second model might generate a different learning curve (check diagram). I will say that one looks better. And at the same time, you might train a third model, which might generate a learning curve (Check diagram), and another one that and so on. So this way you can try a lot of different hyperparameter settings and then just maybe quickly at the end pick the one that works best (curve in blue). Looks like in this example it was, maybe BLUE curve that look best. So to make an analogy, I'm going to call the approach on the left the **panda approach**. When pandas have children, they have very few children, usually one child at a time, and then they really put a lot of effort into making sure that the baby panda survives. So that's really babysitting. One model or one baby panda. Whereas the approach on the right is more like what fish do. I'm going to call this the **caviar strategy**. There's some fish that lay over 100 million eggs in one mating season. But the way fish reproduce is they lay a lot of eggs and don't pay too much attention to any one of them but just see that hopefully one of them, or maybe a bunch of them, will do well. But I'm going to call it the panda approach versus the caviar approach, since that's more fun and memorable. So the way to choose between these two approaches is really a function of how much computational resources you have. If you have enough computers to train a lot of models in parallel, then by all means take the caviar approach and try a lot of different hyperparameters and see what works. But in some application domains, I see this in some online advertising settings as well as in some **computer vision** applications, where there's just so much data and the models you want to train are so big that it's difficult to train a lot of models at the same time. It's really application dependent of course, but I've seen those communities use the panda approach a little bit more, where you are kind of babying a single model along and nudging the parameters up and down and trying to make this one model work. Although, of course, even the panda approach, having trained one model and then seen it work or not work, maybe in the second week or the third week, maybe I should initialize a different model and then baby that one along just like even pandas, I guess, can have multiple children in their lifetime, even if they have only one, or a very small number of children, at any one time.

Batch normalization

Normalizing activations in a network

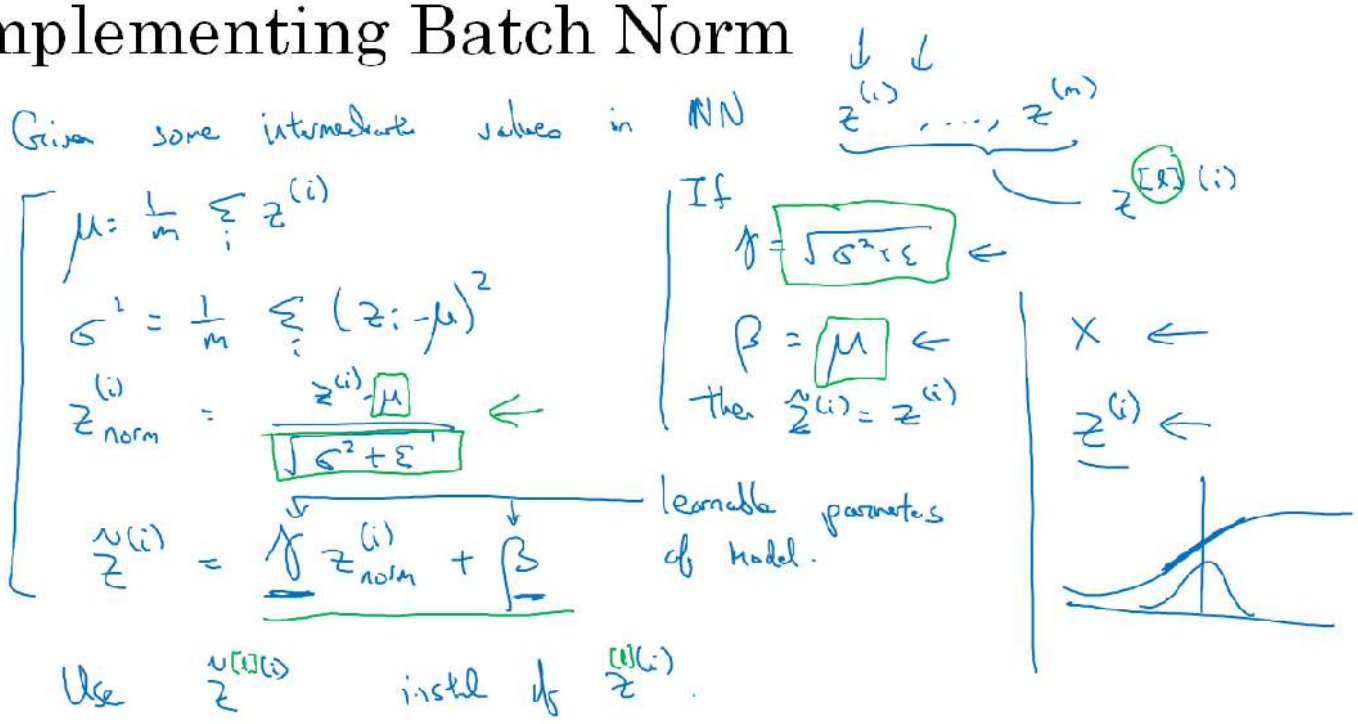
In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization, created by two researchers, Sergey Ioffe and Christian Szegedy. Batch normalization makes your **hyperparameter search problem** much easier, makes your neural network much more robust. The choice of

hyperparameters is a much bigger range of hyperparameters that work well, and will also enable you to much more easily train even very deep networks. Let's see how batch normalization works. When training a model, such as logistic regression, you might remember that normalizing the input features can speed up learnings.



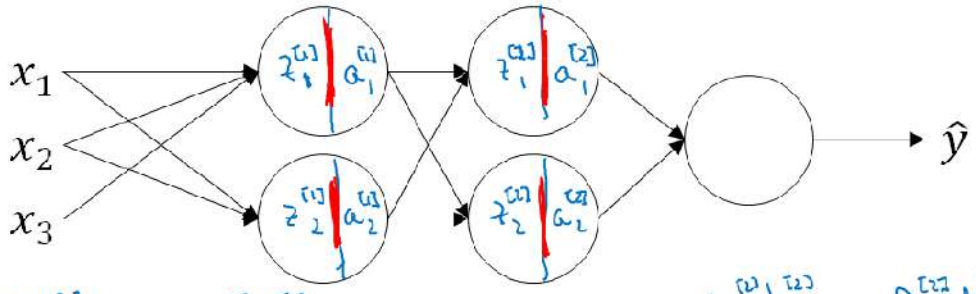
We can see in diagram that we can normalize our data set according to the variances and we saw in an earlier section how this can turn the contours of your learning problem from something that might be very elongated to something that is more round, and easier for an algorithm like **gradient descent to optimize**. So this works, in terms of normalizing the input feature values to a neural network. Now, how about a deeper model? You have not just input features x, but in this layer you have activations **a1**, in this layer, you have activations **a2** and so on. So if you want to train the parameters, say w3, b3, then wouldn't it be nice if you can normalize the mean and variance of **a2** to make the training of w3, b3 more efficient? In the case of logistic regression, we saw how normalizing **x1, x2, x3** maybe helps you train w and b more efficiently. So here, the question is, for any hidden layer, can we normalize, The values of a, let's say a2, in this example but really any hidden layer, so as to train w3 b3 faster, right? Since a2 is the input to the next layer, that therefore affects your training of w3 and b3. So this is what **batch norm does**, **batch normalization**, or batch norm for short, does. Although technically, we'll actually **normalize the values of not a2 but z2**. There are some debates in the deep learning literature about whether you should normalize the value before the activation function, so z2, or whether you should normalize the value after applying the activation function, a2. In practice, normalizing z2 is done much more often. So that's the version I'll present and what I would recommend you use as a default choice. Check below diagram for batch-norm implementation of single layer

Implementing Batch Norm



Fitting Batch Norm into a neural network

Adding Batch Norm to a network



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$

(BN)

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$

$\rightarrow \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$

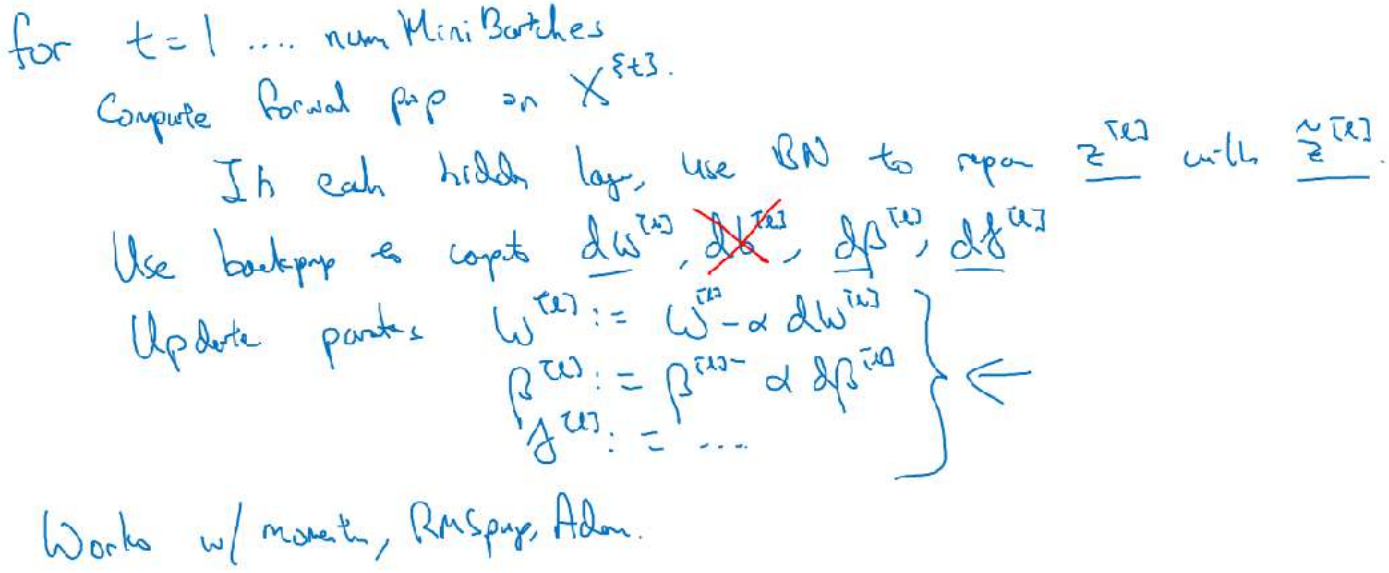
$\rightarrow \beta$

$d\beta^{[l]}$

$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$

tf.nn.batch_normalization

Implementing gradient descent

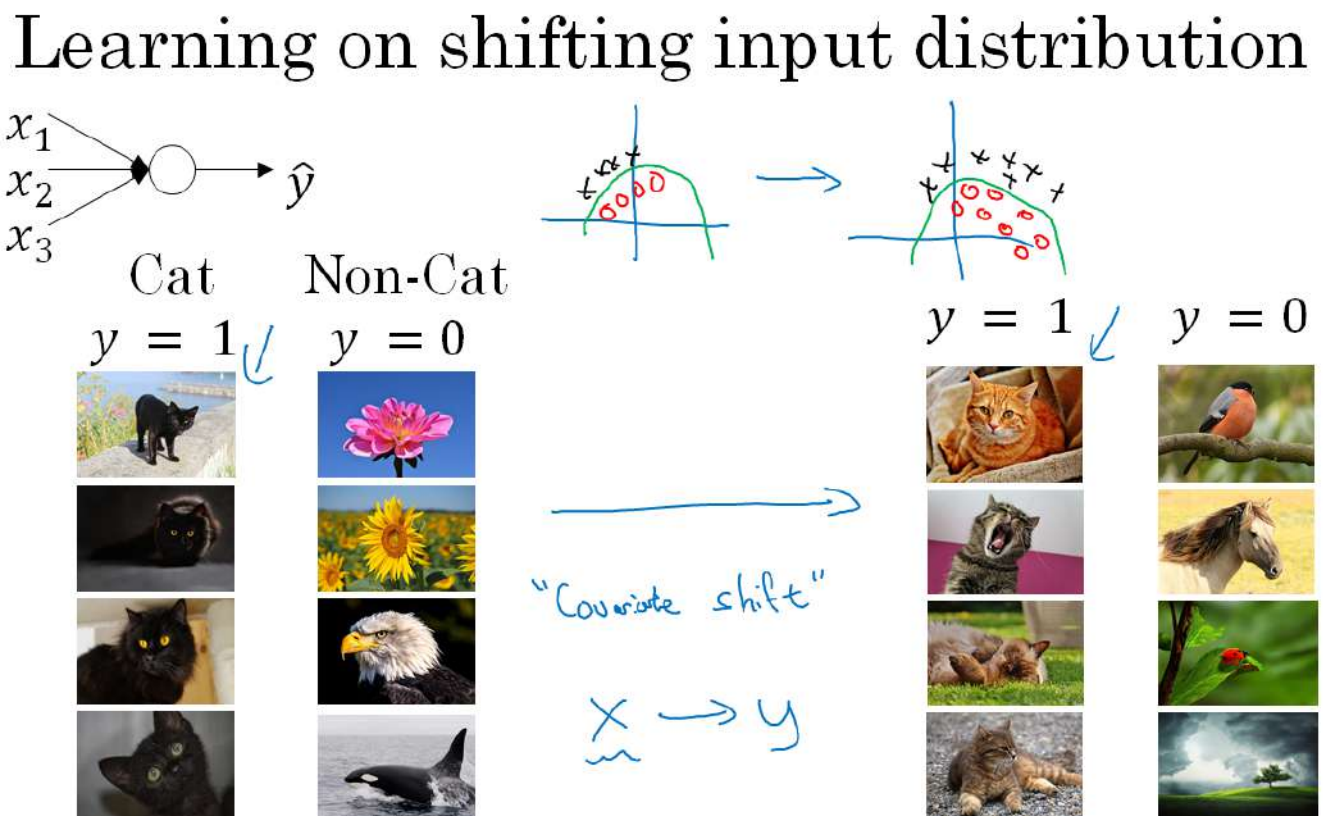


In programming framework like TF implementing Batch-Norm is just a single line of code.

Why does Batch Norm work?

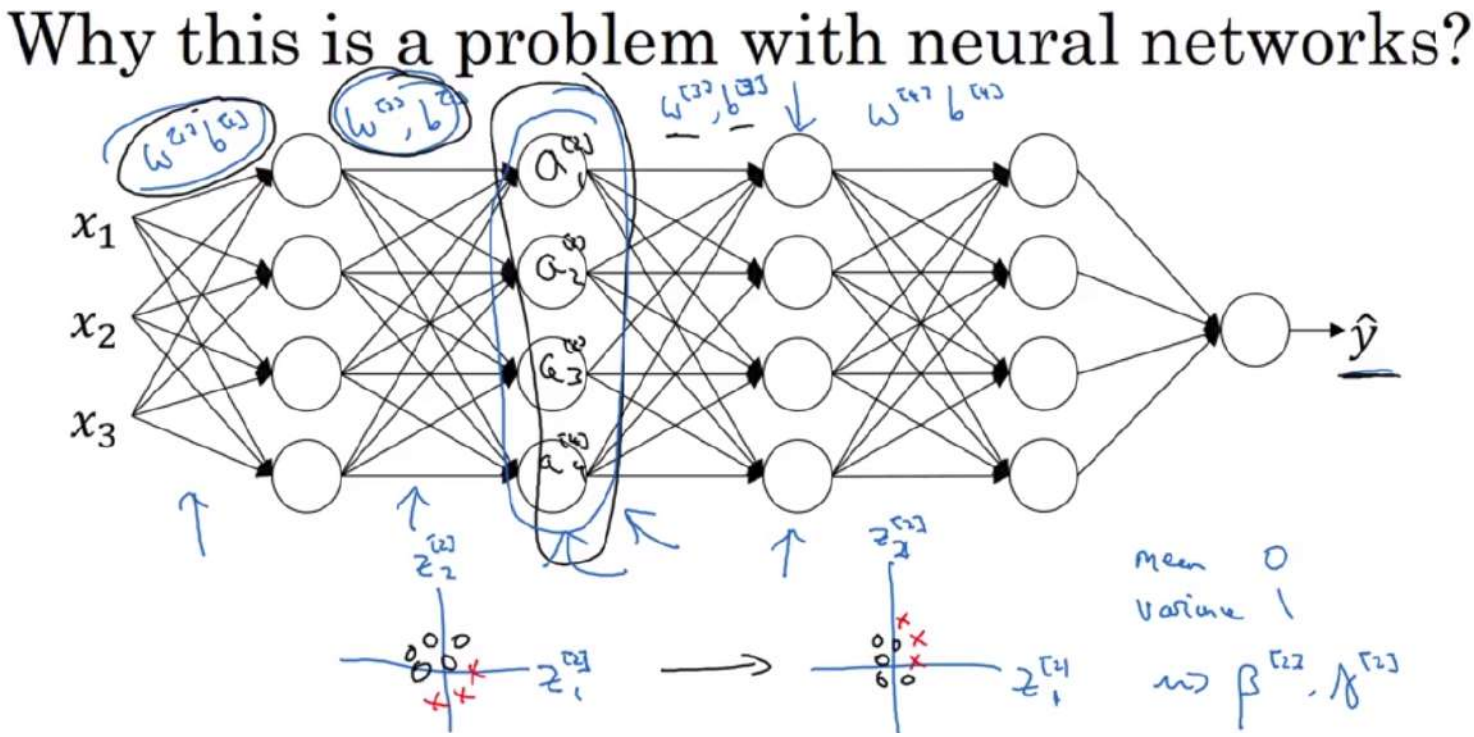
So, why does batch norm work? Here's one reason, we've seen how normalizing the input features, the X's, to mean zero and variance one, how that can speed up learning. So rather than having some features that range from zero to one, and some from one to a 1,000, by normalizing all the features, input features X, to take on a similar range of values that can speed up learning. So, one intuition behind why batch norm works is, this is doing a similar thing, but further values in your hidden units and not just for your input there. Now, this is just a partial picture for what batch norm is doing. There are a couple of further intuitions, that will help you gain a deeper understanding of what batch norm is doing.

Let's take a look at these in this section. A second reason why **batch norm** works, is it makes weights, later or deeper than your network, say the weight on layer 10, more robust to changes to weights in earlier layers of the neural network, say, in layer one. To explain what I mean, let's look at this most vivid example.

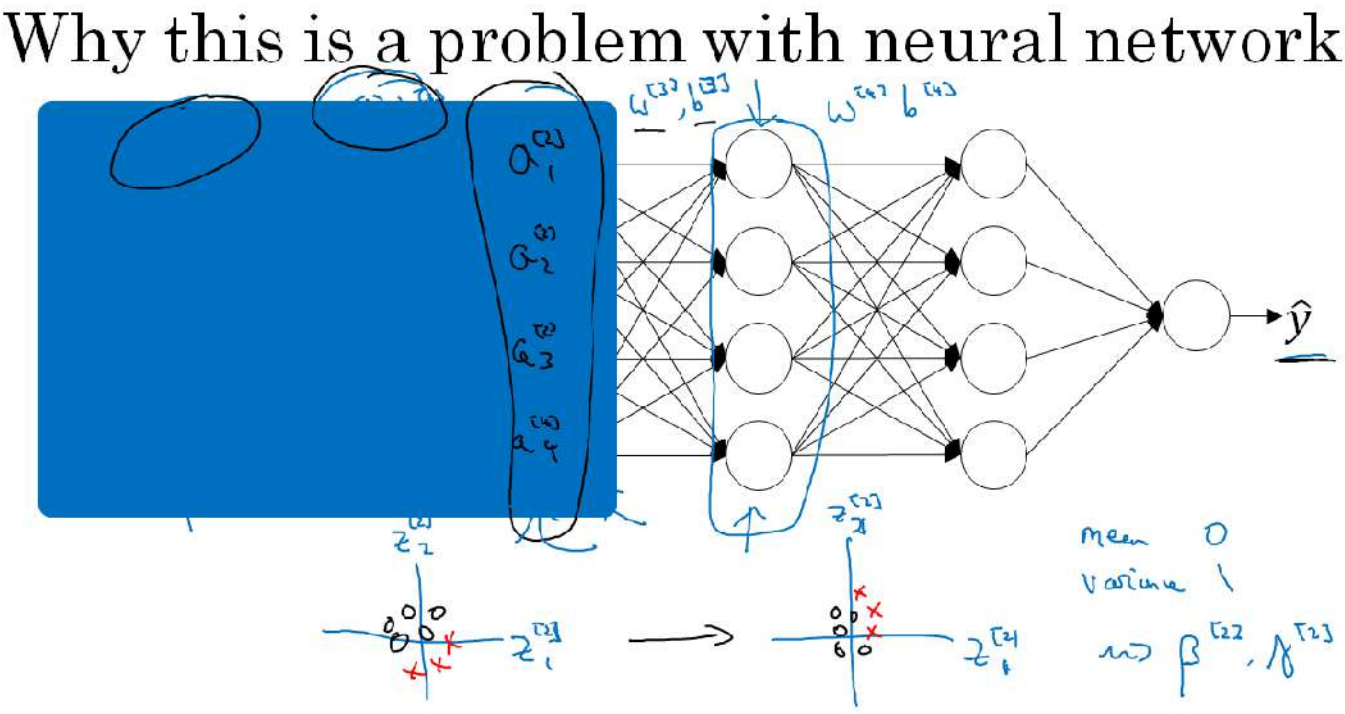


Let's see a training on network, maybe a shallow network, like logistic regression or maybe a deep network, on our famous cat detection toss. But let's say that you've trained our data sets on all images of black cats. If you now try to apply this network to data with colored cats where the positive examples are not just black cats like on the left, but to color cats like on the right, then your classifier might not do very well.

So in pictures, if your training set looks like (x and y axis drawing 1 in image), where you have positive examples (check image) and negative examples (check image), but you were to try to generalize it, to a data set where maybe positive examples are here and the negative examples are here, then you might not expect a module trained on the data on the left to do very well on the data on the right. Even though there might be the same function that actually works well, but you wouldn't expect your learning algorithm to discover that green decision boundary, just looking at the data on the left. So, this idea of your data distribution changing goes by the somewhat fancy name, **covariate shift**. And the idea is that, if you've learned some X to Y mapping, if the distribution of X changes, then **you might need to retrain your learning algorithm**. And this is true even if the function, the ground true function, mapping from X to Y, remains unchanged, which it is in this example, because the ground true function is, is this picture a cat or not. And the need to retain your function becomes even more acute or it becomes even worse if the ground true function shifts as well. So, how does this problem of covariate shift apply to a neural network? Consider a deep network like below

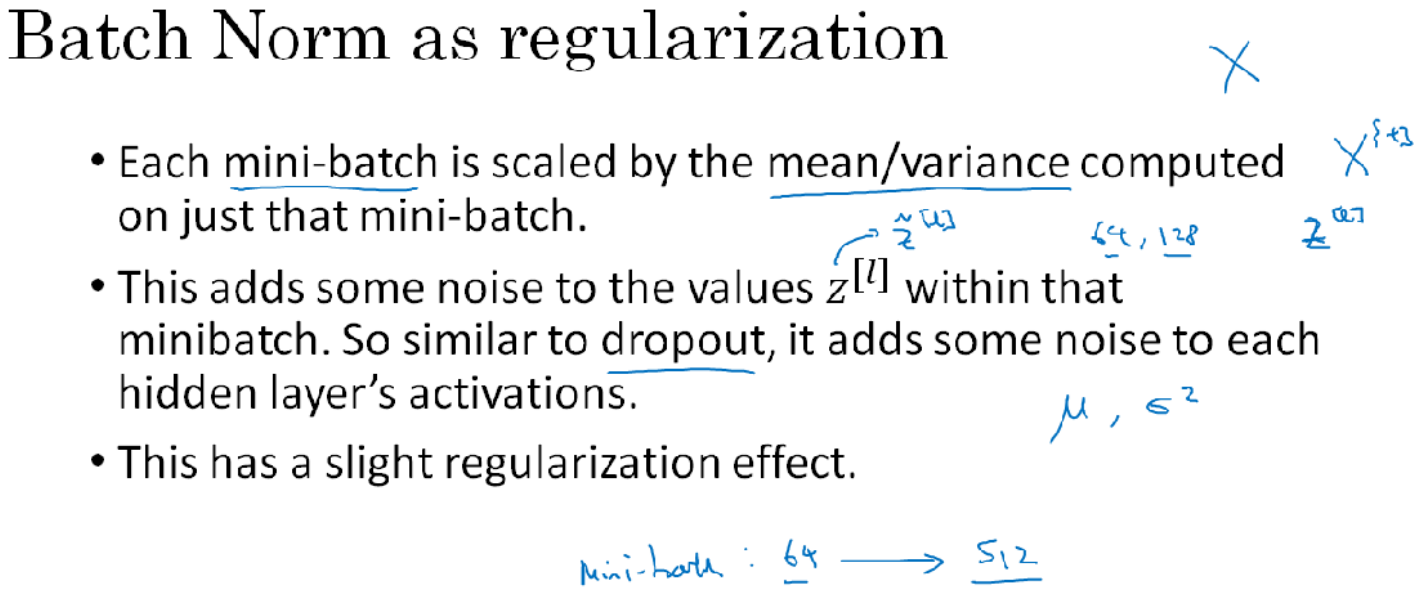


and let's look at the learning process from the perspective of this certain layer, the third hidden layer. So this network has learned the parameters $W^{[3]}$ and $B^{[3]}$. And from the perspective of the third hidden layer, it gets some set of values from the earlier layers, and then it has to do some stuff to hopefully make the output Y-hat close to the ground true value Y. So let me cover up the nose on the left for a second.



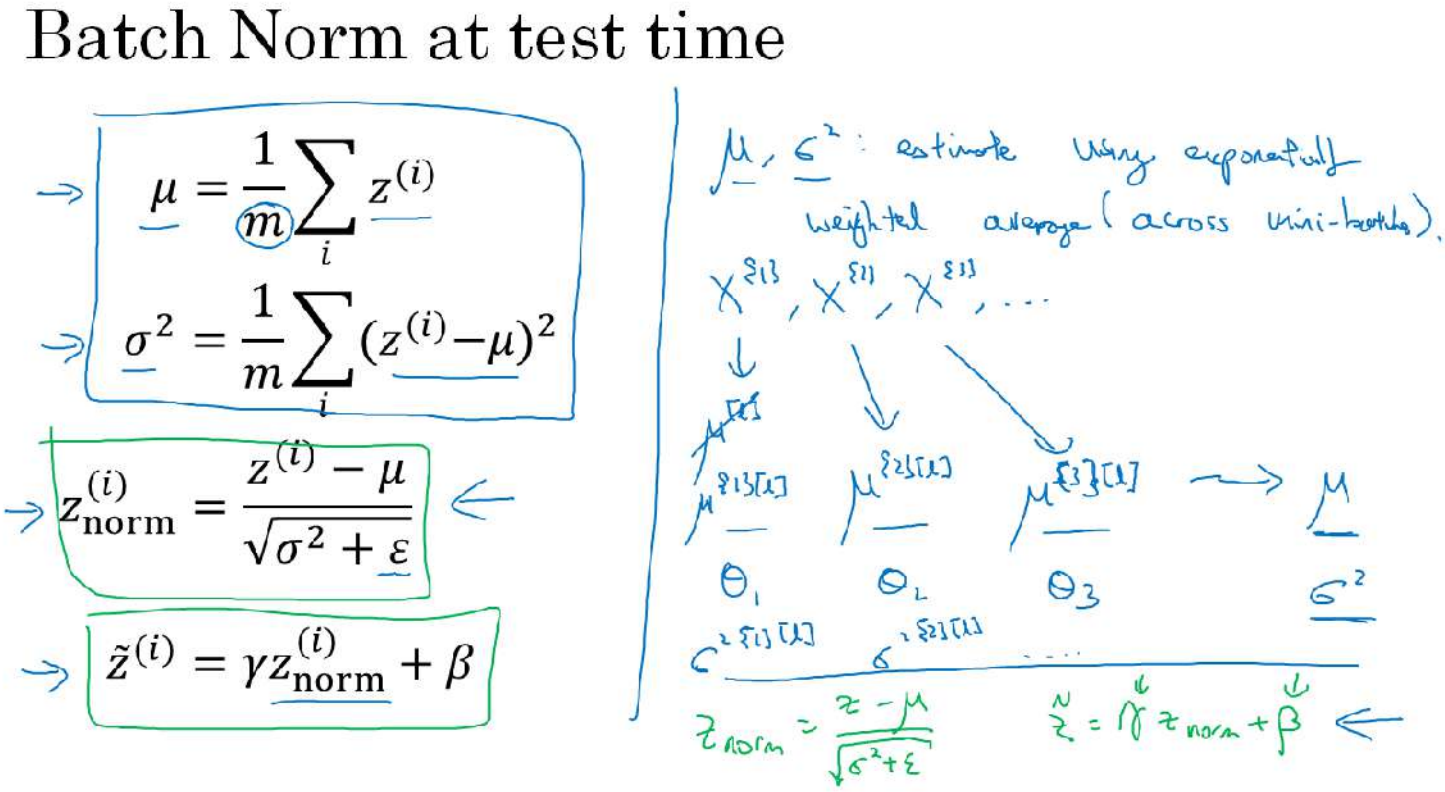
So from the perspective of this third hidden layer, it gets some values, let's call them $a1^{[2]}$, $a2^{[2]}$, $a3^{[2]}$, $a4^{[2]}$ but these values might as well be features x1, x2, x3, x4, and the job of the third hidden layer is to take these values and find a way to map them to Y-hat. So you can imagine doing great intercepts, so that these parameters $W^{[3]}$ $b^{[3]}$ as well as maybe $W^{[4]}$ $b^{[4]}$, and even $W^{[5]}$ $b^{[5]}$, maybe try and learn those parameters, so the network does a good job, mapping from the values we drew in black on the left to the output values Y-hat. But now let's uncover the left of the network again (check first diagram of the section). The network is also adapting parameters $W^{[2]}$ $b^{[2]}$ and $W^{[1]}$ $b^{[1]}$, and so as these parameters change, these values, $a^{[2]}$, will also change. So from the perspective of the third hidden layer, these hidden unit values are changing all the time, and so it's suffering from the **problem of covariate shift** that we talked about on the previous section. So what **batch norm does, is it reduces the amount that the distribution of these hidden unit**

values shifts around. And if it were to plot the distribution of these hidden unit values, maybe this is technically we normalize Z, so this is actually $z1^{[2]}$ $z2^{[2]}$ and I also plot two values instead of four values, so we can visualize in 2D. **What batch norm is saying is that, the values for** $z1^{[2]}$ $z2^{[2]}$ can change, and indeed they will change when the neural network updates the parameters in the earlier layers. **But what batch norm ensures is that no matter how it changes, the mean and variance of $z1^{[2]}$ and $z2^{[2]}$ will remain the same**. **So even if the exact values of $z1^{[2]}$ and $z2^{[2]}$ change, their mean and variance will at least stay same mean zero and variance one**. Or, not necessarily mean zero and variance one, but whatever value is governed by beta 2 and gamma 2 Which, if the neural networks chooses, can force it to be mean=0 and variance=1 or really, any other mean and variance. But what this does is, it limits the amount to which updating the parameters in the earlier layers can affect the distribution of values that the third layer now sees and therefore has to learn on. And **so, batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on**. And even though the input distribution changes a bit, it changes less, and what this does is, even as the earlier layers keep learning, the amounts that this forces the later layers to adapt to as early as layer changes is reduced or, if you will, it weakens the coupling between what the early layers parameters has to do and what the later layers parameters have to do. **And so it allows each layer of the network to learn by itself, a little bit more independently of other layers, and this has the effect of speeding up of learning in the whole network**. Takeaway is that batch norm means that, especially from the perspective of one of the later layers of the neural network, the earlier layers don't get to shift around as much, because they're constrained to have the same mean and variance. And so this makes the job of learning on the later layers easier. It turns out batch norm has a second effect, it has a slight regularization effect. So one non-intuitive thing of a batch norm is that each mini-batch, the mean and variance computed on just that mini-batch as opposed to computed on the entire data set, **that mean and variance has a little bit of noise in it**, because it's computed just on your mini-batch of, say, 64, or 128, or maybe 256 or larger training examples. Batch norm works with mini-batch.



Batch Norm at test time

Batch norm processes your data one mini batch at a time, but the test time you might need to process the examples one at a time. Let's see how you can adapt your network to do that. Recall that during training, here are the equations you'd use to implement batch norm.



So the takeaway from this is that during training time mu and sigma squared are computed on an entire mini batch of say 64 engine, 28 or some number of examples. But that test time, you might need to process a single example at a time. So, the way to do that is to **estimate mu and sigma squared from your training set** and there are many ways to do that. You could in theory run your whole training set through your final network to get mu and sigma squared. But in practice, what **people usually do is implement and exponentially weighted average where you just keep track of the mu and sigma squared values you're seeing during training and use and exponentially the weighted average, also sometimes called the running average, to just get a rough estimate of mu and sigma squared and then you use those values of mu and sigma squared that test time to do the scale and you need the head and unit values Z**. In practice, this process is pretty robust to the exact way you used to estimate mu and sigma squared. So, I wouldn't worry too much about exactly how you do this and if you're using a deep learning framework, they'll usually have some default way to estimate the mu and sigma squared that should work reasonably well as well. But in practice, any reasonable way to estimate the mean and variance of your hidden and unit values Z should work fine at test. So, that's it for batch norm and using it. I think you'll be able to train much deeper networks and get your learning algorithm to run much more quickly. B

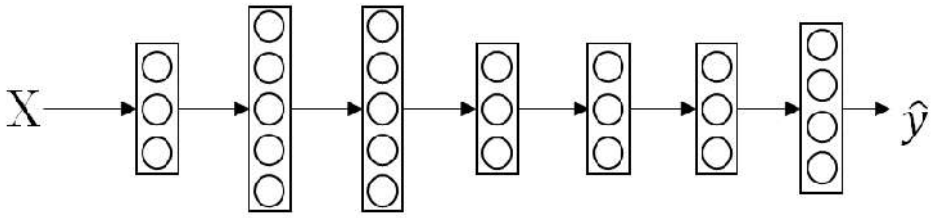
Multi-class classification

Softmax Regression

So far, the classification examples we've talked about have used binary classification, where you had two possible labels, 0 or 1. Is it a cat, is it not a cat? What if we have multiple possible classes? There's a generalization of logistic regression called **Softmax regression**. The less you make predictions where you're trying to recognize one of C or one of multiple classes, rather than just recognize two classes. Let's take a look. Let's say that instead of just recognizing cats you want to recognize cats, dogs, and baby chicks. So we are going to call cats class 1, dogs class 2, baby chicks class 3. And if none of the above, then there's an other or a none of the above class, which we are going to call class 0. So here's an example of the images and the classes they belong to.

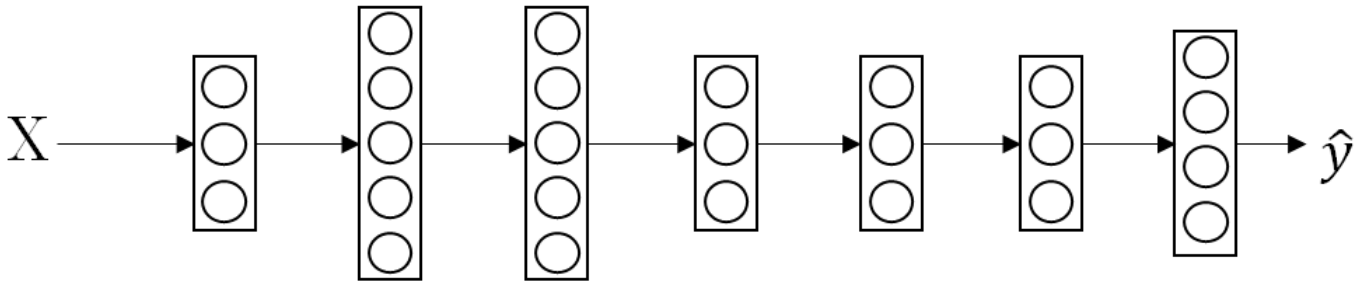
Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1



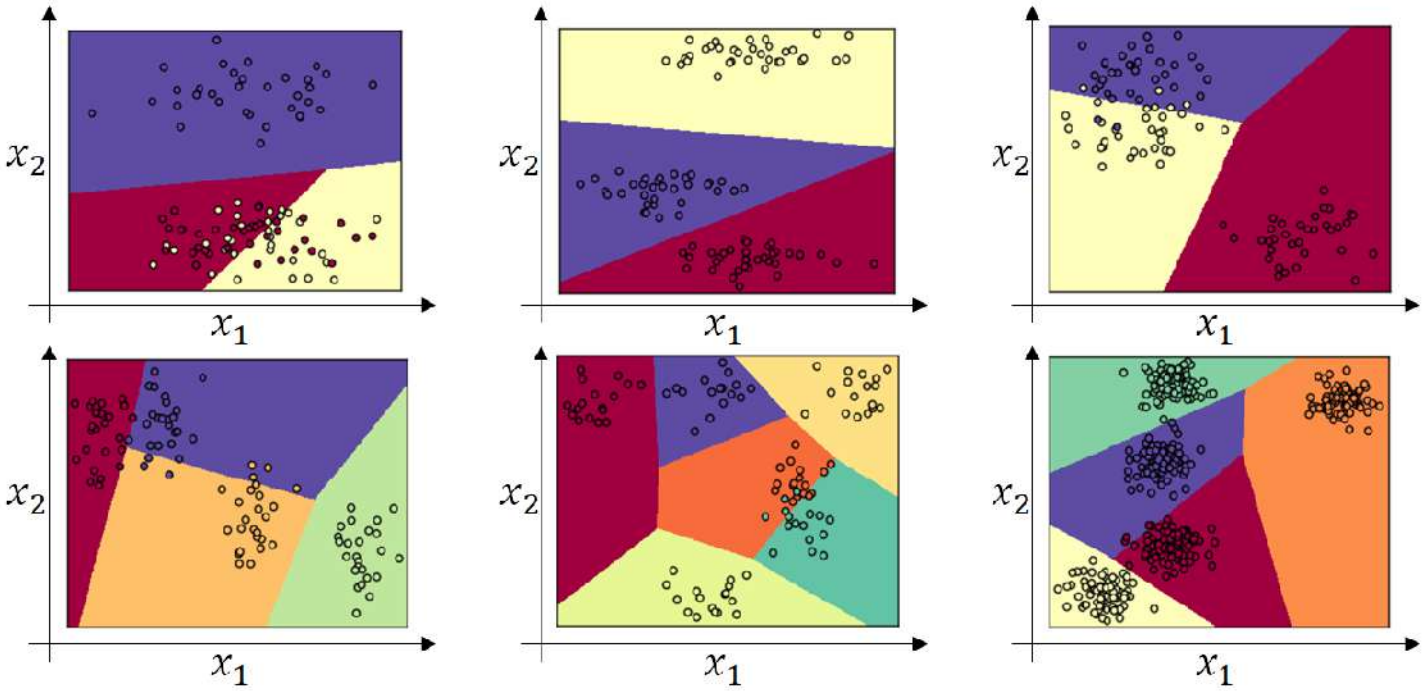
That's a picture of a baby chick, so the class is 3. Cats is class 1, dog is class 2, I guess that's a koala, so that's none of the above, so that is class 0, class 3 and so on. So the notation we're going to use is, we're going to use capital C to denote the number of classes you're trying to categorize your inputs into. And in this case, you have four possible classes, including the other or the none of the above class. So when you have four classes, the numbers indexing your classes would be 0 through capital C minus one. So in other words, that would be zero, one, two or three. In this case, we're going to build a new XY, where the upper layer has four, or in this case the variable capital alphabet C upward units. So N, the number of units upper layer which is layer L is going to equal to 4 or in general this is going to equal to C. And what we want is for the number of units in the upper layer to tell us what is the probability of each of these four classes. So the first node here is supposed to output, or we want it to output the probability that is the other class, given the input x, this will output probability there's a cat. Give an x, this will output probability as a dog. Give an x, that will output the probability. I'm just going to abbreviate baby chick to baby C, given the input x. So here, the output labels y hat is going to be a four by one dimensional vector, because it now has to output four numbers, giving you these four probabilities. And because probabilities should sum to one, the four numbers in the output y hat, they should sum to one. The standard model for getting your network to do this uses what's called a **Softmax layer,** and the output layer in order to generate these outputs. Then write down the map, then you can come back and get some intuition about what the Softmax there is doing.

Softmax layer



So in the final layer of the neural network, you are going to compute as usual the linear part of the layers. So z, capital L, that's the z variable for the final layer. So remember this is layer capital L. So as usual you compute that as wL times the activation of the previous layer plus the biases for that final layer. Now having computer z, you now need to apply what's called the **Softmax activation function**.
let's take a look at how you can train a neural network that uses a Softmax layer.

Softmax examples



The name softmax comes from contrasting it to what's called a **hard max** which would have taken the vector Z and matched it to a vector. So hard max function will look at the elements of Z and just put a 1 in the position of the biggest element of Z and then 0s everywhere else. And so this is a very hard max where the biggest element gets a output of 1 and everything else gets an output of 0. Whereas in contrast, a **softmax** is a more gentle mapping from Z to these probabilities. Softmax regression generalizes logistic regression to C classes. or we can say softmax regression is a generalization of logistic regression to more than two classes.

Course 3: Structuring Machine Learning Projects

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/machine-learning-projects/home/welcome

****
Course 3: Structuring Machine Learning Projects
****
In this course we'll learn how to build a successful machine learning project. If you aspire to be a technical leader in AI, and know how to set direction for your team's work, this course will show you how.

Much of this content has never been taught elsewhere, and is drawn from my experience building and shipping many deep learning products. This course also has two "flight simulators" that let you practice decision-making as a machine learning project leader. This provides "industry experience" that you might otherwise get only after years of ML work experience.
After 2 weeks, you will:
- Understand how to diagnose errors in a machine learning system, and
- Be able to prioritize the most promising directions for reducing error
- Understand complex ML settings, such as mismatched training/test sets, and comparing to and/or surpassing human-level performance
- Know how to apply end-to-end learning, transfer learning, and multi-task learning

Week 1: ML Strategy (1)
Learning Objectives

- Understand why Machine Learning strategy is important
- Apply satisfying and optimizing metrics to set up your goal for ML projects
- Choose a correct train/dev/test split of your dataset
- Understand how to define human-level performance
- Use human-level perform to define your key priorities in ML projects
- Take the correct ML Strategic decision based on observations of performances and dataset

Introduction to ML Strategy
Why ML Strategy

In this course we'll learn how to much more quickly and efficiently get your machine learning systems working. So, what is machine learning strategy. Let's start with a motivating example. Let's say you are working on your cat classifier. And after working it for some time, you've gotten your system to have 90% accuracy, but this isn't good enough for your application. You might then have a lot of ideas as to how to improve your system. For example, you might think well let's **collect more data**, more training data. Or you might say, maybe your training set isn't diverse enough yet, you should collect images of cats in more diverse poses, or maybe a more diverse set of negative examples. Well maybe you want to train the algorithm longer with **gradient descent**. Or maybe you want to try a different optimization algorithm, like the **Adam optimization algorithm**. Or maybe trying a **bigger network** or a smaller network or maybe you want to try to dropout or maybe **L2 regularization**. Or maybe you want to change the network architecture such as changing activation functions, changing the number of hidden units and so on and so on. When trying to improve a deep learning system, you often have a lot of ideas or things you could try. And the problem is that if you choose poorly, it is entirely possible that you end up spending six months charging in some direction only to realize after six months that that didn't do any good. For example, I've seen some teams spend literally six months collecting more data only to realize after six months that it barely improved the performance of their system. So, assuming you don't have six months to waste on your problem, won't it be nice if you had quick and effective ways to figure out which of all of these ideas and maybe even other ideas, are worth pursuing and which ones you can safely discard. So what I hope to do in this course is teach you a

number of strategies, that is, ways of analyzing a machine learning problem that will point you in the direction of the most promising things to try. What I will do in this course also is share with you a number of lessons I've learned through building and shipping large number of deep learning products. And I think these materials are actually quite unique to this course. I don't see a lot of these ideas being taught in universities' deep learning courses for example. It turns out also that machine learning strategy is changing in the era of deep learning because the things you could do are now different with deep learning algorithms than with previous generation of machine learning algorithms. I hope that these ideas will help you become much more effective at getting your deep learning systems to work.

Orthogonalization

One of the challenges with building machine learning systems is that there's so many things you could try, so many things you could change. Including, for example, so many hyperparameters you could tune. One of the things I've noticed is about the most effective machine learning people is they're very clear-eyed about what to tune in order to try to achieve one effect. This is a process we call **orthogonalization**. Let me tell you what I mean.

For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true. First, is that you usually have to make sure that you're at **least doing well on the training set**. So performance on the training set needs to pass some acceptability assessment. For some applications, this might mean doing comparably to human level performance. But this will depend on your application, and we'll talk more about comparing to human level performance next week.

But after doing well on the training sets, you then hope that this leads to also doing well on the **dev set**. And you then hope that this also does well on the **test set**. And finally, you hope that doing well on the test set on the cost function results in your system performing in the real world. So you hope that this resolves in happy cat picture app users, for example. So to relate back to the TV tuning example, if the picture of your TV was either too wide or too narrow, you wanted one knob to tune in order to adjust that. You don't want to have to carefully adjust five different knobs, which also affect different things. You want one knob to just affect the width of your TV image. So in a similar way, if your algorithm is not fitting the training set well on the cost function, you want one knob, yes, that's my attempt to draw a knob. Or maybe one specific set of knobs that you can use, to make sure you can tune your algorithm to make it fit well on the training set. So the knobs you use to tune this are, you might train a bigger network or you might switch to a better optimization algorithm, like the **Adam optimization algorithm**, and so on. In contrast, if you find that the algorithm is not fitting the dev set well, then there's a separate set of knobs. Yes, that's my not very artistic rendering of another knob, you want to have a distinct set of knobs to try. So for example, if your algorithm is not doing well on the dev set, it's doing well on the training set but not on the dev set, then you have a set of knobs around regularization that you can use to try to make it satisfy the second criteria. So the analogy is, now that you've tuned the width of your TV set, if the height of the image isn't quite right, then you want a different knob in order to tune the height of the TV image. And you want to do this hopefully without affecting the width of your TV image too much. And getting a bigger training set would be another knob you could use, that helps your learning algorithm generalize better to the dev set. Now, having adjusted the width and height of your TV image, well, what if it doesn't meet the third criteria? What if you do well on the dev set but not on the test set? If that happens, then the knob you tune is, you probably want to get a bigger dev set. Because if it does well **on the dev set but not the test set, it probably means you've overtuned to your dev set, and you need to go back and find a bigger dev set** And finally, if it does well on the test set, but it isn't delivering to you a happy cat picture app user, then what that means is that you want to go back and change either the dev set or the cost function. Because if doing well on the test set according to some cost function doesn't correspond to your algorithm doing what you need it to do in the real world, then it means that either your dev test set distribution isn't set correctly, or your cost function isn't measuring the right thing.

So Orthogonalization or orthogonality is a system design property that assures that modifying an instruction or a component of an algorithm will not create or propagate side effects to other components of the system. It becomes easier to verify the algorithms independently from one another, it reduces testing and development time. When a supervised learning system is design, these are the 4 assumptions that needs to be true and orthogonal.

1. Fit training set well in cost function
- If it doesn't fit well, the use of a bigger neural network or switching to a better optimization algorithm might help.
2. Fit development set well on cost function
- If it doesn't fit well, regularization or using bigger training set might help.
3. Fit test set well on cost function
- If it doesn't fit well, the use of a bigger development set might help
4. Performs well in real world
- If it doesn't perform well, the development test set is not set correctly or the cost function is not evaluating the right thing

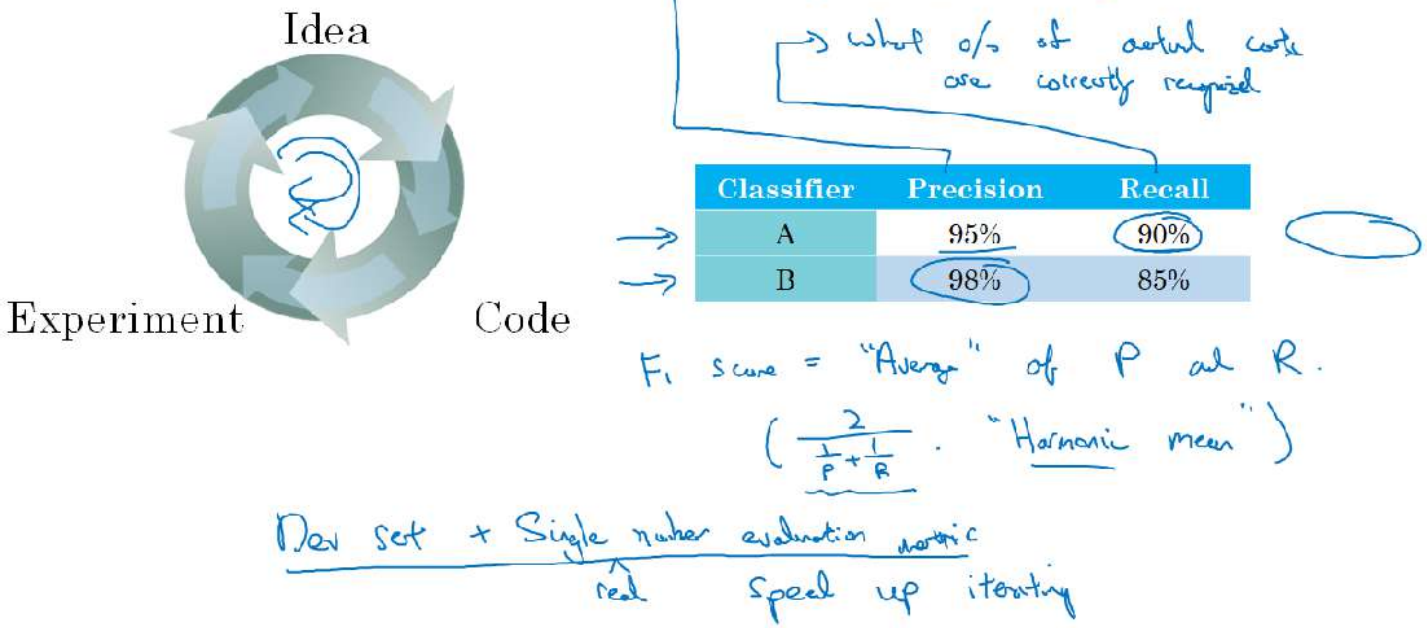
Setting up your goal

Single number evaluation metric

Whether you're tuning hyperparameters, or trying out different ideas for learning algorithms, or just trying out different options for building your machine learning system. You'll find that your progress will be much faster if you have a **single real number evaluation metric** that lets you quickly tell if the new thing you just tried is working better or worse than your last idea. So when teams are starting on a machine learning project, I often recommend that you set up a single real number evaluation metric for your problem. Let's look at an example.

You've heard me say before that applied machine learning is a very empirical process. We often have an idea, code it up, run the experiment to see how it did, and then use the outcome of the experiment to refine your ideas. And then keep going around this loop as you keep on improving your algorithm. So let's say for your classifier, you had previously built some classifier A. And by changing the hyperparameters and the training sets or some other thing, you've now trained a new classifier, B. So one reasonable way to evaluate the performance of your classifiers is to look at its precision and recall. The exact details of what's precision and recall don't matter too much for this example. But briefly, the definition of precision is, of the examples that **your classifier recognizes as cats**, What percentage actually are cats? So if classifier A has 95% precision, this means that when classifier A says something is a cat, there's a 95% chance it really is a cat. And recall is, **of all the images that really are cats**, what percentage were correctly recognized by your classifier? So what percentage of actual cats, Are correctly recognized?

Using a single number evaluation metric



So if classifier A is 90% recall, this means that of all of the images in, say, your dev sets that really are cats, classifier A accurately pulled out 90% of them. So don't worry too much about the definitions of precision and recall. **It turns out that there's often a tradeoff between precision and recall, and you care about both**. You want that, when the classifier says something is a cat, there's a high chance it really is a cat. But of all the images that are cats, you also want it to pull a large fraction of them as cats. So it might be reasonable to try to evaluate the classifiers in terms of its precision and its recall. The problem with using precision recall as your evaluation metric is that if classifier A does better on recall, which it does in the diagram above, the classifier B does better on precision, then you're not sure which classifier is better and if you're trying out a lot of different ideas, a lot of different hyperparameters, you want to rather quickly try out not just two classifiers, but maybe a dozen classifiers and quickly pick out the, quote, best ones, so you can keep on iterating from there and with two evaluation metrics, it is difficult to know how to quickly pick one of the two or quickly pick one of the ten.

So what is recommended is rather than using two numbers, precision and recall, to pick a classifier, you just have to find a new evaluation metric that combines precision and recall. In the machine learning literature, the standard way to combine precision and recall is something called an F1 score. And the details of F1 score aren't too important, but informally, you can think of this as the average of precision, P, and recall, R.
Check below diagram for details on F1 score.

Single number evaluation metric

To choose a classifier, a well-defined development set and an evaluation metric speed up the iteration process.

Example : Cat vs Non- cat

y = 1, cat image detected

Predict class $\hat{y}$	Actual class y: 1	Actual class y: 0
1	True positive	False positive
0	False negative	True negative

Precision

Of all the images we predicted y=1, what fraction of it have cats?

$$\text{Precision (\%)} = \frac{True\ positive}{Number\ of\ predicted\ positive} x\ 100 = \frac{True\ positive}{(True\ positive + False\ positive)}\ x\ 100$$

Recall

Of all the images that actually have cats, what fraction of it did we correctly identifying have cats?

$$\text{Recall (\%)} = \frac{True\ positive}{Number\ of\ predicted\ actually\ positive} x\ 100 = \frac{True\ positive}{(True\ positive + False\ negative)}\ x\ 100$$

Let's compare 2 classifiers A and B used to evaluate if there are cat images:

Classifier	Precision (p)	Recall (r)
A	95%	90%
B	98%	85%

In this case the evaluation metrics are precision and recall.

For classifier A, there is a 95% chance that there is a cat in the image and a 90% chance that it has correctly detected a cat. Whereas for classifier B there is a 98% chance that there is a cat in the image and a 85% chance that it has correctly detected a cat.

The problem with using precision/recall as the evaluation metric is that you are not sure which one is better since in this case, both of them have a good precision et recall. F1-score, a harmonic mean, combine both precision and recall.

$$\text{F1-Score} = \frac{2}{\frac{1}{p}+\frac{1}{r}}$$

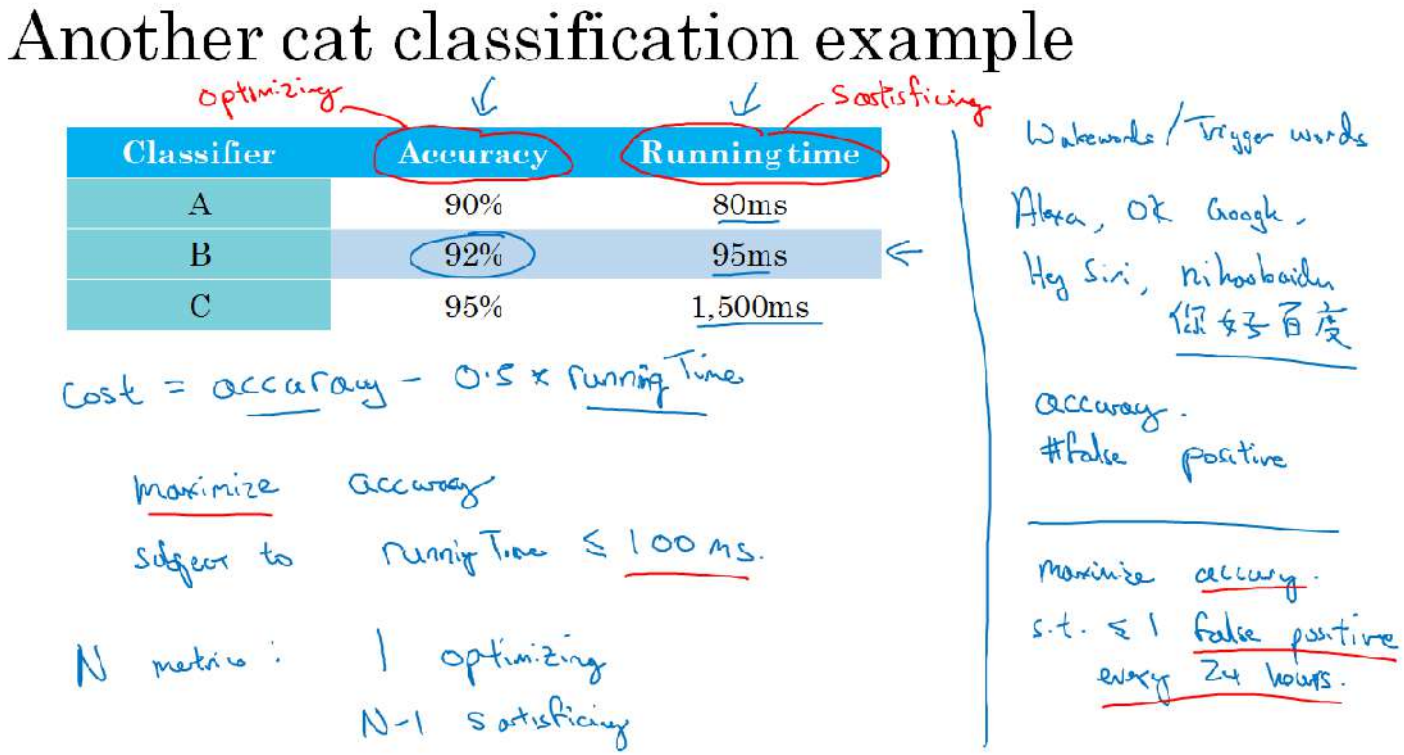
Classifier	Precision (p)	Recall (r)	F1-Score
A	95%	90%	92.4 %
B	98%	85%	91.0%

Classifier A is a better choice. F1-Score is not the only evaluation metric that can be use, the average, for example, could also be an indicator of which classifier to use.

So what we have learned in this section is that having a single number evaluation metric can really improve your efficiency or the efficiency of your tea, in making decisions.

Satisficing and optimistic metric

It's not always easy to combine all the things you care about into a single row number evaluation metric. In those cases I've found it sometimes useful to set up **satisficing as well as optimizing matrix**. Let me show you what I mean. Let's say that you've decided you care about the classification accuracy of your cat's classifier, this could have been **F1 score** or some other measure of accuracy, but let's say that in addition to accuracy you also care about the **running time**.



So how long it takes to classify an image and classifier A takes 80 milliseconds, B takes 95 milliseconds, and C takes 1,500 milliseconds, that's 1.5 seconds to classify an image. So one thing you could do is combine accuracy and running time into an overall evaluation metric. And so the costs such as maybe the overall cost is accuracy minus 0.5 times running time. But maybe it seems a bit artificial to combine accuracy and running time using a formula like this, like a linear weighted sum of these two things. **So here's something else you could do instead which is that you might want to choose a classifier that maximizes accuracy but subject to that the running time, that is the time it takes to classify an image, that that has to be less than or equal to 100 milliseconds. So in this case we would say that accuracy is an optimizing metric because you want to maximize accuracy**. You want to do as well as possible on accuracy but that running time is what we call a **satisficing metric**. Meaning that it just has to be good enough, it just needs to be **less than 100 milliseconds** and beyond that you don't really care, or at least you don't care that much. So this will be a pretty reasonable way to trade off or to put together accuracy as well as running time. And it may be the case that so long as the running time is less that 100 milliseconds, your users won't care that much whether it's 100 milliseconds or 50 milliseconds or even faster. And by defining optimizing as well as satisficing matrix, this gives you a clear way to pick the, quote, best classifier, which in this case would be classifier B because of all the ones with a running time better than 100 milliseconds it has the best accuracy.

Satisficing and optimizing metric

There are different metrics to evaluate the performance of a classifier, they are called evaluation matrices. They can be categorized as satisficing and optimizing matrices. It is important to note that these evaluation matrices must be evaluated on a training set, a development set or on the test set.

Example: Cat vs Non-cat

Classifier	Accuracy	Running time
A	90%	80 ms
B	92%	95 ms
C	95%	1 500 ms

In this case, accuracy and running time are the evaluation matrices. Accuracy is the optimizing metric, because you want the classifier to correctly detect a cat image as accurately as possible. The running time which is set to be under 100 ms in this example, is the satisficing metric which mean that the metric has to meet expectation set.

The general rule is:

$$N_{metric}: \begin{cases} 1 & Optimizing\ metric \\ N_{metric} - 1 & Satisficing\ metric \end{cases}$$

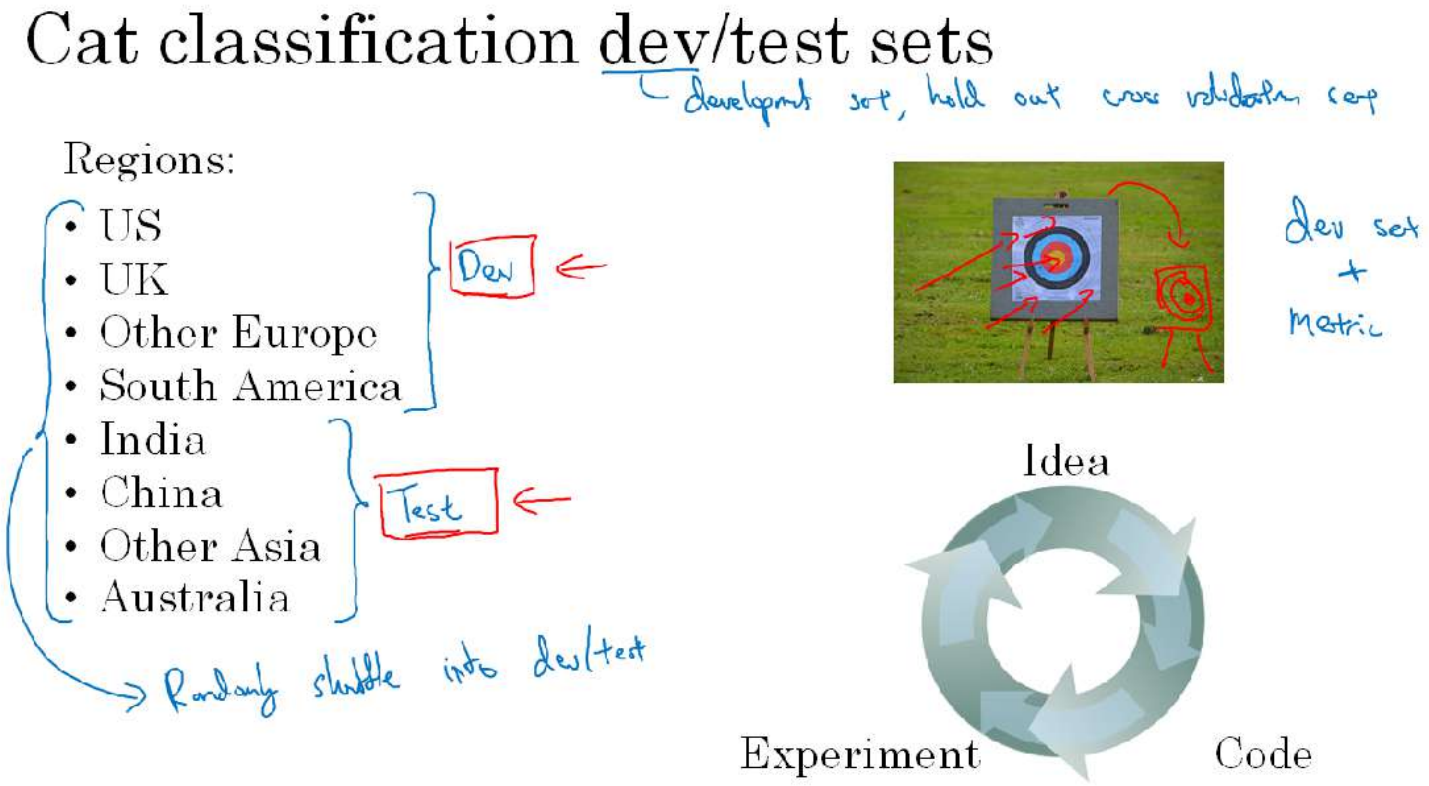
Here's another example. Let's say you're building a system to detect wake words, also called trigger words. So this refers to the voice control devices like the Amazon Echo where you wake up by saying Alexa or some Google devices which you wake up by saying okay Google or some Apple devices which you wake up by saying Hey Siri. So these are the wake words you use to tell one of these voice control devices to wake up and listen to something you want to say so you might care about the accuracy of your trigger word detection system. So when someone says one of these trigger words, how likely are you to actually wake up your device, and you might also care about the number of false positives. So when no one actually said this trigger word, how often does it randomly wake up? So in this case maybe one reasonable way of combining these two evaluation matrix might be to maximize accuracy, so when someone says one of the trigger words, maximize the chance that your device wakes up. And subject to that, you have at most one false positive every 24 hours of operation, right? So that your device randomly wakes up only once per day on average when no one is actually talking to it. So in this case accuracy is the optimizing metric and a number of false positives every 24 hours is the **satisficing metric** where you'd be satisfied so long as there is at most one false positive every 24 hours.

To summarize, if there are multiple things you care about by say there's one as the optimizing metric that you want to do as well as possible on and one or more as satisficing metrics were you'll be satisfice. Almost it does better than some threshold you can now have an almost automatic way of quickly looking at multiple core size and picking the, quote, best one. Now these evaluation matrix must be evaluated or calculated on a training set or a development set or maybe on the test set. So one of the things you also need to do is set up training, dev or development, as well as test sets.

Train/dev/test distributions

The way you set up your training dev, or development sets and test sets, can have a huge impact on how rapidly you or your team can make progress on building machine learning

application. The same teams, even teams in very large companies, set up these data sets in ways that really slows down, rather than speeds up, the progress of the team. Let's take a look at how you can set up these data sets to maximize your team's efficiency. In this section, I want to focus on how you set up your dev and test sets. So, that dev set is also called the development set, or sometimes called the hold out cross validation set. And, workflow in machine learning is that you try a lot of ideas, train up different models on the training set, and then use the dev set to evaluate the different ideas and pick one. And, keep innovating to improve dev set performance until, finally, you have one clause that you're happy with that you then evaluate on your test set. Now, let's say, by way of example, that you're building a cat classifier, and you are operating in these regions: in the U.S, U.K, other European countries, South America, India, China, other Asian countries, and Australia. So, how do you set up your dev set and your test set? Well, one way you could do so is to pick four of these regions (check diagram below). I'm going to use these four but it could be four randomly chosen regions. And say, that data from these four regions will go into the dev set. And, the other four regions, I'm going to use these four, could be randomly chosen four as well, that those will go into the test set.



It turns out, this is a very bad idea because in this example, your dev and test sets come from different distributions. I would, instead, recommend that you find a **way to make your dev and test sets come from the same distribution**. So, here's what I mean. One picture to keep in mind is that, I think, setting up your dev set, plus, your single role number evaluation metric, that's like placing a target and telling your team where you think is the bull's eye you want to aim at. Because, what happen once you've established that dev set and the metric is that, the team can innovate very quickly, try different ideas, run experiments and very quickly use the dev set and the metric to evaluate crossfires and try to pick the best one. So, machine learning teams are often very good at shooting different arrows into targets and innovating to get closer and closer to hitting the bullseye. So, doing well on your metric on your dev sets. And, the problem with how we've set up the dev and test sets in the example on the left is that, your team might spend months innovating to do well on the dev set only to realize that, when you finally go to test them on the test set, that data from these four countries or these four regions at the bottom, might be very different than the regions in your dev set. So, you might have a nasty surprise and realize that, all the months of work you spent optimizing to the dev set, is not giving you good performance on the test set. **So, having dev and test sets from different distributions is like**

setting a target, having your team spend months trying to aim closer and closer to bull's eye, only to realize after months of work that, you'll say, "Oh wait, to test it, I'm going to move target over here." And, the team might say, "Well, why did you make us spend months optimizing for a different bull's eye when suddenly, you can move the bull's eye to a different location somewhere else?" So, to avoid this, what **I recommend instead is that, you take all this randomly shuffled data into the dev and test set**. **So that, both the dev and test sets have data from all eight regions and that the dev and test sets really come from the same distribution, which is the distribution of all of your data mixed together.** Here's another example. This is a, actually, true story but with some details changed. So, I know a machine learning team that actually spent several months optimizing on a dev set which was comprised of loan approvals for medium income zip codes. So, the specific machine learning problem was, "Given an input X about a loan application, can you predict why and which is, whether or not, they'll repay the loan?" So, this helps you decide whether or not to approve a loan. And so, the dev set came from loan applications. They came from medium income zip codes. Zip codes is what we call postal codes in the United States. But, after working on this for a few months, the team then, suddenly decided to test this on data from low income zip codes or low income postal codes. And, of course, the distributional data for medium income and low income zip codes is very different. And, the classifier, that they spend so much time optimizing in the former case, just didn't work well at all on the latter case. And so, this particular team actually wasted about three months of time and had to go back and really re-do a lot of work. And, what happened here was, the team spent three months aiming for one target, and then, after three months, the manager asked, "Oh, how are you doing on hitting this other target?" This is a totally different location. And, it just was a very frustrating experience for the team. So, what I recommand for setting up a dev set and test set is, **choose a dev set and test set to reflect data you expect to get in future and consider important to do well on.** And, in particular, the dev set and the test set here, should come from the same distribution. So, whatever type of data you expect to get in the future, and once you do well on, try to get data that looks like that. And, whatever that data is, put it into both your dev set and your test set. Because that way, you're putting the target where you actually want to hit and you're having the team innovate very efficiently to hitting that same target, hopefully, the same targets well.

Important take away from this section is that, setting up the dev set, as well as the validation metric, is really defining what target you want to aim at. And hopefully, by setting the dev set and the test set to the same distribution, you're really aiming at whatever target you hope your machine learning team will hit. **The way you choose your training set will affect how well you can actually hit that target** but, we can talk about that separately in upcoming section.

Size of the dev and test sets

In the last section, we saw how our dev and test sets should come from the same distribution, but how long should they be? The guidelines to help set up your dev and test sets are changing in the Deep Learning era. Let's take a look at some best practices. You might have heard of the rule of thumb in machine learning of taking all the data you have and using a 70/30 split into a train and test set, or have you had to set up train dev and test sets maybe, you would use a 60% training and 20% dev and 20% tests. In earlier eras of machine learning, this was pretty reasonable, especially back when data set sizes were just smaller. So if you had a hundred examples in total, these 70/30 or 60/20/20 rule of thumb would be pretty reasonable. If you had thousand examples, maybe if you had ten thousand examples, these things are not unreasonable. But in the modern machine learning era, we are now used to working with much larger data set sizes. So let's say you have a million training examples, it might be quite reasonable to set up your data so that you have 98% in the training set, 1% dev, and 1% test because if you have a million examples, then 1% of that, is 10,000 examples, and that might be plenty enough for a dev set or for a test set. So, in the modern Deep Learning era where sometimes we have much larger data sets, It's quite reasonable to use a much smaller than 20 or 30% of your data for a dev set or a test set. And because Deep Learning algorithms have such a huge hunger for data, I'm seeing that, the problems we have large data sets that have much larger fraction of it goes into the training set. So, how about the test set? Remember the purpose of your test set is that, after you finish developing a system, the **test set helps evaluate how good your final system is**. The guideline is, to set your test set to big enough to give high confidence in the overall performance of your system. So, unless you need to have a very accurate measure of how well your final system is performing, maybe you don't need millions and millions of examples in a test set, and maybe for your application if you think that having 10,000

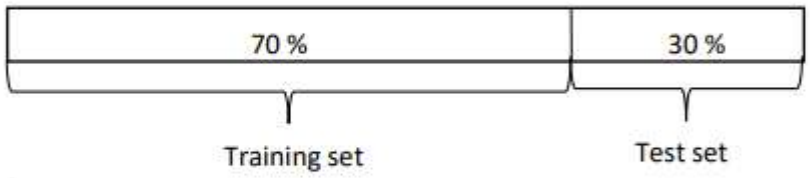
examples gives you enough confidence to find the performance on maybe 100,000 or whatever it is, that might be enough. And this could be much less than, say 30% of the overall data set, depend on how much data you have. For some applications, maybe you don't need a high confidence in the overall performance of your final system. Maybe all you need is a train and dev set, And I think, not having a test set might be okay. In fact, what sometimes happened was, people were talking about using train test splits but what they were actually doing was iterating on the test set. So rather than test set, what they had was a train dev split and no test set. If you're actually tuning to this set, to this dev set and this test set, It's better to call the dev set. Although I think in the history of machine learning, not everyone has been completely clean and completely records of about calling the dev set when it really should be treated as test set. But, if all you care about is having some data that you train on, and having some data to tune to, and you're just going to shake the final system and not worry too much about how it was actually doing, I think it will be healthy and just call the train dev set and acknowledge that you have no test set. This a bit unusual? I'm definitely not recommending not having a test set when building a system. I do find it reassuring to have a separate test set you can use to get an unbiased estimate of how I was doing before you shift it, but if you have a very large dev set so that you think you won't overfit the dev set too badly. Maybe it's not totally unreasonable to just have a train dev set, although it's not what I usually recommend.

So to summarize, in the era of big data, I think the old rule of thumb of a 70/30 is that, that no longer applies. And the trend has been to use more data for training and less for dev and test, especially when you have a very large data sets. And the rule of thumb is really to try to set the dev set to big enough for its purpose, which helps you evaluate different ideas and the purpose of test set is to help you evaluate your final cost. You just have to set your test set big enough for that purpose, and that could be much less than 30% of the data. Check below diagram for summary:

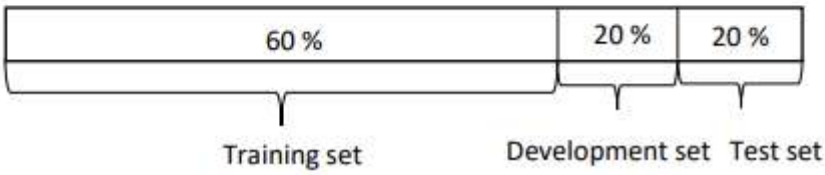
Size of the development and test sets

Old way of splitting data

We had smaller data set therefore we had to use a greater percentage of data to develop and test ideas and models.

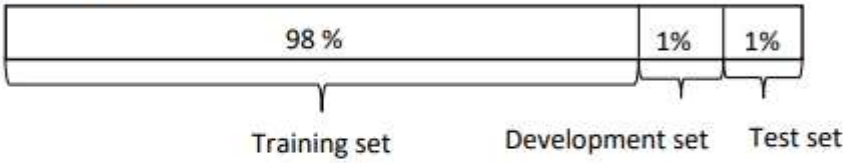


Or



Modern era – Big data

Now, because a large amount of data is available, we don't have to compromised as much and can use a greater portion to train the model.



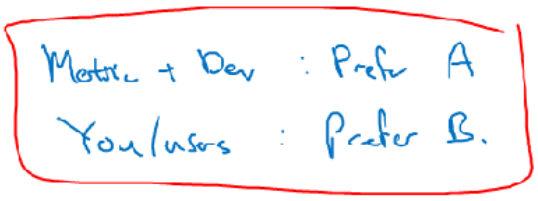
Guidelines

- Set up the size of the test set to give a high confidence in the overall performance of the system.
- Test set helps evaluate the performance of the final classifier which could be less 30% of the whole data set.
- The development set has to be big enough to evaluate different ideas.

When to change dev/test sets and metrics

You've seen how set to have a dev set and evaluation metric is like placing a target somewhere for your team to aim at. But sometimes partway through a project you might realize you put your target in the wrong place. In that case you should move your target. Let's take a look at an example. Let's say you build a cat classifier to try to find lots of pictures of cats to show to your cat loving users and the metric that you decided to use is classification error. So algorithms A and B have, respectively, 3 percent error and 5 percent error, so it seems like Algorithm A is doing better. But let's say you try out these algorithms, you look at these algorithms and Algorithm A, for some reason, is letting through a lot of the pornographic images. So if you shift Algorithm A the users would see more cat images because you'll see 3 percent error and identify cats, but it also shows the users some pornographic images which is totally unacceptable both for your company, as well as for your users. In contrast, Algorithm B has 5 percent error so this classifies fewer images but it doesn't have pornographic images. So from your company's point of view, as well as from a user acceptance point of view, Algorithm B is actually a much better algorithm because it's not letting through any pornographic images. So, what has happened in this example is that Algorithm A is doing better on evaluation metric. It's getting 3 percent error but it is actually a worse algorithm. In this case, the evaluation metric plus the dev set prefers Algorithm A because they're saying, look, Algorithm A has lower error which is the metric you're using but you and your users prefer Algorithm B because it's not letting through pornographic images. So when this happens, when your evaluation metric is no longer correctly rank ordering preferences between algorithms, in this case is mispredicting that Algorithm A is a better algorithm, then that's a sign that you should change your evaluation metric or perhaps your development set or test set.

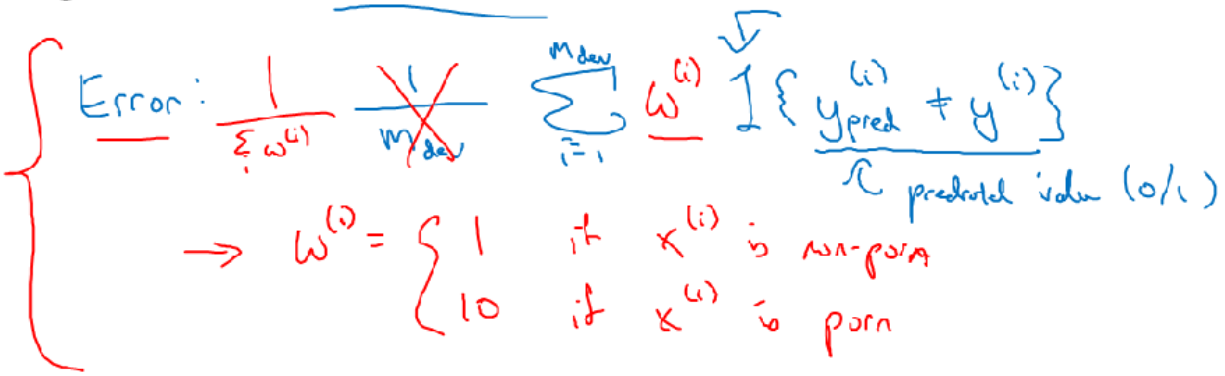
Cat dataset examples



→ Metric: classification error

Algorithm A: 3% error → pornographic

✓ Algorithm B: 5% error



The problem with this evaluation metric is that they treat pornographic and non-pornographic images equally but you really want your classifier to not mislabel pornographic images, like maybe you recognize a pornographic image in cat image and therefore show it to unsuspecting user, therefore very unhappy with unexpectedly seeing porn. One way to change this evaluation metric would be if you add the weight term.

Here, we call this $w^{(i)}$ where $w^{(i)}$ is going to be equal to 1 if $x^{(i)}$ is non-porn and maybe 10 or maybe even large number like a 100 if $x^{(i)}$ is porn. So this way you're giving a much larger weight to examples that are pornographic so that the error term goes up much more if the algorithm makes a mistake on classifying a pornographic image as a cat image. In this example you giving 10 times bigger weights to classify pornographic images correctly. If you want this normalization constant, technically this becomes sum over i of $w^{(i)}$, so then this error would still be between zero and one.

Take away is, if you find that evaluation metric is not giving the correct rank order preference for what is actually better algorithm, then there's a time to think about defining a new evaluation metric. And this is just one possible way that you could define an evaluation metric. The goal of the **evaluation metric is accurately tell you, given two classifiers, which one is better for your application**.

One thing you might notice is that so far we've only talked about how to define a metric to evaluate classifiers. That is, we've defined an evaluation metric that helps us better rank order classifiers when they are performing at varying levels in terms of streaming of porn. And this is actually an example of an **orthogonalization** where I think you should take a machine learning problem and break it into distinct steps.

The overall guideline is if your current metric and data you are evaluating on doesn't correspond to doing well on what you actually care about, then change your metrics and/or your dev/test set to better capture what you need your algorithm to actually do well on. Having an evaluation metric and the dev set allows you to much more quickly make decisions about is Algorithm A or Algorithm B better. It really speeds up how quickly you and your team can iterate. So my recommendation is, even if you can't define the perfect evaluation metric and dev set, just set something up quickly and use that to drive the speed of your team iterating. Here is the summary diagram:

When to change development/test sets and metrics

Example: Cat vs Non-cat

A cat classifier tries to find a great amount of cat images to show to cat loving users. The evaluation metric used is a classification error.

Algorithm	Classification error [%]
A	3%
B	5%

It seems that Algorithm A is better than Algorithm B since there is only a 3% error, however for some reason, Algorithm A is letting through a lot of the pornographic images.

Algorithm B has 5% error thus it classifies fewer images but it doesn't have pornographic images. From a company's point of view, as well as from a user acceptance point of view, Algorithm B is actually a better algorithm. The evaluation metric fails to correctly rank order preferences between algorithms. The evaluation metric or the development set or test set should be changed.

The misclassification error metric can be written as a function as follow:

$$Error: \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} \mathcal{L}\{(\hat{y}^{(i)} \neq y^{(i)}\}$$

This function counts up the number of misclassified examples.

The problem with this evaluation metric is that it treats pornographic vs non-pornographic images equally. On way to change this evaluation metric is to add the weight term $w^{(i)}$.

$$w^{(i)} = \begin{cases} 1 & if\ x^{(i)}\ is\ non-pornographic \\ 10 & if\ x^{(i)}\ is\ pornographic \end{cases}$$

The function becomes:

$$Error: \frac{1}{\sum w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathcal{L}\{(\hat{y}^{(i)} \neq y^{(i)}\}$$

Guideline

1. Define correctly an evaluation metric that helps better rank order classifiers
2. Optimize the evaluation metric

Comparing to human level performance

Why human-level performance?

In the last few years, a lot more machine learning teams have been talking about comparing the machine learning systems to human level performance. Why is this? I think there are two main reasons. First is that because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance. Second, it turns out that the workflow of designing and building a machine learning system, the workflow is much more efficient when you're trying to do something that humans can also do. So in those settings, it becomes natural to talk about comparing, or trying to mimic human-level performance. Let's see a couple examples of what this means. I've seen on a lot of machine learning tasks that as you work on a problem over time, so the x-axis, time, this could be many months or even many years over which some team or some research community is working on a problem. Progress tends to be relatively rapid as you approach human level performance. But then after a while, the algorithm surpasses human-level performance and then progress and accuracy actually slows down. And maybe it keeps getting better but after surpassing human level performance it can still get better, but performance, the slope

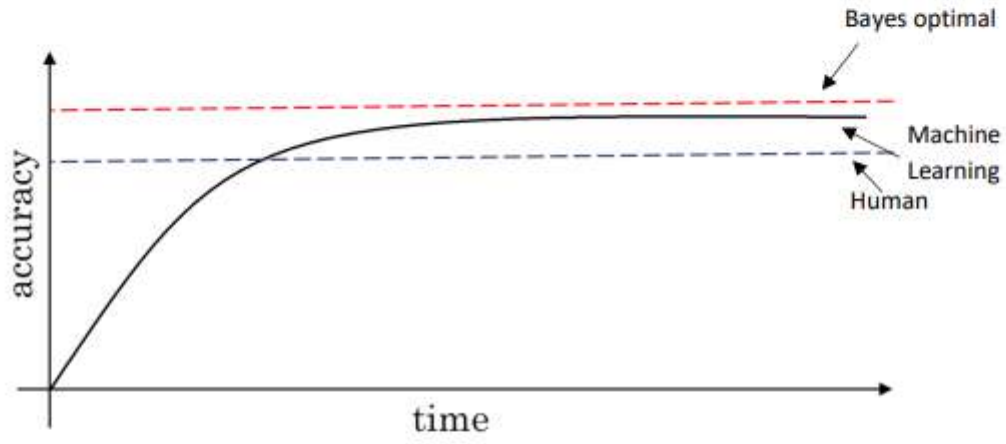
of how rapid the accuracy's going up, often that slows down. And the hope is it achieves some theoretical optimum level of performance.
and over time, as you keep training the algorithm, maybe bigger and bigger models on more and more data, the performance approaches but never surpasses some theoretical limit, which is called the **Bayes optimal erro**r. So **Bayes optimal error, think of this as the best possible error**.

Why human-level performance?

Today, machine learning algorithms can compete with human-level performance since they are more productive and more feasible in a lot of application. Also, the workflow of designing and building a machine learning system, is much more efficient than before.

Moreover, some of the tasks that humans do are close to ''perfection'', which is why machine learning tries to mimic human-level performance.

The graph below shows the performance of humans and machine learning over time.



The

Machine learning progresses slowly when it surpasses human-level performance. One of the reason is that human-level performance can be close to Bayes optimal error, especially for natural perception problem.

Bayes optimal error is defined as the best possible error. In other words, it means that any functions mapping from x to y can't surpass a certain level of accuracy.

Also, when the performance of machine learning is worse than the performance of humans, you can improve it with different tools. They are harder to use once its surpasses human-level performance.

These tools are:

- Get labeled data from humans
- Gain insight from manual error analysis: Why did a person get this right?
- Better analysis of bias/variance.

and that's just the way for any function mapping from x to y to surpass a certain level of accuracy.

So for example, for speech recognition, if x is audio clips, some audio is just so noisy it is impossible to tell what is in the correct transcription. So the perfect error may not be 100%. Or for cat recognition. Maybe some images are so blurry, that it is just impossible for anyone or anything to tell whether or not there's a cat in that picture. So, the perfect level of accuracy may not be 100%. And Bayes optimal error, or Bayesian optimal error, or sometimes Bayes error for short, is the very best theoretical function for mapping from x to y. That can never be surpassed.

So it should be no surprise that this purple line, no matter how many years you work on a problem you can never surpass Bayes error, Bayes optimal error. And it turns out that progress is often quite fast until you surpass human level performance and it sometimes slows down after you surpass human level performance. And I think there are two reasons for that, for why progress often slows down when you surpass human level performance. One reason is that human level performance is for many tasks not that far from Bayes' optimal error. People are very good at looking at images and telling if there's a cat or listening to audio and transcribing

it. So, by the time you surpass human level performance maybe there's not that much head room to still improve.
But the second reason is that so long as your performance is worse than human level performance, then there are actually certain tools you could use to improve performance that are harder to use once you've surpassed human level performance. So here's what I mean. For tasks that humans are quite good at, and this includes looking at pictures and recognizing things, or listening to audio, or reading language, really natural data tasks humans tend to be very good at. For tasks that humans are good at, so long as your machine learning algorithm is still worse than the human, you can get labeled data from humans. That is you can ask people, ask higher humans, to label examples for you so that you can have more data to feed your learning algorithm. Something we'll talk about next week is manual error analysis. But so long as humans are still performing better than any other algorithm, you can ask people to look at examples that your algorithm's getting wrong, and try to gain insight in terms of why a person got it right but the algorithm got it wrong.

Avoidable bias

We talked about how you want your learning algorithm to do well on the training set but sometimes you don't actually want to do too well and knowing what human level performance is, can tell you exactly how well but not too well you want your algorithm to do on the training set. Let me show you what I mean. We have used Cat classification a lot and given a picture, let's say humans have near-perfect accuracy so the human level error is one percent. In that case, if your learning algorithm achieves 8 percent training error and 10 percent dev error, then maybe you wanted to do better on the training set. So the fact that there's a huge gap between how well your algorithm does on your training set versus how humans do shows that your algorithm isn't even fitting the training set well.

So in terms of tools to reduce bias or variance, in this case I would say focus on reducing bias. So you want to do things like train a bigger neural network or run training set longer, just try to do better on the training set. But now let's look at the same training error and dev error and imagine that human level performance was not 1%. So this copy is over but you know in a different application or maybe on a different data set, let's say that human level error is actually 7.5%. Maybe the images in your data set are so blurry that even humans can't tell whether there's a cat in this picture. This example is maybe slightly contrived because humans are actually very good at looking at pictures and telling if there's a cat in it or not. But for the sake of this example, let's say your data sets images are so blurry or so low resolution that even humans get 7.5% error. In this case, even though your training error and dev error are the same as the other example, you see that maybe you're actually doing just fine on the training set. It's doing only a little bit worse than human level performance. And in this second example, you would maybe want to focus on reducing this component, reducing the variance in your learning algorithm. So you might try regularization to try to bring your dev error closer to your training error for example. So in the earlier courses discussion on bias and variance, we were mainly assuming that there were tasks where Bayes error is nearly zero. So to explain what just happened here, for our Cat classification example, think of human level error as a proxy or as a estimate for Bayes error or for Bayes optimal error. And for computer vision tasks, this is a pretty reasonable proxy because humans are actually very good at computer vision and so whatever a human can do is maybe not too far from Bayes error. By definition, human level error is worse than Bayes error because nothing could be better than Bayes error but human level error might not be too far from Bayes error. So the surprising thing we saw here is that depending on what human level error is or really this is really approximately Bayes error or so we assume it to be, but depending on what we think is achievable, with the same training error and dev error in these two cases, we decided to focus on bias reduction tactics or on variance reduction tactics. And what happened is in the example on the left, 8% training error is really high when you think you could get it down to 1% and so bias reduction tactics could help you do that. Whereas in the example on the right, if you think that Bayes error is 7.5% and here we're using human level error as an estimate or as a proxy for Bayes error, but you think that Bayes error is close to seven point five percent then you know there's not that much headroom for reducing your training error further down. You don't really want it to be that much better than 7.5% because you could achieve that only by maybe starting to offer further training so, and instead, there's much more room for improvement in terms of taking this 2% gap and trying to reduce that by using variance reduction techniques such as regularization or maybe getting more training data. So to give these things a couple of names, this is not widely used terminology but I found this useful terminology and a useful way of thinking about it, which is I'm going to call the difference between Bayes error or approximation of Bayes error and the training error to be the avoidable bias. So what you want is maybe keep improving your training performance until you get down to Bayes error but you don't actually want to do better than Bayes error. You can't actually do better than Bayes error unless you're overfitting. And this, the difference between your training area and the dev error, there's a measure still of the variance problem of your algorithm. And the term avoidable bias acknowledges that there's some bias or some minimum level of error that you just cannot get below which is that if Bayes error is 7.5%, you don't actually want to get below that level of error. So rather than saying that

if you're training error is 8%, then the 8% is a measure of bias in this example, you're saying that the avoidable bias is maybe 0.5% or 0.5% is a measure of the avoidable bias whereas 2% is a measure of the variance and so there's much more room in reducing this 2% than in reducing this 0.5%. Whereas in contrast in the example on the left, this 7% is a measure of the avoidable bias, whereas 2% is a measure of how much variance you have. And so in this example on the left, there's much more potential in focusing on reducing that avoidable bias. So in this example, understanding human level error, understanding your estimate of Bayes error really causes you in different scenarios to focus on different tactics, whether bias avoidance tactics or variance avoidance tactics. There's quite a lot more nuance in how you factor in human level performance into how you make decisions in choosing what to focus on.

Avoidable bias

By knowing what the human-level performance is, it is possible to tell when a training set is performing well or not.

Example: Cat vs Non-Cat

	Classification error (%)	
	Scenario A	Scenario B
Humans	1	7.5
Training error	8	8
Development error	10	10

In this case, the human level error as a proxy for Bayes error since humans are good to identify images. If you want to improve the performance of the training set but you can't do better than the Bayes error otherwise the training set is overfitting. By knowing the Bayes error, it is easier to focus on whether bias or variance avoidance tactics will improve the performance of the model.

Scenario A

There is a 7% gap between the performance of the training set and the human level error. It means that the algorithm isn't fitting well with the training set since the target is around 1%. To resolve the issue, we use bias reduction technique such as training a bigger neural network or running the training set longer.

Scenario B

The training set is doing good since there is only a 0.5% difference with the human level error. The difference between the training set and the human level error is called avoidable bias. The focus here is to reduce the variance since the difference between the training error and the development error is 2%. To resolve the issue, we use variance reduction technique such as regularization or have a bigger training set.

Understanding human-level performance

The term human-level performance is sometimes used casually in research articles. But let me show you how we can define it a bit more precisely. And in particular, use the definition of the phrase, human-level performance, that is most useful for helping you drive progress in your machine learning project. So remember from our last section that one of the uses of this phrase, human-level error, is that it gives us a way of estimating Bayes error. What is the best possible error any function could, either now or in the future, ever, ever achieve? So bearing that in mind, let's look at a medical image classification example. Let's say that you want to look at a radiology image like this, and make a diagnosis classification decision and suppose that a typical human,

untrained human, achieves 3% error on this task. A typical doctor, maybe a typical radiologist doctor, achieves 1% error. An experienced doctor does even better, 0.7% error. And a team of experienced doctors, that is if you get a team of experienced doctors and have them all look at the image and discuss and debate the image, together their consensus opinion achieves 0.5% error. So the question I want to pose to you is, how should you define human-level error? Is human-level error 3%, 1%, 0.7% or 0.5%?

Human-level error as a proxy for Bayes error

Medical image classification example:

Suppose:

(a) Typical human ..... 3 % error

(b) Typical doctor ..... 1 % error

(c) Experienced doctor ..... 0.7 % error

(d) Team of experienced doctors .. 0.5 % error

Bayes error $\leq$ 0.5%

What is "human-level" error?

One of the most useful ways to think of human error is as a proxy or an estimate for Bayes error. Which is if you want a proxy or an estimate for Bayes error, then given that a team of experienced doctors discussing and debating can achieve 0.5% error, we know that Bayes error is less than equal to 0.5%. So because some system, team of these doctors can achieve 0.5% error, so by definition, this directly, optimal error has got to be 0.5% or lower. We don't know how much better it is, maybe there's a even larger team of even more experienced doctors who could do even better, so maybe it's even a little bit better than 0.5%. But we know the optimal error cannot be higher than 0.5%. So what I would do in this setting is use 0.5% as our estimate for Bayes error. So I would define human-level performance as 0.5%. Now, for the purpose of publishing a research paper or for the purpose of deploying a system, maybe there's a different definition of human-level error that you can use which is so long as you surpass the performance of a typical doctor. That seems like maybe a very useful result if accomplished, and maybe surpassing a single radiologist, a single doctor's performance might mean the system is good enough to deploy in some context. So maybe the takeaway from this is to be clear about what your purpose is in defining the term human-level error. And if it is to show that you can surpass a single human and therefore argue for deploying your system in some context, maybe this is the appropriate definition. But if your goal is the proxy for Bayes error, then this is the appropriate definition. Let's look at an error analysis example.

Understanding human-level performance

Human-level error gives an estimate of Bayes error.

Example 1: Medical image classification

This is an example of a medical image classification in which the input is a radiology image and the output is a diagnosis classification decision.

	Classification error (%)
Typical human	3.0
Typical doctor	1.0
Experienced doctor	0.7
Team of experienced doctors	0.5

The definition of human-level error depends on the purpose of the analysis, in this case, by definition the Bayes error is lower or equal to 0.5%.

Example 2: Error analysis

	Classification error (%)		
	Scenario A	Scenario B	Scenario C
Human (proxy for Bayes error)	1	1	0.5
	0.7	0.7	
	0.5	0.5	
Training error	5	1	0.7
Development error	6	5	0.8

Scenario A

In this case, the choice of human-level performance doesn't have an impact. The avoidable bias is between 4%-4.5% and the variance is 1%. Therefore, the focus should be on bias reduction technique.

Scenario B

In this case, the choice of human-level performance doesn't have an impact. The avoidable bias is between 0%-0.5% and the variance is 4%. Therefore, the focus should be on variance reduction technique.

Scenario C

In this case, the estimate for Bayes error has to be 0.5% since you can't go lower than the human-level performance otherwise the training set is overfitting. Also, the avoidable bias is 0.2% and the variance is 0.1%. Therefore, the focus should be on bias reduction technique.

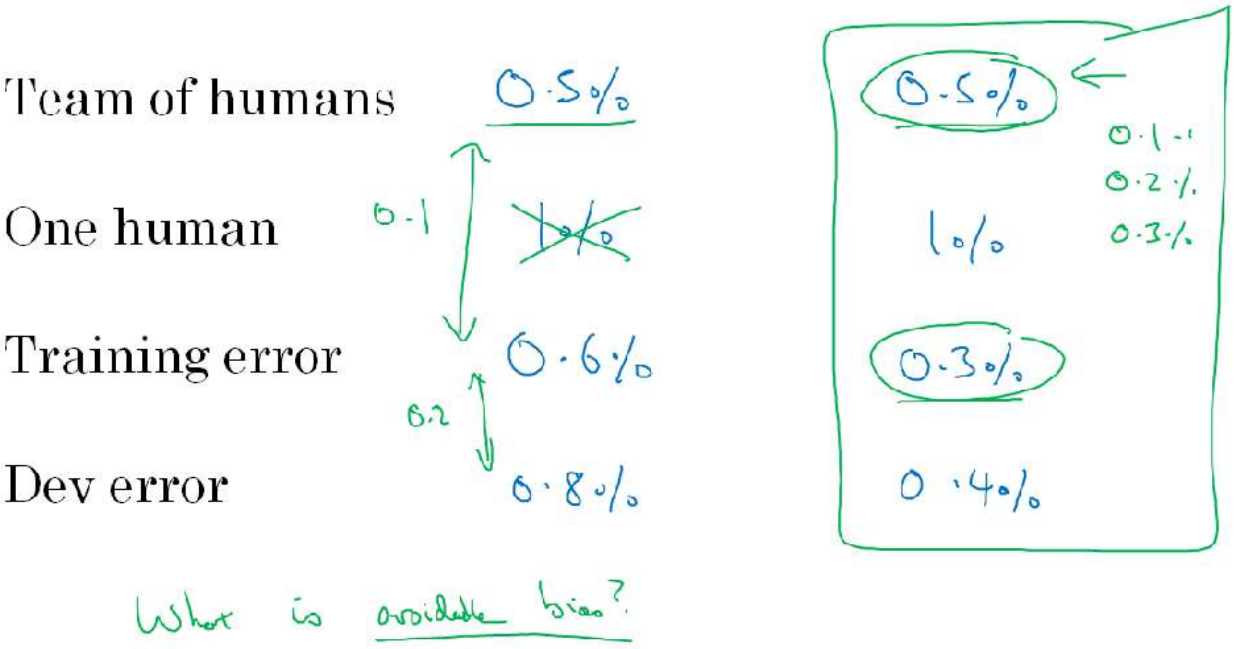
Summary of bias/variance with human-level performance

- Human - level error – proxy for Bayes error
- If the difference between human-level error and the training error is bigger than the difference between the training error and the development error. The focus should be on bias reduction technique
- If the difference between training error and the development error is bigger than the difference between the human-level error and the training error. The focus should be on variance reduction technique

Surpassing human-level performance

Lots of teams often find it exciting to surpass human-level performance on the specific recreational classification task. Let's talk over some of the things you see if you try to accomplish this yourself. We've discussed before how machine learning progress gets harder as you approach or even surpass human-level performance. Let's talk over one more example of why that's the case. Let's say you have a problem where a team of humans discussing and debating achieves 0.5% error, a single human 1% error, and you have an algorithm of 0.6% training error and 0.8% dev error. So in this case, what is the **avoidable bias**? So this one is relatively easier to answer, 0.5% is your estimate of base error, so your avoidable bias is, you're not going to use this 1% number as reference, you can use this difference, so maybe you estimate your avoidable bias is at least 0.1% and your variance as 0.2%. So there's maybe more to do to reduce your variance than your avoidable bias perhaps. But now let's take a harder example, let's say, a team of humans and single human performance, the same as before, but your algorithm gets 0.3% training error, and 0.4% dev error. Now, what is the avoidable bias? It's now actually much harder to answer that. Is the fact that your training error, 0.3%, does this mean you've over-fitted by 0.2%, or is base error, actually 0.1%, or maybe is base error 0.2%, or maybe base error is 0.3%? You don't really know, but based on the information given in this example, you actually don't have enough information to tell if you should focus on reducing bias or reducing variance in your algorithm. So that slows down the efficiency where you should make progress. Moreover, if your error is already better than even a team of humans looking at and discussing and debating the right label, for an example, then it's just also harder to rely on human intuition to tell your algorithm what are ways that your algorithm could still improve the performance? So in this example, once you've surpassed this 0.5% threshold, your options, your ways of making progress on the machine learning problem are just less clear. It doesn't mean you can't make progress, you might still be able to make significant progress, but some of the tools you have for pointing you in a clear direction just don't work as well.

Surpassing human-level performance



Now, there are many problems where machine learning significantly surpasses human-level performance. For example, I think, online advertising, estimating how likely someone is to click on that. Probably, learning algorithms do that much better today than any human could, or making product recommendations, recommending movies or books to you. I think that web sites today can do that much better than maybe even your closest friends can. All logistics predicting how long will take you to drive from A to B or predicting how long to take a delivery vehicle to drive from A to B, or trying to predict whether someone will repay a loan, and therefore, whether or not you should approve a loan offer. All of these are problems where I think today machine learning far surpasses a single human's performance. Notice something about these four examples. All four of these examples are actually learning from structured data, where you might have a database of what has users clicked on, database of proper support for, databases of how long it takes to get from A to B, database of previous loan applications and their outcomes. And these are not natural perception problems, so these are not computer vision, or speech recognition, or natural language processing task. Humans tend to be very good in natural perception task. So it is possible, but it's just a bit harder for computers to surpass human-level performance on natural perception task.

Problems where ML significantly surpasses human-level performance

- Online advertising
- Product recommendations
- Logistics (predicting transit time)
- Loan approvals

Structured data
Not natural perception
Lots of data

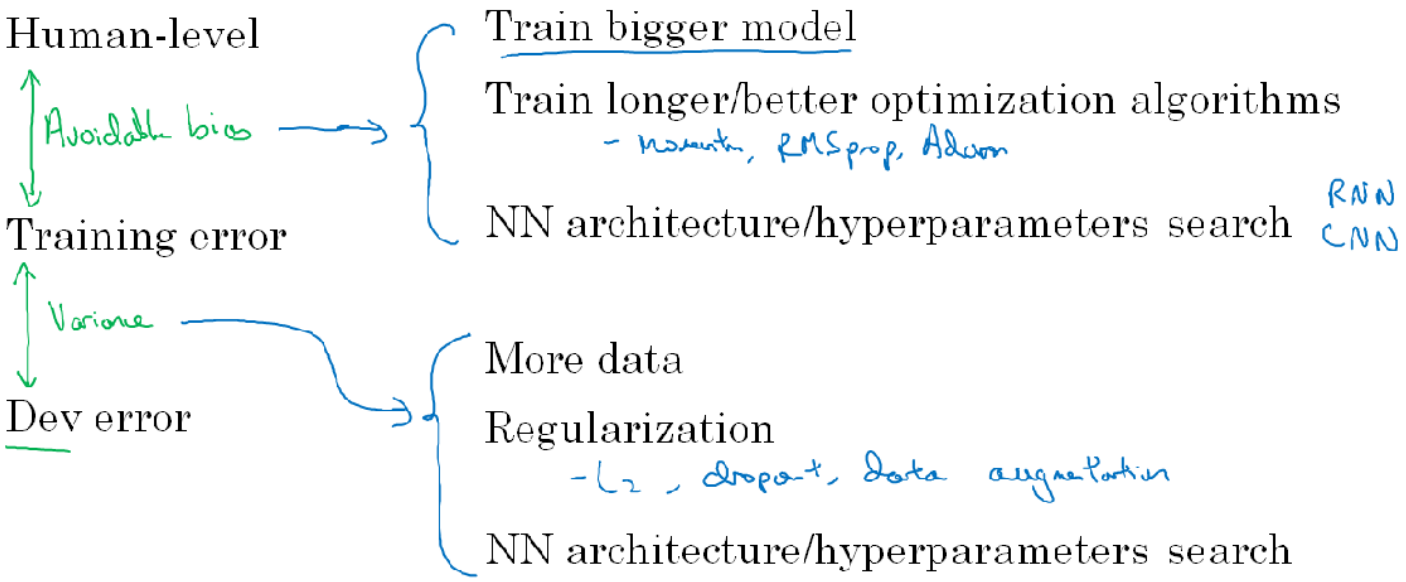
- Speech recognition
- Some image recognition
- Medical
  - ECG, Skin cancer, ...

And finally, all of these are problems where there are teams that have access to huge amounts of data. So for example, the best systems for all four of these applications have probably looked at far more data of that application than any human could possibly look at. And so, that's also made it relatively easy for a computer to surpass human-level performance. Now, the fact that there's so much data that computer could examine, so it can petrifies that's called patterns than even the human mind. Other than these problems, today there are speech recognition systems that can surpass human-level performance. And there are also some computer vision, some image recognition tasks, where computers have surpassed human-level performance. But because humans are very good at this natural perception task, I think it was harder for computers to get there. And then there are some medical tasks, for example, reading ECGs or diagnosing skin cancer, or certain narrow radiology task, where computers are getting really good and maybe surpassing a single human-level performance. And I guess one of the exciting things about recent advances in deep learning is that even fo**r these tasks we can now surpass human-level performance in some cases, but it has been a bit harder because humans tend to be very good at this natural perception task**. So surpassing human-level performance is often not easy, but given enough data there've been lots of deep learning systems have surpassed human-level performance on a single supervisory problem. So that makes sense for an application you're working on.

Improving your model performance

In previous sections we have learned about You **orthogonalization**. How to set up your dev and test sets, human level performance as a proxy for Bayes's error and how to estimate your avoidable bias and variance. Let's pull it all together into a set of guidelines for how to improve the performance of your learning algorithm. So, I think getting a supervised learning algorithm to work well means fundamentally hoping or assuming that you can do two things. **First is that you can fit the training set pretty well and you can think of this as roughly saying that you can achieve low avoidable bias**. And the second **thing you're assuming can do well is that doing well in the training set generalizes pretty well to the dev set or the test set and this is sort of saying that variance is not too bad**.

Reducing (avoidable) bias and variance



How much better do you think you should be trying to do on your training set and then look at the difference between your dev error and your training error as an estimate. So, it's how much of a variance problem you have. In other words, how much harder you should be working to make your performance generalize from the training set to the desk set, that it wasn't trained on explicitly?

So to whatever extent you want to try to reduce avoidable bias, I would try to apply tactics like train a bigger model. So, you can just do better on your training sets or train longer. Use a better optimization algorithm such as momentum or RMS prop, Adam. One of the thing you could try is to just find a better new NN architecture or hyperparameters and this could include everything from changing the activation functions or changing the number of layers or hidden do this or you can try other NN models architectures, such as the recurrent neural network and convolution neural networks. See below diagram with summary.

Improving your model performance

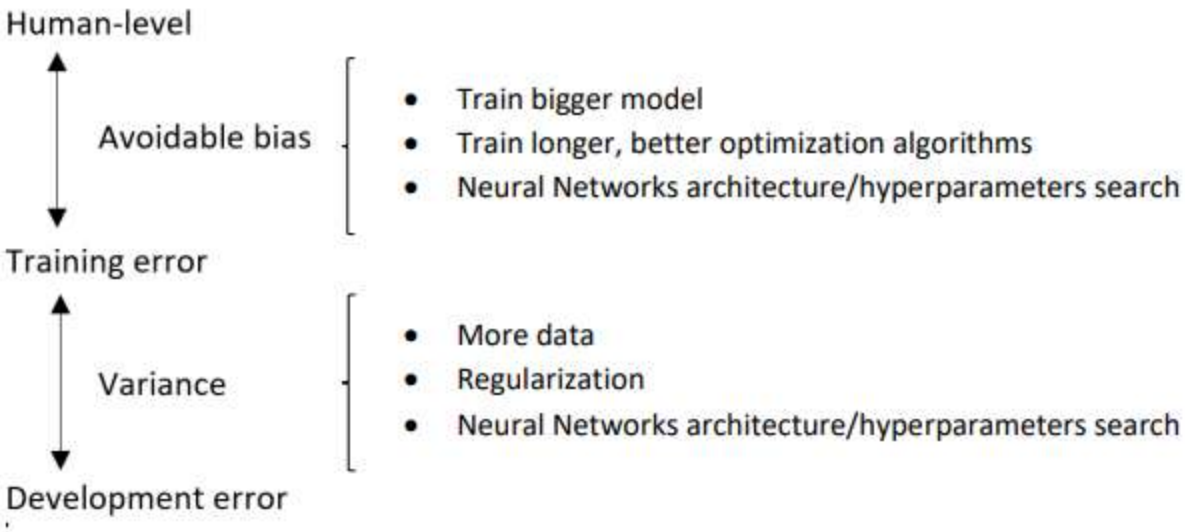
The two fundamental assumptions of supervised learning

There are 2 fundamental assumptions of supervised learning. The first one is to have a low avoidable bias which means that the training set fits well. The second one is to have a low or acceptable variance which means that the training set performance generalizes well to the development set and test set.

If the difference between human-level error and the training error is bigger than the difference between the training error and the development error, the focus should be on bias reduction technique which are training a bigger model, training longer or change the neural networks architecture or try various hyperparameters search.

If the difference between training error and the development error is bigger than the difference between the human-level error and the training error, the focus should be on variance reduction technique which are bigger data set, regularization or change the neural networks architecture or try various hyperparameters search.

Summary



Week 2: ML Strategy (2)

Learning Objectives

- Understand what multi-task learning and transfer learning are
- Recognize bias, variance and data-mismatch by looking at the performances of your algorithm on train/dev/test sets

Error Analysis

Carrying out error analysis

If you're trying to get a learning algorithm to do a task that humans can do and if your learning algorithm is not yet at the performance of a human. Then manually examining mistakes that your

algorithm is making, can give you insights into what to do next. This process is called **error analysis**. Let's start with an example. Let's say you're working on your cat classifier, and you've achieved 90% accuracy, or equivalently 10% error, on your dev set. And let's say this is much worse than you're hoping to do. Maybe one of your teammates looks at some of the examples that the algorithm is misclassifying, and notices that it is miscategorizing some dogs as cats. And if you look at these two dogs, maybe they look a little bit like a cat, at least at first glance. So maybe your teammate comes to you with a proposal for how to make the algorithm do better, specifically on dogs, right? You can imagine building a focus effort, maybe to collect more dog pictures, or maybe to design features specific to dogs, or something. In order to make your cat classifier do better on dogs, so it stops misrecognizing these dogs as cats. So the question is, **should you go ahead and start a project focus on the dog problem**?

There could be several months of works you could do in order to make your algorithm make few mistakes on dog pictures. So is that worth your effort? Well, rather than spending a few months doing this, only to risk finding out at the end that it wasn't that helpful. Here's an **error analysis procedure that can let you very quickly tell whether or not this could be worth your effort**. Here's what I recommend you do. First, get about, say 100 mislabeled dev set examples, then examine them manually. Just count them up one at a time, to see how many of these mislabeled examples in your dev set are actually pictures of dogs. Now, suppose that it turns out that 5% of your 100 mislabeled dev set examples are pictures of dogs. So, that is, if 5 out of 100 of these mislabeled dev set examples are dogs, what this means is that of the 100 examples. Of a typical set of 100 examples you're getting wrong, even if you completely solve the dog problem, you only get 5 out of 100 more correct. Or in other words, if only 5% of your errors are dog pictures, then the best you could easily hope to do, if you spend a lot of time on the dog problem. Is that your error might go down from 10% error, down to 9.5% error, right? So this a 5% relative decrease in error, from 10% down to 9.5%. And so you might reasonably decide that this is not the best use of your time. Or maybe it is, but at least this gives you a ceiling, a upper bound on how much you could improve performance by working on the dog problem. In machine learning, sometimes we call this the ceiling on performance. Which just means, what's in the best case? How well could working on the dog problem help you?

Look at dev examples to evaluate ideas

90% accuracy
→ 10% error

Should you try to make your cat classifier do better on dogs?

Error analysis: → 5-10 min

- Get ~100 mislabeled dev set examples.
- Count up how many are dogs.

"ceiling"

→ 5% → 5/100

10% ↓ 9.5%

→ 50% 50/100

10% ↓ 5%

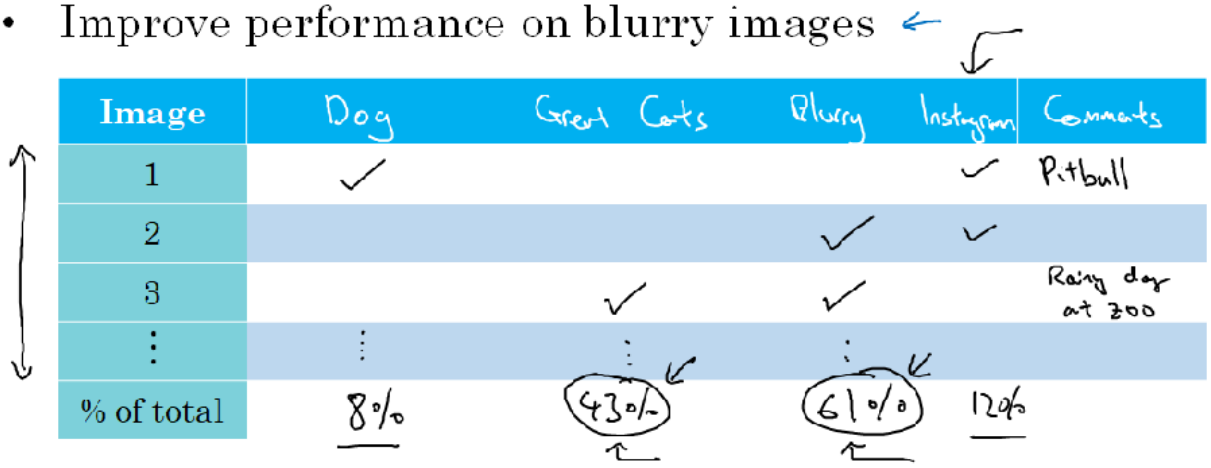
But now, suppose something else happens. Suppose that we look at your 100 mislabeled dev set examples, you find that 50 of them are actually dog images. So 50% of them are dog pictures. Now you could be much more optimistic about spending time on the dog problem. In this case, if you actually solve the dog problem, your error would go down from this 10%, down to potentially 5% error. And you might decide that halving your error could be worth a lot of effort. Focus on reducing the problem of mislabeled dogs. I know that in machine learning, sometimes we speak disparagingly of hand engineering things, or using too much value insight. But if you're building applied systems, then this simple counting procedure, error analysis, can save you a lot of time. In terms of deciding what's the most important, or what's the most promising direction to focus on. In fact, if you're looking at 100 mislabeled dev set examples, maybe this is a 5 to 10 minute effort. To manually go through 100 examples, and manually count up how many of them are dogs. And depending on the outcome, whether there's more like 5%, or 50%, or something else. This, in just 5 to 10 minutes, gives you an estimate of how worthwhile this direction is and could help you make a much better decision, whether or not to spend the next few months focused on trying to find solutions to solve the problem of mislabeled dogs. In this section, we'll describe using error analysis to evaluate whether or not a single idea, dogs in this case, is worth working on. Sometimes you can also evaluate multiple ideas in parallel doing error analysis.

For example, let's say you have several ideas in improving your cat detector. Maybe you can improve performance on dogs? Or maybe you notice that sometimes, what are called great cats, such as lions, panthers, cheetahs, and so on. That they are being recognized as small cats, or house cats. So you could maybe find a way to work on that. Or maybe you find that some of your images are blurry, and it would be nice if you could design something that just works better on blurry images and maybe you have some ideas on how to do that. So if carrying out error analysis to evaluate these three ideas, what I would do is create a table like this and I usually do this in a spreadsheet, but using an ordinary text file will also be okay and on the left side, this goes through the set of images you plan to look at manually. So this maybe goes from 1 to 100, if you look at 100 pictures. And the columns of this table, of the spreadsheet, will correspond to the ideas you're evaluating. So the dog problem, the problem of great cats, and blurry images. And I usually also leave space in the spreadsheet to write comments. So remember, during error analysis, you're just looking at dev set examples that your algorithm has misrecognized. So if you find that the first misrecognized image is a picture of a dog, then I'd put a check mark there. And to help myself remember these images, sometimes I'll make a note in the comments. So maybe that was a pit bull picture. If the second picture was blurry, then make a note there. If the third one was a lion, on a rainy day, in the zoo that was misrecognized. Then that's a great cat, and the blurry data. Make a note in the comment section, rainy day at zoo, and it was the rain that made it blurry, and so on. Then finally, having gone through some set of images, I would count up what percentage of these algorithms. Or what percentage of each of these error categories were attributed to the dog, or great cat, blurry categories. So maybe 8% of these images you examine turn out be dogs, and maybe 43% great cats, and 61% were blurry. So this just means going down each column, and counting up what percentage of images have a check mark in that column. As you're part way through this process, sometimes you notice other categories of mistakes. So, for example, you might find that Instagram style filter, those fancy image filters, are also messing up your classifier. In that case, it's actually okay, part way through the process, to add another column like that. For the multi-colored filters, the Instagram filters, and the Snapchat filters. And then go through and count up those as well, and figure out what percentage comes from that new error category.

Evaluate multiple ideas in parallel

Ideas for cat detection:

- Fix pictures of dogs being recognized as cats
- Fix great cats (lions, panthers, etc..) being misrecognized
- Improve performance on blurry images



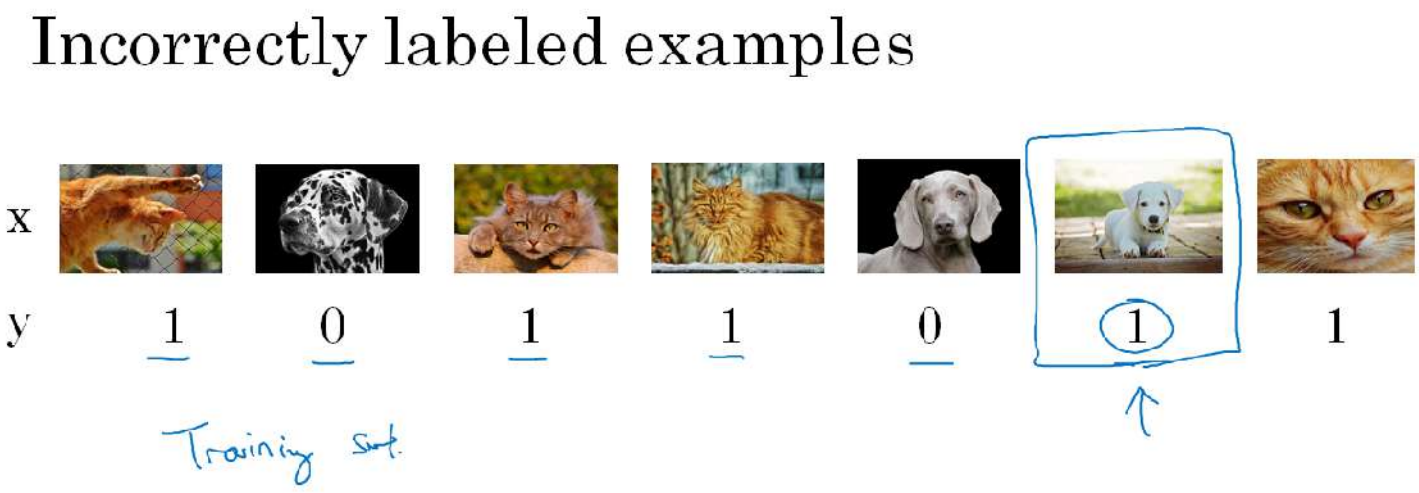
The conclusion of this process gives you an estimate of how worthwhile it might be to work on each of these different categories of errors. For example, clearly in the example (see diagram above), a lot of the mistakes we made on blurry images, and quite a lot on were made on great cat images. And so the outcome of this analysis is not that you must work on blurry images. This doesn't give you a rigid mathematical formula that tells you what to do, but it gives you a sense of the best options to pursue. It also tells you, for example, that no matter how much better you do on dog images, or on Instagram images. You at most improve performance by maybe 8%, or 12%, in these examples. Whereas you can to better on great cat images, or blurry images, the potential improvement. Now there's a ceiling in terms of how much you could improve performance, is much higher. So depending on how many ideas you have for improving performance on great cats, on blurry images. Maybe you could pick one of the two, or if you have enough personnel on your team, maybe you can have two different teams. Have one work on improving errors on great cats, and a different team work on improving errors on blurry images. But this quick counting procedure, which you can often do in, at most, small numbers of hours. Can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.

So to summarize, to carry out error analysis, you should find a set of mislabeled examples, either in your dev set, or in your development set. And look at the mislabeled examples for false positives and false negatives. And just count up the number of errors that fall into various different categories. During this process, you might be inspired to generate new categories of errors, like we saw. If you're looking through the examples and you say gee, there are a lot of Instagram filters, or Snapchat filters, they're also messing up my classifier. You can create new categories during that process. But by counting up the fraction of examples that are mislabeled in different ways, often this will help you prioritize. Or give you inspiration for new directions to go in. Now as you're doing error analysis, sometimes you notice that some of your examples in your dev sets are mislabeled, we'll discuss that in next section.

Cleaning up incorrectly labeled data

The data for your supervised learning problem comprises input X and output labels Y. What if you going through your data and you find that some of these output labels Y are incorrect, you have

data which is incorrectly labeled? Is it worth your while to go in to fix up some of these labels? Let's take a look. In the cat classification problem, Y equals one for cats and zero for non cats.



So if you find that your data has some incorrectly labeled examples, what should you do? Well, first, let's consider the training set. It turns out that deep learning algorithms are quite robust to random errors in the training set. So as long as your errors or your incorrectly labeled examples, so as long as those errors are not too far from random, maybe sometimes the labeler just wasn't paying attention or they accidentally, randomly hit the wrong key on the keyboard. **If the errors are reasonably random, then it's probably okay to just leave the errors as they are and not spend too much time fixing them**. There's certainly no harm to going into your training set and be examining the labels and fixing them. Sometimes that is worth doing but your effort might be okay even if you don't. So as long as the total data set size is big enough and the actual percentage of errors is maybe not too high. So I see a lot of machine learning algorithms that trained even when we know that there are few X mistakes in the training set labels and usually works okay. There is one caveat to this which is that deep learning algorithms are robust to random errors. They are less robust to systematic errors. So for example, if your labeler consistently labels white dogs as cats, then that is a problem because your classifier will learn to classify all white colored dogs as cats. But random errors or near random errors are usually not too bad for most deep learning algorithms. Now, this discussion has focused on what to do about incorrectly labeled examples in your training set. How about incorrectly labeled examples in your dev set or test set? If you're worried about the impact of incorrectly labeled examples on your dev set or test set, what they recommend you do is during error analysis to add one extra column so that you can also count up the number of examples where the label Y was incorrect. So for example, maybe when you count up the impact on a 100 mislabeled dev set examples, so you're going to find a 100 examples where your classifier's output disagrees with the label in your dev set. And sometimes for a few of those examples, your classifier disagrees with the label because the label was wrong, rather than because your classifier was wrong.

Error analysis

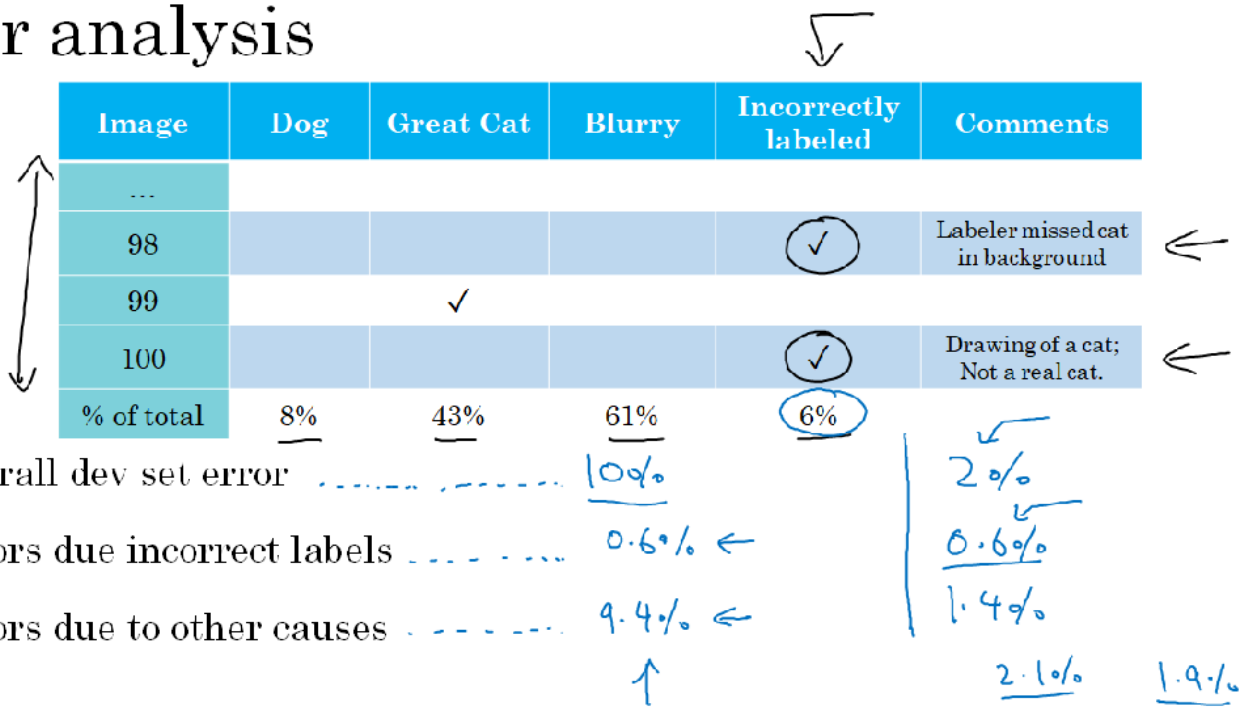


Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

So maybe in this example, you find that the labeler missed a cat in the background. So put the check mark there to signify that example 98 had an incorrect label. And maybe for this one, the picture is actually a picture of a drawing of a cat rather than a real cat. Maybe you want the labeler to have labeled that Y equals zero rather than Y equals one. And so put another check mark there. And just as you count up the percent of errors due to other categories like we saw in the previous video, you'd also count up the fraction of percentage of errors due to incorrect labels. Where the Y value in your dev set was wrong and that accounted for why your learning algorithm made a prediction that differed from what the label on your data says. So the question now is, is it worthwhile going in to try to fix up this 6% of incorrectly labeled examples. My advice is, if it makes a significant difference to your **ability to evaluate algorithms** on your dev set, then go ahead and spend the time to fix incorrect labels. But if it doesn't make a significant difference to your ability to use the dev set to evaluate cost buyers, then it might not be the best use of your time.

Correcting incorrect dev/test set examples

- Apply same process to your dev and test sets to make sure they continue to come from the same distribution
- Consider examining examples your algorithm got right as well as ones it got wrong.
- Train and dev/test data may now come from slightly different distributions.

Let me show you an example that illustrates what I mean by this. So, three numbers I recommend you look at to try to decide if it's worth going in and reducing the number of mislabeled examples are the following. I recommend you look at the overall dev set error. And so in the example we had from the previous video, we said that maybe our system has 90% overall accuracy. So 10% error. Then you should look at the number of errors or the percentage of errors that are due to incorrect labels. So it looks like in this case, 6% of the errors are due to incorrect labels. So 6% of 10% is 0.6%. And then you should look at errors due to all other causes. So if you made 10% error on your dev set and 0.6% of those are because the labels is wrong, then the remainder, 9.4% of them, are due to other causes such as misrecognizing dogs being cats, great cats and their images. So in this case, I would say there's 9.4% worth of error that you could focus on fixing, whereas the errors due to incorrect labels is a relatively small fraction of the overall set of errors. So by all means, go in and fix these incorrect labels if you want but it's maybe not the most important thing to do right now. Now, let's take another example. Suppose you've made a lot more progress on your learning problem. So instead of 10% error, let's say you brought the errors down to 2%, but still 0.6% of your overall errors are due to incorrect labels. So now, if you want to examine a set of mislabeled dev set images, set that comes from just 2% of dev set data you're mislabeling, then a very large fraction of them, 0.6 divided by 2%, so that is actually 30% rather than 6% of your labels. Your incorrect examples are actually due to incorrectly label examples. And so errors due to other causes are now 1.4%. When such a high fraction of your mistakes as measured on your dev set due to incorrect labels, then it maybe seems much more worthwhile to fix up the incorrect labels in your dev set. And if you remember the goal of the dev set, the main purpose of the dev set is, you want to really use it to help you select between two classifiers A and B. So you're trying out two classifiers A and B, and one has 2.1% error and the other has 1.9% error on your dev set. But you don't trust your dev set anymore to be correctly telling you whether this classifier is actually better than this because your 0.6% of these mistakes are due to incorrect labels. Then there's a good reason to go in and fix the incorrect labels in your dev set. Because in this example on the right is just having a very large impact on the overall assessment of the errors of the algorithm, whereas example on the left, the percentage impact is having on your algorithm is still smaller. Now, if you decide to go into a dev set and manually re-examine the labels and try to fix up some of the labels, here are a few additional guidelines or principles to consider. First, I would encourage you to apply whatever process you apply to both your dev and test sets at the same time. We've talk previously about why you want to dev and test sets to come from the same distribution. The dev set is tagging you into target and when you hit it, you want that to generalize to the test set. So your team really

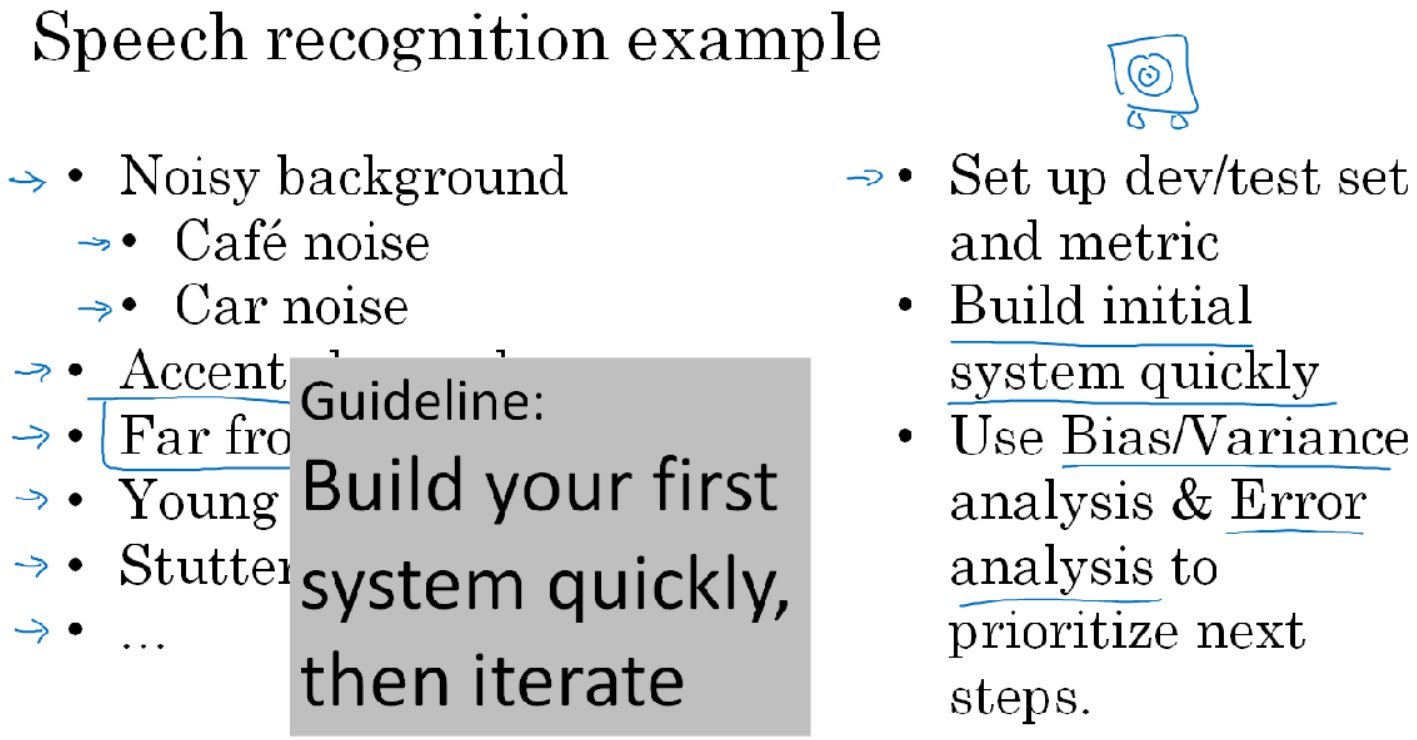
works more efficiently to dev and test sets come from the same distribution. So if you're going in to fix something on the dev set, I would apply the same process to the test set to make sure that they continue to come from the same distribution. So we hire someone to examine the labels more carefully.

Do that for both your dev and test sets. Second, I would urge you to consider examining examples your algorithm got right as well as ones it got wrong. It is easy to look at the examples your algorithm got wrong and just see if any of those need to be fixed. But it's possible that there are some examples that you haven't got right, that should also be fixed. And if you only fix ones that your algorithms got wrong, you end up with more bias estimates of the error of your algorithm. It gives your algorithm a little bit of an unfair advantage. We just try to double check what it got wrong but you don't also double check what it got right because it might have gotten something right, that it was just lucky on fixing the label would cause it to go from being right to being wrong, on that example. The second bullet isn't always easy to do, so it's not always done. The reason it's not always done is because if you classifier's very accurate, then it's getting fewer things wrong than right. So if your classifier has 98% accuracy, then it's getting 2% of things wrong and 98% of things right. So it's much easier to examine and validate the labels on 2% of the data and it takes much longer to validate labels on 98% of the data, so this isn't always done. That's just something to consider.

Finally, if you go into a dev and test data to correct some of the labels there, you may or may not decide to go and apply the same process for the training set. Remember we said that at this other section that it's actually **less important to correct the labels in your training set**. And it's quite possible you decide to just **correct the labels in your dev and test set** which are also often smaller than a training set and you might not invest all that extra effort needed to correct the labels in a much larger training set. This is actually okay. Learning algorithms are quite robust to that. It's super important that your dev and test sets come from the same distribution. But if your training set comes from a slightly different distribution, often that's a pretty reasonable thing to do.So couple of advice, First, deep learning researchers sometimes like to say things like, "I just fed the data to the algorithm. I trained in and it worked." There is a lot of truth to that in the deep learning error. There is more of feeding data in algorithm and just training it and doing less hand engineering and using less human insight. But I think that in building practical systems, often there's also more manual error analysis and more human insight that goes into the systems than sometimes deep learning researchers like to acknowledge. Second is that somehow I've seen some engineers and researchers be reluctant to manually look at the examples. Maybe it's not the most interesting thing to do, to sit down and look at a 100 or a couple hundred examples to counter the number of errors. But this is something that I so do myself. When I'm leading a machine learning team and I want to understand what mistakes it is making, I would actually go in and look at the data myself and try to counter the fraction of errors. And I think that because these minutes or maybe a small number of hours of counting data can really help you prioritize where to go next. I find this a very good use of your time and I urge you to consider doing it if those machines are in your system and you're trying to decide what ideas or what directions to prioritize things.

Build your first system, quickly, then iterate

If you're working on a brand new machine learning application, one of the piece of advice I often give people is that, I think you should build your first system quickly and then iterate. Let me show you what I mean. I've worked on speech recognition for many years. And if you're thinking of building a new speech recognition system, there's actually a lot of directions you could go and a lot of things you could prioritize. For example, there are specific techniques for making speech recognition systems more robust to noisy background.



And noisy background could mean cafe noise, like a lot of people talking in the background or car noise, the sounds of cars and highways or other types of noise. There are ways to make a speech recognition system more robust to accented speech. There are specific problems associated with speakers that are far from the microphone, this is called far-field speech recognition. Young children speech poses special challenges, both in terms of how they pronounce individual words as well as their choice of words and the vocabulary they tend to use. And if sometimes the speaker stutters or if they use nonsensical phrases like oh, ah, um, there are different choices and different techniques for making the transcript that you output, still read more fluently. So, there are these and many other things you could do to improve a speech recognition system. And more generally, for almost any machine learning application, there could be 50 different directions you could go in and each of these directions is reasonable and would make your system better. But the challenge is, how do you pick which of these to focus on. And even though I've worked in speech recognition for many years, if I'm building a new system for a new application domain, I would still find it maybe a little bit difficult to pick without spending some time thinking about the problem. So what we recommend you do, if you're starting on building a brand new machine learning application, **is to build your first system quickly and then iterate**. What I mean by that is I recommend that **you first quickly set up a dev/test set and metric. So this is really deciding where to place your target. And if you get it wrong, you can always move it later, but just set up a target somewhere. And then I recommend you build an initial machine learning system quickly. Find the training set, train it and see**. Start to see and understand how well you're doing against your dev/test set and your values and metric. **When you build your initial system, you then be able to use bias/variance analysis which we talked about earlier as well as error analysis** which we talked about just in the last several sections, to prioritize the next steps. In particular, if error analysis causes you to realize that a lot of the errors are from the speaker being very far from the microphone, which causes special challenges to speech recognition, then that will give you a good reason to focus on techniques to address this called **far-field speech recognition** which basically means handling when the speaker is very far from the microphone. Of all the value of building this initial system, it can be a quick and dirty implementation, you know, don't overthink it, but all the value of the initial system is having some learned system, having some trained system allows you to localize bias/variance, to try to prioritize what to do next, allows you to do error analysis, look at some mistakes, to figure out all the different directions you can go in, which ones are actually the most worthwhile.

So to recap, what I recommend you do is build your first system quickly, then iterate. This advice applies less strongly if you're working on an application area in which you have significant prior experience. It also implies to build less strongly if there's a significant body of academic literature that you can draw on for pretty much the exact same problem you're building. So, for example, there's a large academic literature on face recognition. And if you're trying to build a face recognizer, it might be okay to build a more complex system from the get-go by building on this large body of academic literature. But if you are tackling a new problem for the first time, then I would encourage you to really not overthink or not make your first system too complicated. Well, just build something quick and dirty and then use that to help you prioritize how to improve your system. So I've seen a lot of machine learning projects and I've seen some teams over-think the solution and build something too complicated. I've also seen some teams under-think and then build something maybe too simple. Well on average, I've seen a lot more teams over-think and build something too complicated.

Build system quickly, then iterate

Depending on the area of application, the guideline below will help you prioritize when you build your system.

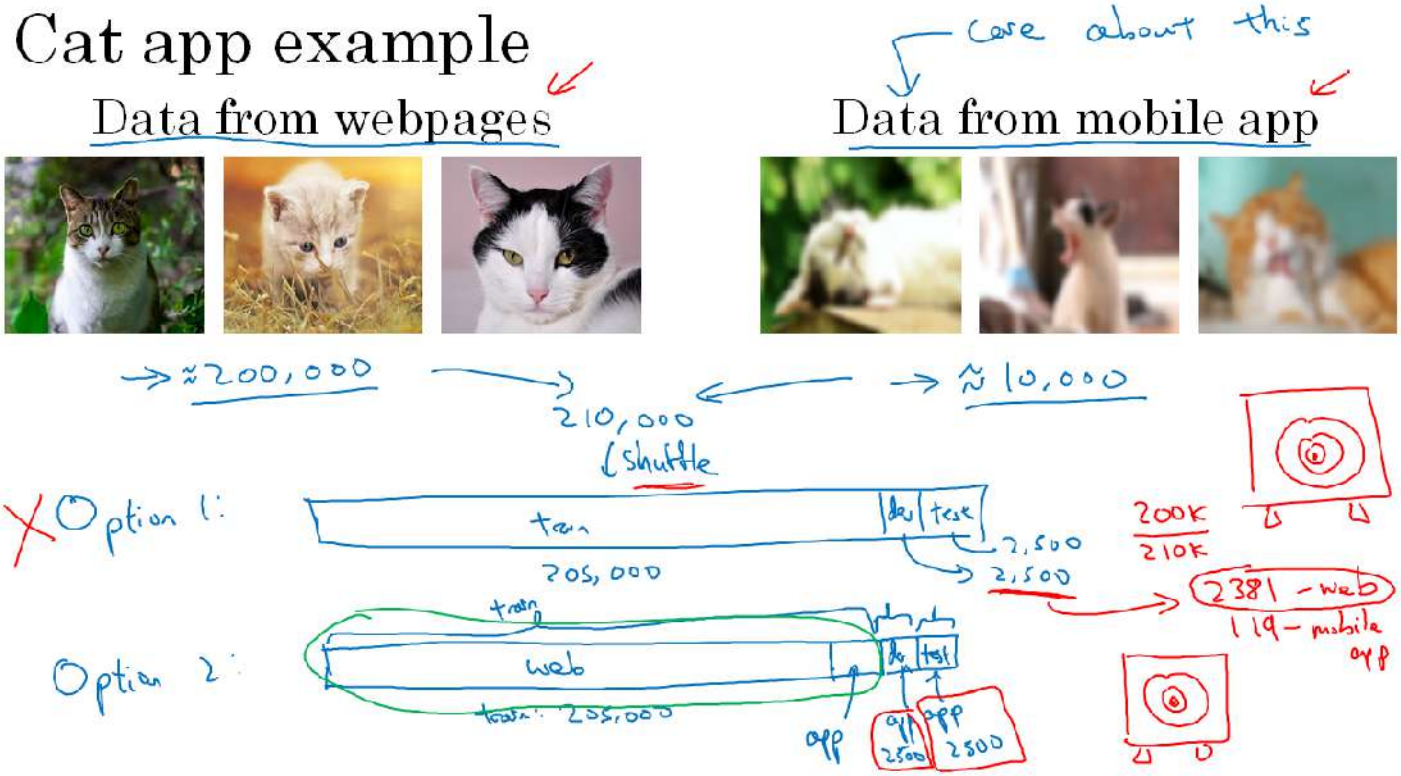
Guideline

1. Set up development/ test set and metrics
   - Set up a target
2. Build an initial system quickly
   - Train training set quickly: Fit the parameters
   - Development set: Tune the parameters
   - Test set: Assess the performance
3. Use Bias/Variance analysis & Error analysis to prioritize next steps

Mismatched training and dev/test set

Training and testing on different distributions

Deep learning algorithms have a huge hunger for training data. They just often work best when you can find enough label training data to put into the training set. This has resulted in many teams sometimes taking whatever data you can find and just shoving it into the training set just to get it more training data. Even if some of this data, or even maybe a lot of this data, doesn't come from the same distribution as your dev and test data. So in a deep learning era, more and more teams are now training on data that comes from a different distribution than your dev and test sets. And there's some subtleties and some best practices for dealing with when you're training and test distributions differ from each other. Let's take a look. Let's say that you're building a mobile app where users will upload pictures taken from their cell phones, and you want to recognize whether the pictures that your users upload from the mobile app is a cat or not. So you can now get two sources of data. One which is the distribution of data you really care about, this data from a mobile app like that on the right, which tends to be less professionally shot, less well framed, maybe even blurrier because it's shot by amateur users. The other source of data you can get is you can crawl the web and just download a lot of, for the sake of this example, let's say you can download a lot of very professionally framed, high resolution, professionally taken images of cats. And let's say you don't have a lot of users yet for your mobile app. So maybe you've gotten 10,000 pictures uploaded from the mobile app. But by crawling the web you can download huge numbers of cat pictures, and maybe you have 200,000 pictures of cats downloaded off the Internet.



So what you really care about is that your final system does well on the mobile app distribution of images, right? Because in the end, your users will be uploading pictures like those on the right and you need your classifier to do well on that. But you now have a bit of a dilemma because you have a relatively small dataset, just 10,000 examples drawn from that distribution. And you have a much bigger dataset that's drawn from a different distribution. There's a different appearance of image than the one you actually want. So you don't want to use just those 10,000 images because it ends up giving you a relatively small training set and using those 200,000 images seems helpful, but the dilemma is this 200,000 images isn't from exactly the distribution you want. So what can you do? Well, here's one option. One thing you can do is put both of these data sets together so you now have 210,000 images. And you can then take the 210,000 images and randomly shuffle them into a train, dev, and test set. And let's say for the sake of argument that you've decided that your dev and test sets will be 2,500 examples each. So your training set will be 205,000 examples.

Now so set up your data this way has some advantages but also disadvantages. The advantage is that now you're training, dev and test sets will all come from the same distribution, so that makes it easier to manage. But the disadvantage, and this is a huge disadvantage, is that if you look at your dev set, of these 2,500 examples, a lot of it will come from the web page distribution of images, rather than what you actually care about, which is the mobile app distribution of images. So it turns out that of your total amount of data, 200,000 or 200k, out of 210,000 or 210k, that comes from web pages. So all of these 2,500 examples on expectation, I think 2,381 of them will come from web pages. This is on expectation, the exact number will vary around depending on how the random shuttle operation went. But on average, only 119 will come from mobile app uploads.

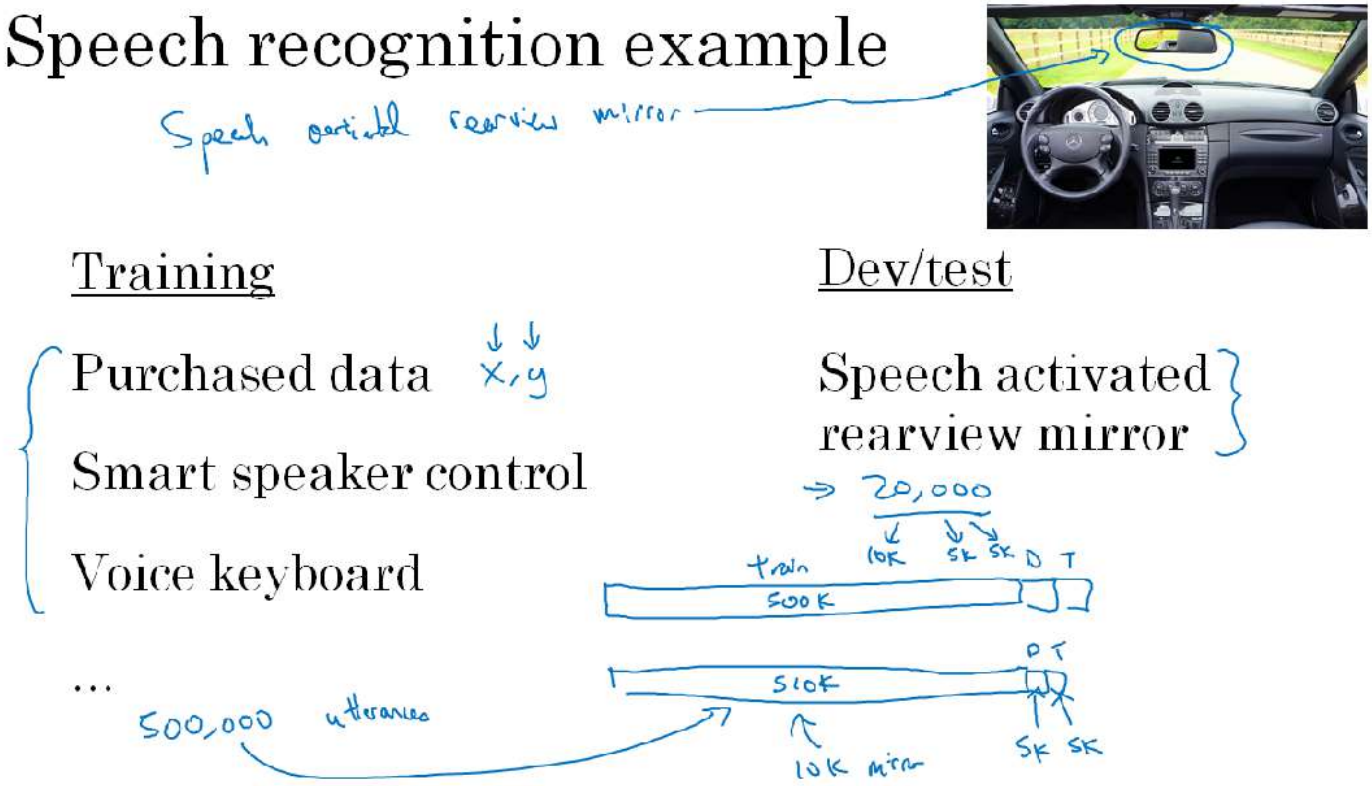
So remember that setting up your dev set is telling your team where to aim the target and the way you're aiming your target, you're saying spend most of the time optimizing for the web page distribution of images, which is really not what you want.

So I would recommend against option one, because this is setting up the dev set to tell your team to optimize for a different distribution of data than what you actually care about.

So instead of doing this, I would recommend that you instead take another option, which is the following. The training set, let's say it's still 205,000 images, I would have the training set have all 200,000 images from the web. And then you can, if you want, add in 5,000 images from the

mobile app. And then for your dev and test sets, I guess my data sets size aren't drawn to scale. Your dev and test sets would be all mobile app images.
So the training set will include 200,000 images from the web and 5,000 from the mobile app. The dev set will be 2,500 images from the mobile app, and the test set will be 2,500 images also from the mobile app. The advantage of this way of splitting up your data into train, dev, and test, is that you're now aiming the target where you want it to be. You're telling your team, my dev set has data uploaded from the mobile app and that's the distribution of images you really care about, so let's try to build a machine learning system that does really well on the mobile app distribution of images. The disadvantage, of course, is that now your training distribution is different from your dev and test set distributions. But it turns out that this split of your data into train, dev and test will get you better performance over the long term. And we'll discuss later some specific techniques for dealing with your training sets coming from different distribution than your dev and test sets.

Let's look at another example. Let's say you're building a brand new product, a speech activated rearview mirror for a car. So this is a real product in China. It's making its way into other countries but you can build a rearview mirror to replace this little thing there, so that you can now talk to the rearview mirror and basically say, dear rearview mirror, please help me find navigational directions to the nearest gas station and it'll deal with it.



So how can you get data to train up a speech recognition system for this product? Well, maybe you've worked on speech recognition for a long time so you have a lot of data from other speech recognition applications, just not from a speech activated rearview mirror. Here's how you could split up your training and your dev and test sets. So for your training, you can take all the speech data you have that you've accumulated from working on other speech problems, such as data you purchased over the years from various speech recognition data vendors. And today you can actually buy data from vendors of x, y pairs, where x is an audio clip and y is a transcript. Or maybe you've worked on smart speakers, smart voice activated speakers, so you have some data from that. Maybe you've worked on voice activated keyboards and so on. And for the sake of argument, maybe you have 500,000 utterences from all of these sources. And for your dev and test set, maybe you have a much smaller data set that actually came from a speech activated rearview mirror because users are asking for navigational queries or trying to find directions to various places. This data set will maybe have a lot more street addresses, right? Please help me

navigate to this street address, or please help me navigate to this gas station. So this distribution of data will be very different than these on the left but this is really the data you care about, because this is what you need your product to do well on, so this is what you set your dev and test set to be. So what you do in this example is set your training set to be the 500,000 utterances on the left, and then your dev and test sets which I'll abbreviate D and T, these could be maybe 10,000 utterances each. That's drawn from actual the speech activated rearview mirror. Or alternatively, if you think you don't need to put all 20,000 examples from your speech activated rearview mirror into the dev and test sets, maybe you can take half of that and put that in the training set. So then the training set could be 510,000 utterances, including all 500 from there and 10,000 from the rearview mirror and then the dev and test sets could maybe be 5,000 utterances each. So of the 20,000 utterances, maybe 10k goes into the training set and 5k into the dev set and 5,000 into the test set. So this would be another reasonable way of splitting your data into train, dev, and test and this gives you a much bigger training set, over 500,000 utterances, than if you were to only use **speech activated rearview mirror data** for your training set. So in this video, you've seen a couple examples of when allowing your training set data to come from a different distribution than your dev and test set allows you to have much more training data and in these examples, it will cause your learning algorithm to perform better. Now one question you might ask is, should you always use all the data you have? The answer is subtle, it is not always yes in next section will see a counter example.
Summary diagram:

Training and testing on different distributions

Example: Cat vs Non-cat

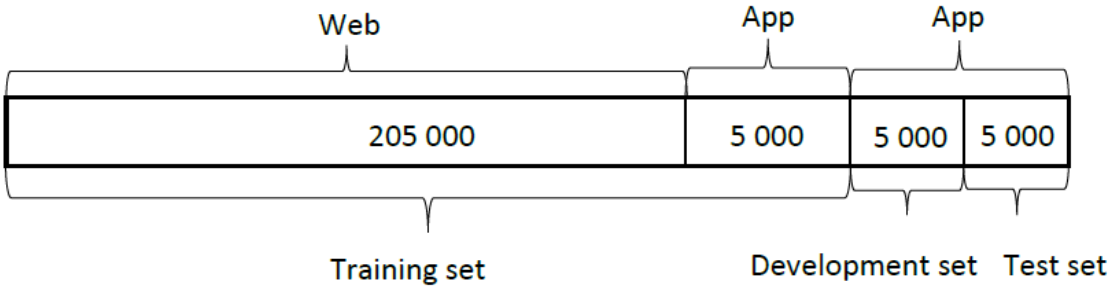
In this example, we want to create a mobile application that will classify and recognize pictures of cats taken and uploaded by users.

There are two sources of data used to develop the mobile app. The first data distribution is small, 10 000 pictures uploaded from the mobile application. Since they are from amateur users, the pictures are not professionally shot, not well framed and blurrier. The second source is from the web, you downloaded 200 000 pictures where cat's pictures are professionally framed and in high resolution.

The problem is that you have a different distribution:

1- small data set from pictures uploaded by users. This distribution is important for the mobile app.
2- bigger data set from the web.

The guideline used is that you have to choose a development set and test set to reflect data you expect to get in the future and consider important to do well.



The advantage of this way of splitting up is that the target is well defined.

The disadvantage is that the training distribution is different from the development and test set distributions. However, this way of splitting the data has a better performance in long term.

Bias and Variance with mismatched data distributions

Estimating the bias and variance of your learning algorithm really helps you prioritize what to work on next. But the way you analyze bias and variance changes when your training set comes from a different distribution than your dev and test sets. Let's see how.

Bias and variance with mismatched data distributions

Example: Cat classifier with mismatch data distribution

When the training set is from a different distribution than the development and test sets, the method to analyze bias and variance changes.

	Classification error (%)					
	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E	Scenario F
Human (proxy for Bayes error)	0	0	0	0	0	4
Training error	1	1	1	10	10	7
Training-development error	-	9	1.5	11	11	10
Development error	10	10	10	12	20	6
Test error	-	-	-	-	-	6

Scenario A

If the development data comes from the same distribution as the training set, then there is a large variance problem and the algorithm is not generalizing well from the training set.

However, since the training data and the development data come from a different distribution, this conclusion cannot be drawn. There isn't necessarily a variance problem. The problem might be that the development set contains images that are more difficult to classify accurately.

When the training set, development and test sets distributions are different, two things change at the same time. First of all, the algorithm trained in the training set but not in the development set. Second of all, the distribution of data in the development set is different.

It's difficult to know which of these two changes what produces this 9% increase in error between the training set and the development set. To resolve this issue, we define a new subset called training-development set. This new subset has the same distribution as the training set, but it is not used for training the neural network.

Scenario B

The error between the training set and the training- development set is 8%. In this case, since the training set and training-development set come from the same distribution, the only difference between them is the neural network sorted the data in the training and not in the training development. The neural network is not generalizing well to data from the same distribution that it hadn't seen before

Therefore, we have really a variance problem.

Scenario C

In this case, we have a mismatch data problem since the 2 data sets come from different distribution.

Scenario D

In this case, the avoidable bias is high since the difference between Bayes error and training error is 10 %.

Scenario D

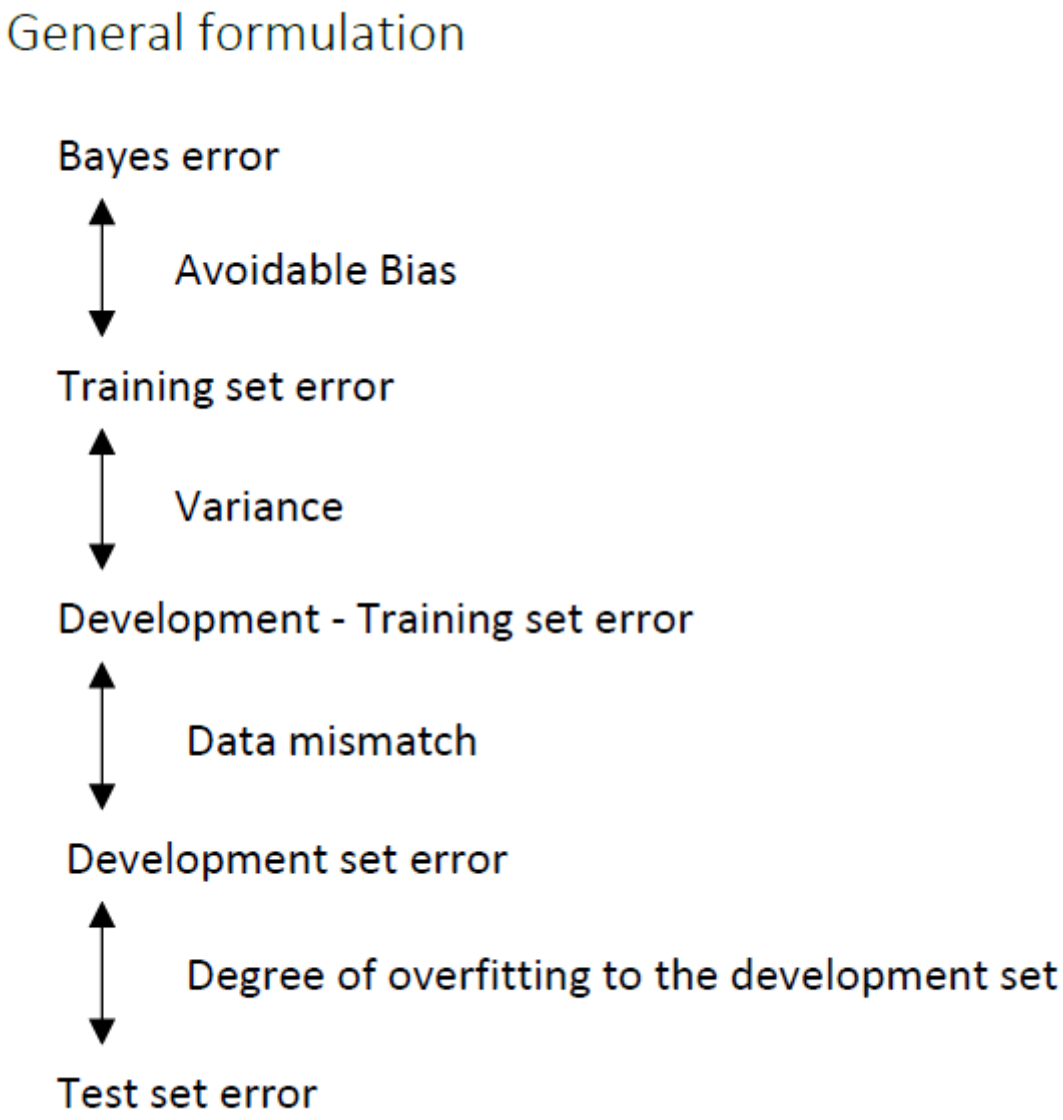
In this case, the avoidable bias is high since the difference between Bayes error and training error is 10 %.

Scenario E

In this case, there are 2 problems. The first one is that the avoidable bias is high since the difference between Bayes error and training error is 10 % and the second one is a data mismatched problem.

Scenario F

Development should never be done on the test set. However, the difference between the development set and the test set gives the degree of overfitting to the development set.



Addressing data mismatch

If your training set comes from a different distribution, than your dev and test set, and if error analysis shows you that you have a data mismatch problem, what can you do? There are completely systematic solutions to this, but let's look at some things you could try. If I find that I have a large data mismatch problem, what I usually do is carry out manual error analysis and try to understand the differences between the training set and the dev/test sets. To avoid overfitting the test set, technically for error analysis, you should manually only look at a dev set and not at the test set. But as a concrete example, if you're building the speech-activated rear-view mirror

application, you might look or, I guess if it's speech, listen to examples in your dev set to try to figure out how your dev set is different than your training set. So, for example, you might find that a lot of dev set examples are very noisy and there's a lot of car noise. And this is one way that your dev set differs from your training set. And maybe you find other categories of errors. For example, in the speech-activated rear-view mirror in your car, you might find that it's often mis-recognizing street numbers because there are a lot more navigational queries which will have street address. So, getting street numbers right is really important. When you have insight into the nature of the dev set errors, or you have insight into how the dev set may be different or harder than your training set, what you can do is then try to find ways to make the training data more similar. Or, alternatively, try to collect more data similar to your dev and test sets. So, for example, if you find that car noise in the background is a major source of error, one thing you could do is simulate noisy in-car data. So a little bit more about how to do this on the next slide. Or you find that you're having a hard time recognizing street numbers, maybe you can go and deliberately try to get more data of people speaking out numbers and add that to your training set. So, if your goal is to make the training data more similar to your dev set, what are some things you can do? One of the techniques you can use is artificial data synthesis.

So, to summarize, if you think you have a data mismatch problem, I recommend you do error analysis, or look at the training set, or look at the dev set to try this figure out, to try to gain insight into how these two distributions of data might differ. And then see if you can find some ways to get more training data that looks a bit more like your dev set. One of the ways we talked about is **artificial data synthesis** and artificial data synthesis does work. In speech recognition, I've seen artificial data synthesis significantly boost the performance of what were already very good speech recognition system. So, it can work very well. But, if you're using artificial data synthesis, just be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

Addressing data mismatch

This is a general guideline to address data mismatch:

- Perform manual error analysis to understand the error differences between training, development/test sets. Development should never be done on test set to avoid overfitting.
- Make training data or collect data similar to development and test sets. To make the training data more similar to your development set, you can use is artificial data synthesis. However, it is possible that if you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

Learning from multiple tasks

Transfer learning

One of the most powerful ideas in deep learning is that sometimes you can take knowledge the neural network has learned from one task and apply that knowledge to a separate task. So for example, maybe you could have the neural network learn to recognize objects like cats and then use that knowledge or use part of that knowledge to help you do a better job reading x-ray scans. This is called **transfer learning**.

Transfer Learning

Transfer learning refers to using the neural network knowledge for another application.

When to use transfer learning

- Task A and B have the same input x
- A lot more data for Task A than Task B
- Low level features from Task A could be helpful for Task B

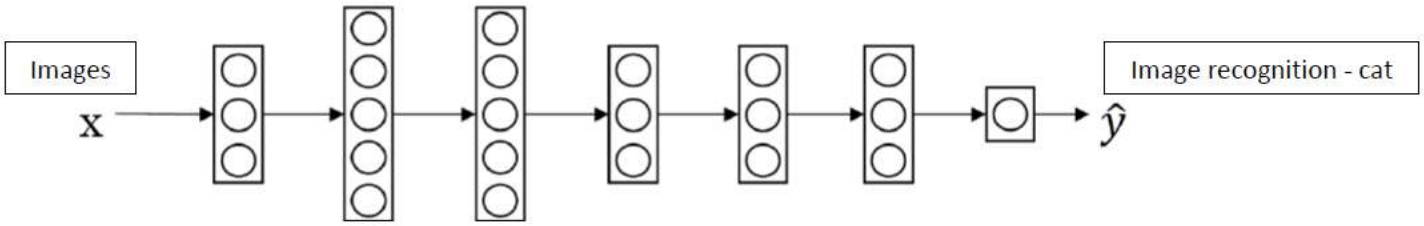
Example 1: Cat recognition - radiology diagnosis

The following neural network is trained for cat recognition, but we want to adapt it for radiology diagnosis. The neural network will learn about the structure and the nature of images. This initial phase of training on image recognition is called pre-training, since it will pre-initialize the weights of the neural network. Updating all the weights afterwards is called fine-tuning.

For cat recognition

Input x: image

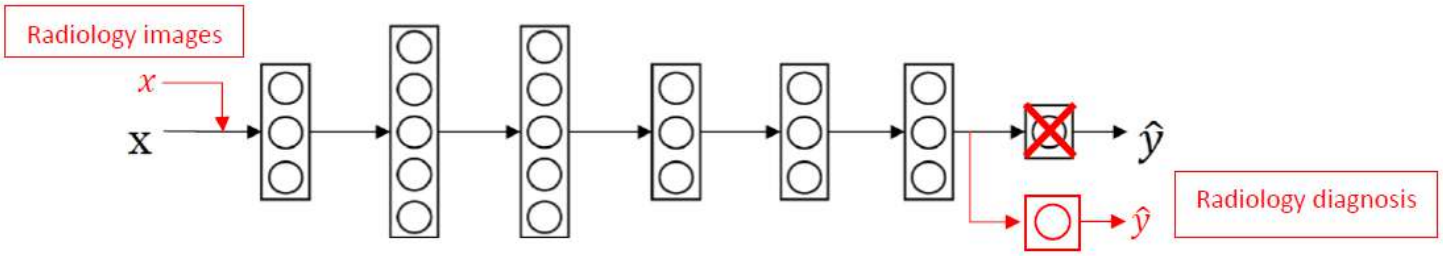
Output y – 1: cat, 0: no cat



Radiology diagnosis

Input x: Radiology images – CT Scan, X-rays

Output y :Radiology diagnosis – 1: tumor malign, 0: tumor benign



Guideline

- Delete last layer of neural network
- Delete weights feeding into the last output layer of the neural network
- Create a new set of randomly initialized weights for the last layer only
- New data set (x, y)

Multi-task learning

So whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B. In multi-task learning, you start off simultaneously, trying to have one neural network do several things at the same time and then each of these task helps hopefully all of the other task. Let's look at an example. Let's say you're building an autonomous vehicle, building a self driving car. Then your self driving car would need to detect several different things such as pedestrians, detect other cars, detect stop signs and also detect traffic lights and also other things.

Multi-task learning

Multi-task learning refers to having one neural network do simultaneously several tasks.

When to use multi-task learning

- Training on a set of tasks that could benefit from having shared lower-level features
- Usually: Amount of data you have for each task is quite similar
- Can train a big enough neural network to do well on all tasks

Example: Simplified autonomous vehicle

The vehicle has to detect simultaneously several things: pedestrians, cars, road signs, traffic lights, cyclists, etc. We could have trained four separate neural networks, instead of train one to do four tasks. However, in this case, the performance of the system is better when one neural network is trained to do four tasks than training four separate neural networks since some of the earlier features in the neural network could be shared between the different types of objects.

The input $x^{(i)}$ is the image with multiple labels
The output $y^{(i)}$ has 4 labels which are represents:

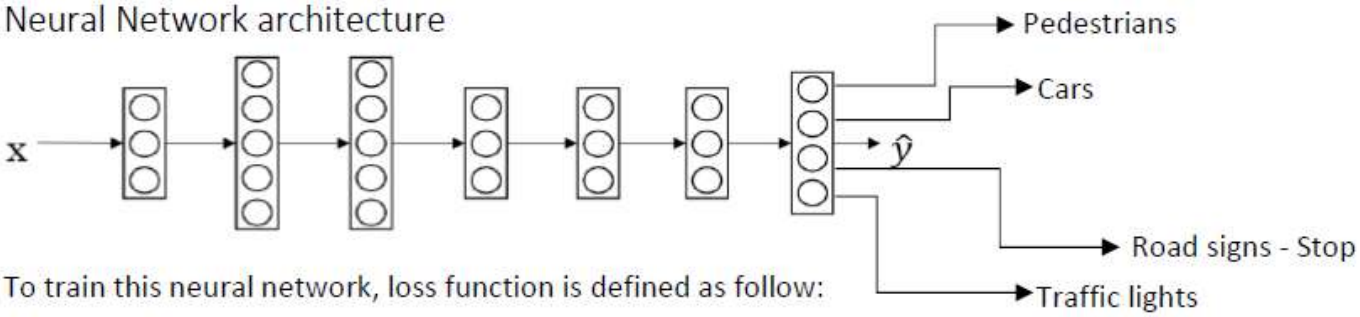
$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \text{Pedestrians} \\ \text{Cars} \\ \text{Road signs - Stop} \\ \text{Traffic lights} \end{matrix}$$

$$Y = \begin{bmatrix} | & | & | & | \\ y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} \\ | & | & | & | \end{bmatrix}$$

$$Y = (4, m)$$

$$Y = (4,1)$$

Neural Network architecture



To train this neural network, loss function is defined as follow:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\left(y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right)+\left(1-y_j^{(i)}\right)\log\left(1-\hat{y}_j^{(i)}\right)\right)$$

Also, the cost can be compute such as it is not influenced by the fact that some entries are not labeled.
Example:

$$Y = \begin{bmatrix} 1 & 0 & ? & ? \\ 0 & 1 & ? & 0 \\ 0 & 1 & ? & 1 \\ ? & 0 & 1 & 0 \end{bmatrix}$$

So what a researcher, Rich Carona, found many years ago was that the only times multi-task learning hurts performance compared to training separate neural networks is if your neural network isn't big enough. But if you can train a big enough neural network, then **multi-task learning** certainly should not or should very rarely hurt performance. And hopefully it will actually help performance compared to if you were training neural networks to do these different tasks in isolation. So that's it for multi-task learning. In practice, multi-task learning is used much less often than transfer learning. I see a lot of applications of transfer learning where you have a problem you want to solve with a small amount of data. So you find a related problem with a lot of data to learn something and transfer that to this new problem. But multi-task learning is just more rare that you have a huge set of tasks you want to use that you want to do well on, you can train all of those tasks at the same time. Maybe the one example is computer vision. In object detection I see more applications of multi-task any where one neural network trying to detect a whole bunch of objects at the same time works better than different neural networks trained separately to detect objects. But I would say that on average transfer learning is used much more today than multi-task learning, but both are useful tools to have in your arsenal.

So to summarize, multi-task learning enables you to train one neural network to do many tasks and this can give you better performance than if you were to do the tasks in isolation. Now one note of caution, in practice I see that transfer learning is used much more often than multi-task learning. So I do see a lot of tasks where if you want to solve a machine learning problem but you have a relatively small data set, then transfer learning can really help. Where if you find a related problem but you have a much bigger data set, you can train in your neural network from there and then transfer it to the problem where we have very low data. So transfer learning is used a lot today. There are some applications of transfer multi-task learning as well, but multi-task learning I think is used much less often than transfer learning. And maybe the one exception is computer vision object detection, where I do see a lot of applications of training a neural network to detect lots of different objects. And that works better than training separate neural networks and detecting the visual objects. But on average I think that even though transfer learning and multi-task learning often you're presented in a similar way, in practice I've seen a lot more applications of transfer learning than of multi-task learning. I think because often it's just difficult to set up or to find so many different tasks that you would actually want to train a single neural network for. Again, with some sort of computer vision, object detection examples being the most notable exception. So that's it for multi-task learning. Multi-task learning and transfer learning are both important tools to have in your tool bag.

End-to-end deep learning

What is end-to-end deep learning?

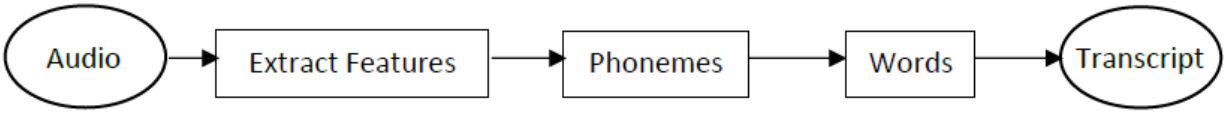
One of the most exciting recent developments in deep learning, has been the rise of end-to-end deep learning. So what is the end-to-end learning? Briefly, there have been some data processing systems, or learning systems that require multiple stages of processing. And what end-to-end deep learning does, is it can take all those multiple stages, and replace it usually with just a single neural network.

What is end-to-end deep learning

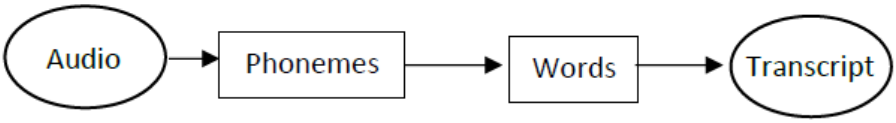
End-to-end deep learning is the simplification of a processing or learning systems into one neural network.

Example - Speech recognition model

The traditional way - small data set



The hybrid way - medium data set



The End-to-End deep learning way – large data set

End-to-end deep learning cannot be used for every problem since it needs a lot of labeled data. It is used mainly in audio transcripts, image captures, image synthesis, machine translation, steering in self-driving

Whether to use end-to-end deep learning

Let's say in building a machine learning system you're trying to decide whether or not to use an end-to-end approach. Let's take a look at some of the pros and cons of end-to-end deep learning so that you can come away with some guidelines on whether or not an end-to-end approach seems promising for your application. Here are some of the benefits of applying end-to-end learning. First is that end-to-end learning really just lets the data speak. So if you have enough X,Y data then whatever is the most appropriate function mapping from X to Y, if you train a big enough neural network, hopefully the neural network will figure it out. And by having a pure machine learning approach, your neural network learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.

Whether to use end-to-end deep learning

Before applying end-to-end deep learning, you need to ask yourself the following question: Do you have enough data to learn a function of the complexity needed to map x and y?

Pro:

- Let the data speak
  - By having a pure machine learning approach, the neural network will learn from x to y. It will be able to find which statistics are in the data, rather than being forced to reflect human preconceptions.

- Less hand-designing of components needed
  - It simplifies the design work flow.

Cons:

- Large amount of labeled data
  - It cannot be used for every problem as it needs a lot of labeled data.

- Excludes potentially useful hand-designed component
  - Data and any hand-design's components or features are the 2 main sources of knowledge for a learning algorithm. If the data set is small than a hand-design system is a way to give manual knowledge into the algorithm.

END OF COURSE

Course 4: Convolutional Neural Networks

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/convolutional-neural-networks/home/welcome

****
Course 4: Convolutional Neural Networks
****
In this course we'll learn how to build convolutional neural networks and apply it to image data. Thanks to deep learning, computer vision is working far better than just two years ago, and this is enabling numerous exciting applications ranging from safe autonomous driving, to accurate face recognition, to automatic reading of radiology images.

You will:
- Understand how to build a convolutional neural network, including recent variations such as residual networks.
- Know how to apply convolutional networks to visual detection and recognition tasks.
- Know to use neural style transfer to generate art.
- Be able to apply these algorithms to a variety of image, video, and other 2D or 3D data.

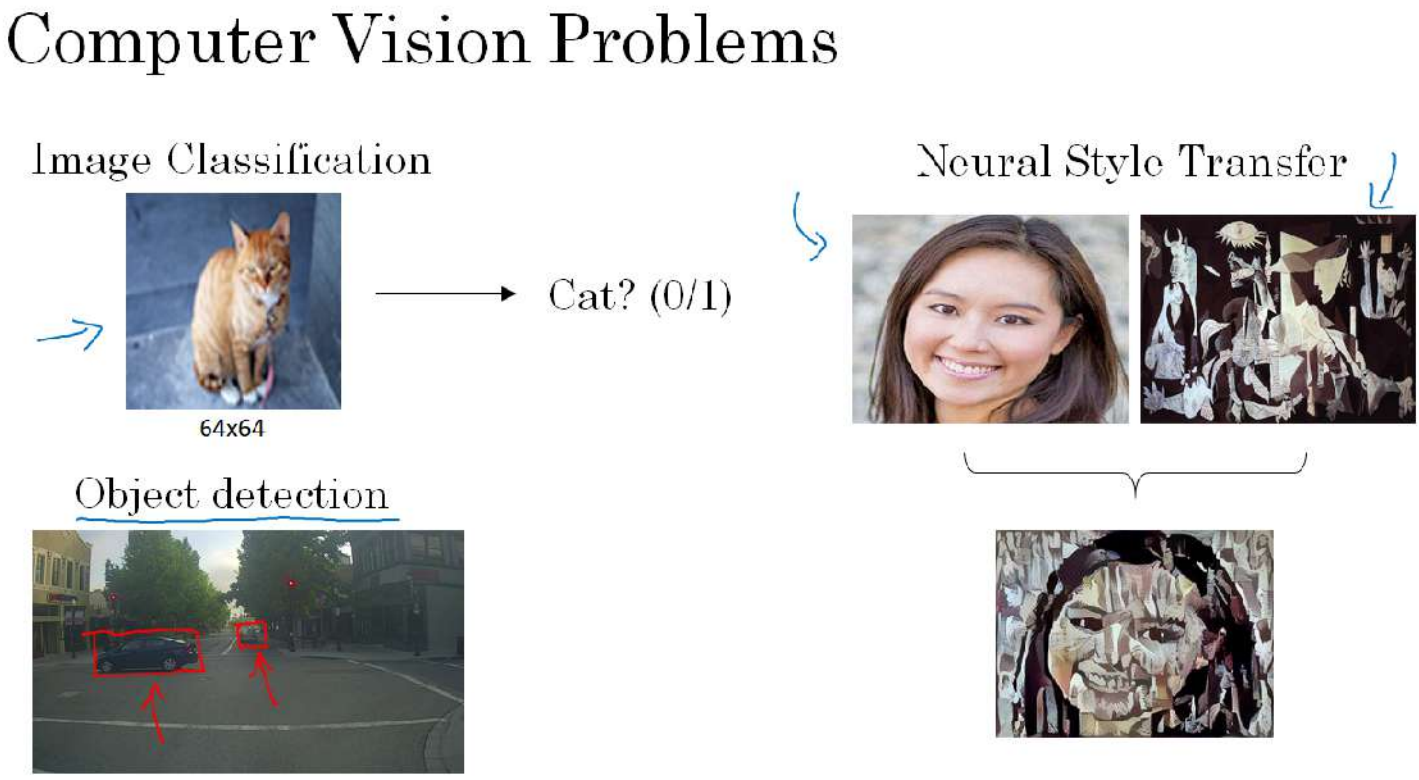
Week 1: Foundations of Convolutional Neural Networks
Learn to implement the foundation layers of CNN's (pooling, convolutions) and to stack them properly in a deep network to solve multi-class image classification problems.

Learning Objectives

- Understand the convolution operation
- Understand the pooling operation
- Remember the vocabulary used in convolutional neural network (padding, stride, filter, ...)
- Build a convolutional neural network for image multi-class classification

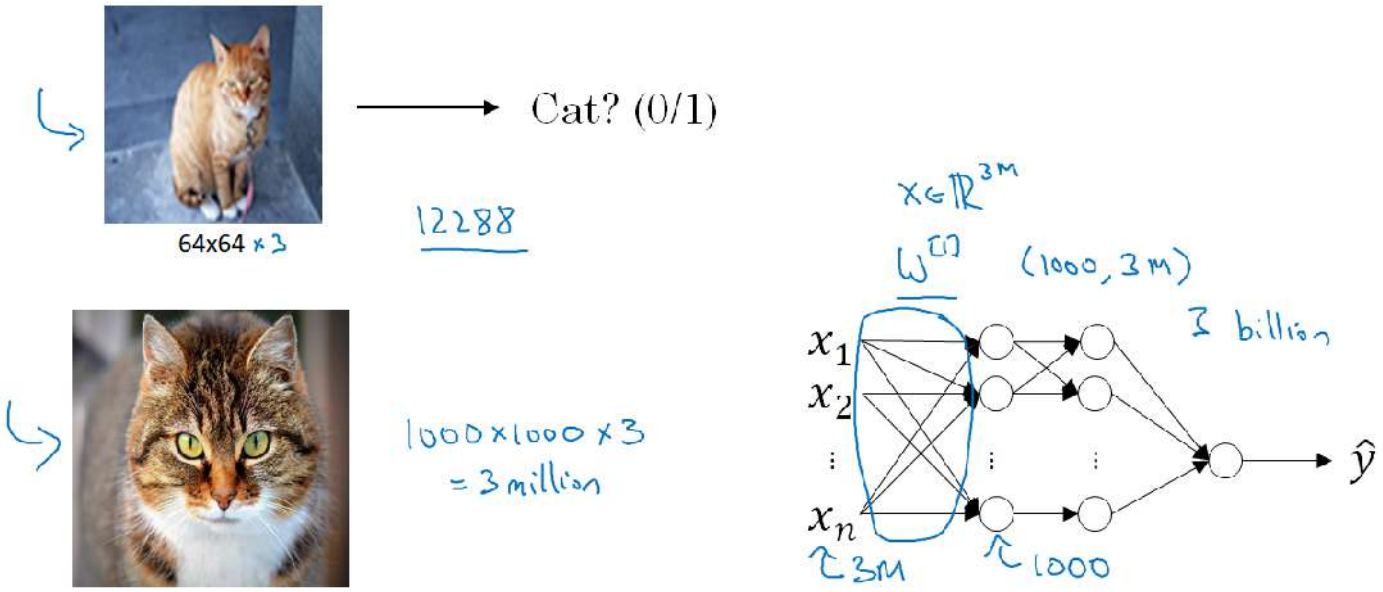
Convolutional Neural Networks
Computer Vision
Computer vision is one of the areas that's been advancing rapidly thanks to deep learning. Deep learning computer vision is now helping self-driving cars figure out where the other cars and pedestrians around so as to avoid them. Is making face recognition work much better than ever before, so that perhaps some of you will soon, or perhaps already, be able to unlock a phone, unlock even a door using just your face and if you look on your cell phone, I bet you have many apps that show you pictures of food, or pictures of a hotel, or just fun pictures of scenery. And some of the companies that build those apps are using deep learning to help show you the most attractive, the most beautiful, or the most relevant pictures and I think deep learning is even enabling new types of art to be created. So, I think the two reasons I'm excited about deep learning for computer vision and why I think you might be too. First, rapid advances in computer vision are enabling brand new applications to view, though they just were impossible a few years ago and by learning these tools, perhaps you will be able to invent some of these new products and applications. Second, even if you don't end up building computer vision systems per se, I found that because the computer vision research community has been so creative and so inventive in coming up with new neural network architectures and algorithms, is actually inspire that creates a lot cross-fertilization into other areas as well. For example, when I was working on speech recognition, I sometimes actually took inspiration from ideas from computer vision and borrowed them into the speech literature. So, even if you don't end up working on computer vision, I hope that you find some of the ideas you learn about in this course hopeful for some of your algorithms and your architectures. So with that, let's get started. Here are some examples of computer vision problems we'll study in this course. You've already seen image classifications, sometimes also

called image recognition, where you might take as input say a 64x64 image and try to figure out, is that a cat? Another example of the computer vision problem is object detection. So, if you're building a self-driving car, maybe you don't just need to figure out that there are other cars in this image. But instead, you need to figure out the position of the other cars in this picture, so that your car can avoid them. In object detection, usually, we have to not just figure out that these other objects say cars and picture, but also draw boxes around them. We have some other way of recognizing where in the picture are these objects. And notice also, in this example, that they can be multiple cars in the same picture, or at least every one of them within a certain distance of your car. Here's another example, maybe a more fun one is neural style transfer. Let's say you have a picture, and you want this picture repainted in a different style. So neural style transfer, you have a content image, and you have a style image. The image on the right is actually a Picasso. And you can have a neural network put them together to repaint the content image (that is the image on the left in diagram below), but in the style of the image on the right, and you end up with the image at the bottom. So, algorithms like these are enabling new types of artwork to be created.



One of the challenges of computer vision problems is that the inputs can get really big. For example, in previous section, we've worked with 64x64 images and so that's 64x64x3 because there are three color channels (RGB) and if you multiply that out, that's 12288. So **x the input features has dimension 12288** and that's not too bad but 64x64 is actually a very small image. If you work with larger images, maybe this is a 1000 pixel by 1000 pixel image, and that's actually just one megapixel but the dimension of the input features will be **1000x1000x3**, because you have three RGB channels, and that's three million. See below the example.

Deep Learning on large images



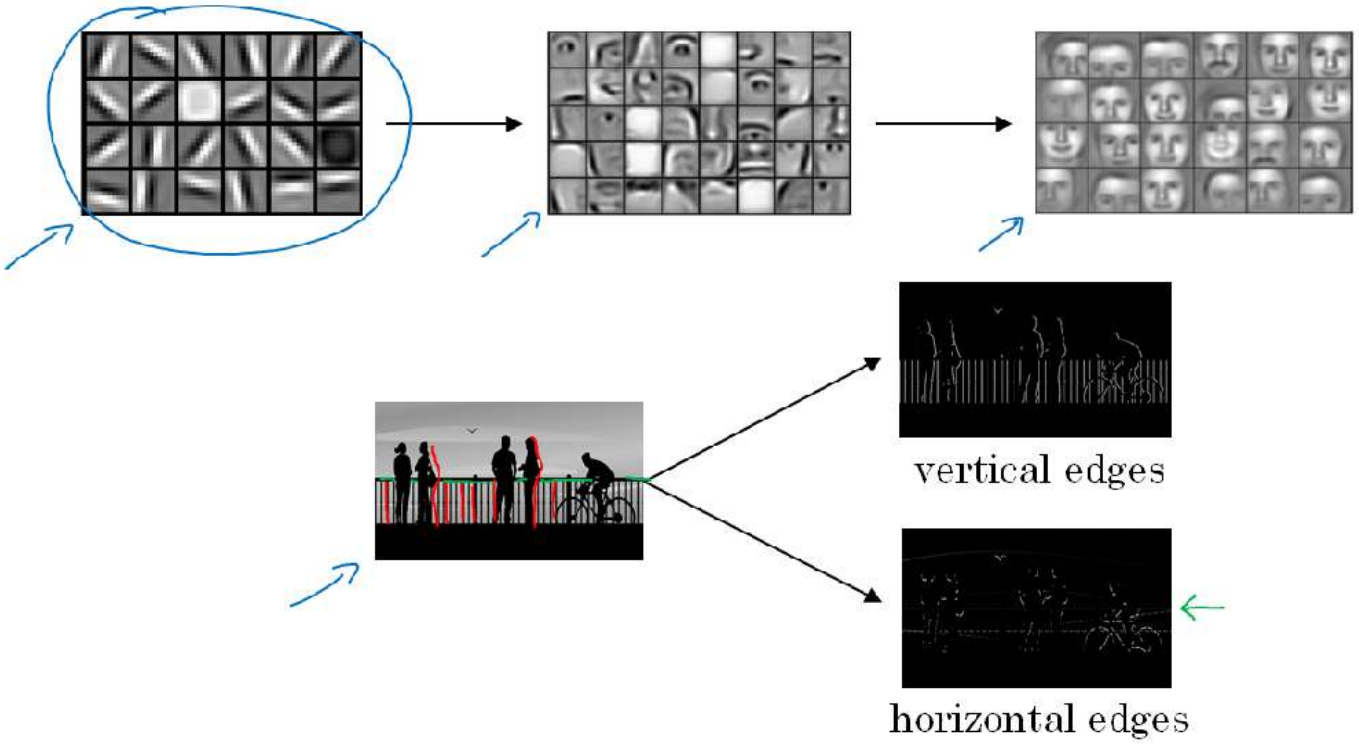
So, if you have three million input features, then this means that X here will be three million dimensional and so, if in the first hidden layer maybe you have just a 1000 hidden units, then the total number of weights that is the matrix W1, if you use a standard or fully connected network like we have seen before. This matrix will be a 1000x3 million dimensional matrix and this means that this matrix here will have three billion parameters which is just very, very large. and with that many parameters, it's difficult to get enough data to prevent a neural network from overfitting and also, the computational requirements and the memory requirements to train a neural network with three billion parameters is just a bit infeasible but for computer vision applications, you don't want to be stuck using only tiny little images. You want to use large images. To do that, you need to better implement the convolution operation, which is one of the fundamental building blocks of convolutional neural networks.

Edge Detection Example

The convolution operation is one of the fundamental building blocks of a convolutional neural network. Using edge detection as the motivating example in this section, you will see how the convolution operation works. In previous section, we've talked about how the **early layers of the neural network might detect edges and then the some later layers might detect cause of objects and then even later layers may detect cause of complete objects like people's faces in this case**. In this section, we'll see how we can detect edges in an image. Lets take an example.

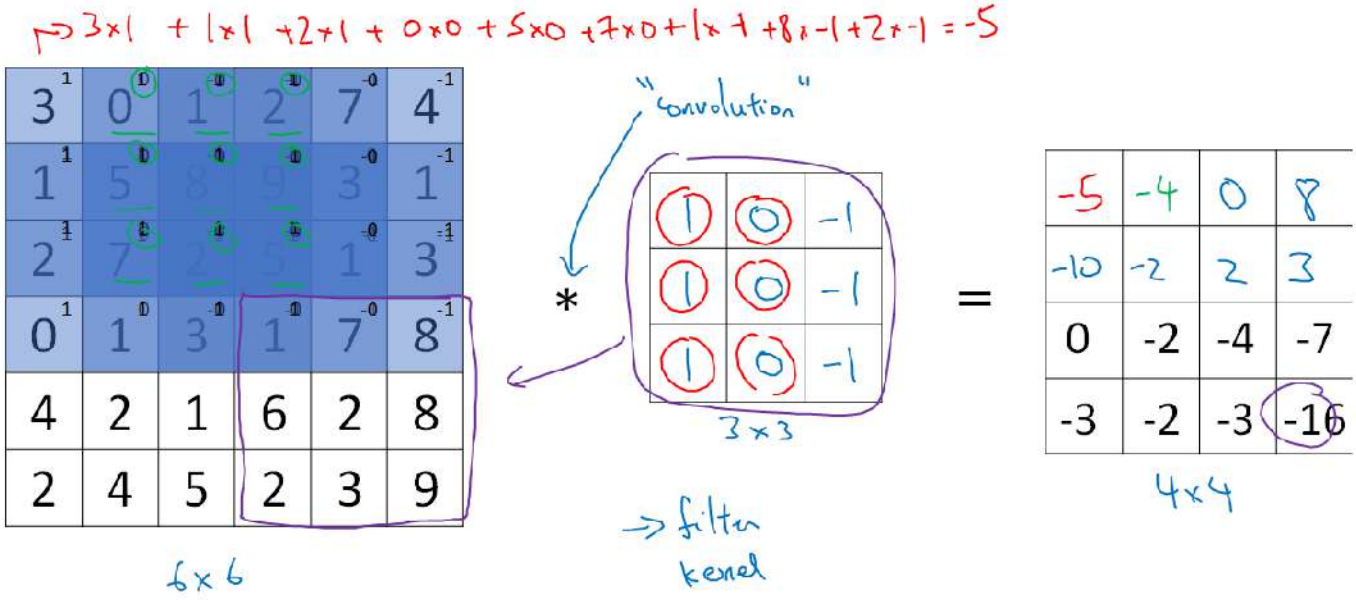
Given a picture like that for a computer to figure out what are the objects in this picture, the first thing you might do is maybe detect vertical edges in this image. For example, this image has all those vertical lines, where the buildings are, as well as kind of vertical lines idea all lines of these pedestrians and so those get detected in this vertical edge detector output. And you might also want to detect horizontal edges so for example, there is a very strong horizontal line where this railing is and that also gets detected sort of roughly here. How do you detect edges in image like this?

Computer Vision Problem



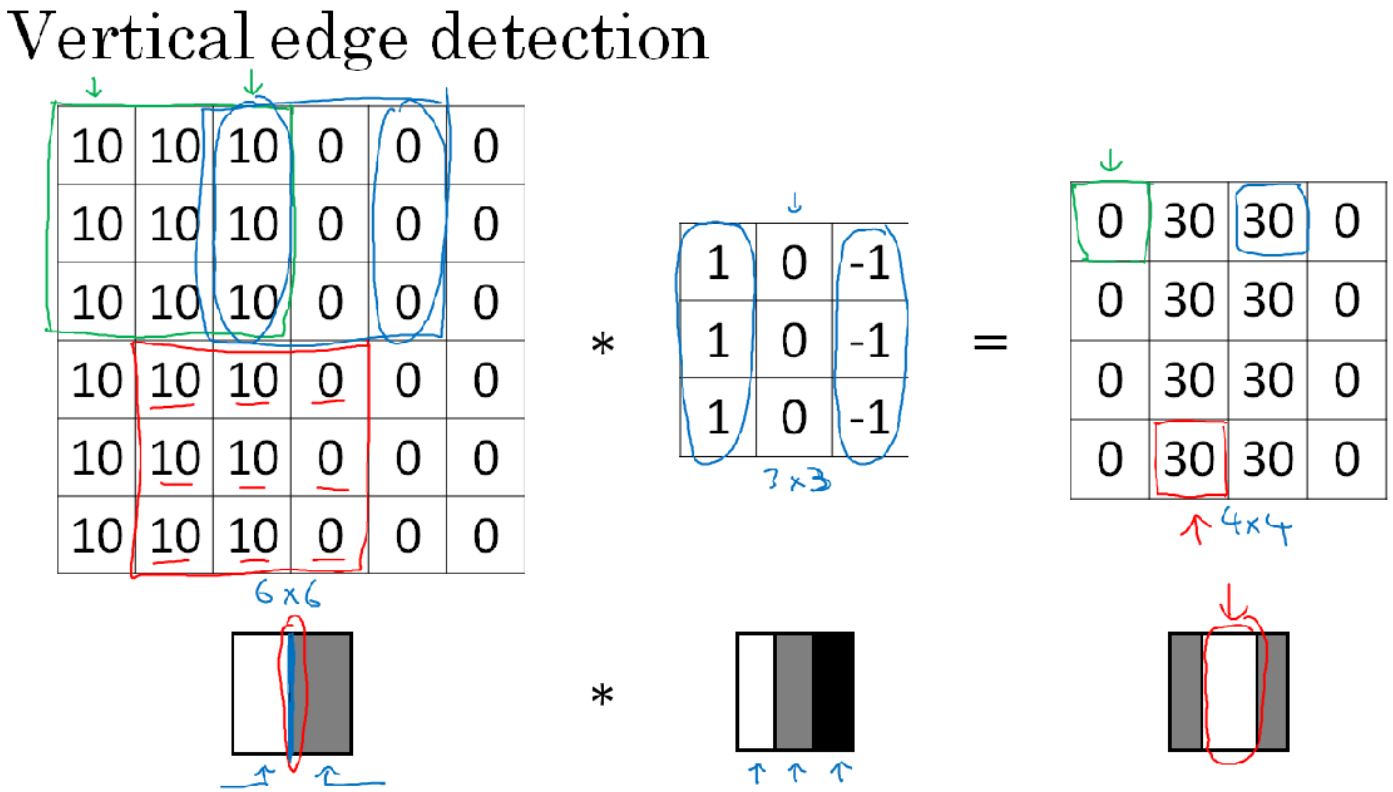
Let us look with an example. Here is a 6x6 grayscale image and because this is a grayscale image, this is just a 6x6x1 matrix rather than 6x6x3 because they are on a separate rgb channels. In order to detect edges or lets say vertical edges in his image, what you can do is construct a 3x3 matrix and in the terminology of convolutional neural networks, this is going to be called a **filter** and we're going to construct a 3x3 filter or 3x3 matrix and what you are going to do is take the 6x6 image and convolve it and the convolution operation is denoted by this asterisk and convolve it with the 3 x3 filter. The output of this convolution operator will be a 4x4 matrix, which you can interpret, which you can think of as a 4x4 image. The way you compute this 4 x4 output is shown in fig below.

Vertical edge detection



Looking on to the diagram to compute the first elements, the upper left element of this 4x4 matrix, what you are going to do is take the 3x3 filter and paste it on top of the 3x3 region of your original input image and what you should do is take the element wise product (check the calculation method at the top of diagram). Similarly we can calculate elements of 4x4 image. These are really just matrices of various dimensions. But the matrix on the left is convenient to interpret as image, and the one in the middle we interpret as a filter and the one on the right, you can interpret that as maybe another image. And this turns out to be a vertical edge detector.

Why is this doing vertical edge detection? Lets look at another example. To illustrate this, we are going to use a simplified image. Here is a simple 6x6 image where the left half of the image is 10 and the right half is zero. If you plot this as a picture, it might look like this, where the left half, the 10s, give you brighter pixel intensive values and the right half gives you darker pixel intensive values. I am using that shade of gray to denote zeros, although maybe it could also be drawn as black. But in this image, there is clearly a very strong vertical edge right down the middle of this image as it transitions from white to black or white to darker color. When you convolve this with the 3x3 filter and so this 3x3 filter can be visualized as follows, where is lighter, brighter pixels on the left and then this mid tone zeroes in the middle and then darker on the right. What you get is this matrix on the right.

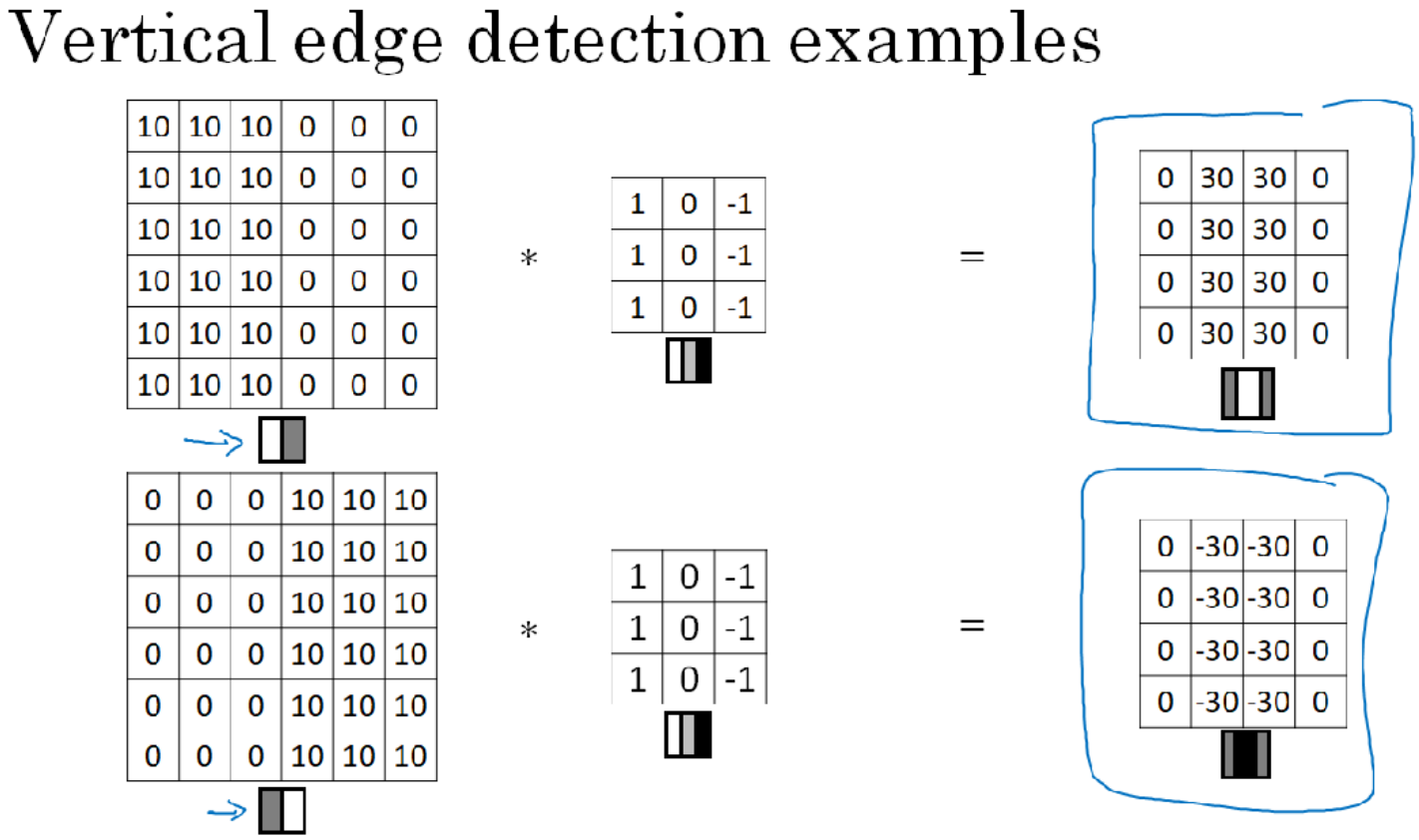


Now, if you plot this right most matrix's image it will look like that where there is this lighter region right in the middle and that corresponds to this having detected this vertical edge down the middle of your 6x6 image. In case the dimensions here seem a little bit wrong that the detected edge seems really thick, that's only because we are working with very small images in this example. And if you are using, say a 1000x1000 image rather than a 6x6 image then you find that this does a pretty good job, really detecting the vertical edges in your image. In this example, this bright region in the middle is just the output images way of saying that it looks like there is a strong vertical edge right down the middle of the image. Maybe one intuition to take away from vertical edge detection is that a vertical edge is a 3x3 region since we are using a 3x3 filter where there are bright pixels on the left, you do not care that much what is in the middle and dark pixels on the right. The middle in this 6x6 image is really where there could be bright pixels on the left and darker pixels on the right and that is why it thinks its a vertical edge over there. The

convolution operation gives you a convenient way to specify how to find these vertical edges in an image. You have now seen how the convolution operator works.

More Edge Detection

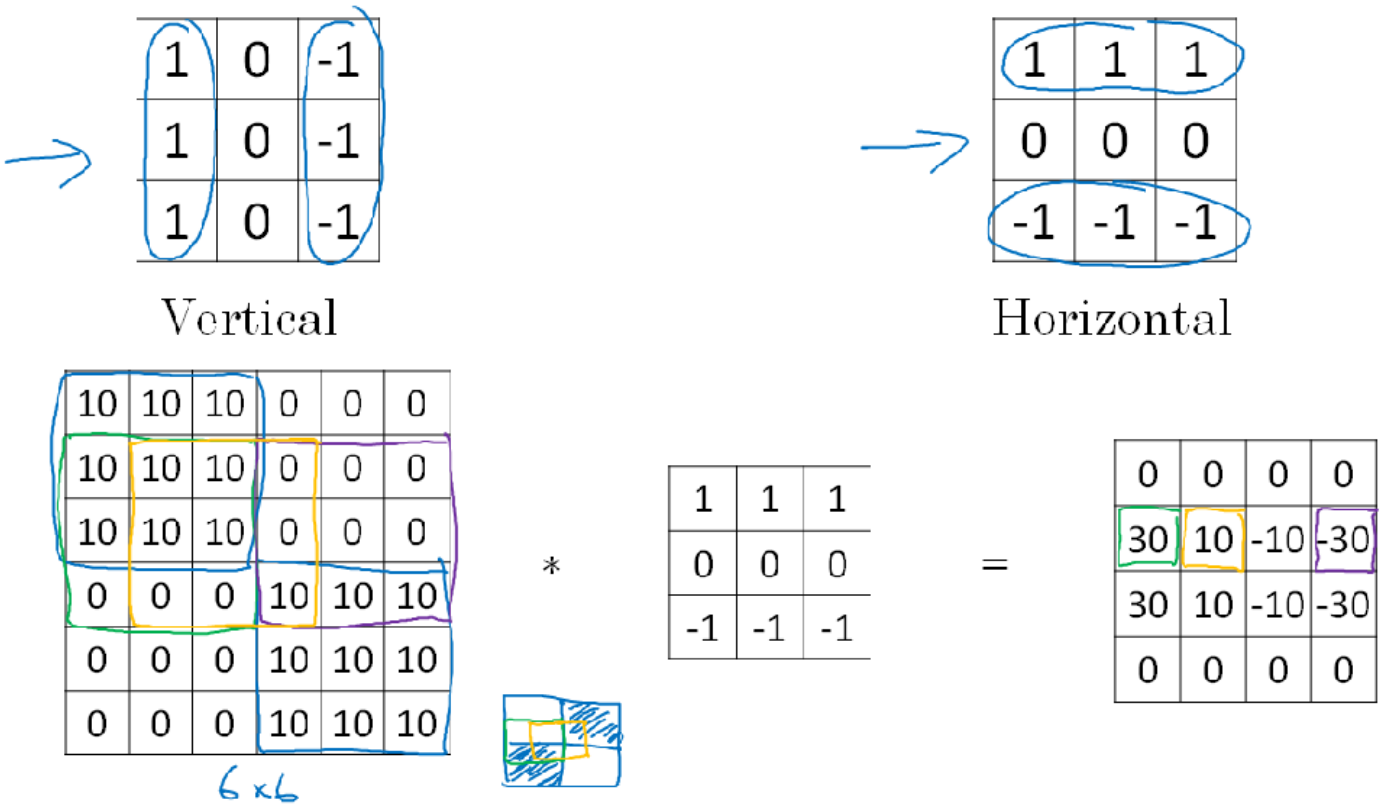
In this section, we'll learn the difference between positive and negative edges, that is, the difference between light to dark versus dark to light edge transitions and we'll also see other types of edge detectors, as well as how to have an algorithm learn, rather than have us hand code an edge detector as we've been doing so far. So let's get started. Here's the example you saw from the previous section, where you have a image, 6x6, there's light on the left and dark on the right, and convolving it with the vertical edge detection filter results in detecting the vertical edge down the middle of the image.



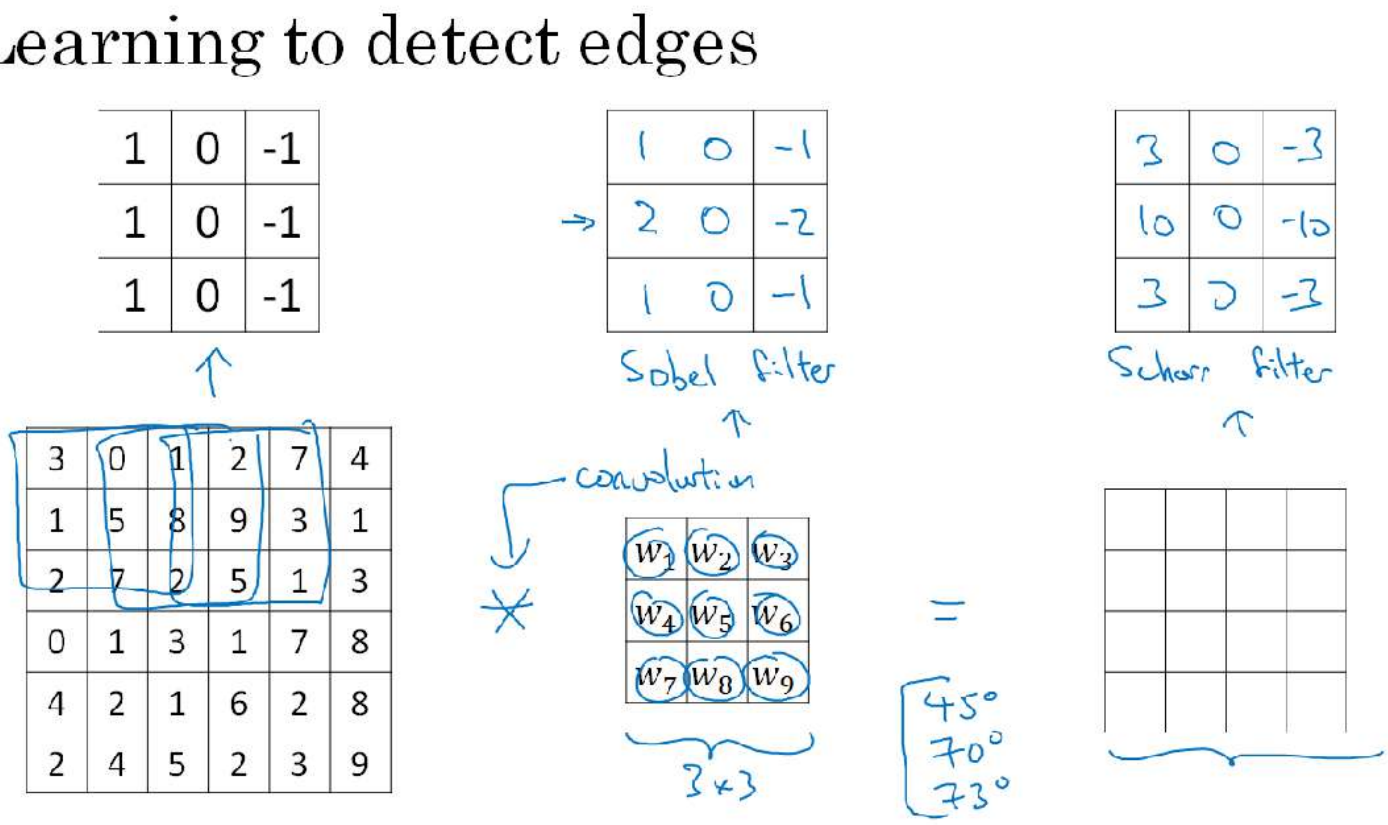
What happens in an image where the colors are flipped, where it is darker on the left and brighter on the right? So the 10s are now on the right half of the image and the 0s on the left. If you convolve it with the same edge detection filter, you end up with negative 30s, instead of 30 down the middle, and you can plot that as a picture that maybe looks like that. So because the shade of the transitions is reversed, the 30s now gets reversed as well and the negative 30s shows that this is a dark to light rather than a light to dark transition and if you don't care which of these two cases it is, you could take absolute values of this output matrix but this particular filter does make a difference between the light to dark versus the dark to light edges.

Let's see some more examples of edge detection. This 3x3 filter we've seen allows you to detect vertical edges. So maybe it should not surprise you too much that this 3x3 filter will allow you to detect horizontal edges. So as a reminder, a vertical edge according to this filter, is a 3x3 region where the pixels are relatively bright on the left part and relatively dark on the right part. So similarly, a horizontal edge would be a 3x3 region where the pixels are relatively bright on top and relatively dark in the bottom row. So here's one example, this is a more complex one, where you have here 10s in the upper left and lower right-hand corners. So if you draw this as an image, this would be an image which is going to be darker where there are 0s, so I'm going to shade in the darker regions, and then lighter in the upper left and lower right-hand corners. And if you convolve this with a horizontal edge detector, you end up with this.

Vertical and Horizontal Edge Detection



So in summary, different filters allow you to find vertical and horizontal edges. It turns out that the 3x3 vertical edge detection filter we've used is just one possible choice and historically, in the computer vision literature, there was a fair amount of debate about what is the best set of numbers to use. So here's something else you could use, which is maybe 1, 2, 1, 0, 0, 0, -1, -2, -1. This is called a Sobel filter. And the advantage of this is it puts a little bit more weight to the central row, the central pixel, and this makes it maybe a little bit more robust. But computer vision researchers will use other sets of numbers as well, like maybe instead of a 1, 2, 1, it should be a 3, 10, 3, right? And then -3, -10, -3. And this is called a Scharr filter. And this has yet other slightly different properties. And this is just for vertical edge detection. And if you flip it 90 degrees, you get horizontal edge detection. And with the rise of deep learning, one of the things we learned is that when you really want to detect edges in some complicated image, maybe you don't need to have computer vision researchers handpick these nine numbers. Maybe you can just learn them and treat the nine numbers of this matrix as parameters, which you can then learn using back propagation. And the goal is to learn nine parameters so that when you take the image, the six by six image, and convolve it with your three by three filter, that this gives you a good edge detector.

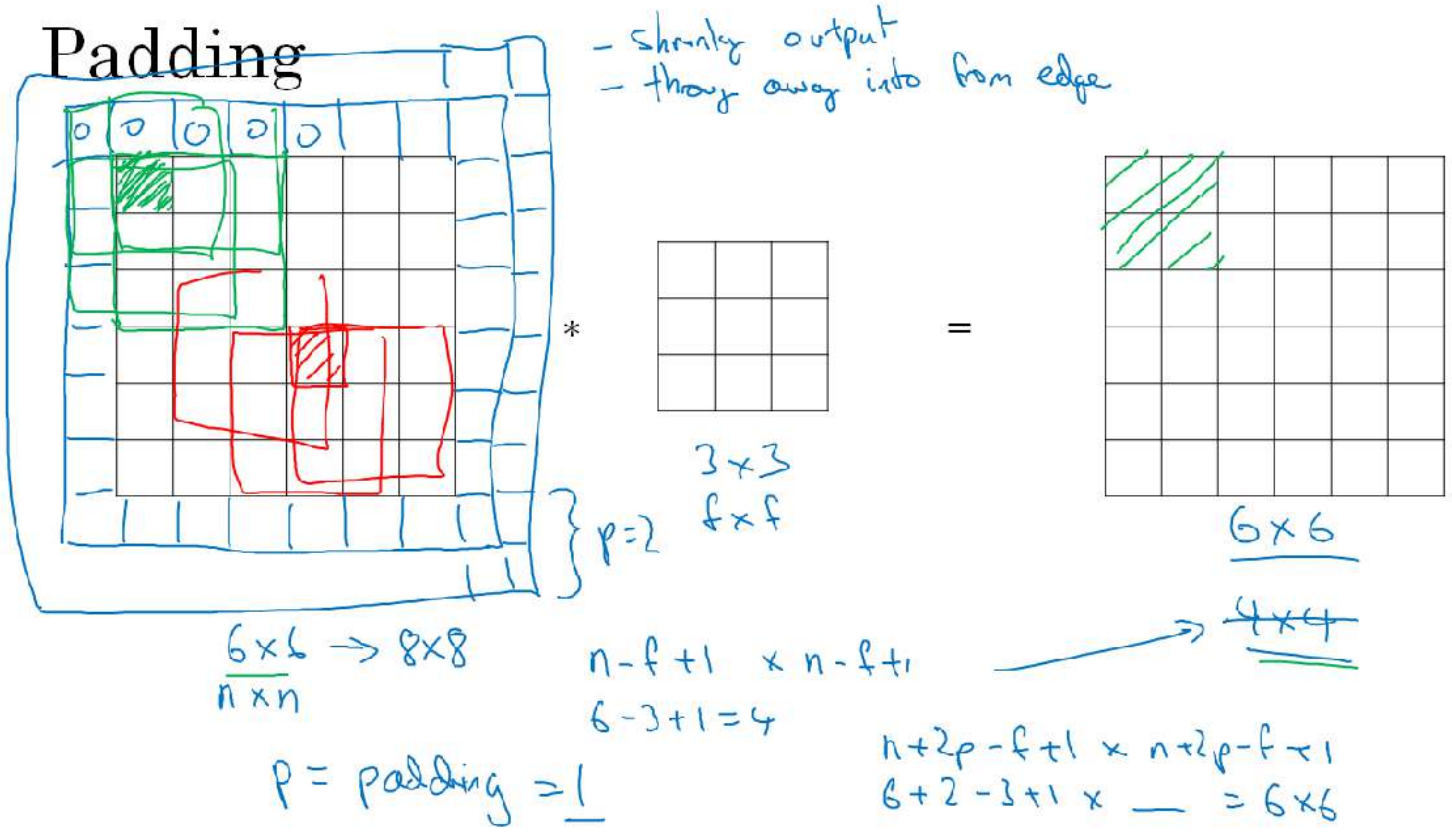


And what you see in later sections is that by just treating these nine numbers as parameters, the backprop can choose to learn 1, 1, 1, 0, 0, 0, -1,-1, if it wants, or learn the Sobel filter or learn the Scharr filter, or more likely learn something else that's even better at capturing the statistics of your data than any of these hand coded filters. And rather than just vertical and horizontal edges, maybe it can learn to detect edges that are at 45 degrees or 70 degrees or 73 degrees or at whatever orientation it chooses. And so by just letting all of these numbers be parameters and learning them automatically from data, we find that neural networks can actually learn low level features, can learn features such as edges, even more robustly than computer vision researchers are generally able to code up these things by hand. But underlying all these computations is still this convolution operation, Which allows back propagation to learn whatever three by three filter it wants and then to apply it throughout the entire image, at this position, at this position, at this position, in order to output whatever feature it's trying to detect. Be it vertical edges, horizontal edges, or edges at some other angle or even some other filter that we might not even have a name for in English.

So the idea you can treat these nine numbers as parameters to be learned has been one of the most powerful ideas in computer vision. And later in this section, we'll actually talk about the details of how you actually go about using back propagation to learn these nine numbers.

Padding

In order to build deep neural networks one modification to the basic convolutional operation that you need to really use is padding. Let's see how it works. What we saw in earlier sections is that if you take a 6x6 image and convolve it with a 3x3 filter, you end up with a 4x4 output with a 4x4 matrix, and that's because the number of possible positions with the 3x3 filter, there are only, sort of, 4x4 possible positions, for the 3x3 filter to fit in your 6x6 matrix and the math of this this turns out to be that if you have a end by end image and to involved that with an fxf filter, then the dimension of the output will be; **n- f +1 x n - f + 1**.



So the two downsides to this; one is that, if every time you apply a convolutional operator, your image shrinks, so you come from 6x6 down to 4x4 then, you can only do this a few times before your image starts getting really small, maybe it shrinks down to 1x1 or something, so maybe, you don't want your image to shrink every time you detect edges or to set other features on it, so that's one downside, and the second downside is that, if you look the pixel at the corner or the edge, this little pixel is touched as used only in one of the outputs, because this touches that 3x3 region. Whereas, if you take a pixel in the middle, say this pixel, then there are a lot of three by three regions that overlap that pixel and so, is as if pixels on the corners or on the edges are use much less in the output. So you're throwing away a lot of the information near the edge of the image. **So, to solve both of these problems, both the shrinking output, and when you build really deep neural networks, you see why you don't want the image to shrink on every step because if you have, maybe a hundred layer of deep net, then it'll shrinks a bit on every layer, then after a hundred layers you end up with a very small image. So that was one problem, the other is throwing away a lot of the information from the edges of the image.** So in order to fix both of these problems, what you can do is the full apply of convolutional operation. You can pad the image. So in this case, let's say you pad the image with an additional one border, with the additional border of one pixel all around the edges. So, if you do that, then instead of a 6x6 image, you've now padded this to 8x8 image and if you convolve an 8x8 with a 3x3 image you now get that out. Now, the 4x4 by the 6x6 image, so you managed to preserve the original input size of 6x6. So by convention when you pad, you padded with zeros and if p is the padding amounts. So in this case, p is equal to one, because we're padding all around with an extra boarder of one pixels, then the output becomes **n + 2p -f + 1 x n + 2p - f -1**. So we end up with a 6x6 image that preserves the size of the original image. So this being pixel actually influences all of these cells of the output and so this effective, maybe not by throwing away but counting less the information from the edge of the corner or the edge of the image is reduced. And we've shown here, the effect of padding deep border with just one pixel. If you want, you can also pad the border with two pixels, in which case I guess, you do add on another border here and they can pad it with even more pixels if you choose. So, I guess what I'm drawing here, this would be a padded equals to p plus two. In terms of how much to pad, it turns out there two common choices that are called, **Valid convolutions and Same convolutions.**

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

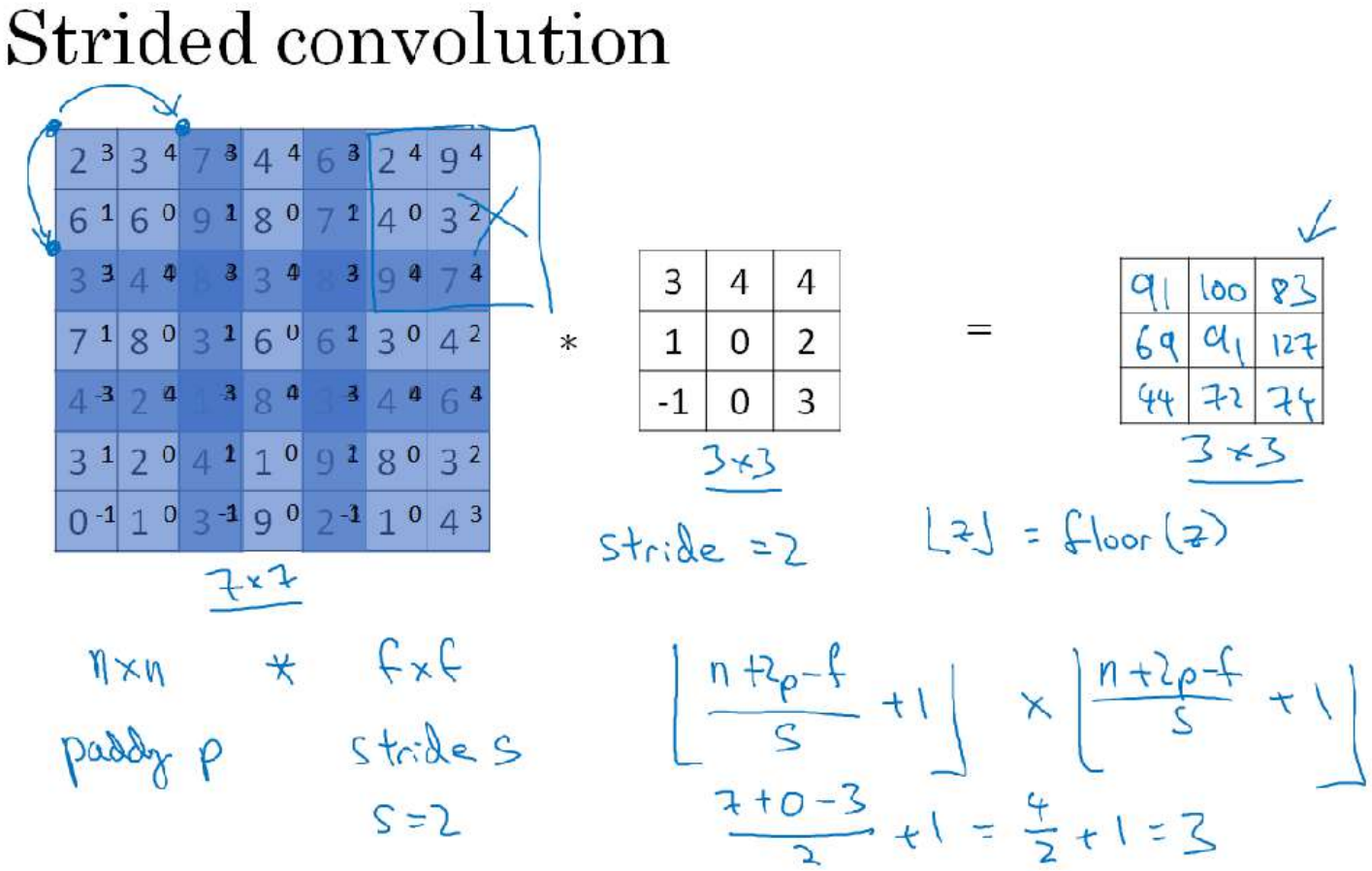
1×1
3×3
5×5
7×7

Not really is a great names but in a valid convolution, this basically means no padding. And so in this case you might have nxn image convolve with an fxf filter and this would give you an n minus f plus one by n minus f plus one dimensional output. So this is like the example we had previously on the previous sections where we had an n by n image convolve with the three by three filter and that gave you a four by four output. The other most common choice of padding is called the same convolution and that means when you pad, so the output size is the same as the input size. So if we actually look at this formula, when you pad by p pixels then, its as if n goes to n plus 2p and then you have from the rest of this, right? Minus f plus one. So we have an n by n image and the padding of a border of p pixels all around, then the output sizes of this dimension is xn plus 2p minus f plus one. And so, if you want n plus 2p minus f plus one to be equal to one, so the output size is same as input size, if you take this and solve for, I guess, n cancels out on both sides and if you solve for p, this implies that p is equal to f minus one over two. So when f is odd, by choosing the padding size to be as follows, you can make sure that the output size is same as the input size and that's why, for example, when the filter was three by three as this had happened in the previous slide, the padding that would make the output size the same as the input size was three minus one over two, which is one. And as another example, if your filter was five by five, so if f is equal to five, then, if you pad it into that equation you find that the padding of two is required to keep the output size the same as the input size when the filter is five by five. And by convention in computer vision, f is usually odd. It's actually almost always odd and you rarely see even numbered filters, filter works using computer vision. And I think that two reasons for that; one is that if f was even, then you need some asymmetric padding. So only if f is odd that this type of same convolution gives a natural padding region, had the same dimension all around rather than pad more on the left and pad less on the right, or something that asymmetric. And then second, when you have an odd dimension filter, such as three by three or five by five, then it has a central position and sometimes in computer vision its nice to have a distinguisher, it's nice to have a pixel, you can call the central pixel so you can talk about the position of the filter. Right, maybe none of this is a great reason for using f to be pretty much always odd but if you look a convolutional literature you see three by three filters are very common. You see some five by five, seven by sevens. And actually sometimes, later we'll also talk about one by one filters and that why that makes sense. But just by convention, I recommend you just use odd number filters as well. I think that you can probably get just fine performance even if you want to use an even

number value for f, but if you stick to the common computer vision convention, I usually just use odd number f. So you've now seen how to use padded convolutions. To specify the padding for your convolution operation, you can either specify the value for p or you can just say that this is a valid convolution, which means p equals zero or you can say this is a same convolution, which means pad as much as you need to make sure the output has same dimension as the input.

Strided Convolutions

Strided convolutions is another piece of the basic building block of convolutions as used in Convolutional Neural Networks. Example: Let's say you want to convolve a 7x7 image with 3x3 filter, except that instead of doing the usual way, we are going to do it with a stride of two.



What that means is you take the element Y's product as usual in this upper left three by three region and then multiply and add and that gives you 91. But then instead of stepping the blue box over by one step, we are going to step over by two steps. So, we are going to make it hop over two steps like so. Notice how the upper left hand corner has gone, jumping over one position. And then you do the usual element Y's product and summing it turns out 100. And now we are going to do they do that again, and make the blue box jump over by two steps. You end up there, and that gives you 83. Now, when you go to the next row, you again actually take two steps instead of one step so going to move the blue box over there. Notice how we are stepping over one of the positions and then this gives you 69, and now you again step over two steps, this gives you 91 and so on so 127. And then for the final row 44, 72, and 74. In this example, we convolve with a seven by seven matrix to this three by three matrix and we get a three by three outputs. The input and output dimensions turns out to be governed by the following formula, if you have an N by N image, they convolve with an F by F filter. And if you use padding P and stride S. In this example, S is equal to two then you end up with an output that is N plus two P minus F, and now because you're stepping S steps of the time, you step just one step of the time, you now divide by S plus one and then can apply the same thing. In our example, we have seven plus zero, minus three, divided by two S stride plus one equals let's see, that's four over two plus one equals three, which is why we wound up with this is three by three output. Now, just one last detail which is

what of this fraction is not an integer? In that case, we're going to round this down so this notation denotes the flow of something. This is also called the flow of Z. It means taking Z and rounding down to the nearest integer. The way this is implemented is that you take this type of blue box multiplication only if the blue box is fully contained within the image or the image plus to the padding and if any of this blue box kind of part of it hangs outside and you just do not do that computation. Then it turns out that if that's the convention that your three by three filter, must lie entirely within your image or the image plus the padding region before there's as a corresponding output generated that's convention. Then the right thing to do to compute the output dimension is to round down in case this N plus two P minus F over S is not an integer. Just to summarize the dimensions, if you have an N by N matrix or N by N image that you convolve with an F by F matrix or F by F filter with padding P N stride S, then the output size will have this dimension. It is nice we can choose all of these numbers so that there is an integer although sometimes you don't have to do that and rounding down is just fine as well. But please feel free to work through a few examples of values of N, F, P and S on yourself to convince yourself if you want, that this formula is correct for the output size.

Summary of convolutions

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

padding p $\quad$ stride s

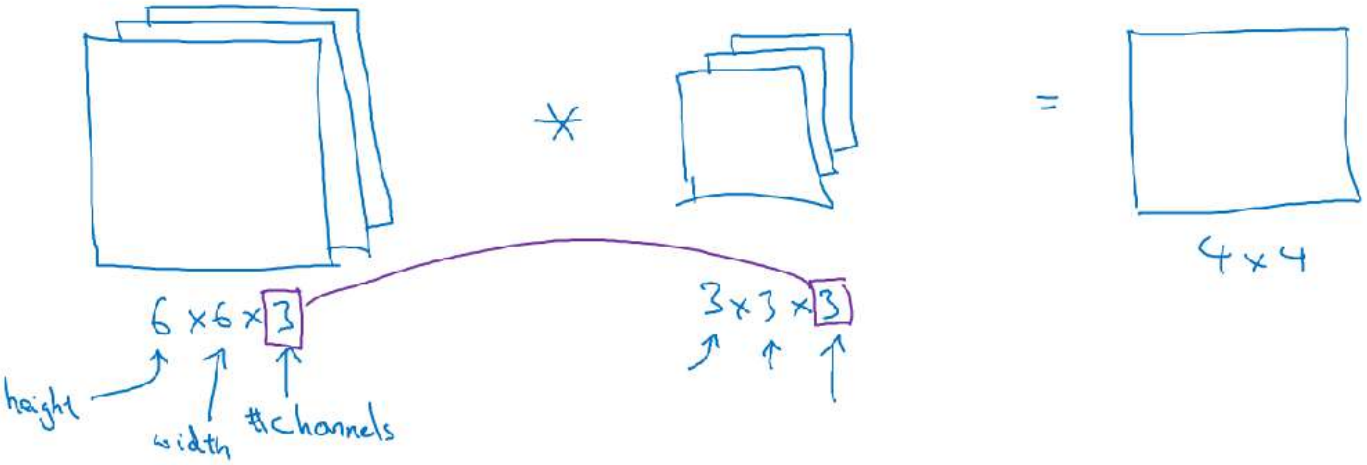
Output Size:

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

Convolutions Over Volume

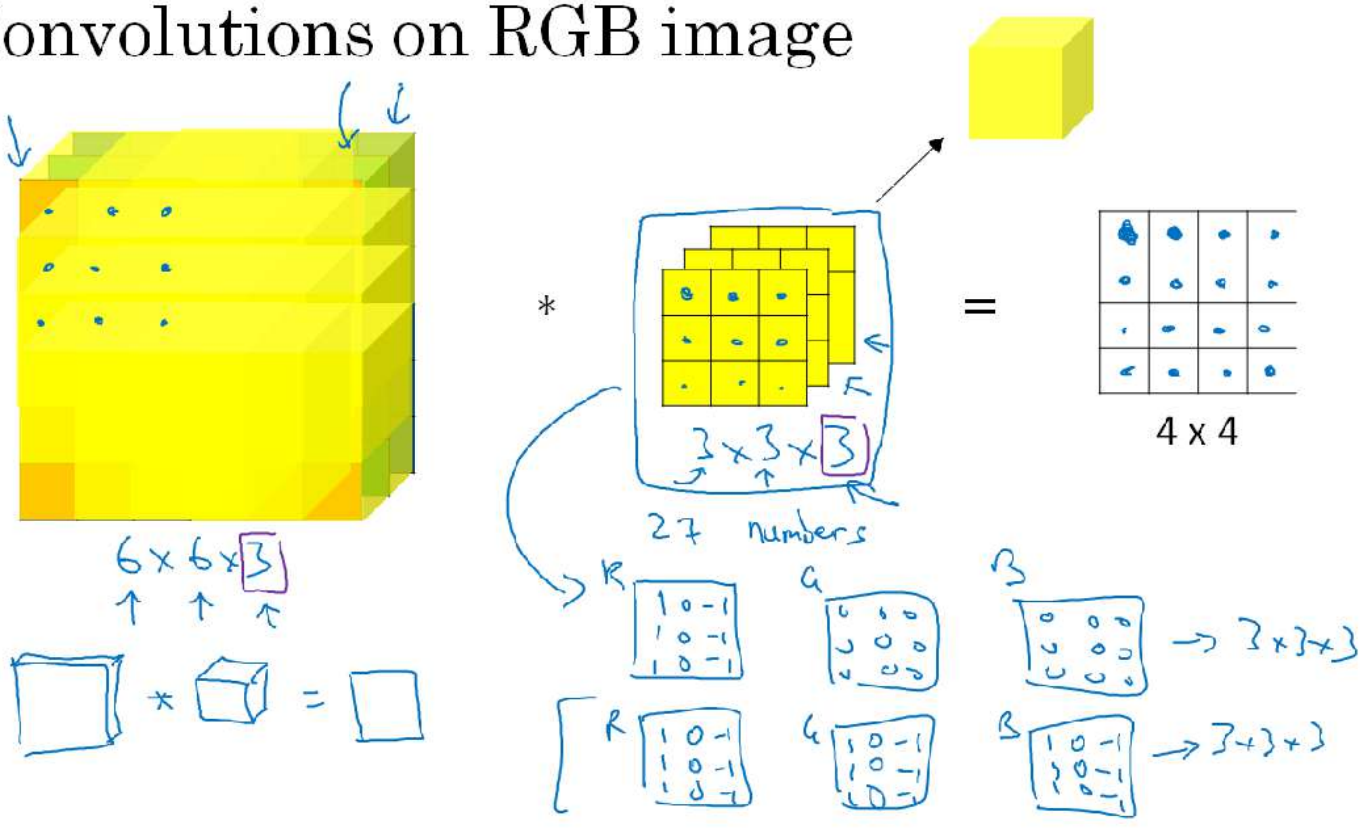
You've seen how convolutions over 2D images works. Now, let's see how you can implement convolutions over, not just 2D images, but over three dimensional volumes. Let's start with an example,

Convolutions on RGB images



let's say you want to detect features, not just in a great scale image, but in a RGB image. So, an RGB image might be instead of a six by six image, it could be six by six by three, where the three here responds to the three color channels. So, you think of this as a stack of three six by six images. In order to detect edges or some other feature in this image, you can vault this, not with a three by three filter, as we have previously, but now with also with a 3D filter, that's going to be 3x3x3. So the filter itself will also have three layers corresponding to the red, green, and blue channels. So to give these things some names, this first six here, that's the height of the image, that's the width, and this three is the number of channels. And your filter also similarly **has a height, a width, and the number of channels**. And the number of channels in your image must match the number of channels in your filter, so these two numbers have to be equal. We'll see on the next section how this convolution operation actually works, but the output of this will be a four by four image. And notice this is four by four by one, there's no longer a three at the end.

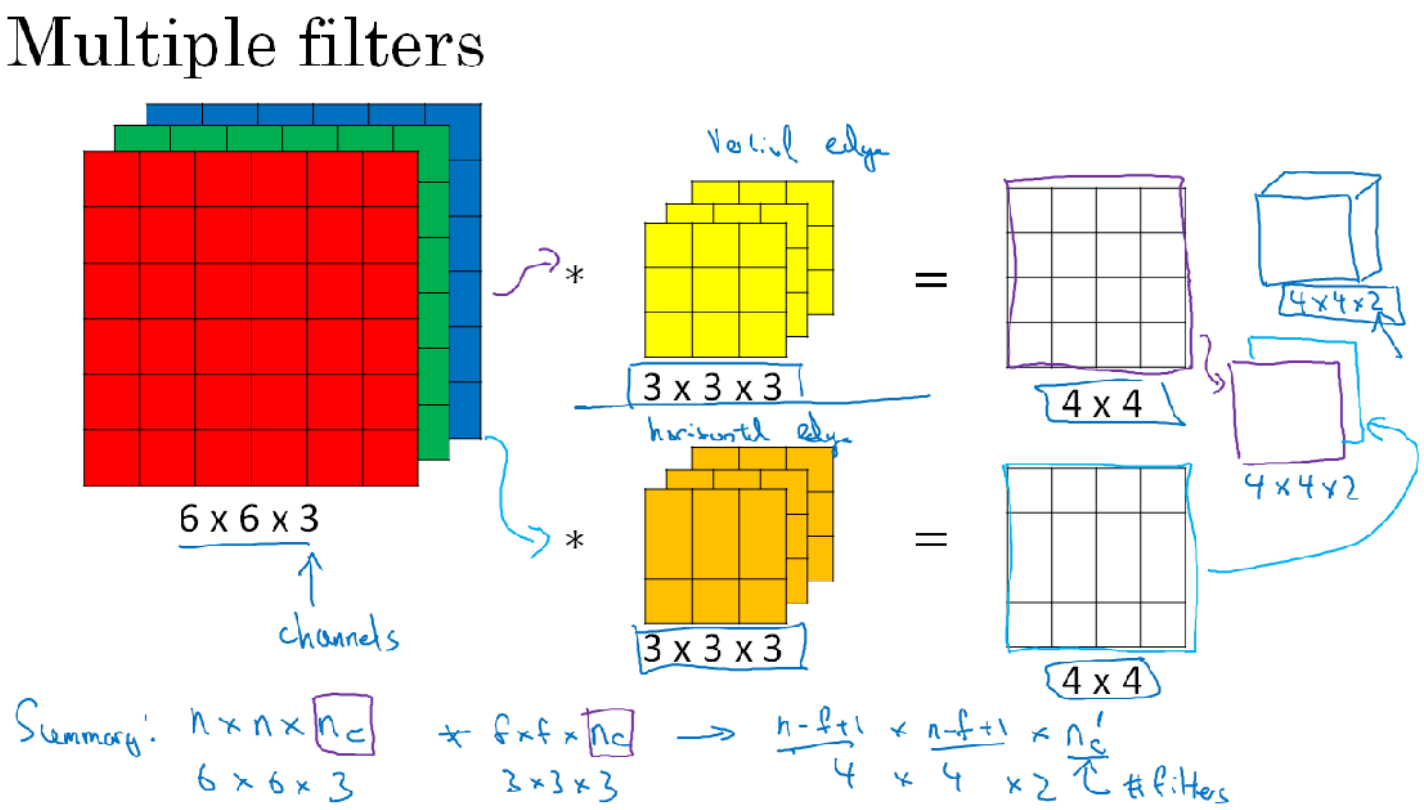
Convolutions on RGB image



Let's go through in detail how this works but let's use a more nicely drawn image. So here's the six by six by three image, and here's a three by three by three filter, and this last number, the number of channels matches the 3D image and the filter. So to simplify the drawing of this three by three by three filter, instead of joining it is a stack of the matrices, I'm also going to, sometimes, just draw it as this three dimensional cube, like that. So to compute the output of this convolutional operation, what you would do is take the three by three by three filter and first, place it in that upper left most position. So, notice that this three by three by three filter has 27 numbers, or 27 parameters, that's three cubes. And so, what you do is take each of these 27 numbers and multiply them with the corresponding numbers from the red, green, and blue channels of the image, so take the first nine numbers from red channel, then the three beneath it to the green channel, then the three beneath it to the blue channel, and multiply it with the corresponding 27 numbers that gets covered by this yellow cube show on the left. Then add up all those numbers and this gives you this first number in the output, and then to compute the next output you take this cube and slide it over by one, and again, due to 27 multiplications, add up the 27 numbers, that gives you this next output, do it for the next number over, for the next position over, that gives the third output and so on. That dives you the forth and then one row down and then the next one, to the next one, to the next one, and so on, you get the idea, until at the very end, that's the position you'll have for that final output.

So, what does this allow you to do? Well, here's an example, this filter is three by three by three. So, if you want to detect edges in the red channel of the image, then you could have the first filter, the one, one, one, one is one, one is one, one is one as usual, and have the green channel be all zeros, and have the blue filter be all zeros. And if you have these three stock together to form your three by three by three filter, then this would be a filter that detect edges, vertical edges but only in the red channel. Alternatively, if you don't care what color the vertical edge is in, then you might have a filter that's like this, whereas this one, one, one, minus one, minus one, minus one, in all three channels. So, by setting this second alternative, set the parameters, you then have a edge detector, a three by three by three edge detector, that detects edges in any color. And with different choices of these parameters you can get different feature detectors out of this three by three by three filter. And by convention, in computer vision, when you have an input with a certain height, a certain width, and a certain number of channels, then your filter will have a potential different height, different width, but the same number of channels. And in theory it's possible to have a filter that maybe only looks at the red channel or maybe a filter looks at only the green channel and a blue channel. And once again, you notice that convolving a volume, a six by six by three convolve with a three by three by three, that gives a four by four, a 2D output. Now that you know how to convolve on volumes, there is one last idea that will be crucial for building convolutional neural networks, which is what if we don't just wanted to detect vertical edges? What if we wanted to detect vertical edges and horizontal edges and maybe 45 degree edges and maybe 70 degree edges as well, but in other words, what if you want to use multiple filters at the

same time?

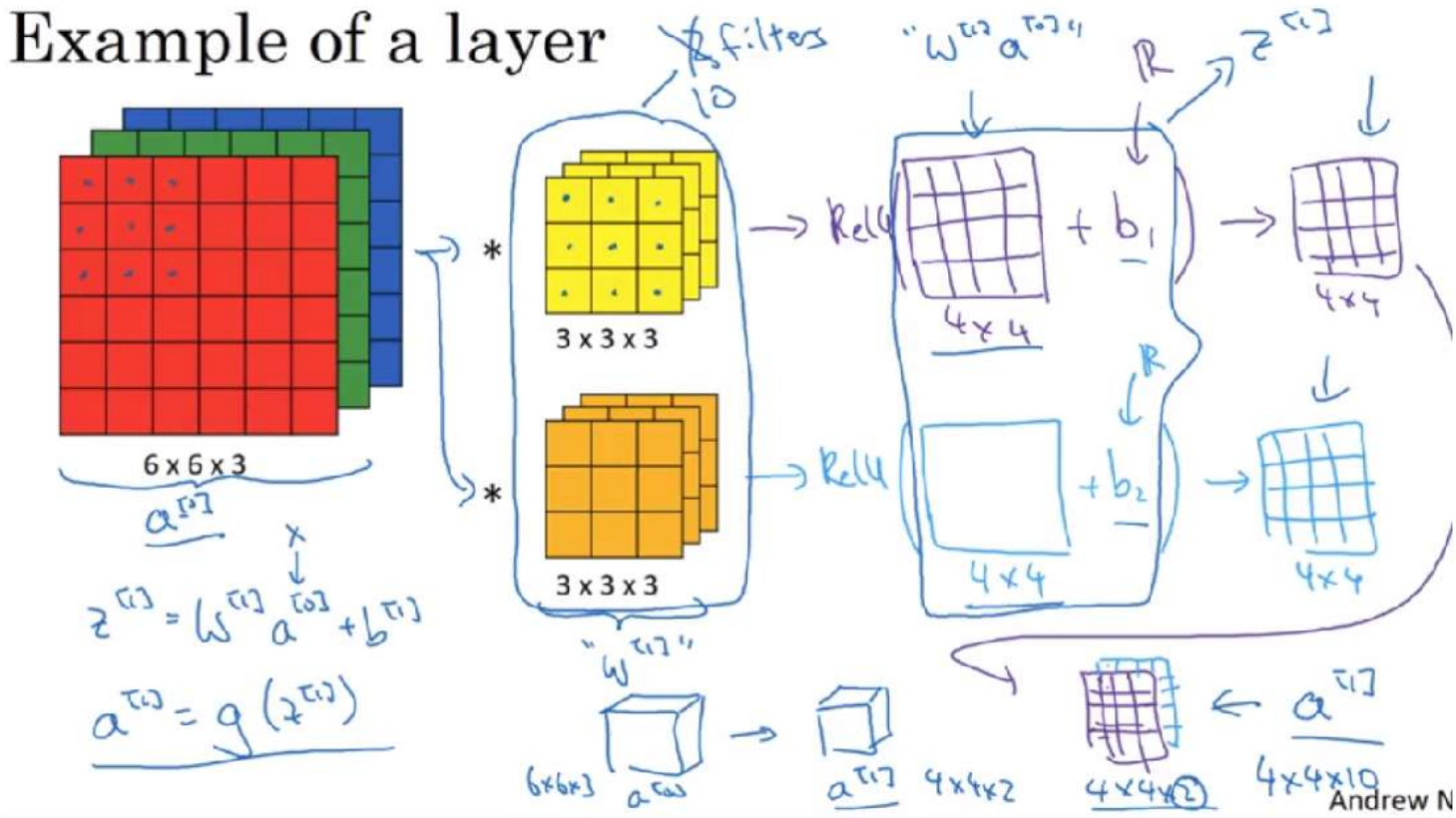


So, here's the picture we had from the previous section, we had six by six by three convolved with the three by three by three, gets four by four, and maybe this is a vertical edge detector, or maybe it's run to detect some other feature. Now, maybe a second filter may be denoted by this orange-ish color, which could be a horizontal edge detector. So, maybe convolving it with the first filter gives you this first four by four output and convolving with the second filter gives you a different four by four output. And what we can do is then take these two four by four outputs, take this first one within the front and you can take this second filter output and well, let me draw it here, put it at back as follows, so that by stacking these two together, you end up with a four by four by two output volume, right? And you can think of the volume as if we draw this is a box, I guess it would look like this. So this would be a four by four by two output volume, which is the result of taking your six by six by three image and convolving it or applying two different three by three filters to it, resulting in two four by four outputs that then gets stacked up to form a four by four by two volume. And the two here comes from the fact that we used two different filters.

So, let's just summarize the dimensions, if you have nxn number of channels input image, so an example, there's a 6x6x3, where n subscript C is the number of channels, and you convolve that with a f by f by, and again, this should be the same nC, so this was, three by three by three, and by convention this and this have to be the same number. Then, what you get is n minus f plus one by n minus f plus one by and you want to use this nC prime, or its really nC of the next layer, but this is the number of filters that you use. So this in our example would be be four by four by two. And I wrote this assuming that you use a stride of one and no padding. But if you used a different stride of padding than this n minus F plus one would be affected in a usual way, as we see in the previous videos. So this idea of convolution on volumes, turns out to be really powerful. Only a small part of it is that you can now operate directly on RGB images with three channels. But even more important is that you can now detect two features, like vertical, horizontal edges, or 10, or maybe a 128, or maybe several hundreds of different features. And the output will then have a number of channels equal to the number of filters you are detecting. And as a note of notation, I've been using your number of channels to denote this last dimension in the literature, people will also often call this the depth of this 3D volume and both notations, channels or depth, are commonly used in the literature.

One Layer of a Convolution Network

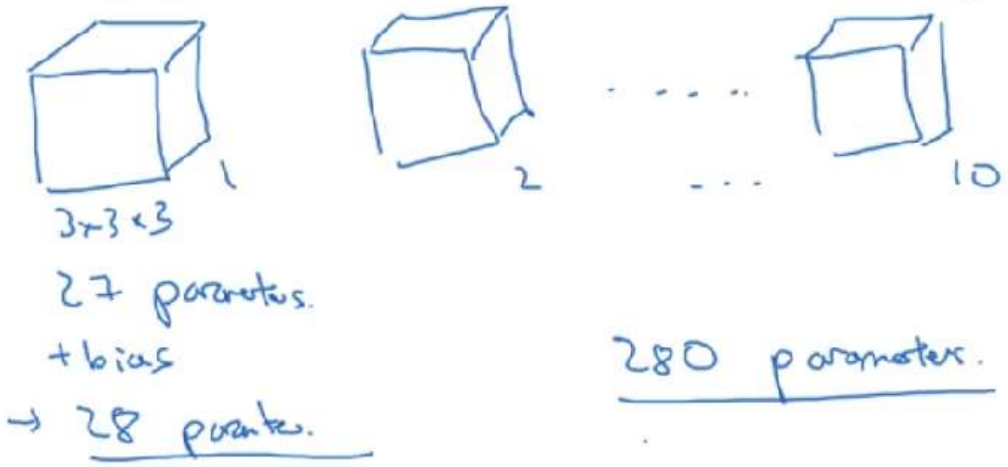
We are now ready to see how to build one layer of a convolutional neural network, let's go through the example. You've seen at the previous section how to take a 3D volume and convolve it with say two different filters. In order to get in this example to different 4 by 4 outputs. So let's say convolving with the first filter gives this first 4 by 4 output, and convolving with this second filter gives a different 4 by 4 output. The final thing to turn this into a convolutional neural net layer, is that for each of these we're going to add it bias, so this is going to be a real number. And where python broadcasting, you kind of have to add the same number so every one of these 16 elements. And then apply a non-linearity which for this illustration that says relative non-linearity, and this gives you a 4 by 4 output, all right? After applying the bias and the non-linearity. And then for this thing at the bottom as well, you add some different bias, again, this is a real number. So you add the single number to all 16 numbers, and then apply some non-linearity, let's say a real non-linearity. And this gives you a different 4 by 4 output. Then same as we did before, if we take this and stack it up as follows, so we ends up with a 4 by 4 by 2 outputs. Then this computation where you come from a 6 by 6 by 3 to 4 by 4 by 4, this is one layer of a convolutional neural network. So to map this back to one layer of four propagation in the standard neural network, in a non-convolutional neural network. Remember that one step before the prop was something like this, right? z1 = w1* a0, a0 was also equal to x, and then plus b[1]. And you apply the non-linearity to get a[1], so that's g(z[1]). So this input here, in this analogy this is a[0], this is x3 and these filters here, this plays a role similar to w1. And you remember during the convolution operation, you were taking these 27 numbers, or really well, 27 times 2, because you have two filters. You're taking all of these numbers and multiplying them. So you're really computing a linear function to get this 4 x 4 matrix. So that 4 x 4 matrix, the output of the convolution operation, that plays a role similar to w1 * a0. That's really maybe the output of this 4 x 4 as well as that 4 x 4. And then the other thing you do is add the bias. So, this thing here before applying value, this plays a role similar to z. And then it's finally by applying the non-linearity, this kind of this I guess. So, this output plays a role, this really becomes your activation at the next layer. So this is how you go from a0 to a1, as far as the linear operation and then convolution has all these multiples. So the convolution is really applying a linear operation and you have the biases and the applied value operation and you've gone from a 6 by 6 by 3, dimensional a0, through one layer of neural network to, I guess a 4 by 4 by 2 dimensional a(1). And so 6 by 6 by 3 has gone to 4 by 4 by 2, and so that is one layer of convolutional net.



Now in this example we have two filters, so we had two features of you will, which is why we round up with our output 4 x4x2. But if for example we instead had 10 filters instead of 2, then we would have round up with the 4x4x10 dimensional output volume. Because we'll be taking 10 of these not just two of them, and stacking them up to form a 4 x4x10 output volume, and that's what a1 would be. So, to make sure you understand this, let's go through an exercise. Let's suppose you have 10 filters, not just two filters, that are 3x3x3 and 1 layer of a neural network, how many parameters does this layer have? Well, let's figure this out. Each filter, is a 3 x 3 x 3 volume, so 3 x 3 x 3, so each fill has 27 parameters, all right? There's 27 numbers to be run, and plus the bias. So that was the b parameter, so this gives you 28 parameters.

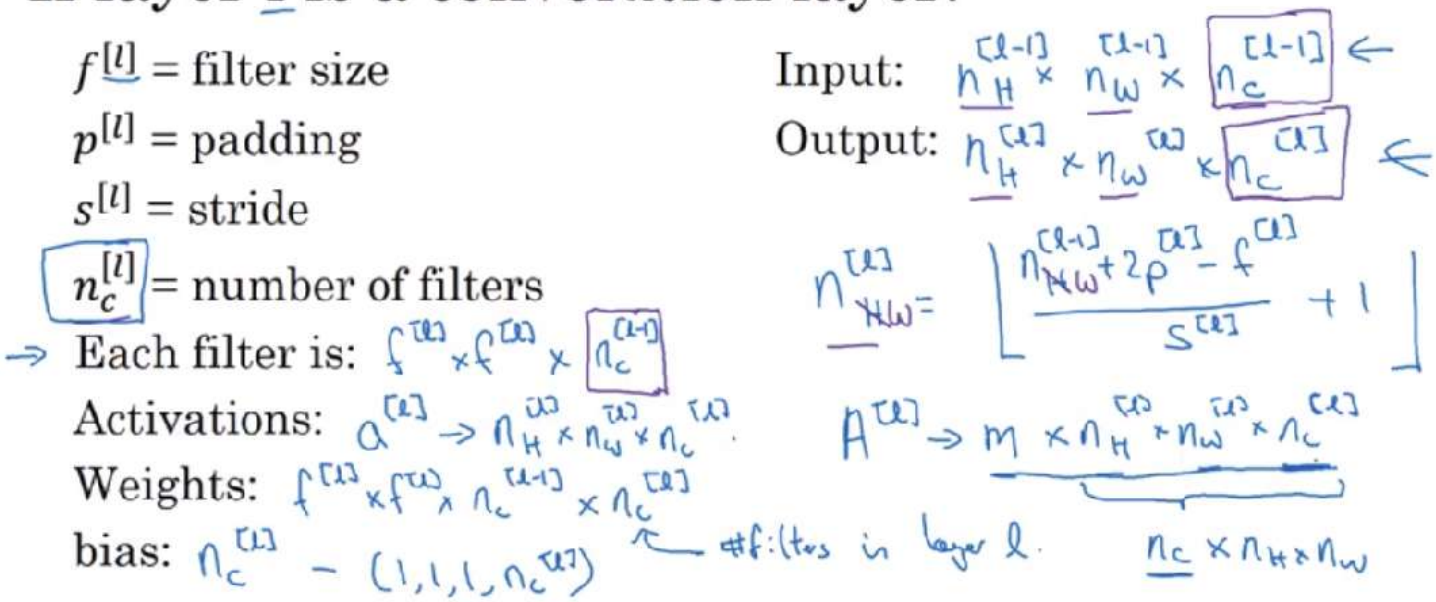
Number of parameters in one layer

If you have 10 filters that are 3 x 3 x 3 in one layer of a neural network, how many parameters does that layer have?



and then if you imagine that on the previous section we had drawn two filters, but now if you imagine that you actually have ten of these, right? 1, 2..., 10 of these, then all together you'll have 28 times 10, so that will be 280 parameters. Notice one nice thing about this, is that no matter how big the input image is, the input image could be 1,000x1,000 or 5,000x5,000, but the number of parameters you have still remains fixed as 280. And you can use these **ten filters to detect features, vertical edges, horizontal edges maybe other features anywhere even in a very, very large image is just a very small number of parameters.** So these is really one property of convolution neural network that makes less prone to overfitting then if you could. So once you've learned 10 feature detectors that work, you could apply this even to large images. And the number of parameters still is fixed and relatively small, as 280 in this example. Let's just summarize the notation we are going to use to describe one layer to describe a covolutional layer in a convolutional neural network.

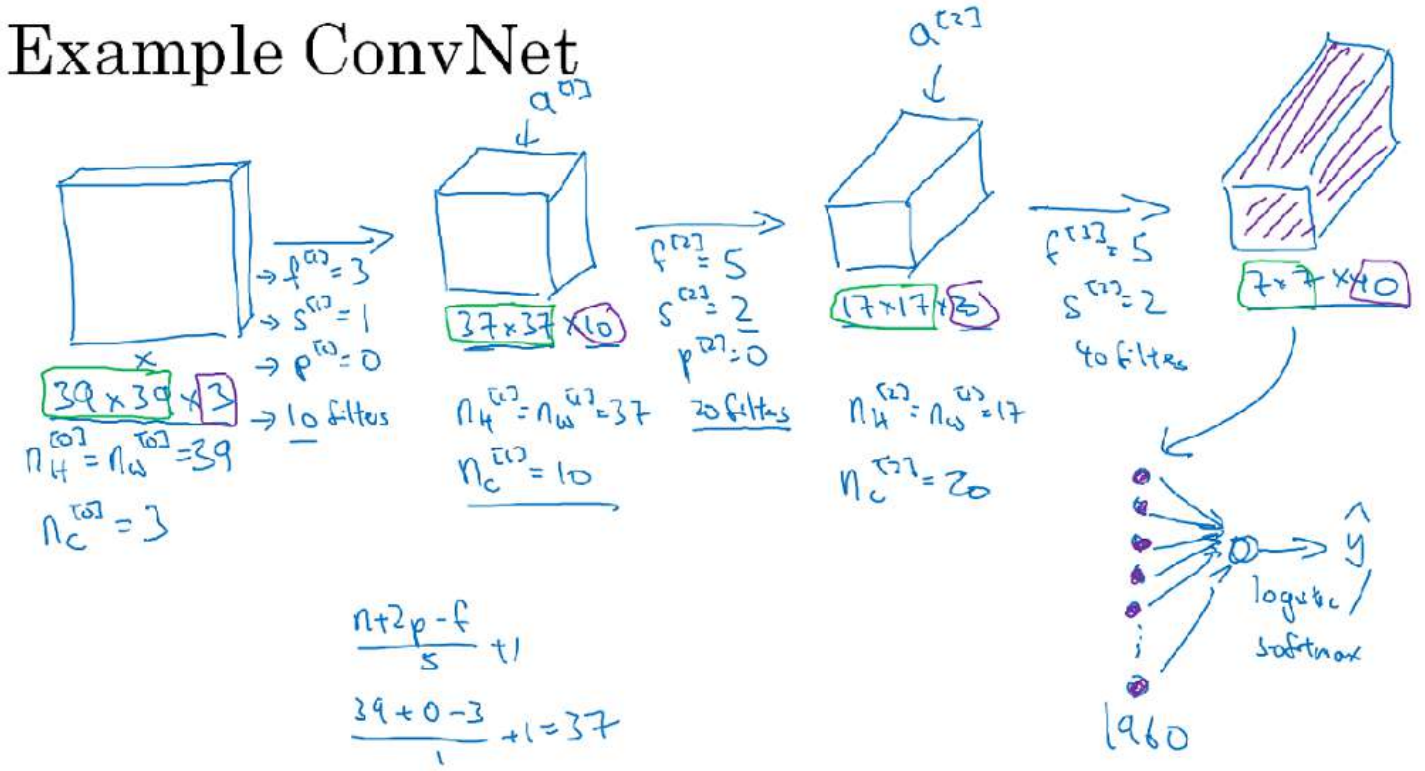
Summary of notation



Simple Convolutions Network Example

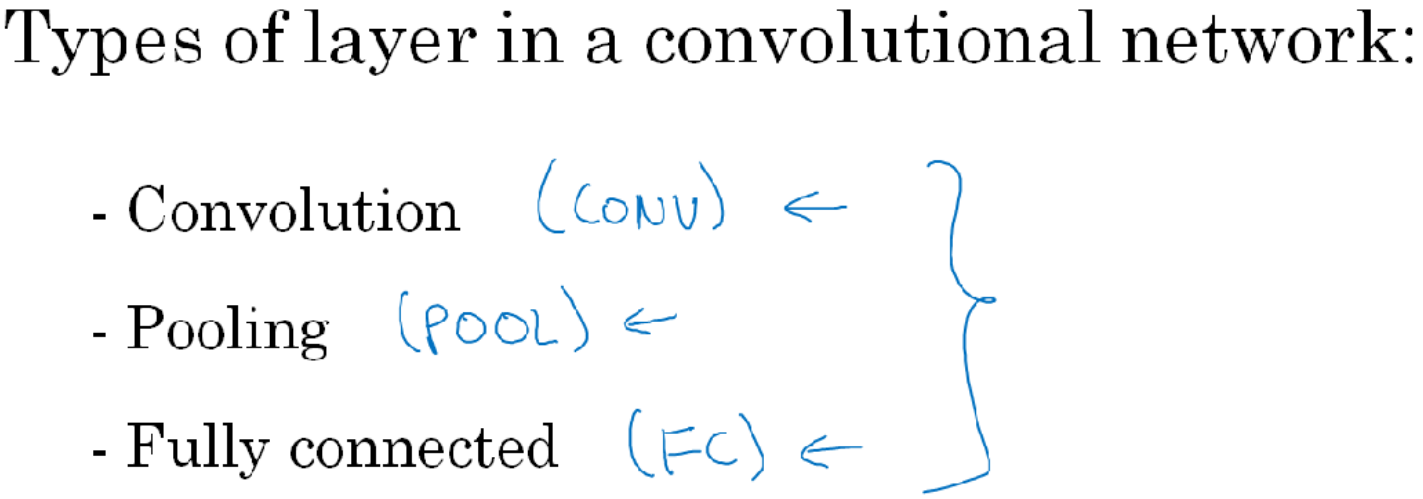
In the last section, we saw the building blocks of a single layer, of a single convolution layer in the ConvNet. Now let's go through a concrete example of a deep convolutional neural network and this will give you some practice with the notation that we introduced toward the end of the last video as well.

Let's say you have an image, and you want to do image classification, or image recognition. Where you want to take as input an image, x, and decide is this a cat or not, 0 or 1, so it's a classification problem. Let's build an example of a ConvNet you could use for this task. For the sake of this example, I'm going to use a fairly small image. Let's say this image is 39 x 39 x 3. This choice just makes some of the numbers work out a bit better. And so, nH in layer 0 will be equal to nw height and width are equal to 39 and the number of channels and layer 0 is equal to 3. Let's say the first layer uses a set of 3 by 3 filters to detect features, so f = 3 or really f1 = 3, because we're using a 3 by 3 process. And let's say we're using a stride of 1, and no padding. So using a same convolution, and let's say you have 10 filters.



Then the activations in this next layer of the neutral network will be 37 x 37 x 10, and this 10 comes from the fact that you use 10 filters. And 37 comes from this formula n + 2p- f over s + 1. Right, then I guess you have 39 + 0- 3 over 1 + 1 that's = to 37. So that's why the output is 37 by 37, it's a valid convolution and that's the output size. So in our notation you would have nh[1] = nw[1] = 37 and nc[1] = 10, so nc[1] is also equal to the number of filters from the first layer. And so this becomes the dimension of the activation at the first layer. Let's say you now have another convolutional layer and let's say this time you use 5 by 5 filters. So, in our notation f[2] at the next neural network = 5, and let's say use a stride of 2 this time. And maybe you have no padding and say, 20 filters. So then the output of this will be another volume, this time it will be 17 x 17 x 20. Notice that, because you're now using a stride of 2, the dimension has shrunk much faster. 37 x 37 has gone down in size by slightly more than a factor of 2, to 17 x 17. And because you're using 20 filters, the number of channels now is 20. So it's this activation a2 would be that dimension and so nh[2] = nw[2] = 17 and nc[2] = 20. All right, let's apply one last convolutional layer. So let's say that you use a 5 by 5 filter again, and again, a stride of 2. So if you do that, I'll skip the math, but you end up with a 7 x 7, and let's say you use 40 filters, no padding, 40 filters. You end up with 7 x 7 x 40. So now what you've done is taken your 39 x 39 x 3 input image and computed your 7 x 7 x 40 features for this image. And then finally, what's commonly done is if you take this 7 x 7 x 40, 7 times 7 times 40 is actually 1,960. And so what we can do is take this volume and flatten it or unroll it into just 1,960 units, right? Just flatten it out into a vector, and then feed this to a **logistic regression unit, or a softmax** unit. Depending on whether you're trying to recognize or trying to recognize any one of key different objects and then just have this give the final predicted output for the neural network. So just be clear, this last step is just taking all of these numbers, all 1,960 numbers, and unrolling them into a very long vector. So then you just have one long vector that you can feed into softmax until it's just a regression in order to make prediction for the final output. So this would be a pretty typical example of a ConvNet. A lot of the work in designing convolutional neural net is selecting hyperparameters like these, deciding what's the total size? What's the stride? What's the padding and how many filters are used? and both later this week as well as next week, we'll give some suggestions and some guidelines on how to make these choices. But for now, maybe one thing to take away from this is that as you go deeper in a neural network, typically you start off with larger images, 39 by 39. And then the height and width will stay the same for a while and gradually trend down as you go deeper in the neural network. It's gone from 39 to 37 to 17 to 14. Excuse me, it's gone from 39 to 37 to 17 to

7. Whereas the number of channels will generally increase. It's gone from 3 to 10 to 20 to 40, and you see this general trend in a lot of other convolutional neural networks as well. So we'll get more guidelines about how to design these parameters in later sections. So it turns out that in a typical ConvNet, there are usually three types of layers. One is the **convolutional layer**, and often we'll denote that as a **Conv layer** and that's what we've been using in the previous network. It turns out that there are two other common types of layers that you haven't seen yet but we'll talk about in the next couple of videos. One is called a pooling layer, often I'll call this pool. And then the last is a fully connected layer called FC. And although it's possible to design a pretty good neural network using just convolutional layers, most neural network architectures will also have a few pooling layers and a few fully connected layers.

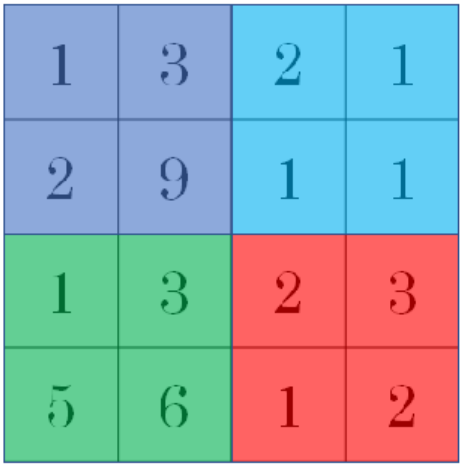


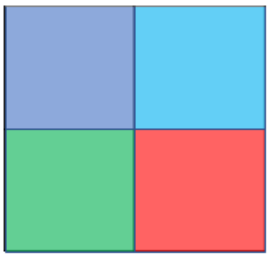
Fortunately pooling layers and fully connected layers are a bit simpler than convolutional layers to define. So we'll do that quickly in the next two videos and then you have a sense of all of the most common types of layers in a convolutional neural network. And you will put together even more powerful networks than the one we just saw.

Pooling Layers

Other than convolutional layers, ConvNets often also use pooling layers to reduce the size of the representation, to speed the computation, as well as make some of the features that detects a bit more robust. Let's take a look. Let's go through an example of pooling, and then we'll talk about why you might want to do this. Suppose you have a four by four input, and you want to apply a type of pooling called max pooling. And the output of this particular implementation of max pooling will be a two by two output. And the way you do that is quite simple.

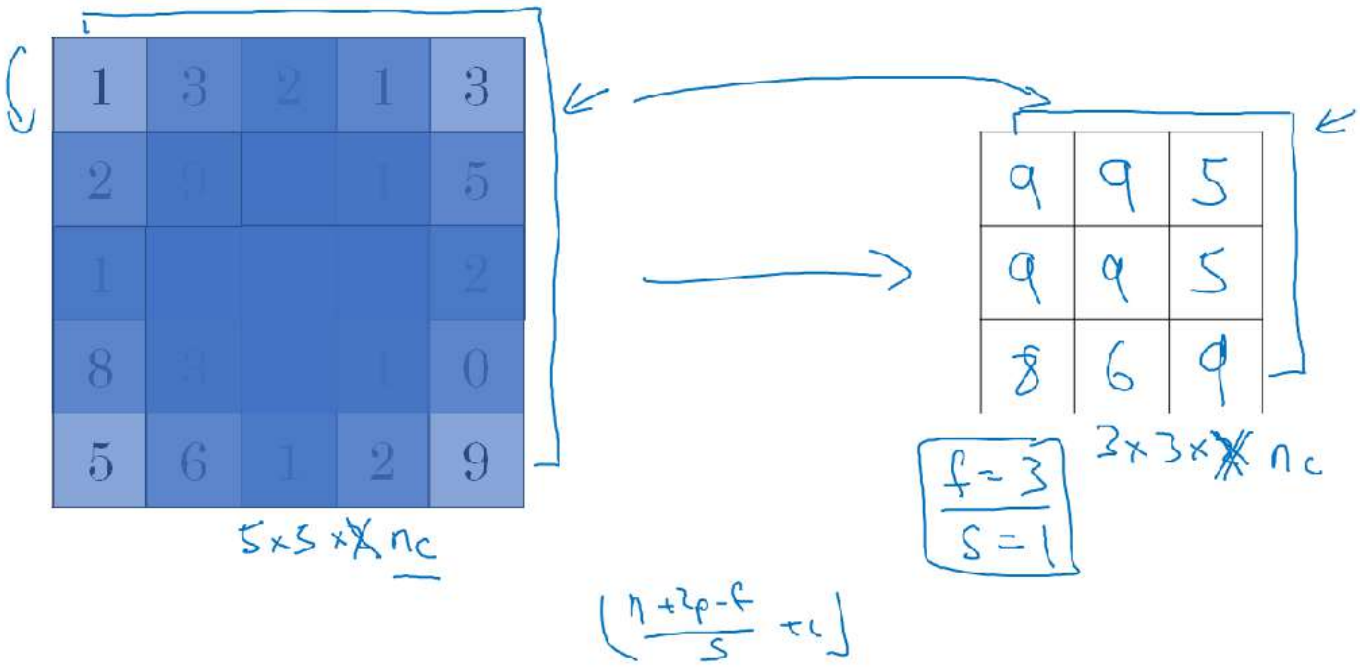
Pooling layer: Max pooling





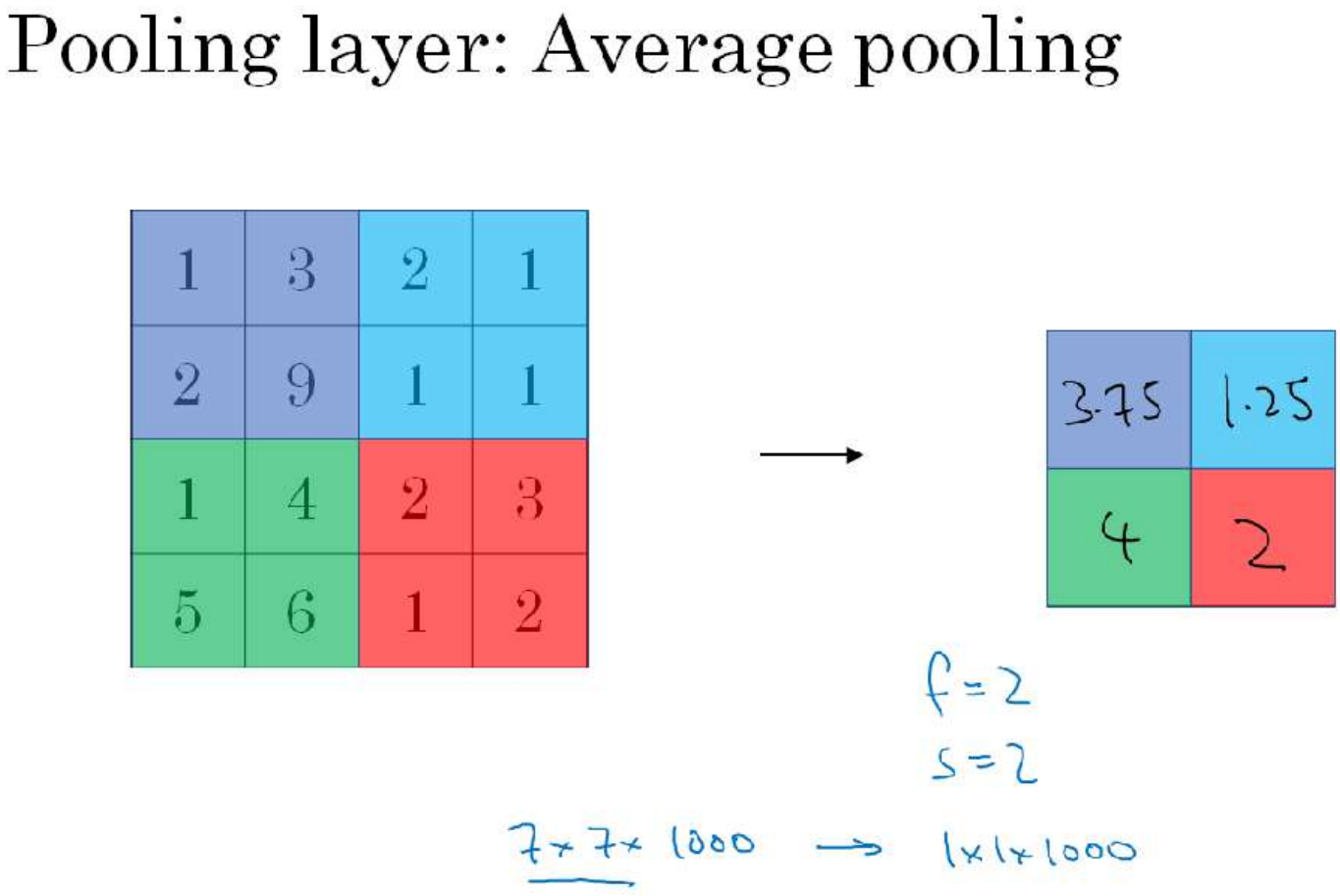
Take your four by four input and break it into different regions and I'm going to color the four regions as follows. And then, in the output, which is two by two, each of the outputs will just be the max from the corresponding reshaded region. So the upper left, I guess, the max of these four numbers is nine. On upper right, the max of the blue numbers is two. Lower left, the biggest number is six, and lower right, the biggest number is three. So to compute each of the numbers on the right, we took the max over a two by two regions.

Pooling layer: Max pooling

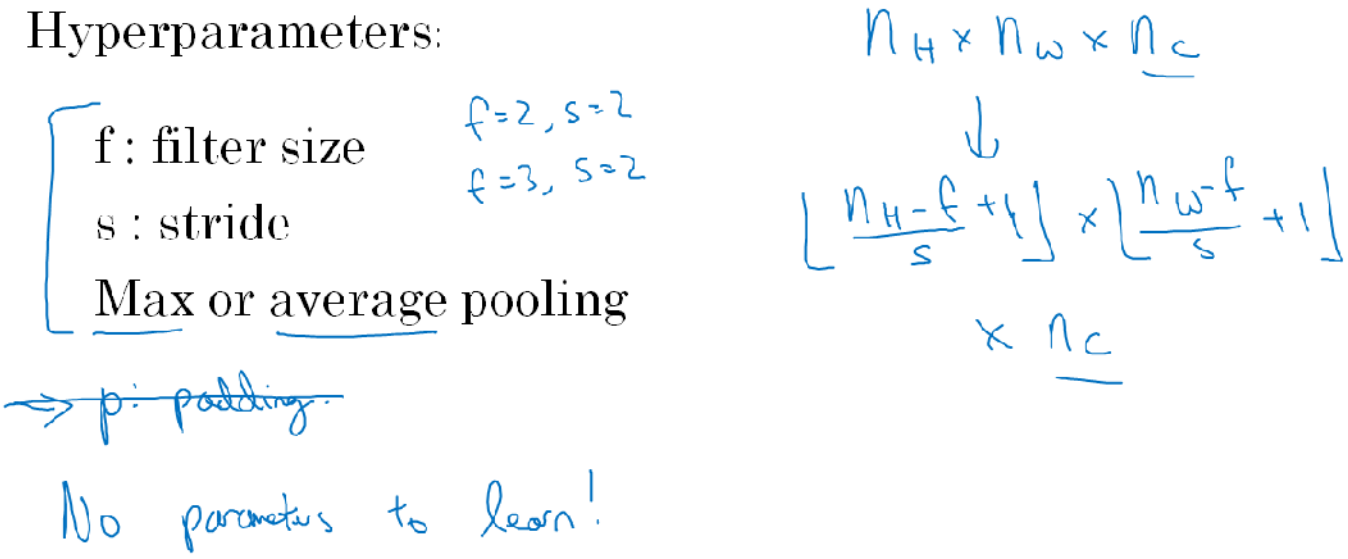


So, this is as if you apply a filter size of two because you're taking a two by two regions and you're taking a stride of two. So, these are actually the hyperparameters of max pooling because we start from this filter size. It's like a two by two region that gives you the nine. And then, you step all over two steps to look at this region, to give you the two, and then for the next row, you step it

down two steps to give you the six, and then step to the right by two steps to give you three. So because the squares are two by two, f is equal to two, and because you stride by two, s is equal to two. So here's the intuition behind what max pooling is doing. 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. I don't have anyone knows if that's the real underlying reason that max pooling works well in ConvNets. **One interesting property of max pooling is that it has a set of hyperparameters but it has no parameters to learn. There's actually nothing for gradient descent to learn**. Once you fix f and s, it's just a fixed computation and gradient descent doesn't change anything.



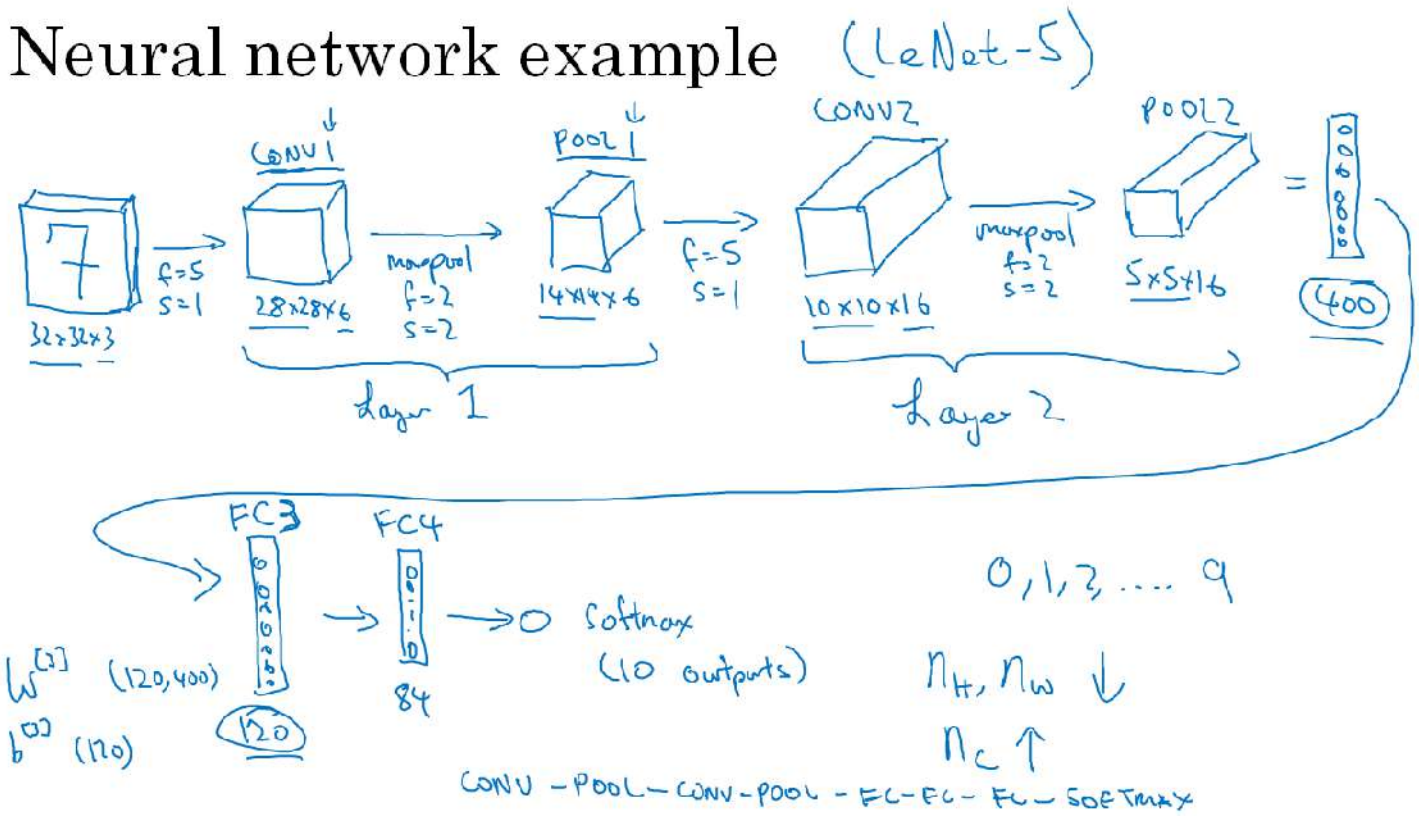
Summary of pooling



CNN Example

You now know pretty much all the building blocks of building a full convolutional neural network. Let's look at an example. Let's say you're inputting an image which is 32 x 32 x 3, so it's an RGB image and maybe you're trying to do handwritten digit recognition. So you have a number like 7 in a 32 x 32 RGB initiate trying to recognize which one of the 10 digits from zero to nine is this. Let's throw the neural network to do this. And what I'm going to use in this slide is inspired, it's actually quite similar to one of the classic neural networks called LeNet-5, which is created by Yann LeCun many years ago. What I'll show here isn't exactly LeNet-5 but it's inspired by it, but many parameter choices were inspired by it. So with a 32 x 32 x 3 input let's say that the first layer uses a 5 x 5 filter and a stride of 1, and no padding. So the output of this layer, if you use 6 filters would be 28 x 28 x 6, and we're going to call this layer conv 1. So you apply 6 filters, add a bias, apply the non-linearity, maybe a real non-linearity, and that's the conv 1 output. Next, let's apply a pooling layer, so I am going to apply mass pooling here and let's use a f=2, s=2. When I don't write a padding use a pad easy with a 0. Next let's apply a pooling layer, I am going to apply, let's see max pooling with a 2 x 2 filter and the stride equals 2. So this is should reduce the height and width of the representation by a factor of 2. So 28 x 28 now becomes 14 x 14, and the number of channels remains the same so 14 x 14 x 6, and we're going to call this the Pool 1 output. So, it turns out that in the literature of a ConvNet there are two conventions which are inside the inconsistent about what you call a layer. One convention is that this is called one layer. So this will be layer one of the neural network, and now the conversion will be to call they convey layer as a layer and the pool layer as a layer. When people report the number of layers in a neural network usually people just record the number of layers that have weight, that have parameters. And because the pooling layer has no weights, has no parameters, only a few hyper parameters, I'm going to use a convention that Conv 1 and Pool 1 shared together. I'm going to treat that as Layer 1, although sometimes you see people maybe read articles online and read research papers, you hear about the conv layer and the pooling layer as if they are two separate layers. But this is maybe two slightly inconsistent notation terminologies, but when I count layers, I'm just going to count layers that have weights. So achieve both of this together as Layer 1. And the name Conv1 and Pool1 use here the 1 at the end also refers the fact that I view both of this is part of Layer 1 of the neural network. And Pool 1 is grouped into Layer 1 because it doesn't have its own weights. Next, given a 14 x 14 bx 6 volume, let's apply another convolutional layer to it, let's use a filter size that's 5 x 5, and let's use a stride of 1, and let's use 10 filters this time. So now you end up with, A 10 x 10 x 10 volume, so I'll call this Comv 2, and then in this network let's do max pulling

with f=2, s=2 again. So you could probably guess the output of this, f=2, s=2, this should reduce the height and width by a factor of 2, so you're left with 5 x 5 x 10. And so I'm going to call this Pool 2, and in our convention this is Layer 2 of the neural network. Now let's apply another convolutional layer to this. I'm going to use a 5 x 5 filter, so f = 5, and let's try this, 1, and I don't write the padding, means there's no padding. And this will give you the Conv 2 output, and that's your 16 filters. So this would be a 10 x 10 x 16 dimensional output. So we look at that, and this is the Conv 2 layer. And then let's apply max pooling to this with f=2, s=2. You can probably guess the output of this, we're at 10 x 10 x 16 with max pooling with f=2, s=2. This will half the height and width, you can probably guess the result of this, right? Left pooling with f = 2, s = 2. This should halve the height and width so you end up with a 5 x 5 x 16 volume, same number of channels as before. We're going to call this Pool 2. And in our convention this is Layer 2 because this has one set of weights and your Conv 2 layer. Now 5 x 5 x 16, 5 x 5 x 16 is equal to 400. So let's now fatten our Pool 2 into a 400 x 1 dimensional vector. So think of this as fatting this up into these set of neurons, like so. And what we're going to do is then take these 400 units and let's build the next layer, As having 120 units. So this is actually our first fully connected layer. I'm going to call this FC3 because we have 400 units densely connected to 120 units.



So this fully connected unit, this fully connected layer is just like the single neural network layer that you saw in Courses 1 and 2. This is just a standard neural network where you have a weight matrix that's called W3 of dimension 120 x 400. And this is fully connected because each of the 400 units here is connected to each of the 120 units here, and you also have the bias parameter, yes that's going to be just a 120 dimensional, this is 120 outputs. And then lastly let's take 120 units and add another layer, this time smaller but let's say we had 84 units here, I'm going to call this fully connected Layer 4. And finally we now have 84 real numbers that you can fit to a [INAUDIBLE] unit. And if you're trying to do handwritten digital recognition, to recognize this hand it is 0, 1, 2, and so on up to 9. Then this would be a softmax with 10 outputs. So this is a vis-a-vis typical example of what a convolutional neural network might look like. And I know this seems like there a lot of hyper parameters. We'll give you some more specific suggestions later for how to choose these types of hyper parameters. Maybe one common guideline is to actually not try to invent your own settings of hyper parameters, but to look in the literature to see what hyper parameters you work for others. And to just choose an architecture that has worked well for someone else, and there's a chance that will work for your application as well. We'll see more

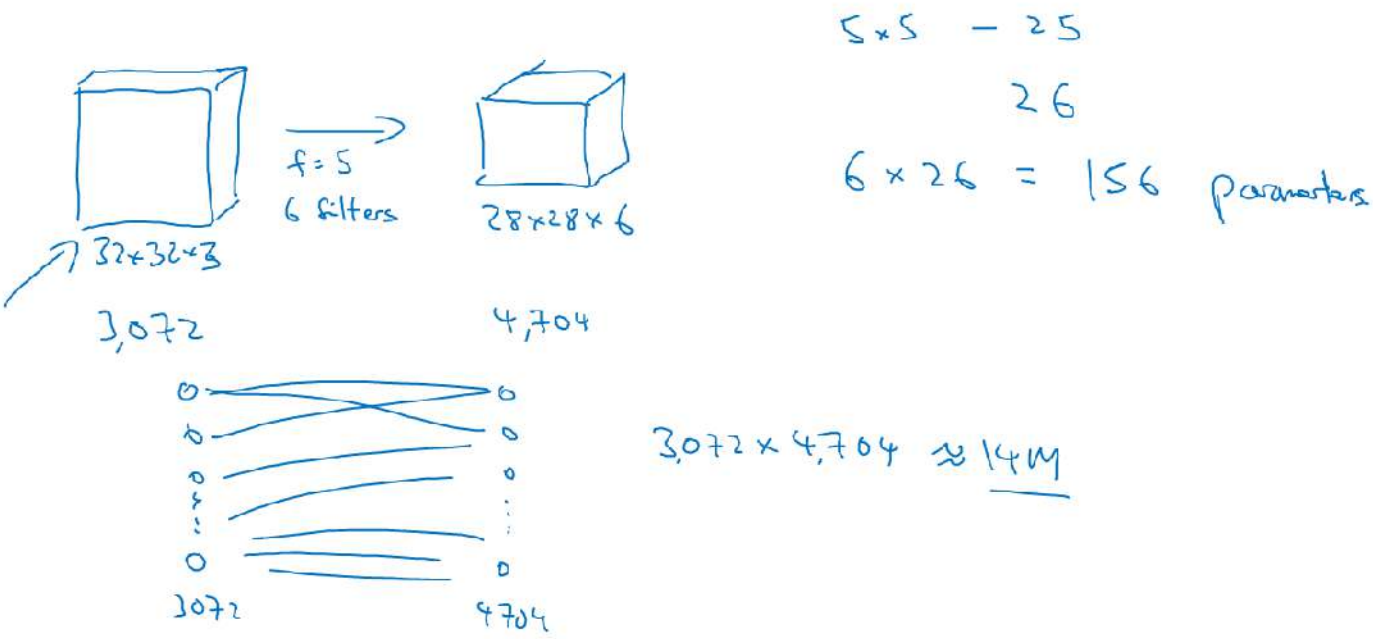
about that next week. But for now I'll just point out that as you go deeper in the neural network, usually nh and nw to height and width will decrease. Pointed this out earlier, but it goes from 32 x 32, to 20 x 20, to 14 x 14, to 10 x 10, to 5 x 5. So as you go deeper usually the height and width will decrease, whereas the number of channels will increase. It's gone from 3 to 6 to 16, and then your fully connected layer is at the end. And another pretty common pattern you see in neural networks is to have conv layers, maybe one or more conv layers followed by a pooling layer, and then one or more conv layers followed by pooling layer. And then at the end you have a few fully connected layers and then followed by maybe a softmax. And this is another pretty common pattern you see in neural networks. So let's just go through for this neural network some more details of what are the activation shape, the activation size, and the number of parameters in this network. So the input was 32 x 30 x 3, and if you multiply out those numbers you should get 3,072. So the activation, a0 has dimension 3072. Well it's really 32 x 32 x 3. And there are no parameters I guess at the input layer. And as you look at the different layers, feel free to work out the details yourself. These are the activation shape and the activation sizes of these different layers.

Why Convolutions?

For this final section for this week, let's talk a bit about why convolutions are so useful when you include them in your neural networks. And then finally, let's briefly talk about how to put this all together and how you could train a convolution neural network when you have a label training set.

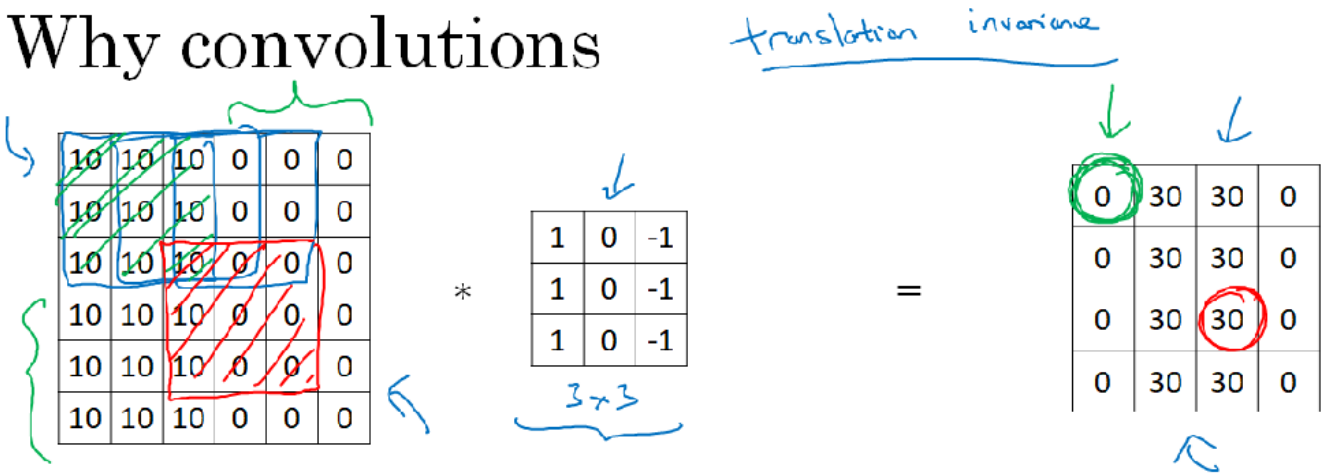
I think there are two main advantages of convolutional layers over just using fully connected layers and the advantages are **parameter sharing and sparsity of connections**. Let me illustrate with an example. Let's say you have a 32x32x3 dimensional image, and this actually comes from the example from the previous section, but let's say you use five by five filter with six filters and so, this gives you a 28 by 28 by 6 dimensional output. So, 32 by 32 by 3 is 3,072, and 28 by 28 by 6 if you multiply all those numbers is 4,704. And so, if you were to create a neural network with 3,072 units in one layer, and with 4,704 units in the next layer, and if you were to connect every one of these neurons, then the weight matrix, the number of parameters in a weight matrix would be 3,072 times 4,704 which is about 14 million. So, that's just a lot of parameters to train. And today you can train neural networks with even more parameters than 14 million, but considering that this is just a pretty small image, this is a lot of parameters to train. And of course, if this were to be 1,000 by 1,000 image, then your display matrix will just become invisibly large. But if you look at the number of parameters in this convolutional layer, each filter is five by five. So, each filter has 25 parameters, plus a bias parameter miss of 26 parameters per a filter, and you have six filters, so, the total number of parameters is that, which is equal to 156 parameters. And so, the number of parameters in this conv layer remains quite small. And the reason that a consonant has run to these small parameters is really two reasons.

Why convolutions



One is parameter sharing. And parameter sharing is motivated by the observation that feature detector such as vertical edge detector, that's useful in one part of the image is probably useful in another part of the image. And what that means is that, if you've figured out say a three by three filter for detecting vertical edges, you can then apply the same three by three filter over here, and then the next position over, and the next position over, and so on. And so, each of these feature detectors, each of these aqua's can use the same parameters in lots of different positions in your input image in order to detect say a vertical edge or some other feature. And I think this is true for low-level features like edges, as well as the higher level features, like maybe, detecting the eye that indicates a face or a cat or something there. But being with a share in this case the same nine parameters to compute all 16 of these aquas, is one of the ways the number of parameters is reduced. And it also just seems intuitive that a feature detector like a vertical edge detector computes it for the upper left-hand corner of the image. The same feature seems like it will probably be useful, has a good chance of being useful for the lower right-hand corner of the image. So, maybe you don't need to learn separate feature detectors for the upper left and the lower right-hand corners of the image. And maybe you do have a dataset where you have the upper left-hand corner and lower right-hand corner have different distributions, so, they maybe look a little bit different but they might be similar enough, they're sharing feature detectors all across the image, works just fine. The second way that consonants get away with having relatively few parameters is by having sparse connections. So, here's what I mean, if you look at the zero, this is computed via three by three convolution. And so, it depends only on this three by three inputs grid or cells. So, it is as if this output units on the right is connected only to nine out of these six by six, 36 input features. And in particular, the rest of these pixel values, all of these pixel values do not have any effects on the other output. So, that's what I mean by sparsity of connections.

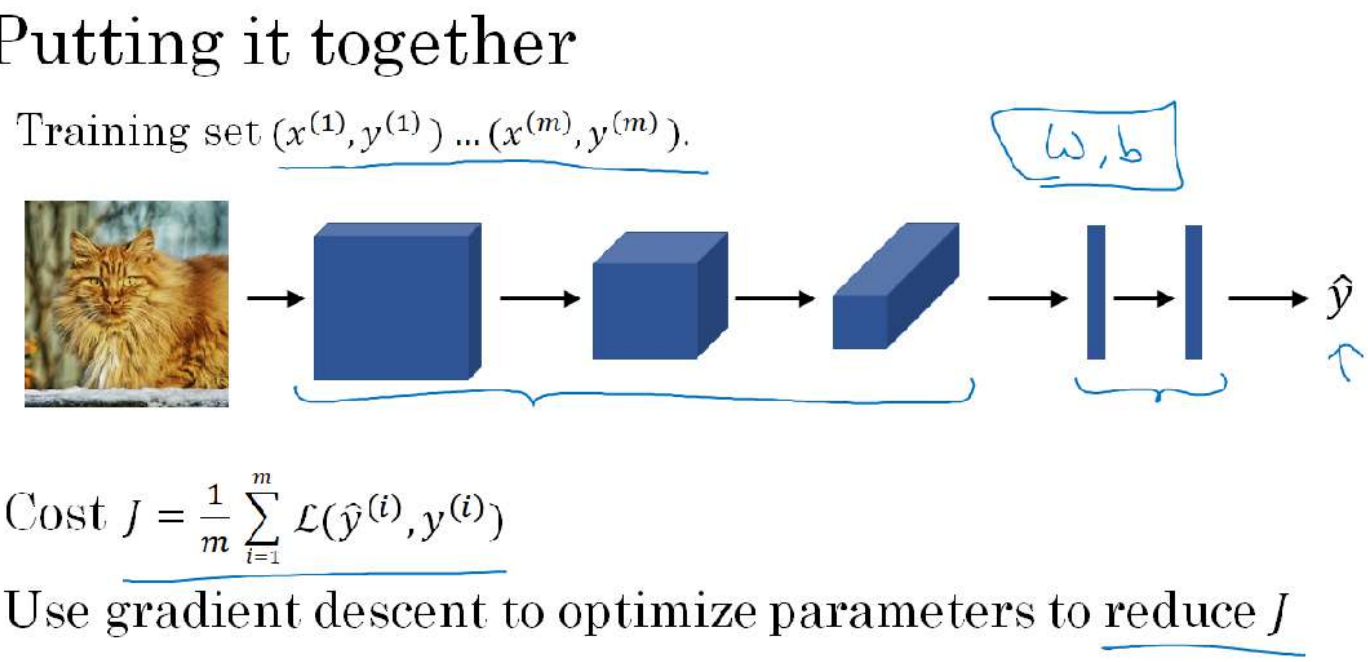
Why convolutions



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.

As another example, this output depends only on these nine input features. And so, it's as if only those nine input features are connected to this output, and the other pixels just don't affect this output at all. And so, through these two mechanisms, a neural network has a lot fewer parameters which allows it to be trained with smaller training cells and is less prone to be over 30. And so, sometimes you also hear about convolutional neural networks being very good at capturing translation invariance. And that's the observation that a picture of a cat shifted a couple of pixels to the right, is still pretty clearly a cat. And convolutional structure helps the neural network encode the fact that an image shifted a few pixels should result in pretty similar features and should probably be assigned the same oval label. And the fact that you are applying to same filter, knows all the positions of the image, both in the early layers and in the late layers that helps a neural network automatically learn to be more robust or to better capture the desirable property of translation invariance. So, these are maybe a couple of the reasons why convolutions or convolutional neural network work so well in computer vision. Finally, let's put it all together and see how you can train one of these networks. Let's say you want to build a cat detector and you have a labeled training sets as follows, where now, X is an image. And the y's can be binary labels, or one of K causes. And let's say you've chosen a convolutional neural network structure, may be inserted the image and then having neural convolutional and pulling layers and then some fully connected layers followed by a software output that then operates Y hat. The conv layers and the fully connected layers will have various parameters, W, as well as bias's B. And so, any setting of the parameters, therefore, lets you define a cost function similar to what we have seen in the previous courses, where we've randomly initialized parameters W and B. You can compute the cause J, as the sum of losses of the neural networks predictions on your entire training set, maybe divide it by M. So, to train this neural network, all you need to do is then use gradient descents or some of the algorithm like, gradient descent momentum, or RMSProp or Adam, or something else, in order to optimize all the parameters of the neural network to try to reduce the cost function J. And you find that if you do this, you can build a very effective cat detector or some other detector.



Week 2: Deep convolutional models: case studies

Learn about the practical tricks and methods used in deep CNNs straight from the research papers.

Learning Objectives

- Understand multiple foundational papers of convolutional neural networks
- Analyze the dimensionality reduction of a volume in a very deep network
- Understand and Implement a Residual network
- Build a deep neural network using Keras
- Implement a skip-connection in your network
- Clone a repository from github and use transfer learning

Case studies

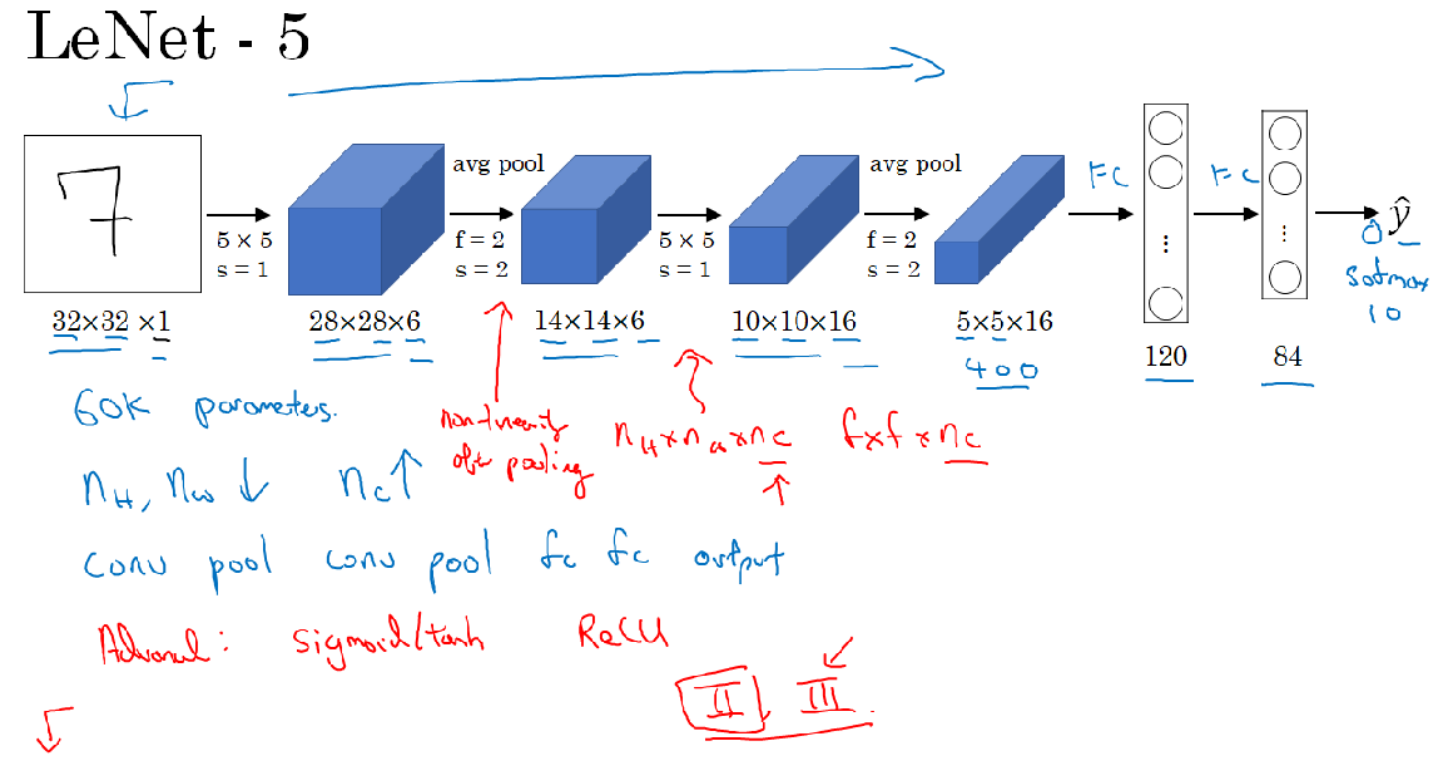
Why look at case studies?

In this section the first thing we'll do is show you a number of case studies of the factor convolutional neural networks. So why look at case studies? In last section we learned about the basic building blocks such as convolutional layers, pooling layers and fully connected layers of conv nets. It turns out a lot of the past few years of computer vision research has been on how to put together these basic building blocks to form effective convolutional neural networks and one of the best ways for you to get intuition yourself is to see some of these examples. I think just as many of you may have learned to write codes by reading other people's codes, I think that a good way to get intuition on how to build conv nets is to read or to see other examples of effective conv nets and it turns out that a neural network architecture that works well on one computer vision task often works well on other tasks as well such as maybe on your task. So if someone else is training neural network as speak it out in your network architecture is very good at recognizing cats and dogs and people but you have a different computer vision task like maybe you're trying to sell self-driving car. You might well be able to take someone else's neural network architecture and apply that to your problem. Also, in upcoming sections we'll be able to read some of the research papers from the computer vision and I hope that you might find it satisfying as well. You don't have to do this as a class but I hope you might find it satisfying to be able to read some of these seminal computer vision research paper and see yourself able to understand them. So with that, let's get started. As an outline of what we'll do in the next few sections, we'll first show you a few classic networks. The LeNet-5 network which came from, I guess, in 1980s, AlexNet which is often cited and the VGG network and these are examples of pretty effective neural networks and

some of the ideas lay the foundation for modern computer vision. Then we'll see the ResNet or conv residual network and you might have heard that neural networks are getting deeper and deeper. The ResNet neural network trained a very, very deep 152-layer neural network that has some very interesting tricks, interesting ideas how to do that effectively and then finally you also see a case study of the Inception neural network. After seeing these neural networks, I think you have much better intuition about how to built effective convolutional neural networks and even if you end up not working computer vision yourself, I think you find a lot of the ideas from some of these examples, such as ResNet Inception network, many of these ideas are cross-fertilizing on making their way into other disciplines. So even if you don't end up building computer vision applications yourself, I think you'll find some of these ideas very interesting and helpful for your work.

Classic Networks

In this section, we'll learn about some of the classic neural network architecture starting with LeNet-5, and then AlexNet, and then VGGNet. Let's take a look. Here is the LeNet-5 architecture.

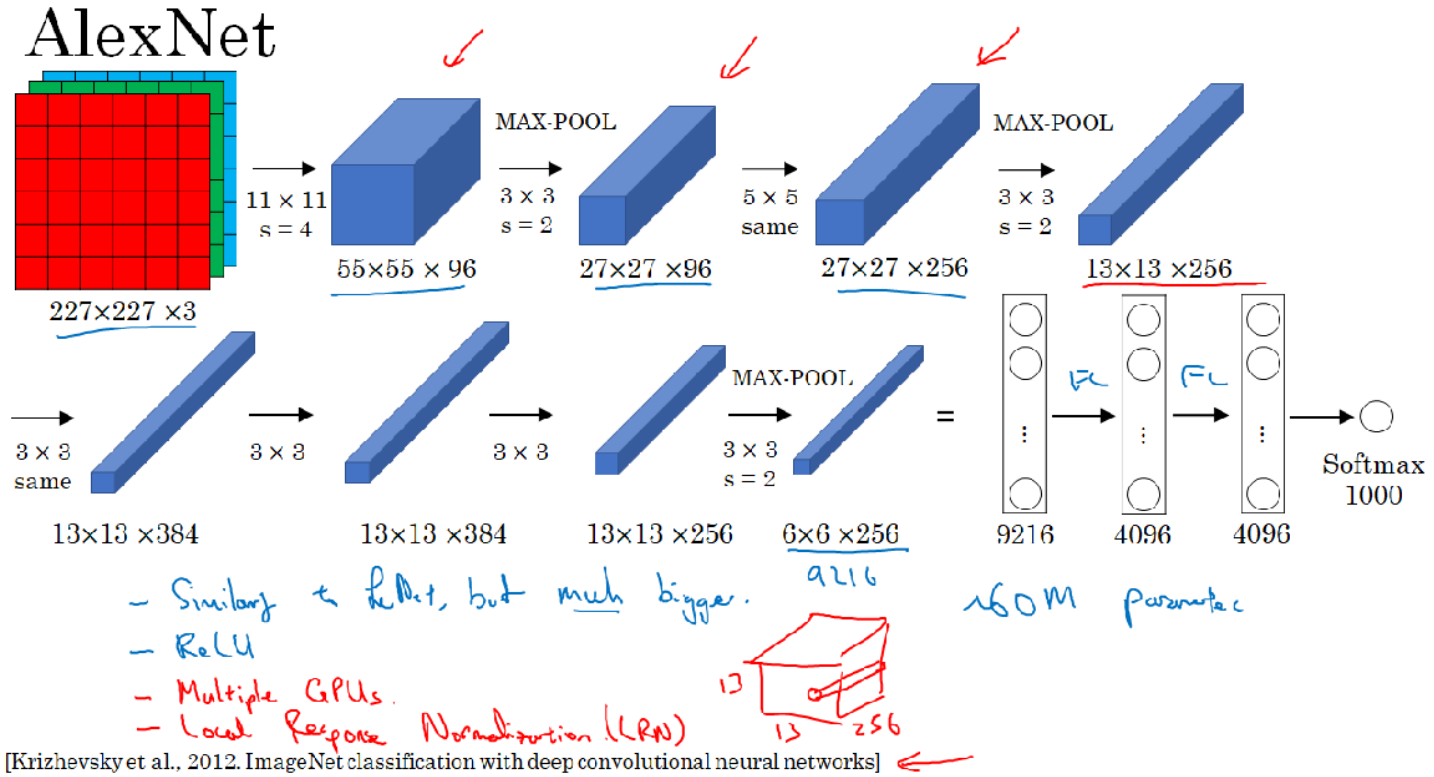


[LeCun et al., 1998. Gradient-based learning applied to document recognition]

You start off with an image which say, 32x32x1 and the goal of LeNet-5 was to **recognize handwritten digits**, so maybe an image of a digits like shown in diagram and **LeNet-5** was trained on grayscale images, which is why it's 32x32x1. In the first step, you use a set of six, 5x5 filters with a stride of one because you use six filters you end up with a 20x20x6 over there and with a stride of one and no padding, the image dimensions reduces from 32x32 down to 28x28. Then the LeNet neural network applies pooling and back then when this paper was written, people use average pooling much more. If you're building a modern variant, you probably use max pooling instead. But in this example, you average pool and with a filter width two and a stride of two, you wind up reducing the dimensions, the height and width by a factor of two, so we now end up with a 14x14x6 volume. I guess the height and width of these volumes aren't entirely drawn to scale. Now technically, if I were drawing these volumes to scale, the height and width would be stronger by a factor of two. Next, you apply another convolutional layer. This time you use a set of 16 filters, the 5x5, so you end up with 16 channels to the next volume and back when this paper was written in 1998, people didn't really use padding or you always using valid convolutions, which is why every time you apply convolutional layer, they heightened with strengths. So that's why, here, you go from 14x14 down to 10x10. Then another pooling layer, so that reduces the height

and width by a factor of two, then you end up with 5x5 over there and if you multiply all these numbers 5x5x16, this multiplies up to 400. That's 25 times 16 is 400. And the next layer is then a fully connected layer that fully connects each of these 400 nodes with every one of 120 neurons, so there's a fully connected layer and sometimes, that would draw out exclusively a layer with 400 nodes. There's a fully connected layer and then another a fully connected layer. And then the final step is it uses these essentially 84 features and uses it with one final output. I guess you could draw one more node here to make a prediction for ŷ. And ŷ took on 10 possible values corresponding to recognising each of the digits from 0 to 9. A modern version of this neural network, we'll use a **softmax** layer with a 10 way classification output. Although back then, LeNet-5 actually use a different classifier at the output layer, one that's useless today. So this neural network was small by modern standards, had about 60,000 parameters. And today, you often see neural networks with anywhere from 10 million to 100 million parameters, and it's not unusual to see networks that are literally about a thousand times bigger than this network. But one thing you do see is that as you go deeper in a network, so as you go from left to right, the height and width tend to go down. So you went from 32 by 32, to 28 to 14, to 10 to 5, whereas the number of channels does increase. It goes from 1 to 6 to 16 as you go deeper into the layers of the network. One other pattern you see in this neural network that's still often repeated today is that you might have some one or more conv layers followed by pooling layer, and then one or sometimes more than one conv layer followed by a pooling layer, and then some fully connected layers and then the outputs. So this type of arrangement of layers is quite common. Now finally, this is maybe only for those of you that want to try reading the paper. So it turns out that if you read the original paper, back then, people used sigmoid and tanh nonlinearities, and people weren't using value nonlinearities back then. So if you look at the paper, you see sigmoid and tanh referred to and there are also some funny ways about this network was wired that is funny by modern standards. So for example, you've seen how if you have a nh by nw by nc network with nc channels then you use f by f by nc dimensional filter, where everything looks at every one of these channels. But back then, computers were much slower. And so to save on computation as well as some parameters, the original LeNet-5 had some crazy complicated way where different filters would look at different channels of the input block. And so the paper talks about those details, but the more modern implementation wouldn't have that type of complexity these days. And then one last thing that was done back then I guess but isn't really done right now is that the original LeNet-5 had a non-linearity after pooling, and I think it actually uses sigmoid non-linearity after the pooling layer.

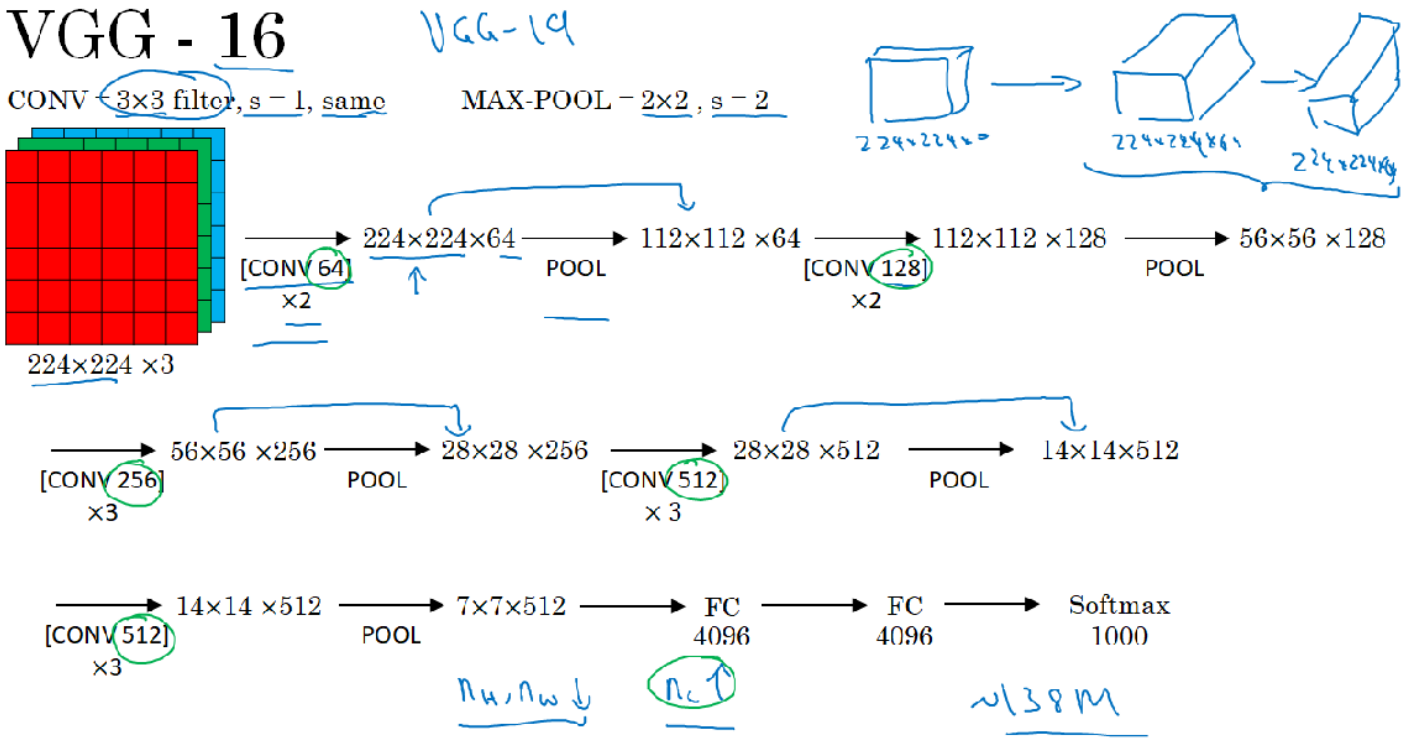
The second example of a neural network I want to show you is **AlexNet**, named after Alex Krizhevsky, who was the first author of the paper describing this work. The other author's were Ilya Sutskever and Geoffrey Hinton.



So, AlexNet input starts with 227x227x3 images and if you read the paper, the paper refers to 224x224x3 images. But if you look at the numbers, I think that the numbers make sense only of actually 227x227 and then the first layer applies a set of 96, 11x11 filters with a stride of four and because it uses a large stride of four, the dimensions shrinks to 55x55. So roughly, going down by a factor of 4 because of a large stride and then it applies max pooling with a 3x3 filter. So f equals three and a stride of two. So this reduces the volume to 27x27x96, and then it performs a 5x5 same convolution, same padding, so you end up with 27x27x276. Max pooling again, this reduces the height and width to 13. And then another same convolution, so same padding. So it's 13 by 13 by now 384 filters and then 3 by 3, same convolution again, gives you that. Then 3 by 3, same convolution, gives you that. Then max pool, brings it down to 6 by 6 by 256. If you multiply all these numbers,6 times 6 times 256, that's 9216. So we're going to unroll this into 9216 nodes. And then finally, it has a few fully connected layers. And then finally, it uses a softmax to output which one of 1000 causes the object could be. So this neural network actually had a lot of similarities to LeNet, but it was much bigger. So whereas the **LeNet-5 from previous slide had about 60,000 parameters, this AlexNet that had about 60 million parameters** and the fact that they could take pretty similar basic building blocks that have a lot more hidden units and training on a lot more data, they trained on the image that dataset that allowed it to have a just remarkable performance. Another aspect of this architecture that made it much better than LeNet was using the value activation function. And then again, just if you read the bay paper some more advanced details that you don't really need to worry about if you don't read the paper, one is that, when this paper was written, GPUs was still a little bit slower, so it had a complicated way of training on two GPUs. And the basic idea was that, a lot of these layers was actually split across two different GPUs and there was a thoughtful way for when the two GPUs would communicate with each other. And the paper also, the original AlexNet architecture also had another set of a layer called a **Local Response Normalization** and this type of layer isn't really used much, which is why I didn't talk about it. But the basic idea of Local Response Normalization is, if you look at one of these blocks, one of these volumes that we have on top, let's say for the sake of argument, this one, 13x13x256, what Local Response Normalization, (LRN) does, is you look at one position. So one position height and width, and look down this across all the channels, look at all 256 numbers and normalize them. And the motivation for this Local Response Normalization was that for each position in this 13x13 image, maybe you don't want too many neurons with a very high

activation. But subsequently, many researchers have found that this doesn't help that much so this is one of those ideas I guess I'm drawing in red because it's less important for you to understand this one. And in practice, I don't really use local response normalizations really in the networks language trained today. So if you are interested in the history of deep learning, I think even before AlexNet, deep learning was starting to gain attraction in speech recognition and a few other areas, but it was really just paper that convinced a lot of the computer vision community to take a serious look at deep learning to convince them that deep learning really works in computer vision. And then it grew on to have a huge impact not just in computer vision but beyond computer vision as well. And if you want to try reading some of these papers yourself and you really don't have to for this course, but if you want to try reading some of these papers, this one is one of the easier ones to read so this might be a good one to take a look at. So whereas AlexNet had a relatively complicated architecture, there's just a lot of hyperparameters, right? Where you have all these numbers that Alex Krizhevsky and his co-authors had to come up with.

Let me show you a third and final example on this section called the **VGG** or **VGG-16** network and a remarkable thing about the VGG-16 net is that they said, instead of having so many hyperparameters, let's use a much simpler network where you focus on just having conv-layers that are just three-by-three filters with a stride of one and always use same padding and make all your max pulling layers two-by-two with a stride of two. And so, one very nice thing about the VGG network was it really simplified this neural network architectures. So, let's go through the architecture.



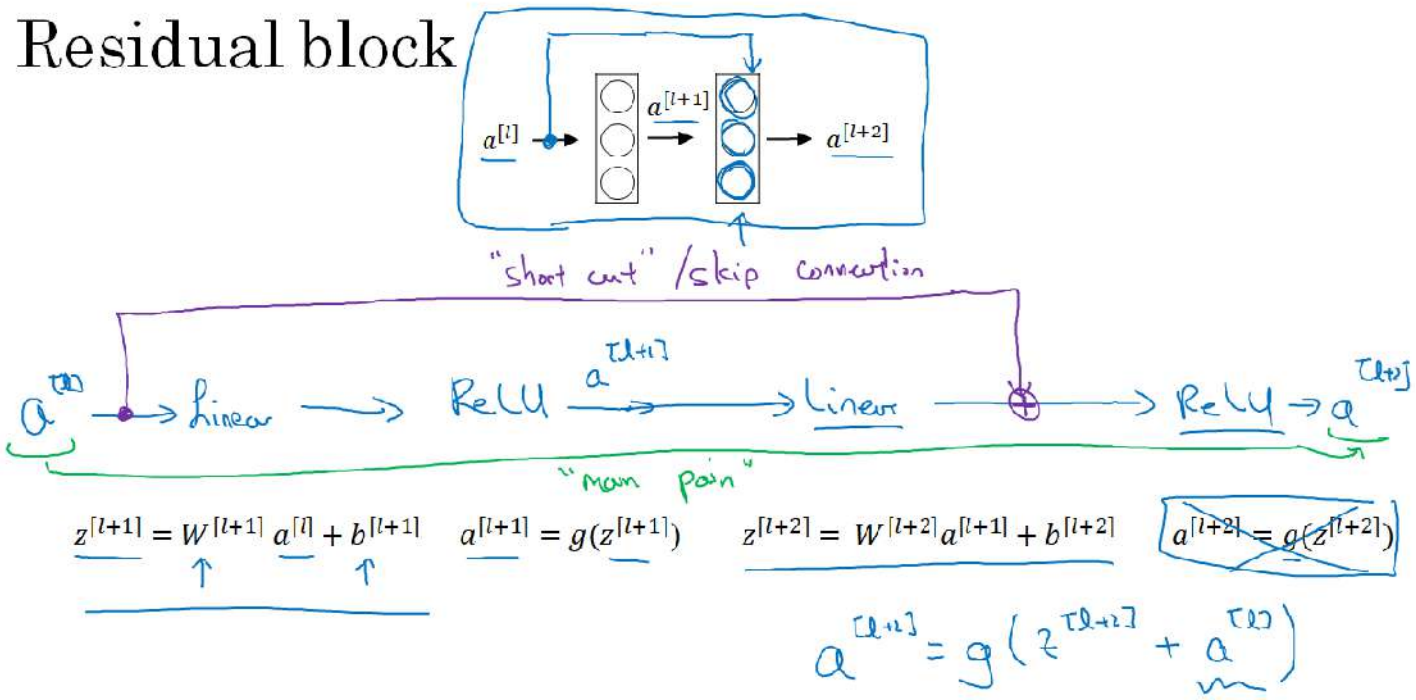
[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

So, you solve up with an image for them and then the first two layers are convolutions, which are therefore these three-by-three filters and in the first two layers use 64 filters. You end up with a 224x224 because using same convolutions and then with 64 channels. So because VGG-16 is a relatively deep network, am going to not draw all the volumes here. So what this little picture denotes is what we would previously have drawn as this 224x224x3 and then a convolution that results in I guess a 224x224x64 is to be drawn as a deeper volume, and then another layer that results in 224x224x64. So this conv64 times two represents that you're doing two conv-layers with 64 filters and as I mentioned earlier, the filters are always three-by-three with a stride of one and they are always same convolutions. So rather than drawing all these volumes, am just going to use text to represent this network. Next, then uses are pooling layer, so the pooling layer will reduce. I think it goes from 224x224 down to what? Right. Goes to 112x112x64 and then it has a

couple more conv-layers. So this means it has 128 filters and because these are the same convolutions, let's see what is the new dimension. Right? It will be 112 by 112 by 128 and then pulling layer so you can figure out what's the new dimension of that. And now, three conv-layers with 256 filters to the pulling layer and then a few more conv-layers, pooling layer, more conv-layers, pooling layer. And then it takes this final 7x7x512 these in to fully connected layer, fully connected with four thousand ninety six units and then a softmax output one of a thousand classes. By the way, the 16 in the VGG-16 refers to the fact that this has 16 layers that have weights and this is a pretty large network, this network has a total of about 138 million parameters. And that's pretty large even by modern standards. But the simplicity of the VGG-16 architecture made it quite appealing. You can tell his architecture is really quite uniform. There is a few conv-layers followed by a pooling layer, which reduces the height and width, right? So the pooling layers reduce the height and width. You have a few of them here. But then also, if you look at the number of filters in the conv-layers, here you have 64 filters and then you double to 128 double to 256 doubles to 512. And then I guess the authors thought 512 was big enough and did double on the game here. But this sort of roughly doubling on every step, or doubling through every stack of conv-layers was another simple principle used to design the architecture of this network. And so I think the relative uniformity of this architecture made it quite attractive to researchers. The main downside was that it was a pretty large network in terms of the number of parameters you had to train. And if you read the literature, you sometimes see people talk about the VGG-19, that is an even bigger version of this network. And you could see the details in the paper cited at the bottom by Karen Simonyan and Andrew Zisserman. But because VGG-16 does almost as well as VGG-19. A lot of people will use VGG-16. But the thing I liked most about this was that, this made this pattern of how, as you go deeper and height and width goes down, it just goes down by a factor of two each time for the pulling layers whereas the number of channels increases. And here roughly goes up by a factor of two every time you have a new set of conv-layers. So by making the rate at which it goes down and that go up very systematic, I thought this paper was very attractive from that perspective. So that's it for the three classic architecture's. If you want, you should really now read some of these papers. I recommend starting with the AlexNet paper followed by the VGG net paper and then the LeNet paper is a bit harder to read but it is a good classic once you go over that. But next, let's go beyond these classic networks and look at some even more advanced, even more powerful neural network architectures.

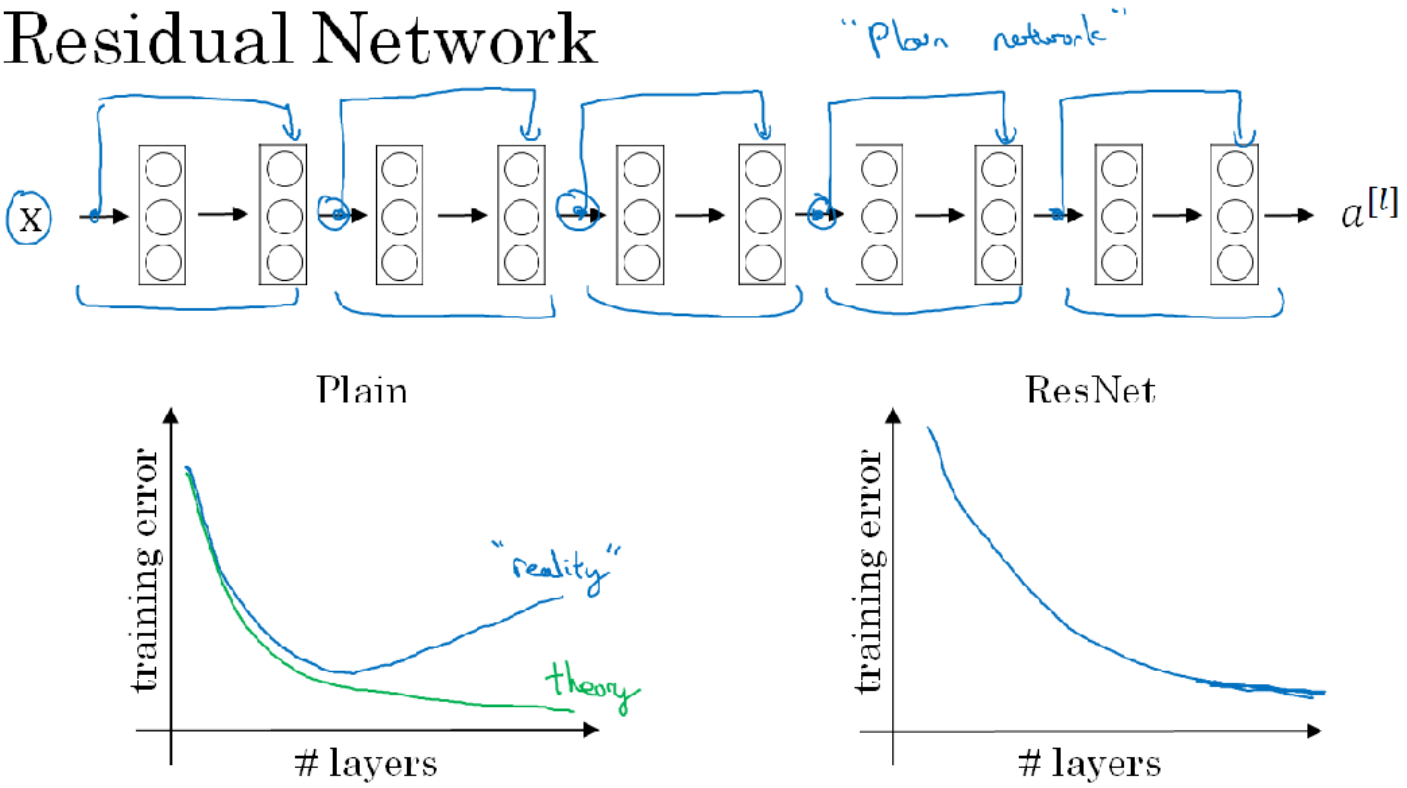
ResNets

Very, very deep neural networks are difficult to train because of vanishing and exploding gradient types of problems. In this section, we'll learn about skip connections which allows you to take the activation from one layer and suddenly feed it to another layer even much deeper in the neural network and using that, we'll build ResNet which enables you to train very, very deep networks. Sometimes even networks of over 100 layers. Let's take a look. ResNets are built out of something called a residual block, let's first describe what that is.



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

Here are two layers of a neural network where you start off with some activations in layer $a^{[l]}$, then goes $a^{[l+1]}$ and then deactivation two layers later is $a^{[l+2]}$. So let's to through the steps in this computation you have $a^{[l]}$, and then the first thing you do is you apply this linear operator to it, which is governed by this equation. So you go from $a^{[l]}$ to compute $z^{[l+1]}$ by multiplying by the weight matrix and adding that bias vector. After that, you apply the ReLU nonlinearity, to get $a^{[l+1]}$ and that's governed by this equation where $a^{[l+1]}$ is $g(z^{[l+1]})$. Then in the next layer, you apply this linear step again, so is governed by that equation. So this is quite similar to this equation we saw on the left. And then finally, you apply another ReLU operation which is now governed by that equation where G here would be the ReLU nonlinearity and this gives you $a^{[l+2]}$. So in other words, for information from $a^{[l]}$ to flow to $a^{[l+2]}$, it needs to go through all of these steps which I'm going to call the main path of this set of layers. In a residual net, we're going to make a change to this. We're going to take $a^{[l]}$, and just first forward it, copy it, match further into the neural network to here, and just at $a^{[l]}$, before applying to non-linearity, the ReLU non-linearity and I'm going to call this the shortcut. So rather than needing to follow the main path, the information from $a^{[l]}$ can now follow a shortcut to go much deeper into the neural network. and what that means is that this last equation goes away and we instead have that the output $a^{[l+2]}$ is the ReLU non-linearity g applied to $z^{[l+2]}$ as before, but now plus $a^{[l]}$. So, the addition of this $a^{[l]}$ here, it makes this a residual block. And in pictures, you can also modify this picture on top by drawing this picture shortcut to go here. And we are going to draw it as it going into this second layer here because the short cut is actually added before the ReLU non-linearity. So each of these nodes here, where there applies a linear function and a ReLU. So $a^{[l]}$ is being injected after the linear part but before the ReLU part and sometimes instead of a term short cut, you also hear the term skip connection, and that refers to $a^{[l]}$ just skipping over a layer or kind of skipping over almost two layers in order to process information deeper into the neural network. So, what the inventors of ResNet, so that'll will be Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. **What they found was that using residual blocks allows you to train much deeper neural networks. And the way you build a ResNet is by taking many of these residual blocks, blocks like these, and stacking them together to form a deep network.** So, let's look at this network. This is not the residual network, this is called as a plain network. This is the terminology of the ResNet paper.



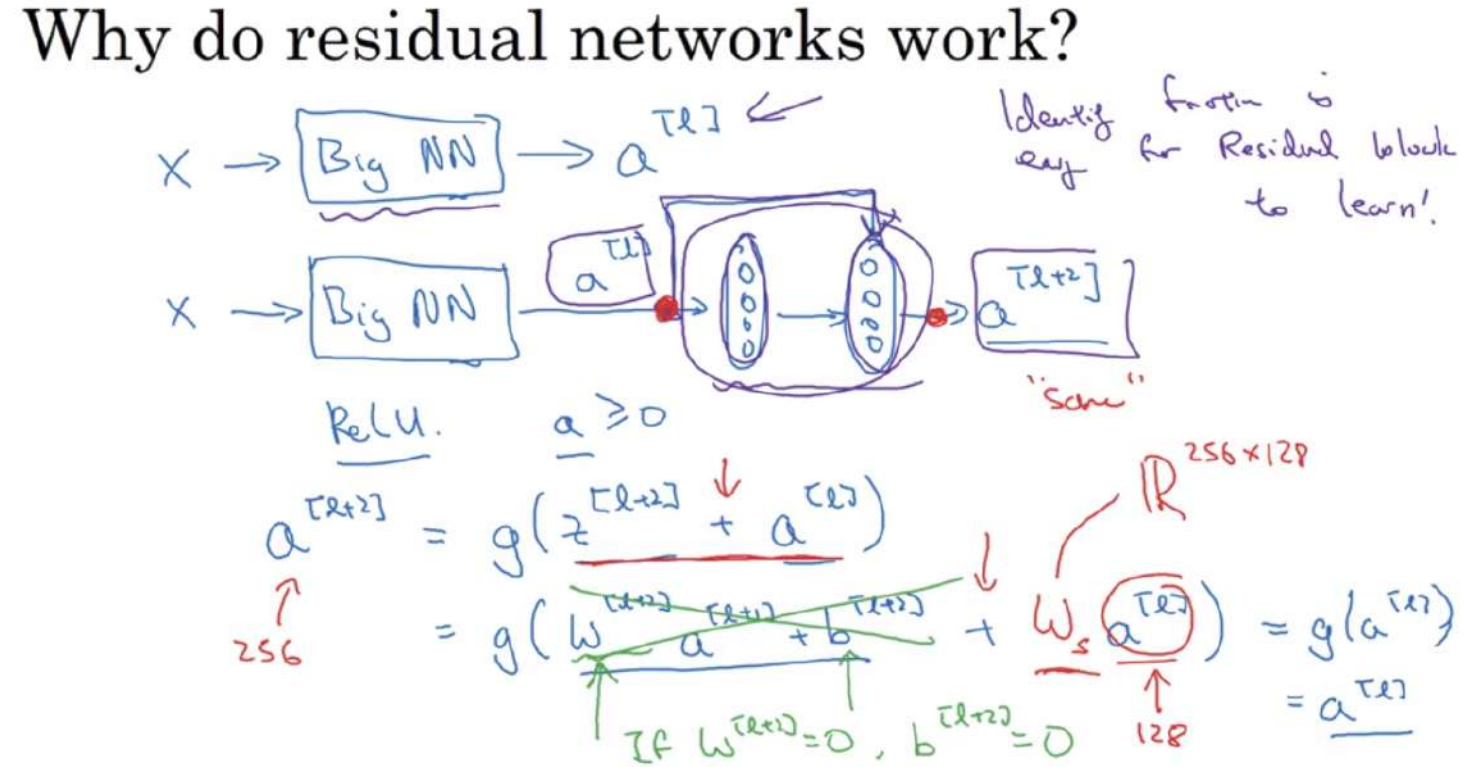
[He et al., 2015. Deep residual networks for image recognition]

To turn this into a ResNet, what you do is you add all those skip connections although those short like a connections like so. So every two layers ends up with that additional change that we saw on the previous slide to turn each of these into residual block. So this picture shows five residual blocks stacked together, and this is a residual network and it turns out that if you use your standard optimization algorithm such as a gradient descent or one of the fancier optimization algorithms to the train or plain network. So without all the extra residual, without all the extra short cuts or skip connections I just drew in. Empirically, you find that as you increase the number of layers, the training error will tend to decrease after a while but then they'll tend to go back up. And in theory as you make a neural network deeper, it should only do better and better on the training set. So, in theory, having a deeper network should only help. But in practice or in reality, having a plain network, so no ResNet, having a **plain network that is very deep means that all your optimization algorithm just has a much harder time training**. And so, in reality, your training error gets worse if you pick a network that's too deep. But what happens with ResNet is that even as the number of layers gets deeper, you can have the performance of the training error kind of keep on going down. Even if we train a network with over a hundred layers. And then now some people experimenting with networks of over a thousand layers although I don't see that it used much in practice yet. But by taking these activations be it X of these intermediate activations and allowing it to go much deeper in the neural network, this really helps with the vanishing and exploding gradient problems and allows you to train much deeper neural networks without really appreciable loss in performance, and maybe at some point, this will plateau, this will flatten out, and it doesn't help that much deeper and deeper networks. **But ResNet is not even effective at helping train very deep networks**.

Why ResNet Work

So, why do ResNets work so well? Let's go through one example that illustrates why ResNets work so well, at least in the sense of how you can make them deeper and deeper without really hurting your ability to at least get them to do well on the training set. And hopefully as you've understood from the third course in this sequence, doing well on the training set is usually a prerequisite to doing well on your hold up or on your depth or on your test sets. So, being able to

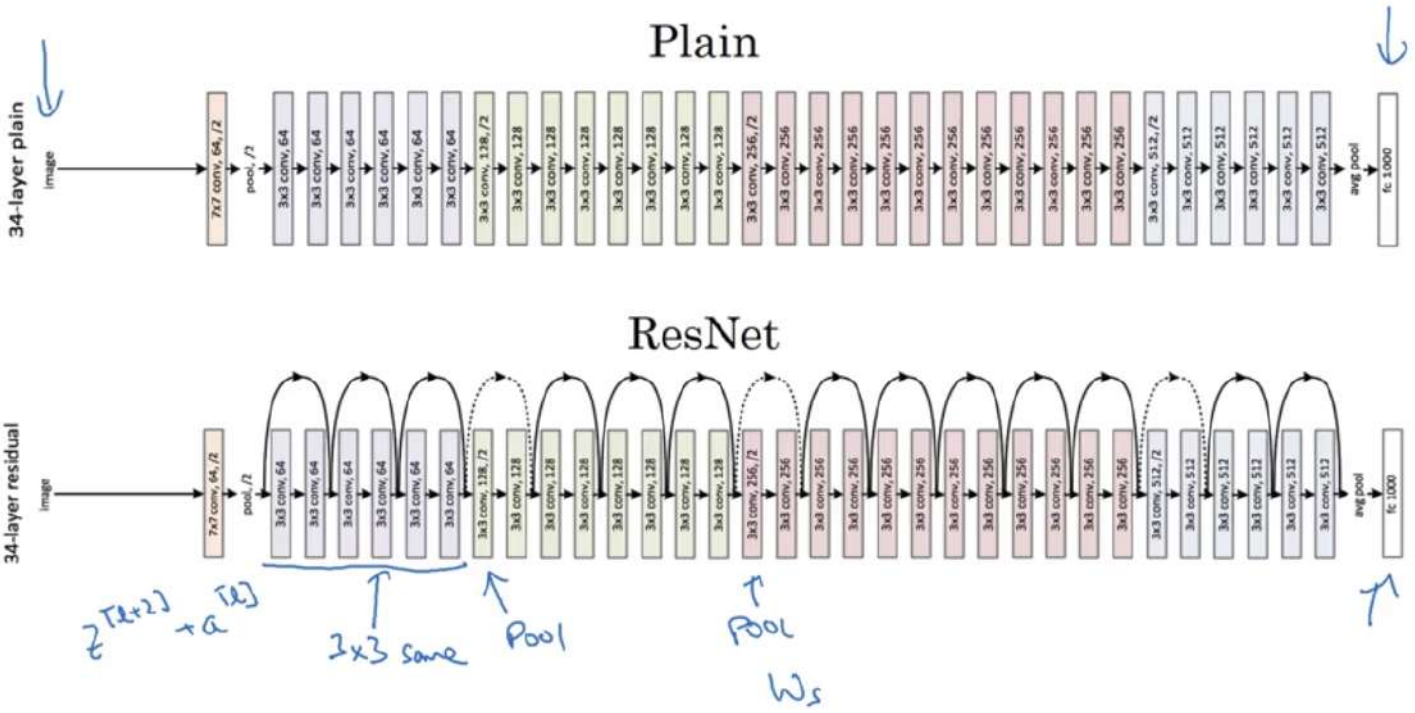
at least train ResNet to do well on the training set is a good first step toward that. Let's look at an example. What we saw on the last section was that if you make a network deeper, it can hurt your ability to train the network to do well on the training set and that's why sometimes you don't want a network that is too deep. But this is not true or at least is much less true when you training a ResNet. So let's go through an example.



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. At least is easier to go from a decent baseline of not hurting performance and then great in decent can only improve the solution from there.

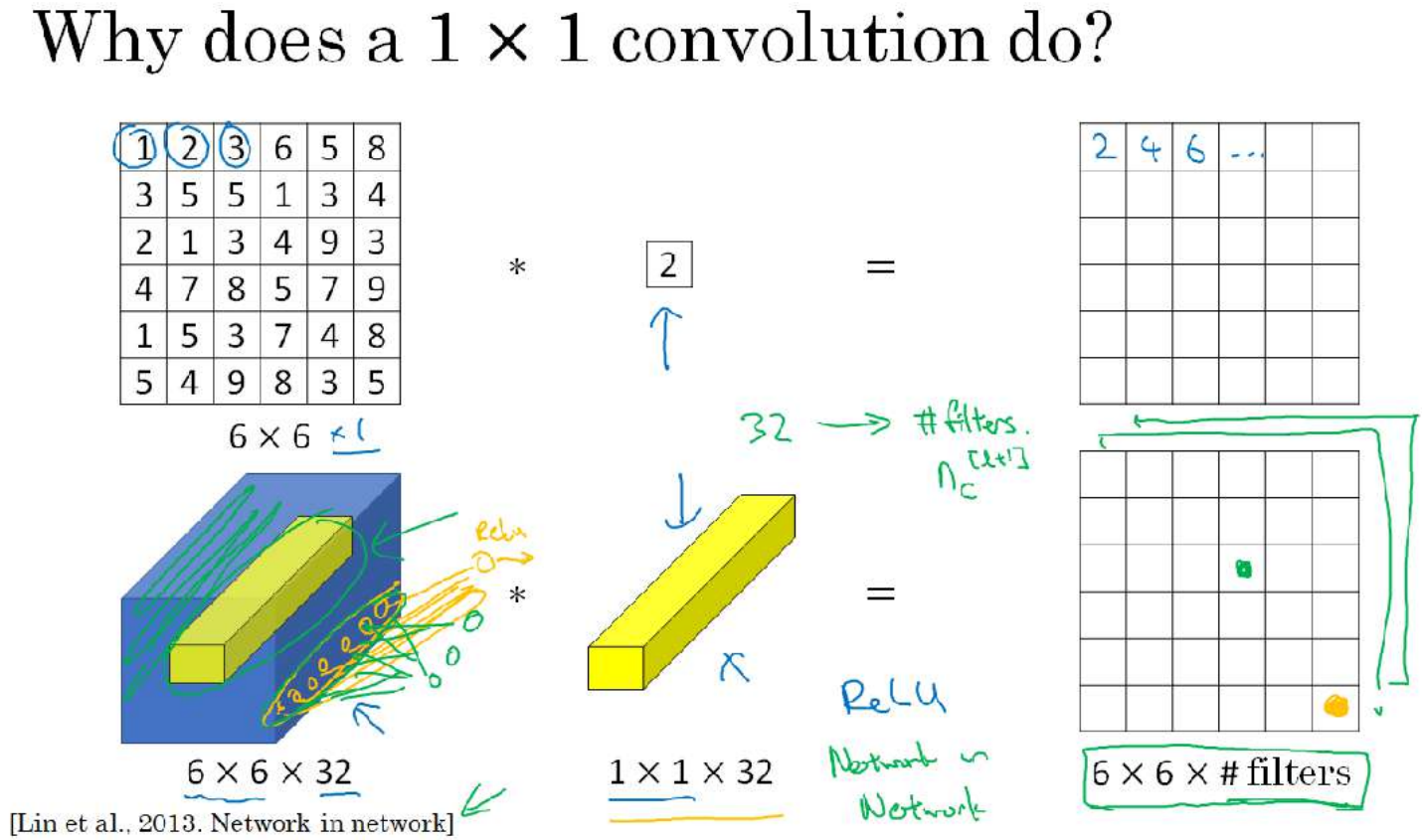
So finally, let's take a look at ResNets on images. So these are images I got from the paper by Harlow. This is an example of a plain network and in which you input an image and then have a number of conv layers until eventually you have a softmax output at the end. To turn this into a ResNet, you add those extra skip connections. And I'll just mention a few details, there are a lot of three by three convolutions here and most of these are three by three same convolutions and that's why you're adding equal dimension feature vectors.

ResNet



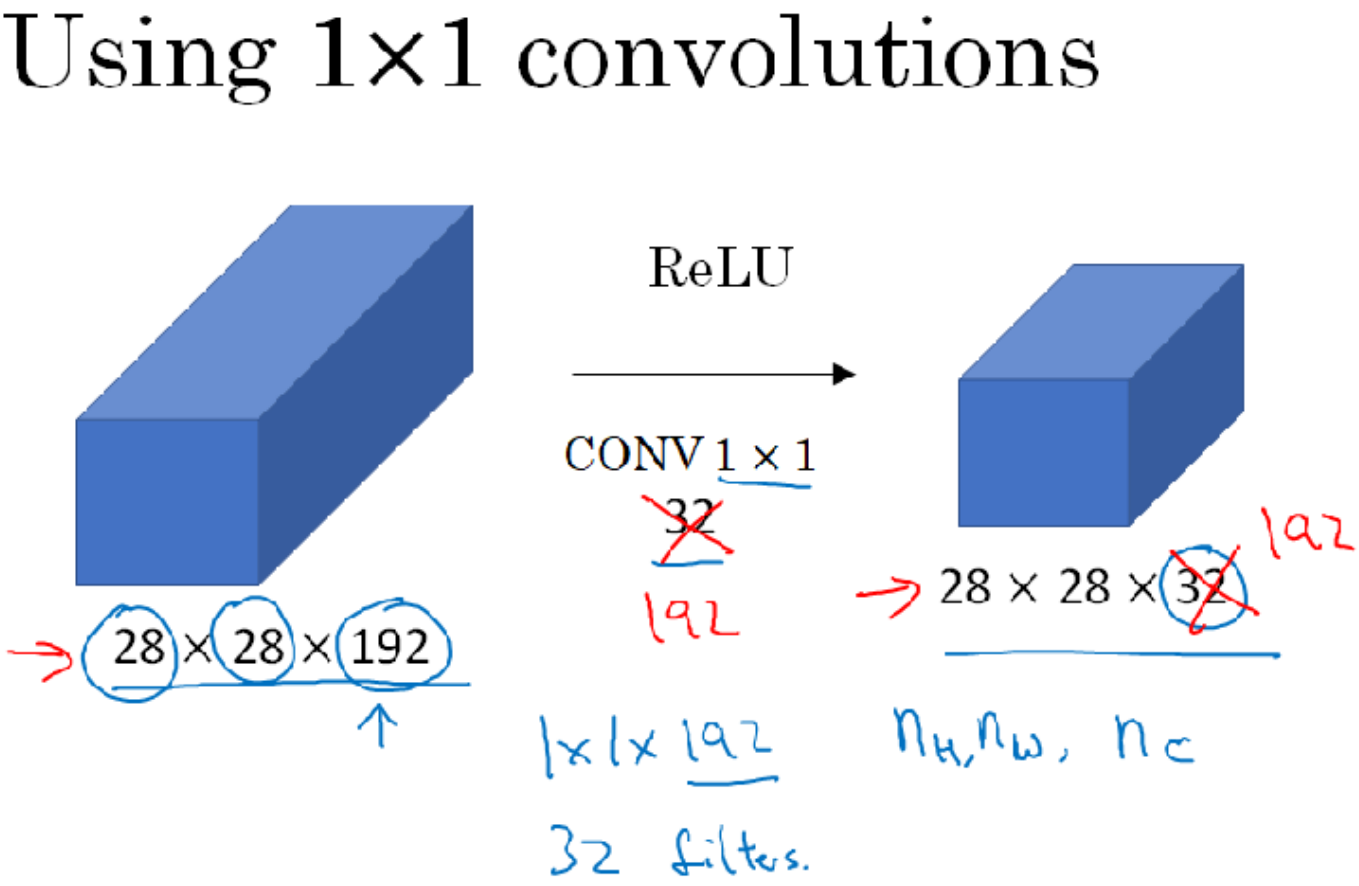
Networks in Networks and 1x1 Convolutions

In terms of designing content architectures, one of the ideas that really helps is using a 1x1 convolution. Now, you might be wondering, what does a 1x1 convolution do? Isn't that just multiplying by numbers? That seems like a funny thing to do. Turns out it's not quite like that.



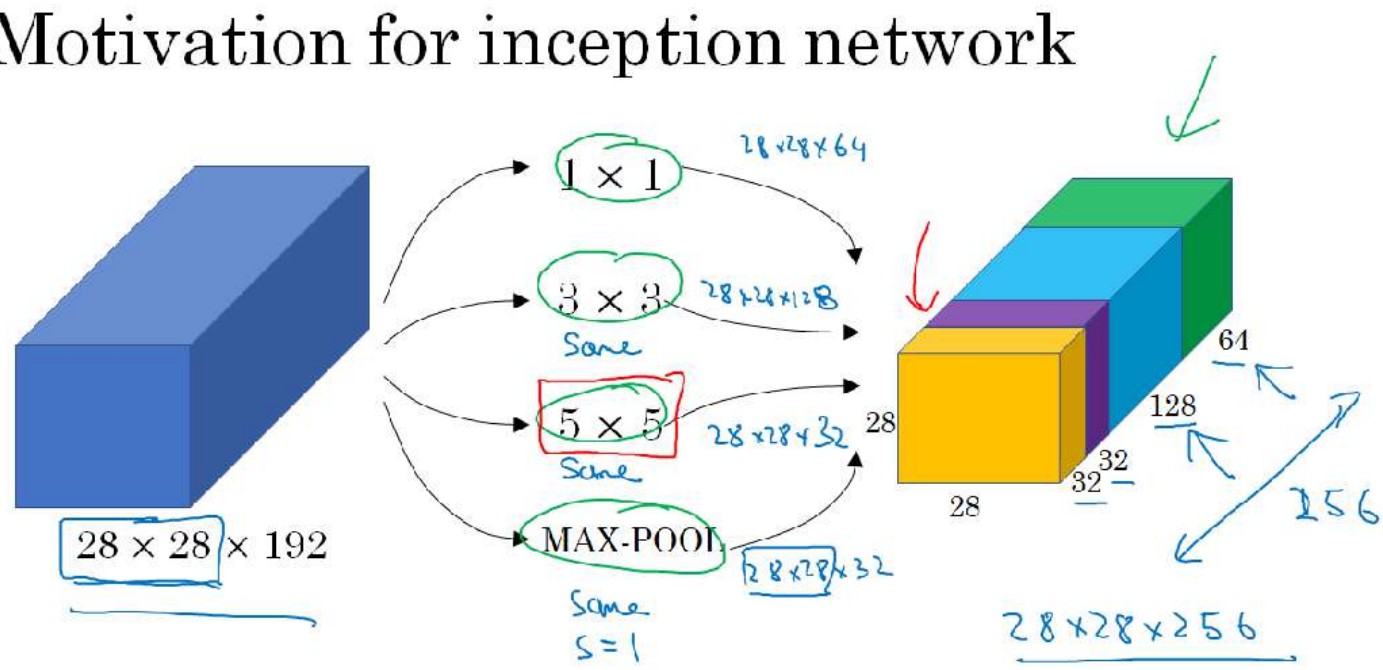
Let's take a look at the diagram. So you'll see one by one filter, we'll put in number two there, and if you take the six by six image, six by six by one and convolve it with this one by one by one

filter, you end up just taking the image and multiplying it by two. So, one, two, three ends up being two, four, six, and so on. And so, a convolution by a one by one filter, doesn't seem particularly useful. You just multiply it by some number. But that's the case of six by six by one channel images. If you have a 6 by 6 by 32 instead of by 1, then a convolution with a 1 by 1 filter can do something that makes much more sense and in particular, what a one by one convolution will do is it will look at each of the 36 different positions here, and it will take the element wise product between 32 numbers on the left and 32 numbers in the filter. And then apply a ReLU non-linearity to it after that. So, to look at one of the 36 positions, maybe one slice through this value, you take these 36 numbers multiply it by 1 slice through the volume like that, and you end up with a single real number which then gets plotted in one of the outputs like that. And in fact, one way to think about the 32 numbers you have in this 1x1x32 filters is that, it's as if you have neuron that is taking as input, 32 numbers multiplying each of these 32 numbers in one slice of the same position heightened with by these 32 different channels, multiplying them by 32 weights and then applying a ReLU non-linearity to it and then outputting the corresponding thing over there. And more generally, if you have not just one filter, but if you have multiple filters, then it's as if you have not just one unit, but multiple units, taken as input all the numbers in one slice, and then building them up into an output of six by six by number of filters. So one way to think about the **1x1 convolution is that, it is basically having a fully connected neuron network, that applies to each of the 62 different positions** and what does fully connected neural network does? Is it puts 32 numbers and outputs number of filters outputs. So I guess the point on notation, this is really a $\mathbf{n_C^{[l+1]}}$, if that's the next layer and by doing this at each of the 36 positions, each of the six by six positions, you end up with an output that is six by six by the number of filters. and this can carry out a pretty non-trivial computation on your input volume and this idea is often called a 1x1 convolution but it's sometimes also called Network in Network, and is described in this paper, by Min Lin, Qiang Chen, and Schuicheng Yan. And even though the details of the architecture in this paper aren't used widely, this idea of a one by one convolution or this sometimes called Network in Network idea has been very influential, has influenced many other neural network architectures including the inception network which we'll see in the next section. But to give you an example of where one by one convolution is useful, here's something you could do with it. Let's say you have a 28 by 28 by 192 volume. If you want to shrink the height and width, you can use a pooling layer. So we know how to do that. But one of a number of channels has gotten too big and we want to shrink that. How do you shrink it to a 28 by 28 by 32 dimensional volume? Well, what you can do is use 32 filters that are one by one. And technically, each filter would be of dimension 1 by 1 by 192, because the number of channels in your filter has to match the number of channels in your input volume, but you use 32 filters and the output of this process will be a 28 by 28 by 32 volume. So this is a way to let you shrink nc as well, whereas pooling layers, I used just to shrink nH and nW, the height and width these volumes. and we'll see later how this idea of one by one convolutions allows you to shrink the number of channels and therefore, save on computation in some networks. But of course, if you want to keep the number of channels at 192, that's fine too and the effect of the one by one convolution is it just adds non-linearity. It allows you to learn the more complex function of your network by adding another layer that inputs 28 by 28 by 192 and outputs 28 by 28 by 192. So, that's how a one by one convolutional layer is actually doing something pretty non-trivial and adds non-linearity to your neural network and allow you to decrease or keep the same or if you want, increase the number of channels in your volumes.



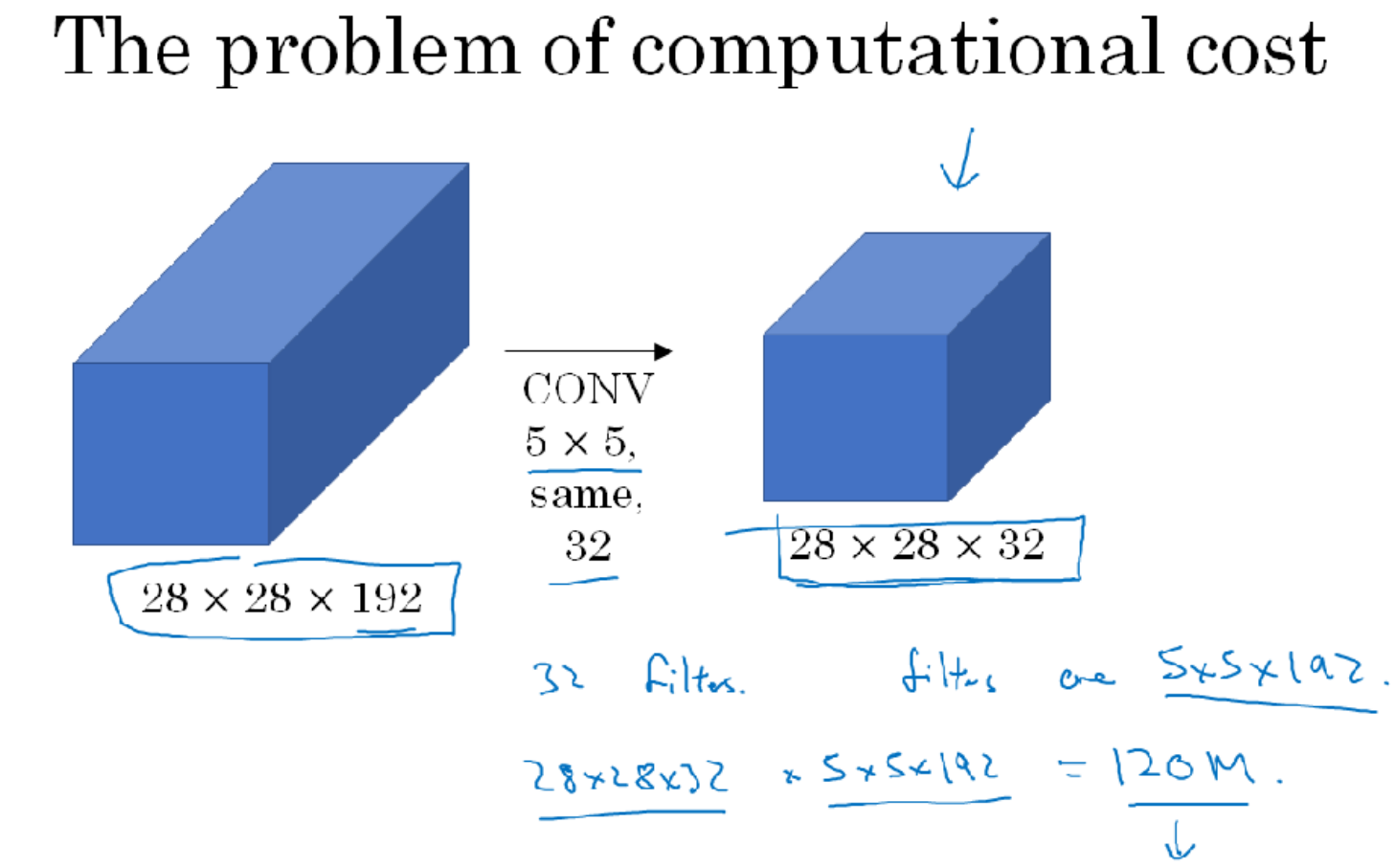
Inception Network Motivation

When designing a layer for a ConvNet, you might have to pick, do you want a 1x3 filter, or 3x3, or 5x5, or do you want a pooling layer? What the inception network does is it says, why should you do them all? and this makes the network architecture more complicated, but it also works remarkably well.



Let's see how this works. Let's say for the sake of example that you have inputted a 28 by 28 by 192 dimensional volume. So what the inception network or what an inception layer says is,

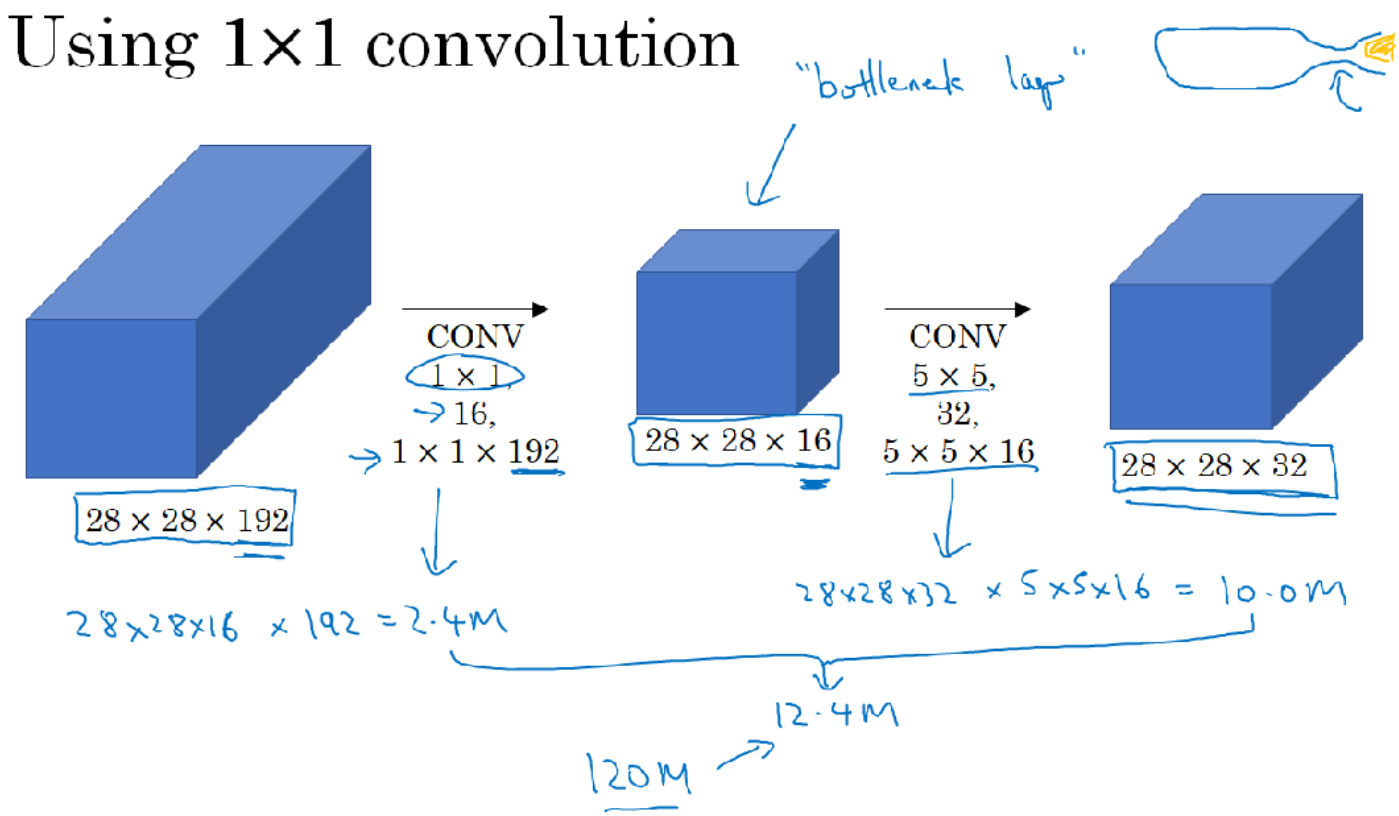
instead choosing what filter size you want in a Conv layer, or even do you want a convolutional layer or a pooling layer? Let's do them all. So what if you can use a 1 by 1 convolution, and that will output a 28 by 28 by something. Let's say 28 by 28 by 64 output, and you just have a volume there. But maybe you also want to try a 3 by 3 and that might output a 20 by 20 by 128. And then what you do is just stack up this second volume next to the first volume. And to make the dimensions match up, let's make this a same convolution. So the output dimension is still 28 by 28, same as the input dimension in terms of height and width. But 28 by 28 by in this example 128. Maybe a 5 by 5 filter works better. So let's do that too and have that output a 28 by 28 by 32. And again you use the same convolution to keep the dimensions the same. And maybe you don't want to convolutional layer. Let's apply pooling, and that has some other output and let's stack that up as well. And here pooling outputs 28 by 28 by 32. Now in order to make all the dimensions match, you actually need to use padding for max pooling. So this is an unusual formal pooling because if you want the input to have a higher than 28 by 28 and have the output, you'll match the dimension everything else also by 28 by 28, then you need to use the same padding as well as a stride of one for pooling. So this detail might seem a bit funny to you now, but let's keep going. And we'll make this all work later. But with a inception module like this, you can input some volume and output. In this case I guess if you add up all these numbers, 32 plus 32 plus 128 plus 64, that's equal to 256. So you will have one inception module input 28 by 28 by 129, and output 28 by 28 by 256. And this is the heart of the inception network which is due to Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. And the basic idea is that instead of you needing to pick one of these filter sizes or pooling you want and committing to that, you can do them all and just concatenate all the outputs, and let the network learn whatever parameters it wants to use, whatever the combinations of these filter sizes it wants. Now it turns out that there is a problem with the inception layer as we've described it here, which is computational cost. Let's figure out what's the computational cost of this 5 by 5 filter resulting in this block over here.



So just focusing on the 5 by 5 pot on the previous diagram, we had as input a 28 by 28 by 192 block, and you implement a 5 by 5 same convolution of 32 filters to output 28 by 28 by 32. On the

previous diagram we had drawn this as a thin purple slide. So I'm just going draw this as a more normal looking blue block here. So let's look at the computational costs of outputting this 20 by 20 by 32. So you have 32 filters because the outputs has 32 channels, and each filter is going to be 5 by 5 by 192. And so the output size is 20 by 20 by 32, and so you need to compute 28 by 28 by 32 numbers. And for each of them you need to do these many multiplications, right? 5 by 5 by 192. So the total number of multiplies you need is the number of multiplies you need to compute each of the output values times the number of output values you need to compute. And if you multiply all of these numbers, this is equal to 120 million. And so, while you can do 120 million multiplies on the modern computer, this is still a pretty expensive operation. On the next diagram you see how using the idea of 1 by 1 convolutions, which you learnt about in the previous section, you'll be able to reduce the computational costs by about a factor of 10. To go from about 120 million multiplies to about one tenth of that. So please remember the number 120 so you can compare it with what you see on the next diagram, 120 million. Here is an alternative architecture for inputting 28 by 28 by 192, and outputting 28 by 28 by 32, which is falling. You are going to input the volume, use a 1 by 1 convolution to reduce the volume to 16 channels instead of 192 channels, and then on this much smaller volume, run your 5 by 5 convolution to give you your final output. So notice the input and output dimensions are still the same. You input 28 by 28 by 192 and output 28 by 28 by 32, same as the previous diagram. But what we've done is we're taking this huge volume we had on the left, and we shrunk it to this much smaller intermediate volume, which only has 16 instead of 192 channels. Sometimes this is called a **bottleneck layer**. I guess because a bottleneck is usually the smallest part of something. So I guess if you have a glass bottle that looks like this, then you know this is I guess where the cork goes. And then the bottleneck is the smallest part of this bottle. So in the same way, the bottleneck layer is the smallest part of this network. We shrink the representation before increasing the size again. Now let's look at the computational costs involved. To apply this 1 by 1 convolution, we have 16 filters. Each of the filters is going to be of dimension 1 by 1 by 192, this 192 matches that 192. And so the cost of computing this 28 by 28 by 16 volumes is going to be well, you need these many outputs, and for each of them you need to do 192 multiplications. I could have written 1 times 1 times 192, right? Which is this. And if you multiply this out, this is 2.4 million, it's about 2.4 million. How about the second? So that's the cost of this first convolutional layer. The cost of this second convolutional layer would be that well, you have these many outputs. So 28 by 28 by 32. And then for each of the outputs you have to apply a 5 by 5 by 16 dimensional filter. And so by 5 by 5 by 16. And you multiply that out is equals to 10.0. And so the total number of multiplications you need to do is the sum of those which is 12.4 million multiplications and you compare this with what we had on the previous slide, you reduce the computational cost from about 120 million multiplies, down to about one tenth of that, to 12.4 million multiplications and the number of additions you need to do is about very similar to the number of multiplications you

need to do. So that's why I'm just counting the number of multiplications.

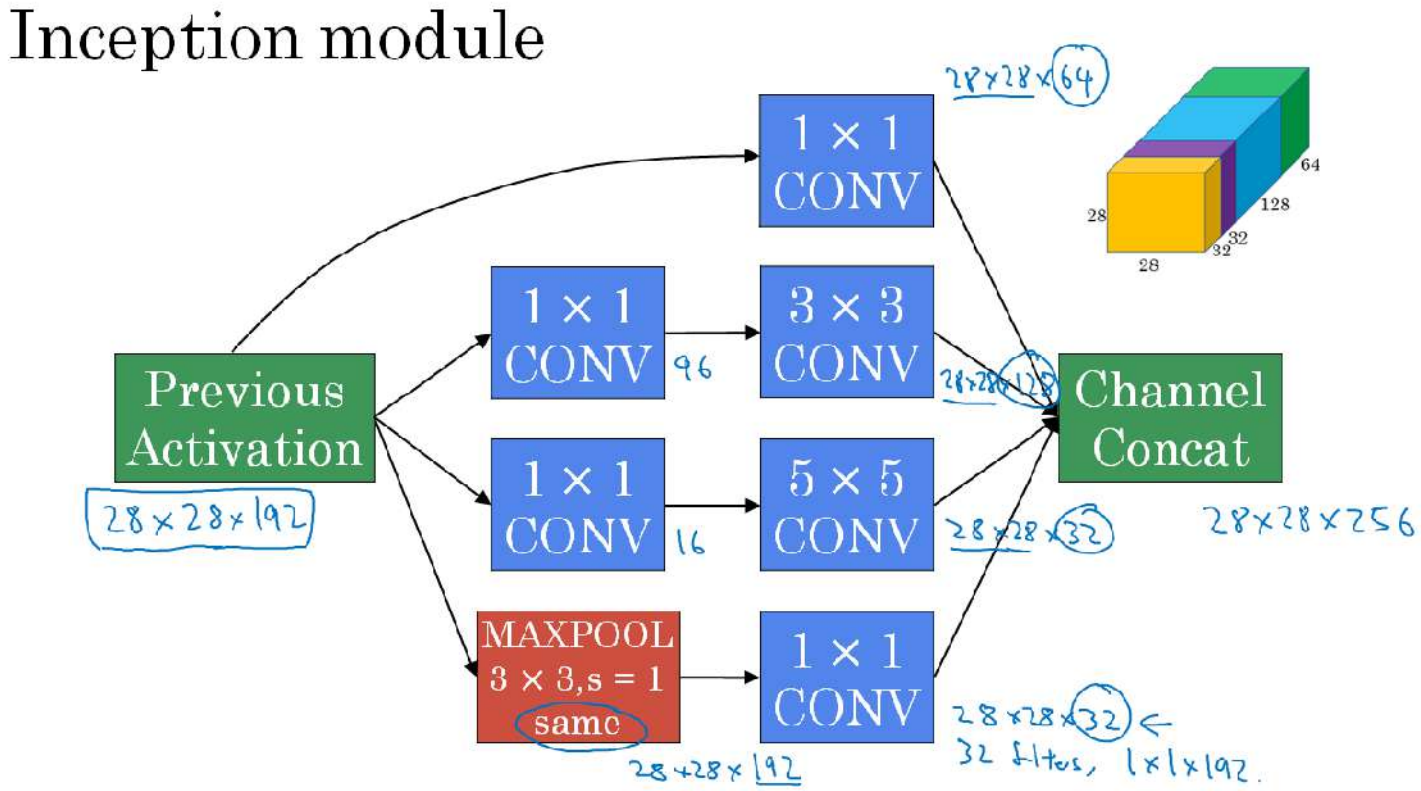


So to summarize, if you are building a layer of a neural network and you don't want to have to decide, do you want a 1 by 1, or 3 by 3, or 5 by 5, or pooling layer, the inception module let's you say let's do them all, and let's concatenate the results. And then we run to the problem of computational cost and what you saw here was how using a 1 by 1 convolution, you can create this bottleneck layer thereby reducing the computational cost significantly. Now you might be wondering, does shrinking down the representation size so dramatically, does it hurt the performance of your neural network? It turns out that so long as you implement this bottleneck layer so that within reason, you can shrink down the representation size significantly, and it doesn't seem to hurt the performance, but saves you a lot of computation. So these are the key ideas of the inception module. Let's put them together and in the next section will show you what the full inception network looks like.

Inception Network

In the previous section, we've already seen all the basic building blocks of the Inception network. In this section, let's see how we can put these building blocks together to build your own Inception network. So the Inception module takes as input the activation or the output from some previous layer. So let's say for the sake of argument, this is 28 by 28 by 192, same as our previous section. The example we worked through in depth was the one by one followed by five by five layer. So maybe the one by one has 16 channels, and then the five by five will output a 28 by 28 by let say, 32 channels and this was the example we worked through on the last diagram of the previous section. Then, to save computation on your three by three convolution, you can also do the same here and then the three by three outputs 28 by 28 by 128 and then, maybe you want to consider a one by one convolution as well. There's no need to do a one by one conv followed by another one by one conv. So there's just one step here. And let's say this outputs 28 by 28 by 64 and then finally is the pooling layer. And here we're going to do something funny. So we are going to use- we want to use a same padding for max pooling. So the output is 28 by 28 and by- so here we're going to do something funny. In order to deliver concatenate all of these outputs at the end, we are going to use same type of padding for pooling so that the output item with is still 28 by 28. So we can concatenate it with these other outputs. But notice that if you do max pooling, even with same padding three by three filter is right at one. The output here will be twenty 28 by 28 by 192.

You will have the same number of channels and the same depth as the input that we had here. So this seems like it has a lot of channels. So what we're going to do is actually add one more one by one conv layer to then do what we saw in the one by one convolutional layer to shrink the number of channels so as to get this down to 28 by 28 by, let's say, 32 and the way you do that is to use 32 filters of dimension one by one by 192. So that's why the output dimension has the number of channels shrunk down to 32, so then you don't end up with the pooling layer taking up all the channels in the final output and then finally, you take these all of these blocks and you do channel concatenation, just concatenate across this 64 plus 128 plus 32 plus 32, and this if you add it up, this gives you a 28 by 28 by 256 dimensional output. Channel concat is just concatenating the blocks that we saw in the previous section. So this is one Inception module and **what the Inception network does is more or less put a lot of these modules together**. Below id the picture of the Inception that were taken from the paper by Szegety et al. and you notice a lot of repeated blocks in this.



Maybe this picture looks really complicated but you look at one of the blocks there, that block is basically the Inception module that you saw on the previous section. There's some extra Max pooling layers here to change the dimension of the height and width. But there's another Inception block, and then there's another max pool here to change the height and width but basically there's another Inception block. But the Inception network is just a lot of these blocks that you've learned about repeated to different positions of the network and so if you understand the Inception block from the previous section, then you understand the Inception network. It turns out that this one last detail to the Inception network if you read the original research paper, which is that there are these additional side branches that I just added. So what do they do? Well, the last few layers of the network is a fully connected layer followed by a softmax layer to try to make a prediction. What these side branches do is it take some hidden layer, and it tries to use that to make a prediction. So this is actually a softmax output, and so is that. And this other side branch, again takes a hidden layer passes through a few layers, a few fully connected layers, and then as a softmax tried to predict what's the output label. You should think of this as maybe just another detail of the Inception network, but what it does is it helps ensure that the features computes even in the hidden units, even at that intermediate layers that they're not too bad for predicting the output cause of a image and this appears to have a regularizing effect on the Inception network and helps prevent this network from overfitting.

So to summarize, **if you understand the Inception module, then you understand the Inception network, which is largely the Inception module repeated a bunch of times throughout the network**. Since the development of the original Inception module the authors and others have built on it and come up with other versions as well. So there are research papers on newer versions of the Inception algorithm and you sometimes see people use some of these later versions as well in their work, like Inception V2, Inception V3, Inception V4. There's also an Inception version that's combined with the resident idea of having skip connections and that sometimes works even better. But all of these variations are built on the basic idea that you learned about in this and the previous video of coming up with the Inception module and then stacking up a bunch of them together.

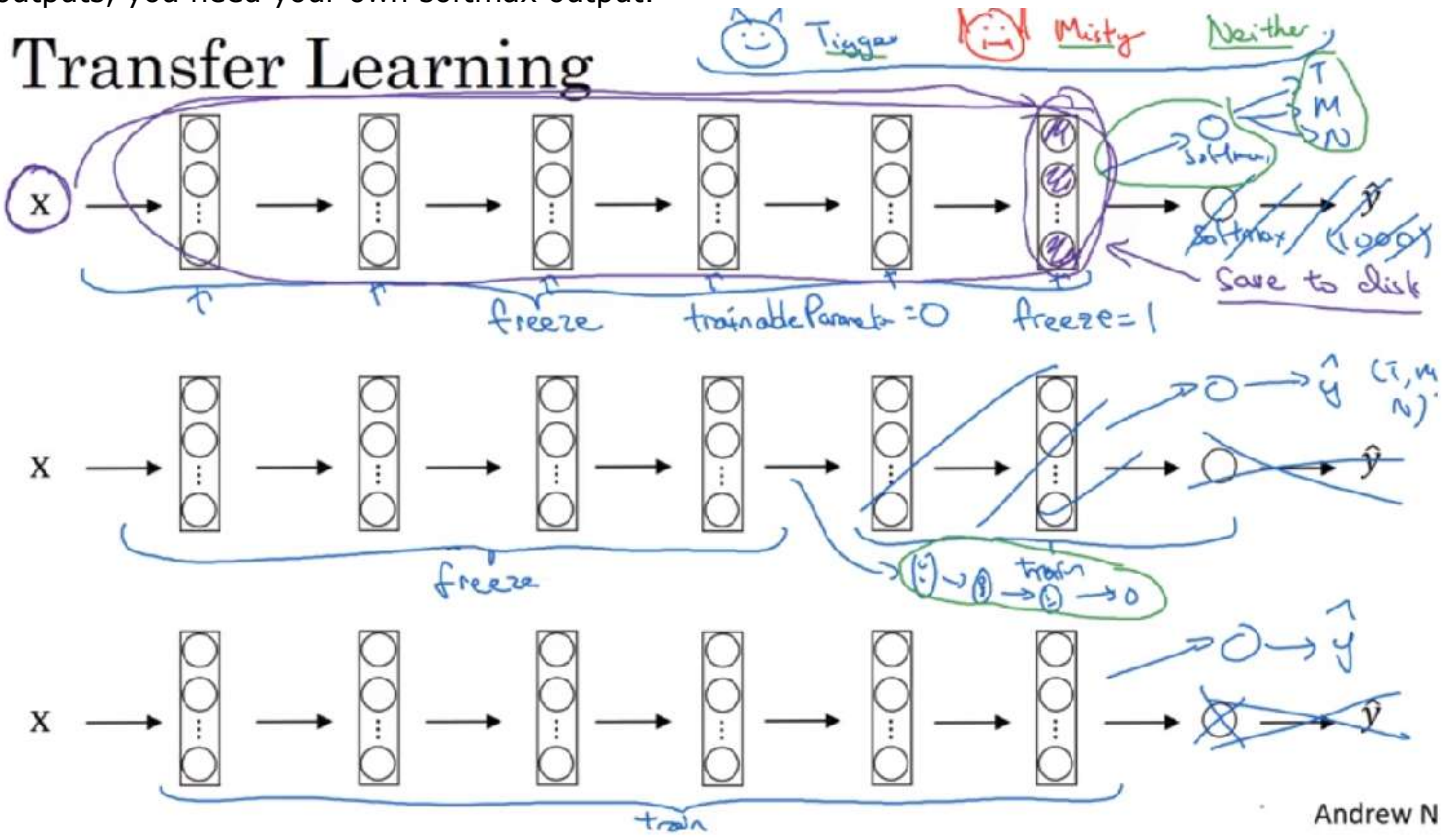
Practical advice for using ConvNets

Using Open-Source Implementation

Transfer Learning

If you're building a computer vision application rather than training the ways from scratch, from random initialization, you often make much faster progress if you download ways that someone else has already trained on the network architecture and use that as pre-training and transfer that to a new task that you might be interested in. The computer vision research community has been pretty good at posting lots of data sets on the Internet so if you hear of things like Image Net, or MS COCO, or Pascal types of data sets, these are the names of different data sets that people have post online and a lot of computer researchers have trained their algorithms on. Sometimes these training takes several weeks and might take many GP use and the fact that someone else has done this and gone through the painful high-performance search process, means that you can often download open source ways that took someone else many weeks or months to figure out and use that as a very good initialization for your own neural network. And use transfer learning to sort of transfer knowledge from some of these very large public data sets to your own problem. Let's take a deeper look at how to do this. Let's start with the example, let's say you're building a cat detector to recognize your own pet cat. According to the internet, Tigger is a common cat name and Misty is another common cat name. Let's say your cats are called Tiger and Misty and there's also neither. You have a classification problem with three clauses. Is this picture Tigger, or is it Misty, or is it neither. And in all the case of both of you cats appearing in the picture. Now, you probably don't have a lot of pictures of Tigger or Misty so your training set will be small. What can you do? I recommend you go online and download some open source implementation of a neural network and download not just the code but also the weights. There are a lot of networks you can download that have been trained on for example, the Init Net data sets which has a thousand different clauses so the network might have a softmax unit that outputs one of a thousand possible clauses. What you can do is then get rid of the softmax layer and create your own softmax unit that outputs Tigger or Misty or neither. In terms of the network, I'd encourage you to think of all of these layers as frozen so you freeze the parameters in all of these layers of the network and you would then just train the parameters associated with your softmax layer. Which is the softmax layer with three possible outputs, Tigger, Misty or neither. By using someone else's free trade ways, you might probably get pretty good performance on this even with a small data set. Fortunately, a lot of people learning frameworks support this mode of operation and in fact, depending on the framework it might have things like trainable parameter equals zero, you might set that for some of these early layers. In others they just say, don't train those ways or sometimes you have a parameter like freeze equals one and these are different ways and different deep learning program frameworks that let you specify whether or not to train the ways associated with particular layer. In this case, you will train only the softmax layers ways but freeze all of the earlier layers ways. One other metric that may help for some implementations is that because all of these early leads are frozen, there are some fixed function that doesn't change because you're not changing it, you not training it that takes this input image acts and maps it to some set of activations in that layer. One of the trick that could speed up training is we just pre-compute that layer, the features of re-activations from that layer and just save them to disk. What you're doing is using this fixed function, in this first part of the neural network, to take this input any image X

and compute some feature vector for it and then you're training a shallow softmax model from this feature vector to make a prediction. One step that could help your computation as you just pre-compute that layers activation, for all the examples in training sets and save them to disk and then just train the softmax clause right on top of that. The advantage of the save to disk or the pre-compute method or the save to disk is that you don't need to recompute those activations everytime you take a epoch or take a post through a training set. This is what you do if you have a pretty small training set for your task. Whether you have a larger training set. One rule of thumb is if you have a larger label data set so maybe you just have a ton of pictures of Tigger, Misty as well as I guess pictures neither of them, one thing you could do is then freeze fewer layers. Maybe you freeze just these layers and then train these later layers. Although if the output layer has different clauses then you need to have your own output unit any way Tigger, Misty or neither. There are a couple of ways to do this. You could take the last few layers ways and just use that as initialization and do gradient descent from there or you can also lower weight of these last few layers and just use your own new hidden units and in your own final softmax outputs. Either of these matters could be worth trying. But maybe one pattern is if you have more data, the number of layers you've freeze could be smaller and then the number of layers you train on top could be greater and the idea is that if you pick a data set and maybe have enough data not just to train a single softmax unit but to train some other size neural network that comprises the last few layers of this final network that you end up using. Finally, if you have a lot of data, one thing you might do is take this open source network and ways and use the whole thing just as initialization and train the whole network. Although again if this was a thousand of softmax and you have just three outputs, you need your own softmax output.

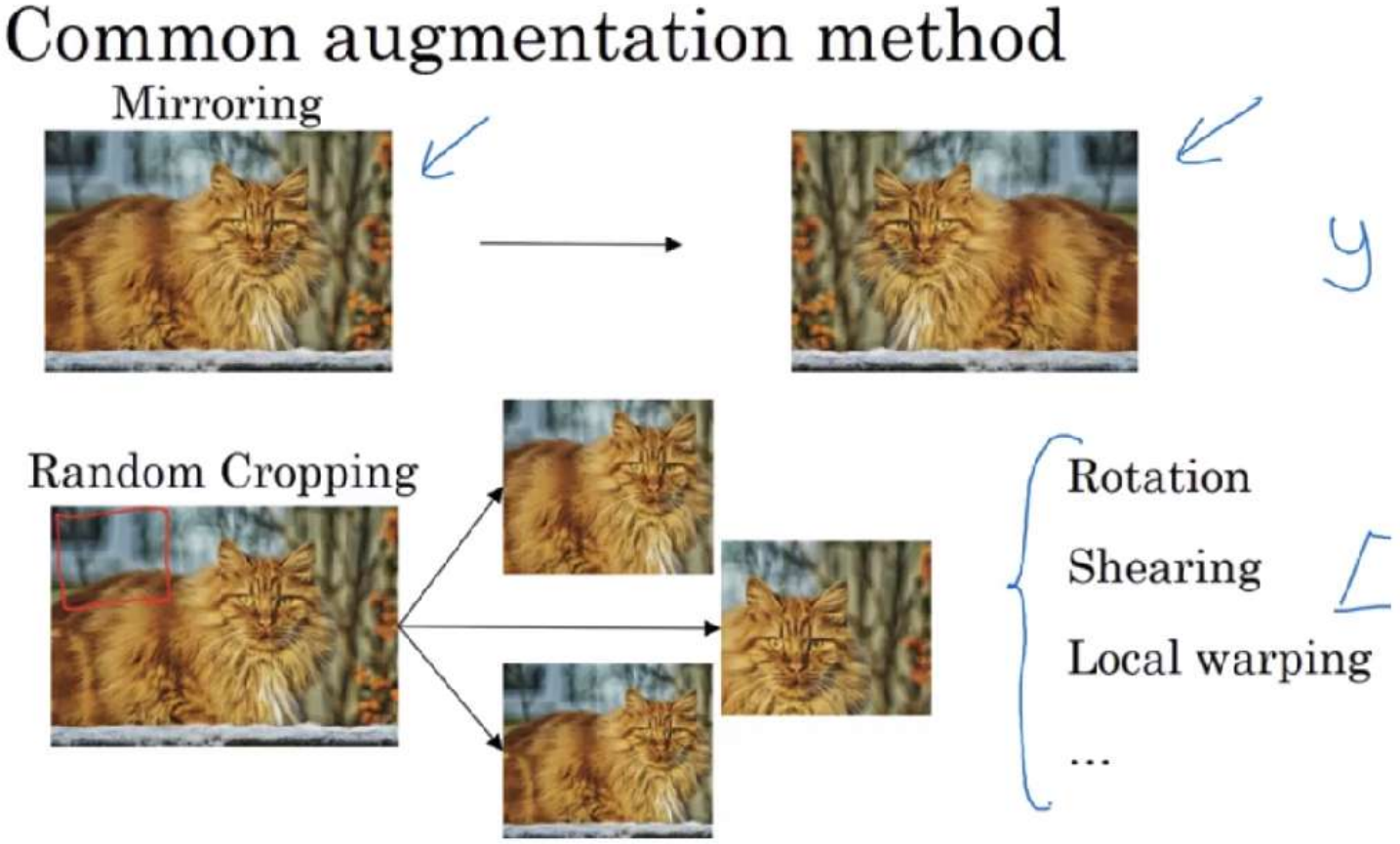


The output of labels you care about but the more label data you have for your task or the more pictures you have of Tigger, Misty and neither, the more layers you could train and in the extreme case, you could use the ways you download just as initialization so they would replace random initialization and then could do gradient descent, training updating all the ways and all the layers of the network. That's transfer learning for the training of ConvNets. In practice, because the open data sets on the internet are so big and the ways you can download that someone else has spent weeks training has learned from so much data, you find that for a lot of computer vision applications, you just do much better if you download someone else's open source ways and use that as initialization for your problem. In all the different disciplines, in all the different

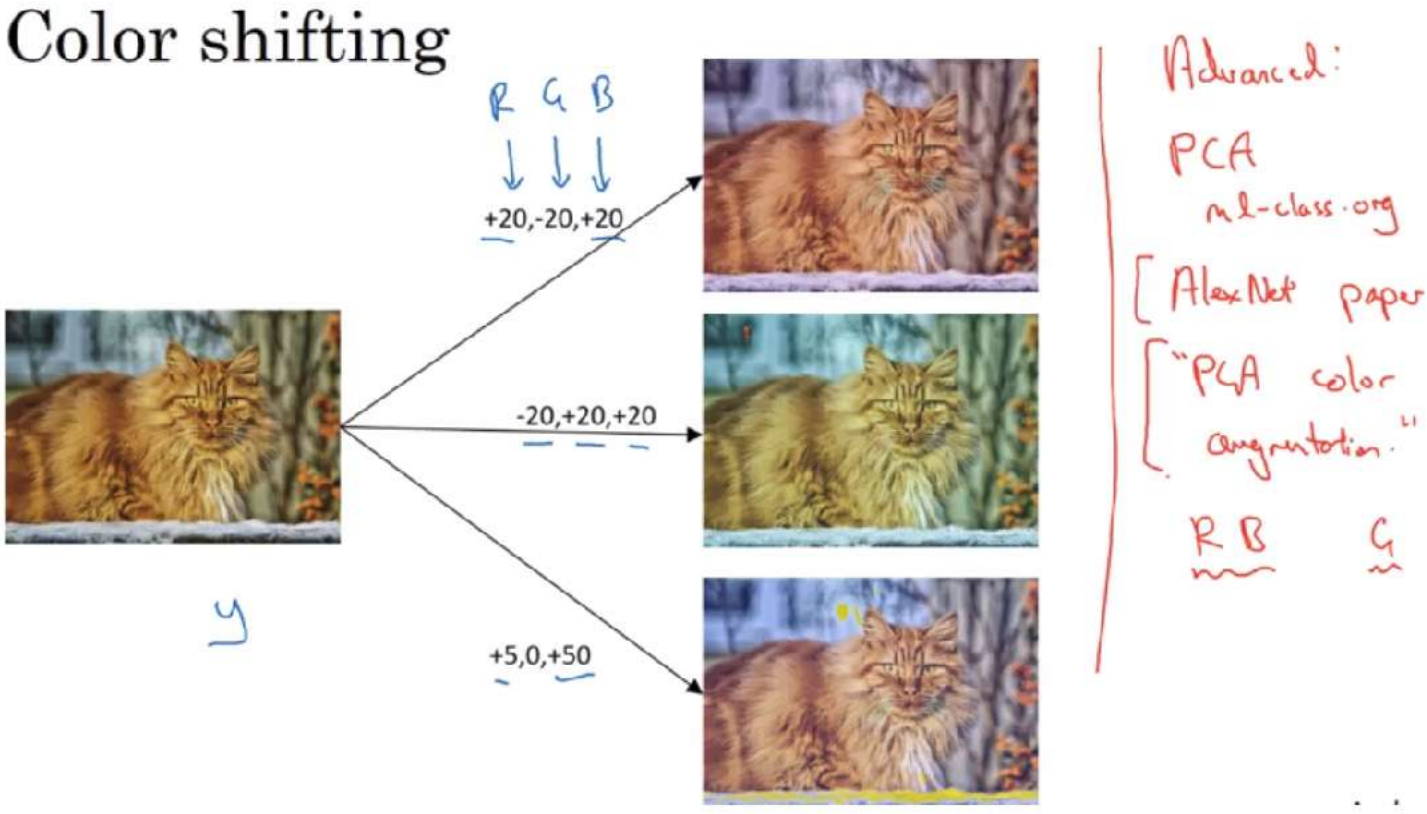
applications of deep learning, I think that computer vision is one where transfer learning is something that you should almost always do unless, you have an exceptionally large data set to train everything else from scratch yourself. But transfer learning is just very worth seriously considering unless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

Data Augmentation

Most computer vision task could use more data. And so data augmentation is one of the techniques that is often used to improve the performance of computer vision systems. I think that computer vision is a pretty complicated task. You have to input this image, all these pixels and then figure out what is in this picture and it seems like you need to learn the decently complicated function to do that and in practice, there almost all competing visions task having more data will help. This is unlike some other domains where sometimes you can get enough data, they don't feel as much pressure to get even more data. But I think today, this data computer vision is that, for the majority of computer vision problems, we feel like we just can't get enough data and this is not true for all applications of machine learning, but it does feel like it's true for computer vision. So, what that means is that when you're training in computer vision model, often **data augmentation** will help and this is true whether you're using transfer learning or using someone else's pre-trained ways to start, or whether you're trying to train something yourself from scratch. Let's take a look at the common data augmentation that is in computer vision. Perhaps the simplest data augmentation method is mirroring on the vertical axis, where if you have this example in your training set, you flip it horizontally to get that image on the right and for most computer vision task, if the left picture is a cat then mirroring it is though a cat and if the mirroring operation preserves whatever you're trying to recognize in the picture, this would be a good data augmentation technique to use. Another commonly used technique is **random cropping**. So given this dataset, let's pick a few random crops. So you might pick that, and take that crop or you might take that, to that crop, take this, take that crop and so this gives you different examples to feed in your training sample, sort of different random crops of your datasets. So random cropping isn't a perfect data augmentation. What if you randomly end up taking that crop which will look much like a cat but in practice and worthwhile so long as your random crops are reasonably large subsets of the actual image. So, **mirroring and random cropping are frequently used and in theory, you could also use things like rotation, shearing of the image**, so that's if you do this to the image, distort it that way, introduce various forms of local warping and so on. And there's really no harm with trying all of these things as well, although in practice they seem to be used a bit less, or perhaps because of their complexity.



The second type of data augmentation that is commonly used is **color shifting**. So, given a picture like this,



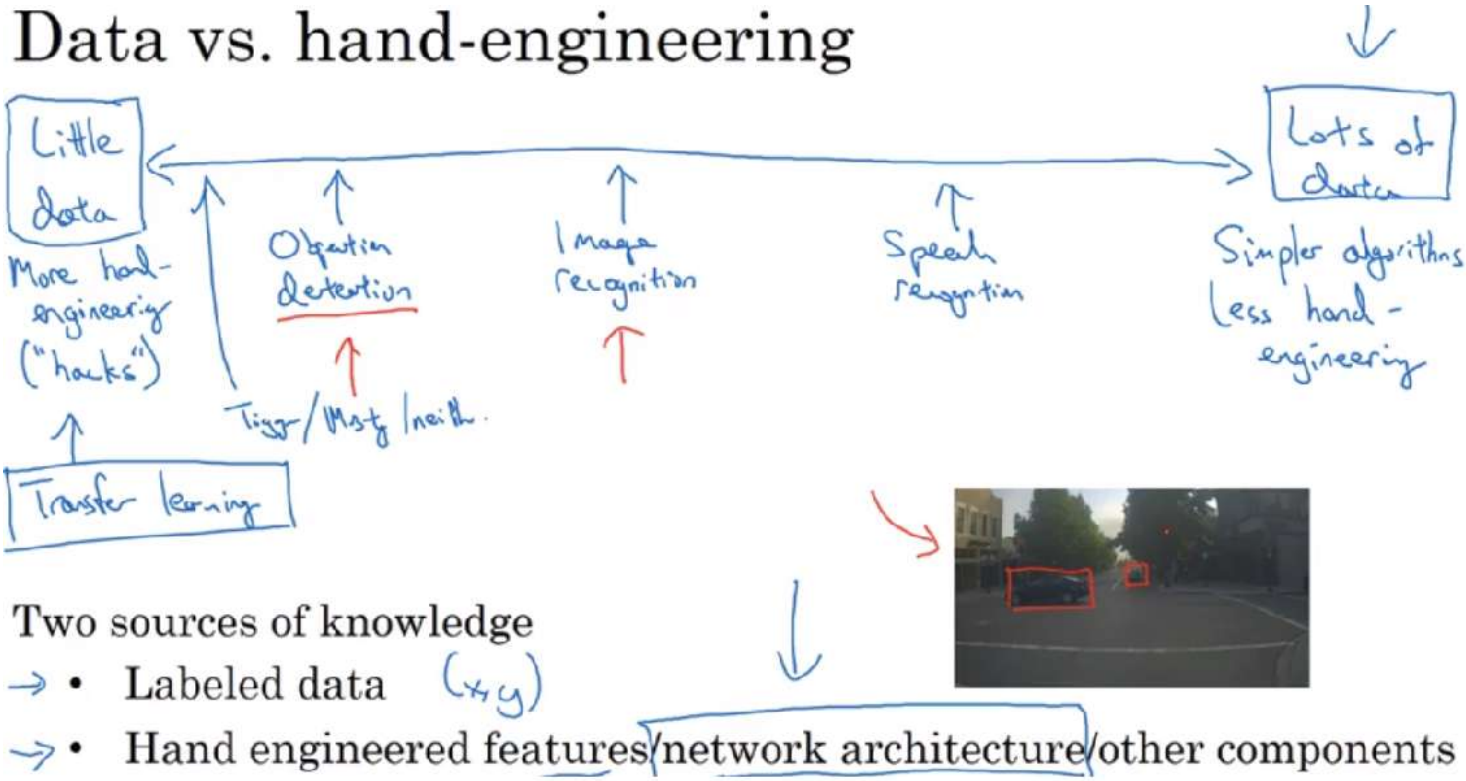
let's say you add to the R, G and B channels different distortions. In this example, we are adding to the red and blue channels and subtracting from the green channel. So, red and blue make purple. So, this makes the whole image a bit more purpley and that creates a distorted image for

training set. For illustration purposes, I'm making somewhat dramatic changes to the colors and practice, you draw R, G and B from some distribution that could be quite small as well. But what you do is take different values of R, G, and B and use them to distort the color channels. So, in the second example, we are making a less red, and more green and more blue, so that turns our image a bit more yellowish. And here, we are making it much more blue, just a tiny little bit longer. But in practice, the values R, G and B, are drawn from some probability distribution. And the motivation for this is that if maybe the sunlight was a bit yellow or maybe the in-goal illumination was a bit more yellow, that could easily change the color of an image, but the identity of the cat or the identity of the content, the label y, just still stay the same. And so introducing these color distortions or by doing color shifting, this makes your learning algorithm more robust to changes in the colors of your images. Just a comment for the advanced learners in this course, that is okay if you don't understand what I'm about to say when using red. There are different ways to sample R, G, and B. One of the ways to implement color distortion uses an algorithm called PCA. This is called **Principles Component Analysis**, which I talked about in the ml-class.org Machine Learning Course on Coursera. But the details of this are actually given in the AlexNet paper, and sometimes called **PCA Color Augmentation**. But the rough idea at the time PCA Color Augmentation is for example, if your image is mainly purple, if it mainly has red and blue tints, and very little green, then PCA Color Augmentation, will add and subtract a lot to red and blue, where it balance all the greens, so kind of keeps the overall color of the tint the same. If you didn't understand any of this, don't worry about it. But if you can search online for that, you can and if you want to read about the details of it in the AlexNet paper, and you can also find some open source implementations of the PCA Color Augmentation, and just use that. So, you might have your training data stored in a hard disk and uses symbol, this round bucket symbol to represent your hard disk. And if you have a small training set, you can do almost anything and you'll be okay. But the very last training set and this is how people will often implement it, which is you might have a CPU thread that is constantly loading images of your hard disk. So, you have this stream of images coming in from your hard disk. And what you can do is use maybe a CPU thread to implement the distortions, yet the random cropping, or the color shifting, or the mirroring, but for each image, you might then end up with some distorted version of it. So, let's see this image, I'm going to mirror it and if you also implement colors distortion and so on. And if this image ends up being color shifted, so you end up with some different colored cat. And so your CPU thread is constantly loading data as well as implementing whether the distortions are needed to form a batch or really many batches of data. And this data is then constantly passed to some other thread or some other process for implementing training and this could be done on the CPU or really increasingly on the GPU if you have a large neural network to train. And so, a pretty common way of implementing data augmentation is to really have one thread, almost four threads, that is responsible for loading the data and implementing distortions, and then passing that to some other thread or some other process that then does the training. And often, this and this, can run in parallel. So, that's it for data augmentation. And similar to other parts of training a deep neural network, the data augmentation process also has a few hyperparameters such as how much color shifting do you implement and exactly what parameters you use for random cropping? So, similar to elsewhere in computer vision, a good place to get started might be to use someone else's open source implementation for how they use data augmentation. But of course, if you want to capture more in variances, then you think someone else's open source implementation isn't, it might be reasonable also to use hyperparameters yourself. So with that, I hope that you're going to use data augmentation, to get your computer vision applications to work better.

State of Computer Vision

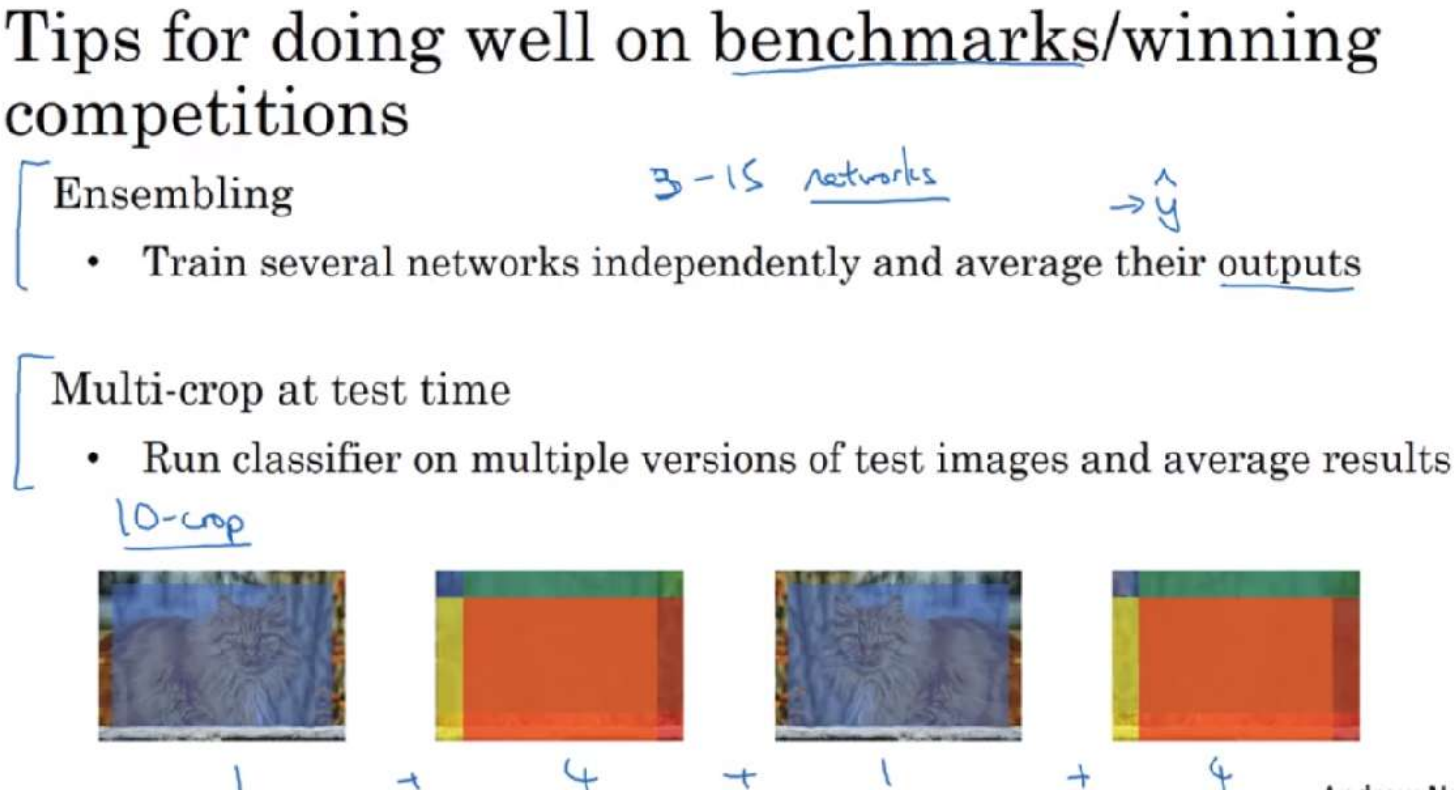
Deep learning has been successfully applied to computer vision, natural language processing, speech recognition, online advertising, logistics, many, many, many problems. There are a few things that are unique about the application of deep learning to computer vision, about the status of computer vision. In this video, I will share with you some of my observations about deep learning for computer vision and I hope that that will help you better navigate the literature, and the set of ideas out there, and how you build these systems yourself for computer vision. So, you

can think of most machine learning problems as falling somewhere on the spectrum between where you have relatively little data to where you have lots of data. So, for example I think that today we have a decent amount of data for speech recognition and it's relative to the complexity of the problem. And even though there are reasonably large data sets today for image recognition or image classification, because image recognition is just a complicated problem to look at all those pixels and figure out what it is. It feels like even though the online data sets are quite big like over a million images, feels like we still wish we had more data. And there are some problems like object detection where we have even less data. So, just as a reminder image recognition was the problem of looking at a picture and telling you is this a cattle or not. Whereas object detection is look in the picture and actually you're putting the bounding boxes are telling you where in the picture the objects such as the car as well. And so because of the cost of getting the bounding boxes is just more expensive to label the objects and the bounding boxes. So, we tend to have less data for object detection than for image recognition. And object detection is something we'll discuss next week. So, if you look across a broad spectrum of machine learning problems, you see on average that when you have a lot of data you tend to find people getting away with using simpler algorithms as well as less hand-engineering. So, there's just less needing to carefully design features for the problem, but instead you can have a giant neural network, even a simpler architecture, and have a neural network. Just learn whether we want to learn we have a lot of data. Whereas, in contrast when you don't have that much data then on average you see people engaging in more hand-engineering. And if you want to be ungenerous you can say there are more hacks. But I think when you don't have much data then hand-engineering is actually the best way to get good performance. So, when I look at machine learning applications I think usually we have the learning algorithm has two sources of knowledge. One source of knowledge is the labeled data, really the (x,y) pairs you use for supervised learning. And the second source of knowledge is the hand-engineering. And there are lots of ways to hand-engineer a system. It can be from carefully hand designing the features, to carefully hand designing the network architectures to maybe other components of your system. And so when you don't have much labeled data you just have to call more on hand-engineering. And so I think computer vision is trying to learn a really complex function. And it often feels like we don't have enough data for computer vision. Even though data sets are getting bigger and bigger, often we just don't have as much data as we need. And this is why this data computer vision historically and even today has relied more on hand-engineering. And I think this is also why that either computer vision has developed rather complex network architectures, is because in the absence of more data the way to get good performance is to spend more time architecting, or fooling around with the network architecture. And in case you think I'm being derogatory of hand-engineering that's not at all my intent. When you don't have enough data hand-engineering is a very difficult, very skillful task that requires a lot of insight. And someone that is insightful with hand-engineering will get better performance, and is a great contribution to a project to do that hand-engineering when you don't have enough data. It's just when you have lots of data then I wouldn't spend time hand-engineering, I would spend time building up the learning system instead. But I think historically the fear the computer vision has used very small data sets, and so historically the computer vision literature has relied on a lot of hand-engineering. And even though in the last few years the amount of data with the right computer vision task has increased dramatically, I think that that has resulted in a significant reduction in the amount of hand-engineering that's being done. But there's still a lot of hand-engineering of network architectures and computer vision. Which is why you see very complicated hyper frantic choices in computer vision, are more complex than you do in a lot of other disciplines. And in fact, because you usually have smaller object detection data sets than image recognition data sets, when we talk about object detection that is task like this next week. You see that the algorithms become even more complex and has even more specialized components. Fortunately, one thing that helps a lot when you have little data is transfer learning. And I would say for the example from the previous slide of the tigger, misty, neither detection problem, you have soluble data that transfer learning will help a lot.



Another set of techniques that's used a lot for when you have relatively little data. If you look at the computer vision literature, and look at the sort of ideas out there, you also find that people are really enthusiastic. They're really into doing well on standardized benchmark data sets and on winning competitions. And for computer vision researchers if you do well and the benchmark is easier to get the paper published. So, there's just a lot of attention on doing well on these benchmarks. And the positive side of this is that, it helps the whole community figure out what are the most effective algorithms. But you also see in the papers people do things that allow you to do well on a benchmark, but that you wouldn't really use in a production or a system that you deploy in an actual application. So, here are a few tips on doing well on benchmarks. These are things that I don't myself pretty much ever use if I'm putting a system to production that is actually to serve customers. But one is ensembling. And what that means is, after you've figured out what neural network you want, train several neural networks independently and average their outputs. So, initialize say 3, or 5, or 7 neural networks randomly and train up all of these neural networks, and then average their outputs. And by the way it is important to average their outputs y hats. Don't average their weights that won't work. Look and you say seven neural networks that have seven different predictions and average that. And this will cause you to do maybe 1% better, or 2% better. So is a little bit better on some benchmark. And this will cause you to do a little bit better. Maybe sometimes as much as 1 or 2% which really help win a competition. But because ensembling means that to test on each image, you might need to run an image through anywhere from say 3 to 15 different networks quite typical. This slows down your running time by a factor of 3 to 15, or sometimes even more. And so ensembling is one of those tips that people use doing well in benchmarks and for winning competitions. But that I think is almost never use in production to serve actual customers. I guess unless you have huge computational budget and don't mind burning a lot more of it per customer image. Another thing you see in papers that really helps on benchmarks, is multi-crop at test time. So, what I mean by that is you've seen how you can do data augmentation. And multi-crop is a form of applying data augmentation to your test image as well. So, for example let's see a cat image and just copy it four times including two more versions. There's a technique called the 10-crop, which basically says let's say you take this central region that crop, and run it through your crossfire. And then take that crop up the left hand corner run through a crossfire, up right hand corner shown in green, lower left shown in yellow, lower right shown in orange, and run that through the crossfire. And then do the same thing with the mirrored image. Right. So I'll take the central crop, then take the four corners crops. So, that's one central crop here and here, there's four corners crop here and here. And if

you add these up that's 10 different crops that you mentioned. So hence the name 10-crop. And so what you do, is you run these 10 images through your crossfire and then average the results. So, if you have the computational budget you could do this. Maybe you don't need as many as 10-crops, you can use a few crops. And this might get you a little bit better performance in a production system. By production I mean a system you're deploying for actual users. But this is another technique that is used much more for doing well on benchmarks than in actual production systems. And one of the big problems of ensembling is that you need to keep all these different networks around. And so that just takes up a lot more computer memory. For multi-crop I guess at least you keep just one network around. So it doesn't suck up as much memory, but it still slows down your run time quite a bit. So, these are tips you see and research papers will refer to these tips as well. But I personally do not tend to use these methods when building production systems even though they are great for doing better on benchmarks and on winning competitions. Because a lot of the computer vision problems are in the small data regime, others have done a lot of hand-engineering of the network architectures. And a neural network that works well on one vision problem often may be surprisingly, but they just often would work on other vision problems as well. So, to build a practical system often you do well starting off with someone else's neural network architecture.



And you can use an open source implementation if possible, because the open source implementation might have figured out all the finicky details like the learning rate, case scheduler, and other hyper parameters. And finally someone else may have spent weeks training a model on half a dozen GP use and on over a million images. And so by using someone else's pre-trained model and fine tuning on your data set, you can often get going much faster on an application. But of course if you have the compute resources and the inclination, don't let me stop you from training your own networks from scratch and in fact if you want to invent your own computer vision algorithm, that's what you might have to do.

Week 3: Object detection

Learn how to apply your knowledge of CNNs to one of the toughest but hottest field of computer vision: Object detection

Learning Objectives

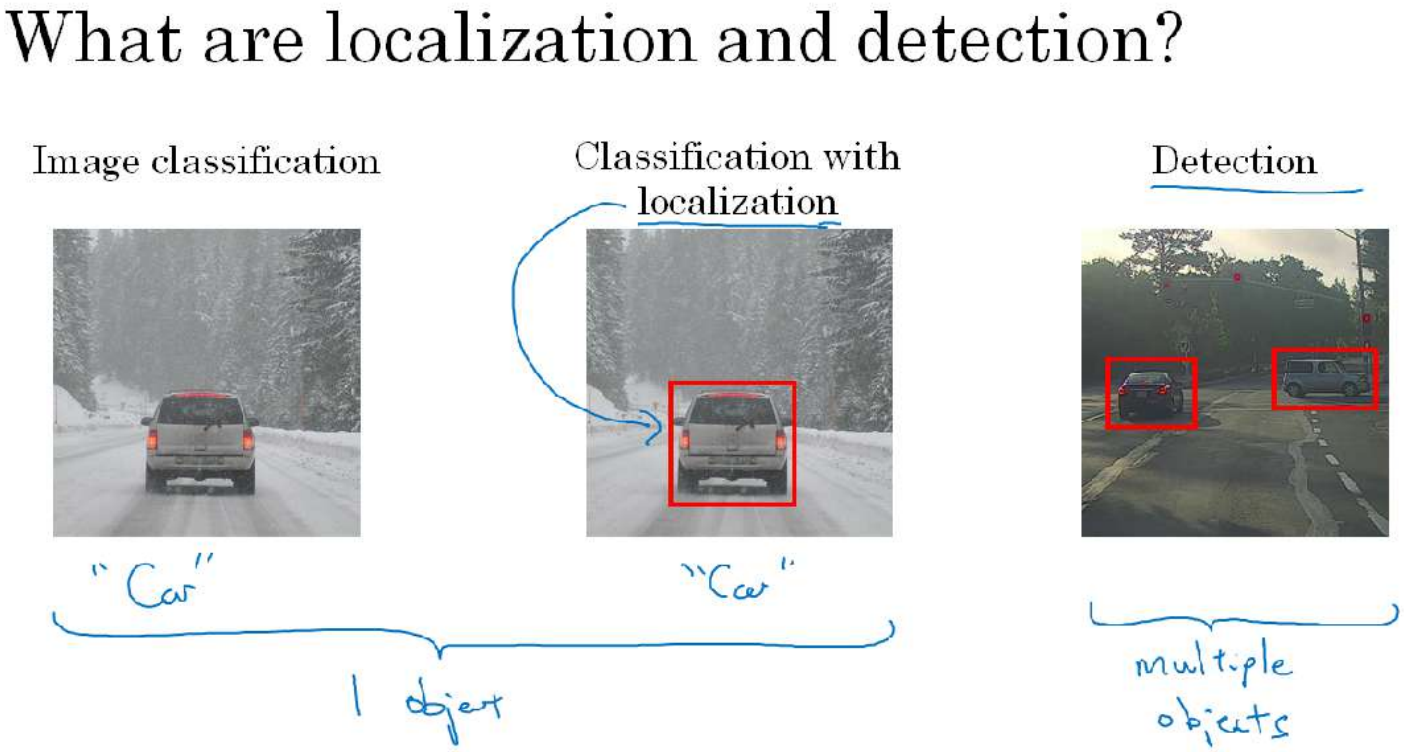
- Understand the challenges of Object Localization, Object Detection and Landmark Finding

- Understand and implement non-max suppression
- Understand and implement intersection over union
- Understand how we label a dataset for an object detection application
- Remember the vocabulary of object detection (landmark, anchor, bounding box, grid, ...)

Detection algorithms

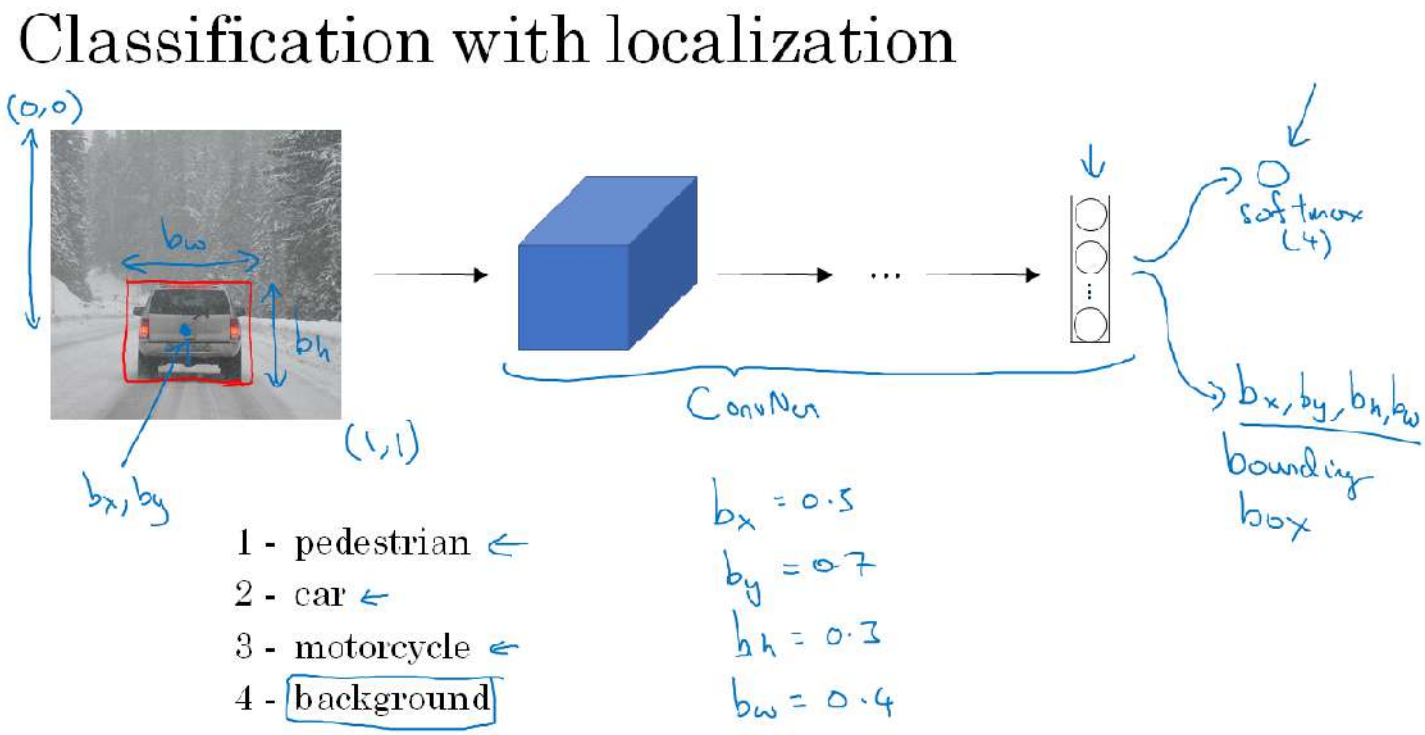
Object Localization

In this section we'll learn about **object detection**. This is one of the areas of computer vision that's just exploding and is working so much better than just a couple of years ago. In order to build up to object detection, you first learn about object localization. Let's start by defining what that means. You're already familiar with the image classification task where an algorithm looks at this picture below and might be responsible for saying this is a car. So that was classification.



So, not only do we have to label this as say a car but the algorithm also is responsible for putting a bounding box, or drawing a red rectangle around the position of the car in the image. So that's called the **classification with localization** problem. Where the term localization refers to figuring out where in the picture is the car you've detective. Later we'll also learn about the detection problem where now there might be multiple objects in the picture and you have to detect them all and and localized them all and if you're doing this for an autonomous driving application, then you might need to detect not just other cars, but maybe other pedestrians and motorcycles and maybe even other objects. So in the terminology we'll use this week, the classification and 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 you've learned about for image classification will be useful for classification with localization and that the ideas you learn for localization will then turn out to be useful for detection. So let's start by talking about classification with localization. You're already familiar with the image classification problem, in which you might input a picture into a ConfNet with multiple layers so that's our ConfNet and this results in a vector features that is fed to maybe a softmax unit that outputs the predicted output. So if you are building a self driving car, maybe your object categories are the following. Where you might have a pedestrian, or a car, or a motorcycle, or a background. This means none of the above. So if there's no pedestrian, no car, no motorcycle, then you might have an output background. So these are your classes, they have a

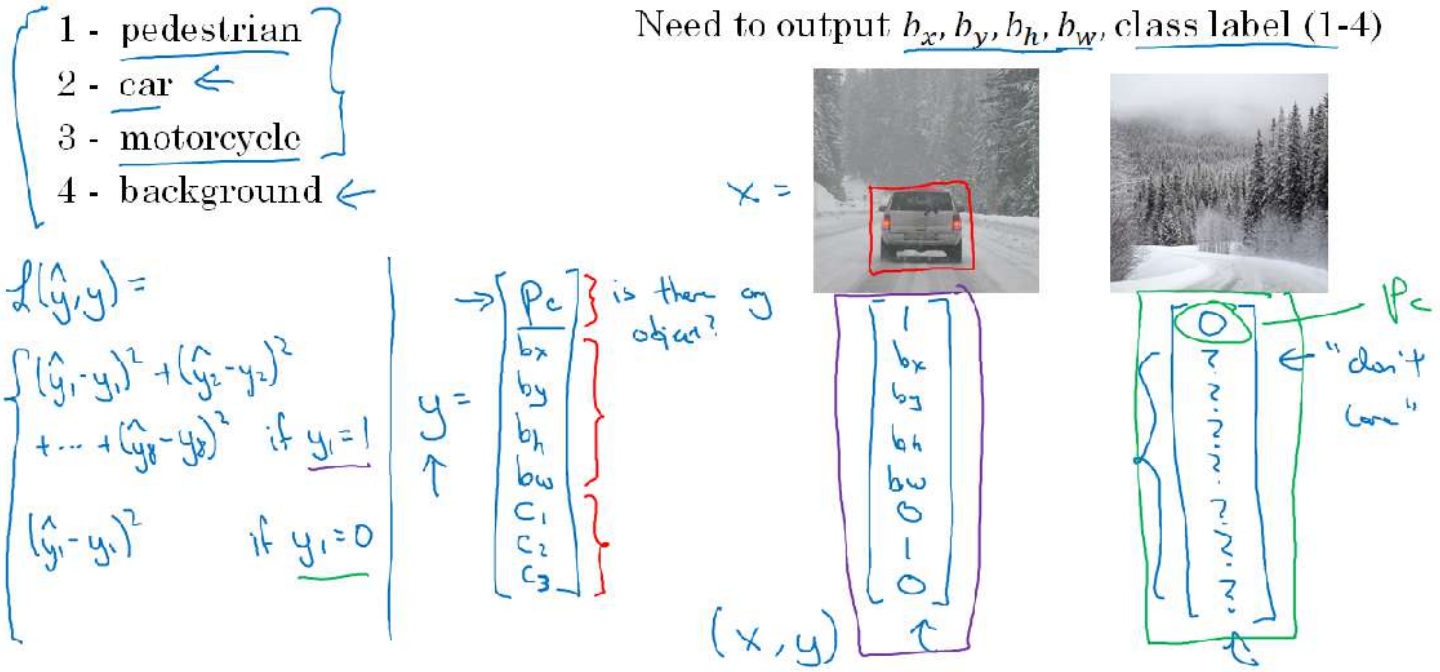
softmax with four possible outputs. So this is the standard classification pipeline. How about if you want to localize the car in the image as well. To do that, you can change your neural network to have a few more output units that output a bounding box. So, in particular, you can have the neural network output four more numbers, and I'm going to call them bx, by, bh, and bw. And these four numbers parameterized the bounding box of the detected object.



So in these sections, we are going to use the notational convention that the upper left of the image, I'm going to denote as the coordinate (0,0), and at the lower right is (1,1). So, specifying the bounding box, the red rectangle requires specifying the midpoint. So that's the point bx, by as well as the height, that would be bh, as well as the width, bw of this bounding box. So now if your training set contains not just the object cross label, which a neural network is trying to predict up here, but it also contains four additional numbers. Giving the bounding box then you can use supervised learning to make your algorithm outputs not just a class label but also the four parameters to tell you where is the bounding box of the object you detected. So in this example the ideal bx might be about 0.5 because this is about halfway to the right to the image. by might be about 0.7 since it's about maybe 70% to the way down to the image. bh might be about 0.3 because the height of this red square is about 30% of the overall height of the image. And bw might be about 0.4 let's say because the width of the red box is about 0.4 of the overall width of the entire image.

So let's formalize this a bit more in terms of how we define the target label y for this as a supervised learning task. So just as a reminder these are our four classes, and the neural network now outputs those four numbers as well as a class label, or maybe probabilities of the class labels. So, let's define the target label y as follows. Is going to be a vector where the first component pc is going to be, is there an object?

Defining the target label y



So, if the object is, classes 1, 2 or 3, pc will be equal to 1 and if it's the background class, so if it's none of the objects you're trying to detect, then pc will be 0. And pc you can think of that as standing for the probability that there's an object. Probability that one of the classes you're trying to detect is there. So something other than the background class. Next if there is an object, then you wanted to output bx, by, bh and bw, the bounding box for the object you detected and finally if there is an object, so if pc is equal to 1, you wanted to also output c1, c2 and c3 which tells us is it the class 1, class 2 or class 3. So is it a pedestrian, a car or a motorcycle and remember in the problem we're addressing we assume that your image has only one object. So at most, one of these objects appears in the picture, in this classification with localization problem. So let's go through a couple of examples. If this is a training set image, so if that is x, then y will be the first component pc will be equal to 1 because there is an object, then bx, by, by, bh and bw will specify the bounding box. So your labeled training set will need bounding boxes in the labels and then finally this is a car, so it's class 2. So c1 will be 0 because it's not a pedestrian, c2 will be 1 because it is car, c3 will be 0 since it is not a motorcycle. So among c1, c2 and c3 at most one of them should be equal to 1. So that's if there's an object in the image. What if there's no object in the image? What if we have a training example where x is equal to that? In this case, pc would be equal to 0, and the rest of the elements of this, will be don't cares, so I'm going to write question marks in all of them. So this is a don't care, because if there is no object in this image, then you don't care what bounding box the neural network outputs as well as which of the three objects, c1, c2, c3 it thinks it is. So given a set of label training examples, this is how you will construct x, the input image as well as y, the cost label both for images where there is an object and for images where there is no object and the set of this will then define your training set. Finally, let's describe the loss function you use to train the neural network. So the ground true label was y and the neural network outputs some yhat. What should be the loss be? Well if you're using squared error then the loss can be (y1 hat- y1) squared + (y2 hat- y2) squared + ...+(y8 hat- y8) squared. Notice that y here has eight components. So that goes from sum of the squares of the difference of the elements and that's the loss if y1=1. So that's the case where there is an object. So y1= pc. So, pc = 1, that if there is an object in the image then the loss can be the sum of squares of all the different elements.

The other case is if y1=0, so that's if this pc = 0. In that case the loss can be just (y1 hat-y1) squared, because in that second case, all of the rest of the components are don't care us and so all you care about is how accurately is the neural network ourputting pc in that case.

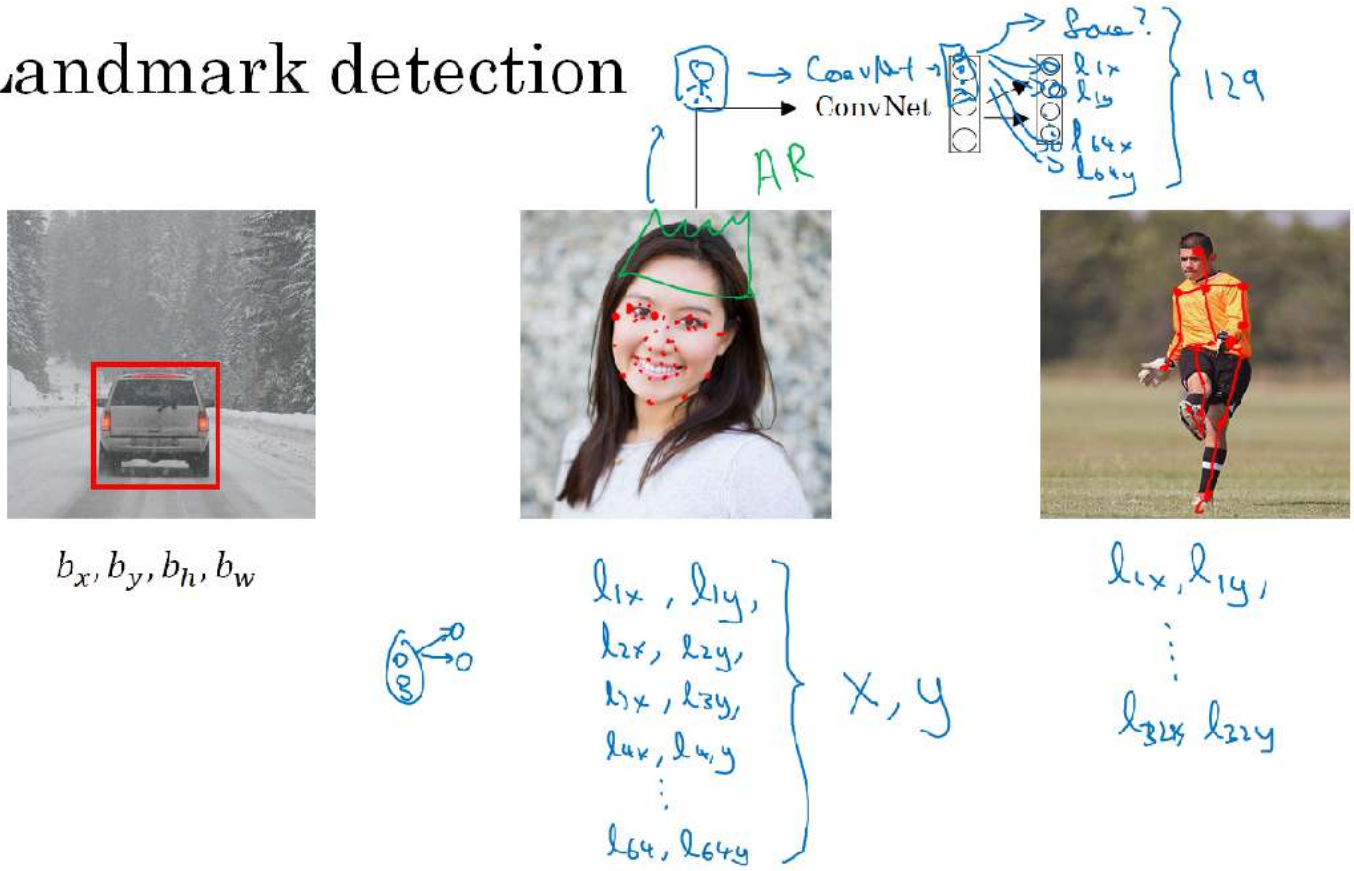
So just a recap, if y1 = 1, that's this case, then you can use squared error to penalize square deviation from the predicted, and the actual output of all eight components. Whereas if y1 = 0, then the second to the eighth components I don't care. So all you care about is how accurately is your neural network estimating y1, which is equal to pc.

Landmark Detection

In the previous section, we saw how you can get a neural network to output four numbers of bx, by, bh, and bw to specify the bounding box of an object you want a neural network to localize. In more general cases, you can have a neural network just output X and Y coordinates of important points and image, sometimes called **landmarks**, that you want the neural networks to recognize. Let's go through few examples.

Let's say you're building a face recognition application and for some reason, you want the algorithm to tell you where is the corner of someone's eye. So that point has an X and Y coordinate, so you can just have a neural network have its final layer and have it just output two more numbers which I'm going to call our lx and ly to just tell you the coordinates of that corner of the person's eye. Now, what if you want it to tell you all four corners of the eye, really of both eyes. So, if we call the points, the first, second, third and fourth points going from left to right, then you could modify the neural network now to output l_{1x}, l_{1y} for the first point and l_{2x}, l_{2y} for the second point and so on, so that the neural network can output the estimated position of all those four points of the person's face. But what if you don't want just those four points? What do you want to output this point, and this point and this point and this point along the eye? Maybe I'll put some key points along the mouth, so you can extract the mouth shape and tell if the person is smiling or frowning, maybe extract a few key points along the edges of the nose but you could define some number, for the sake of argument, let's say 64 points or 64 landmarks on the face. Maybe even some points that help you define the edge of the face, defines the jaw line but by selecting a number of landmarks and generating a label training sets that contains all of these landmarks, you can then have the neural network to tell you where are all the key positions or the key landmarks on a face.

Landmark detection

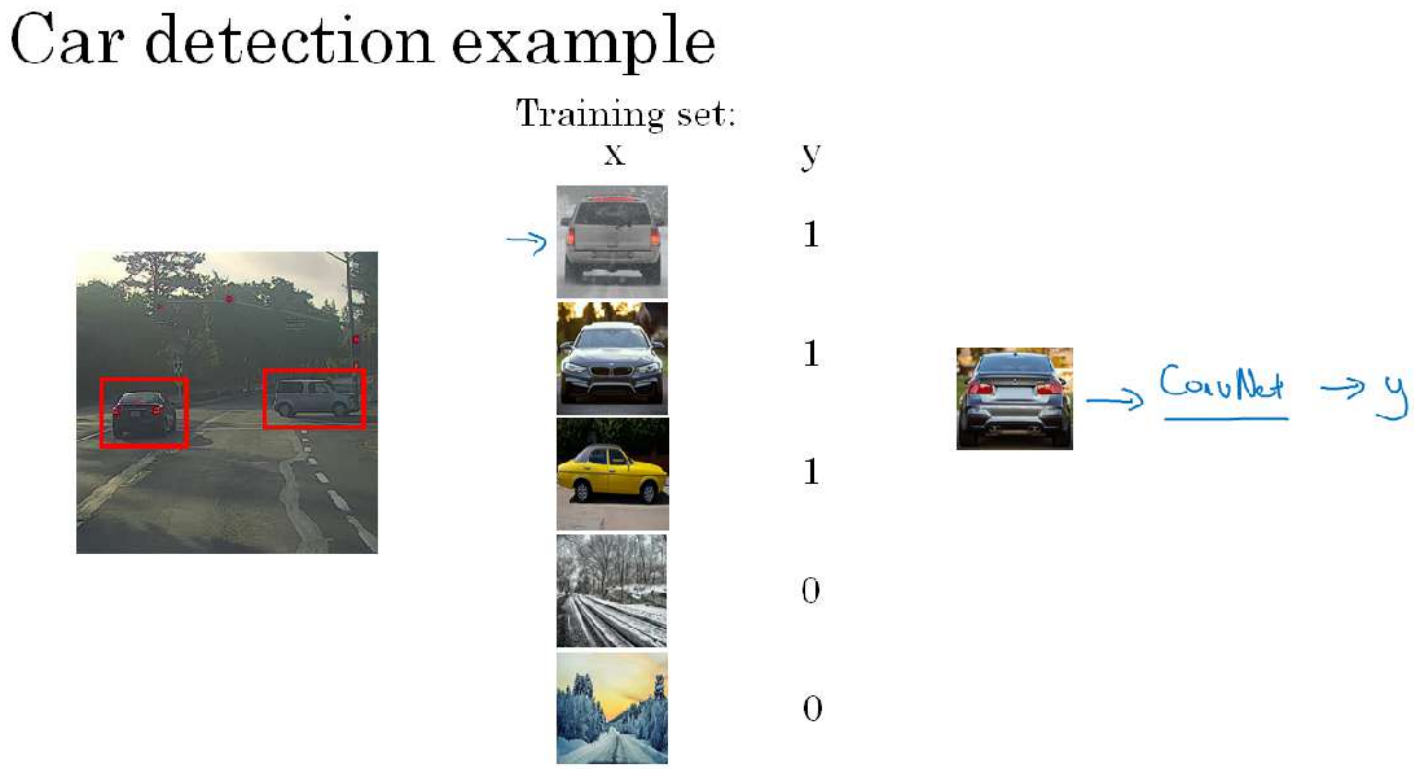


So what you do is you have this image, a person's face as input, have it go through a convnet and then have some set of features, maybe have it output 0 or 1, like zero face changes or not and then have it also output l_{1x}, l_{1y} and so on down to l_{64x}, l_{64y} and here I'm using **l** to stand for a landmark. So this example would have 129 output units, one for is your face or not? and then if you have 64 landmarks, that's sixty-four times two, so 128 plus one output units and this can tell you if there's a face as well as where all the key landmarks on the face. So, this is a basic building block for recognizing emotions from faces and if you played with the Snapchat and the other entertainment, also AR augmented reality filters like the Snapchat photos can draw a crown on the face and have other special effects. Being able to detect these landmarks on the face, there's also a key building block for the computer graphics effects that warp the face or drawing various special effects like putting a crown or a hat on the person. Of course, in order to treat a network like this, you will need a label training set. We have a set of images as well as labels Y where people, where someone will have had to go through and laboriously annotate all of these landmarks. One last example, if you are interested in people pose detection, you could also define a few key positions like the midpoint of the chest, the left shoulder, left elbow, the wrist, and so on, and just have a neural network to annotate key positions in the person's pose as well and by having a neural network output, all of those points I'm annotating, you could also have the neural network output the pose of the person and of course, to do that you also need to specify on these key landmarks like maybe l_{1x} and l_{1y} is the midpoint of the chest down to maybe l_{32x}, l_{32y} if you use 32 coordinates to specify the pose of the person. So, this idea might seem quite simple of just adding a bunch of output units to output the X,Y coordinates of different landmarks you want to recognize. To be clear, the identity of landmark one must be consistent across different images like maybe landmark one is always this corner of the eye, landmark two is always this corner of the eye, landmark three, landmark four, and so on. So, the labels have to be consistent across different images but if you can hire labelers or label yourself a big enough data set to do this, then a neural network can output all of these landmarks which is going to used to carry out other interesting effect such as with the pose of the person, maybe try to recognize someone's emotion from a picture, and so on. So that's it for **landmark detection**. Next, let's take these building blocks and use it to start building up towards object detection.

Object Detection

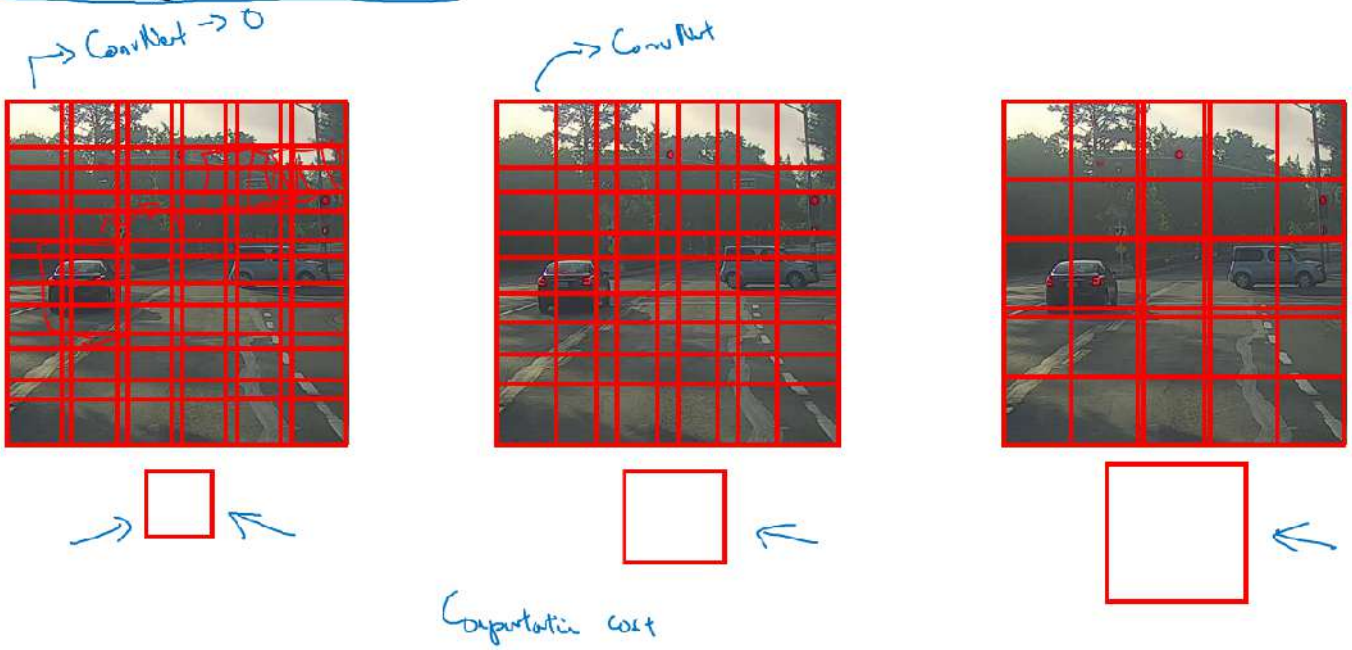
We've learned about Object Localization as well as Landmark Detection. Now, let's build up to other object detection algorithm. In this section, we'll learn how to use a ConvNet to perform object detection using something called the **Sliding Windows Detection Algorithm**.

Let's say you want to build a car detection algorithm. Here's what you can do. You can first create a label training set, so x and y with closely cropped examples of cars.



So, this is image x has a positive example, there's a car, here's a car, here's a car, and then there's not a car, there's not a car and for our purposes in this training set, you can start off with the one with the car closely cropped images. Meaning that x is pretty much only the car. So, you can take a picture and crop out and just cut out anything else that's not part of a car. So you end up with the car centered in pretty much the entire image. Given this label training set, you can then train a ConvNet that inputs an image, like one of these closely cropped images and then the job of the ConvNet is to output y, zero or one, is there a car or not. Once you've trained up this ConvNet, you can then use it in Sliding Windows Detection. So the way you do that is, if you have a test image, what you do is you start by picking a certain window size, shown down there and then you would input into this ConvNet a small rectangular region. So, take just this below red square, input that into the ConvNet, and have a ConvNet make a prediction and presumably for that little region in the red square, it'll say, no that little red square does not contain a car.

Sliding windows detection

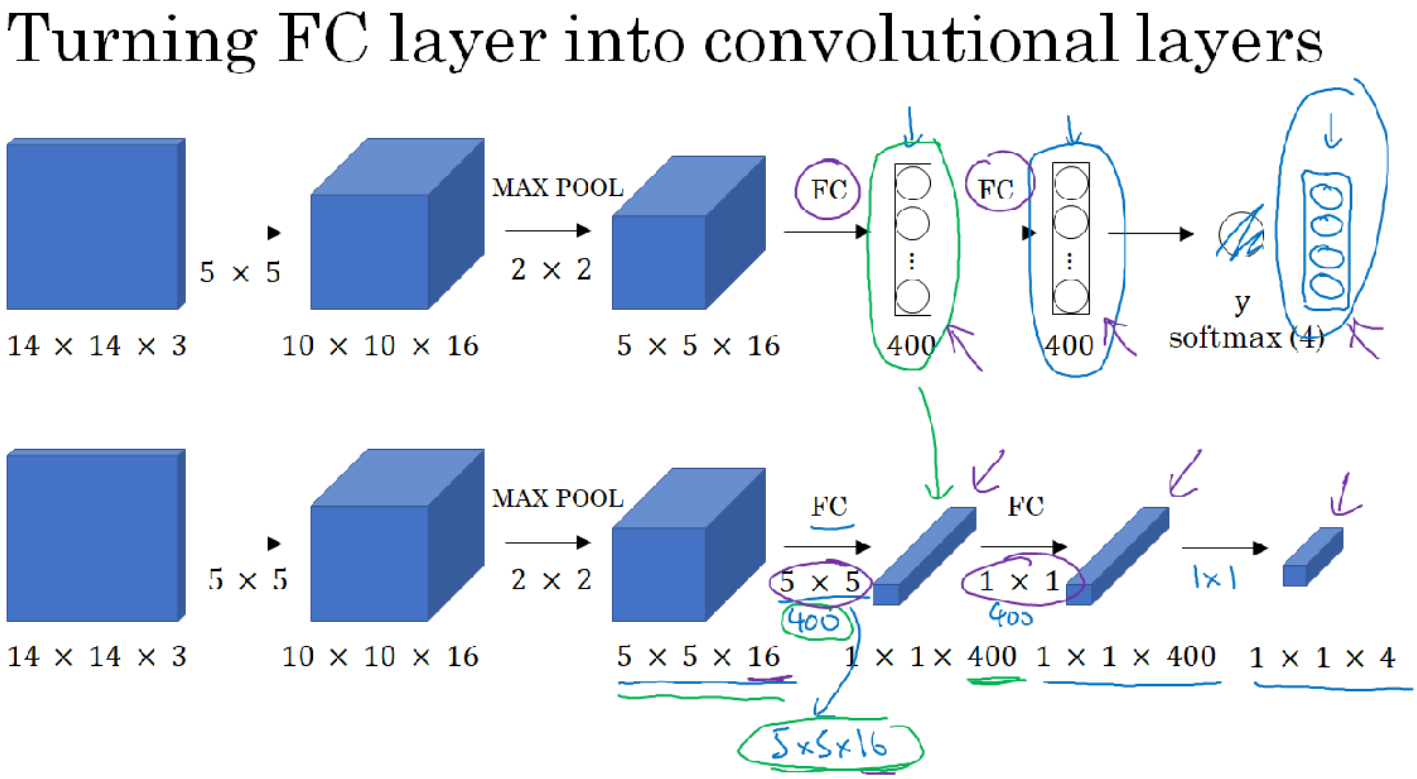


In the Sliding Windows Detection Algorithm, what you do is you then pass as input a second image now bounded by this red square shifted a little bit over and feed that to the ConvNet. So, you're feeding just the region of the image in the red squares of the ConvNet and run the ConvNet again and then you do that with a third image and so on and you keep going until you've slid the window across every position in the image and I'm using a pretty large stride in this example just to make the animation go faster. But the idea is you basically go through every region of this size, and pass lots of little cropped images into the ConvNet and have it classified zero or one for each position as some stride. Now, having done this once with running this was called the sliding window through the image. You then repeat it, but now use a larger window. So, now you take a slightly larger region and run that region. So, resize this region into whatever input size the cofinite is expecting, and feed that to the ConvNet and have it output zero or one and then slide the window over again using some stride and so on and you run that throughout your entire image until you get to the end and then you might do the third time using even larger windows and so on. Right. And the hope is that if you do this, then so long as there's a car somewhere in the image that there will be a window where, for example if you are passing in this window into the ConvNet, hopefully the ConvNet will have outputs one for that input region. So then you detect that there is a car there. So this algorithm is called **Sliding Windows Detection because you take these windows, these square boxes, and slide them across the entire image and classify every square region with some stride as containing a car or not.** Now there's a huge disadvantage of Sliding Windows Detection, which is the computational cost. Because you're cropping out so many different square regions in the image and running each of them independently through a ConvNet and if you use a very coarse stride, a very big stride, a very big step size, then that will reduce the number of windows you need to pass through the cofinite, but that courser granularity may hurt performance. Whereas if you use a very fine granularity or a very small stride, then the huge number of all these little regions you're passing through the ConvNet means that means there is a very high computational cost. So, before the rise of Neural Networks people used to use much simpler classifiers like a simple linear classifier over hand engineer features in order to perform object detection and in that era because each classifier was relatively cheap to compute, it was just a linear function, Sliding Windows Detection ran okay. It was not a bad method, but with ConvNet now running a single classification task is much more expensive and sliding windows this way is infeasibily slow. And unless you use a very fine granularity or a very small stride, you end up not able to localize the objects that accurately within the image as well. Fortunately however, this problem of computational cost has a pretty good

solution. In particular, the Sliding Windows Object Detector can be implemented convolutionally or much more efficiently.

Convolutional Implementation of Sliding Windows

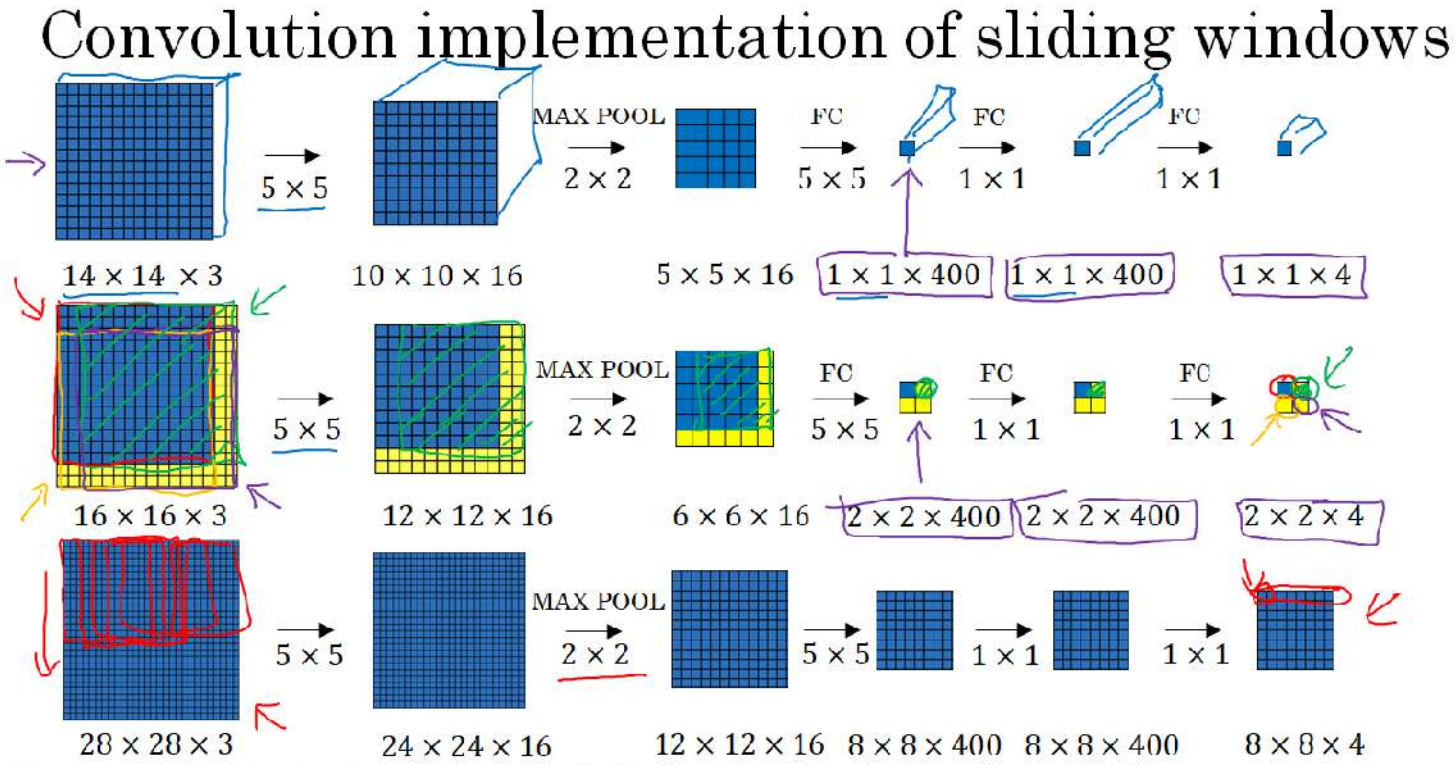
In the last section, we've learned about the sliding windows object detection algorithm using a convnet but we saw that it was too slow. In this section, we'll learn how to implement that algorithm convolutionally. Let's see what this means. To build up towards the convolutional implementation of sliding windows let's first see how you can turn fully connected layers in neural network into convolutional layers. We'll do that first on this slide and then the next slide, we'll use the ideas from this slide to show you the convolutional implementation. So let's say that your object detection algorithm inputs 14x14x3 images. This is quite small but just for illustrative purposes, and let's say it then uses 5x5 filters, and let's say it uses 16 of them to map it from 14x14x3 to 10x10x16. And then does a 2x2 max pooling to reduce it to 5x5x16. Then has a fully connected layer to connect to 400 units. Then now they're fully connected layer and then finally outputs a Y using a softmax unit.



Lets take Y as four numbers, corresponding to the cause probabilities of the four causes that softmax units is classified amongst and the full causes could be pedestrian, car, motorcycle, and background or something else. Now, what we'd like to do is show how these layers can be turned into convolutional layers. So, the convnet will draw same as before for the first few layers. And now, one way of implementing this next layer, this fully connected layer is to implement this as a 5 by 5 filter and let's use 400, 5 by 5 filters. So if you take a 5 by 5 by 16 image and convolve it with a 5 by 5 filter, remember, a 5 by 5 filter is implemented as 5 by 5 by 16 because our convention is that the filter looks across all 16 channels. So the 16 must match with other 16 so the outputs will be 1 by 1 and if you have 400 of these 5 by 5 by 16 filters, then the output dimension is going to be 1x1x400. So rather than viewing these 400 as just a set of nodes, we're going to view this as a 1x1x400 volume. Mathematically, this is the same as a fully connected layer because each of these 400 nodes has a filter of dimension 5 by 5 by 16. So each of those 400 values is some arbitrary linear function of these 5 by 5 by 16 activations from the previous layer.

Next, to implement the next convolutional layer, we're going to implement a 1 by 1 convolution. If you have 400 1 by 1 filters then, with 400 filters the next layer will again be 1 by 1 by 400. So that gives you this next fully connected layer and then finally, we're going to have another 1 by 1

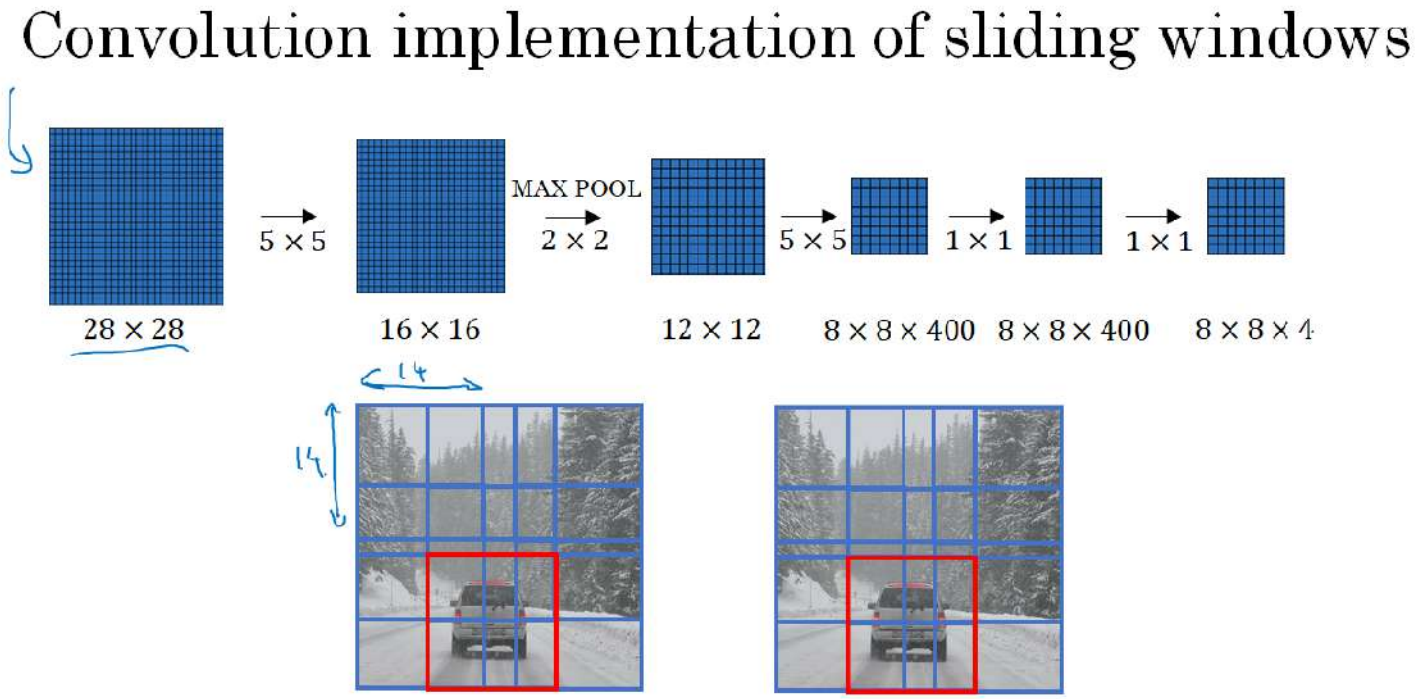
filter, followed by a softmax activation. So as to give a 1 by 1 by 4 volume to take the place of these four numbers that the network was operating. So this shows how you can take these fully connected layers and implement them using convolutional layers so that these sets of units instead are not implemented as 1 by 1 by 400 and 1 by 1 by 4 volumes. After this conversion, let's see how you can have a convolutional implementation of sliding windows object detection.



[Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks]

The presentation on this slide is based on the OverFeat paper, referenced at the bottom, by Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Robert Fergus and Yann Lecun. Let's say that your sliding windows convnet inputs 14 by 14 by 3 images and again, I'm just using small numbers like the 14 by 14 image in this slide mainly to make the numbers and illustrations simpler. So as before, you have a neural network as follows that eventually outputs a 1 by 1 by 4 volume, which is the output of your softmax. Again, to simplify the drawing here, 14 by 14 by 3 is technically a volume 5 by 5 or 10 by 10 by 16, the second clear volume. But to simplify the drawing for this slide, I'm just going to draw the front face of this volume. So instead of drawing 1 by 1 by 400 volume, I'm just going to draw the 1 by 1 cause of all of these. So just dropped the three components of these drawings, just for this slide. So let's say that your convnet inputs 14 by 14 images or 14 by 14 by 3 images and your tested image is 16 by 16 by 3. So now added that yellow stripe to the border of this image. In the original sliding windows algorithm, you might want to input the blue region into a convnet and run that once to generate a consecration 01 and then slightly down a bit, least he uses a stride of two pixels and then you might slide that to the right by two pixels to input this green rectangle into the convnet and we run the whole convnet and get another label, 01. Then you might input this orange region into the convnet and run it one more time to get another label. And then do it the fourth and final time with this lower right purple square. To run sliding windows on this 16 by 16 by 3 image is pretty small image. You run this convnet four times in order to get four labels. But it turns out a lot of this computation done by these four convnets is highly duplicative. So what the convolutional implementation of sliding windows does is it allows these four pauses in the convnet to share a lot of computation. Specifically, here's what you can do. You can take the convnet and just run it same parameters, the same 5 by 5 filters, also 16, 5 by 5 filters and run it. Now, you can have a 12 by 12 by 16 output volume. Then do the max pool, same as before. Now you have a 6 by 6 by 16, runs through your same 400, 5 by 5 filters to get now your 2 by 2 by 40 volume. So now instead of a 1 by 1 by 400 volume, we have instead a 2 by 2 by 400 volume. Run it through a 1 by 1 filter gives

you another 2 by 2 by 400 instead of 1 by 1 like 400. Do that one more time and now you're left with a 2 by 2 by 4 output volume instead of 1 by 1 by 4. It turns out that this blue 1 by 1 by 4 subset gives you the result of running in the upper left hand corner 14 by 14 image. This upper right 1 by 1 by 4 volume gives you the upper right result. The lower left gives you the results of implementing the convnet on the lower left 14 by 14 region. And the lower right 1 by 1 by 4 volume gives you the same result as running the convnet on the lower right 14 by 14 medium. And if you step through all the steps of the calculation, let's look at the green example, if you had cropped out just this region and passed it through the convnet through the convnet on top, then the first layer's activations would have been exactly this region. The next layer's activation after max pooling would have been exactly this region and then the next layer, the next layer would have been as follows. So what this process does, what this convolution implementation does is, instead of forcing you to run four propagation on four subsets of the input image independently, Instead, it combines all four into one form of computation and shares a lot of the computation in the regions of image that are common. So all four of the 14 by 14 patches we saw here.
Now let's just go through a bigger example.

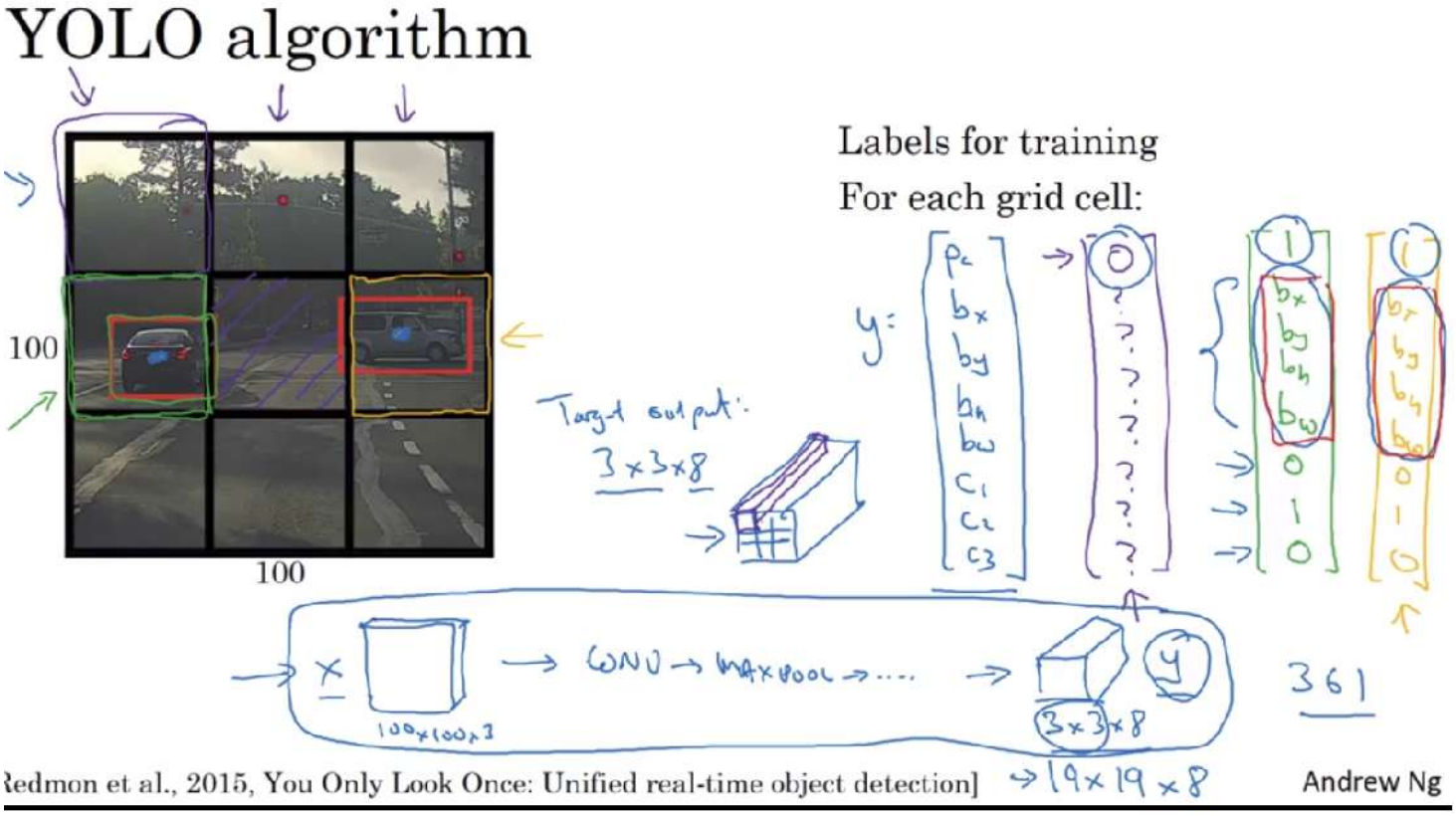


Let's say you now want to run sliding windows on a 28 by 28 by 3 image. It turns out If you run four from the same way then you end up with an 8 by 8 by 4 output. And just go small and surviving sliding windows with that 14 by 14 region. And that corresponds to running a sliding windows first on that region thus, giving you the output corresponding the upper left hand corner. Then using a slider too to shift one window over, one window over, one window over and so on and the eight positions. So that gives you this first row and then as you go down the image as well, that gives you all of these 8 by 8 by 4 outputs. Because of the max pooling up too that this corresponds to running your neural network with a stride of two on the original image. So just to recap, to implement sliding windows, previously, what you do is you crop out a region. Let's say this is 14 by 14 and run that through your convnet and do that for the next region over, then do that for the next 14 by 14 region, then the next one, then the next one, then the next one, then the next one and so on, until hopefully that one recognizes the car. But now, instead of doing it sequentially, with this convolutional implementation that you saw in the previous slide, you can implement the entire image, all maybe 28 by 28 and convolutionally make all the predictions at the same time by one forward pass through this big convnet and hopefully have it recognize the position of the car. So that's how you implement sliding windows convolutionally and it makes the whole thing much more efficient. Now, this still has one weakness, which is the position of the bounding boxes is not going to be too accurate.

Bounding Box Predictions

In the last section we've learned how to use a convolutional implementation of sliding windows. That's more computationally efficient, but it still has a problem of not quite outputting the most accurate bounding boxes. In this section, let's see how you can get your bounding box predictions to be more accurate. With sliding windows, you take this three sets of locations and run the crossfire through it and in this case, none of the boxes really match up perfectly with the position of the car. So, maybe that box is the best match and also, it looks like in drawn through, the perfect bounding box isn't even quite square, it's actually has a slightly wider rectangle or slightly horizontal aspect ratio. So, is there a way to get this algorithm to outputs more accurate bounding boxes? A good way to get this output more accurate bounding boxes is with the **YOLO algorithm**. YOLO stands for, You Only Look Once. And is an algorithm due to Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi. Here's what you do.

Let's say you have an input image at 100 by 100, you're going to place down a grid on this image and for the purposes of illustration, we're going to use a 3 by 3 grid. Although in an actual implementation, you use a finer one, like maybe a 19 by 19 grid and the basic idea is you're going to take the image classification and localization algorithm that you saw a few sections back, and apply it to each of the nine grids and the basic idea is you're going to take the image classification and localization algorithm that you saw in the first section of this week and apply that to each of the nine grid cells of this image. So the more concrete, here's how you define the labels you use for training. So for each of the nine grid cells, you specify a label Y, where the label Y is this eight dimensional vector, same as you saw previously. Your first output PC 01 depending on whether or not there's an image in that grid cell and then BX, BY, BH, BW to specify the bounding box if there is an image, if there is an object associated with that grid cell. And then say, C1, C2, C3, if you try and recognize three classes not counting the background class. So you try to recognize pedestrian's class, motorcycles and the background class. Then C1 C2 C3 can be the pedestrian, car and motorcycle classes.



So in this image, we have nine grid cells, so you have a vector like this for each of the grid cells. So let's start with the upper left grid cell, this one up here. For that one, there is no object. So, the label vector Y for the upper left grid cell would be zero, and then don't cares for the rest of these. The output label Y would be the same for this grid cell, and this grid cell, and all the grid cells with nothing, with no interesting object in them. Now, how about this grid cell? To give a bit

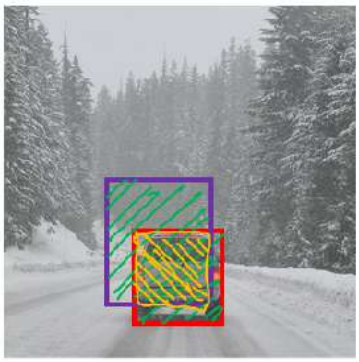
more detail, this image has two objects. And what the **YOLO algorithm does is it takes the midpoint of reach of the two objects and then assigns the object to the grid cell containing the midpoint. So the left car is assigned to this grid cell, and the car on the right, which is this midpoint, is assigned to this grid cell**. And so even though the central grid cell has some parts of both cars, we'll pretend the central grid cell has no interesting object so that the central grid cell the class label Y also looks like this vector with no object, and so the first component PC, and then the rest are don't cares. Whereas for this cell, this cell that I have circled in green on the left, the target label Y would be as follows. There is an object, and then you write BX, BY, BH, BW, to specify the position of this bounding box. And then you have, let's see, if class one was a pedestrian, then that was zero. Class two is a car, that's one. Class three was a motorcycle, that's zero. And then similarly, for the grid cell on their right because that does have an object in it, it will also have some vector like this as the target label corresponding to the grid cell on the right. So, for each of these nine grid cells, you end up with a eight dimensional output vector. And because you have 3 by 3 grid cells, you have nine grid cells, the total volume of the output is going to be 3 by 3 by 8. So the target output is going to be 3 by 3 by 8 because you have 3 by 3 grid cells. And for each of the 3 by 3 grid cells, you have a eight dimensional Y vector. So the target output volume is 3 by 3 by 8. Where for example, this 1 by 1 by 8 volume in the upper left corresponds to the target output vector for the upper left of the nine grid cells. And so for each of the 3 by 3 positions, for each of these nine grid cells, does it correspond in eight dimensional target vector Y that you want to the output. Some of which could be don't cares, if there's no object there. And that's why the total target outputs, the output label for this image is now itself a 3 by 3 by 8 volume. So now, to train your neural network, the input is 100 by 100 by 3, that's the input image. And then you have a usual convnet with conv, layers of max pool layers, and so on. So that in the end, you have this, should choose the conv layers and the max pool layers, and so on, so that this eventually maps to a 3 by 3 by 8 output volume. And so what you do is you have an input X which is the input image like that, and you have these target labels Y which are 3 by 3 by 8, and you use map propagation to train the neural network to map from any input X to this type of output volume Y. So the advantage of this algorithm is that the neural network outputs precise bounding boxes as follows. So at test time, what you do is you feed an input image X and run forward prop until you get this output Y. And then for each of the nine outputs of each of the 3 by 3 positions in which of the output, you can then just read off 1 or 0. Is there an object associated with that one of the nine positions? And that there is an object, what object it is, and where is the bounding box for the object in that grid cell? And so long as you don't have more than one object in each grid cell, this algorithm should work okay. And the problem of having multiple objects within the grid cell is something we'll address later. Of use a relatively small 3 by 3 grid, in practice, you might use a much finer, grid maybe 19 by 19. So you end up with 19 by 19 by 8, and that also makes your grid much finer. It reduces the chance that there are multiple objects assigned to the same grid cell. And just as a reminder, the way you assign an object to grid cell as you look at the midpoint of an object and then you assign that object to whichever one grid cell contains the midpoint of the object. So each object, even if the objects spends multiple grid cells, that object is assigned only to one of the nine grid cells, or one of the 3 by 3, or one of the 19 by 19 grid cells. Algorithm of a 19 by 19 grid, the chance of an object of two midpoints of objects appearing in the same grid cell is just a bit smaller. So notice two things, first, this is a lot like the image classification and localization algorithm that we talked about in the first video of this week. And that it outputs the bounding balls coordinates explicitly. And so this allows in your network to output bounding boxes of any aspect ratio, as well as, output much more precise coordinates that aren't just dictated by the stripe size of your sliding windows classifier. And second, this is a convolutional implementation and you're not implementing this algorithm nine times on the 3 by 3 grid or if you're using a 19 by 19 grid.19 squared is 361. So, you're not running the same algorithm 361 times or 19 squared times. Instead, this is one single convolutional implantation, where you use one consonant with a lot of shared computation between all the computations needed for all of your 3 by 3 or all of your 19 by 19 grid cells. So, this is a pretty efficient algorithm. And in fact, one nice thing about the YOLO algorithm, which is constant popularity is because this is a convolutional implementation, it actually runs very fast. So

this works even for real time object detection. Now, before wrapping up, there's one more detail I want to share with you, which is, how do you encode these bounding boxes bx, by, BH, BW? Let's discuss that on the next slide. So, given these two cars, remember, we have the 3 by 3 grid. Let's take the example of the car on the right. So, in this grid cell there is an object and so the target label y will be one, that was PC is equal to one. And then bx, by, BH, BW, and then 0 1 0. So, how do you specify the bounding box? In the YOLO algorithm, relative to this square, when I take the convention that the upper left point here is 0 0 and this lower right point is 1 1. So to specify the position of that midpoint, that orange dot, bx might be, let's say x looks like is about 0.4. Maybe its about 0.4 of the way to their right. And then y, looks I guess maybe 0.3. And then the height of the bounding box is specified as a fraction of the overall width of this box. So, the width of this red box is maybe 90% of that blue line. And so BH is 0.9 and the height of this is maybe one half of the overall height of the grid cell. So in that case, BW would be, let's say 0.5. So, in other words, this bx, by, BH, BW as specified relative to the grid cell. And so bx and by, this has to be between 0 and 1, right? Because pretty much by definition that orange dot is within the bounds of that grid cell is assigned to. If it wasn't between 0 and 1 it was outside the square, then we'll have been assigned to a different grid cell. But these could be greater than one. In particular if you have a car where the bounding box was that, then the height and width of the bounding box, this could be greater than one. So, there are multiple ways of specifying the bounding boxes, but this would be one convention that's quite reasonable. Although, if you read the YOLO research papers, the YOLO research line there were other parameterizations that work even a little bit better, but I hope this gives one reasonable condition that should work okay. Although, there are some more complicated parameterizations involving sigmoid functions to make sure this is between 0 and 1. And using an explanation parameterization to make sure that these are non-negative, since 0.9, 0.5, this has to be greater or equal to zero. There are some other more advanced parameterizations that work things a little bit better, but the one you saw here should work okay. So, that's it for the **YOLO** or the **You Only Look Once** algorithm. And in the next few videos I'll show you a few other ideas that will help make this algorithm even better. In the meantime, if you want, you can take a look at YOLO paper reference at the bottom of these past couple slides I use. Although, just one warning, if you take a look at these papers which is the YOLO paper is one of the harder papers to read. I remember, when I was reading this paper for the first time, I had a really hard time figuring out what was going on. And I wound up asking a couple of my friends, very good researchers to help me figure it out, and even they had a hard time understanding some of the details of the paper. So, if you look at the paper, it's okay if you have a hard time figuring it out. I wish it was more uncommon, but it's not that uncommon, sadly, for even senior researchers, that review research papers and have a hard time figuring out the details. And have to look at open source code, or contact the authors, or something else to figure out the details of these outcomes. But don't let me stop you from taking a look at the paper yourself though if you wish, but this is one of the harder ones. So, that though, you now understand the basics of the YOLO algorithm. Let's go on to some additional pieces that will make this algorithm work even better.

Intersection Over Union

So how do you tell if your object detection algorithm is working well? In this section, we'll learn about a function called, "**Intersection Over Union**". and as we use both for evaluating your object detection algorithm, as well as in the next section, using it to add another component to your object detection algorithm, to make it work even better. Let's get started. In the object detection task, you expected to localize the object as well.

Evaluating object localization



Intersection over Union (IoU)

$$= \frac{\text{Size of}}{\text{Size of}}$$

"Correct" if IoU $\geq$ 0.5 ←

0.6 ←

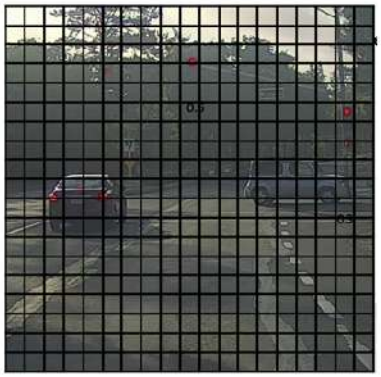
More generally, IoU is a measure of the overlap between two bounding boxes.

So if that's the ground-truth bounding box, and if your algorithm outputs this bounding box in purple, is this a good outcome or a bad one? So what the intersection over union function does, or IoU does, is it computes the intersection over union of these two bounding boxes. So, the union of these two bounding boxes is this area, is really the area that is contained in either bounding boxes, whereas the intersection is this smaller region here. So what the intersection of a union does is it computes the size of the intersection. So that orange shaded area, and divided by the size of the union, which is that green shaded area. And by convention, the low compute division task will judge that your answer is correct if the IoU is greater than 0.5. And if the predicted and the ground-truth bounding boxes overlapped perfectly, the IoU would be one, because the intersection would equal to the union. But in general, so long as the IoU is greater than or equal to 0.5, then the answer will look okay, look pretty decent. And by convention, very often 0.5 is used as a threshold to judge as whether the predicted bounding box is correct or not. This is just a convention. If you want to be more stringent, you can judge an answer as correct, only if the IoU is greater than equal to 0.6 or some other number. But the higher the IoUs, the more accurate the bounding the box. And so, this is one way to map localization, to accuracy where you just count up the number of times an algorithm correctly detects and localizes an object where you could use a definition like this, of whether or not the object is correctly localized. And again 0.5 is just a human chosen convention. There's no particularly deep theoretical reason for it. You can also choose some other threshold like 0.6 if you want to be more stringent. I sometimes see people use more stringent criteria like 0.6 or maybe 0.7. I rarely see people drop the threshold below 0.5. Now, what motivates the definition of IoU, as a way to evaluate whether or not your object localization algorithm is accurate or not. But more generally, IoU is a measure of the overlap between two bounding boxes. Where if you have two boxes, you can compute the intersection, compute the union, and take the ratio of the two areas. And so this is also a way of measuring how similar two boxes are to each other. And we'll see this use again this way in the next video when we talk about non-max suppression. So that's it for IoU or Intersection over Union. Not to be confused with the promissory note concept in IoU, where if you lend someone money they write you a note that says, " Oh I owe you this much money," so that's also called an IoU. It's totally a different concept, that maybe it's cool that these two things have a similar name. So now, onto this definition of IoU, Intersection of Union. In the next section, we'll discuss non-max suppression, which is a tool you can use to make the outputs of YOLO work even better.

Non-max Suppression

One of the problems of Object Detection as you've learned about this so far, is that your algorithm may find multiple detections of the same objects. Rather than detecting an object just once, it might detect it multiple times. Non-max suppression is a way for you to make sure that your algorithm detects each object only once.

Non-max suppression algorithm



19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3

Discard all boxes with $p_c \leq 0.6$

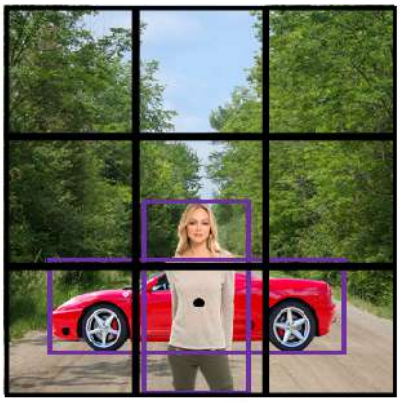
While there are any remaining boxes:

- 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

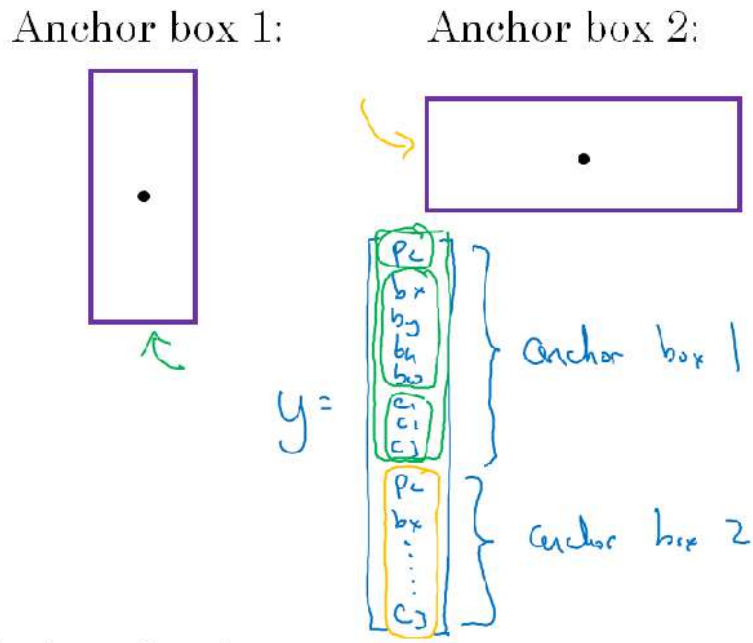
Anchor Boxes

One of the problems with object detection as you have seen it so far is that each of the grid cells can detect only one object. What if a grid cell wants to detect multiple objects? Here is what you can do. You can use the idea of anchor boxes. Let's start with an example. Let's say you have an image (see below).

Overlapping objects:



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



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

And for this example, I am going to continue to use a 3 by 3 grid. Notice that the midpoint of the pedestrian and the midpoint of the car are in almost the same place and both of them fall into the same grid cell. So, for that grid cell, if Y outputs this vector where you are detecting three causes, pedestrians, cars and motorcycles, it won't be able to output two detections. So I have to pick one of the two detections to output. With the idea of anchor boxes, what you are going to do, is pre-define two different shapes called, anchor boxes or anchor box shapes. And what you are going to do is now, be able to associate two predictions with the two anchor boxes. And in general, you might use more anchor boxes, maybe five or even more. But for this video, I am just going to use two anchor boxes just to make the description easier. So what you do is you define the cross label to be, instead of this vector on the left, you basically repeat this twice. S, you will have PC, PX, PY, PH, PW, C1, C2, C3, and these are the eight outputs associated with anchor box 1. And then you repeat that PC, PX and so on down to C1, C2, C3, and other eight outputs associated with anchor box 2. So, because the shape of the pedestrian is more similar to the shape of anchor box 1 and anchor box 2, you can use these eight numbers to encode that PC as one, yes there is a pedestrian. Use this to encode the bounding box around the pedestrian, and then use this to encode that that object is a pedestrian. And then because the box around the car is more similar to the shape of anchor box 2 than anchor box 1, you can then use this to encode that the second object here is the car, and have the bounding box and so on be all the parameters associated with the detected car.

So to summarize, previously, before you are using anchor boxes, you did the following, which is for each object in the training set and the training set image, it was assigned to the grid cell that corresponds to that object's midpoint. And so the output Y was 3 by 3 by 8 because you have a 3 by 3 grid. And for each grid position, we had that output vector which is PC, then the bounding box, and C1, C2, C3. With the anchor box, you now do that following. Now, each object is assigned to the same grid cell as before, assigned to the grid cell that contains the object's midpoint, but it is assigned to a grid cell and anchor box with the highest IoU with the object's shape.

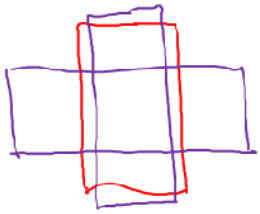
Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y:

$3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box)

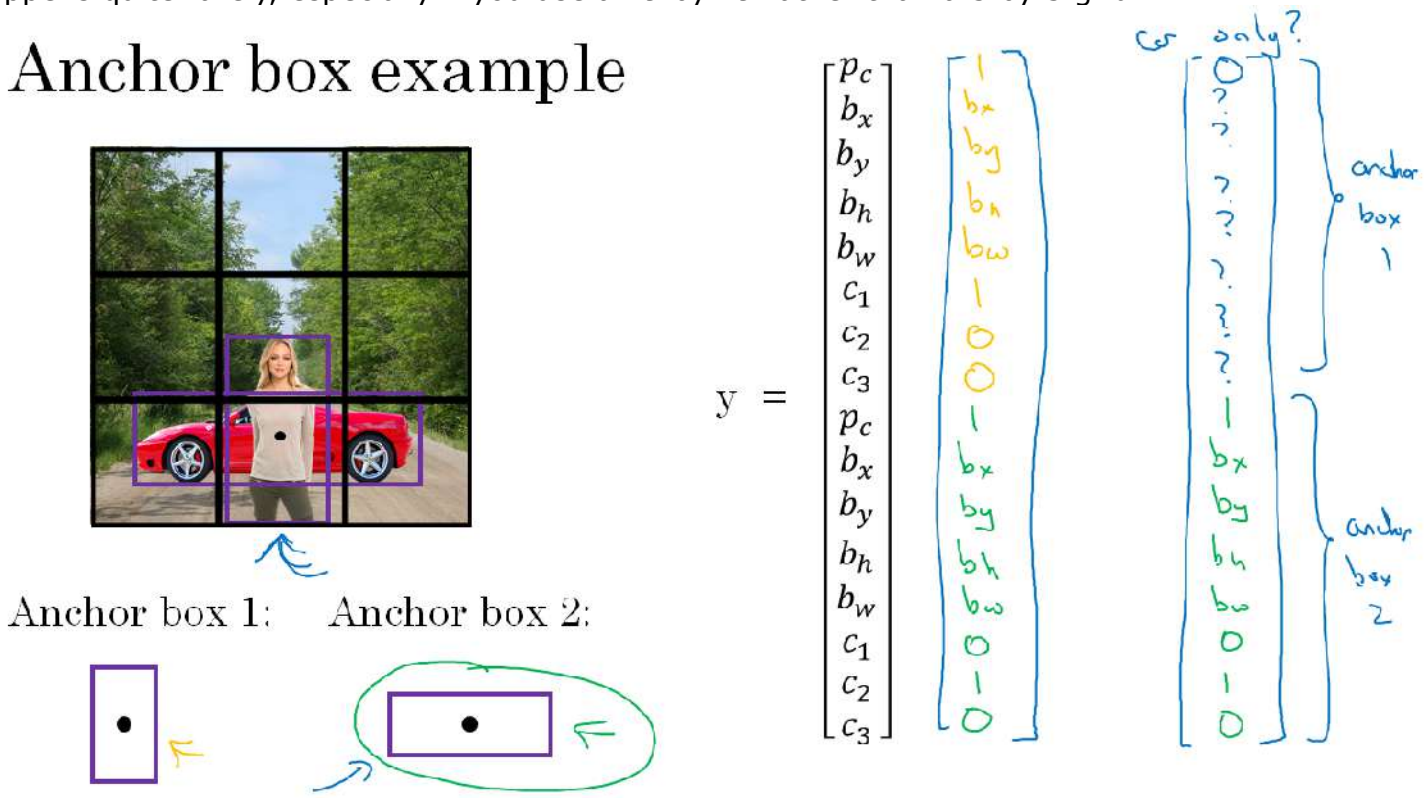
Output y:

$3 \times 3 \times 16$

$3 \times 3 \times 2 \times 8$

So, you have two anchor boxes, you will take an object and see. So if you have an object with this shape, what you do is take your two anchor boxes. Maybe one anchor box is this this shape that's anchor box 1, maybe anchor box 2 is this shape, and then you see which of the two anchor boxes has a higher IoU, will be drawn through bounding box. And whichever it is, that object then gets assigned not just to a grid cell but to a pair. It gets assigned to grid cell comma anchor box pair. And that's how that object gets encoded in the target label. And so now, the output Y is going to be 3 by 3 by 16. Because as you saw on the previous slide, Y is now 16 dimensional. Or if you want, you can also view this as 3 by 3 by 2 by 8 because there are now two anchor boxes and Y is eight dimensional. And dimension of Y being eight was because we have three objects causes if you have more objects than the dimension of Y would be even higher. So let's go through a complete example. For this grid cell, let's specify what is Y. So the pedestrian is more similar to the shape of anchor box 1. So for the pedestrian, we're going to assign it to the top half of this vector. So yes, there is an object, there will be some bounding box associated at the pedestrian. And I guess if a pedestrian is cos one, then we see one as one, and then zero, zero. And then the shape of the car is more similar to anchor box 2. And so the rest of this vector will be one and then the bounding box associated with the car, and then the car is C2, so there's zero, one, zero. And so that's the label Y for that lower middle grid cell that this arrow was pointing to. Now, what if this grid cell only had a car and had no pedestrian? If it only had a car, then assuming that the shape of the bounding box around the car is still more similar to anchor box 2, then the target label Y, if there was just a car there and the pedestrian had gone away, it will still be the same for the anchor box 2 component. Remember that this is a part of the vector corresponding to anchor box 2. And for the part of the vector corresponding to anchor box 1, what you do is you just say there is no object there. So PC is zero, and then the rest of these will be don't cares. Now, just some additional details. What if you have two anchor boxes but three objects in the same grid cell? That's one case that this algorithm doesn't handle well. Hopefully, it won't happen. But if it does, this algorithm doesn't have a great way of handling it. I will just influence some default tiebreaker for that case. Or what if you have two objects associated with the same grid cell, but both of them have the same anchor box shape? Again, that's another case that this algorithm doesn't handle well. If you influence some default way of tiebreaking if that happens, hopefully this won't happen with your data set, it won't happen much at all. And so, it shouldn't affect performance as much. So, that's it for anchor boxes. And even though I'd motivated anchor boxes

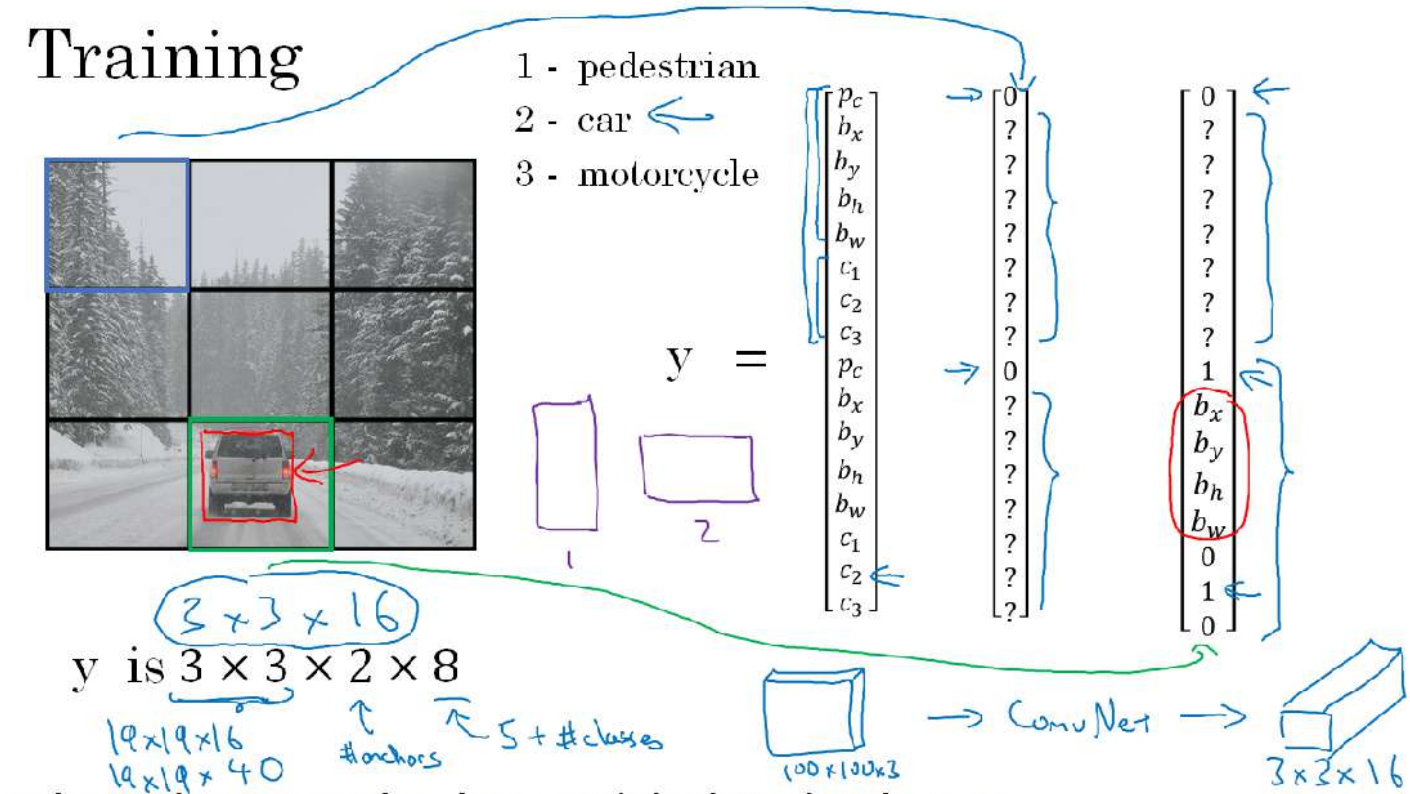
as a way to deal with what happens if two objects appear in the same grid cell, in practice, that happens quite rarely, especially if you use a 19 by 19 rather than a 3 by 3 grid.



The chance of two objects having the same midpoint rather these 361 cells, it does happen, but it doesn't happen that often. Maybe even better motivation or even better results that anchor boxes gives you is it allows your learning algorithm to specialize better. In particular, if your data set has some tall, skinny objects like pedestrians, and some white objects like cars, then this allows your learning algorithm to specialize so that some of the outputs can specialize in detecting white, fat objects like cars, and some of the output units can specialize in detecting tall, skinny objects like pedestrians. So finally, how do you choose the anchor boxes? And people used to just choose them by hand or choose maybe five or 10 anchor box shapes that spans a variety of shapes that seems to cover the types of objects you seem to detect. As a much more advanced version, just in the advance common for those of who have other knowledge in machine learning, and even better way to do this in one of the later YOLO research papers, is to use a K-means algorithm, to group together two types of objects shapes you tend to get. And then to use that to select a set of anchor boxes that this most stereotypically representative of the maybe multiple, of the maybe dozens of object causes you're trying to detect. But that's a more advanced way to automatically choose the anchor boxes. And if you just choose by hand a variety of shapes that reasonably expands the set of object shapes, you expect to detect some tall, skinny ones, some fat, white ones. That should work with these as well. So that's it for anchor boxes. In the next video, let's take everything we've seen and tie it back together into the YOLO algorithm.

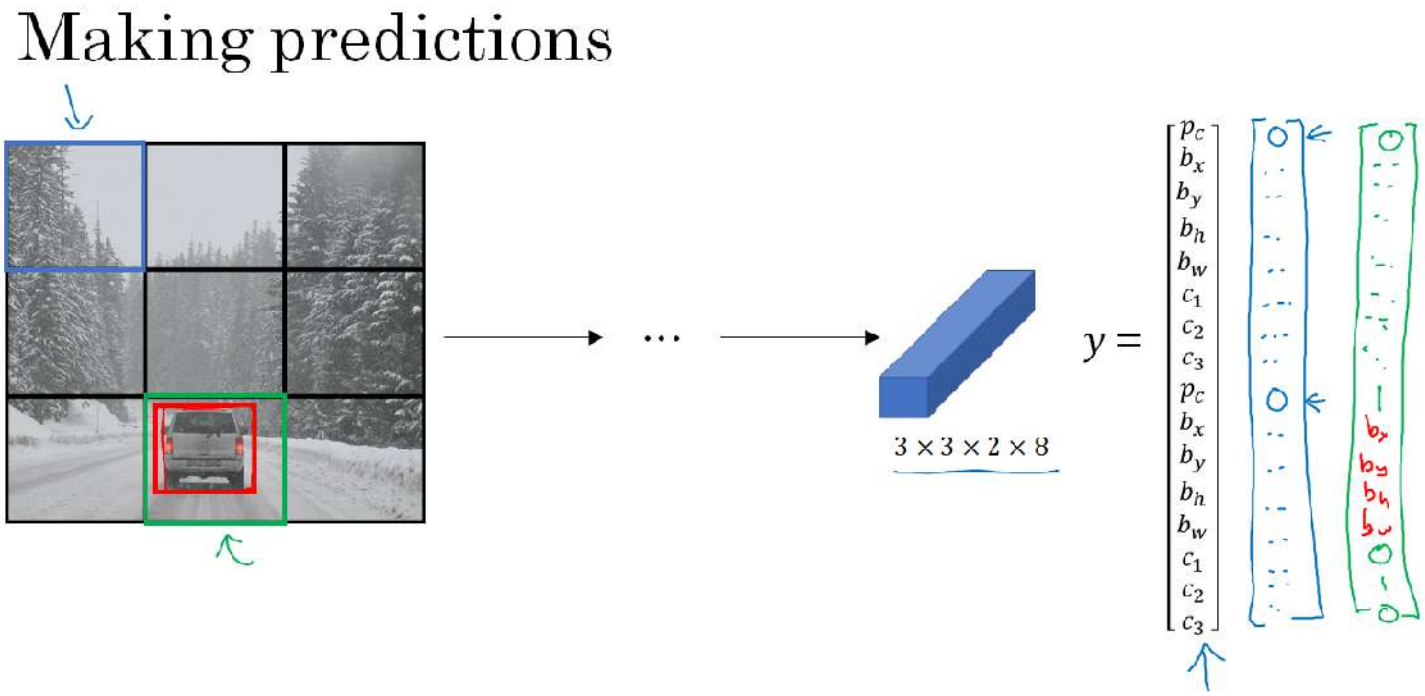
YOLO Algorithm

We've already seen most of the components of object detection. In this section, let's put all the components together to form the **YOLO object detection algorithm**. First, let's see how you construct your training set. Suppose you're trying to train an algorithm to detect three objects: pedestrians, cars, and motorcycles.



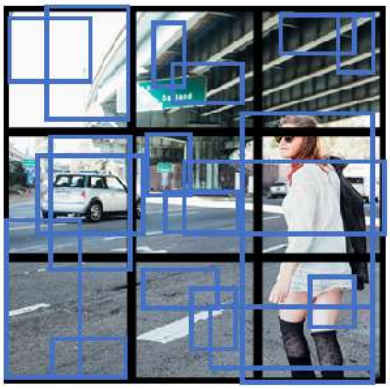
[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]

And you will need to explicitly have the full background class, so just the class labels here. If you're using two anchor boxes, then the outputs y will be three by three because you are using three by three grid cell, by two, this is the number of anchors, by eight because that's the dimension of this. Eight is actually five which is plus the number of classes. So five because you have Pc and then the bounding boxes, that's five, and then C1, C2, C3. That dimension is equal to the number of classes. And you can either view this as three by three by two by eight, or by three by three by sixteen. So to construct the training set, you go through each of these nine grid cells and form the appropriate target vector y. So take this first grid cell, there's nothing worth detecting in that grid cell. None of the three classes pedestrian, car and motocycle, appear in the upper left grid cell and so, the target y corresponding to that grid cell would be equal to this. Where Pc for the first anchor box is zero because there's nothing associated for the first anchor box, and is also zero for the second anchor box and so on all of these other values are don't cares. Now, most of the grid cells have nothing in them, but for that box over there, you would have this target vector y. So assuming that your training set has a bounding box like this for the car, it's just a little bit wider than it is tall. And so if your anchor boxes are that, this is a anchor box one, this is anchor box two, then the red box has just slightly higher IoU with anchor box two. And so the car gets associated with this lower portion of the vector. So notice then that Pc associate anchor box one is zero. So you have don't cares all these components. Then you have this Pc is equal to one, then you should use these to specify the position of the red bounding box, and then specify that the correct object is class two. Right that it is a car. So you go through this and for each of your nine grid positions each of your three by three grid positions, you would come up with a vector like this. Come up with a 16 dimensional vector. And so that's why the final output volume is going to be 3 by 3 by 16. Oh and as usual for simplicity on the slide I've used a 3 by 3 the grid. In practice it might be more like a 19 by 19 by 16. Or in fact if you use more anchor boxes, maybe 19 by 19 by 5 x 8 because five times eight is 40. So it will be 19 by 19 by 40. That's if you use five anchor boxes. So that's training and you train ConvNet that inputs an image, maybe 100 by 100 by 3, and your ConvNet would then finally output this output volume in our example, 3 by 3 by 16 or 3 by 3 by 2 by 8. Next, let's look at how your algorithm can make predictions. Given an image, your neural network will output this by 3 by 3 by 2 by 8 volume, where for each of the nine grid cells you get a vector like that.



So for the grid cell here on the upper left, if there's no object there, hopefully, your neural network will output zero here, and zero here, and it will output some other values. Your neural network can't output a question mark, can't output a don't care. So I'll put some numbers for the rest. But these numbers will basically be ignored because the neural network is telling you that there's no object there. So it doesn't really matter whether the output is a bounding box or there's is a car. So basically just be some set of numbers, more or less noise. In contrast, for this box over here hopefully, the value of y to the output for that box at the bottom left, hopefully would be something like zero for bounding box one. And then just open a bunch of numbers, just noise. Hopefully, you'll also output a set of numbers that corresponds to specifying a pretty accurate bounding box for the car. So that's how the neural network will make predictions. Finally, you run this through non-max suppression. So just to make it interesting. Let's look at the new test set image. Here's how you would run non-max suppression. If you're using two anchor boxes, then for each of the non-grid cells, you get two predicted bounding boxes. Some of them will have very low probability, very low Pc, but you still get two predicted bounding boxes for each of the nine grid cells. So let's say, those are the bounding boxes you get. And notice that some of the bounding boxes can go outside the height and width of the grid cell that they came from. Next, you then get rid of the low probability predictions. So get rid of the ones that even the neural network says, gee this object probably isn't there. So get rid of those. And then finally if you have three classes you're trying to detect, you're trying to detect pedestrians, cars and motorcycles. What you do is, for each of the three classes, independently run non-max suppression for the objects that were predicted to come from that class. But use non-max suppression for the predictions of the pedestrians class, run non-max suppression for the car class, and non-max suppression for the motorcycle class.

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.

But run that basically three times to generate the final predictions. And so the output of this is hopefully that you will have detected all the cars and all the pedestrians in this image. So that's it for the YOLO object detection algorithm. Which is really one of the most effective object detection algorithms, that also encompasses many of the best ideas across the entire computer vision literature that relate to object detection.

Week 4: Special applications: Face recognition & Neural style transfer

Discover how CNNs can be applied to multiple fields, including art generation and face recognition. Implement your own algorithm to generate art and recognize faces!

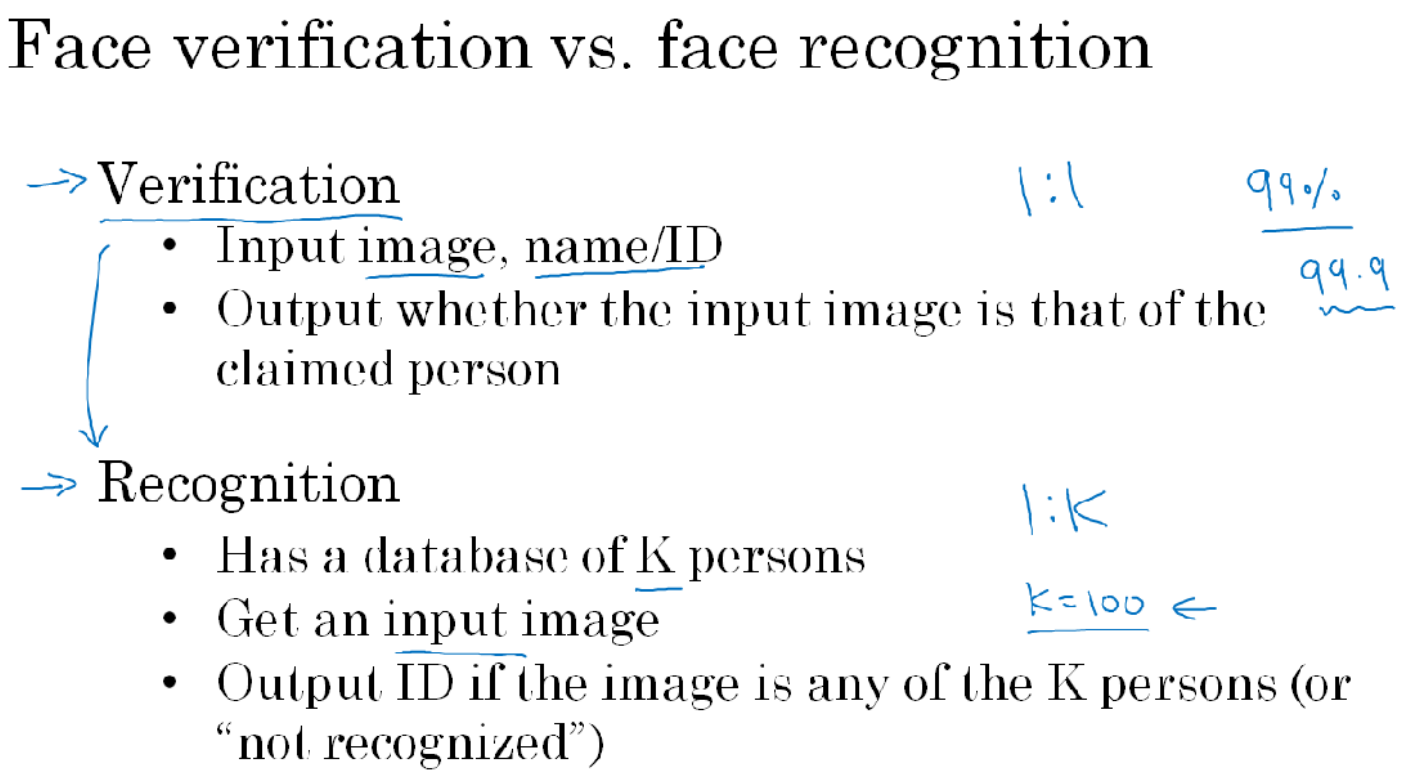
Face Recognition

What is face recognition?

We've already learned a lot about convnet. I this week we'll see a couple of importatnt applications of convnets. We'll start with the **face recognition**, and then go on later this week to neural style transfer. But first, let's start the face recognition and just for fun, I want to show you a demo. When I was leading by those AI group, one of the teams I worked with led by Yuanqing Lin had built a face recognition system that I thought is really cool. Let's take a look. So, I'm going to play this video here, but I can also get whoever is editing this raw video configure out to this better to splice in the raw video or take the one I'm playing here. I want to show you a face recognition demo. I'm in Baidu's headquarters in China. Most companies require that to get inside, you swipe an ID card like this one but here we don't need that. Using face recognition, check what I can do. When I walk up, it recognizes my face, it says, "Welcome Andrew," and I just walk right through without ever having to use my ID card. Let me show you something else. I'm actually here with Lin Yuanqing, the director of IDL which developed all of this face recognition technology. I'm gonna hand him my ID card, which has my face printed on it, and he's going to use it to try to sneak in using my picture instead of a live human. I'm gonna use Andrew's card and try to sneak in and see what happens. So the system is not recognizing it, it refuses to recognize. Okay. Now, I'm going to use my own face. So face recognition technology like this is taking off very rapidly in China and I hope that this type of technology soon makes it way to other countries.. So, pretty cool, right? The video you just saw demoed both face recognition as well as liveness detection. The latter meaning making sure that you are a live human. It turns out liveness detection can be implemented using supervised learning as well to predict live human versus not live human but I want to spend less time on that. Instead, I want to focus our time on talking about how to build the face recognition portion of the system.

First, let's start by going over some of the terminology used in **face recognition**. In the face recognition literature, people often talk about **face verification** and **face recognition**. This is the

face verification problem which is if you're given an input image as well as a name or ID of a person and the job of the system is to verify whether or not the input image is that of the claimed person. So, sometimes this is also called a one to one problem where you just want to know if the person is the person they claim to be. So, the **recognition problem is much harder than the verification problem**.

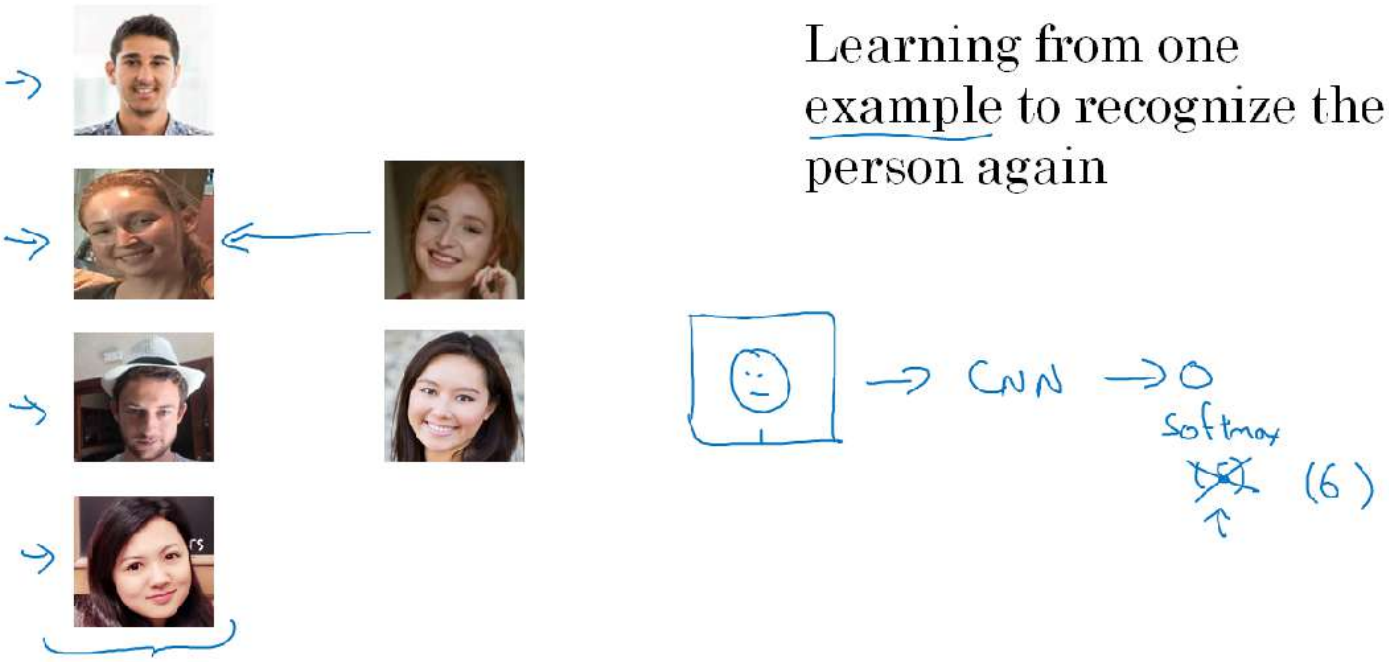


To see why, let's say, you have a verification system that's 99 percent accurate. So, 99 percent might not be too bad but now suppose that K is equal to 100 in a recognition system. If you apply this system to a recognition task with a 100 people in your database, you now have a hundred times of chance of making a mistake and if the chance of making mistakes on each person is just one percent. So, if you have a database of a 100 persons and if you want an acceptable recognition error, you might actually need a verification system with maybe 99.9 or even higher accuracy before you can run it on a database of 100 persons that have a high chance and still have a high chance of getting incorrect. In fact, if you have a database of 100 persons currently just be even quite a bit higher than 99 percent for that to work well. But what we do in the next few chapters is focus on building a face verification system as a building block and then if the accuracy is high enough, then you probably use that in a recognition system as well.

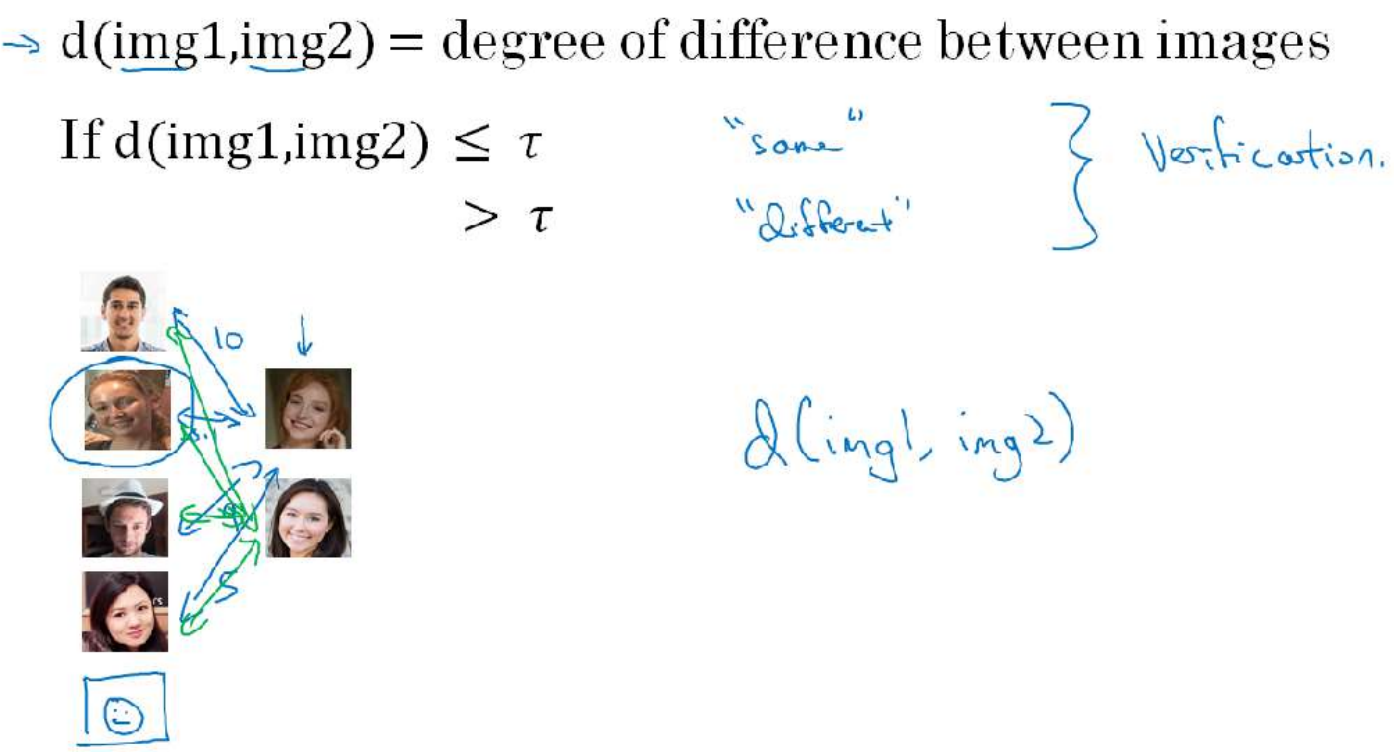
One Shot Learning

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. Let's see an example of what this means and talk about how to address this problem. Let's say you have a database of four pictures of employees in you're organization.

One-shot learning



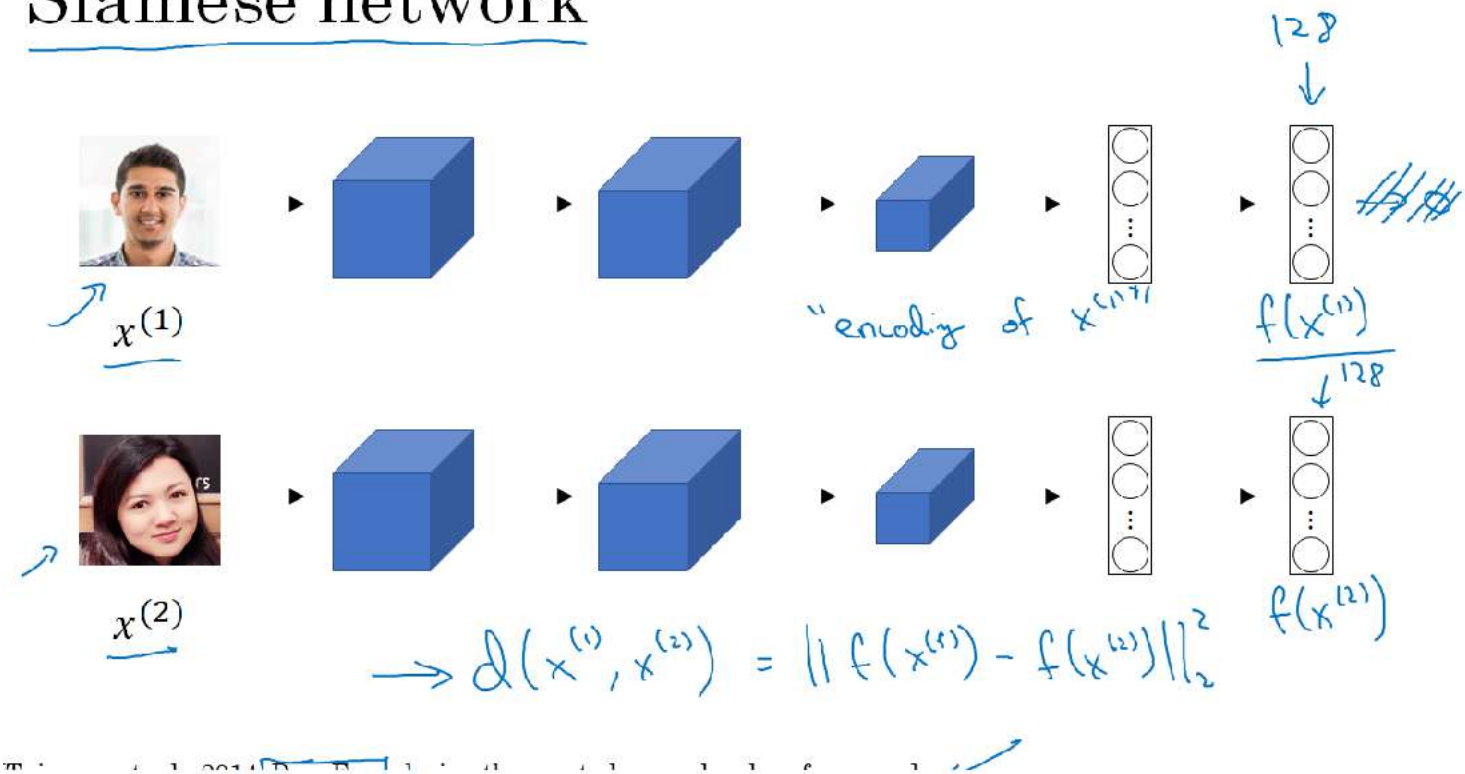
These are actually some of my colleagues at Deeplearning.AI, Khan, Danielle, Younes and Thian. Now let's say someone shows up at the office and they want to be let through the turnstile. What the system has to do is, despite ever having seen only one image of Danielle, to recognize that this is actually the same person and, in contrast, if it sees someone that's not in this database, then it should recognize that this is not any of the four persons in the database. So in the **one shot learning problem, you have to learn from just one example to recognize the person again** and you need this for most face recognition systems because you might have only one picture of each of your employees or of your team members in your employee database. So one approach you could try is to input the image of the person, feed it to a ConvNet and having a output label, y, using a softmax unit with four outputs or maybe five outputs corresponding to each of these four persons or none of the above. So that would be 5 outputs in the softmax. But this really doesn't work well. Because if you have such a small training set it is really not enough to train a robust neural network for this task and also what if a new person joins your team? So now you have 5 persons you need to recognize, so there should now be six outputs. Do you have to retrain the ConvNet every time? That just doesn't seem like a good approach. So to carry out face recognition, to carry out one-shot learning. So instead, to make this work, what you're going to do instead is learn a similarity function. In particular, you want a neural network to learn a function which going to denote d, which inputs two images and outputs the degree of difference between the two images. So if the two images are of the same person, you want this to output a small number and if the two images are of two very different people you want it to output a large number. So during recognition time, if the degree of difference between them is less than some threshold called tau, which is a hyperparameter. Then you would predict that these two pictures are the same person and if it is greater than tau, you would predict that these are different persons and so this is how you address the face verification problem.



To use this for a recognition task, what you do is, given this new picture, you will use this function d to compare these two images and maybe I'll output a very large number, let's say 10, for this example. And then you compare this with the second image in your database. And because these two are the same person, hopefully you output a very small number. You do this for the other images in your database and so on and based on this, you would figure out that this is actually that person, which is Danielle and in contrast, if someone not in your database shows up, as you use the function d to make all of these pairwise comparisons, hopefully d will output have a very large number for all four pairwise comparisons and then you say that this is not any one of the four persons in the database. Notice how this allows you to solve the one-shot learning problem. So long as you can learn this function d, which inputs a pair of images and tells you, basically, if they're the same person or different persons. Then if you have someone new join your team, you can add a fifth person to your database, and it just works fine. So you've seen how learning this function d, which inputs two images, allows you to address the one-shot learning problem. In the next section, let's take a look at how you can actually train the neural network to learn dysfunction d.

Siamese Network

The job of the function d, which you learned about in the last section, is to input two faces and tell you how similar or how different they are. A good way to do this is to use a **Siamese network**. Let's take a look. You're used to seeing pictures of convnet like these where you input an image, let's say x1. And through a sequence of convolutional and pulling and fully connected layers, end up with a feature vector.



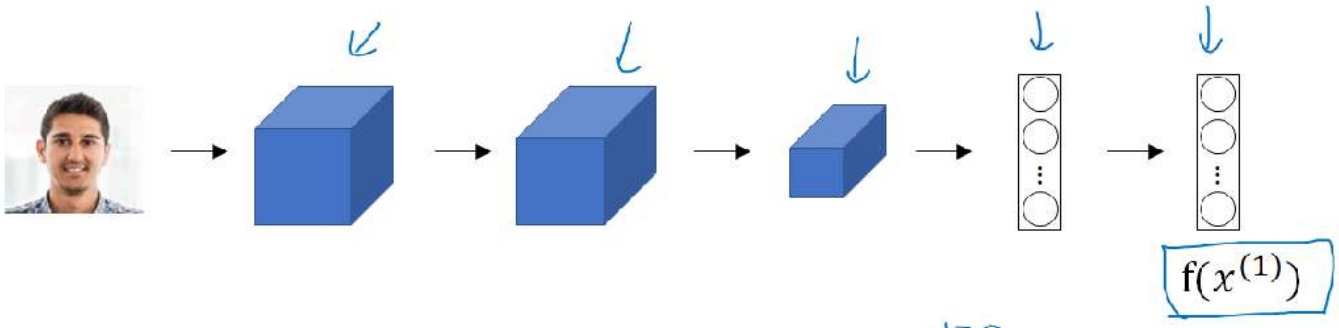
And sometimes this is fed to a softmax unit to make a classification. We're going to focus on this vector of let's say 128 numbers computed by some fully connected layer that is deeper in the network.

And I'm going to give this list of 128 numbers a name. I'm going to call this $f(x^{(1)})$, and you should think of $x^{(1)}$ as an encoding of the input image $x^{(1)}$. So it's taken the input image, here this picture of Kian, and is re-representing it as a vector of 128 numbers. The way you can build a face recognition system is then that if you want to compare two pictures, let's say this first picture with this second picture here. What you can do is feed this second picture to the same neural network with the same parameters and get a different vector of 128 numbers, which encodes this second picture. So we will this second picture.

So we're going to call this encoding of this second picture f of $x^{(2)}$, and here we're using $x^{(1)}$ and $x^{(2)}$ just to denote two input images. They don't necessarily have to be the first and second examples in your training sets. It can be any two pictures. Finally, if you believe that these encodings are a good representation of these two images, what you can do is then define the image d of distance between $x^{(1)}$ and $x^{(2)}$ as the norm of the difference between the encodings of these two images. So this idea of running two identical, convolutional neural networks on two different inputs and then comparing them, sometimes that's called a **Siamese neural network architecture** and a lot of the ideas presented here came from this paper due to Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf in the research system that they developed called **DeepFace**.

So how do you train this Siamese neural network? Remember that these two neural networks have the same parameters. So what you want to do is really train the neural network so that the encoding that it computes results in a function d that tells you when two pictures are of the same person. So more formally, the parameters of the neural network define an encoding $f(x^{(i)})$.

Goal of learning



Parameters of NN define an encoding $f(x^{(i)})$ — 128

Learn parameters so that:

If $x^{(i)}, x^{(j)}$ are the same person, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is small.

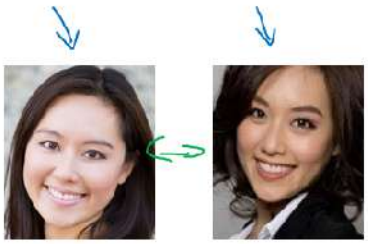
If $x^{(i)}, x^{(j)}$ are different persons, $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is large.

So given any input image $x^{(i)}$, the neural network outputs this 128 dimensional encoding $f(x^{(i)})$ So more formally, what you want to do is learn parameters so that if two pictures, $x^{(i)}$ and $x^{(j)}$, are of the same person, then you want that **distance between their encodings to be small** and in the previous section, we were using $x^{(1)}$ and $x^{(2)}$, but it's really any pair $x^{(i)}$and $x^{(j)}$ from your training set and in contrast, if $x^{(i)}$ and $x^{(j)}$ are of different persons, then you want that **distance between their encodings to be large**. So as you vary the parameters in all of these layers of the neural network, you end up with different encodings and what you can do is use back propagation and vary all those parameters in order to make sure these conditions are satisfied. So in this section we've learned about the Siamese network architecture and have a sense of what you want the neural network to output for you in terms of what would make a good encoding. But how do you actually define an objective function to make a neural network learn to do what we just discussed here? Let's see how you can do that in the next section using the triplet loss function.

Triplet Loss

One way to learn the parameters of the neural network so that it gives you a good encoding for your pictures of faces is to define an applied gradient descent on the triplet loss function. Let's see what that means. To apply the triplet loss, you need to compare pairs of images. For example, given this picture, to learn the parameters of the neural network, you have to look at several pictures at the same time. For example, given this pair of images, you want their encodings to be similar because these are the same person. Whereas, given this pair of images, you want their encodings to be quite different because these are different persons. In the terminology of the **triplet loss**, what you're going do is always look at one anchor image and then you want to distance between the anchor and the positive image, really a positive example, meaning as the same person to be similar. Whereas, you want the anchor when pairs are compared to the negative example for their distances to be much further apart. So, this is what gives rise to the term triplet loss, which is that **you'll always be looking at three images at a time**. **You'll be looking at an anchor image, a positive image, as well as a negative image**.
Check below diagrams to show an example of face recognition using triplet loss.

Learning Objective




Anchor (A) Positive (P) Anchor (A) Negative (N)

$d(A,P) \leq 0.5$ $d(A,N) = 0.7$

Want: $\underbrace{\|f(A)-f(P)\|^2}_{d(A,P)} + \alpha \leq \underbrace{\|f(A)-f(N)\|^2}_{d(A,N)}$ ($\alpha \nearrow 0.2$)

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

0 0 margin

$f(\text{img}) = \vec{0}$

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

Loss function

Given 3 images A, P, N:

$$\mathcal{L}(A,P,N) = \max\left(\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha, 0\right)$$

≥ 0 > 0

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

A, P

Training set: 10k pictures of 1k persons

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

Choosing the triplets A,P,N

During training, if A,P,N are chosen randomly, $d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.

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

Choose triplets that're "hard" to train on.

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

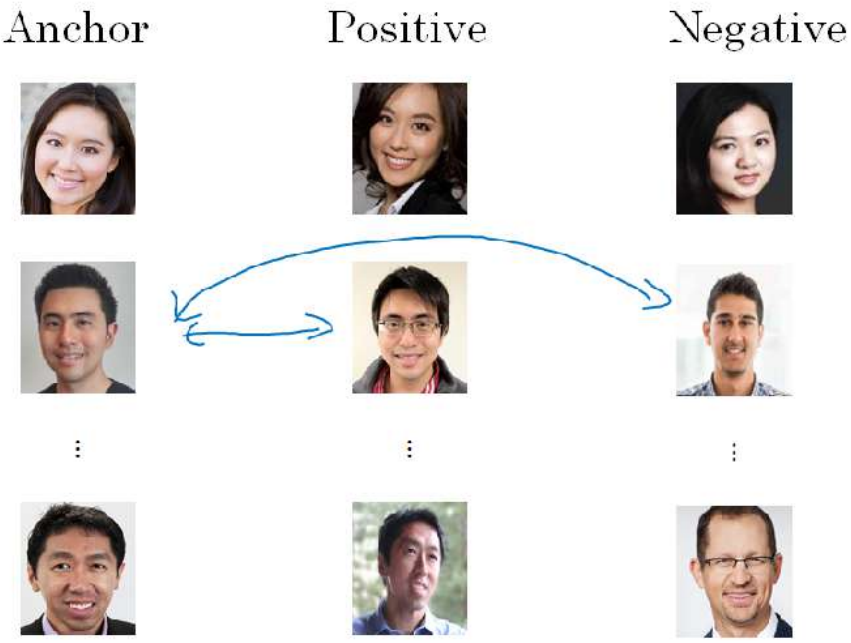
$d(A,P) \approx d(A,N)$

Face Net

Deep Face

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

Training set using triplet loss



J

$d(x^{(i)}, x^{(j)})$

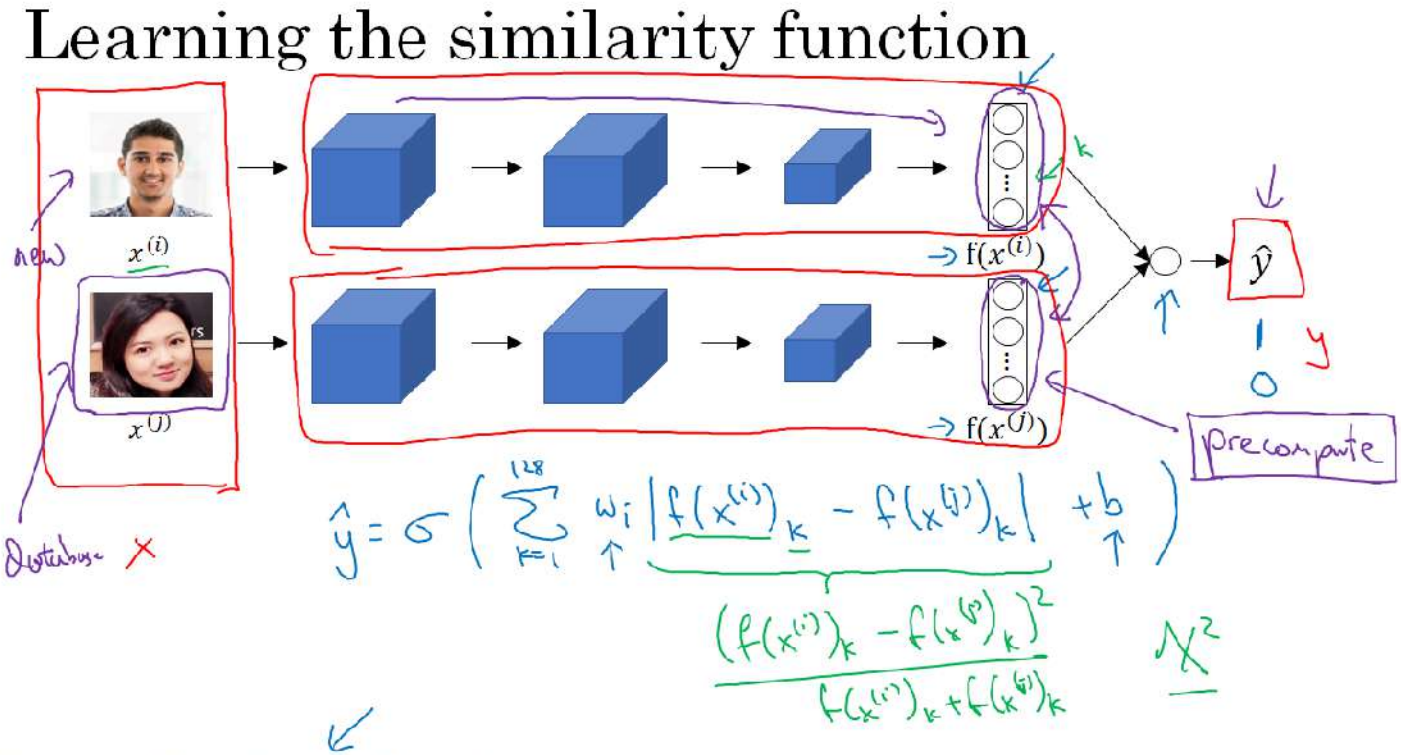
All the details are presented in this paper by Florian Schroff, Dmitry Kalinichenko, and James Philbin, where they have a system called **FaceNet**.

So, just to wrap up, **to train on triplet loss, you need to take your training set and map it to a lot of triples**. So, here (check last diagram) is our triple with an anchor and a positive, both for the same person and the negative of a different person. Here's another one where the

anchor and positive are of the same person but the anchor and negative are of different persons and so on and what you do having defined this training sets of anchor positive and negative triples is **use gradient descent to try to minimize the cost function J we defined on an earlier section and that will have the effect of that propagating to all of the parameters of the neural network in order to learn an encoding so that $d(x^{(i)},x^{(j)})$ of two images will be small when these two images are of the same person, and they'll be large when these are two images of different persons**. So, that's it for the triplet loss and how you can train a neural network for learning and an encoding for face recognition. Now, it turns out that commercial face recognition systems are trained on fairly large datasets at this point. Often, million images and there are some commercial companies talking about using over 100 million images. So these are very large datasets even by modern standards, these dataset assets are not easy to acquire. Fortunately, some of these companies have trained these large networks and posted parameters online. So, rather than trying to train one of these networks from scratch, this is one domain where because of the share data volume sizes, this is one domain where often it might be useful for you to download someone else's pre-train model, rather than do everything from scratch yourself. But even if you do download someone else's pre-train model, I think it's still useful to know how these algorithms were trained or in case you need to apply these ideas from scratch yourself for some application. So that's it for the triplet loss. In the next section, we'll see some other variations on siamese networks and how to train these systems.

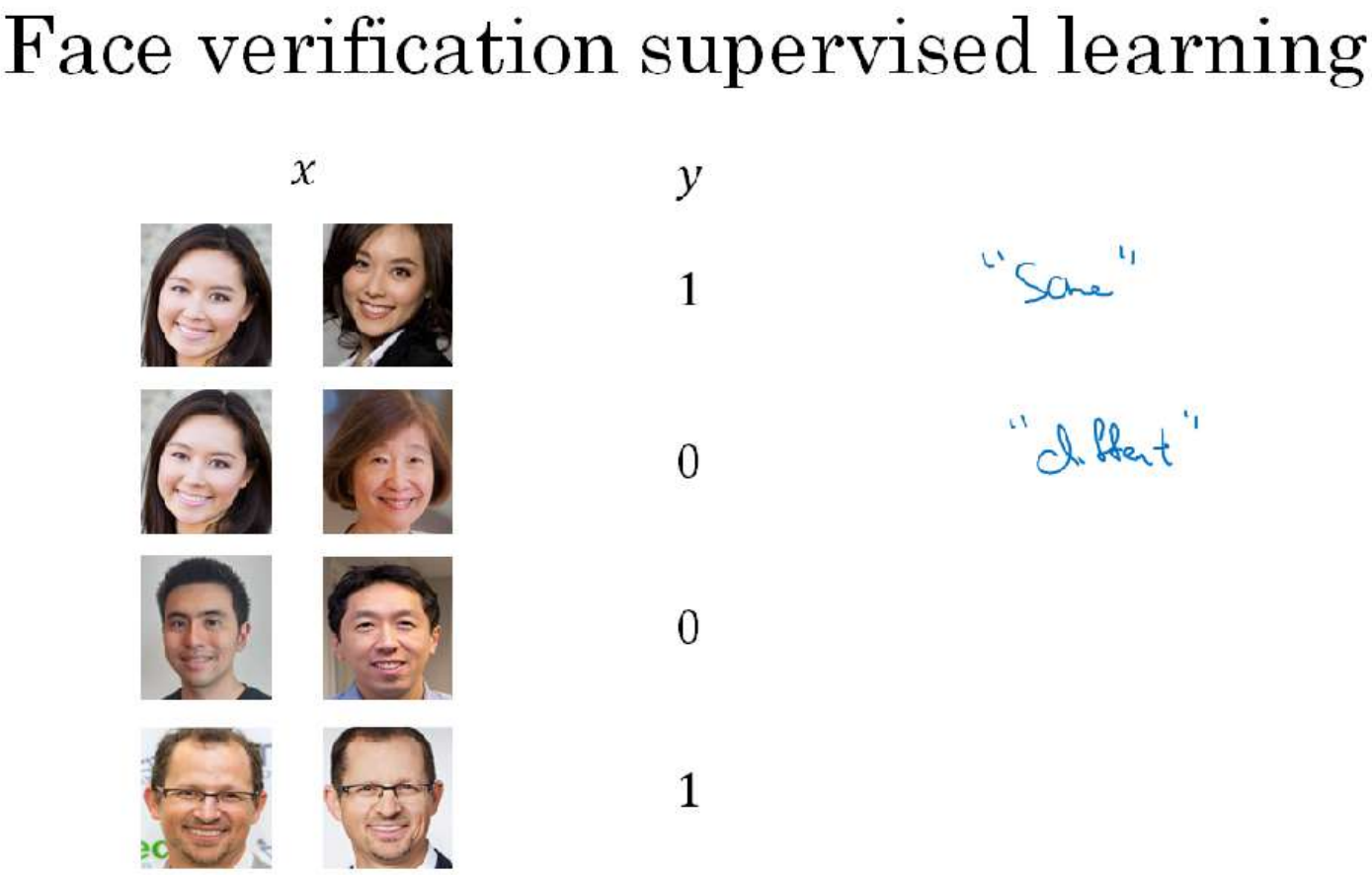
Face Verification and Binary Classification

The Triplet Loss is one good way to learn the parameters of a continent for face recognition. There's another way to learn these parameters. Let me show you how face recognition can also be posed as a straight **binary classification problem**. Another way to train a neural network, is to take this pair of neural networks to take this Siamese Network and have them both compute these embeddings, maybe 128 dimensional embeddings, maybe even higher dimensional, and then have these be input to a **logistic regression** unit to then just make a prediction. Where the target output will be one if both of these are the same persons, and zero if both of these are of different persons. So, this is a way to treat face recognition just as a binary classification problem and this is an alternative to the triplet loss for training a system like this. Now, what does this final logistic regression unit actually do?



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

The output yhat will be a sigmoid function, applied to some set of features but rather than just feeding in, these encodings, what you can do is take the differences between the encodings.



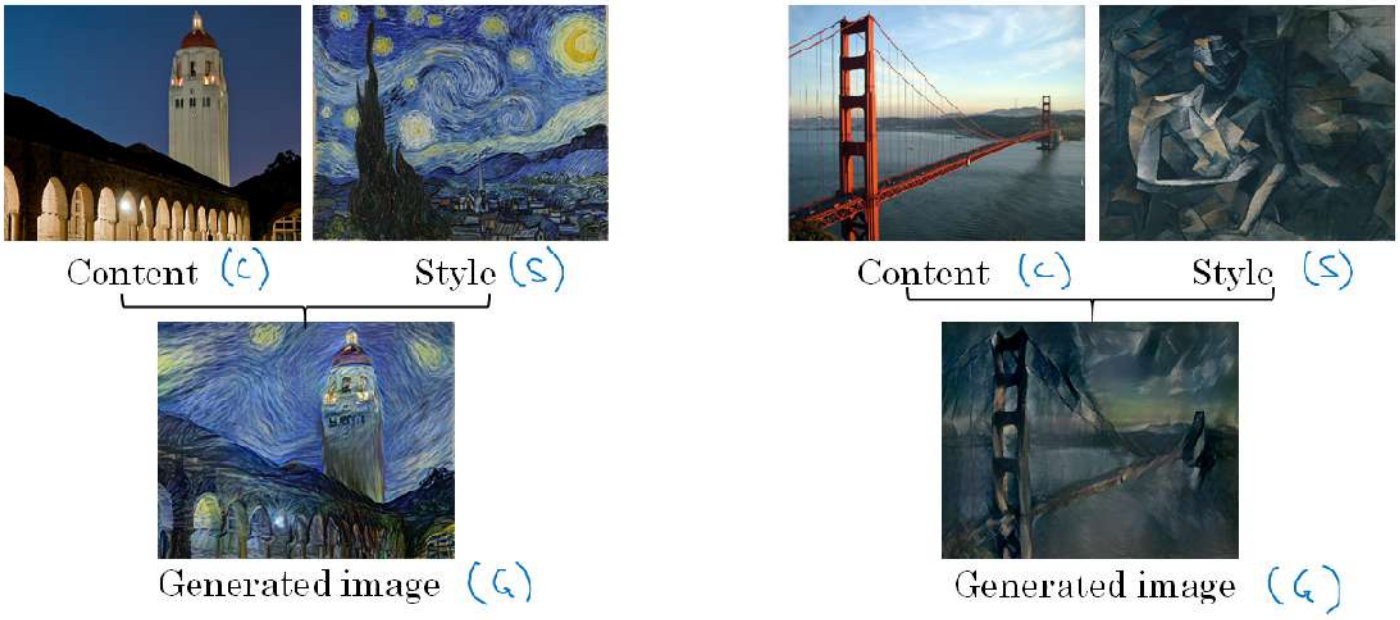
[Taigman et. al., 2014. DeepFace closing the gap to human level performance]

So just to wrap up, to **treat face verification supervised learning, you create a training set of just pairs of images now is of triplets of pairs of images where the target label is one. When these are a pair of pictures of the same person and where the tag label is zero, when these are pictures of different persons and you use different pairs to train the neural network to train the scientists that were using back propagation**. So, this version that you just saw of treating face verification and by extension face recognition as a binary classification problem, this works quite well as well.

Neural Style Transfer

What is neural style transfer?

One of the most fun and exciting applications of ConvNet recently has been **Neural Style Transfer**. What is Neural Style Transfer? Let me take an examples. Let's say you take this image (image 1 in diagram below)



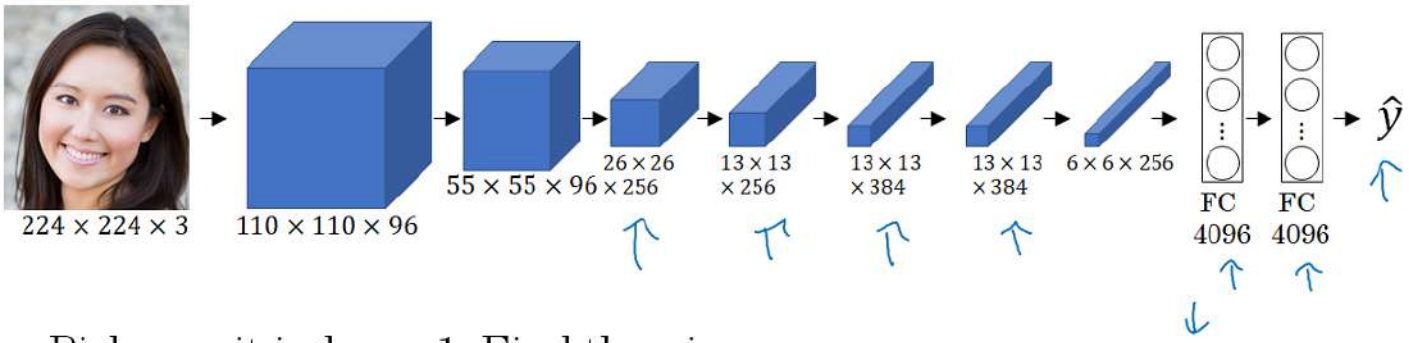
[Images generated by Justin Johnson]

this is actually taken from the Stanford University not far from my Stanford office and you want this picture recreated in the style of this image on the right. This is actually Van Gogh's, Starry Night painting. What Neural Style Transfer allows you to do is generated new image like the one below which is a picture of the Stanford University Campus that painted but drawn in the style of the image on the right. In order to describe how you can implement this yourself, I'm going to use C to denote the content image, S to denote the style image, and G to denote the image you will generate. Here's another example, let's say you have this content image so let's see this is of the Golden Gate Bridge in San Francisco and you have this style image, this is actually Pablo Picasso image. You can then combine these to generate this image G which is the Golden Gate painted in the style of that Picasso shown on the right. What we'll learn in the next few sections is how you can generate these images yourself. In order to implement Neural Style Transfer, you need to look at the features extracted by ConvNet at various layers, the shallow and the deeper layers of a ConvNet.

What are deep ConvNets learning

What are deep ConvNets really learning? In this section, we'll see some visualizations that will help you hold your intuition about what the deeper layers of a ConvNet really are doing and this will help us think through how you can implement neural style transfer as well. Let's start with an example. Lets say you've trained a ConvNet, this is an alex net like network, and you want to visualize what the hidden units in different layers are computing. Here's what you can do.

Visualizing what a deep network is learning



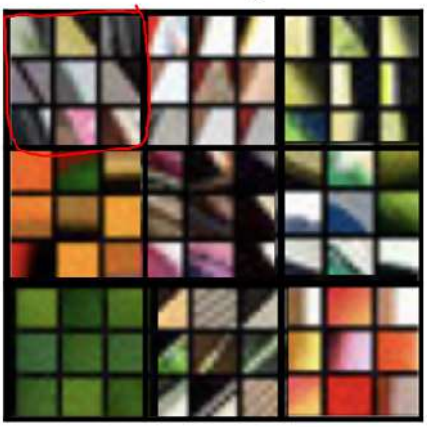
Pick a unit in layer 1. Find the nine image patches that maximize the unit's activation.

Repeat for other units.

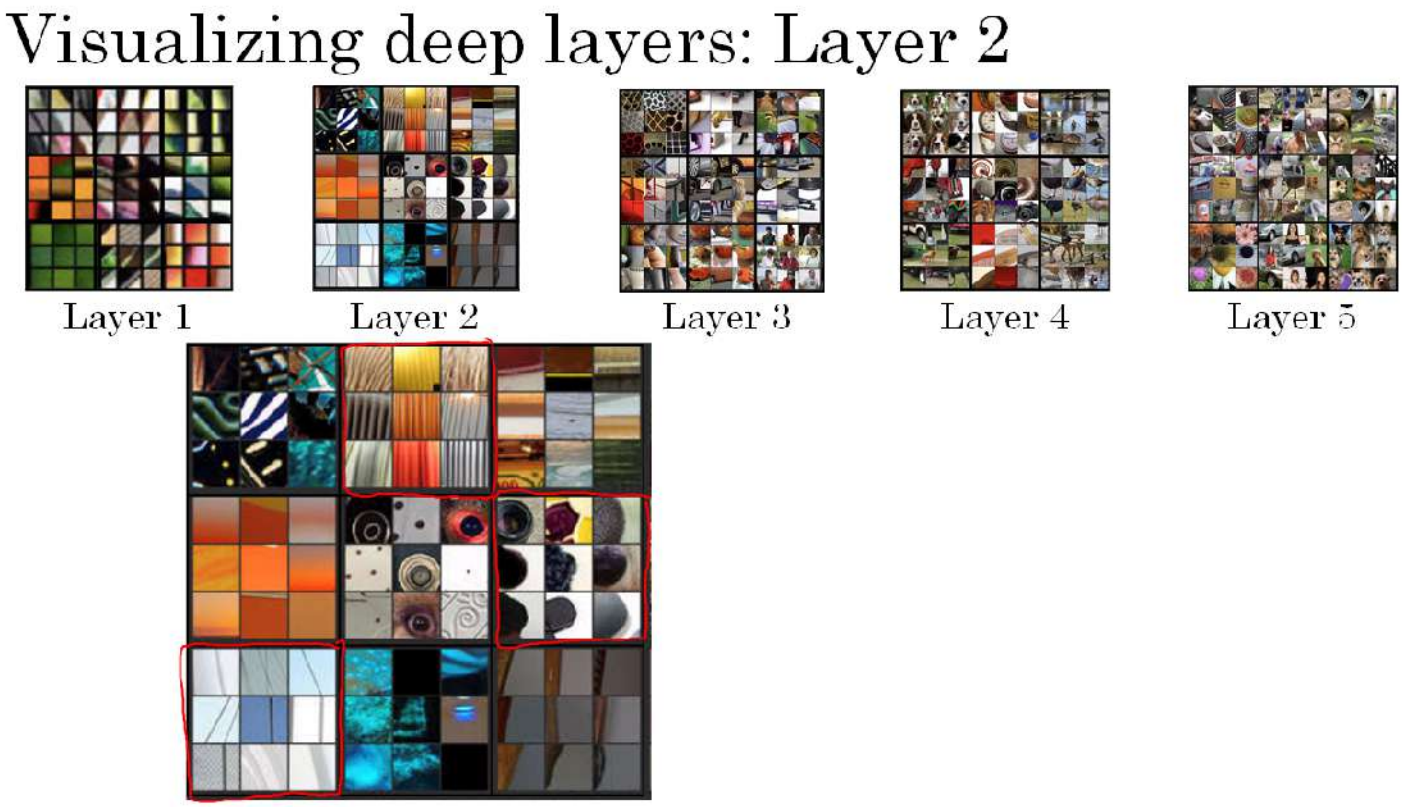
[Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks]

Let's start with a hidden unit in layer 1 and suppose you scan through your training sets and find out what are the images or what are the image patches that maximize that unit's activation. So in other words pause your training set through your neural network, and figure out what is the image that maximizes that particular unit's activation. Now, notice that a hidden unit in layer 1, will see only a relatively small portion of the neural network and so if you visualize, if you plot what activated unit's activation, it makes makes sense to plot just a small image patches, because all of the image that that particular unit sees. So if you pick one hidden unit and find the nine input images that maximizes that unit's activation, you might find nine image patches like shown in diagram below.

Visualizing deep layers: Layer 1

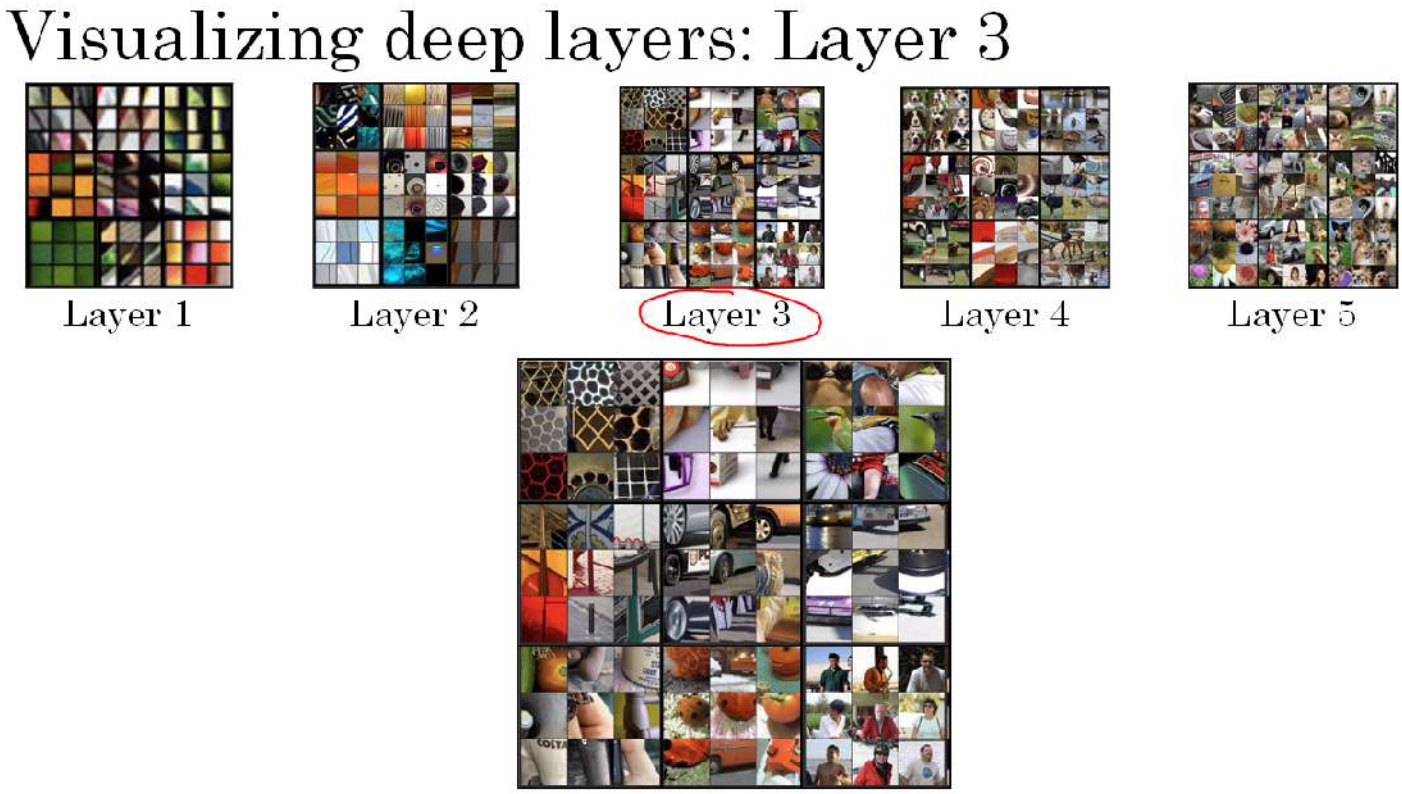


So looks like that in the lower region of an image that this particular hidden unit sees, it's looking for an egde or a line that looks like that. So those are the nine image patches that maximally activate one hidden unit's activation. Now, you can then pick a different hidden unit in layer 1 and do the same thing. So that's a different hidden unit, and looks like this second one, represented by these 9 image patches. Looks like this hidden unit is looking for a line sort of in that portion of its input region, we'll also call this receptive field. And if you do this for other hidden units, you'll find other hidden units, tend to activate in image patches that look like that. This one seems to have a preference for a vertical light edge, but with a preference that the left side of it be green. This one really prefers orange colors, and this is an interesting image patch. This red and green together will make a brownish or a brownish-orangish color, but the neuron is still happy to activate with that, and so on. So this is nine different representative neurons and for each of them the nine image patches that they maximally activate on. So this gives you a sense that, units, train hidden units in layer 1, they're often looking for **relatively simple features such as edge or a particular shade of color** and all of the examples we're using in this section come from this paper by Mathew Zeiler and Rob Fergus, titled **visualizing and understanding convolutional networks**. and we're just going to use one of the simpler ways to visualize what a hidden unit in a neural network is computing. If you read their paper, they have some other more sophisticated ways of visualizing when the ConvNet is running as well.
But now you have repeated this procedure several times for nine hidden units in layer 1. What if you do this for some of the hidden units in the deeper layers of the neuron network. And what does the neural network then learning at a deeper layers. So in the deeper layers, a hidden unit will see a larger region of the image. Where at the extreme end each pixel could hypothetically affect the output of these later layers of the neural network. So l**ater units are actually seen larger image patches**, I'm still going to plot the image patches as the same size on these slides. But if we repeat this procedure, this is what you had previously for layer 1, and this is a visualization of what maximally activates nine different hidden units in layer 2.

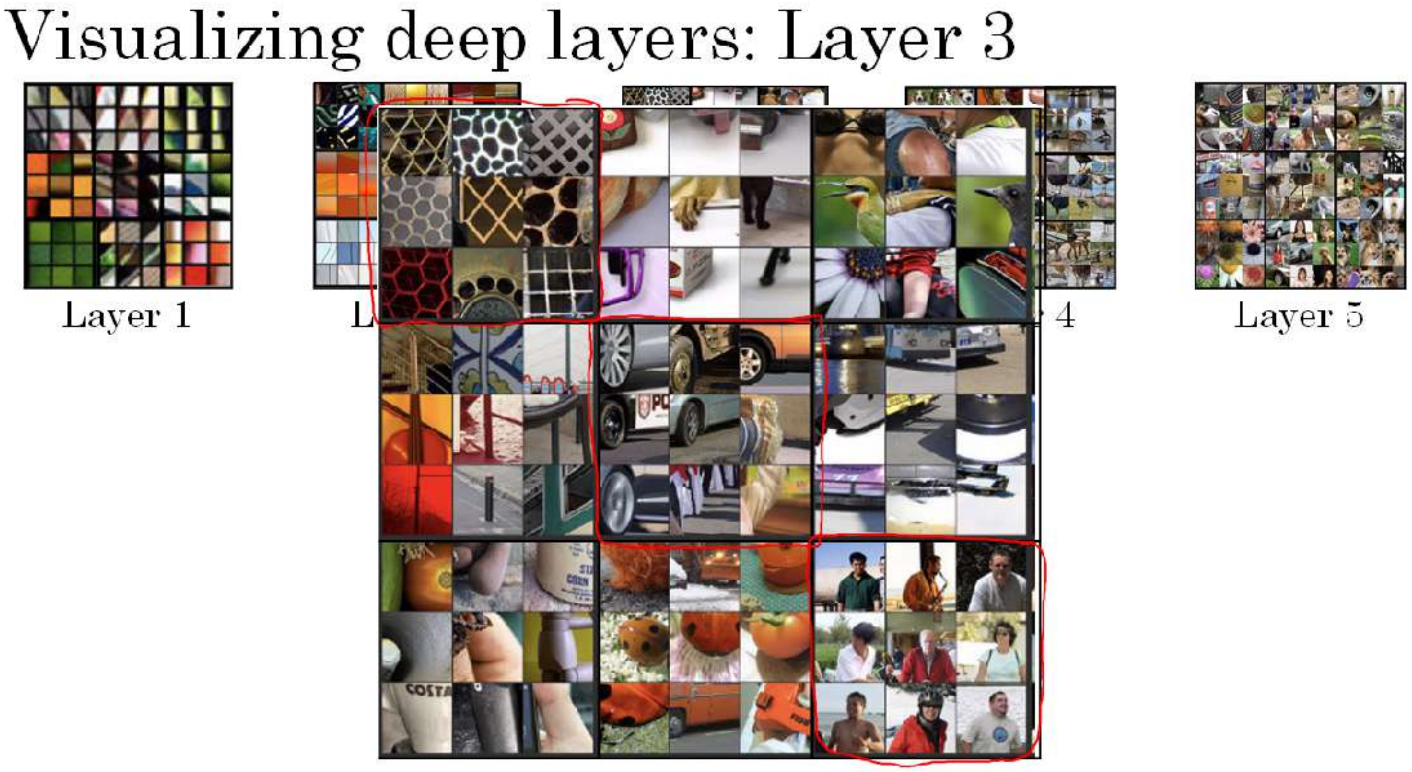


So I want to be clear about what this visualization is. These are the nine patches that cause one hidden unit to be highly activated. And then each grouping, this is a different set of nine image patches that cause one hidden unit to be activated. So this visualization shows nine hidden units in layer 2, and for each of them shows nine image patches that causes that hidden unit to have a very large output, a very large activation. And you can repeat these for deeper layers as well.

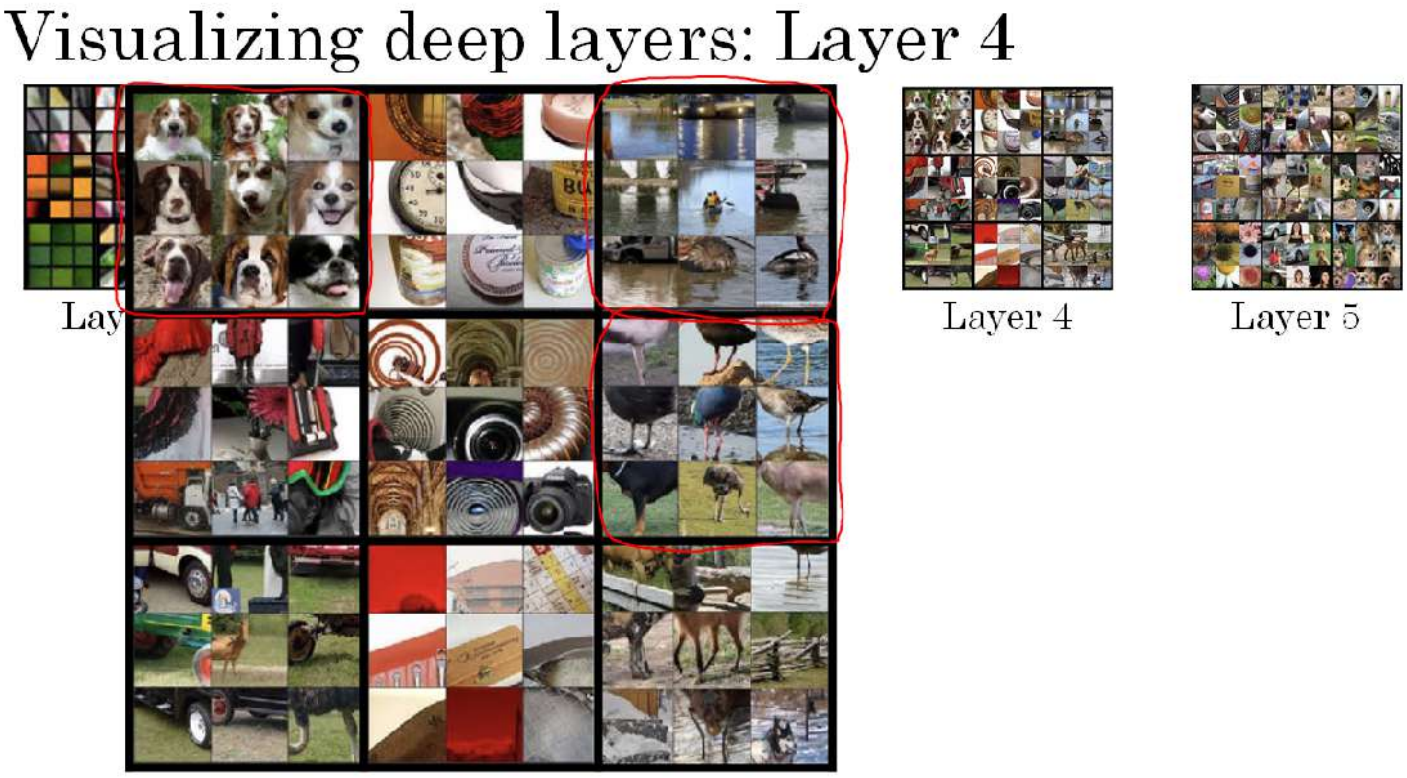
Now, on this section, I know it's kind of hard to see these tiny little image patches, so let me zoom in for some of them. For layer 1, this is what you saw. So for example, this is that first unit we saw which was highly activated, if in the region of the input image, you can see there's an edge maybe at that angle. Now let's zoom in for layer 2 as well, to that visualization. So this is interesting, layer 2 looks it's detecting more complex shapes and patterns. So for example, this hidden unit looks like it's looking for a vertical texture with lots of vertical lines. This hidden unit looks like its highly activated when there's a rounder shape to the left part of the image. Here's one that is looking for very thin vertical lines and so on. And so the features the second layer is detecting are getting more complicated. How about layer 3?



Let's zoom into that, in fact let me zoom in even bigger, so you can see this better, these are the things that maximally activate layer 3. But let's zoom in even bigger, and so this is pretty interesting again. It looks like there is a hidden unit that seems to respond highly to a rounder shape in the lower left hand portion of the image, maybe. So that ends up detecting a lot of cars, dogs and wonders is even starting to detect people.

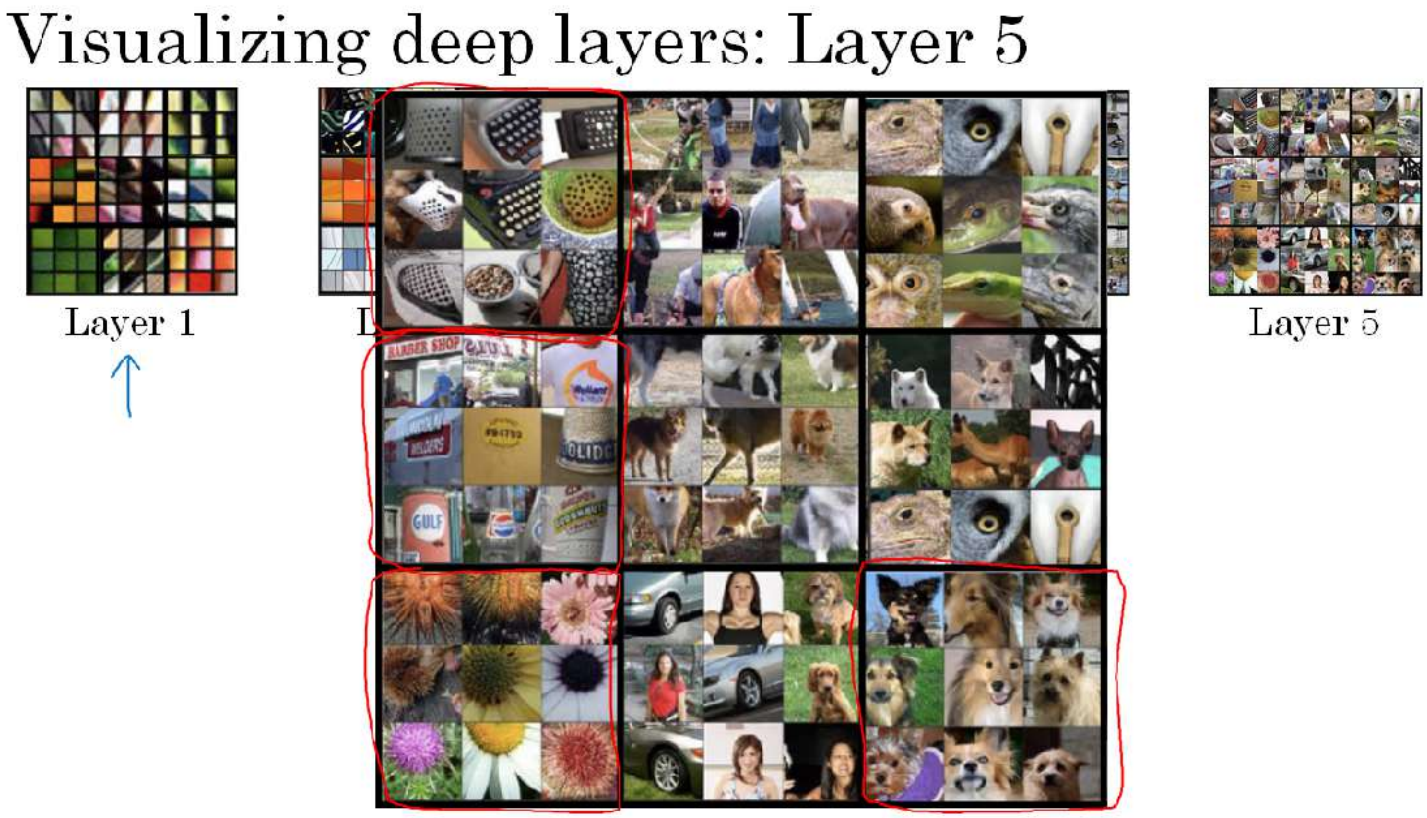


And this one look like it is detecting certain textures like honeycomb shapes, or square shapes, this irregular texture. And some of these it's difficult to look at and manually figure out what is it detecting, but it is clearly starting to detect more complex patterns. How about the next layer? Well, here is layer 4,



and you'll see that the features or the patterns is detecting or even more complex. It looks like this has learned almost a dog detector, but all these dogs likewise similar, right? Is this, I don't know what dog species or dog breed this is. But now all those are dogs, but they look relatively similar as dogs go. Looks like this hidden unit and therefore it is detecting water. This looks like it

is actually detecting the legs of a bird and so on. And then layer 5 is detecting even more sophisticated things.



Cost function

To build a Neural Style Transfer system, let's define a cost function for the generated image. What you see later is that by minimizing this cost function, you can generate the image that you want. Remember what the problem formulation is. You're given a content image C, given a style image S and you goal is to generate a new image G. In order to implement neural style transfer, what you're going to do is define a cost function J of G that measures how good is a particular generated image and we'll use gradient to descent to minimize J of G in order to generate this image. How good is a particular image?

Neural style transfer cost function

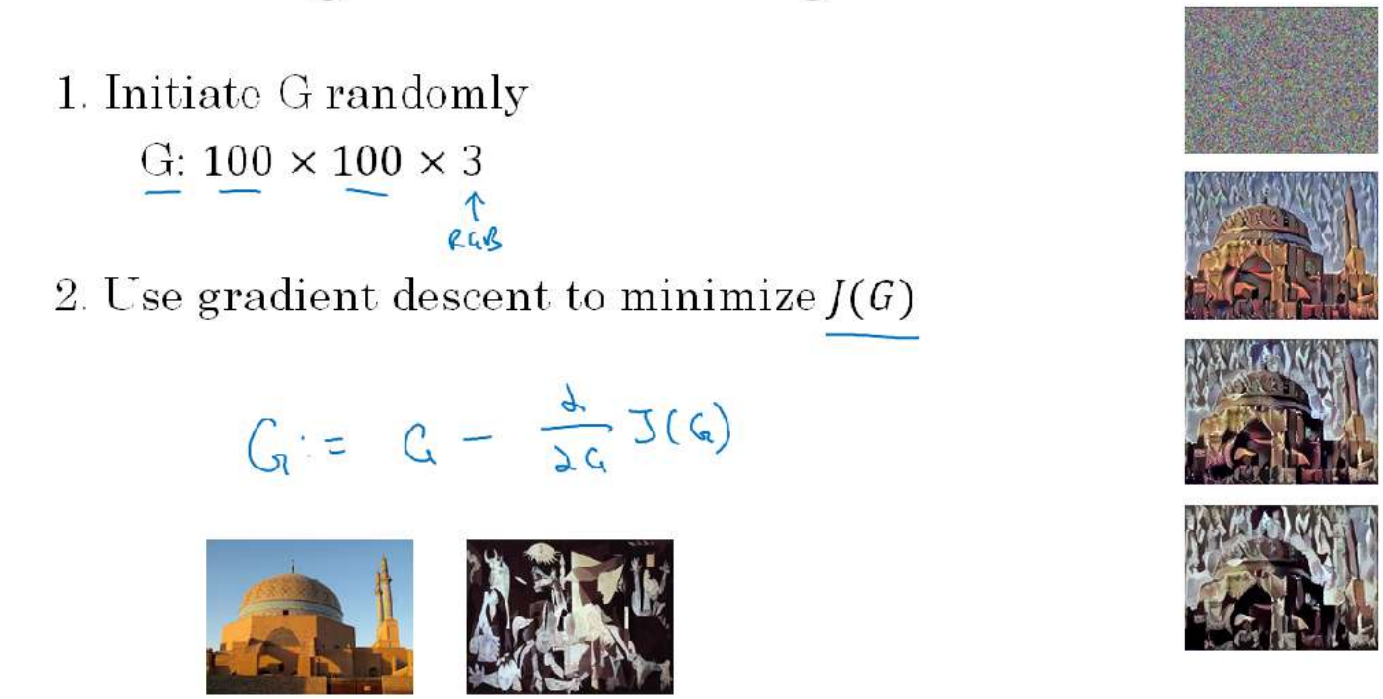
Content C Style S

Generated image G

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

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

Well, we're going to define two parts to this cost function. The first part is called the content cost. This is a function of the content image and of the generated image and what it does is it measures how similar is the contents of the generated image to the content of the content image C. And then going to add that to a style cost function which is now a function of S,G and what this does is it measures how similar is the style of the image G to the style of the image S. Finally, we'll weight these with two hyper parameters alpha and beta to specify the relative weighting between the content costs and the style cost. It seems redundant to use two different hyper parameters to specify the relative cost of the weighting. One hyper parameter seems like it would be enough but the original authors of the Neural Style Transfer Algorithm, use two different hyper parameters. I'm just going to follow their convention here. The Neural Style Transfer Algorithm is going to present in the next few sections is due to Leon Gatys, Alexander Ecker and Matthias. Their papers is not too hard to read so after watching these few sections if you wish, I certainly encourage you to take a look at their paper as well if you want. The way the algorithm would run is as follows, having to find the cost function J(G) in order to actually generate a new image what you do is the following. You would initialize the generated image G randomly so it might be 100 by 100 by 3 or 500 by 500 by 3 or whatever dimension you want it to be. Then we'll define the cost function J(G) on the previous section. What you can do is use gradient descent to minimize this so you can update G as G minus the derivative respect to the cost function of J(G). In this process, you're actually updating the pixel values of this image G which is a 100 by 100 by 3 maybe RGB channel image. Here's an example, let's say you start with this content image and this style image.



This is a another probably Picasso image. Then when you initialize G randomly, you're initial randomly generated image is just this white noise image with each pixel value chosen at random. As you run gradient descent, you minimize the cost function J(G) slowly through the pixel value so then you get slowly an image that looks more and more like your content image rendered in the style of your style image. In this section, we saw the overall outline of the Neural Style Transfer Algorithm where you define a cost function for the generated image G and minimize it.

Content Cost Function

The cost function of the neural style transfer algorithm had a content cost component and a style cost component. Let's start by defining the content cost component. Remember that this is the overall cost function of the neural style transfer algorithm. So, let's figure out what should the content cost function be. Let's say that you use hidden layer l to compute the content cost. If l is a very small number, if you use hidden layer one, then it will really force your generated image to pixel values very similar to your content image. Whereas, if you use a very deep layer, then it's just asking, "Well, if there is a dog in your content image, then make sure there is a dog somewhere in your generated image. " So in practice, layer l chosen somewhere in between. It's neither too shallow nor too deep in the neural network. Check below diagram for summary.

Content cost function

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

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

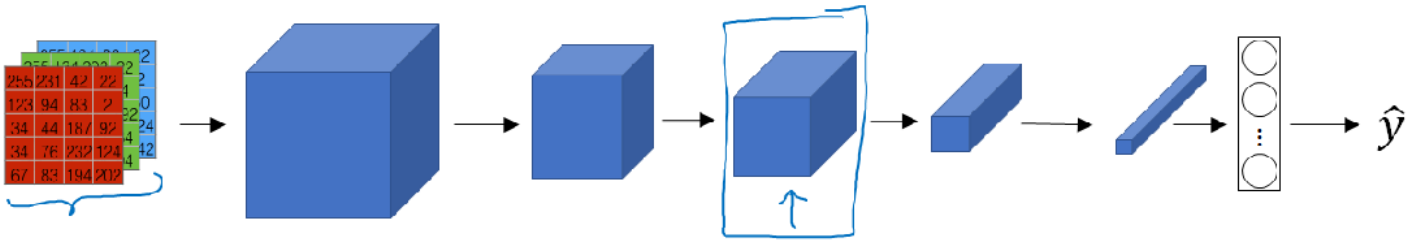
$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

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

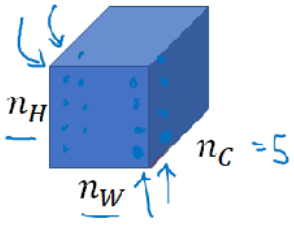
Style Cost Function

In the last section, we saw how to define the content cost function for the neural style transfer. Next, let's take a look at the **style cost function**. So, what is the style of an image mean? Please check the diagram below to understand the style cost function:

Meaning of the "style" of an image



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

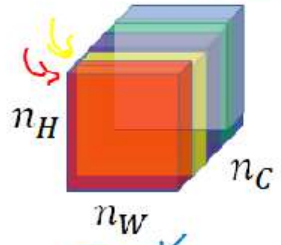


How correlated are the activations across different channels?

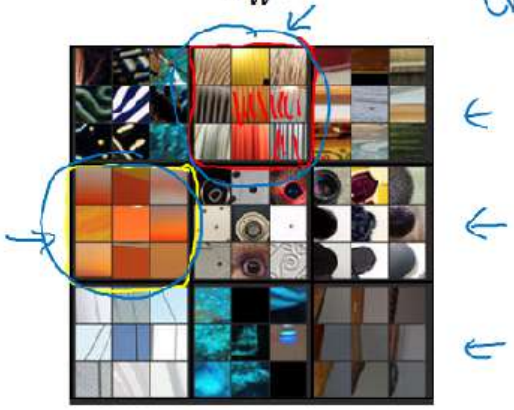
[Gatys et al., 2015. A neural algorithm of artistic style]

Intuition about style of an image

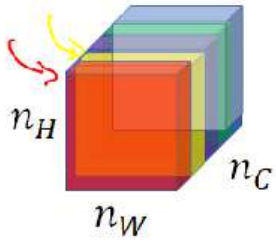
Style image



Correlated?
Uncorrelated



Generated Image



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

Style matrix

H W C

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

n_c

$G^{[l]}_{kk'}$

$k = 1, \ldots, n_c^{[l]}$

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

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

"Gram matrix"

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

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

β

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

Style cost function

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

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

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

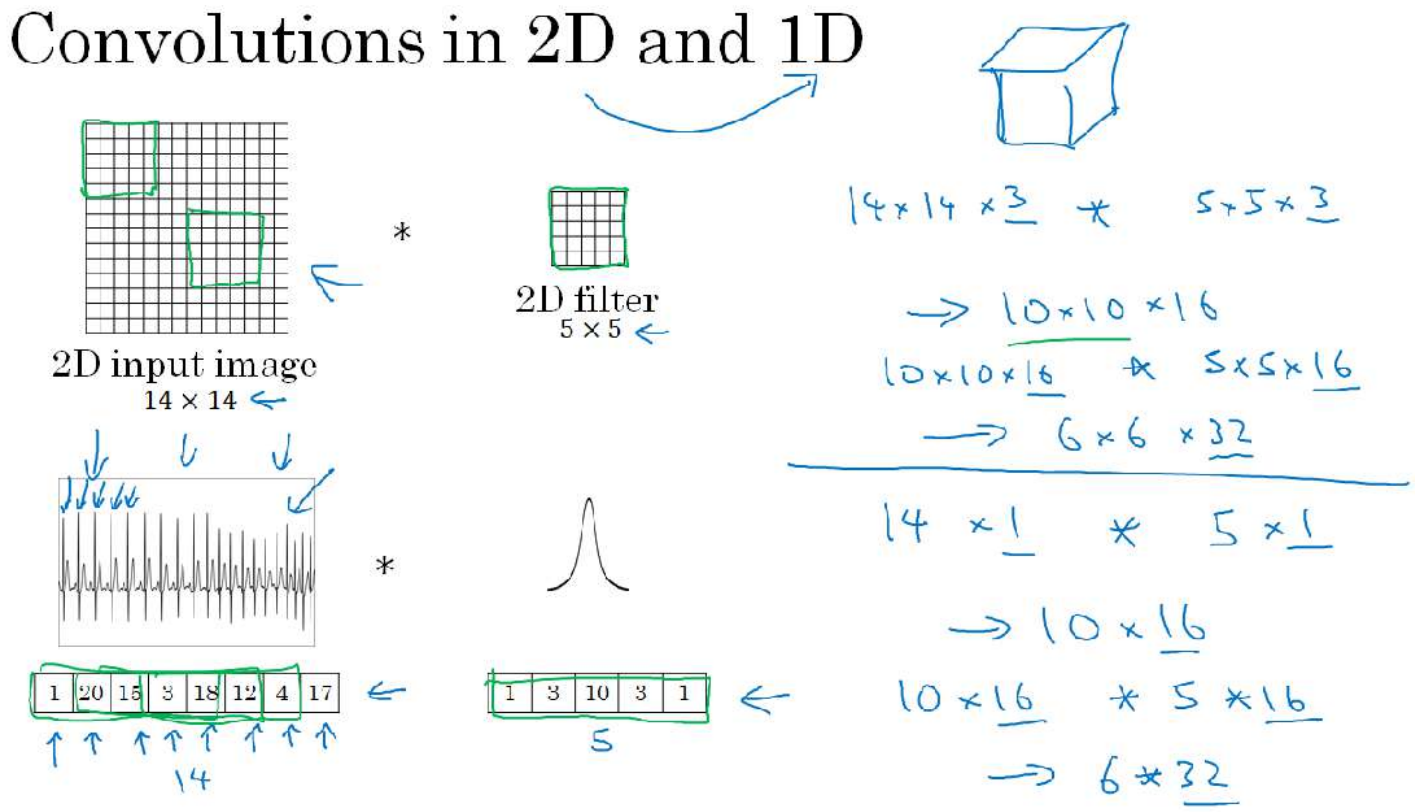
[Gatys et al., 2015. A neural algorithm of artistic style]

So, the overall style cost function, you can define as sum over all the different layers of the style cost function for that layer. We should define the book weighted by some set of parameters, by some set of additional hyperparameters, which we'll denote as lambda l here. So what it does is allows you to use different layers in a neural network. Well of the early ones, which measure relatively simpler low level features like edges as well as some later layers, which measure high level features and cause a neural network to take both low level and high level correlations into account when computing style.

1D and 3D Generalizations

We've learned a lot about ConvNets, everything ranging from the architecture of the ConvNet to how to use it for image recognition, to object detection, to face recognition and neural-style transfer. And even though most of the discussion has focused on images, on sort of 2D data, because images are so pervasive.

It turns out that many of the ideas you've learned about also apply, not just to 2D images but also to 1D data as well as to 3D data. Let's take a look.



In the first week of this course, you learned about the 2D convolution, where you might input a 14 x 14 image and convolve that with a 5 x 5 filter. And you saw how 14 x 14 convolved with 5 x 5, this gives you a 10 x 10 output and if you have multiple channels, maybe those 14 x 14 x 3, then it would be 5 x 5 that matches the same 3. And then if you have multiple filters, say 16 filters, you end up with 10 x 10 x 16. It turns out that a similar idea can be applied to 1D data as well. For example, on the left is an EKG signal, also called an electrocardioagram. Basically if you place an electrode over your chest, this measures the little voltages that vary across your chest as your heart beats. Because the little electric waves generated by your heart's beating can be measured with a pair of electrodes. And so this is an EKG of someone's heart beating. And so each of these peaks corresponds to one heartbeat. So if you want to use EKG signals to make medical diagnoses, for example, then you would have 1D data because what EKG data is, is it's a time series showing the voltage at each instant in time. So rather than a 14 x 14 dimensional input, maybe you just have a 14 dimensional input. And in that case, you might want to convolve this with a 1 dimensional filter. So rather than the 5 by 5, you just have 5 dimensional filter. So with 2D data what a convolution will allow you to do was to take the same 5 x 5 feature detector and apply it across at different positions throughout the image. And that's how you wound up with your 10 x 10 output. What a 1D filter allows you to do is take your 5 dimensional filter and similarly apply that in lots of different positions throughout this 1D signal. And so if you apply this convolution, what you find is that a 14 dimensional thing convolved with this 5 dimensional thing, this would give you a 10 dimensional output. And again, if you have multiple channels, you might have in this case you can use just 1 channel, if you have 1 lead or 1 electrode for EKG, so times 5 x 1. And if you have 16 filters, maybe end up with 10 x 16 over there, and this could be one layer of your ConvNet. And then for the next layer of your ConvNet, if you input a 10 x 16 dimensional input and you might convolve that with a 5 dimensional filter again. Then these have 16 channels, so that has a match. And we have 32 filters, then the output of another layer would be 6 x 32, if you have 32 filters, right? And the analogy to the the 2D data, this is similar to all of the 10 x 10 x 16 data and convolve it with a 5 x 5 x 16, and that has to match. That will give you a 6 by 6 dimensional output, and you have 32 filters, that's where the 32 comes from. So all of these ideas apply also to 1D data, where you can have the same feature detector, such as this, apply to a variety of positions. For example, to detect the different heartbeats in an EKG signal. But to use the same set of features to detect the heartbeats even at different positions along these time

series, and so ConvNet can be used even on 1D data. For along with 1D data applications, you actually use a recurrent neural network, which you learn about in the next course. But some people can also try using ConvNets in these problems. And in the next course on sequence models, which we will talk about recurring neural networks and LCM and other models like that. We'll talk about the pros and cons of using 1D ConvNets versus some of those other models that are explicitly designed to sequenced data. So that's the generalization from 2D to 1D.
How about 3D data? Instead of having a 1D list of numbers or a 2D matrix of numbers, you now have a 3D block, a three dimensional input volume of numbers. CAT scans, medical scans as one example of 3D volumes. But another example of data, you could treat as a 3D volume would be movie data, where the different slices could be different slices in time through a movie and you could use this to detect motion or people taking actions in movies. So that's it on generalization of ConvNets from 2D data to also 1D as well as 3D data. Image data is so pervasive that the vast majority of ConvNets are on 2D data, on image data, but I hope that these other models will be helpful to you as well.

**********END OF COURSE**********

Course 5: Sequence Models

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/nlp-sequence-models

****

Course 5: Sequence Models

****

In this course we'll learn how to build models for natural language, audio, and other sequence data. Thanks to deep learning, sequence algorithms are working far better than just two years ago, and this is enabling numerous exciting applications in speech recognition, music synthesis, chatbots, machine translation, natural language understanding, and many others.

You will:
- Understand how to build and train Recurrent Neural Networks (RNNs), and commonly-used variants such as GRUs and LSTMs.
- Be able to apply sequence models to natural language problems, including text synthesis.
- Be able to apply sequence models to audio applications, including speech recognition and music synthesis.

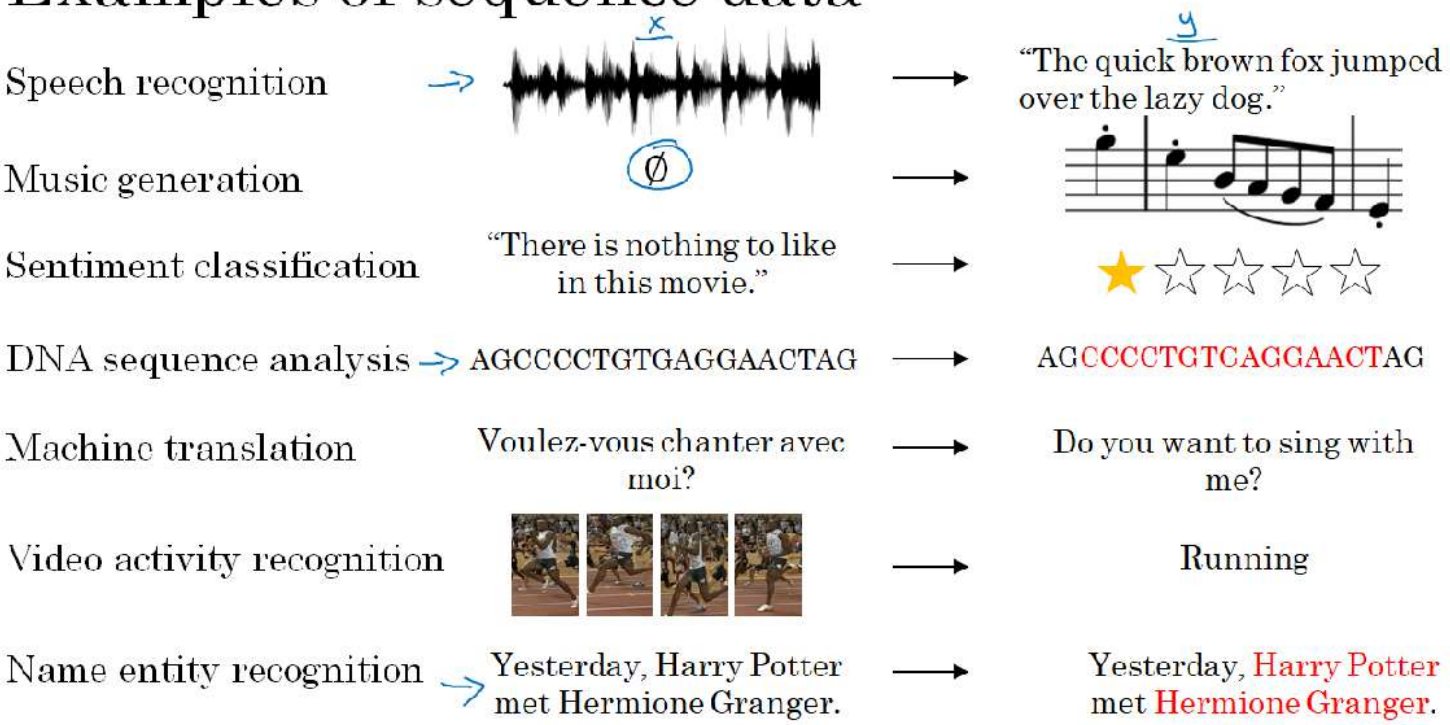
Week 1: Foundations of Convolutional Neural Networks

Learn about recurrent neural networks. This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and Bidirectional RNNs, which you are going to learn about in this section.

Recurrent Neural Networks

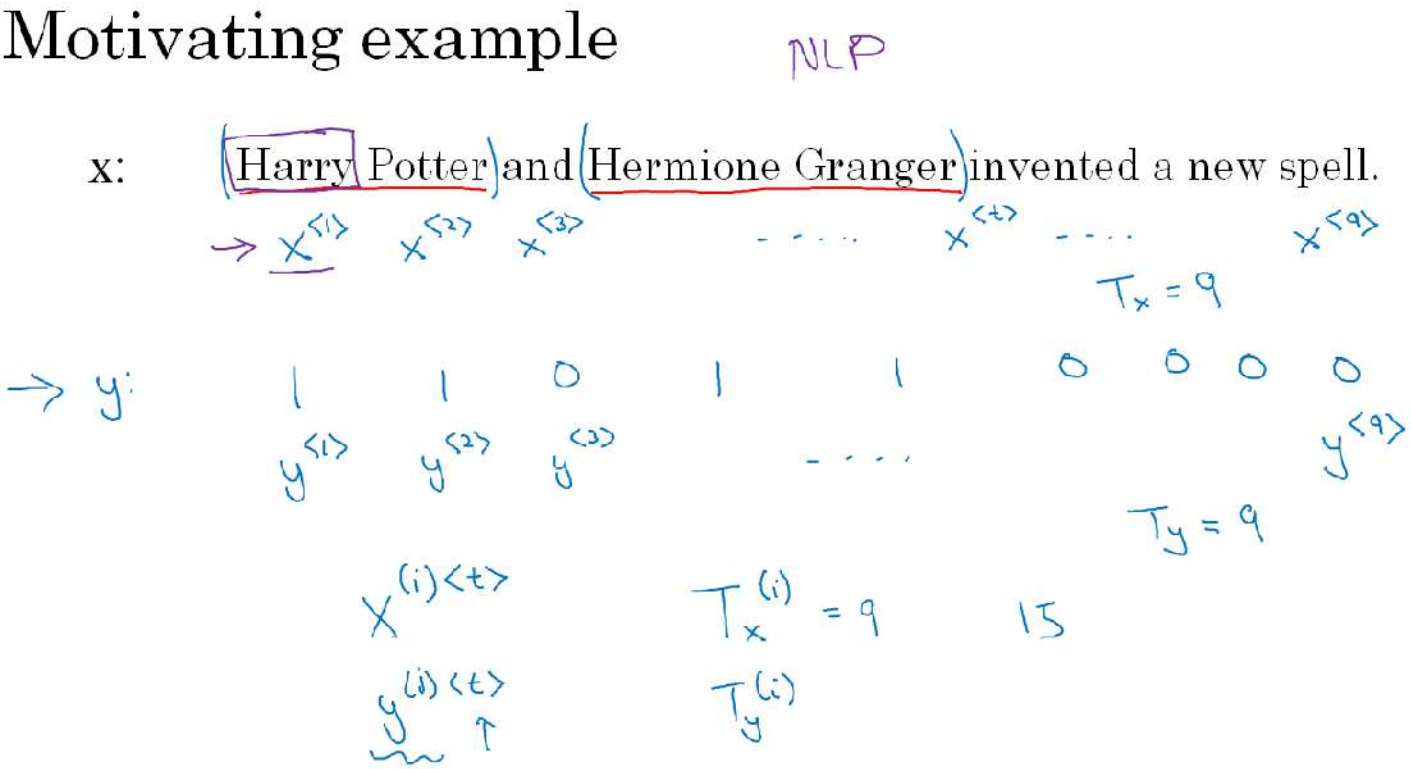
Why sequence models

In this course, we learn about sequence models, one of the most exciting areas in deep learning. Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas and in this course, we'll learn how to build these models for yourself. Let's start by looking at a few examples of where sequence models can be useful. In speech recognition you are given an input audio clip X and asked to map it to a text transcript Y. Both the input and the output here are sequence data, because X is an audio clip and so that plays out over time and Y, the output, is a sequence of words. So sequence models such as a recurrent neural networks and other variations, we'll learn about in a little bit have been very useful for speech recognition. Music generation is another example of a problem with sequence data. In this case, only the output Y is a sequence, the input can be the empty set, or it can be a single integer, maybe referring to the genre of music you want to generate or maybe the first few notes of the piece of music you want. But here X can be nothing or maybe just an integer and output Y is a sequence. In sentiment classification the input X is a sequence, so given the input phrase like, "There is nothing to like in this movie" how many stars do you think this review will be? Sequence models are also very useful for DNA sequence analysis. So your DNA is represented via the four alphabets A, C, G, and T. And so given a DNA sequence can you label which part of this DNA sequence say corresponds to a protein. In machine translation you are given an input sentence, voulez-vou chante avec moi? And you're asked to output the translation in a different language. In video activity recognition you might be given a sequence of video frames and asked to recognize the activity and in name entity recognition you might be given a sentence and asked to identify the people in that sentence. So all of these problems can be addressed as supervised learning with label data X, Y as the training set. But, as you can tell from this list of examples, there are a lot of different types of sequence problems. In some, both the input X and the output Y are sequences, and in that case, sometimes X and Y can have different lengths, or in this example and this example, X and Y have the same length and in some of these examples only either X or only the opposite Y is a sequence. So in this course we'll learn about sequence models are applicable, so all of these different settings. See diagram below of some of the examples of sequence data.



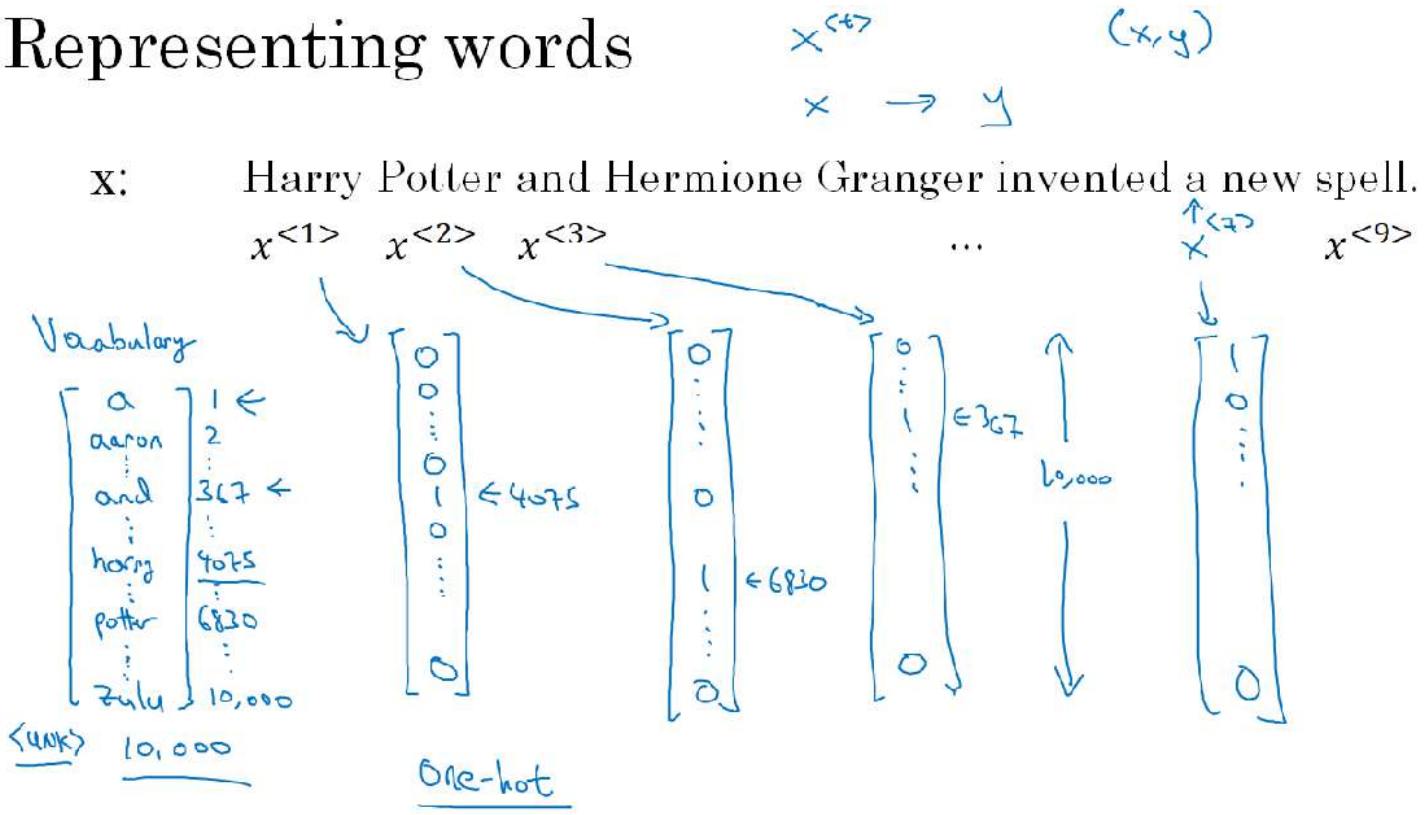
Notation

In the last section, we saw some of the wide range of applications through which you can apply sequence models. Let's start by defining a notation that we'll use to build up these sequence models. As a motivating example, let's say you want to build a sequence model to input a sentence like this, Harry Potter and Hermione Granger invented a new spell and these are characters by the way, from the Harry Potter sequence of novels by J. K. Rowling. And let say you want a sequence model to automatically tell you where are the peoples names in this sentence. So, this is a problem called **Named-entity recognition** and this is used by search engines for example, to index all of say the last 24 hours news of all the people mentioned in the news articles so that they can index them appropriately and name into the recognition systems can be used to find people's names, companies names, times, locations, countries names, currency names, and so on in different types of text. Now, given this input x let's say that you want a model to operate y that has one outputs per input word and the target output the design y tells you for each of the input words is that part of a person's name and technically this maybe isn't the best output representation, there are some more sophisticated output representations that tells you not just is a word part of a person's name, but tells you where are the start and ends of people's names their sentence



you want to know Harry Potter starts here, and ends here, starts here, and ends here. But for this motivating example, I'm just going to stick with this simpler output representation. Now, the input is the sequence of nine words. So, eventually we're going to have nine sets of features to represent these nine words, and index into the positions and sequence, I'm going to use X and then superscript angle brackets 1, 2, 3 and so on up to X angle brackets nine to index into the different positions. I'm going to use $X^{<t>}$ with the index t to index into positions, in the middle of the sequence and t implies that these are **temporal sequences** although whether the sequences are temporal one or not, I'm going to use the index t to index into the positions in the sequence and similarly for the outputs, we're going to refer to these outputs as y and go back at 1, 2, 3 and so on up to y nine. Let's also used T_X to denote the length of the input sequence, so in this case there are nine words. So T_x is equal to 9 and we used Ty to denote the length of the output sequence. In this example T_x is equal to Ty but you saw on the last section T_x and T_y can be different. So, you will remember that in the notation we've been using, we've been writing X round brackets i to denote the i training example. So, to refer to the t^{th} element or the t^{th} element in the sequence of training example i will use this notation and if Tx is the length of a sequence then different examples in your training set can have different lengths. And so $T_x^{(i)}$ would be the input sequence length for training example i, and similarly $y^{(i)<t>}$ means the t^{th} element in the output sequence of the i for an example and $Ty^{(i)}$ will be the length of the output sequence in the i training example. So into this example, $T_x^{(i)}$ is equal to 9 would be the highly different training example with a sentence of 15 words and $T_x^{(i)}$ will be close to 15 for that different training example. Now, that we're starting to work in **NLP or Natural Language Processing**. **Now, this is our first serious foray into NLP or Natural Language Processing** and one of the things we need to decide is, how to represent individual words in the sequence. So, how do you represent a word like Harry, and why should $x^{<1>}$ really be? Let's next talk about how we would represent individual words in a sentence.

Representing words



So, to represent a word in the sentence the first thing you do is come up with a Vocabulary. Sometimes also called a **Dictionary** and that means making a list of the words that you will use in your representations. So the first word in the vocabulary is a, that will be the first word in the dictionary. The second word is Aaron and then a little bit further down is the word and, and then eventually you get to the words Harry then eventually the word Potter, and then all the way down to maybe the last word in dictionary is Zulu. And so, a will be word one, Aaron is word two, and in my dictionary the word and appears in positional index 367. Harry appears in position 4075, Potter in position 6830, and Zulu is the last word to the dictionary is maybe word 10,000.

So in this example, I'm going to use a dictionary with size 10,000 words. This is quite small by modern NLP applications. For commercial applications, for visual size commercial applications, dictionary sizes of 30 to 50,000 are more common and 100,000 is not uncommon and then some of the large Internet companies will use dictionary sizes that are maybe a million words or even bigger than that. But you see a lot of commercial applications used dictionary sizes of maybe 30,000 or maybe 50,000 words. But I'm going to use 10,000 for illustration since it's a nice round number. So, if you have chosen a dictionary of 10,000 words and one way to build this dictionary will be be to look through your training sets and find the top 10,000 occurring words, also look through some of the online dictionaries that tells you what are the most common 10,000 words in the English Language saved. What you can do is then use one hot representations to represent each of these words. For example, $x^{<1>}$ which represents the word Harry would be a vector with all zeros except for a 1 in position 4075 because that was the position of Harry in the dictionary and then $x^{<2>}$ will be again similarly a vector of all zeros except for a 1 in position 6830 and then zeros everywhere else. The word and was represented as position 367 so $x^{<3>}$ would be a vector with zeros of 1 in position 367 and then zeros everywhere else and each of these would be a 10,000 dimensional vector if your vocabulary has 10,000 words and the word "a" is the first word of the dictionary, then $x^{<7>}$ which corresponds to word a, that would be the vector 1. This is the first element of the dictionary and then zero everywhere else. So in this representation, $x^{<t>}$ for each of the values of t in a sentence will be a **one-hot vector**, one-hot because there's exactly one one is on and zero everywhere else and you will have nine of them to represent the nine words in this sentence and the goal is given this representation for X to learn a mapping using a sequence model to then target output y, I will do this as a supervised learning problem, I'm sure given the table data with both x and y.

Representing words

x: Harry Potter and Hermione Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ ... $x^{<9>}$

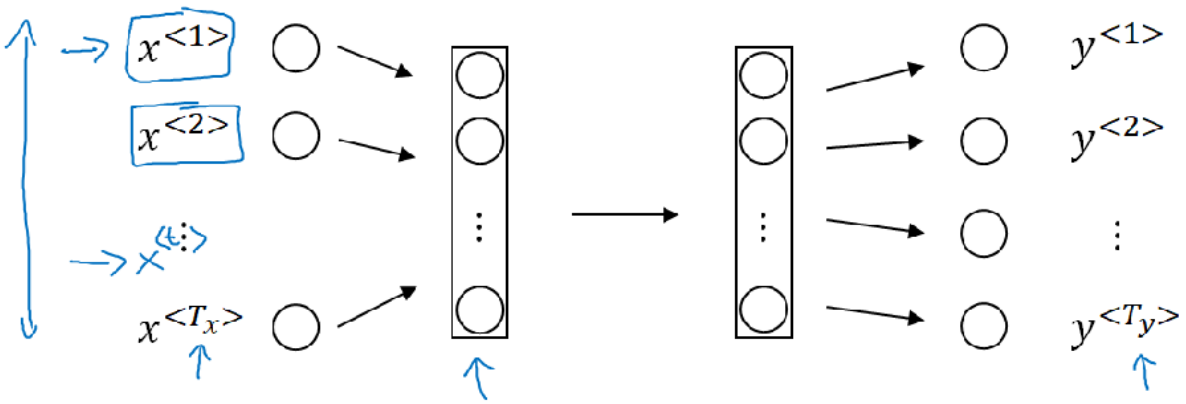
And = 367
Invented = 4700
A = 1
New = 5976
Spell = 8376
Harry = 4075
Potter = 6830
Hermione = 4200
Gran... = 4000

Then just one last detail, which we'll talk more about in a later section is, what if you encounter a word that is not in your vocabulary? Well the answer is, you create a new token or a new fake word called Unknown Word which under note as follows and go back as **UNK** to represent words not in your vocabulary, we'll come more to talk more about this later. So, to summarize in this section, we described a notation for describing your training set for both x and y for sequence data. In the next section let's start to describe a Recurrent Neural Networks for learning the mapping from X to Y.

Recurrent Neural Networks Model

In the last section, we saw the notation we used to define sequence learning problems. Now, let's talk about how you can build a model, build a neural network to drawing the mapping from X to Y. Now, one thing you could do is try to use a standard neural network for this task. So in our previous example, we had nine input words. So you could imagine trying to take these nine input words, maybe the nine one hot vectors and feeding them into a standard neural network, maybe a few hidden layers and then eventually, have this output the nine values zero or one that tell you whether each word is part of a person's name. But this turns out not to work well, and there are really two main problems with this. The first is that the inputs and outputs can be different lengths in different examples. So it's not as if every single example has the same input length T_x or the same output length T_y and maybe if every sentence had a maximum length, maybe you could pad, or zero pad every input up to that maximum length, but this still doesn't seem like a good representation and in a second, it might be more serious problem is that a naive neural network architecture like this below

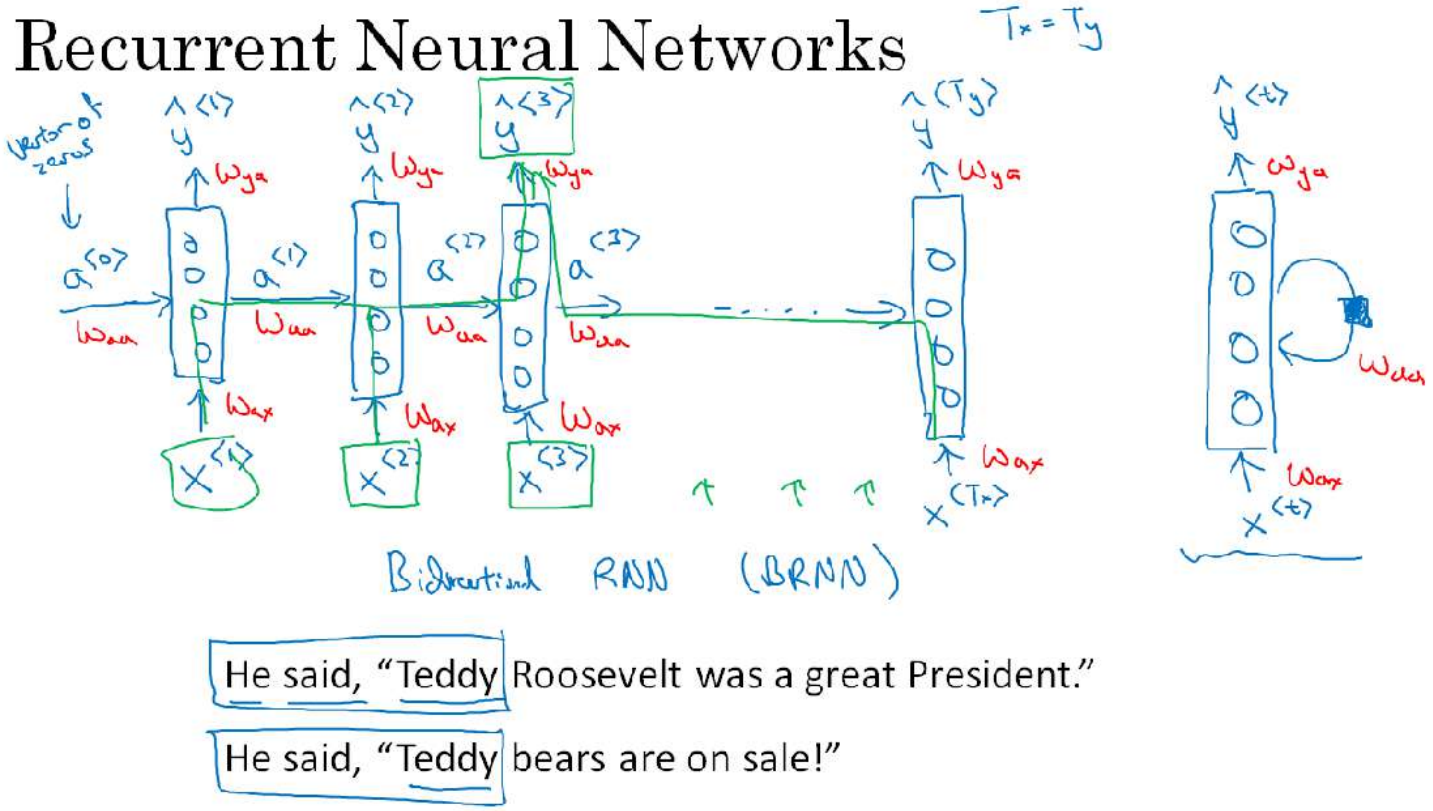
Why not a standard network?



Problems:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text.

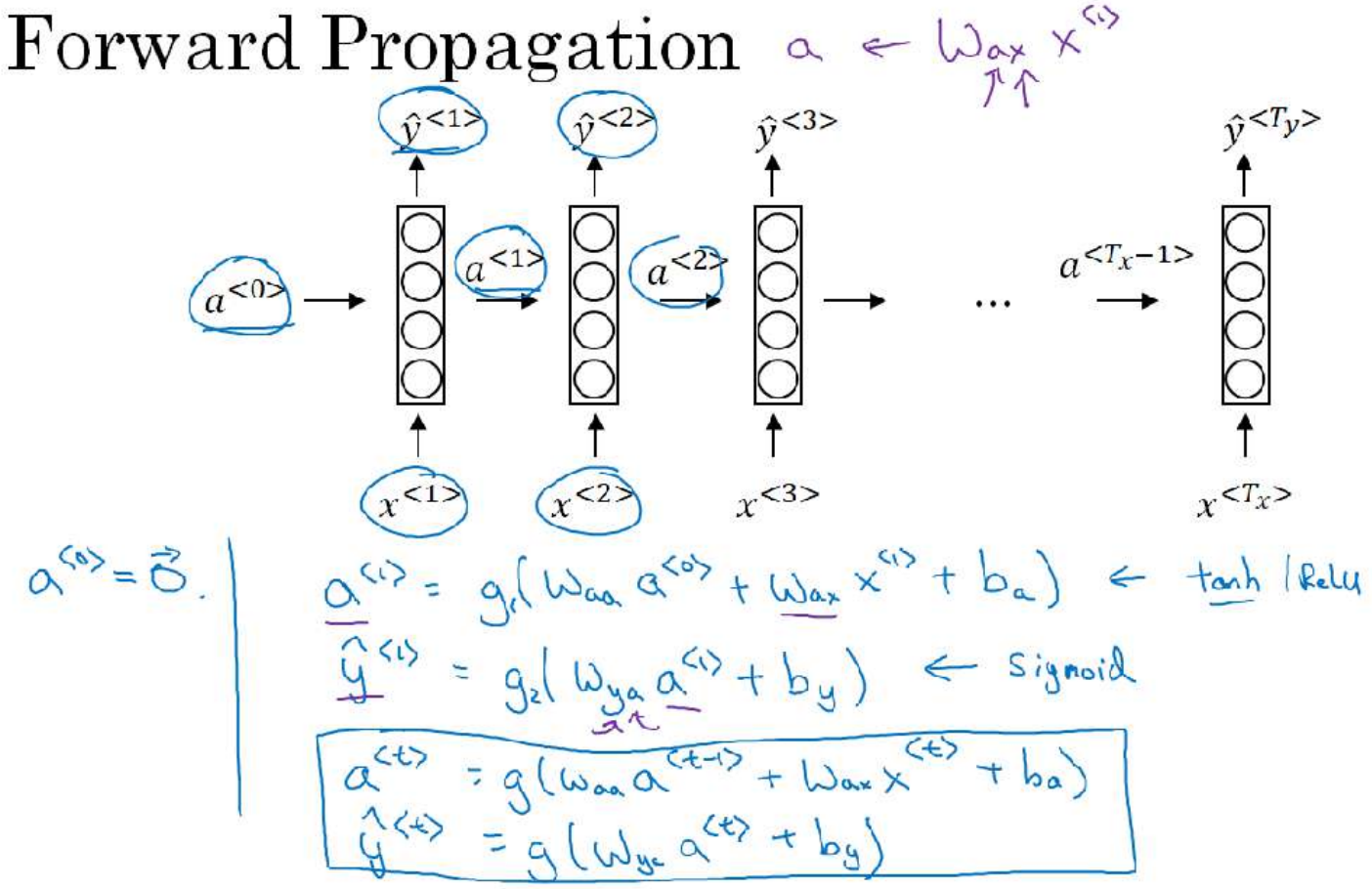
it doesn't share features learned across different positions of techs. In particular, if the neural network has learned that maybe the word heavy appearing in position one gives a sign that that is part of a person's name, then one would be nice if it automatically figures out that heavy appearing in some other position, XT also means that that might be a person's name. And this is maybe similar to what you saw in convolutional neural networks where you want things learned for one part of the image to generalize quickly to other parts of the image, and we'd like similar effect for sequence data as well. And similar to what you saw with convnets using a better representation will also let you reduce the number of parameters in your model.



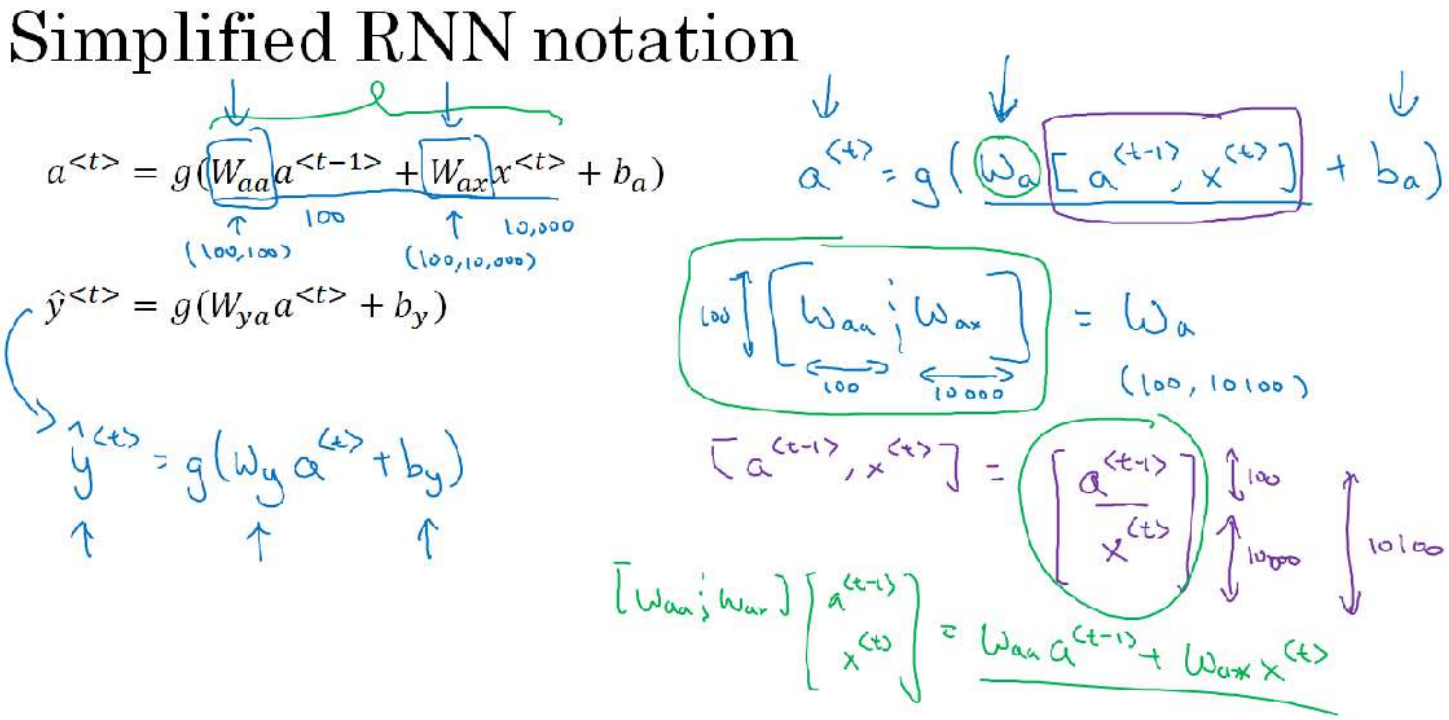
So previously, we said that each of these is a 10,000 dimensional one vector. And so, this is just a very large input layer. If the total input size was maximum number of words times 10,000, and the weight matrix of this first layer would end up having an enormous number of parameters. So a recurrent neural network which will start to describe in the next slide, does not have either of these disadvantages. So what is a recurrent neural network? Let's build one out. So if you are reading the sentence from left to right, the first word you read is the some first where say X1. What we're going to do is take the first word and feed it into a neural network layer. I'm going to draw it like this. So that's a hidden layer of the first neural network. And look at how the neural network maybe try to predict the output. So is this part of a person's name or not? And what a **recurrent neural network does is when it then goes on to read the second word in a sentence, say X2, instead of just predicting Y2 using only X2, it also gets to input some information from whether a computer that time-step ones. So in particular, the activation value from time-step one is passed on to time-step 2. And then, at the next time-step, a recurrent neural network inputs the third word X3, and it tries to predict, output some prediction y-hat 3, and so on, up until the last time-step where inputs XTx, and then it outputs Y hat TY.** In this example, Tx=Ty, and the architecture will change a bit if Tx and Ty are not identical. **And so, at each time-step, the recurrent neural network passes on this activation to the next time-step for it to use. And to kick off the whole thing, we'll also have some made up activation at time zero. This is usually the vector of zeroes.** Some researchers will initialize a zero randomly have other ways to initialize a zero but really having a vector zero is just a fake. Time Zero activation is the most common choice. And so that does input into the neural network. In some research papers or in some books, you see this type of neural network drawn with the following diagram in which every time-step, you input X and output Y hat, maybe sometimes there will be a T index there, and then to denote the recurrent connection, sometimes people will draw a loop like that, that the layer feeds back to itself. Sometimes they'll draw a shaded box to denote that this is the shaded box here, denotes a time delay of one step. I personally find these recurrent diagrams much harder to interpret. And so throughout this course, I will tend to draw the on the road diagram like the one you have on the left. But if you see something like the diagram on the right in a textbook or in a research paper, what it really means, or the way I tend to think about it is the mentally unrolled into the diagram you have on the left hand side. The recurrent neural network scans through the data from left to right. And the parameters it uses for each time step are shared. So there will be a set of parameters which we'll describe in greater detail on the next section, but the parameters governing

the connection from X1 to the hidden layer will be some set of the parameters we're going to write as WAX, and it's the same parameters WAX that it uses for every time-step I guess you could write WAX there as well. And the activations, the horizontal connections, will be governed by some set of parameters WAA, and is the same parameters WAA use on every time-step, and similarly, the sum WYA that governs the output predictions. And I'll describe in the next slide exactly how these parameters work. So in this recurrent neural network, what this means is that we're making the prediction for Y3 against the information not only from X3, but also the information from X1 and X2, because the information of X1 can pass through this way to help the prediction with Y3. Now one weakness of this RNN is that it only uses the information that is earlier in the sequence to make a prediction, in particular, when predicting Y3, it doesn't use information about the words X4, X5, X6 and so on. And so this is a problem because if you're given a sentence, he said, "Teddy Roosevelt was a great president." In order to decide whether or not the word Teddy is part of a person's name, it be really useful to know not just information from the first two words but to know information from the later words in the sentence as well, because the sentence could also happen, he said, "Teddy bears are on sale!" And so, given just the first three words, it's not possible to know for sure whether the word Teddy is part of a person's name. In the first example, it is, in the second example, is not, but you can't tell the difference if you look only at the first three words. So one limitation of this particular neural network structure is that the prediction at a certain time uses inputs or uses information from the inputs earlier in the sequence but not information later in the sequence. We will address this in a later video where we talk about a bidirectional recurrent neural networks or BRNNs. But for now, this simpler uni-directional neural network architecture will suffice for us to explain the key concepts. And we just have to make a quick modifications in these ideas later to enable say the prediction of Y-hat 3 to use both information earlier in the sequence as well as information later in the sequence, but we'll get to that in a later video. So let's not write to explicitly what are the calculations that this neural network does. Here's a cleaned out version of the picture of the neural network. As I mentioned previously, typically, you started off with the input a0 equals the vector of all zeroes.

Next. This is what a forward propagation looks like.



To compute a1, you would compute that as an activation function g, applied to Waa times a0 plus W a x times x1 plus a bias was going to write it as ba, and then to compute y hat 1 the prediction of times that one, that will be some activation function, maybe a different activation function, than the one above. But apply to WYA times a1 plus b y. And the notation convention I'm going to use for the sub zero of these matrices like that example, W a x. The second index means that this W a x is going to be multiplied by some x like quantity, and this means that this is used to compute some a like quantity. Like like so. And similarly, you notice that here WYA is multiplied by a sum a like quantity to compute a y type quantity. The activation function used in- to compute the activations will often be a tonnage and the choice of an RNN and sometimes, values are also used although the tonnage is actually a pretty common choice. And we have other ways of preventing the vanishing gradient problem which we'll talk about later this week. And depending on what your output y is, if it is a binary classification problem, then I guess you would use a sigmoid activation function or it could be a soft Max if you have a ky classification problem. But the choice of activation function here would depend on what type of output y you have. So, for the name entity recognition task, where Y was either zero or one. I guess the second g could be a signal and activation function. And I guess you could write g2 if you want to distinguish that this is these could be different activation functions but I usually won't do that. And then, more generally at time t, a t will be g of W a a times a, from the previous time-step, plus W a x of x from the current time-step plus B a, and y hat t is equal to g, again, it could be different activation functions but g of WYA times a t plus B y. So, these equations define for propagation in the neural network. Where you would start off with a zeroes [inaudible] and then using a zero and X1, you will compute a1 and y hat one, and then you, take X2 and use X2 and A1 to compute A2 and Y hat two and so on, and you carry out for propagation going from the left to the right of this picture. Now, in order to help us develop the more complex neural networks, I'm actually going to take this notation and simplify it a little bit.

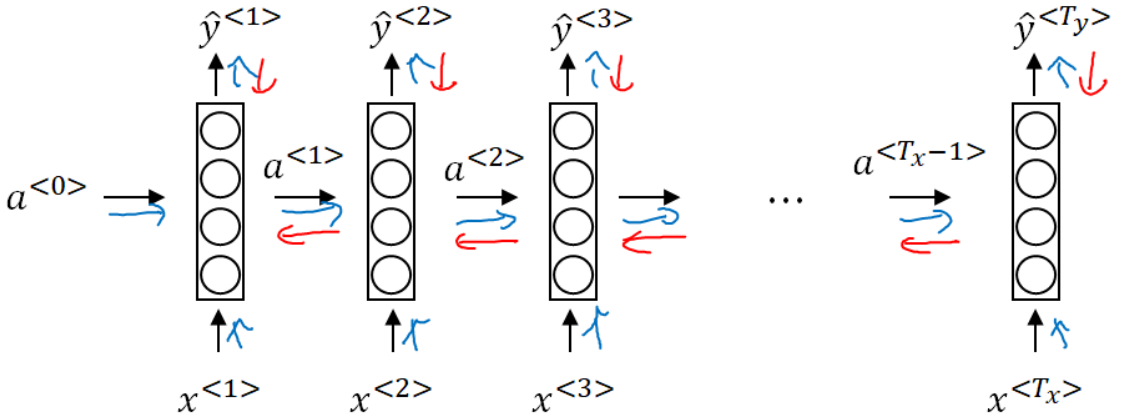


Backpropagation through time

We've already learned about the basic structure of an RNN. In this section, you'll see how backpropagation in a recurrent neural network works. As usual, when you implement this in one of the programming frameworks, often, the programming framework will automatically take care of backpropagation. But I think it's still useful to have a rough sense of how backprop works in RNNs. Let's take a look. You've seen how, for forward prop, you would computes these activations from left to right as follows in the neural network, and so you've outputs all of the predictions. In backprop, as you might already have guessed, you end up carrying backpropagation calculations in basically the

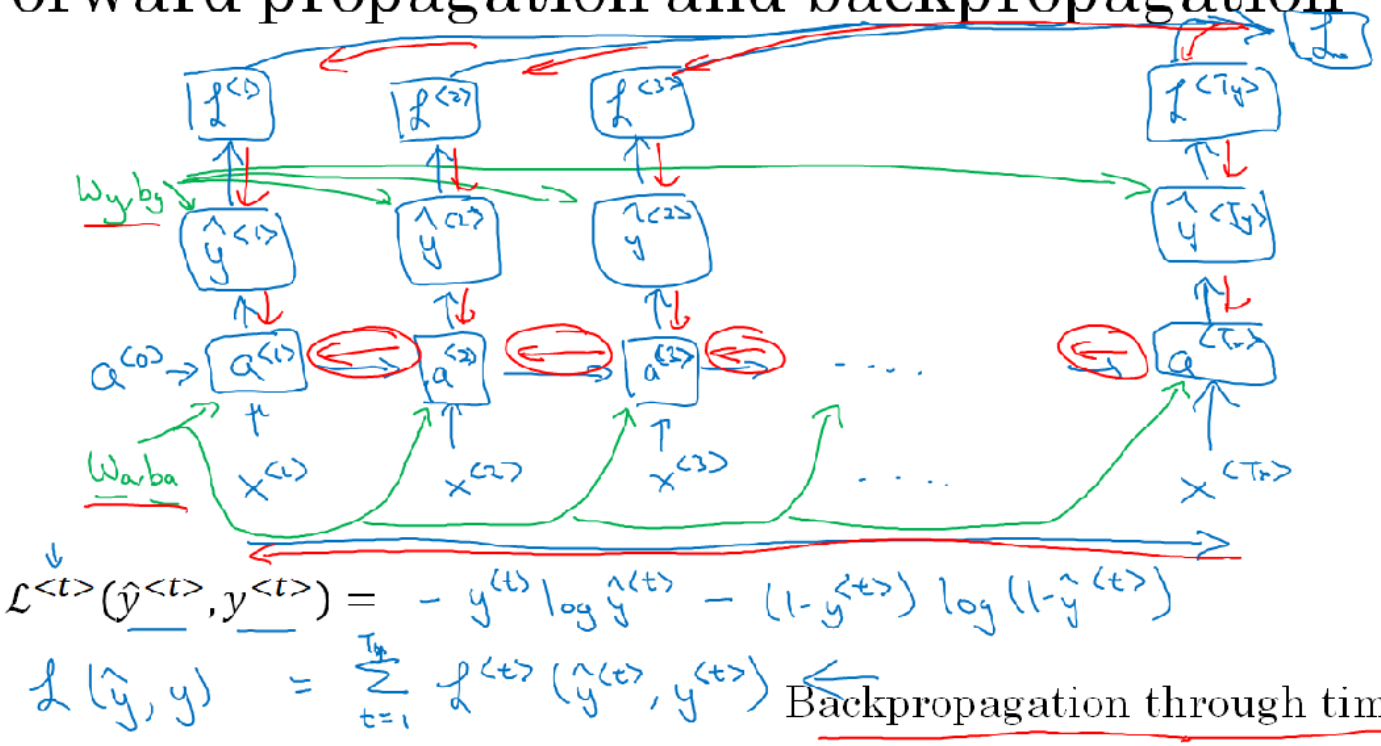
opposite direction of the forward prop arrows. So, let's go through the forward propagation calculation. You're given this input sequence x_1, x_2, x_3, up to x_tx. And then using x_1 and say, a_0, you're going to compute the activation, times that one, and then together, x_2 together with a_1 are used to compute a_2, and then a_3, and so on, up to a_tx. All right. And then to actually compute a_1, you also need the parameters.

Forward propagation and backpropagation



We'll just draw this in green, W_a and b_a, those are the parameters that are used to compute a_1. And then, these parameters are actually used for every single timestep so, these parameters are actually used to compute a_2, a_3, and so on, all the activations up to last timestep depend on the parameters W_a and b_a. Let's keep fleshing out this graph. Now, given a_1, your neural network can then compute the first prediction, y-hat_1, and then the second timestep, y-hat_2, y-hat_3, and so on, with y-hat_ty. And let me again draw the parameters of a different color. So, to compute y-hat, you need the parameters, W_y as well as b_y, and this goes into this node as well as all the others. So, I'll draw this in green as well. Next, in order to compute backpropagation, you need a loss function.

Forward propagation and backpropagation

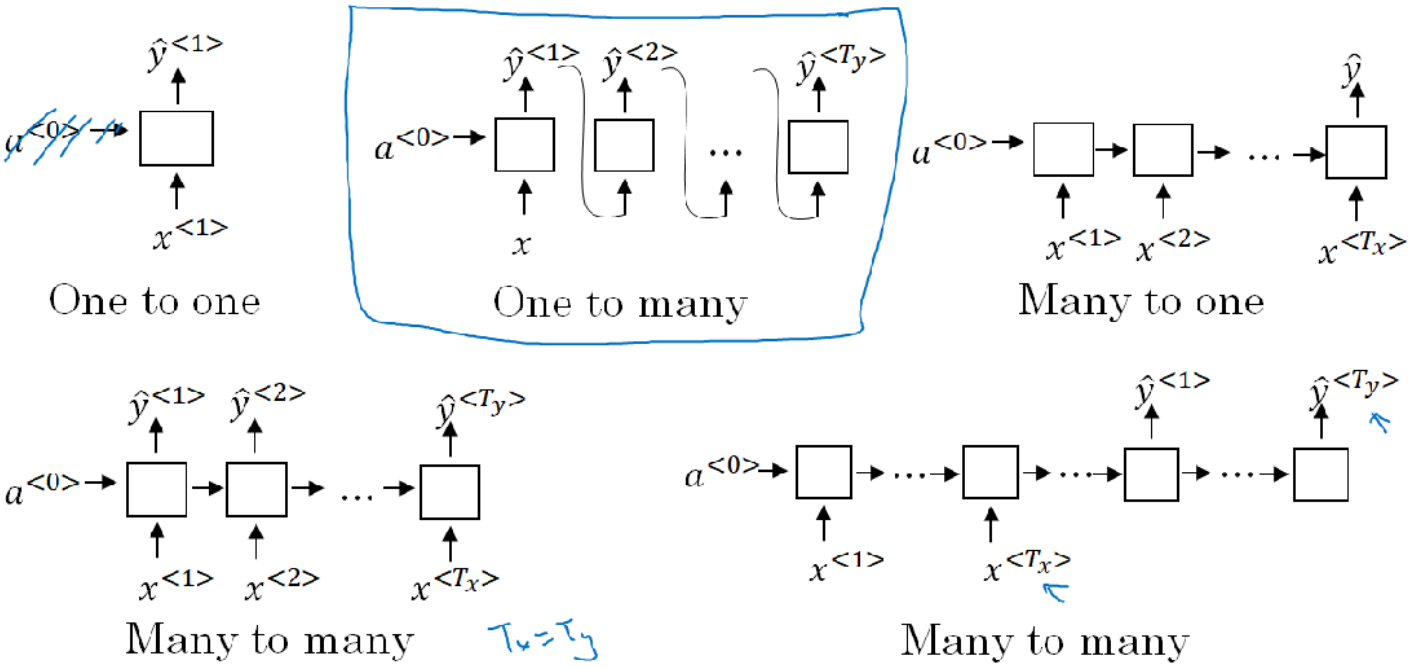


So let's define an element-wise loss force, which is supposed for a certain word in the sequence. It is a person's name, so y_t is one. And your neural network outputs some probability of maybe 0.1 of the particular word being a person's name. So I'm going to define this as the standard logistic regression loss, also called the cross entropy loss. This may look familiar to you from where we were previously looking at binary classification problems. So this is the loss associated with a single prediction at a single position or at a single time set, t, for a single word. Let's now define the overall loss of the entire sequence, so L will be defined as the sum overall t equals one to, i guess, T_x or T_y. T_x is equals to T_y in this example of the losses for the individual timesteps, comma y_t. And then, so, just have to L without this superscript T. This is the loss for the entire sequence. So, in a computation graph, to compute the loss given y-hat_1, you can then compute the loss for the first timestep given that you compute the loss for the second timestep, the loss for the third timestep, and so on, the loss for the final timestep. And then lastly, to compute the overall loss, we will take these and sum them all up to compute the final L using that equation, which is the sum of the individual per timestep losses. So, this is the computation problem and from the earlier examples **you've seen of backpropagation, it shouldn't surprise you that backprop then just requires doing computations or parsing messages in the opposite directions**. So, all of the four propagation steps arrows, so you end up doing that. And that then, allows you to compute all the appropriate quantities that lets you then, take the riveters, respected parameters, and update the parameters using gradient descent. **Now, in this back propagation procedure, the most significant message or the most significant recursive calculation is this one, which goes from right to left, and that's why it gives this algorithm as well, a pretty fast full name called backpropagation through time. And the motivation for this name is that for forward prop, you are scanning from left to right, increasing indices of the time, t, whereas, the backpropagation, you're going from right to left, you're kind of going backwards in time. So this gives this, I think a really cool name, backpropagation through time, where you're going backwards in time, right? That phrase really makes it sound like you need a time machine to implement this output, but I just thought that backprop through time is just one of the coolest names for an algorithm**. So, I hope that gives you a sense of how forward prop and backprop in RNN works. Now, so far, you've only seen this main motivating example in RNN, in which the length of the input sequence was equal to the length of the output sequence. In the next section, I want to show you a much wider range of RNN architecture, so I'll let you tackle a much wider set of applications.

Different types of RNNs

So far, you've seen an RNN architecture where the number of inputs, Tx, is equal to the number of outputs, Ty. It turns out that for other applications, Tx and Ty may not always be the same, and in this section, we'll see a much richer family of RNN architectures. You might remember from the first section of this week, where the input x and the output y can be many different types and it's not always the case that Tx has to be equal to Ty. In particular, in this example, Tx can be length one or even an empty set and then, an example like movie sentiment classification, the output y could be just an integer from 1 to 5, whereas the input is a sequence and in name entity recognition, in the example we're using, the input length and the output length are identical, but there are also some problems were the input length and the output length can be different.They're both our sequences but have different lengths, such as machine translation where a French sentence and English sentence can mean two different numbers of words to say the same thing. So it turns out that we could modify the basic RNN architecture to address all of these problems and the presentation in this section was inspired by a blog post by Andrej Karpathy, titled, **"The Unreasonable Effectiveness of Recurrent Neural Networks"**.

Summary of RNN types



Language model and sequence generation

Language modeling is one of the most basic and important tasks in natural language processing. There's also one that RNNs do very well. Where you build a language model and use it to generate Shakespeare-like texting, other types of text. Let's get started. So what is a language model? Let's say you're building this speech recognition system and you hear the sentence, the apple and pear salad was delicious. So what did you just hear me say? Did I say the apple and pair salad, or did I say the apple and pear salad?

You probably think the second sentence is much more likely, and in fact, that's what a good speech recognition system would help with even though these two sentences sound exactly the same and the way a speech recognition system picks the second sentence is by using a language model which tells it what the probability is of either of these two sentences.

What is language modelling?

Speech recognition

The apple and pair salad.

→ The apple and pear salad.

$P(\text{The apple and pair salad}) = 3.2 \times 10^{-13}$

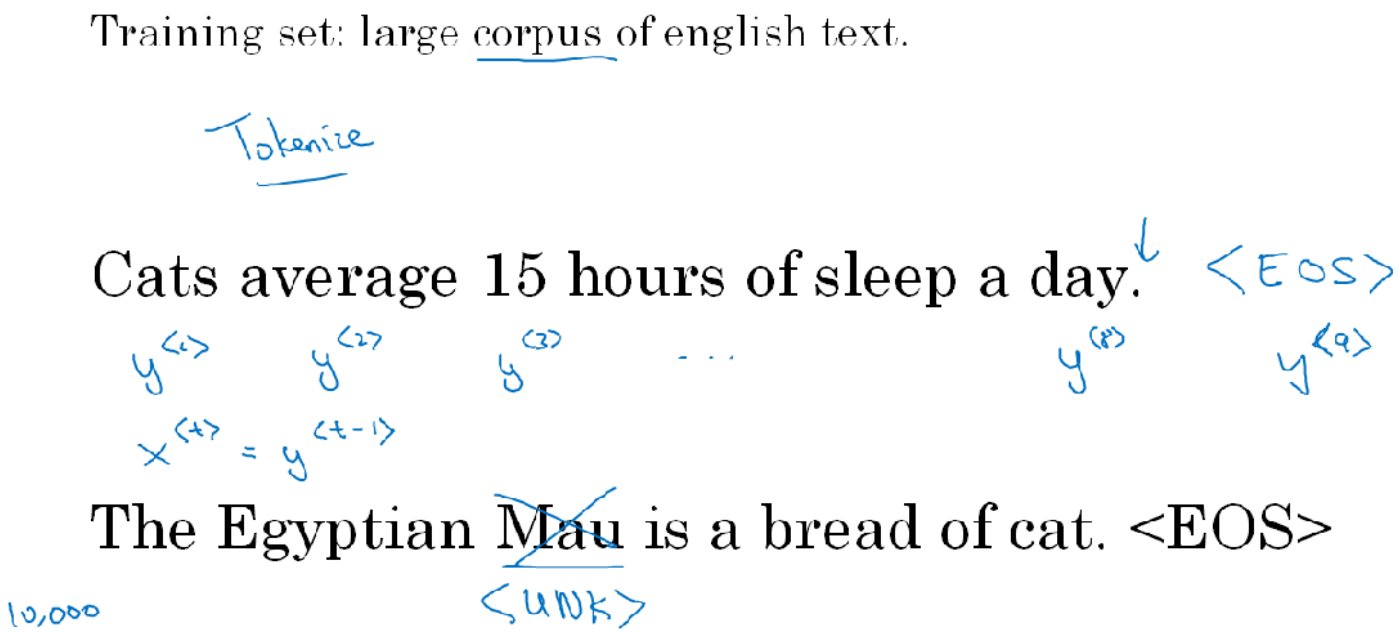
$P(\text{The apple and pear salad}) = 5.7 \times 10^{-10}$

$P(\text{Sentence}) = ?$ $\quad P(y^{<1>}, y^{<2>}, \ldots, y^{<T_y>})$

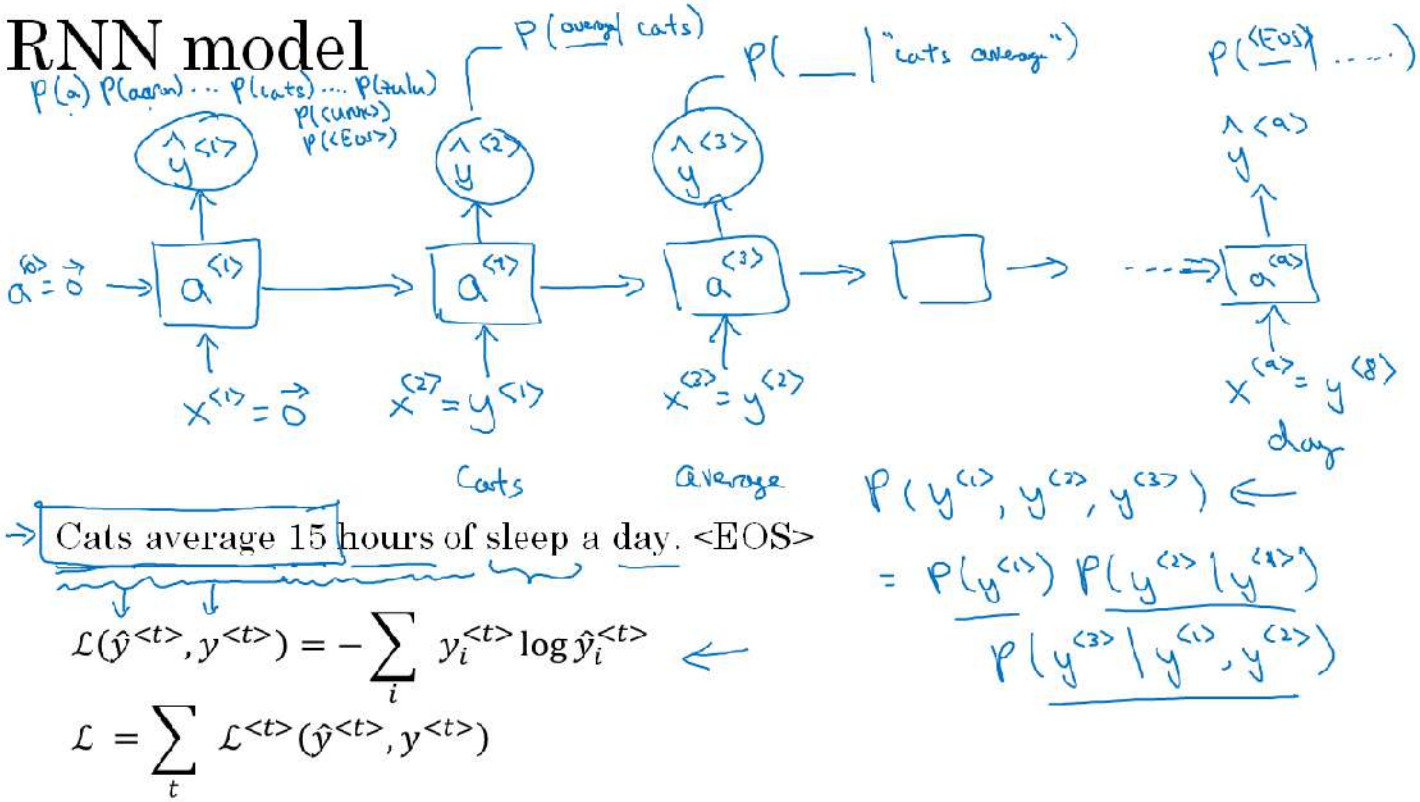
For example, a language model might say that the chance for the first sentence is 3.2×10^{-13} and the chance of the second sentence is say 5.7×10^{-10} and so, with these probabilities, the second sentence is much more likely by over a factor of 10^3 compared to the first sentence and that's why speech recognition system will pick the second choice. So what a language model does is given any sentence is job is to tell you what is the probability of a sentence, of that particular sentence and by probability of sentence I mean, if you want to pick up a random newspaper, open a random email or pick a random webpage or listen to the next thing someone says, the friend of you says. What is the chance that the next sentence you use somewhere out there in the world will be a particular sentence like the apple and pear salad? and this is a fundamental component for both speech recognition systems as you've just seen, as well as for machine translation systems where translation systems wants output only sentences that are likely and so the basic job of a language model is to input a sentence, which I'm going to write as a sequence $y^{<1>}$, $y^{<2>}$ up to $y^{<Ty>}$ and for language model will be useful to represent a sentences as outputs y rather than inputs x. But what the language model does is it estimates the probability of that particular sequence of words.

So how do you build a language model? To build such a model using an RNN you would first need a training set comprising a large corpus of english text. Or text from whatever language you want to build a language model of. And the word corpus is an NLP terminology that just means a large body or a very large set of english text of english sentences. So let's say you get a sentence in your training set as follows.

Language modelling with an RNN



Cats average 15 hours of sleep a day. The first thing you would do is tokenize this sentence and that means you would form a vocabulary as we saw in an earlier section and then map each of these words to, say, one hot vectors, alter indices in your vocabulary. One thing you might also want to do is model when sentences end. So another common thing to do is to add an extra token called a EOS. That stands for End Of Sentence that can help you figure out when a sentence ends. We'll talk more about this later, but the EOS token can be appended to the end of every sentence in your training sets if you want your models explicitly capture when sentences end.



So in this example, we have y1, y2, y3, 4, 5, 6, 7, 8, 9. Nine inputs in this example if you append the end of sentence token to the end. And doing the tokenization step, you can decide whether or not the period should be a token as well. In this example, I'm just ignoring punctuation. So I'm just using

day as another token. And omitting the period, if you want to treat the period or other punctuation as explicit token, then you can add the period to you vocabulary as well. Now, one other detail would be what if some of the words in your training set, are not in your vocabulary. So if your vocabulary uses 10,000 words, maybe the 10,000 most common words in English, then the term Mau as in the Egyptian Mau is a breed of cat, that might not be in one of your top 10,000 tokens. So in that case you could take the word Mau and replace it with a unique token called UNK or stands for unknown words and would just model, the chance of the unknown word instead of the specific word now. Having carried out the tokenization step which basically means taking the input sentence and mapping out to the individual tokens or the individual words in your vocabulary. Next let's build an RNN to model the chance of these different sequences. And one of the things we'll see on the next slide is that you end up setting the inputs x<t> = y<t-1> or you see that in a little bit. So let's go on to built the RNN model and I'm going to continue to use this sentence as the running example. This will be an RNN architecture. At time 0 you're going to end up computing some activation a1 as a function of some inputs x1, and x1 will just be set it to the set of all zeroes, to 0 vector. And the previous A0, by convention, also set that to vector zeroes. But what A1 does is it will make a soft max prediction to try to figure out what is the probability of the first words y. And so that's going to be y<1>. So what this step does is really, it has a soft max it's trying to predict. What is the probability of any word in the dictionary? That the first one is a, what's the chance that the first word is Aaron? And then what's the chance that the first word is cats? All the way to what's the chance the first word is Zulu? Or what's the first chance that the first word is an unknown word? Or what's the first chance that the first word is the in the sentence they'll have, shouldn't have to read? Right, so y hat 1 is output to a soft max, it just predicts what's the chance of the first word being, whatever it ends up being. And in our example, it wind up being the word cats, so this would be a 10,000 way soft max output, if you have a 10,000-word vocabulary. Or 10,002, I guess you could call unknown word and the sentence is two additional tokens. Then, the RNN steps forward to the next step and has some activation, a<1> to the next step. And at this step, his job is try figure out, what is the second word?

But now we will also give it the correct first word. So we'll tell it that, gee, in reality, the first word was actually Cats so that's y1. So tell it cats, and this is why y1 = x2, and so at the second step the output is again predicted by a soft max. The RNN's jobs to predict was the chance of a being whatever the word it is. Is it a or Aaron, or Cats or Zulu or unknown whether EOS or whatever given what had come previously. So in this case, I guess the right answer was average since the sentence starts with cats average. And then you go on to the next step of the RNN. Where you now compute a3. But to predict what is the third word, which is 15, we can now give it the first two words. So we're going to tell it cats average are the first two words. So this next input here, x<3> = y<2>, so the word average is input, and this job is to figure out what is the next word in the sequence. So in other words trying to figure out what is the probability of anywhere than dictionary given that what just came before was cats.

Average, right? And in this case, the right answer is 15 and so on.

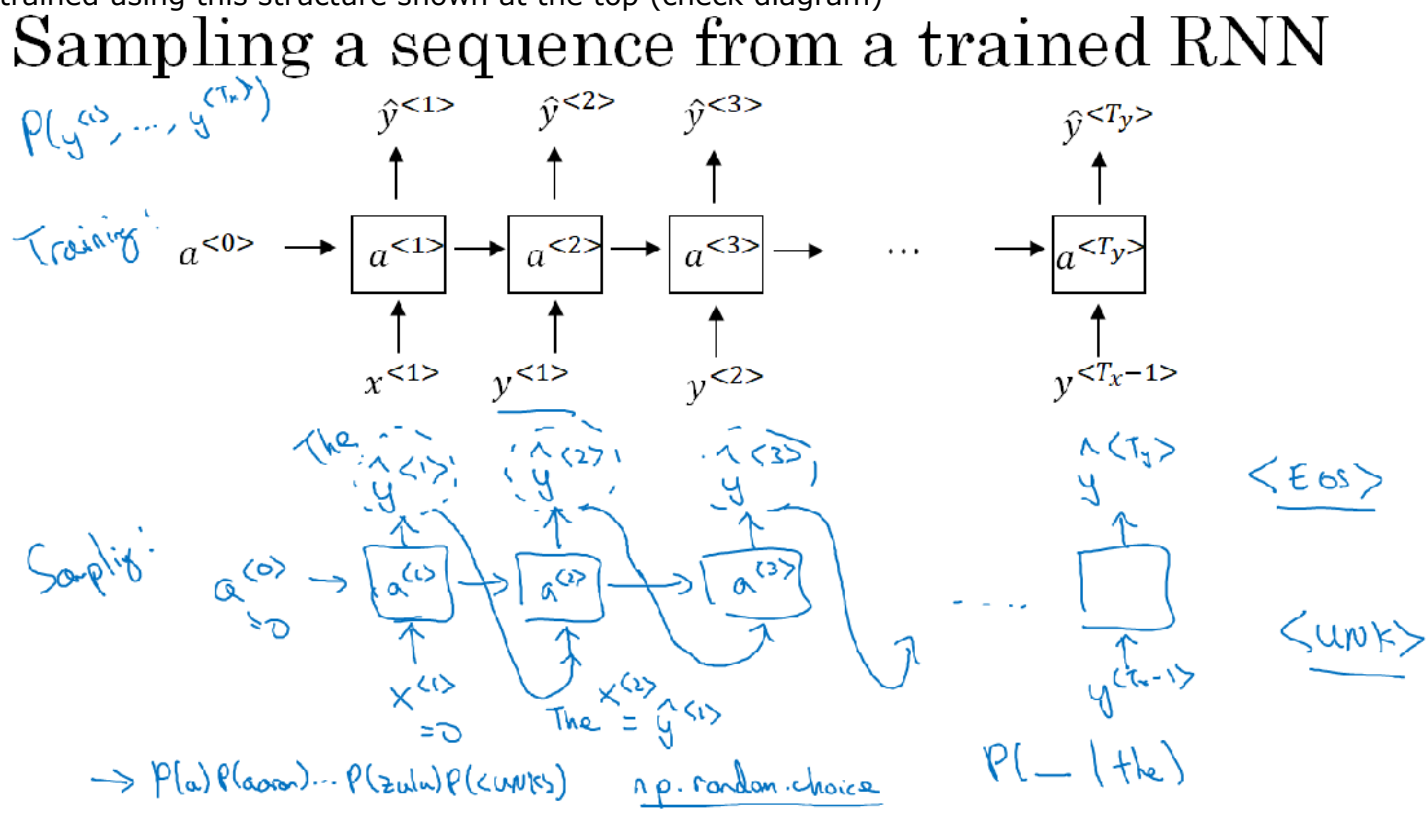
Until at the end, you end up at, I guess, time step 9, you end up feeding it x(9), which is equal to y(8), which is the word, day. And then this has A(9), and its jpob iws to output y hat 9, and this happens to be the EOS token. So what's the chance of whatever this given, everything that comes before, and hopefully it will predict that there's a high chance of it, EOS and the sentence token. So each step in the RNN will look at some set of preceding words such as, given the first three words, what is the distribution over the next word? And so this RNN learns to predict one word at a time going from left to right. Next to train us to a network, we're going to define the cos function. So, at a certain time, t, if the true word was yt and the new networks soft max predicted some y hat t, then this is the soft max loss function that you should already be familiar with. And then the overall loss is just the sum overall time steps of the loss associated with the individual predictions. And if you train this RNN on the last training set, what you'll be able to do is given any initial set of words, such as cats average 15 hours of, it can predict what is the chance of the next word.

And given a new sentence say, y(1), y(2), y(3)with just a three words, for simplicity, the way you can figure out what is the chance of this entire sentence would be. Well, the first soft max tells you what's the chance of y(1). That would be this first output. And then the second one can tell you what's the chance of p of y(2) given y(1). And then the third one tells you what's the chance of y(3) given y(1) and y(2). And so by multiplying out these three probabilities. And you'll see much more details of this

in the previous exercise. By multiply these three, you end up with the probability of the three sentence, of the three-word sentence. So that's the basic structure of how you can train a language model using an RNN.

Sampling novel sequences

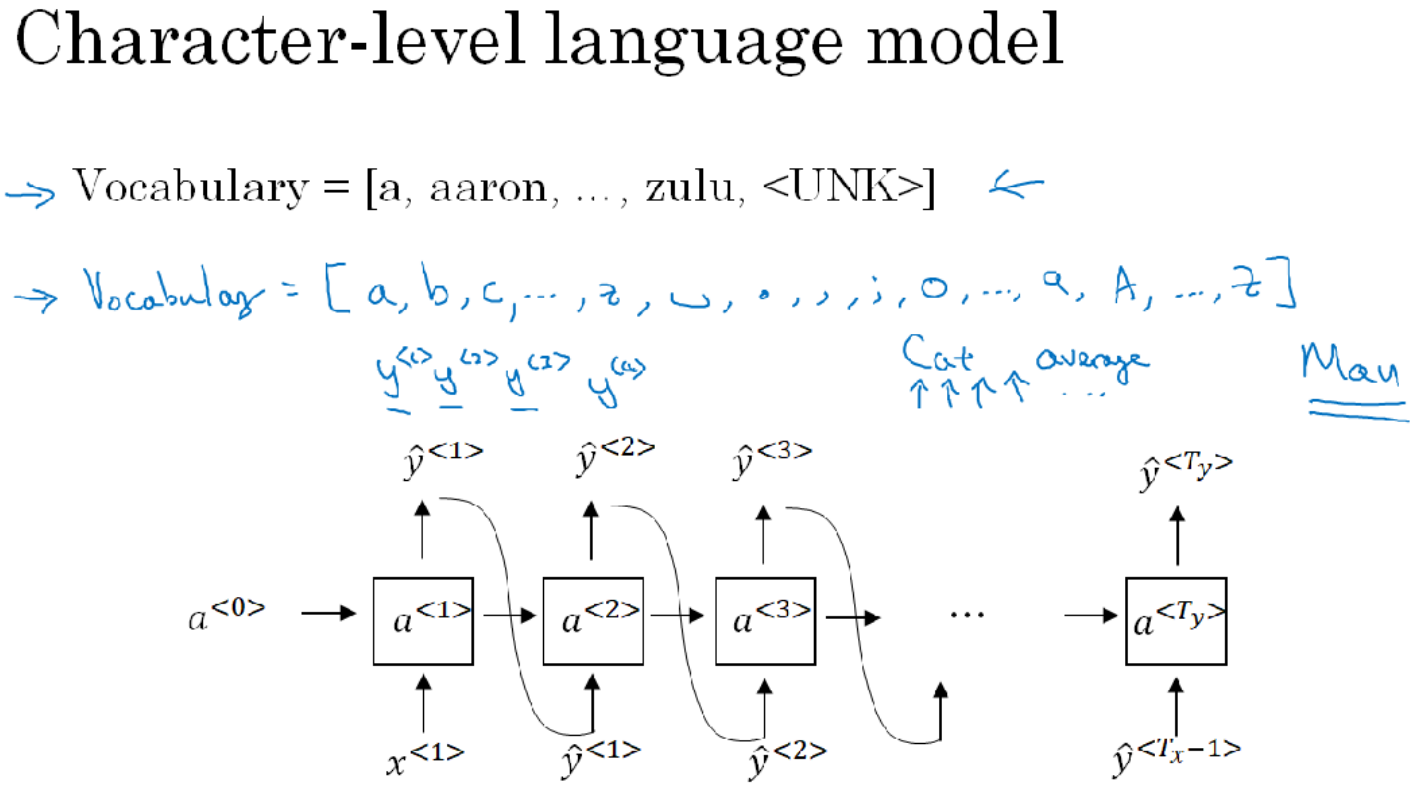
After you train a sequence model, one of the ways you can informally get a sense of what is learned is to have a sample novel sequences. Let's take a look at how you could do that. So remember that a sequence model, models the chance of any particular sequence of words as follows, and so what we like to do is sample from this distribution to generate noble sequences of words. So the network was trained using this structure shown at the top (check diagram)



But to sample, you do something slightly different, so what you want to do is first sample what is the first word you want your model to generate and so for that you input the usual x<1> equals 0, a<0> equals 0. And now your first time stamp will have some max probability over possible outputs. So what you do is you then randomly sample according to this soft max distribution. So what the soft max distribution gives you is it tells you what is the chance that it refers to this word "a", what is the chance that it refers to word "Aaron"? What's the chance it refers to "Zulu", what is the chance that the first word is the Unknown word token. Maybe it was a chance it was a end of sentence token and then you take this vector and use, for example, the numpy command **np.random.choice** to sample according to distribution defined by the vector probabilities, and that lets you sample the first words. Next you then go on to the second time step, and now remember that the second time step is expecting this y<1> as input but what you do is you then take the y1 hat that you just sampled and pass that in here as the input to the next timestep. So whatever works, you just chose the first time step passes this input in the second position, and then this soft max will make a prediction for what is y hat 2.

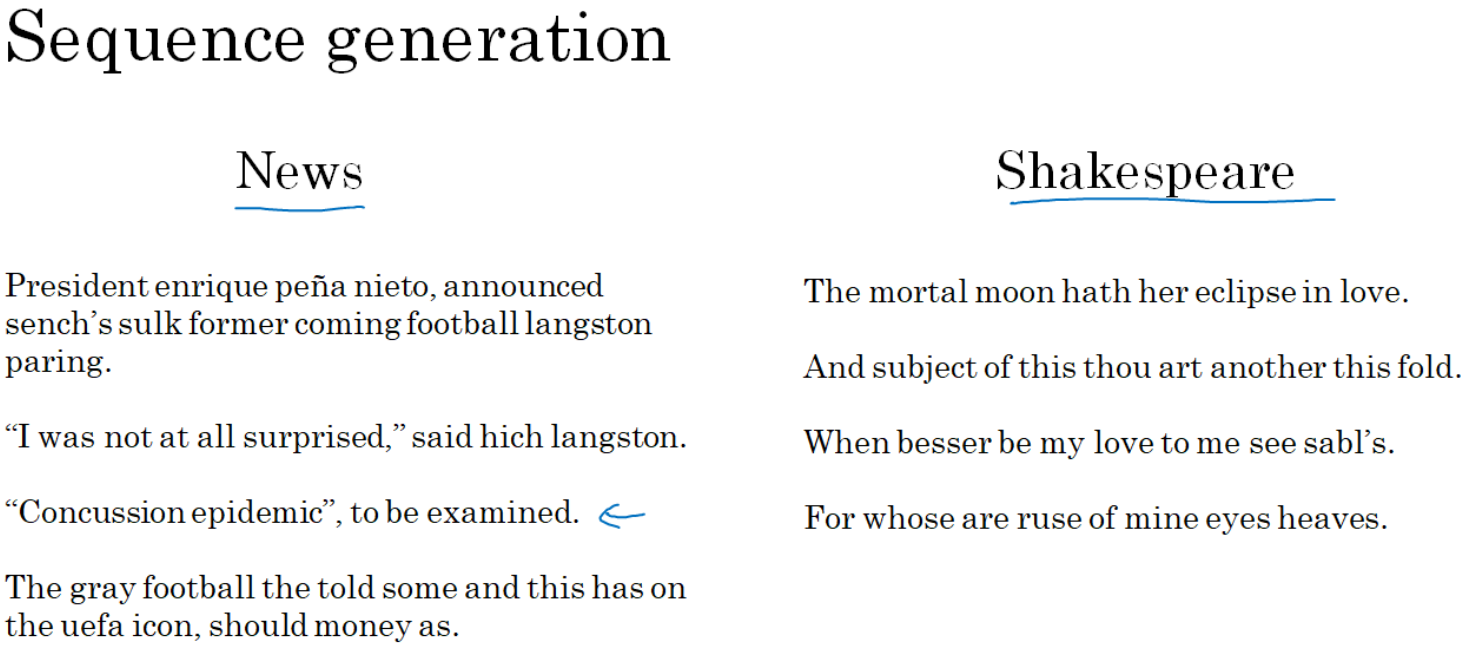
Example, let's say that after you sample the first word, the first word happened to be z, which is very common choice of first word. Then you pass in v as x2, which is now equal to y hat 1 and now you're trying to figure out what is the chance of what the second word is given that the first word is d. And this is going to be y hat 2. Then you again use this type of sampling function to sample y hat 2. And then at the next time stamp, you take whatever choice you had represented say as a one hard encoding. And pass that to next timestep and then you sample the third word to that whatever you chose, and you keep going until you get to the last time step. And so how do you know when the sequence ends? Well, one thing you could do is if the end of sentence token is part of your

vocabulary, you could keep sampling until you generate an EOS token and that tells you you've hit the end of a sentence and you can stop. Or alternatively, if you do not include this in your vocabulary then you can also just decide to sample 20 words or 100 words or something, and then keep going until you've reached that number of time steps and this particular procedure will sometimes generate an unknown word token. If you want to make sure that your algorithm never generates this token, one thing you could do is just reject any sample that came out as unknown word token and just keep resampling from the rest of the vocabulary until you get a word that's not an unknown word. Or you can just leave it in the output as well if you don't mind having an unknown word output. So this is how you would generate a randomly chosen sentence from your RNN language model. Now, so far we've been building a words level RNN, by which I mean the vocabulary are words from English. Depending on your application, one thing you can do is also build a character level RNN.



So in this case your vocabulary will just be the alphabets. Up to z, and as well as maybe space, punctuation if you wish, the digits 0 to 9 and if you want to distinguish the uppercase and lowercase, you can include the uppercase alphabets as well, and one thing you can do as you just look at your training set and look at the characters that appears there and use that to define the vocabulary and if you build a character level language model rather than a word level language model, then your sequence y1, y2, y3, would be the individual characters in your training data, rather than the individual words in your training data. So for our previous example, the sentence cats average 15 hours of sleep a day. In this example, c would be y1, a would be y2, t will be y3, the space will be y4 and so on. Using a character level language model has some pros and cons. **One is that you don't ever have to worry about unknown word tokens. In particular, a character level language model is able to assign a sequence like mau, a non-zero probability. Whereas if mau was not in your vocabulary for the word level language model, you just have to assign it the unknown word token. But the main disadvantage of the character level language model is that you end up with much longer sequences. So many english sentences will have 10 to 20 words but may have many, many dozens of characters. And so character language models are not as good as word level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence. And character level models are also just more computationally expensive to train.** So the trend I've been seeing in natural language processing is that for the most part, word level language model are still used, but as computers gets faster there are more and more applications where people are, at least in some special cases, starting to look at more character level models. But they tend to be much hardware, much more computationally expensive to train, so they

are not in widespread use today. Except for maybe specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary. So under these methods, what you can now do is build an RNN to look at the purpose of English text, build a word level, build a character language model, sample from the language model that you've trained. So here are some examples of text that were examples from a language model, actually from a culture level language model.

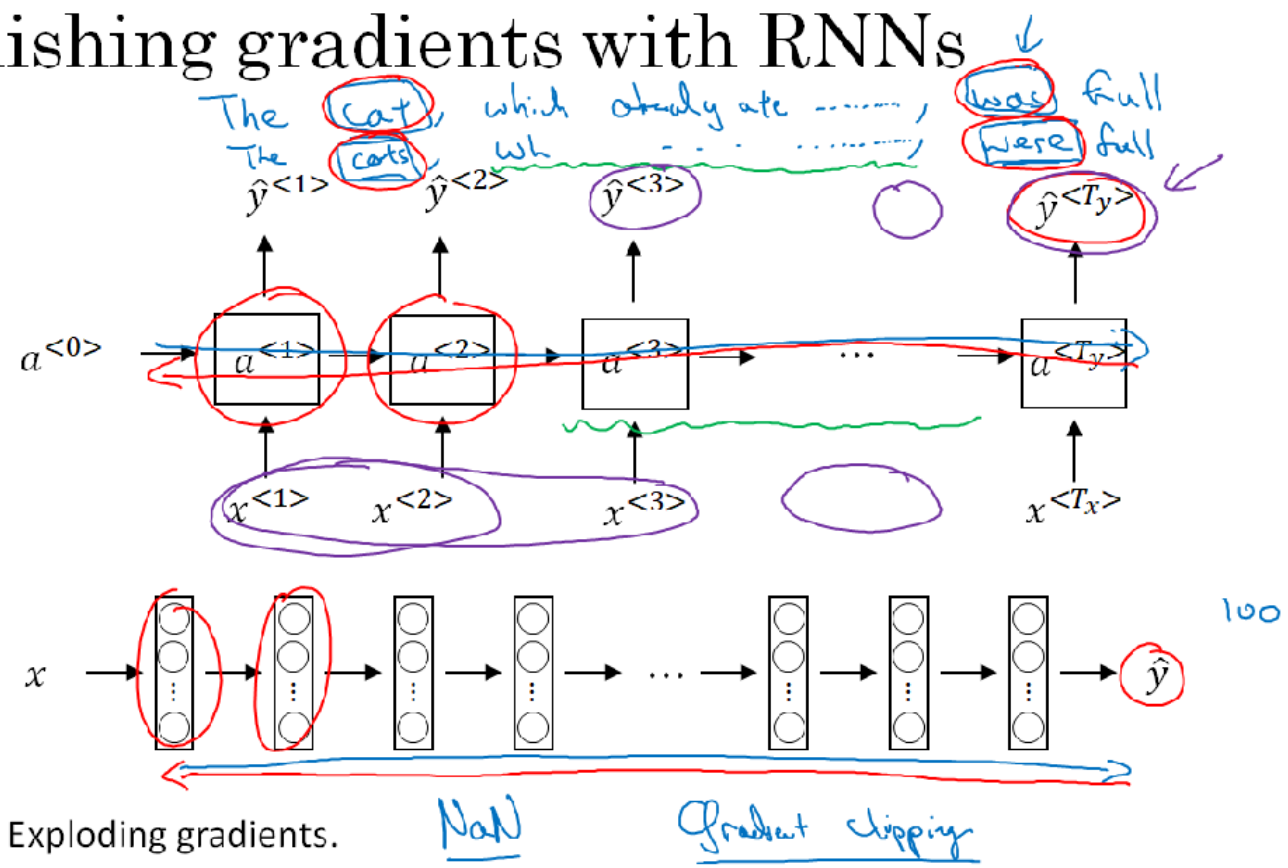


If the model was trained on news articles, then it generates texts like that shown on the left. And this looks vaguely like news text, not quite grammatical, but maybe sounds a little bit like things that could be appearing news, concussion epidemic to be examined. And it was trained on Shakespearean text and then it generates stuff that sounds like Shakespeare could have written it. 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.

So that's it for the basic RNN, and how you can build a language model using it, as well as sample from the language model that you've trained. In the next few sections, we'lll discuss further some of the challenges of training RNNs, as well as how to adjust some of these challenges, specifically vanishing gradients by building even more powerful models of the RNN. So in the next section we'll talk about the problem of vanishing the gradient and we will go on to talk about the GRU, Gate Recurring Unit as well as the LSTM models.

Vanishing gradients with RNNs

We've learned about how RNNs work and how they can be applied to problems like name entity recognition, as well as to language modeling, and we saw how backpropagation can be used to train in RNN. It turns out that one of the problems with a basic RNN algorithm is that it runs into **vanishing gradient problems**. Let's discuss that, and then in the next few sections, we'll talk about some solutions that will help to address this problem. So, you've seen pictures of RNNS that look like diagram shown below:



and let's take a language modeling example. Let's say you see this sentence, "The cat which already ate and maybe already ate a bunch of food that was delicious dot, dot, dot, dot, was full." And so, to be consistent, just because cat is singular, it should be the cat was, were then was, "The cats which already ate a bunch of food was delicious, and apples, and pears, and so on, were full." So to be consistent, it should be cat was or cats were. And this is one example of when language can have very long-term dependencies, where it worked at this much earlier can affect what needs to come much later in the sentence. But it turns out the basics RNN we've seen so far it's not very good at capturing very long-term dependencies. To explain why, you might remember from our early discussions of training very deep neural networks, that we talked about the vanishing gradients problem. So the diagram shown is a very, very deep neural network say, 100 layers or even much deeper than you would carry out forward prop, from left to right and then back prop and we said that, if this is a very deep neural network, then the gradient from just output y, would have a very hard time propagating back to affect the weights of these earlier layers, to affect the computations in the earlier layers and for an RNN with a similar problem, you have forward prop came from left to right, and then back prop, going from right to left and it can be quite difficult, because of the same vanishing gradients problem, for the outputs of the errors associated with the later time steps to affect the computations that are earlier and so in practice, what this means is, it might be difficult to get a neural network to realize that it needs to memorize the just see a singular noun or a plural noun, so that later on in the sequence that can generate either was or were, depending on whether it was singular or plural and notice that in English, this stuff in the middle could be arbitrarily long, right? So you might need to memorize the singular/plural for a very long time before you get to use that bit of information. So because of this problem, the basic RNN model has many local influences, meaning that the output y^<3> is mainly influenced by values close to y^<3> and a value here is mainly influenced by inputs that are somewhere close and it's difficult for the output here to be strongly influenced by an input that was very early in the sequence and this is because whatever the output is, whether this got it right, this got it wrong, it's just very difficult for the area to backpropagate all the way to the beginning of the sequence, and therefore to modify how the neural network is doing computations earlier in the sequence. So this is a weakness of the basic RNN algorithm. One, which was not addressed in the next few sections but if we don't address it, then RNNs tend not to be very good at capturing long-range dependencies and even though this discussion

has focused on vanishing gradients, you will remember when we talked about very deep neural networks, that we also talked about exploding gradients. We're doing back prop, the gradients should not just decrease exponentially, they may also increase exponentially with the number of layers you go through. **It turns out that vanishing gradients tends to be the bigger problem with training RNNs, although when exploding gradients happens, it can be catastrophic because the exponentially large gradients can cause your parameters to become so large that your neural network parameters get really messed up**. So it turns out that exploding gradients are easier to spot because the parameters just blow up and you might often see NaNs, or not a numbers, meaning results of a numerical overflow in your neural network computation and if you do see exploding gradients, one solution to that is apply gradient clipping. and what that really means, all that means is look at your gradient vectors, and if it is bigger than some threshold, re-scale some of your gradient vector so that is not too big. So there are clips according to some maximum value. So if you see exploding gradients, if your derivatives do explode or you see NaNs, just apply gradient clipping, and that's a relatively robust solution that will take care of exploding gradients. But vanishing gradients is much harder to solve and it will be the subject of the next few sections.

So to summarize, in an earlier course, you saw how the training of very deep neural network, you can run into a vanishing gradient or exploding gradient problems with the derivative, either decreases exponentially or grows exponentially as a function of the number of layers and in RNN, say in RNN processing data over a thousand times sets, over 10,000 times sets, that's basically a 1,000 layer or they go 10,000 layer neural network, and so, it too runs into these types of problems. **Exploding gradients, you could sort of address by just using gradient clipping, but vanishing gradients will take more work to address**. So what we do in the next sections is talk about GRU, the greater recurrent units, which is a very effective solution for addressing the vanishing gradient problem and will allow your neural network to capture much longer range dependencies.

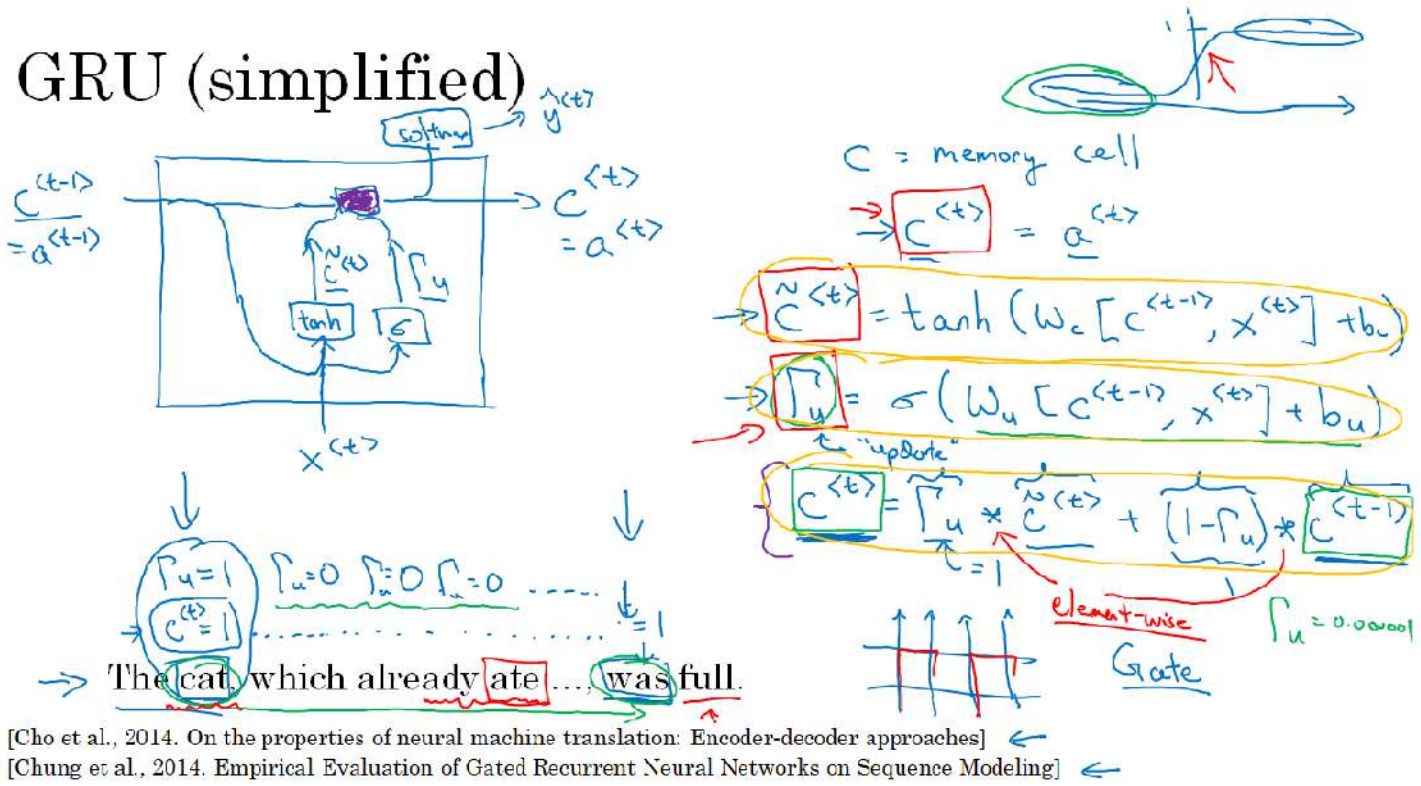
Gated Recurrent Unit (GRU)

You've seen how a basic RNN works. In this section, you learn about the **Gated Recurrent Unit** which is a modification to the RNN hidden layer that makes it much better capturing long range connections and helps a lot with the **vanishing gradient problems**. Let's take a look.

we've already seen the formula for computing the activations at time t of RNN. It's the activation function applied to the parameter Wa times the activations in the previous time set, the current input and then plus ba. So we're going to draw this as a picture. So the RNN unit, we're going to draw as a picture,
drawn as a box which inputs a of t-1, the activation for the last time-step and also inputs x<t> and these two go together and after some weights and after this type of linear calculation, if g is a tanh activation function, then after the tanh, it computes the output activation a. And the output activation a(t) might also be passed to say a softener unit or something that could then be used to output y<t>. So this is maybe a visualization of the RNN unit of the hidden layer of the RNN in terms of a picture. And I want to show you this picture because we're going to use a similar picture to explain the GRU or the Gated Recurrent Unit. Lots of the idea of GRU were due to these two papers respectively by Yu Young Chang, Kagawa, Gaza Hera, Chang Hung Chu and Jose Banjo. And I'm sometimes going to refer to this sentence which we'd seen in the last section to motivate that. Given a sentence like this, you might need to remember the cat was singular, to make sure you understand why that was rather than were. So the cat was for or the cats were for. So as we read in this sentence from left to right, the GRU unit is going to have a new variable called c, which stands for cell, for memory cell. And what the memory cell do is it will provide a bit of memory to remember, for example, whether cat was singular or plural, so that when it gets much further into the sentence it can still work under consideration whether the subject of the sentence was singular or plural. And so at time t the memory cell will have some value c of t. And what we see is that the GRU unit will actually output an activation value a of t that's equal to c of t. And for now I wanted to use different symbol c and a to denote the memory cell value and the output activation value, even though they are the same. I'm using this notation because when we talk about LSTMs, a little bit later, these will be two different values. But for now, for the GRU, c of t is equal to the output activation a of t. So these are the equations that govern the computations of a GRU unit. And every time-step, we're going to consider overwriting the memory cell with a value c tilde of t. So this is going to be a

candidate for replacing c of t. And we're going to compute this using an activation function tanh of Wc. And so that's the parameter to make sure it's Wc and we'll plus this parameter matrix, the previous value of the memory cell, the activation value as well as the current input value x<t>, and then plus the bias. So c tilde of t is going to be a candidate for replacing c<t>. And then the key, really the important idea of the GRU it will be that we have a gate. So the gate, I'm going to call gamma u. This is the capital Greek alphabet gamma subscript u, and u stands for update gate, and this will be a value between zero and one. And to develop your intuition about how GRUs work, think of gamma u, this gate value, as being always zero or one. Although in practice, your compute it with a sigmoid function applied to this. So remember that the sigmoid function looks like this. And so it's value is always between zero and one. And for most of the possible ranges of the input, the sigmoid function is either very, very close to zero or very, very close to one. So for intuition, think of gamma as being either zero or one most of the time. And this alphabet u stands for- I chose the alphabet gamma for this because if you look at a gate fence, looks a bit like this I guess, then there are a lot of gammas in this fence. So that's why gamma u, we're going to use to denote the gate. Also Greek alphabet G, right. G for gate. So G for gamma and G for gate. And then next, the key part of the GRU is this equation which is that we have come up with a candidate where we're thinking of updating c using c tilde, and then the gate will decide whether or not we actually update it. And so the way to think about it is maybe this memory cell c is going to be set to either zero or one depending on whether the word you are considering, really the subject of the sentence is singular or plural. So because it's singular, let's say that we set this to one. And if it was plural, maybe we would set this to zero, and then the GRU unit would memorize the value of the c<t> all the way until here, where this is still equal to one and so that tells it, oh, it's singular so use the choice was. And the job of the gate, of gamma u, is to decide when do you update these values. In particular, when you see the phrase, the cat, you know they you're talking about a new concept the especially subject of the sentence cat. So that would be a good time to update this bit and then maybe when you're done using it, the cat blah blah blah was full, then you know, okay, I don't need to memorize anymore, I can just forget that. So the specific equation we'll use for the GRU is the following. Which is that the actual value of c<t> will be equal to this gate times the candidate value plus one minus the gate times the old value, c<t> minus one. So you notice that if the gate, if this update value, this equal to one, then it's saying set the new value of c<t> equal to this candidate value. So that's like over here, set gate equal to one so go ahead and update that bit. And then for all of these values in the middle, you should have the gate equals zero. So this is saying don't update it, don't update it, don't update it, just hang onto the old value. Because if gamma u is equal to zero, then this would be zero, and this would be one. And so it's just setting c<t> equal to the old value, even as you scan the sentence from left to right. So when the gate is equal to zero, we're saying don't update it, don't update it, just hang on to the value and don't forget what this value was. And so that way even when you get all the way down here, hopefully you've just been setting c<t> equals c<t> minus one all along. And it still memorizes, the cat was singular. So let me also draw a picture to denote the GRU unit. And by the way, when you look in online blog posts and textbooks and tutorials these types of pictures are quite popular for explaining GRUs as well as we'll see later, LSTM units. I personally find the equations easier to understand in a pictures. So if the picture doesn't make sense. Don't worry about it, but I'll just draw in case helps some of you. So a GRU unit inputs c<t> minus one, for the previous time-step and just happens to be equal to 80 minus one. So take that as input and then it also takes as input x<t>, then these two things get combined together. And with some appropriate weighting and some tanh, this gives you c tilde t which is a candidate for placing c<t>, and then with a different set of parameters and through a sigmoid activation function, this gives you gamma u, which is the update gate. And then finally, all of these things combine together through another operation. And I won't write out the formula, but this box here which wish I shaded in purple represents this equation which we had down there. So that's what this purple operation represents. And it takes as input the gate value, the candidate new value, or there is this gate value again and the old value for c<t>, right. So it takes as input this, this and this and together they generate the new value for the memory cell. And so that's c<t> equals a. And if you wish you could also use this process to soft max or something to make some prediction for y<t>. So that is the GRU unit or at least a slightly simplified version of it. And what is remarkably good at is through the gates deciding that when you're scanning the sentence from left to right say, that's a good time to update one particular memory cell and then to not change, not change it until you get to the point where you really need it to use this memory cell

that is set even earlier in the sentence. And because the sigmoid value, now, because the gate is quite easy to set to zero right. So long as this quantity is a large negative value, then up to numerical around off the uptake gate will be essentially zero. Very, very, very close to zero. So when that's the case, then this updated equation and subsetting c<t> equals c<t> minus one. And so this is very good at maintaining the value for the cell. And because gamma can be so close to zero, can be 0.000001 or even smaller than that, it doesn't suffer from much of a vanishing gradient problem. Because when you say gamma so close to zero this becomes essentially c<t> equals c<t> minus one and the value of c<t> is maintained pretty much exactly even across many many many many time-steps. So this can help significantly with the vanishing gradient problem and therefore allow a neural network to go on even very long range dependencies, such as a cat and was related even if they're separated by a lot of words in the middle. Now I just want to talk over some more details of how you implement this. In the equations I've written, c<t> can be a vector. So if you have 100 dimensional or hidden activation value then c<t> can be a 100 dimensional say. And so c tilde t would also be the same dimension, and gamma would also be the same dimension as the other things on drawing boxes. And in that case, these asterisks are actually element wise multiplication. So here if gamma u, if the gate is 100 dimensional vector, what it is really a 100 dimensional vector of bits, the value is mostly zero and one. That tells you of this 100 dimensional memory cell which are the bits you want to update. And, of course, in practice gamma won't be exactly zero or one. Sometimes it takes values in the middle as well but it is convenient for intuition to think of it as mostly taking on values that are exactly zero, pretty much exactly zero or pretty much exactly one. And what these element wise multiplications do is it just element wise tells the GRU unit which other bits in your- It just tells your GRU which are the dimensions of your memory cell vector to update at every time-step. So you can choose to keep some bits constant while updating other bits. So, for example, maybe you use one bit to remember the singular or plural cat and maybe use some other bits to realize that you're talking about food. And so because you're talk about eating and talk about food, then you'd expect to talk about whether the cat is four letter, right. You can use different bits and change only a subset of the bits every point in time. You now understand the most important ideas of the GRU. What I'm presenting in this slide is actually a slightly simplified GRU unit. Let me describe the full GRU unit. So to do that, let me copy the three main equations. This one, this one and this one to the next slide. So here they are. And for the full GRU unit, I'm sure to make one change to this which is, for the first equation which was calculating the candidate new value for the memory cell, I'm going just to add one term. Let me pushed that a little bit to the right, and I'm going to add one more gate. So this is another gate gamma r. You can think of r as standing for relevance. So this gate gamma r tells you how relevant is c<t> minus one to computing the next candidate for c<t>. And this gate gamma r is computed pretty much as you'd expect with a new parameter matrix Wr, and then the same things as input x<t> plus br. So as you can imagine there are multiple ways to design these types of neural networks. And why do we have gamma r? Why not use a simpler version from the previous slides? So it turns out that over many years researchers have experimented with many, many different possible versions of how to design these units, to try to have longer range connections, to try to have more the longer range effects and also address vanishing gradient problems. And the GRU is one of the most commonly used versions that researchers have converged to and found as robust and useful for many different problems. If you wish you could try to invent new versions of these units if you want, but the GRU is a standard one, that's just common used. Although you can imagine that researchers have tried other versions that are similar but not exactly the same as what I'm writing down here as well. And the other common version is called an LSTM which stands for Long Short Term Memory which we'll talk about in the next video. But GRUs and LSTMs are two specific instantiations of this set of ideas that are most commonly used. Just one note on notation. I tried to define a consistent notation to make these ideas easier to understand. If you look at the academic literature, you sometimes see people- If you look at the academic literature sometimes you see people using alternative notation to be x tilde, u, r and h to refer to these quantities as well. But I try to use a more consistent notation between GRUs and LSTMs as well as using a more consistent notation gamma to refer to the gates, so hopefully make these ideas easier to understand. So that's it for the GRU, for the Gate Recurrent Unit. This is one of the ideas in RNN that has enabled them to become much better at capturing very long range dependencies has made RNN much more effective. Next, as I briefly mentioned, the other most commonly used variation of this class of idea is something called the LSTM unit, Long Short Term Memory unit. Let's take a look at that in the next video.



[Cho et al., 2014. On the properties of neural machine translation: Encoder-decoder approaches]
[Chung et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling]

Full GRU

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

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

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

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

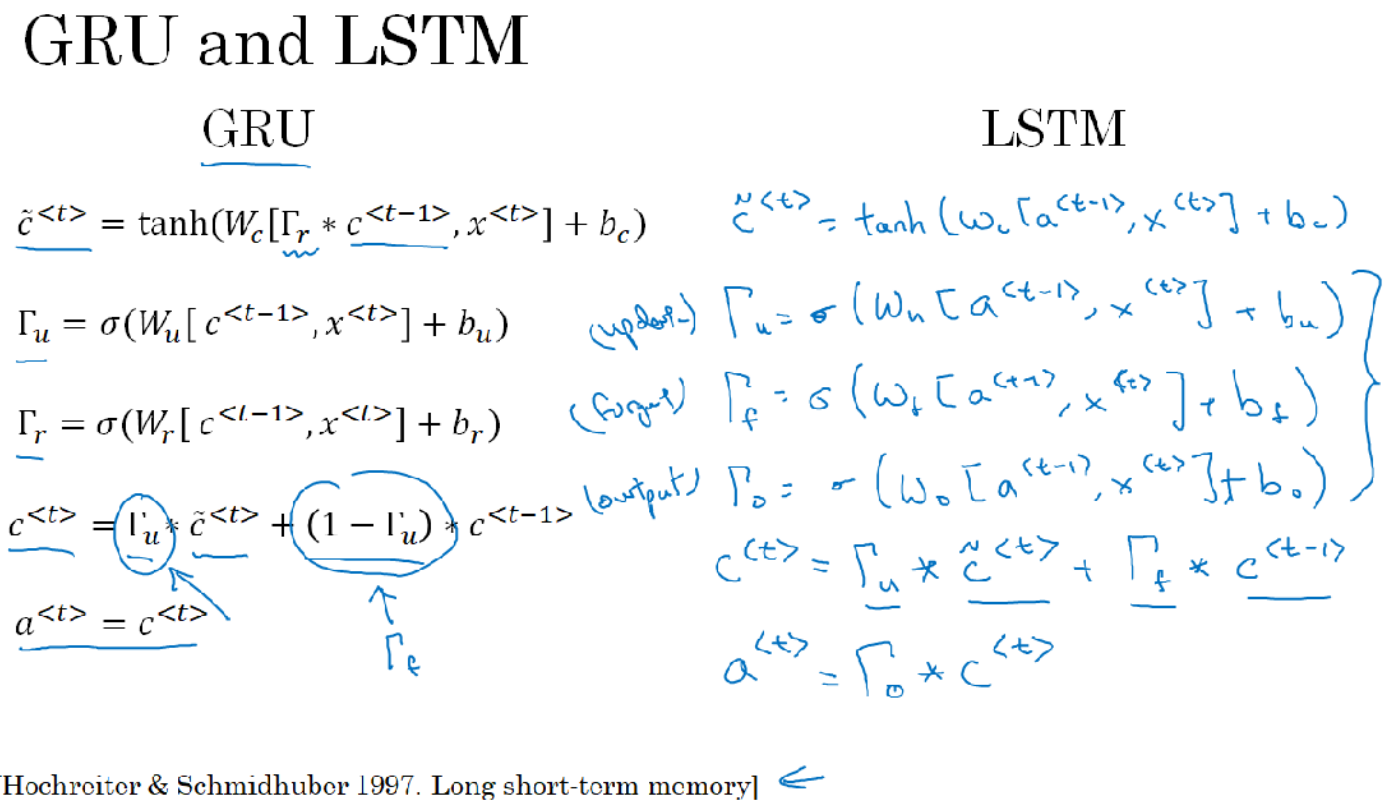
LSTM

The cat, which ate already, was full.

Long Short Term Memory (LSTM)

In the last section, we've learned about the GRU, the **gated recurrent units**, and how that can allow you to learn very long range connections in a sequence. The other type of unit that allows you to do this very well is the LSTM or the **long short term memory units**, and this is even more

powerful than the GRU. Let's take a look. Below are the equations from the previous section for the GRU.



[Hochreiter & Schmidhuber 1997. Long short-term memory]

And for the GRU, we had $a^{<t>}$ equals $c^{<t>}$, and two gates, the optic gate and the relevance gate, $\tilde{c}^{<t>}$, which is a candidate for replacing the memory cell, and then we use the update gate, gamma_u, to decide whether or not to update $c^{<t>}$using $\tilde{c}^{<t>}$. The LSTM is an even slightly more powerful and more general version of the GRU, and is due to Sepp Hochreiter and Jurgen Schmidhuber. And this was a really seminal paper, a huge impact on sequence modelling. I think this paper is one of the more difficult to read. It goes quite along into theory of vanishing gradients. And so, I think more people have learned about the details of LSTM through maybe other places than from this particular paper even though I think this paper has had a wonderful impact on the Deep Learning community. But these are the equations that govern the LSTM. So, the book continued to the memory cell, c, and the candidate value for updating it, c_tilde_t, will be this, and so on. Notice that for the LSTM, we will no longer have the case that a_t is equal to c_t. So, this is what we use. And so, this is just like the equation on the left except that with now, more specially use a_t there or a_t minus one instead of c_t minus one. And we're not using this gamma or this relevance gate. Although you could have a variation of the LSTM where you put that back in, but with the more common version of the LSTM, doesn't bother with that. And then we will have an update gate, same as before. So, W updates and we're going to use a_t minus one here, x_t plus b_u. And one new property of the LSTM is, instead of having one update gate control, both of these terms, we're going to have two separate terms. So instead of gamma_u and one minus gamma_u, we're going have gamma_u here. And forget gate, which we're going to call gamma_f. So, this gate, gamma_f, is going to be sigmoid of pretty much what you'd expect, x_t plus b_f. And then, we're going to have a new output gate which is sigma of W_o. And then again, pretty much what you'd expect, plus b_o. And then, the update value to the memory so will be c_t equals gamma u. And this asterisk denotes element-wise multiplication. This is a vector-vector element-wise multiplication, plus, and instead of one minus gamma u, we're going to have a separate forget gate, gamma_f, times c of t minus one. So this gives the memory cell the option of keeping the old value c_t minus one and then just adding to it, this new value, c tilde of t. So, use a separate update and forget gates. So, this stands for update, forget, and output gate. And then finally, instead of a_t equals c_t a_t is a_t equal to the output gate element-wise multiplied by c_t. So, these are the equations that govern the LSTM and you can tell it has three gates instead of two. So, it's a bit more complicated and it places the gates into slightly different places. So, here again are the equations governing the behavior of the LSTM. Once again, it's traditional to explain

these things using pictures. So let me draw one here. And if these pictures are too complicated, don't worry about it. I personally find the equations easier to understand than the picture. But I'll just show the picture here for the intuitions it conveys. The bigger picture here was very much inspired by a blog post due to Chris Ola, title, Understanding LSTM Network, and the diagram drawing here is quite similar to one that he drew in his blog post. But the key thing is to take away from this picture are maybe that you use a_t minus one and x_t to compute all the gate values.

LSTM units

GRU

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

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

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

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

$$a^{<t>} = c^{<t>}$$

LSTM

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

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

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

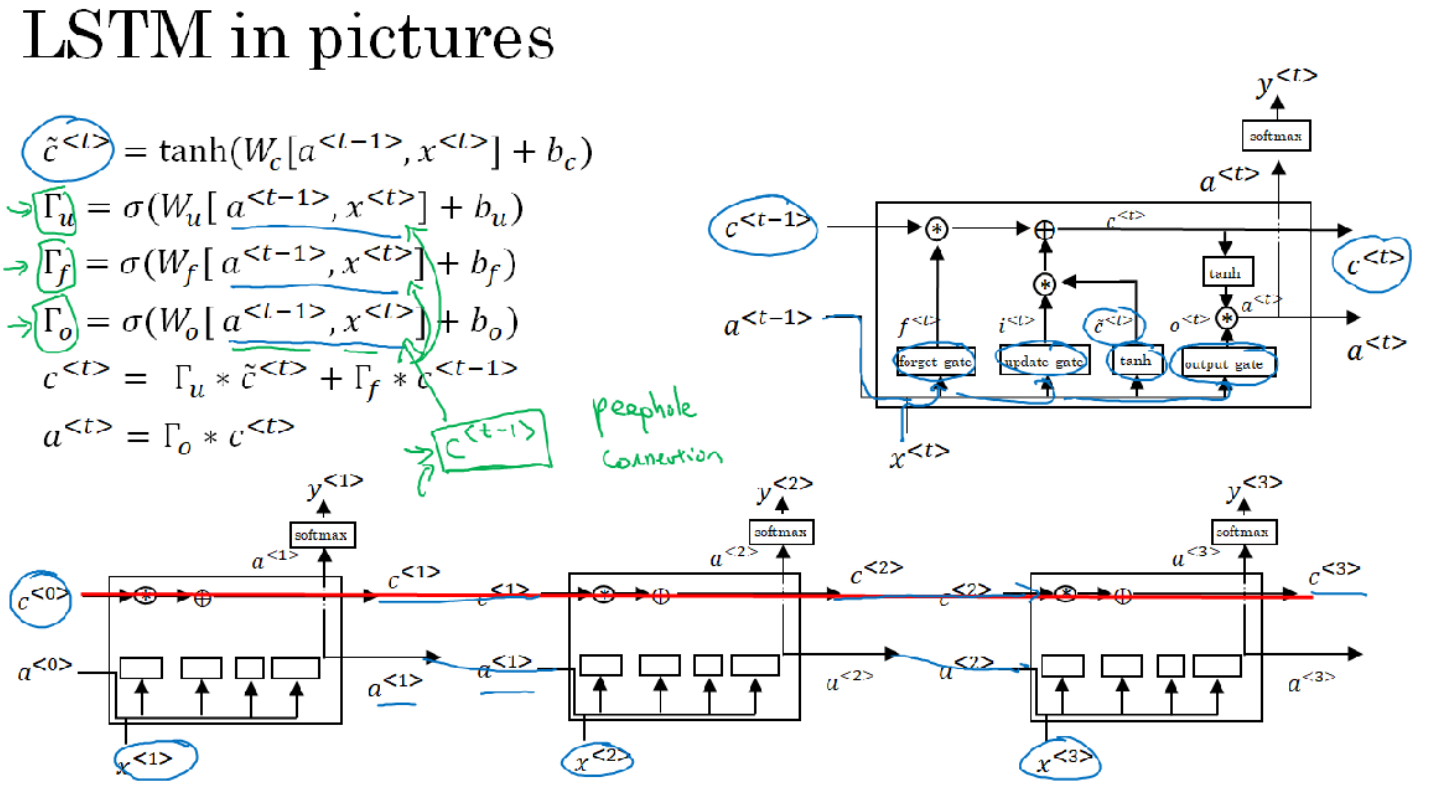
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * c^{<t>}$$

[Hochreiter & Schmidhuber 1997. Long short-term memory]

In this picture, you have a_t minus one, x_t coming together to compute the forget gate, to compute the update gates, and to compute output gate. And they also go through a tanh to compute c_tilde_of_t. And then these values are combined in these complicated ways with element-wise multiplies and so on, to get c_t from the previous c_t minus one. Now, one element of this is interesting as you have a bunch of these in parallel. So, that's one of them and you connect them. You then connect these temporally. So it does the input x_1 then x_2, x_3. So, you can take these units and just hold them up as follows, where the output a at the previous timestep is the input a at the next timestep, the c. I've simplified to diagrams a little bit in the bottom. And one cool thing about this you'll notice is that there's this line at the top that shows how, so long as you set the forget and the update gate appropriately, it is relatively easy for the LSTM to have some value c_0 and have that be passed all the way to the right to have your, maybe, c_3 equals c_0. And this is why the LSTM, as well as the GRU, is very good at memorizing certain values even for a long time, for certain real values stored in the memory cell even for many, many timesteps. So, that's it for the LSTM. As you can imagine, there are also a few variations on this that people use. Perhaps, the most common one is that instead of just having the gate values be dependent only on a_t minus one, x_t, sometimes, people also sneak in there the values c_t minus one as well. This is called a peephole connection. Not a great name maybe but you'll see, peephole connection. What that means is that the gate values may depend not just on a_t minus one and on x_t, but also on the previous memory cell value, and the peephole connection can go into all three of these gates' computations. So that's one common variation you see of LSTMs. One technical detail is that these are, say, 100-dimensional vectors. So if you have a 100-dimensional hidden memory cell unit, and so is this. And the, say, fifth element of c_t minus one affects only the fifth element of the corresponding gates, so that relationship is one-to-one, where not every element of the 100-dimensional c_t minus one can affect all elements of the case. But instead, the first element of c_t minus one affects the first element of the case, second element affects the second element, and so on. But if you ever read the paper and see someone talk about the peephole connection, that's when they mean that c_t minus one is used

to affect the gate value as well. So, that's it for the LSTM. When should you use a GRU? And when should you use an LSTM? There isn't widespread consensus in this. And even though I presented GRUs first, in the history of deep learning, LSTMs actually came much earlier, and then GRUs were relatively recent invention that were maybe derived as Pavia's simplification of the more complicated LSTM model. Researchers have tried both of these models on many different problems, and on different problems, different algorithms will win out. So, there isn't a universally-superior algorithm which is why I want to show you both of them. But I feel like when I am using these, the advantage of the GRU is that it's a simpler model and so it is actually easier to build a much bigger network, it only has two gates, so computationally, it runs a bit faster. So, it scales the building somewhat bigger models but the LSTM is more powerful and more effective since it has three gates instead of two. If you want to pick one to use, I think LSTM has been the historically more proven choice. So, if you had to pick one, I think most people today will still use the LSTM as the default first thing to try. Although, I think in the last few years, GRUs had been gaining a lot of momentum and I feel like more and more teams are also using GRUs because they're a bit simpler but often work just as well. It might be easier to scale them to even bigger problems. So, that's it for LSTMs. Well, either GRUs or LSTMs, you'll be able to build neural network that can capture a much longer range depends.



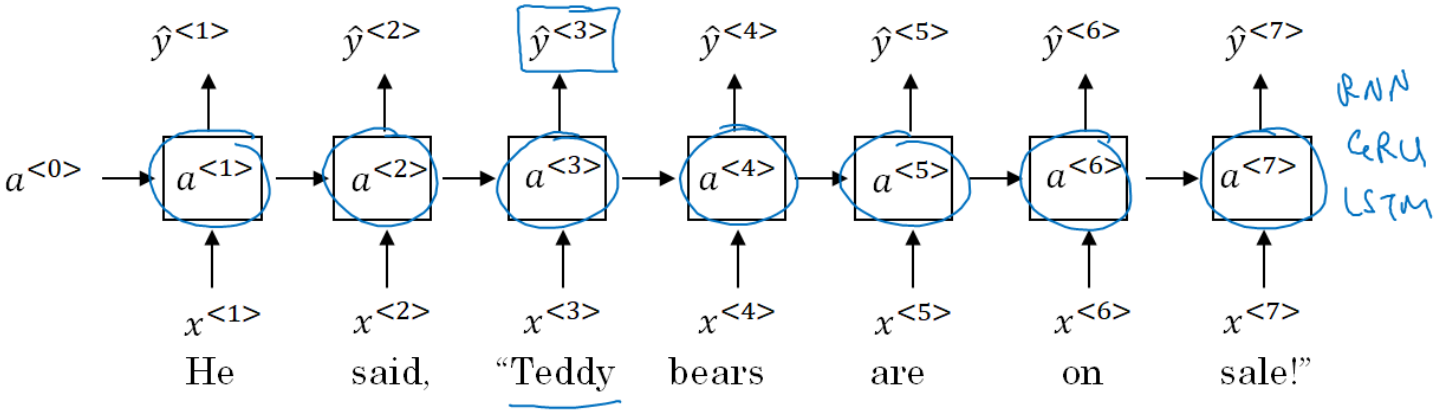
Bidirectional RNN

By now, we've seen most of the key building blocks of RNNs. But, there are just two more ideas that let you build much more powerful models. One is **bidirectional RNNs**, which lets you at a point in time to take information from both earlier and later in the sequence, so we'll talk about that in this section and second, is **deep RNNs**, which we'll see in the next section. So let's start with Bidirectional RNNs. So, to motivate bidirectional RNNs, let's look at this network which you've seen a few times before in the context of named entity recognition and one of the problems of this network is that, to figure out whether the third word Teddy is a part of the person's name, it's not enough to just look at the first part of the sentence.

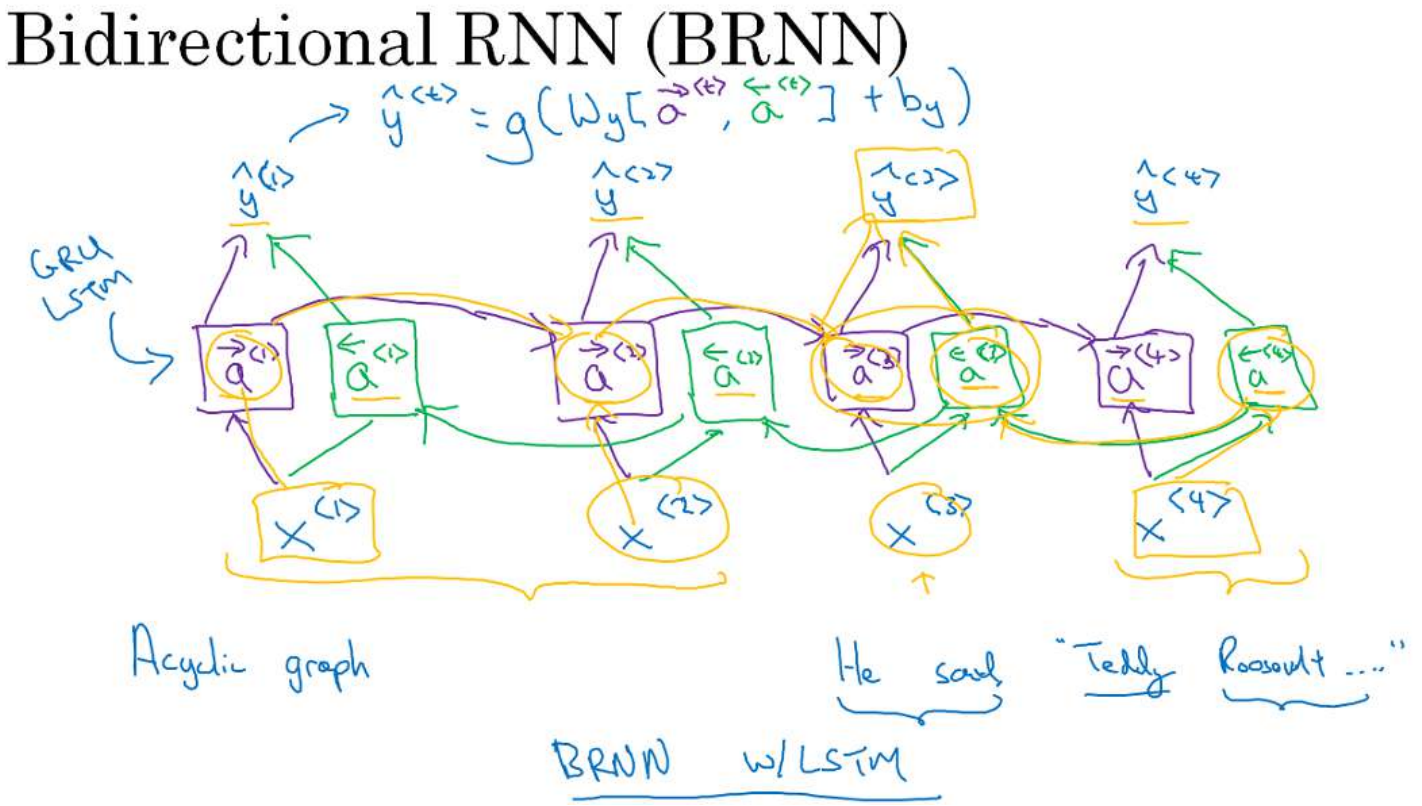
Getting information from the future

He said, "Teddy bears are on sale!"

He said, "Teddy Roosevelt was a great President!"



So to tell, if yhat$^{<3>}$ should be zero or one, you need more information than just the first three words because the first three words doesn't tell you if they'll talking about Teddy bears or talk about the former US president, Teddy Roosevelt. So this is a unidirectional or forward directional only RNN and, this comment I just made is true, whether these cells are standard RNN blocks or whether they're GRU units or whether they're LSTM blocks. But all of these blocks are in a forward only direction. So what a bidirectional RNN does or BRNN, is fix this issue. So, a bidirectional RNN works as follows. I'm going to use a simplified four inputs or maybe a four word sentence. So we have four inputs. x$^{<1>}$ through x$^{<4>}$.



So this networks heading there will have a forward recurrent components. So we're going to call this, a$^{<1>}$, a$^{<2>}$, a$^{<3>}$ and a$^{<4>}$ and we're going to draw a right arrow over that to denote this is the forward recurrent component, and so they'll be connected as shown in above diagram and so, each of

these four recurrent units inputs the current x, and then feeds in to help predict $\hat{y}^{<1>}$, $\hat{y}^{<2>}$, $\hat{y}^{<3>}$, and $\hat{y}^{<4>}$. So far we've drawn the RNN from the previous section, but with the arrows placed in slightly funny positions. But as we draw the arrows in this slightly funny positions because what we're going to do is add a backward recurrent layer. So we'd have $a^{<1>}$, left arrow to denote this is a backward connection, and then $a^{<2>}$ backwards, $a^{<3>}$ backwards and $a^{<4>}$ backwards, so the left arrow denotes that it is a backward connection and so, we're then going to connect to network up as follows and A backward connections will be connected to each other going backward in time. So, notice that this network defines a **Acyclic graph** and so, given an input sequence, $x^{<1>}$ through $x^{<4>}$, the fourth sequence will first compute A forward one, then use that to compute A forward two, then A forward three, then A forward four. Whereas, the backward sequence would start by computing A backward four, and then go back and compute A backward three, and then as you are computing network activation, this is not backward this is forward prop. But the forward prop has part of the computation going from left to right and part of computation going from right to left in this diagram. But having computed A backward three, you can then use those activations to compute A backward two, and then A backward one, and then finally having computed all you had in the activations, you can then make your predictions.

So, for example, to make the predictions, your network will have something like yhat at time t is an activation function applied to WY with both the forward activation at time t, and the backward activation at time t being fed in to make that prediction at time t. So, if you look at the prediction at time set three for example, then information from $x^{<1>}$ can flow through here, forward one to forward two, they're are all stated in the function here, to forward three to $\hat{y}^{<3>}$. So information from $x^{<1>}$, $x^{<2>}$, $x^{<3>}$ are all taken into account with information from $x^{<4>}$ can flow through a backward four to a backward three to $y^{<3>}$. So this allows the prediction at time three to take as input both information from the past, as well as information from the present which goes into both the forward and the backward things at this step, as well as information from the future.

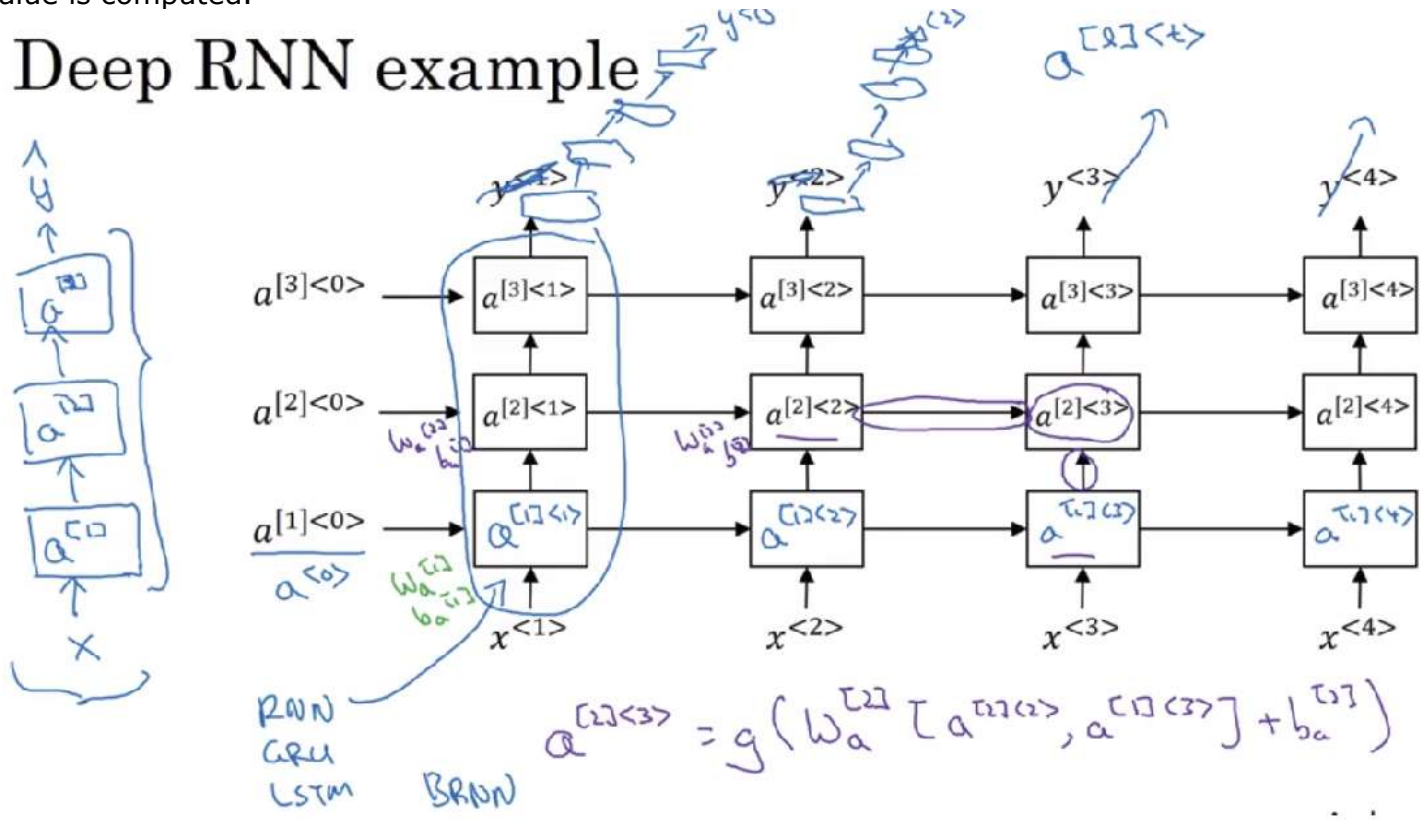
So, in particular, given a phrase like, "He said, Teddy Roosevelt..." To predict whether Teddy is a part of the person's name, you take into account information from the past and from the future. So this is the bidirectional recurrent neural network and these blocks here can be not just the standard RNN block but they can also be GRU blocks or LSTM blocks. In fact, for a lots of NLP problems, for a lot of text with natural language processing problems, **a bidirectional RNN with a LSTM appears to be commonly used. So, we have NLP problem and you have the complete sentence, you try to label things in the sentence, a bidirectional RNN with LSTM blocks both forward and backward would be a pretty views of first thing to try.**

So, that's it for the bidirectional RNN and this is a modification they can make to the basic RNN architecture or the GRU or the LSTM, and by making this change you can have a model that uses RNN and or GRU or LSTM and is able to make predictions anywhere even in the middle of a sequence by taking into account information potentially from the entire sequence. **The disadvantage of the bidirectional RNN is that you do need the entire sequence of data before you can make predictions anywhere.** So, for example, if you're building a speech recognition system, then the BRNN will let you take into account the entire speech utterance but if you use this straightforward implementation, you need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction. So for a real type speech recognition applications, they're somewhat more complex modules as well rather than just using the standard bidirectional RNN as you've seen here. **But for a lot of natural language processing applications where you can get the entire sentence all the same time, the standard BRNN algorithm is actually very effective**. So, that's it for BRNNs and next and final SECTION for this week, let's talk about how to take all of these ideas RNNs, LSTMs and GRUs and the bidirectional versions and construct deep versions of them.

Deep RNNs

The different versions of RNNs you've seen so far will already work quite well by themselves. But for learning very complex functions sometimes is useful to stack multiple layers of RNNs together to build even deeper versions of these models. In this section, we'll see how to build these deeper RNNs. Let's take a look. So we remember for a standard neural network, you will have an input x and then that's stacked to some hidden layer and so that might have activations of say, a1 for the first

hidden layer, and then that's stacked to the next layer with activations a2, then maybe another layer, activations a3 and then you make a prediction ŷ. So a deep RNN is a bit like this, by taking this network that (shown in diagram) and unrolling that in time. So let's take a look. So here's the standard RNN that you've seen so far. But we've changed the notation a little bit which is that, instead of writing this as a0 for the activation time zero, I've added this square bracket 1 to denote that this is for layer one. So the notation we're going to use is a[l] to denote that it's an activation associated with layer l and then <t> to denote that that's associated over time t. So this will have an activation on $a^{[1]<1>}$, $a^{[1]<2>}$, $a^{[1]<3>}$, $a^{[1]<4>}$ and then we can just stack these things on top and so this will be a new network with three hidden layers. So let's look at an example of how this value is computed.



So $a^{[2]<3>}$ has two inputs. It has the input coming from the bottom, and there's the input coming from the left. So the computer has an activation function g applied to a way matrix. Check the equation at the end of diagram which calculates the activation value and so the same parameters $Wa^{[2]}$ and $ba^{[2]}$ are used for every one of these computations at this layer. Whereas, in contrast, the first layer would have its own parameters $Wa^{[1]}$ and $ba^{[1]}$. So whereas for standard RNNs like the one on the left, you know we've seen neural networks that are very, very deep, maybe over 100 layers. For RNNs, having three layers is already quite a lot. Because of the temporal dimension, these networks can already get quite big even if you have just a small handful of layers and you don't usually see these stacked up to be like 100 layers. One thing you do see sometimes is that you have recurrent layers that are stacked on top of each other. But then you might take the output here, let's get rid of this, and then just have a bunch of deep layers that are not connected horizontally but have a deep network here that then finally predicts $y^{<1>}$ and you can have the same deep network here that predicts $y^{<2>}$. So this is a type of network architecture that we're seeing a little bit more where you have three recurrent units that connected in time, followed by a network, followed by a network after that, as we seen for $y^{<3>}$ and $y^{<4>}$, of course. There's a deep network, but that does not have the horizontal connections. So that's one type of architecture we seem to be seeing more of and quite often, these blocks don't just have to be standard RNN, the simple RNN model. They can also be GRU blocks LSTM blocks and finally, you can also build deep versions of the bidirectional RNN. Because deep RNNs are quite computationally expensive to train, there's often a large temporal extent already, though you just don't see as many deep recurrent layers. In this case three deep recurrent layers that are connected in time. You don't see as many deep recurrent layers as you would see in a number of layers in a deep conventional neural network. So that's it for deep RNNs. With what you've

seen this week, ranging from the basic RNN, the basic recurrent unit, to the GRU, to the LSTM, to the bidirectional RNN, to the deep versions of this that you just saw, you now have a very rich toolbox for constructing very powerful models for learning sequence models.

Week 2: Natural Language Processing & Word Embeddings

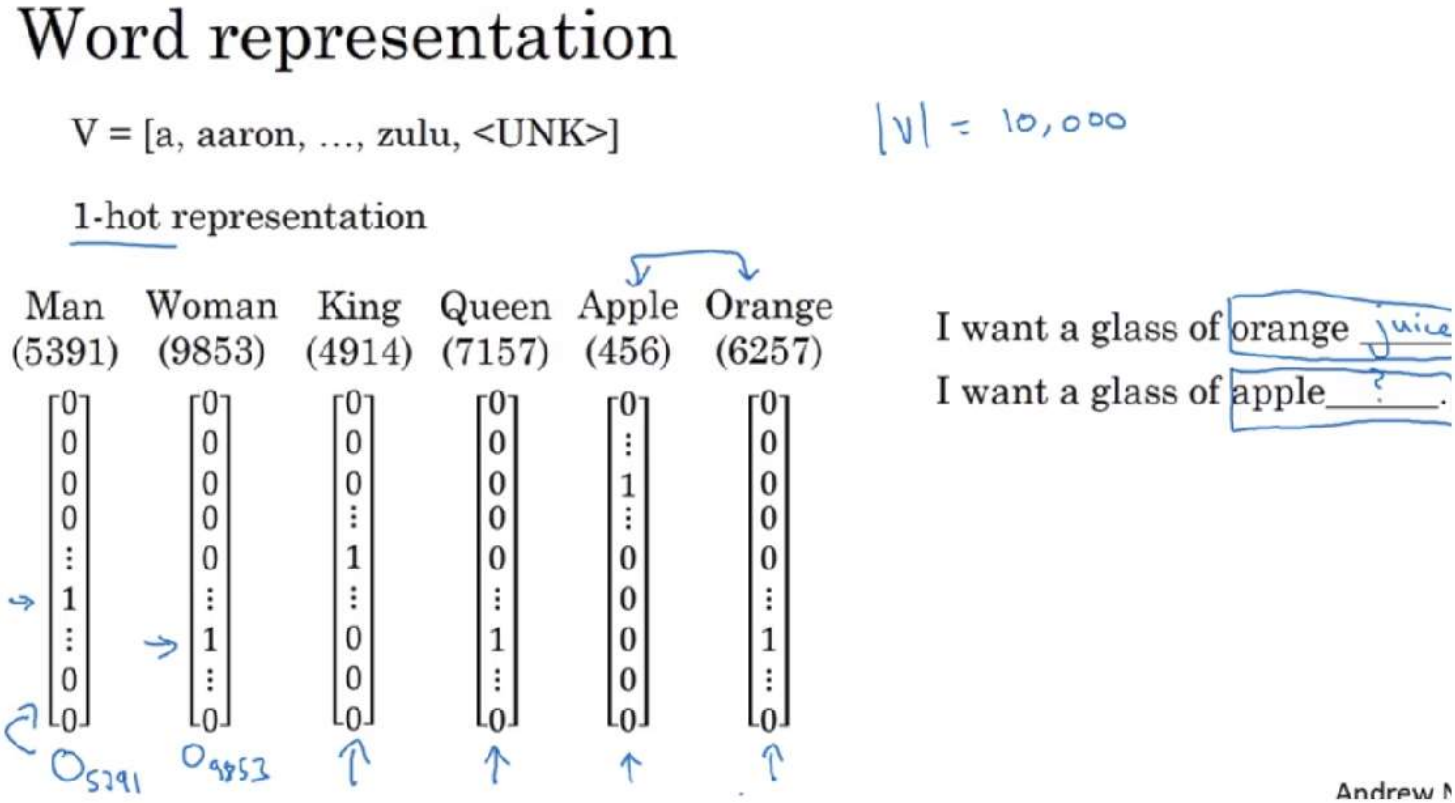
Natural language processing with deep learning is an important combination. Using word vector representations and embedding layers you can train recurrent neural networks with outstanding performances in a wide variety of industries. Examples of applications are sentiment analysis, named entity recognition and machine translation.

Introduction to Word Embeddings

Word Representation

In the last week, we learned about RNNs, GRUs, and LSTMs. In this week, we'll discuss how many of these ideas can be applied to NLP or Natural Language Processing, which is one of the features of AI because it's really being revolutionized by deep learning. One of the key ideas you learn about is word embeddings, which is a way of representing words. The less your algorithms automatically understand analogies like that, man is to woman, as king is to queen, and many other examples and through these ideas of word embeddings, we'll be able to build NLP applications, even can model with usually of relatively small label training sets. Finally towards the end of the week, we'll see how to debias word embeddings. That's to reduce undesirable gender or ethnicity or other types of bias that learning algorithms can sometimes pick up. So with that, let's get started with a discussion on word representation.

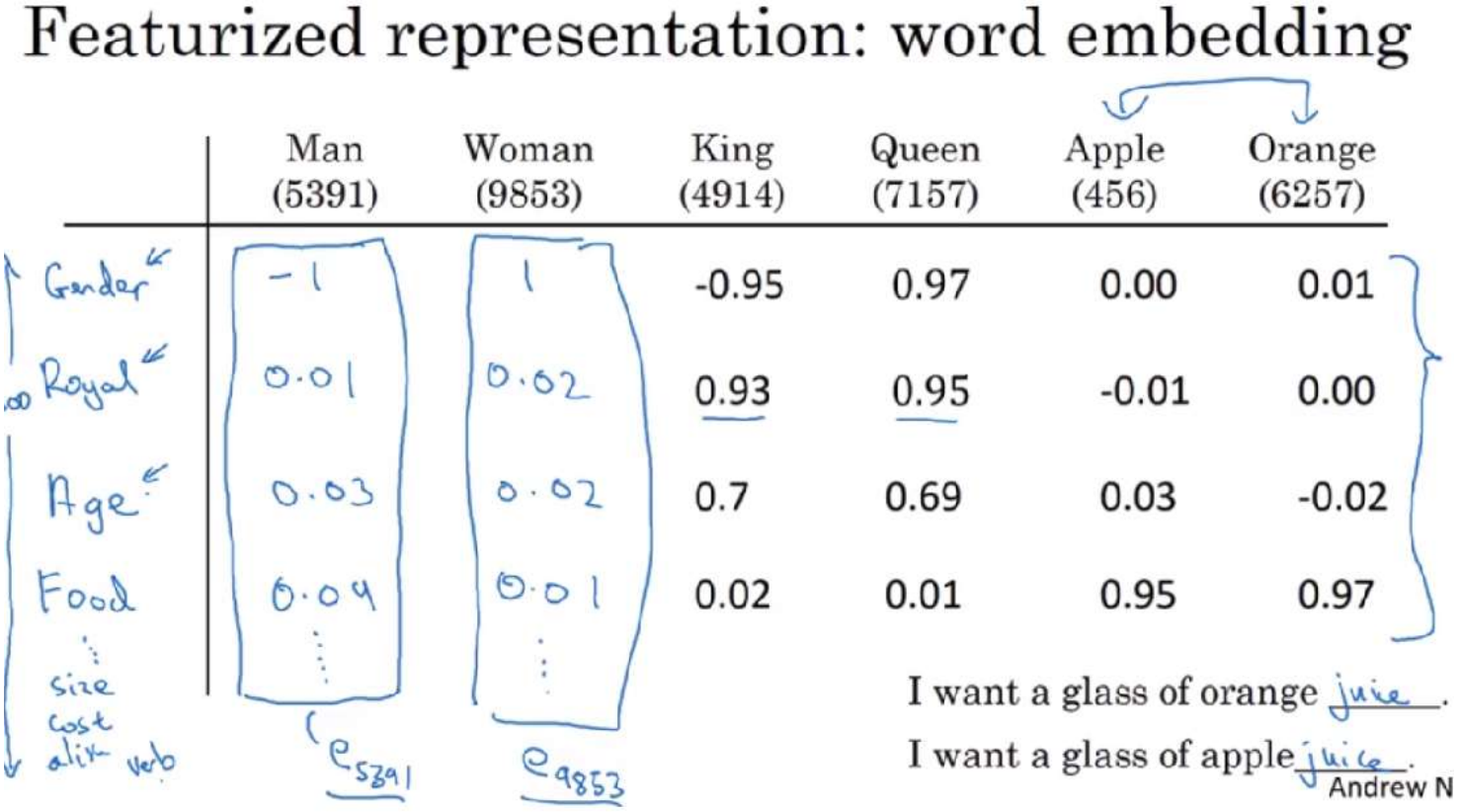
So far, we've been representing words using a vocabulary of words, and a vocabulary from the previous week might be say, 10,000 words. And we've been representing words using a one-hot vector. So for example,



if man is word number 5391 in this dictionary, then you represent him with a vector with one in position 5391. And I'm also going to use O subscript 5391 to represent this factor, where O here stands for one-hot. And then, if woman is word number 9853, then you represent it with O subscript 9853 which just has a one in position 9853 and zeros elsewhere. And then other words king, queen, apple, orange will be similarly represented with one-hot vector. One of the weaknesses of this representation is that it treats each word as a thing unto itself, and it doesn't allow an algorithm to easily generalize the cross words. For example, let's say you have a language model that has learned

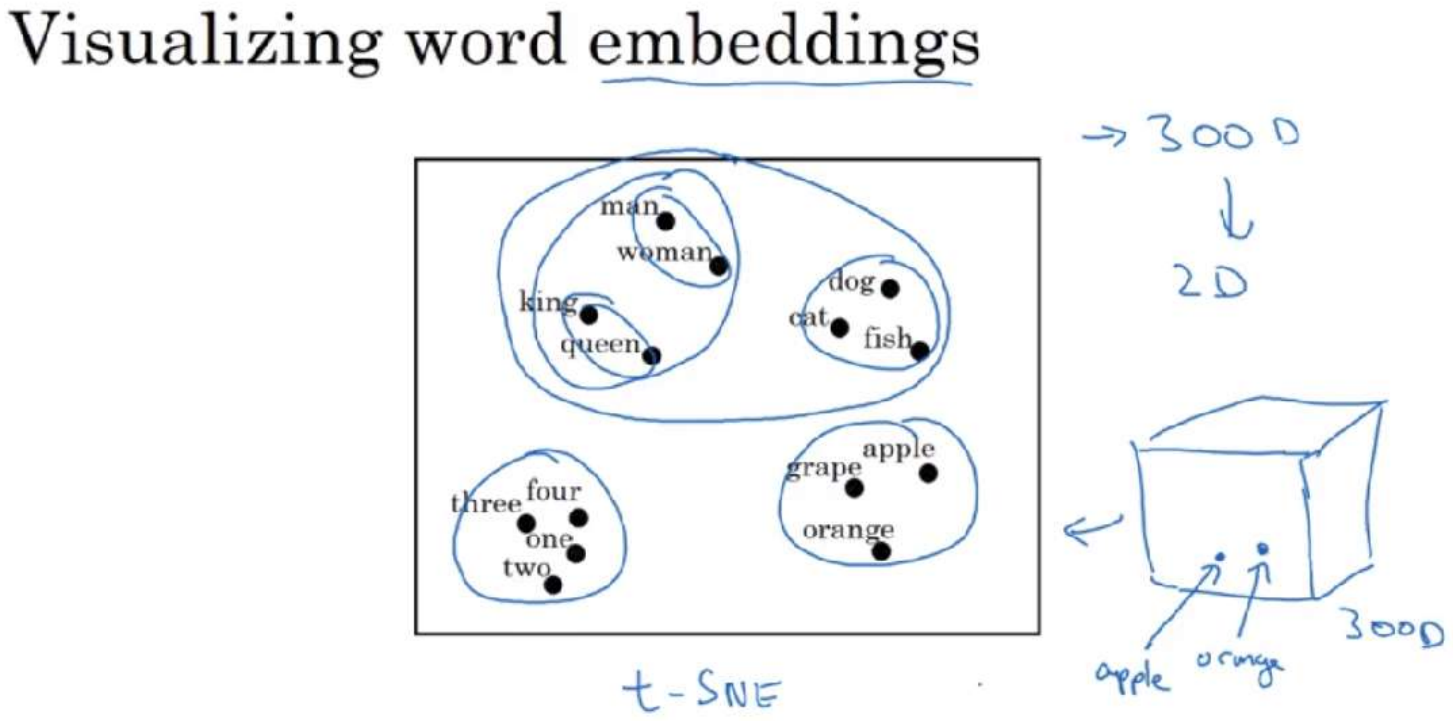
that when you see I want a glass of orange blank. Well, what do you think the next word will be? Very likely, it'll be juice. But even if the learning algorithm has learned that I want a glass of orange juice is a likely sentence, if it sees I want a glass of apple blank. As far as it knows the relationship between apple and orange is not any closer as the relationship between any of the other words man, woman, king, queen, and orange. And so, it's not easy for the learning algorithm to generalize from knowing that orange juice is a popular thing, to recognizing that apple juice might also be a popular thing or a popular phrase. And this is because the any product between any two different one-hot vector is zero. If you take any two vectors say, queen and king and any product of them, the end product is zero. If you take apple and orange and any product of them, the end product is zero. And you couldn't distance between any pair of these vectors is also the same. So it just doesn't know that somehow apple and orange are much more similar than king and orange or queen and orange.

So, won't it be nice if instead of a one-hot presentation we can instead learn a featured representation with each of these words, a man, woman, king, queen, apple, orange or really for every word in the dictionary, we could learn a set of features and values for each of them.



So for example, each of these words, we want to know what is the gender associated with each of these things. So, if gender goes from minus one for male to plus one for female, then the gender associated with man might be minus one, for woman might be plus one and then eventually, learning these things maybe for king you get minus 0.95, for queen plus 0.97, and for apple and orange sort of genderless. Another feature might be, well how royal are these things and so the terms, man and woman are not really royal, so they might have feature values close to zero. Whereas king and queen are highly royal. And apple and orange are not really loyal. How about age? Well, man and woman doesn't connotes much about age. Maybe men and woman implies that they're adults, but maybe neither necessarily young nor old. So maybe values close to zero. Whereas kings and queens are always almost always adults and apple and orange might be more neutral with respect to age and then, another feature for here, is this is a food? Well, man is not a food, woman is not a food, neither are kings and queens, but apples and oranges are foods. And they can be many other features as well ranging from, what is the size of this? What is the cost? Is this something that is a live? Is this an action, or is this a noun, or is this a verb, or is it something else? And so on. So you can imagine coming up with many features. And for the sake of the illustration let's say, 300 different features, and what that does is, it allows you to take this list of numbers, I've only written four here, but this could be a list of 300 numbers, that then becomes a 300 dimensional vector for representing the word man. And I'm going to use the notation e_{5391} to denote a representation like this. And

similarly, this vector, this 300 dimensional vector or 300 dimensional vector like this, I would denote e9853 to denote a 300 dimensional vector we could use to represent the word woman and similarly, for the other examples here. Now, if you use this representation to represent the words orange and apple, then notice that the representations for orange and apple are now quite similar. Some of the features will differ because of the color of an orange, the color an apple, the taste, or some of the features would differ. But by a large, a lot of the features of apple and orange are actually the same, or take on very similar values and so, this increases the odds of the learning algorithm that has figured out that orange juice is a thing, to also quickly figure out that apple juice is a thing. So this allows it to generalize better across different words. So over the next few videos, we'll find a way to learn words embeddings. We just need you to learn high dimensional feature vectors like these, that gives a better representation than one-hot vectors for representing different words. And the features we'll end up learning, won't have a easy to interpret interpretation like that component one is gender, component two is royal, component three is age and so on. Exactly what they're representing will be a bit harder to figure out. But nonetheless, the featurized representations we will learn, will allow an algorithm to quickly figure out that apple and orange are more similar than say, king and orange or queen and orange. If we're able to learn a 300 dimensional feature vector or 300 dimensional embedding for each words, one of the popular things to do is also to take this 300 dimensional data and embed it say, in a two dimensional space so that you can visualize them. And so, one common algorithm for doing this is the **t-SNE algorithm** due to Laurens van der Maaten and Geoff Hinton

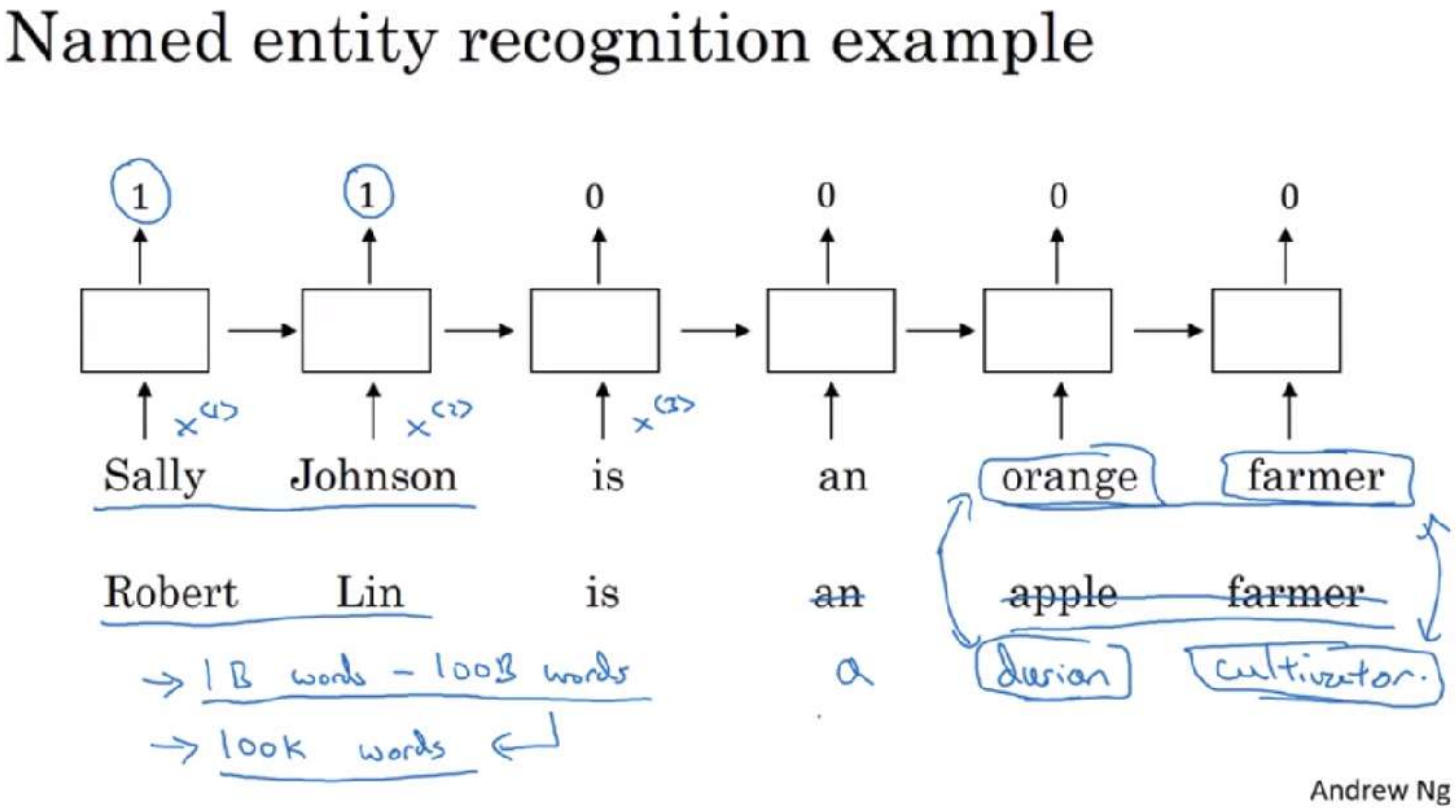


and if you look at one of these embeddings, one of these representations, you find that words like man and woman tend to get grouped together, king and queen tend to get grouped together, and these are the people which tends to get grouped together. Those are animals who can get grouped together. Fruits will tend to be close to each other. Numbers like one, two, three, four, will be close to each other. And then, maybe the animate objects as whole will also tend to be grouped together. But you see plots like these sometimes on the internet to visualize some of these 300 or higher dimensional embeddings and maybe this gives you a sense that, word embeddings algorithms like this can learn similar features for concepts that feel like they should be more related, as visualized by that concept that seem to you and me like they should be more similar, end up getting mapped to a more similar feature vectors and these representations will use these sort of featurized representations in maybe a 300 dimensional space, **these are called embeddings and the reason we call them embeddings is, you can think of a 300 dimensional space and again, they can't draw out here in two dimensional space because it's a 3D one and what you do is you take every words like orange, and have a three dimensional feature vector so that word orange gets embedded to a point in this 300 dimensional space and the word apple, gets**

embedded to a different point in this 300 dimensional space and of course to visualize it, algorithms like **t-SNE**, map this to a much lower dimensional space, you can actually plot the 2D data and look at it. But that's what the term embedding comes from. Word embeddings has been one of the most important ideas in NLP, in Natural Language Processing. In this section, we saw why you might want to learn or use word embeddings. In the next section, let's take a deeper look at how you'll be able to use these algorithms, to build NLP algorithims.

Using word embeddings

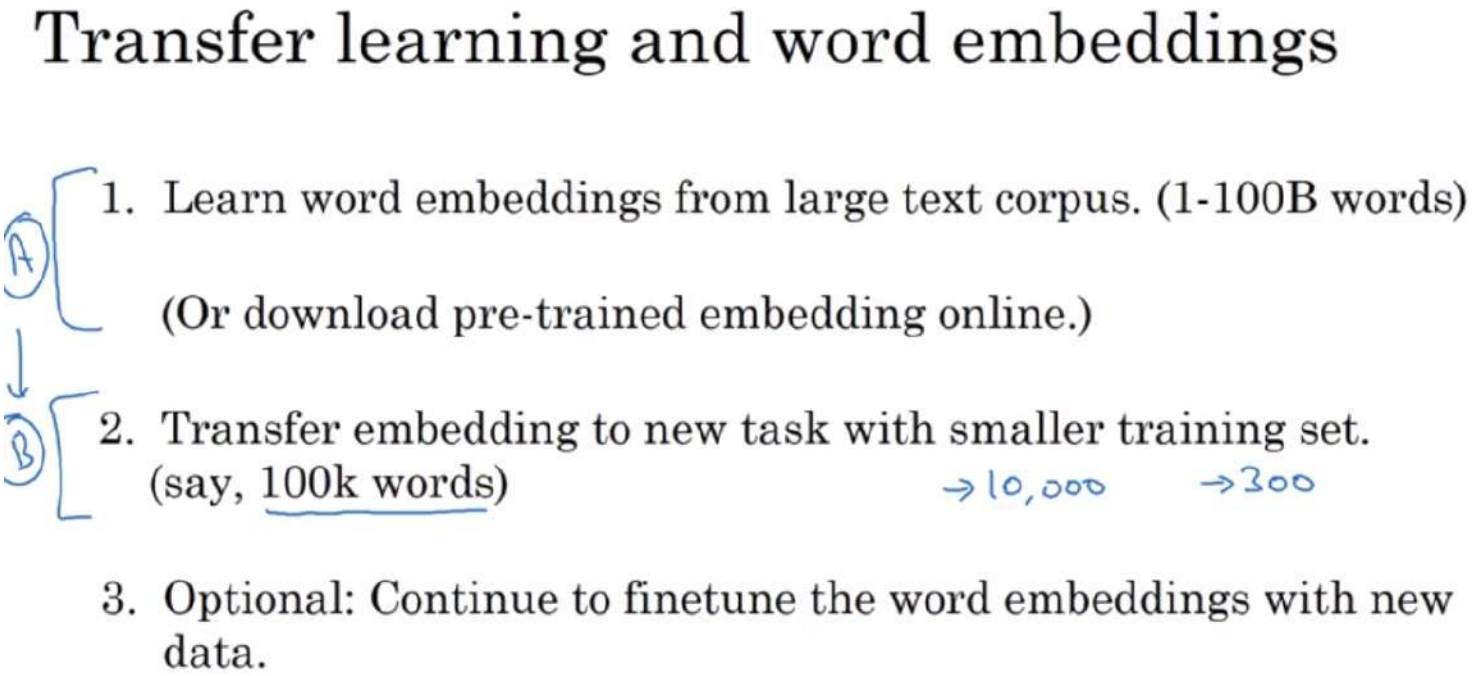
In the last section, we saw what it might mean to learn a featurized representations of different words. In this section, we'll see how we can take these representations and plug them into NLP applications. Let's start with an example.



Continuing with the named entity recognition example, if you're trying to detect people's names. Given a sentence like Sally Johnson is an orange farmer, hopefully, you'll figure out that Sally Johnson is a person's name, hence, the outputs 1 like that. And one way to be sure that Sally Johnson has to be a person, rather than say the name of the corporation is that you know orange farmer is a person. So previously, we had talked about one hot representations to represent these words, x(1), x(2), and so on. But if you can now use the featurized representations, the embedding vectors that we talked about in the last video. Then after having trained a model that uses word embeddings as the inputs, if you now see a new input, Robert Lin is an apple farmer. Knowing that orange and apple are very similar will make it easier for your learning algorithm to generalize to figure out that Robert Lin is also a human, is also a person's name. One of the most interesting cases will be, what if in your test set you see not Robert Lin is an apple farmer, but you see much less common words? What if you see Robert Lin is a durian cultivator? A durian is a rare type of fruit, popular in Singapore and a few other countries. But if you have a small label training set for the **named entity recognition** task, you might not even have seen the word durian or seen the word cultivator in your training set. I guess technically, this should be a durian cultivator. But if you have learned a word embedding that tells you that durian is a fruit, so it's like an orange, and a cultivator, someone that cultivates is like a farmer, then you might still be generalize from having seen an orange farmer in your training set to knowing that a durian cultivator is also probably a person. So one of the reasons that word embeddings will be able to do this is the algorithms to learning word embeddings can examine very large text corpuses, maybe found off the Internet. So you can examine very large data sets, maybe a billion words, maybe even up to 100 billion words would be quite reasonable. So very large training sets of just

unlabeled text and by examining tons of unlabeled text, which you can download more or less for free, you can figure out that orange and durian are similar and farmer and cultivator are similar, and therefore, learn embeddings, that groups them together. Now having discovered that orange and durian are both fruits by reading massive amounts of Internet text, what you can do is then take this word embedding and apply it to your named entity recognition task, for which you might have a much smaller training set, maybe just 100,000 words in your training set, or even much smaller and so this allows you to carry out transfer learning, where you take information you've learned from huge amounts of unlabeled text that you can suck down essentially for free off the Internet to figure out that orange, apple, and durian are fruits and then transfer that knowledge to a task, such as named entity recognition, for which you may have a relatively small labeled training set and, of course, for simplicity, l drew this for it only as a unidirectional RNN. If you actually want to carry out the named entity recognition task, you should, of course, use a bidirectional RNN rather than a simpler one I've drawn here.

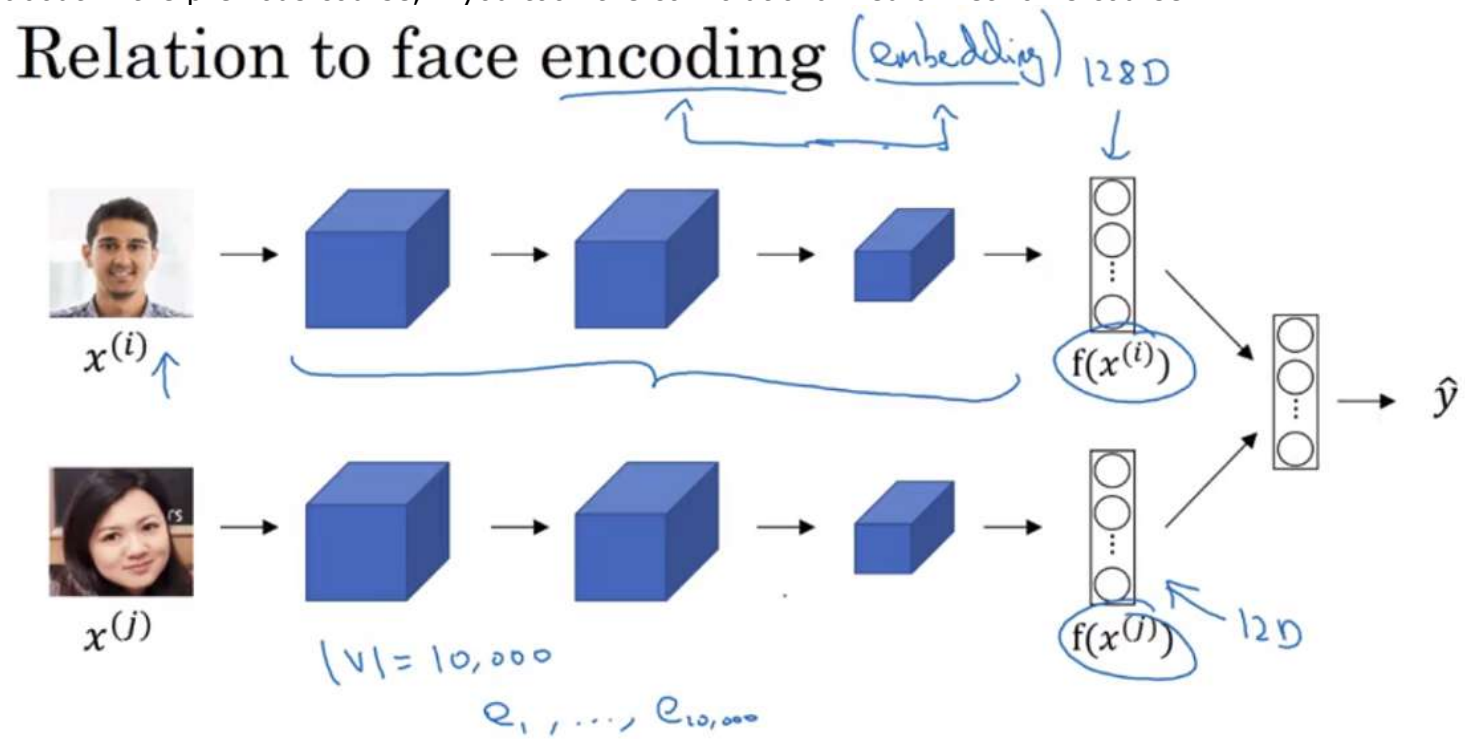
But to summarize, this is how you can carry out transfer learning using word embeddings.



Step 1 is to learn word embeddings from a large text corpus, a very large text corpus or you can also download pre-trained word embeddings online. There are several word embeddings that you can find online under very permissive licenses and you can then take these word embeddings and transfer the embedding to new task, where you have a much smaller labeled training sets and use this, let's say, 300 dimensional embedding, to represent your words. One nice thing also about this is you can now use relatively lower dimensional feature vectors. So rather than using a 10,000 dimensional one-hot vector, you can now instead use maybe a 300 dimensional dense vector. Although the one-hot vector is fast and the 300 dimensional vector that you might learn for your embedding will be a dense vector and then, finally, as you train your model on your new task, on your named entity recognition task with a smaller label data set, one thing you can optionally do is to continue to fine tune, continue to adjust the word embeddings with the new data. In practice, you would do this only if this task 2 has a pretty big data set. If your label data set for step 2 is quite small, then usually, I would not bother to continue to fine tune the word embeddings. So word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set. So it has been useful for many NLP tasks and I'll just name a few. It has been useful for named entity recognition, for text summarization, for co-reference resolution, for parsing. These are all maybe pretty standard NLP tasks. It has been less useful for language modeling, machine translation, especially if you're accessing a language modeling or machine translation task for which you have a lot of data just dedicated to that task. So as seen in other transfer learning settings, if you're trying to transfer from some task A to some task B, the process of transfer learning is just most useful when you happen to

have a ton of data for A and a relatively smaller data set for B. And so that's true for a lot of NLP tasks, and just less true for some language modeling and machine translation settings.

Finally, word embeddings has a interesting relationship to the face encoding ideas that you learned about in the previous course, if you took the convolutional neural networks course.



So you will remember that for face recognition, we train this Siamese network architecture that would learn, say, a 128 dimensional representation for different faces. And then you can compare these encodings in order to figure out if these two pictures are of the same face. The words encoding and embedding mean fairly similar things. So in the face recognition literature, people also use the term encoding to refer to these vectors, f(x(i)) and f(x(j)). One difference between the face recognition literature and what we do in word embeddings is that, for face recognition, you wanted to train a neural network that can take as input any face picture, even a picture you've never seen before, and have a neural network compute an encoding for that new picture. Whereas what we'll do, and you'll understand this better when we go through the next few videos, whereas what we'll do for learning word embeddings is that we'll have a fixed vocabulary of, say, 10,000 words. And we'll learn a vector e_1 through, say, $e_{10,000}$ that just learns a fixed encoding or learns a fixed embedding for each of the words in our vocabulary. So that's one difference between the set of ideas you saw for face recognition versus what the algorithms we'll discuss in the next few videos. But the terms encoding and embedding are used somewhat interchangeably. So the difference I just described is not represented by the difference in terminologies. It's just a difference in how we need to use these algorithms in face recognition, where there's unlimited sea of pictures you could see in the future. Versus natural language processing, where there might be just a fixed vocabulary, and everything else like that we'll just declare as an unknown word.

So in this section, you saw how using word embeddings allows you to implement this type of transfer learning. And how, by replacing the one-hot vectors we're using previously with the embedding vectors, you can allow your algorithms to generalize much better, or you can learn from much less label data. Next, I want to show you just a few more properties of these word embeddings and then after that, we will talk about algorithms for actually learning these word embeddings.

Properties of word embeddings

From the previous sections we have seen how word embeddings can help in building NLP applications. One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning and while reasonable analogies may not be by itself the most important NLP application, they might also help convey a sense of what these word embeddings are doing, what these word embeddings can do.

Let see a diagram where we have a featurized representations of a set of words that you might hope a word embedding could capture.

Analogies

	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	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

e_{5391} e_{man} e_{woman}

$e_{man} - e_{woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Man → Woman as King → ? Queen

$e_{king} - e_{queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

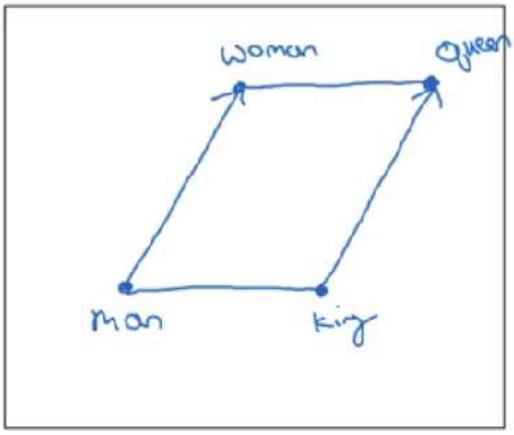
$e_{man} - e_{woman} \approx e_{king} - e_?$ e_{queen}

[Mikolov et. al., 2013, Linguistic regularities in continuous space word representations]

Andre

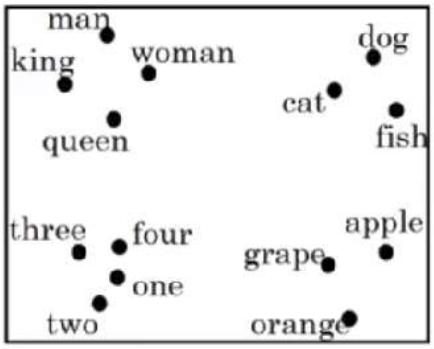
Let's say I pose a question, man is to woman as king is to what? Many of you will say, man is to woman as king is to queen. But is it possible to have an algorithm figure this out automatically?

Analogies using word vectors



300D

3000 → 2D



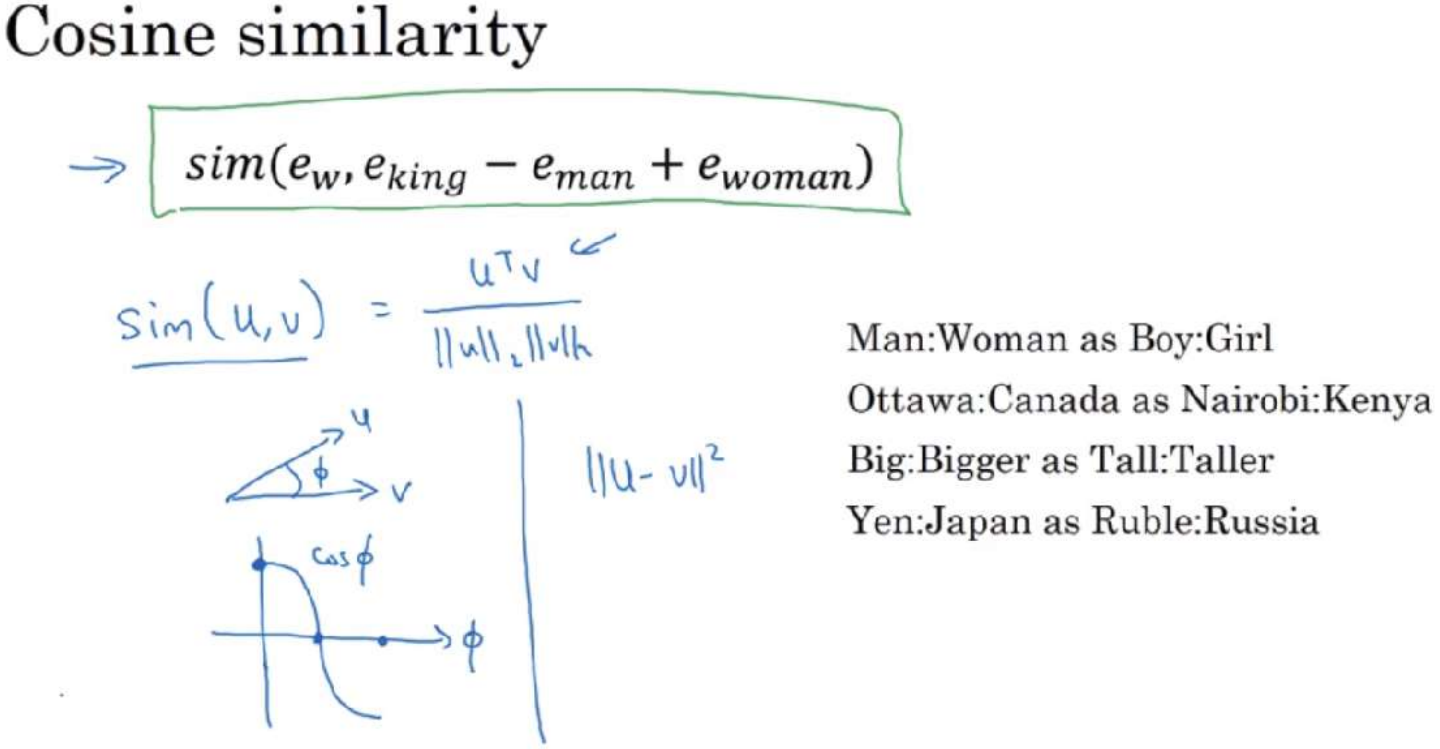
t-SNE

$e_{man} - e_{woman} \approx e_{king} - e_?$ e_w

Find word w: $\arg\max_w$ $sim(e_w, e_{king} - e_{man} + e_{woman})$

30 - 75%

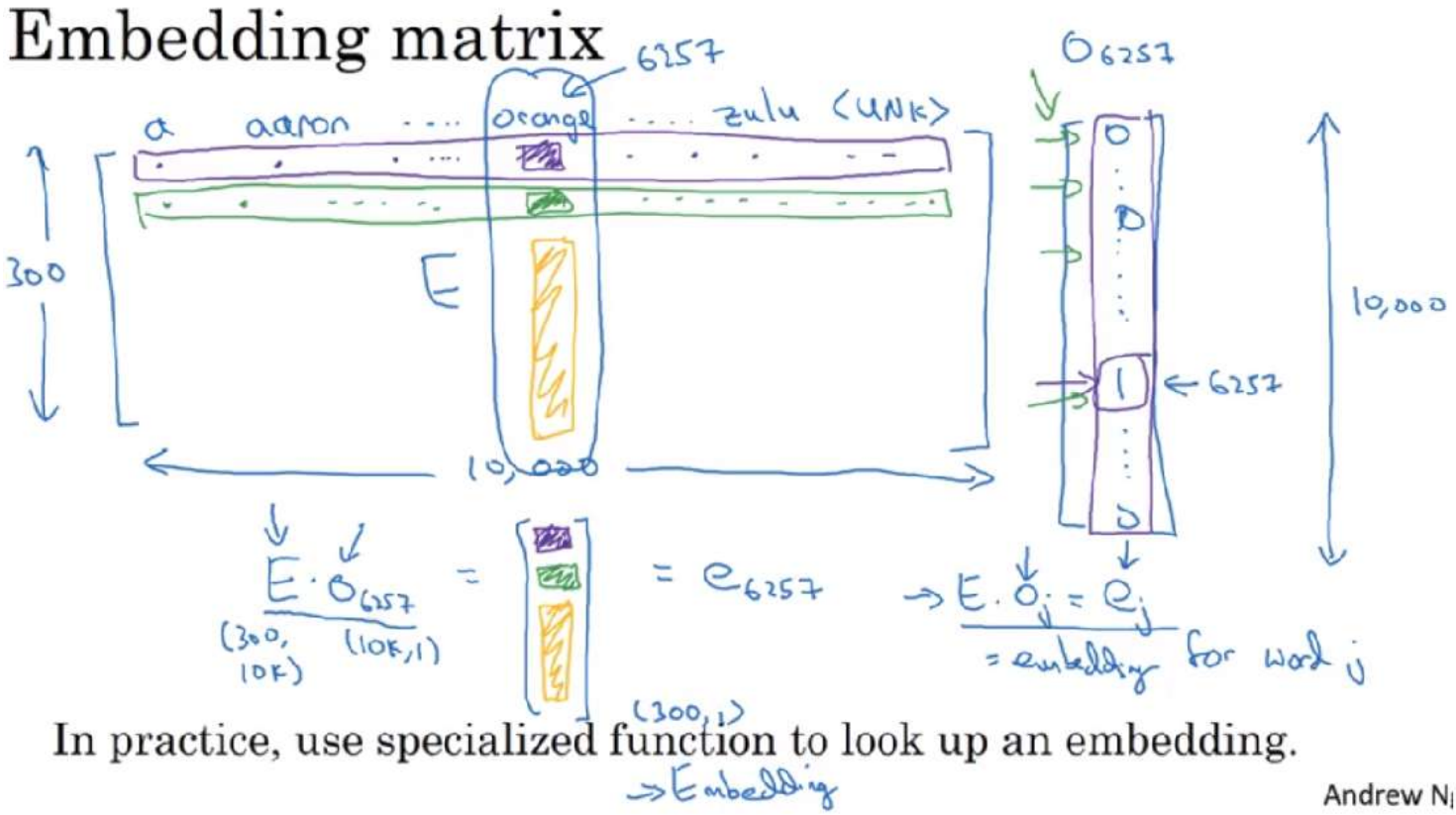
Andrew



So one of the remarkable results about word embeddings is the generality of analogy relationships they can learn. So for example, it can learn that man is to woman as boy is to girl, because the vector difference between man and woman, similar to king and queen and boy and girl, is primarily just the gender. It can learn that Ottawa, which is the capital of Canada, that Ottawa is to Canada as Nairobi is to Kenya. So that's the city capital is to the name of the country. It can learn that big is to bigger as tall is to taller, and it can learn things like that. Yen is to Japan, since yen is the currency of Japan, as ruble is to Russia and all of these things can be learned just by running a word embedding learning algorithm on the large text corpus. It can spot all of these patterns by itself, just by running from very large bodies of text. So in this section, we saw how word embeddings can be used for analogy reasoning and while you might not be trying to build an analogy reasoning system yourself as an application, this I hope conveys some intuition about the types of feature-like representations that these representations can learn. And you also saw how cosine similarity can be a way to measure the similarity between two different word embeddings. Now, we talked a lot about properties of these embeddings and how you can use them. Next, let's talk about how you'd actually learn these word embeddings.

Embedding matrix

Let's start to formalize the problem of learning a good word embedding. When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix. Let's take a look at what that means. Let's say, as usual we're using our 10,000-word vocabulary. So, the vocabulary has A, Aaron, Orange, Zulu, maybe also unknown word as a token. What we're going to do is learn embedding matrix E, which is going to be a 300 dimensional by 10,000 dimensional matrix, if you have 10,000 words vocabulary or maybe 10,001 is our word token, there's one extra token and the columns of this matrix would be the different embeddings for the 10,000 different words you have in your vocabulary.



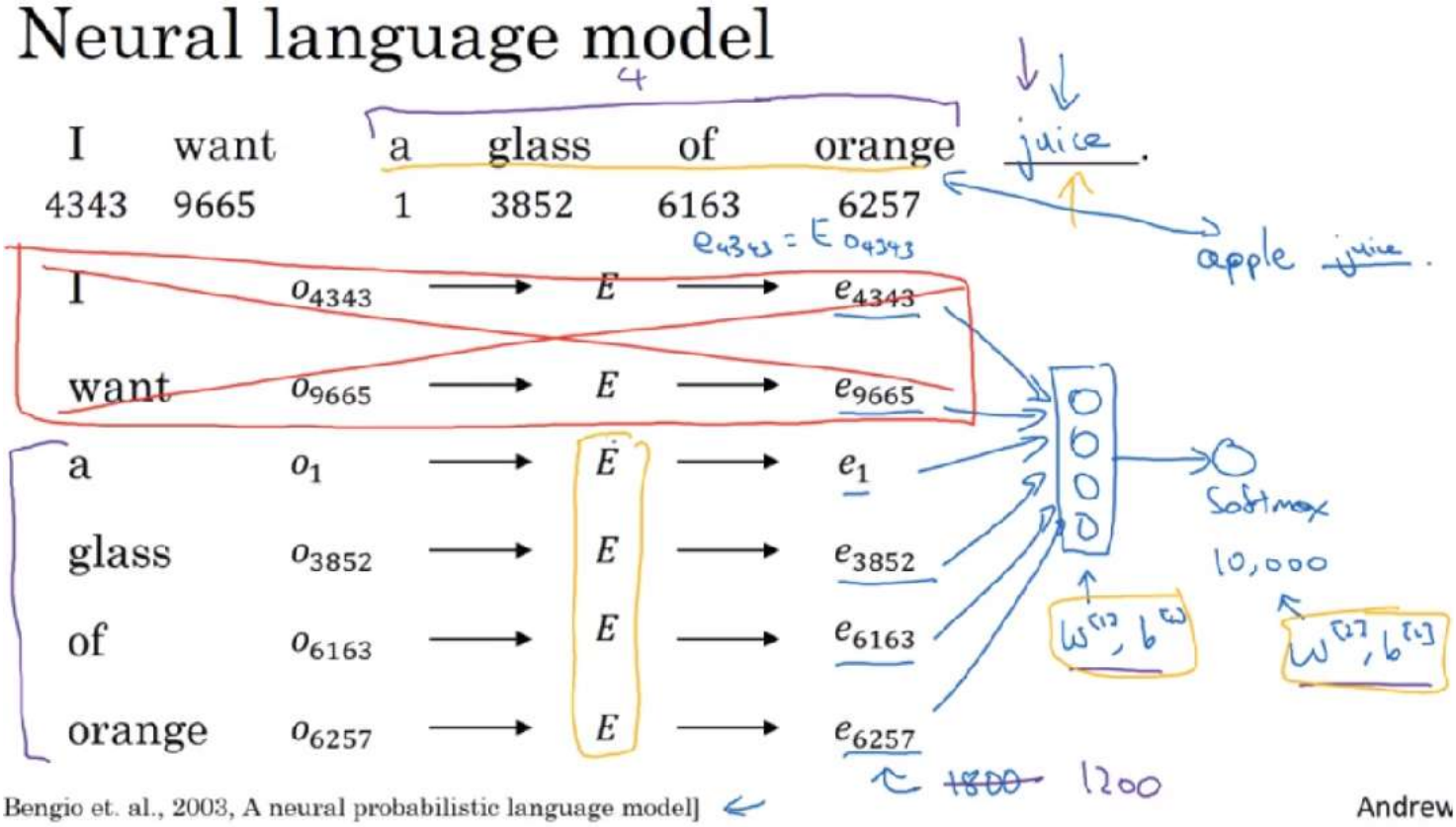
So, Orange was word number 6257 in our vocabulary of 10,000 words. So, one piece of notation we'll use is that O_{6257} was the one-hot vector with zeros everywhere and a one in position 6257. And so, this will be a 10,000-dimensional vector with a one in just one position. So, this isn't quite a drawn scale. Yes, this should be as tall as the embedding matrix on the left is wide. And if the embedding matrix is called capital E then notice that if you take E and multiply it by just one-hot vector by 0 of 6257, then this will be a 300-dimensional vector. So, E is 300x10,000 and O is 10,000x1. So, the product will be 300x1, so with 300-dimensional vector and notice that to compute the first element of this vector, of this 300-dimensional vector, what you do is you will multiply the first row of the matrix E with this. But all of these elements are zero except for element 6257 and so you end up with zero times this, zero times this, zero times this, and so on and then, 1 times whatever this is, and zero times this, zero times this, zero times and so on and so, you end up with the first element as whatever is that elements up there, under the Orange column and then, for the second element of this 300-dimensional vector we're computing, you would take the vector 6257 and multiply it by the second row with the matrix E. So again, you have zero times this, plus zero times this, plus zero times all of these are the elements and then one times this, and then zero times everything else and add that together. So you end up with this and so on as you go down the rest of this column. So, that's why the embedding matrix E times this one-hot vector here winds up selecting out this 300-dimensional column corresponding to the word Orange. So, this is going to be equal to e_{6257} which is the notation we're going to use to represent the embedding vector that 300 by one dimensional vector for the word Orange. And more generally, E for a specific word W, this is going to be embedding for a word W and more generally, E times O substitute J, one-hot vector with one that position J, this is going to be E_J and that's going to be the embedding for word J in the vocabulary. So, the thing to remember from this slide is that our goal will be to learn an embedding matrix E and what you see in the next section is you initialize E randomly and you're straight in the sense to learn all the parameters of this 300 by 10,000 dimensional matrix and E times this one-hot vector gives you the embedding vector. Now just one note, when we're writing the equation, it'll be convenient to write this type of notation where you take the matrix E and multiply it by the one-hot vector O. But if when you're implementing this, it is not efficient to actually implement this as a mass matrix vector multiplication because the one-hot vectors, now this is a relatively high dimensional vector and most of these elements are zero. So, it's actually not efficient to use a matrix vector multiplication to implement this because if we multiply a whole bunch of things by zeros and so the practice, you would actually use a specialized function to just look up a column of the Matrix E rather than do this

with the matrix multiplication. But writing of the map, it is just convenient to write it out this way. So, **in Keras for example there is a embedding layer and we use the embedding layer then it more efficiently just pulls out the column you want from the embedding matrix rather than does it with a much slower matrix vector multiplication**. So, in this section you saw the notations were used to describe algorithms to learning these embeddings and the key terminology is this matrix capital E which contain all the embeddings for the words of the vocabulary. In the next section, we'll start to talk about specific algorithms for learning this matrix E.

Learning Word Embeddings: Word2vec & GloVe

Learning word embeddings

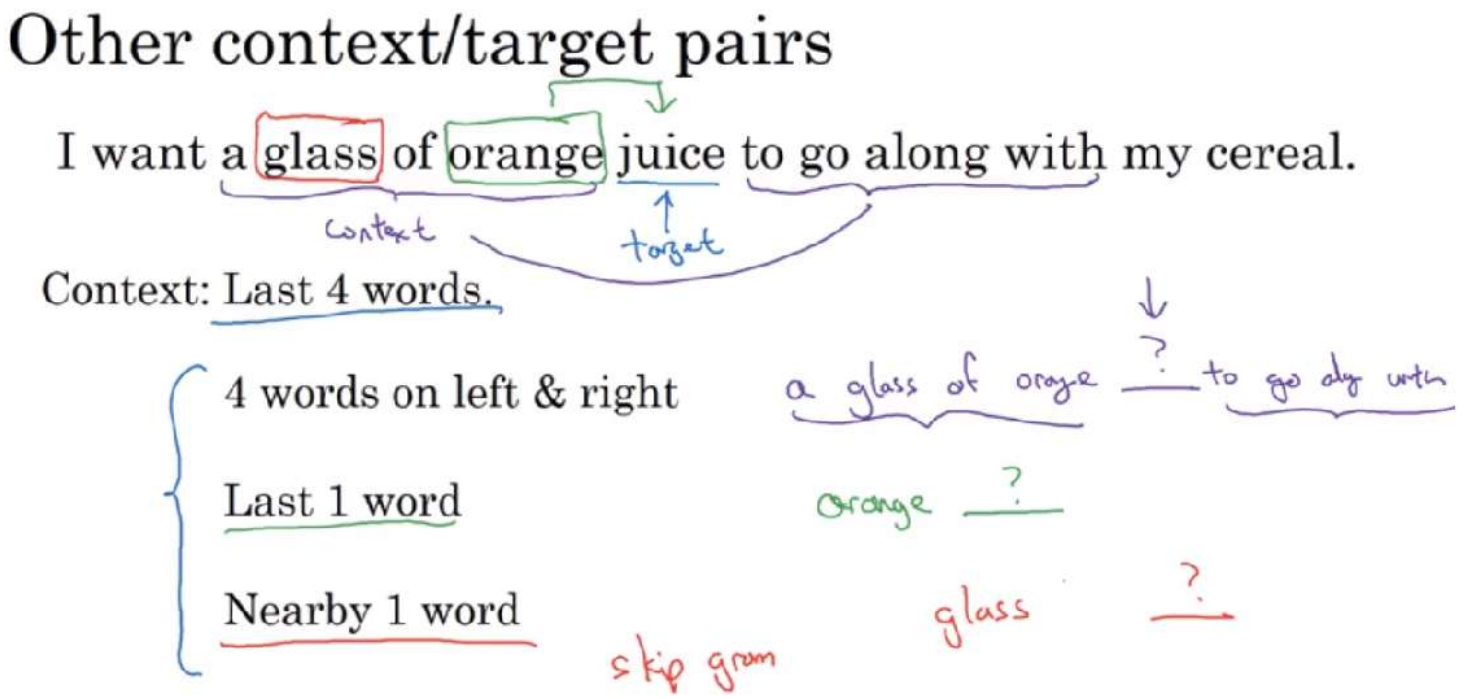
In this section, you'll start to learn some concrete algorithms for learning word embeddings. In the history of deep learning as applied to learning word embeddings, people actually started off with relatively complex algorithms and then over time, researchers discovered they can use simpler and simpler and simpler algorithms and still get very good results especially for a large dataset. But what happened is, some of the algorithms that are most popular today, they are so simple that if I present them first, it might seem almost a little bit magical, how can something this simple work? So, what I'm going to do is start off with some of the slightly more complex algorithms because I think it's actually easier to develop intuition about why they should work, and then we'll move on to simplify these algorithms and show you some of the simple algorithms that also give very good results. So, let's get started. Let's say you're building a language model and you do it with a neural network. So, during training, you might want your neural network to do something like input, I want a glass of orange, and then predict the next word in the sequence and below each of these words, I have also written down the index in the vocabulary of the different words. So it turns out that building a neural language model is the small way to learn a set of embeddings and the ideas I present on this slide were due to Yoshua Bengio, Rejean Ducharme, Pascals Vincent, and Christian Jauvin. So, here's how you can build a neural network to predict the next word in the sequence.



Let me take the list of words, I want a glass of orange, and let's start with the first word I. So I'm going to construct one add vector corresponding to the word I. So there's a one add vector with a one in position, 4343. So this is going to be 10,000 dimensional vector and what we're going to do is then have a matrix of parameters E, and take E times O to get an embedding vector e4343, and this step really means that e4343 is obtained by the matrix E times the one hot vector 43 and then we'll do the same for all of the other words. So the word want, is where 9665 one add vector, multiply by E

to get the embedding vector and similarly, for all the other words. A, is a first word in dictionary, alphabetic comes first, so there is O one, gets this E one. And similarly, for the other words in this phrase. So now you have a bunch of three dimensional embedding, so each of this is a 300 dimensional embedding vector. and what we can do, is feed all of them into a neural network. So here is the neural network layer. And then this neural network feeds to a softmax, which has it's own parameters as well. And a softmax classifies among the 10,000 possible outputs in the vocab for those final word we're trying to predict. And so, if in the training slide we saw the word juice then, the target for the softmax in training repeat that it should predict the other word juice was what came after this. So this hidden name here will have his own parameters. So have some, I'm going to call this W1 and there's also B1. The softmax there was this own parameters W2, B2, and they're using 300 dimensional word embeddings, then here we have six words. So, this would be 6x300. So this layer or this input will be a 1,800 dimensional vector obtained by taking your six embedding vectors and stacking them together. Well, what's actually more commonly done is to have a fixed historical window. So for example, you might decide that you always want to predict the next word given say the previous four words, where four here is a hyperparameter of the algorithm. So this is how you adjust to either very long or very short sentences or you decide to always just look at the previous four words, so you say, I will still use those four words and so, if you're always using a four word history, this means that your neural network will input a 1,200 dimensional feature vector, go into this layer, then have a softmax and try to predict the output. And again, variety of choices. And using a fixed history, just means that you can deal with even arbitrarily long sentences because the input sizes are always fixed. So, the parameters of this model will be this matrix E, and use the same matrix E for all the words. So you don't have different matrices for different positions in the proceedings four words, is the same matrix E and then, these weights are also parameters of the algorithm and you can use that crop to perform gradient to sent to maximize the likelihood of your training set to just repeatedly predict given four words in a sequence, what is the next word in your text corpus? And it turns out that this algorithm we'll learn pretty decent word embeddings and the reason is, if you remember our orange juice, apple juice example, is in the algorithm's incentive to learn pretty similar word embeddings for orange and apple because doing so allows it to fit the training set better because it's going to see orange juice sometimes, or see apple juice sometimes, and so, if you have only a 300 dimensional feature vector to represent all of these words, the algorithm will find that it fits the training set fast. If apples, oranges, and grapes, and pears, and so on and maybe also durians which is a very rare fruit and that with similar feature vectors. **So, this is one of the earlier and pretty successful algorithms for learning word embeddings, for learning this matrix E**.

But now let's generalize this algorithm and see how we can derive even simpler algorithms. So, I want to illustrate the other algorithms using a more complex sentence as our example. Let's say that in your training set, you have this longer sentence, I want a glass of orange juice to go along with my cereal. So, what we saw on the last slide was that the job of the algorithm was to predict some word juice, which we are going to call the target words, and it was given some context which was the last four words. And so, if your goal is to learn a embedding of researchers I've experimented with many different types of context. If it goes to build a language model then is natural for the context to be a few words right before the target word. But if your goal is into learn the language model per se, then you can choose other contexts. For example, you can pose a learning problem where the context is the four words on the left and right. So, you can take the four words on the left and right as the context, and what that means is that we're posing a learning problem where the algorithm is given four words on the left. So, a glass of orange, and four words on the right, to go along with, and this has to predict the word in the middle. And posing a learning problem like this where you have the embeddings of the left four words and the right four words feed into a neural network, similar to what you saw in the previous slide, to try to predict the word in the middle, try to put it target word in the middle, this can also be used to learn word embeddings. Or if you want to use a simpler context, maybe you'll just use the last one word. So given just the word orange, what comes after orange? So this will be different learning problem where you tell it one word, orange, and will say well, what do you think is the next word and you can construct a neural network that just fits in the word, the one previous word or the embedding of the one previous word to a neural network as you try to predict the next word. Or, one thing that works surprisingly well is to take a nearby one word.



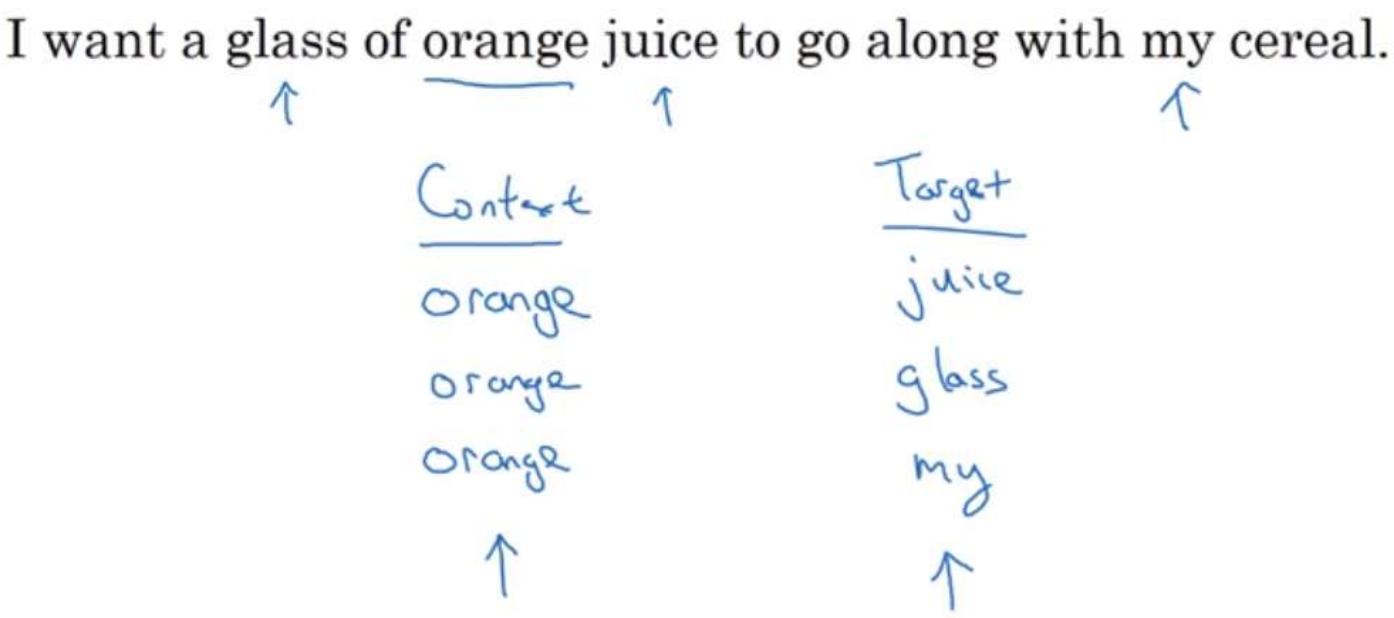
Some might tell you that, well, take the word glass, is somewhere close by. Some might say, I saw the word glass and then there's another words somewhere close to glass, what do you think that word is? So, that'll be using nearby one word as the context and we'll formalize this in the next section but this is the idea of a **Skip-Gram model**, and just an example of a simpler algorithm where the context is now much simpler, is just one word rather than four words, but this works remarkably well. 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 a 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 work embeddings as well. I will formalize the details of this in the next section where we talk about the Word2Vec model.

To summarize, in this section you saw how the language modeling problem which causes the pose of machines learning problem where you input the context like the last four words and predicts some target words, how posing that problem allows you to learn input word embedding. In the next section, you'll see how using even simpler context and even simpler learning algorithms to mark from context to target word, can also allow you to learn a good word embedding.

Word2Vec

In the last section, we saw how you can learn a natural language model in order to get good word embeddings. In this section, you see the Word2Vec algorithm which is simple and comfortably more efficient way to learn this types of embeddings. Lets take a look. Most of the ideas I'll present in this section are due to Tomas Mikolov, Kai Chen, Greg Corrado, and Jeff Dean. Let's say you're given this sentence in your training set. In the skip-gram model, what we're going to do is come up with a few context to target errors to create our supervised learning problem. So rather than having the context be always the last four words or the last end words immediately before the target word, what we're going to do is, say, randomly pick a word to be the context word and let's say we choose the word orange and what we're going to do is randomly pick another word within some window. Say +-5 words or +-10 words of the context word and we choose that to be target word. So maybe just by chance you might pick juice to be a target word, that's just one word later. Or you might choose two words before. So you have another pair where the target could be glass or, maybe just by chance you choose the word my as the target and so we'll set up a supervised learning problem where given the context word, you're asked to predict what is a randomly chosen word within say, a +-10 word window, or +-5 or 10 word window of that input context word and obviously, this is not a very easy learning problem, because within +- 10 words of the word orange, it could be a lot of different words. But a goal that's setting up this supervised learning problem, isn't to do well on the supervised learning problem per se, it is that we want to use this learning problem to learn good word embeddings.

Skip-grams



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

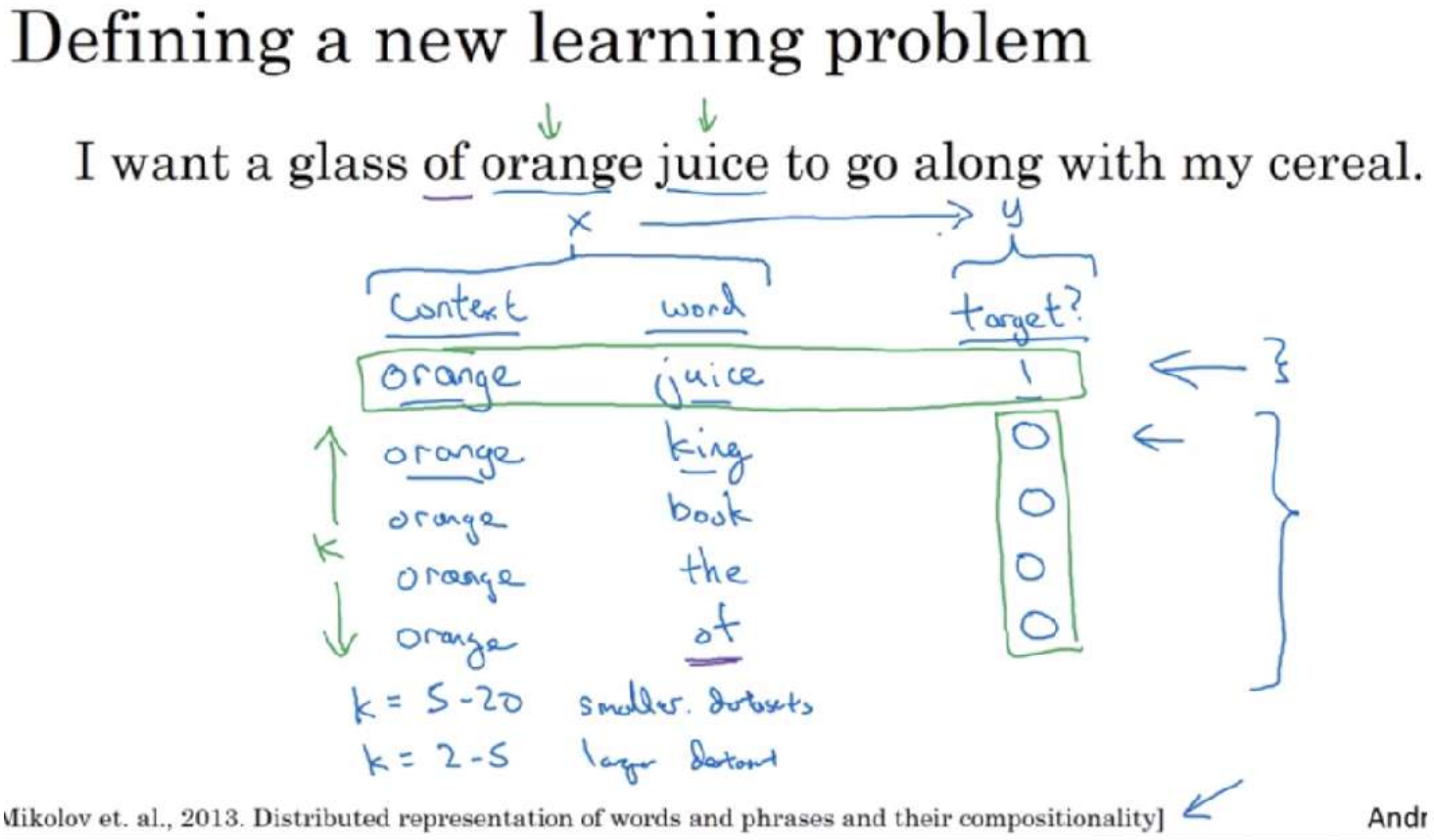
So, here are the details of the model. Let's say that we'll continue to our vocab of 10,000 words and some have been on vocab sizes that exceeds a million words but the basic supervised learning problem we're going to solve is that we want to learn the mapping from some Context c, such as the word orange to some target, which we will call t, which might be the word juice or the word glass or the word my, if we use the example from the previous section. So in our vocabulary, orange is word 6257, and the word
juice is the word 4834 in our vocab of 10,000 words and so that's the input x that you want to learn to map to that open y. So to represent the input such as the word orange, we can start out with some one hot vector which is going to be write as O subscript C, so there's a one hot vector for the context words and then similar to what we saw on the last section we can take the embedding matrix E, multiply E by the vector O subscript C, and this gives you your embedding vector for the input context word, so here EC is equal to capital E times that one hot vector. Then in this neural network that we formed we're going to take this vector EC and feed it to a softmax unit. So I've been drawing softmax unit as a node in a neural network. That's not an o, that's a softmax unit and then there's a drop in the softmax unit to output y hat.
So to write out this model in detail. This is the model, the softmax model, probability of different tanka words given the input context word as e to the e, theta t transpose, ec. Divided by some over all words, so we're going to say, sum from J equals one to all 10,000 words of e to the theta j transposed ec. So here theta T is the parameter associated with, I'll put t, but really there's a chance of a particular word, t, being the label.
So I've left off the biased term to solve mass but we could include that too if we wish.
And then finally the loss function for softmax will be the usual.
So we use y to represent the target word. And we use a one-hot representation for y hat and y here. Then the lost would be The negative log liklihood, so sum from i equals 1 to 10,000 of yi log yi hat. So that's a usual loss for softmax where we're representing the target y as a one hot vector. So this would be a one hot vector with just 1 1 and the rest zeros. And if the target word is juice, then it'd be element 4834 from up here. That is equal to 1 and the rest will be equal to 0. And similarly Y hat will

be a 10,000 dimensional vector output by the softmax unit with probabilities for all 10,000 possible targets words. So to summarize, this is the overall little model, little neural network with basically looking up the embedding and then just a soft max unit. And the matrix E will have a lot of parameters, so the matrix E has parameters corresponding to all of these embedding vectors, E subscript C. And then the softmax unit also has parameters that gives the theta T parameters but if you optimize this loss function with respect to the all of these parameters, you actually get a pretty good set of embedding vectors. So this is called the skip-gram model because is taking as input one word like orange and then trying to predict some words skipping a few words from the left or the right side. To predict what comes little bit before little bit after the context words. Now, it turns out there are a couple problems with using this algorithm. And the primary problem is computational speed. In particular, for the softmax model, every time you want to evaluate this probability, you need to carry out a sum over all 10,000 words in your vocabulary. And maybe 10,000 isn't too bad, but if you're using a vocabulary of size 100,000 or a 1,000,000, it gets really slow to sum up over this denominator every single time. And, in fact, 10,000 is actually already that will be quite slow, but it makes even harder to scale to larger vocabularies. So there are a few solutions to this, one which you see in the literature is to use a hierarchical softmax classifier. And what that means is, instead of trying to categorize something into all 10,000 carries on one go. Imagine if you have one classifier, it tells you is the target word in the first 5,000 words in the vocabulary? Or is in the second 5,000 words in the vocabulary? And lets say this binary cost that it tells you this is in the first 5,000 words, think of second class to tell you that this in the first 2,500 words of vocab or in the second 2,500 words vocab and so on. Until eventually you get down to classify exactly what word it is, so that the leaf of this tree, and so having a tree of classifiers like this, means that each of the retriever nodes of the tree can be just a binding classifier. And so you don't need to sum over all 10,000 words or else it will capsize in order to make a single classification. In fact, the computational classifying tree like this scales like log of the vocab size rather than linear in vocab size. So this is called a hierarchical softmax classifier. I should mention in practice, the hierarchical softmax classifier doesn't use a perfectly balanced tree or this perfectly symmetric tree, with equal numbers of words on the left and right sides of each branch. In practice, the hierarchical software classifier can be developed so that the common words tend to be on top, whereas the less common words like durian can be buried much deeper in the tree. Because you see the more common words more often, and so you might need only a few traversals to get to common words like the and of. Whereas you see less frequent words like durian much less often, so it says okay that are buried deep in the tree because you don't need to go that deep. So there are various heuristics for building the tree how you used to build the hierarchical software spire.
So this is one idea you see in the literature, the speeding up the softmax classification.
But I won't spend too much more time.
And you can read more details of this on the paper that I referenced by Thomas
and others, on the first slide. But I won't spend too much more time on this. Because in the next video, where she talk about a different method, called nectar sampling, which I think is even simpler. And also works really well for speeding up the softmax classifier and the problem of needing the sum over the entire cap size in the denominator. So you see more of that in the next video. But before moving on, one quick Topic I want you to understand is how to sample the context C. So once you sample the context C, the target T can be sampled within, say, a plus minus ten word window of the context C, but how do you choose the context C? One thing you could do is just sample uniformly, at random, from your training corpus. When we do that, you find that there are some words like the, of, a, and, to and so on that appear extremely frequently. And so, if you do that, you find that in your context to target mapping pairs just get these these types of words extremely frequently, whereas there are other words like orange, apple, and also durian that don't appear that often. And maybe you don't want your training site to be dominated by these extremely frequently or current words, because then you spend almost all the effort updating ec, for those frequently occurring words. But you want to make sure that you spend some time updating the embedding, even for these less common words like e durian. So in practice the distribution of words pc isn't taken just entirely uniformly at random for the training set purpose, but instead there are different heuristics that you could use in order to balance out something from the common words together with the less common words.

So that's it for the Word2Vec skip-gram model. If you read the original paper by that I referenced earlier, you find that that paper actually had two versions of this Word2Vec model, the skip gram was one. And the other one is called the CBow, the continuous backwards model, which takes the surrounding contexts from middle word, and uses the surrounding words to try to predict the middle word, and that algorithm also works, it has some advantages and disadvantages. But the key problem with this algorithm with the skip-gram model as presented so far is that the softmax step is very expensive to calculate because needing to sum over your entire vocabulary size into the denominator of the soft packs. In the next video I show you an algorithm that modifies the training objective that makes it run much more efficiently therefore lets you apply this in a much bigger fitting set as well and therefore learn much better word embeddings.

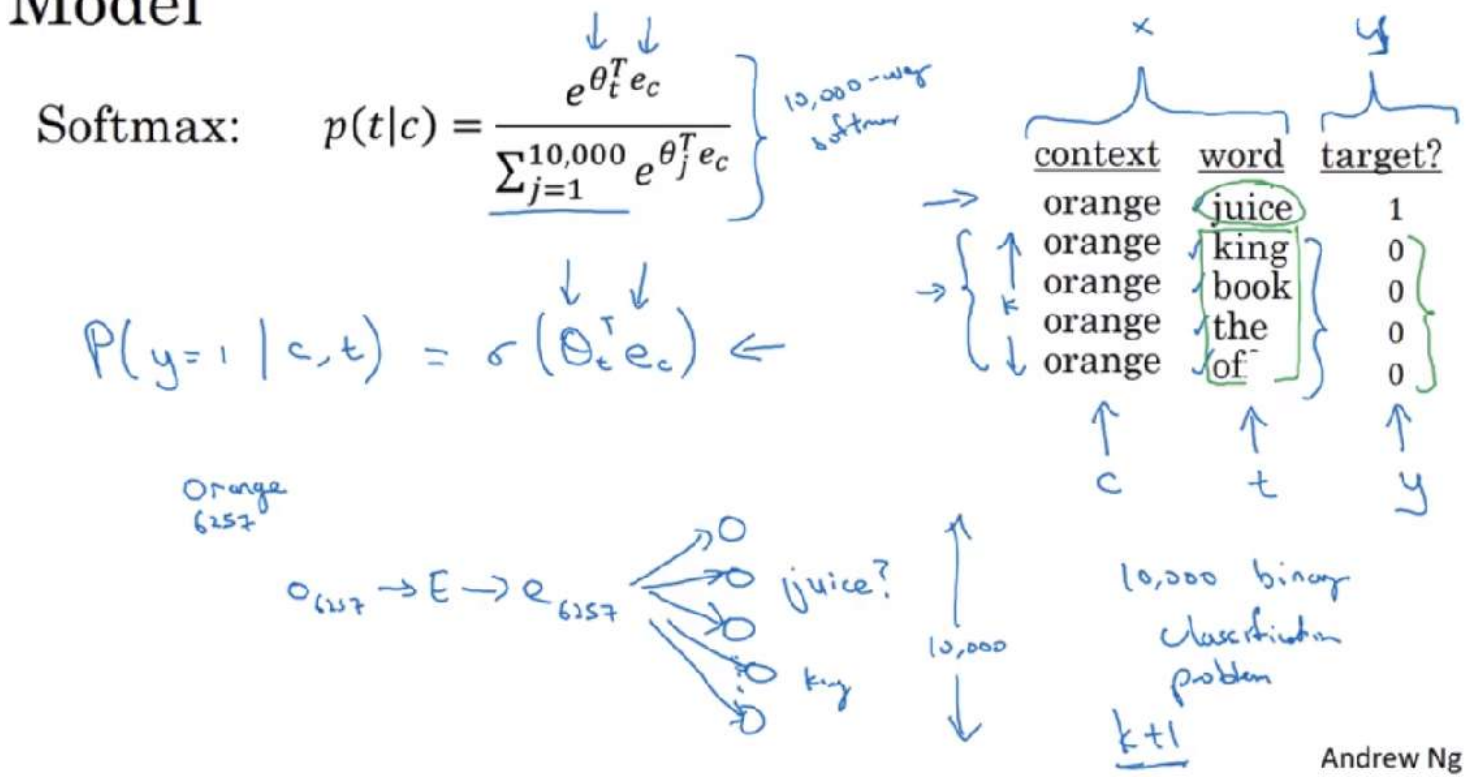
Negative Sampling

In the last section, we saw how the Skip-Gram model allows us to construct a supervised learning task. So we map from context to target and how that allows you to learn a useful word embedding. But the downside of that was the Softmax objective was slow to compute. In this section, we'll see a modified learning problem called **negative sampling** that allows us to do something similar to the **Skip-Gram model** we saw just now, but with a much more efficient learning algorithm. Let's see how we can do this. Most of the ideas presented in this video are due to Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. So what we're going to do in this algorithm is create a new supervised learning problem and the problem is, given a pair of words like orange and juice, we're going to predict, is this a context-target pair? So in this example, orange juice was a positive example. And how about orange and king? Well, that's a negative example, so we're going to write 0 for the target. So what we're going to do is we're actually going to sample a context and a target word. So in this case, we have orange and juice and we'll associate that with a label of 1, so just put words in the middle and then having generated a positive example, so the positive example is generated exactly how we generated it in the previous section. Sample a context word, look around a window of say, plus-minus ten words and pick a target word. So that's how we generate the first row of this table with orange, juice, 1 and then to generate a negative example, we're going to take the same context word and then just pick a word at random from the dictionary.



So in this case, we chose the word king at random and we will label that as 0 and then let's take orange and let's pick another random word from the dictionary. Under the assumption that if we pick

a random word, it probably won't be associated with the word orange, so orange, book, 0 and let's pick a few others, orange, maybe just by chance, we'll pick the 0 and then orange and then orange, and maybe just by chance, we'll pick the word of and we'll put a 0 there and notice that all of these are labeled as 0 even though the word of actually appears next to orange as well. So to summarize, the way we generated this data set is, we'll pick a context word and then pick a target word and that is the first row of this table. That gives us a positive example. So context, target, and then give that a label of 1 and then what we'll do is for some number of times say, k times, we're going to take the same context word and then pick random words from the dictionary, king, book, the, of, whatever comes out at random from the dictionary and label all those 0, and those will be our negative examples and it's okay if just by chance, one of those words we picked at random from the dictionary happens to appear in the window, in a plus-minus ten word window say, next to the context word, orange. Then we're going to create a supervised learning problem where the learning algorithm inputs x, inputs this pair of words, and it has to predict the target label to predict the output y. So the problem is really given a pair of words like orange and juice, do you think they appear together? Do you think I got these two words by sampling two words close to each other? Or do you think I got them as one word from the text and one word chosen at random from the dictionary? It's really to try to distinguish between these two types of distributions from which you might sample a pair of words. So this is how you generate the training set. How do you choose k, Mikolov et al, recommend that maybe **k is 5 to 20** for smaller data sets and if you have a very large data set, then chose k to be smaller. So k equals 2 to 5 for larger data sets, and large values of k for smaller data sets. Okay, and in this example, we'll just use k = 4. Next, let's describe the supervised learning model for learning a mapping from x to y. So here was the Softmax model you saw from the previous section and here is the training set we got from the previous section where again, this is going to be the new input x and this is going to be the value of y you're trying to predict.



So to define the model, I'm going to use this to denote, this was c for the context word, this to denote the possible target word, t, and this, I'll use y to denote 0, 1, this is a context target pair. So what we're going to do is define a logistic regression model. Say, that the chance of y = 1, given the input c, t pair, we're going to model this as basically a regression model, but the specific formula we'll use s sigma applied to theta transpose, theta t transpose, e c. So the parameters are similar as before, you have one parameter vector theta for each possible target word. And a separate parameter vector, really the embedding vector, for each possible context word. And we're going to use this formula to estimate the probability that y is equal to 1. So if you have k examples here, then you can think of this as having a k to 1 ratio of negative to positive examples. So for every positive

examples, you have k negative examples with which to train this logistic regression-like model and so to draw this as a neural network, if the input word is orange, which is word 6257, then what you do is, you input the one hot vector passing through e, do the multiplication to get the embedding vector 6257 and then what you have is really 10,000 possible logistic regression classification problems. Where one of these will be the classifier corresponding to, well, is the target word juice or not? And then there will be other words, for example, there might be ones somewhere down here which is predicting, is the word king or not and so on, for these possible words in your vocabulary. So think of this as having 10,000 binary logistic regression classifiers, but instead of training all 10,000 of them on every iteration, we're only going to train five of them. We're going to train the one responding to the actual target word we got and then train four randomly chosen negative examples. And this is for the case where k is equal to 4. So instead of having one giant 10,000 way Softmax, which is very expensive to compute, we've instead turned it into 10,000 binary classification problems, each of which is quite cheap to compute. And on every iteration, we're only going to train five of them or more generally, k + 1 of them, of k negative examples and one positive examples. And this is why the computation cost of this algorithm is much lower because you're updating k + 1, let's just say units, k + 1 binary classification problems. Which is relatively cheap to do on every iteration rather than updating a 10,000 way Softmax classifier. So you get to play with this algorithm in the problem exercise for this week as well. S**o this technique is called negative sampling because what you're doing is, you have a positive example, the orange and then juice. And then you will go and deliberately generate a bunch of negative examples, negative samplings,** hence, the name negative sampling, with which to train four more of these binary classifiers and on every iteration, you choose four different random negative words with which to train your algorithm on. Now, before wrapping up, one more important detail with this algorithm is, how do you choose the negative examples? So after having chosen the context word orange, how do you sample these words to generate the negative examples? So one thing you could do is sample the words in the middle, the candidate target words. One thing you could do is sample it according to the empirical frequency of words in your corpus. So just sample it according to how often different words appears. But the problem with that is that you end up with a very high representation of words like the, of, and, and so on. One other extreme would be to say, you use 1 over the vocab size, sample the negative examples uniformly at random, but that's also very non-representative of the distribution of English words. So the authors, Mikolov et al, reported that empirically, what they found to work best was to take this heuristic value, which is a little bit in between the two extremes of sampling from the empirical frequencies, meaning from whatever's the observed distribution in English text to the uniform distribution. And what they did was they sampled proportional to their frequency of a word to the power of three-fourths. So if f of wi is the observed frequency of a particular word in the English language or in your training set corpus, then by taking it to the power of three-fourths, this is somewhere in-between the extreme of taking uniform distribution. And the other extreme of just taking whatever was the observed distribution in your training set and so'm not sure this is very theoretically justified, but multiple researchers are now using this heuristic, and it seems to work decently well.

So to summarize, we've seen how you can learn word vectors in a Softmax classier, but it's very computationally expensive and in this section, we saw how by changing that to a bunch of binary classification problems, you can very efficiently learn words vectors and if we run this algorithm, we'll be able to learn pretty good word vectors. Now of course, as is the case in other areas of deep learning as well, there are open source implementations and there are also pre-trained word vectors that others have trained and released online under permissive licenses.

GloVe word vectors

We learned about several algorithms for computing words embeddings. Another algorithm that has some momentum in the NLP community is the GloVe algorithm. This is not used as much as the Word2Vec or the skip-gram models, but it has some enthusiasts. Because I think, in part of its simplicity. Let's take a look. The **GloVe algorithm** was created by Jeffrey Pennington, Richard Socher, and Chris Manning and **GloVe stands for global vectors for word representation**. I think one confusing part of this algorithm is, if you look at equation below

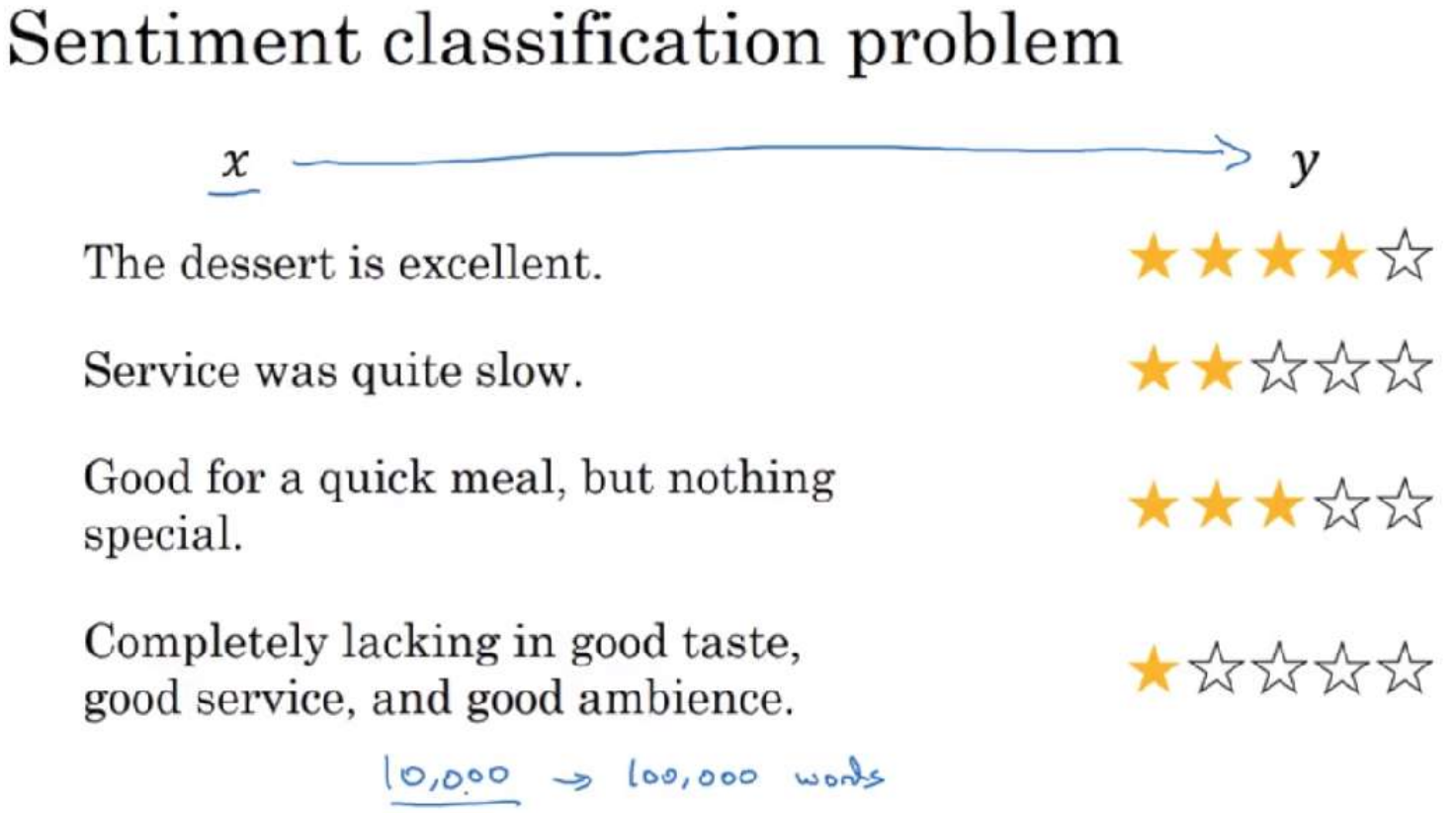
$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i - b_j' - \log X_{ij})^2$$

It seems almost too simple. How could it be that just minimizing a **square cost function** like this allows you to learn meaningful word embeddings? But it turns out that this works and the way that the inventors end up with this algorithm was, they were building on the history of much more complicated algorithms like the newer language model, and then later, there came the **Word2Vec skip-gram model**, and then this came later. And we really hope to simplify all of the earlier algorithms. Before concluding our discussion of algorithms concerning word embeddings, there's one more property of them that we should discuss briefly. Which is that? We started off with this featurization view as the motivation for learning word vectors. We said, "Well, maybe the first component of the embedding vector to represent gender, the second component to represent how royal it is, then the age and then whether it's a food, and so on." But when you learn a word embedding using one of the algorithms that we've seen, such as the GloVe algorithm that we just saw, what happens is, you cannot guarantee that the individual components of the embeddings are interpretable. So, that's it for learning word embeddings. We've now seen a variety of algorithms for learning these word embeddings. Next, we'll show how we can use these algorithms to carry out sentiment classification.

Applications using Word Embeddings

Sentiment Classification

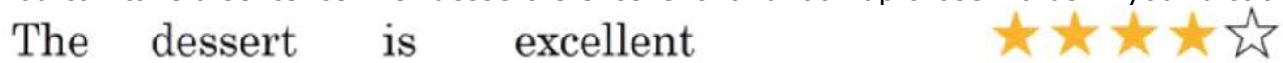
Sentiment classification is the task of looking at a piece of text and telling if someone likes or dislikes the thing they're talking about. It is one of the most important building blocks in NLP and is used in many applications. One of the challenges of sentiment classification is you might not have a huge label training set for it. But with word embeddings, you're able to build good sentiment classifiers even with only modest-size label training sets. Let's see how you can do that. So here's an example of a sentiment classification problem.



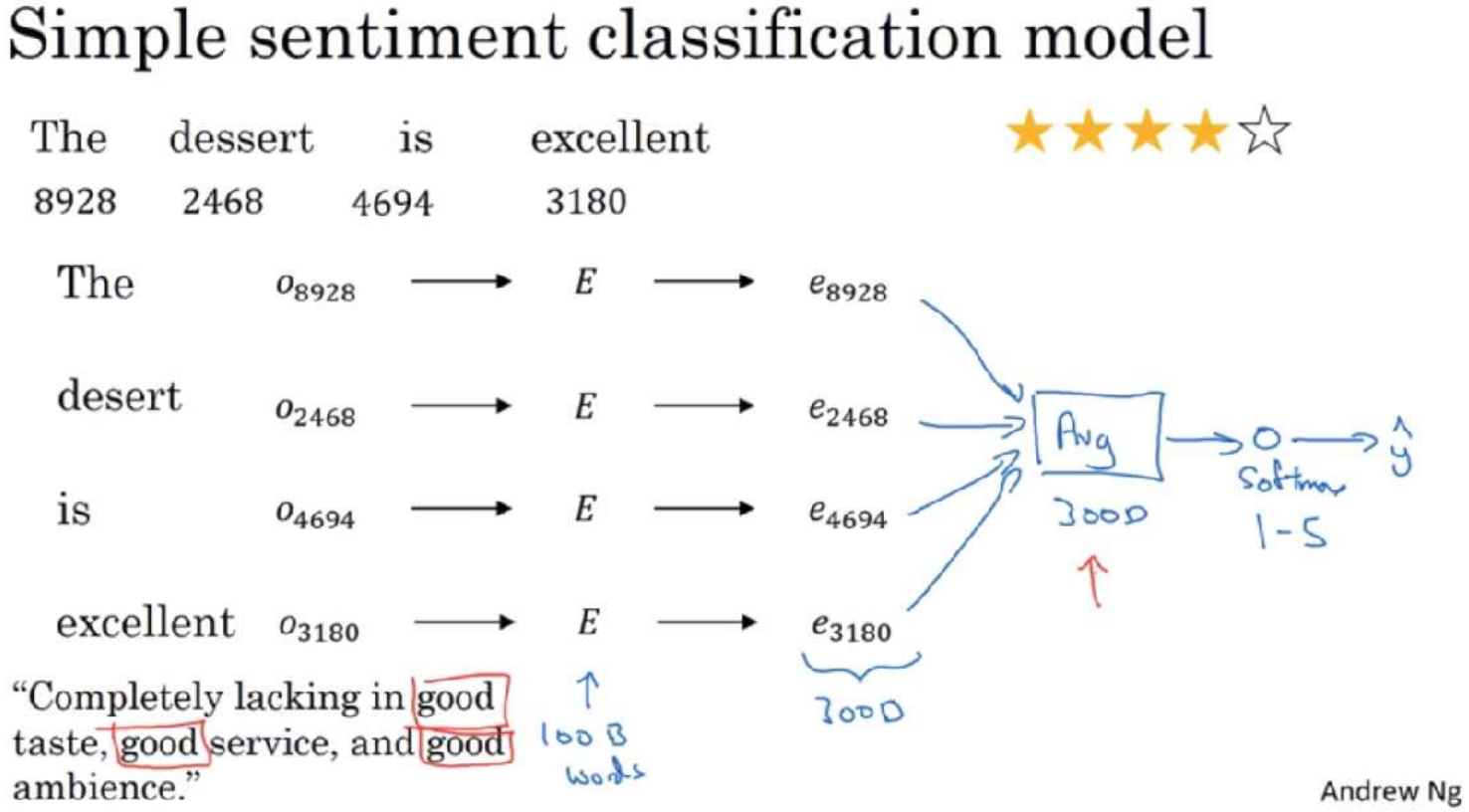
The input X is a piece of text and the output Y that you want to predict is what is the sentiment, such as the star rating of, let's say, a restaurant review. So if someone says, "The dessert is excellent" and they give it a four-star review, "Service was quite slow" two-star review, "Good for a quick meal but nothing special" three-star review. And this is a pretty harsh review, "Completely lacking in good

taste, good service, and good ambiance." That's a one-star review. So if you can train a system to map from X or Y based on a label data set like this, then you could use it to monitor comments that people are saying about maybe a restaurant that you run. So people might also post messages about your restaurant on social media, on Twitter, or Facebook, or Instagram, or other forms of social media. And if you have a sentiment classifier, they can look just a piece of text and figure out how positive or negative is the sentiment of the poster toward your restaurant. Then you can also be able to keep track of whether or not there are any problems or if your restaurant is getting better or worse over time. So one of the challenges of sentiment classification is you might not have a huge label data set. So for sentimental classification task, training sets with maybe anywhere from 10,000 to maybe 100,000 words would not be uncommon. Sometimes even smaller than 10,000 words and word embeddings that you can take can help you to much better understand especially when you have a small training set. So here's what you can do. We'll go for a couple different algorithms in this section. Here's a simple sentiment classification model.

You can take a sentence like "dessert is excellent" and look up those words in your dictionary.

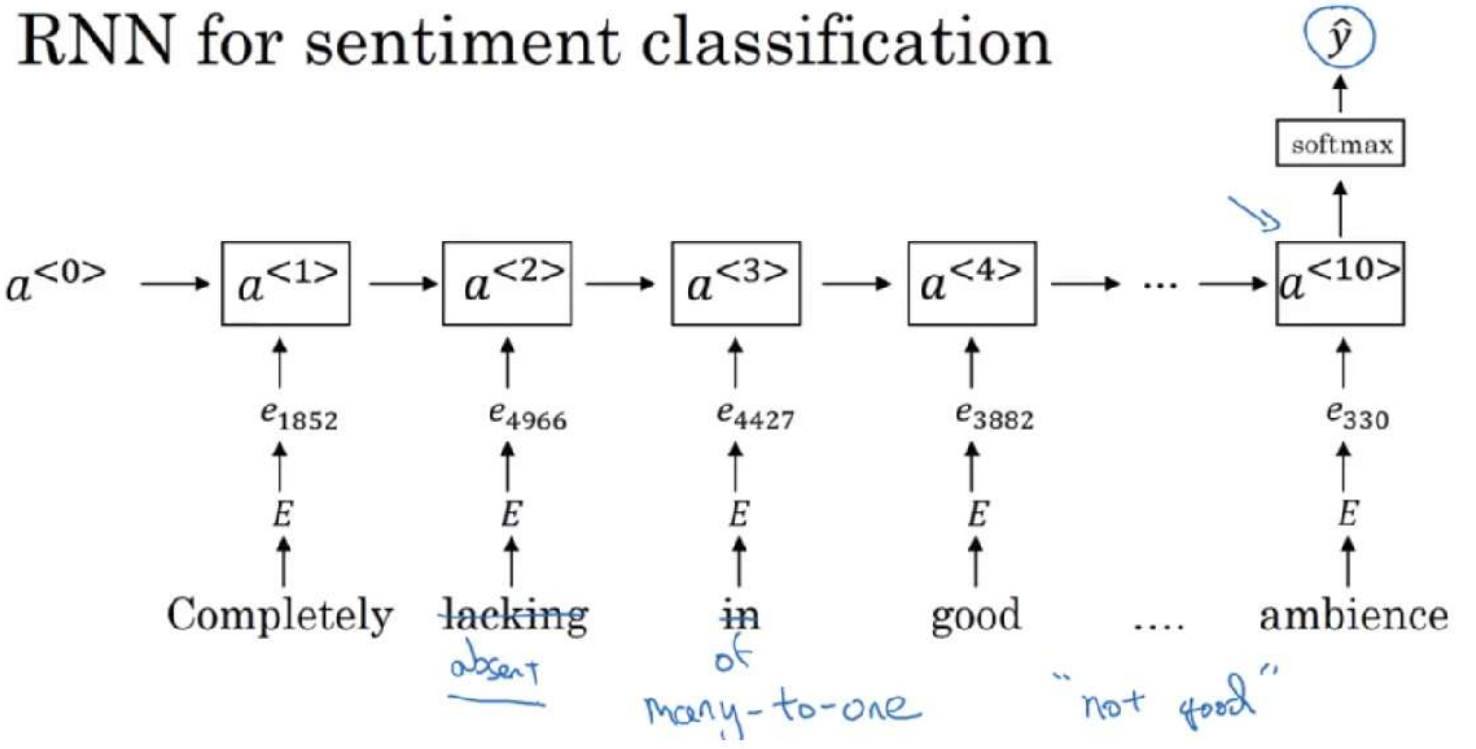


We use a 10,000-word dictionary as usual. And let's build a classifier to map it to the output Y that this was four stars. So given these four words, as usual, we can take these four words and look up the one-hot vector. So there's 0 8 9 2 8 which is a one-hot vector multiplied by the embedding matrix E, which can learn from a much larger text corpus. It can learn in embedding from, say, a billion words or a hundred billion words, and use that to extract out the embedding vector for the word "the", and then do the same for "dessert", do the same for "is" and do the same for "excellent". And if this was trained on a very large data set, like a hundred billion words, then this allows you to take a lot of knowledge even from infrequent words and apply them to your problem, even words that weren't in your labeled training set. Now here's one way to build a classifier, which is that you can take these vectors, let's say these are 300-dimensional vectors, and you could then just sum or average them and I'm just going to put a bigger average operator here and you could use sum or average. And this gives you a 300-dimensional feature vector that you then pass to a soft-max classifier which then outputs Y-hat and so the softmax can output what are the probabilities of the five possible outcomes from one-star up to five-star. So this will be assortment of the five possible outcomes to predict what is Y. So notice that by using the average operation here, this particular algorithm works for reviews that are short or long because even if a review that is 100 words long, you can just sum or average all the feature vectors for all hundred words and so that gives you a representation, a 300-dimensional feature representation, that you can then pass into your sentiment classifier. So this average will work decently well. And what it does is it really averages the meanings of all the words or sums the meaning of all the words in your example. And this will work to [inaudible]. So one of the problems with this algorithm is it ignores word order. In particular, this is a very negative review, "Completely lacking in good taste, good service, and good ambiance". But the word good appears a lot.



This is a lot. Good, good, good. So if you use an algorithm like this that ignores word order and just sums or averages all of the embeddings for the different words, then you end up having a lot of the representation of good in your final feature vector and your classifier will probably think this is a good review even though this is actually very harsh. This is a one-star review.

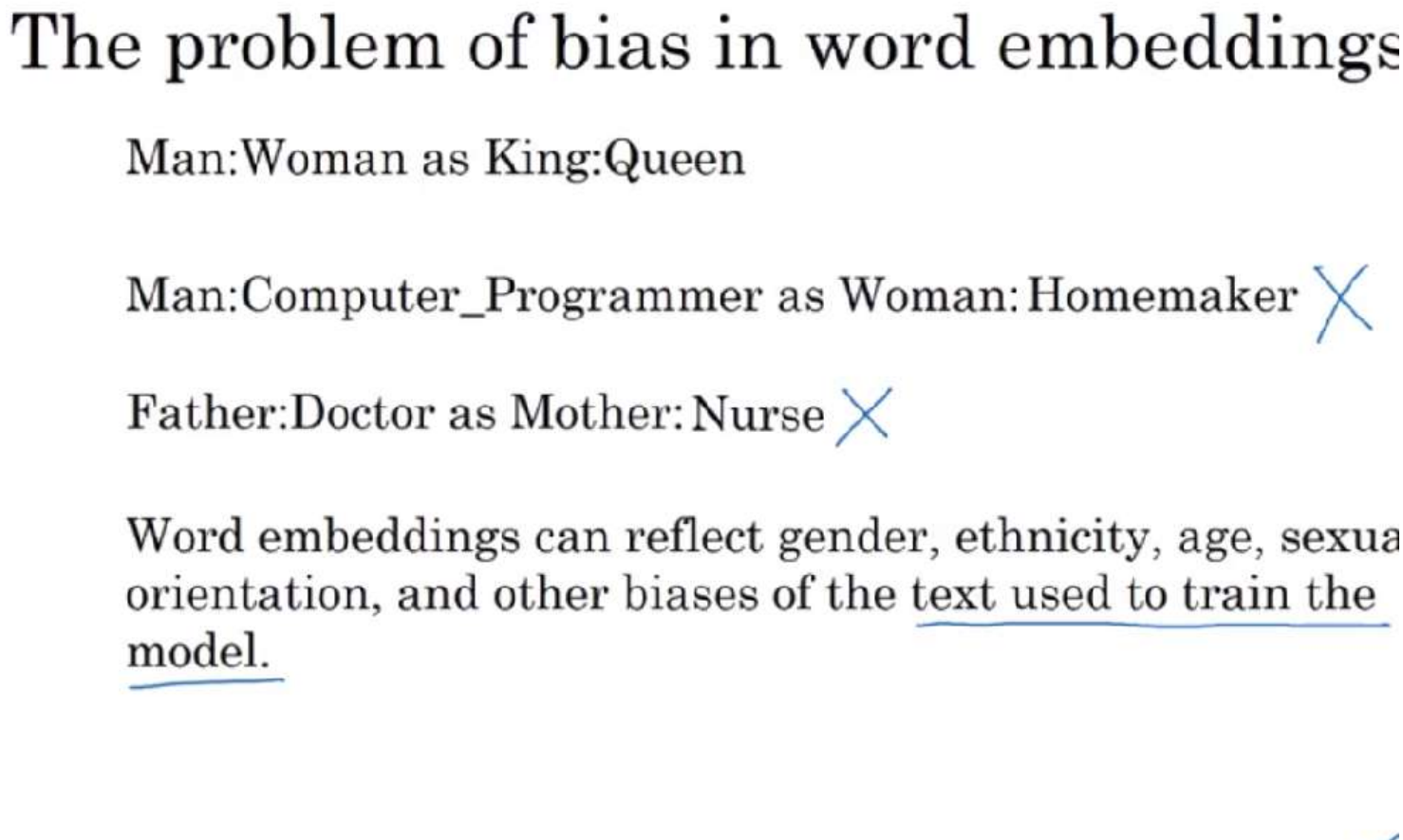
So here's a more sophisticated model which is that, instead of just summing all of your word embeddings, you can instead use a RNN for sentiment classification. So here's what you can do. You can take that review, "Completely lacking in good taste, good service, and good ambiance", and find for each of them, the one-hot vector and so I'm going to just skip the one-hot vector representation but take the one-hot vectors, multiply it by the embedding matrix E as usual, then this gives you the embedding vectors and then you can feed these into an RNN and the job of the RNN is to then compute the representation at the last time step that allows you to predict Y-hat.



So this is an example of a many-to-one RNN architecture which we saw in the previous section and with an algorithm like this, it will be much better at taking word sequence into account and realize that "things are lacking in good taste" is a negative review and "not good" a negative review unlike the previous algorithm, which just sums everything together into a big-word vector and doesn't realize that "not good" has a very different meaning than the words "good" or "lacking in good taste" and so on and so if you train this algorithm, you end up with a pretty decent sentiment classification algorithm and because your word embeddings can be trained from a much larger data set, this will do a better job generalizing to maybe even new words now that you'll see in your training set, such as if someone else says, "Completely absent of good taste, good service, and good ambiance" or something, then even if the word "absent" is not in your label training set, if it was in your 1 billion or 100 billion word corpus used to train the word embeddings, it might still get this right and generalize much better even to words that were in the training set used to train the word embeddings but not necessarily in the label training set that you had for specifically the sentiment classification problem. So that's it for sentiment classification, and I hope this gives you a sense of how once you've learned or downloaded from online a word embedding, this allows you to quite quickly build pretty effective NLP systems.

Debiasing word embeddings

Machine learning and AI algorithms are increasingly trusted to help with, or to make, extremely important decisions and so we like to make sure that as much as possible that they're free of undesirable forms of bias, such as gender bias, ethnicity bias and so on. What I want to do in this section is show you some of the ideas for diminishing or eliminating these forms of bias in word embeddings. When I use the term bias in this section, I don't mean the bias variants. Sense the bias, instead I mean gender, ethnicity, sexual orientation bias. That's a different sense of bias then is typically used in the technical discussion on machine learning. But mostly the problem, we talked about how word embeddings can learn analogies like man is to woman as king is to queen. But what if you ask it, man is to computer programmer as woman is to what? And so the authors of this paper Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai found a somewhat horrifying result where a learned word embedding might output Man:Computer_Programmer as Woman:Homemaker and that just seems wrong and it enforces a very unhealthy gender stereotype. It'd be much more preferable to have algorithm output man is to computer programmer as a woman is to computer programmer and they found also, Father:Doctor as Mother is to what? and the really unfortunate result is that some learned word embeddings would output Mother:Nurse.

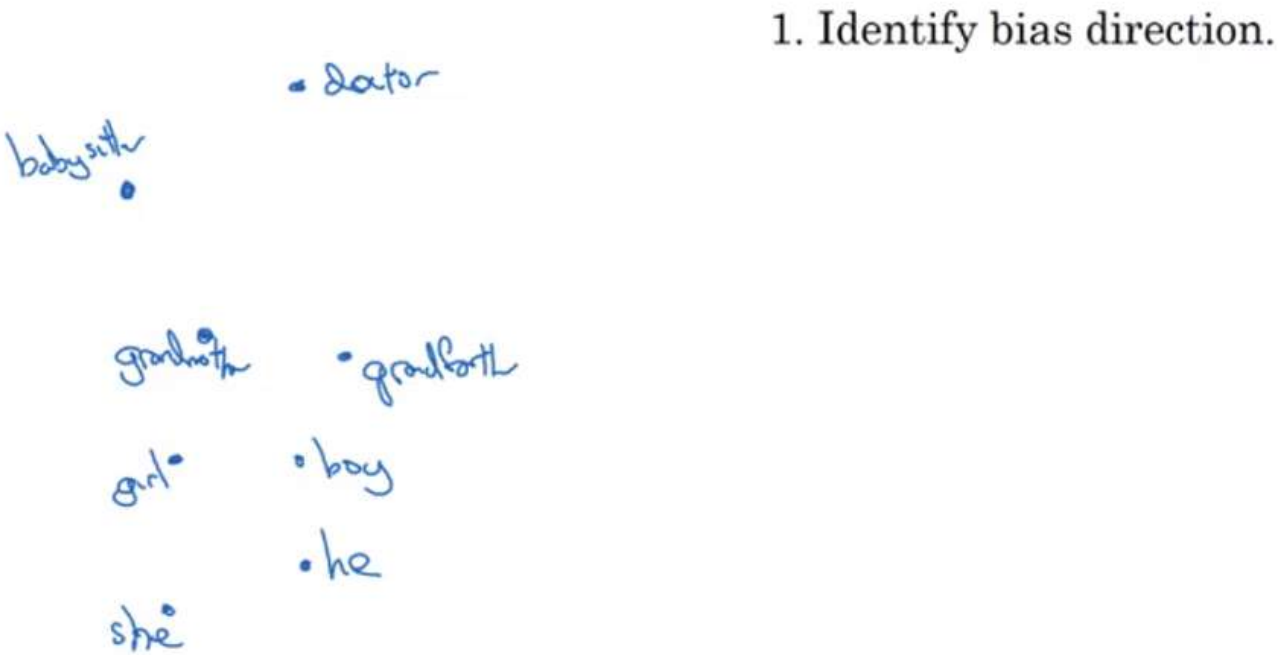


Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings]

So word embeddings can reflect the gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. One that I'm especially passionate about is bias relating to socioeconomic status. I think that every person, whether you come from a wealthy family, or a low income family, or anywhere in between, I think everyone should have great opportunities and because machine learning algorithms are being used to make very important decisions. They're influencing everything ranging from college admissions, to the way people find jobs, to loan applications, whether your application for a loan gets approved, to in the criminal justice system, even sentencing guidelines. Learning algorithms are making very important decisions and so I think it's important that we try to change learning algorithms to diminish as much as is possible, or, ideally, eliminate these types of undesirable biases. Now in the case of word embeddings, they can pick up the biases of the text used to train the model and so the biases they pick up or tend to reflect the biases in text as is written by people. Over many decades, over many centuries, I think humanity has made progress in reducing these types of bias and I think maybe fortunately for AI, I think we actually have better ideas for quickly reducing the bias in AI than for quickly reducing the bias in the human race. Although I think we're by no means done for AI as well and there's still a lot of research and hard work to be done to reduce these types of biases in our learning algorithms. But what I want to do in this video is share with you one example of a set of ideas due to the paper referenced at the bottom by Bolukbasi and others on reducing the bias in word embeddings.

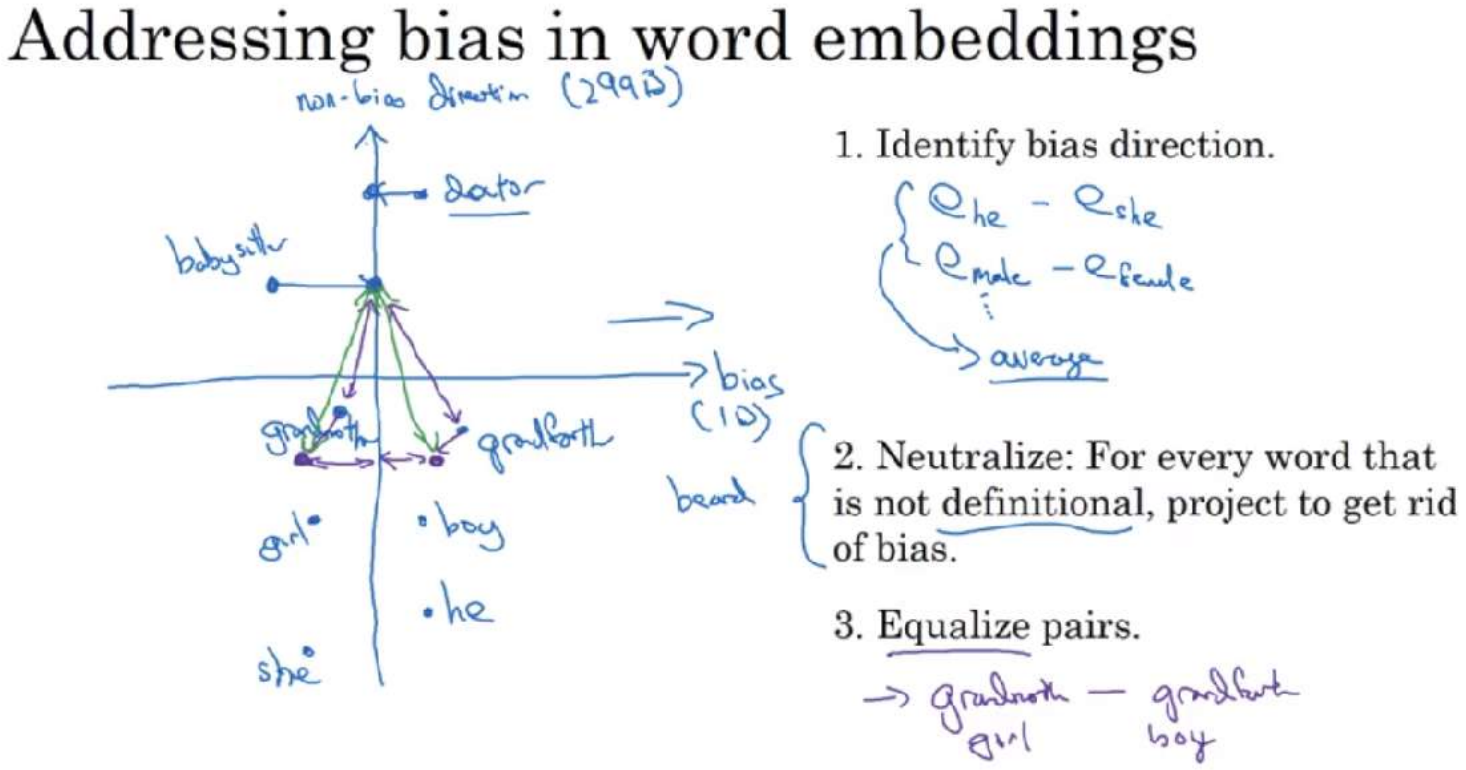
So here's the idea. Let's say that we've already learned a word embedding, so the word babysitter is here, the word doctor is here. We have grandmother here, and grandfather here. Maybe the word girl is embedded there, the word boy is embedded there and maybe she is embedded here, and he is embedded there. check diagram:

Addressing bias in word embeddings



So the first thing we're going to do it is identify the direction corresponding to a particular bias we want to reduce or eliminate and, for illustration, we'll focus on gender bias but these ideas are applicable to all of the other types of bias that we mention on the previous section as well and so how do you identify the direction corresponding to the bias? For the case of gender, what we can do is take the embedding vector for he and subtract the embedding vector for she, because that differs by gender and take e male, subtract e female, and take a few of these and average them, right? And take a few of these differences and basically average them and this will allow you to figure out in this case that what looks like this direction is the gender direction, or the bias direction. Whereas this direction is unrelated to the particular bias we're trying to address. So this is the non-bias direction. And in this case, the bias direction, think of this as a 1D subspace whereas a non-bias direction, this will be 299-dimensional subspace. Okay, and I've simplified the description a little bit in the original paper. The bias direction can be higher than 1-dimensional, and rather than take an average, as I'm describing it here, it's actually found using a more complicated algorithm called a SVU, a **singular value decomposition**. Which is closely related to, if you're familiar with **principle component analysis**, it uses ideas similar to the **PCA algorithm**. After that, the next step is a neutralization step. So for every word that's not definitional, project it to get rid of bias. So there are some words that intrinsically capture gender. So words like grandmother, grandfather, girl, boy, she, he, a gender is intrinsic in the definition. Whereas there are other word like doctor and babysitter that we want to be gender neutral and really, in the more general case, you might want words like doctor or babysitter to be ethnicity neutral or sexual orientation neutral, and so on, but we'll just use gender as the illustrating example here. But so for every word that is not definitional, this basically means not words like grandmother and grandfather, which really have a very legitimate gender component, because, by definition, grandmothers are female, and grandfathers are male. So for words like doctor and babysitter, let's just project them onto this axis to reduce their components, or to eliminate their component, in the bias direction. So reduce their component in this horizontal direction.
So that's the second neutralize step and then the final step is called equalization in which you might have pairs of words such as grandmother and grandfather, or girl and boy, where you want the only difference in their embedding to be the gender and so, why do you want that? Well in this example, the distance, or the similarity, between babysitter and grandmother is actually smaller than the distance between babysitter and grandfather and so this maybe reinforces an unhealthy, or maybe undesirable, bias that grandmothers end up babysitting more than grandfathers. So in the final equalization step, what we'd like to do is to make sure that words like grandmother and grandfather are both exactly the same similarity, or exactly the same distance, from words that should be gender

neutral, such as babysitter or such as doctor. So there are a few linear algebra steps for that. But what it will basically do is move grandmother and grandfather to a pair of points that are equidistant from this axis in the middle and so the effect of that is that now the distance between babysitter, compared to these two words, will be exactly the same and so, in general, there are many pairs of words like this grandmother-grandfather, boy-girl, sorority-fraternity, girlhood-boyhood, sister-brother, niece-nephew, daughter-son, that you might want to carry out through this equalization step.



So the final detail is, how do you decide what word to neutralize? So for example, the word doctor seems like a word you should neutralize to make it non-gender-specific or non-ethnicity-specific. Whereas the words grandmother and grandmother should not be made non-gender-specific and there are also words like beard, right, that it's just a statistical fact that men are much more likely to have beards than women, so maybe beards should be closer to male than female. And so what the authors did is train a classifier to try to figure out what words are definitional, what words should be gender-specific and what words should not be and it turns out that most words in the English language are not definitional, meaning that gender is not part of the definition and it's such a relatively small subset of words like this, grandmother-grandfather, girl-boy, sorority-fraternity, and so on that should not be neutralized and so a linear classifier can tell you what words to pass through the neutralization step to project out this bias direction, to project it on to this essentially 299-dimensional subspace and then, finally, the number of pairs you want to equalize, that's actually also relatively small, and is, at least for the gender example, it is quite feasible to hand-pick most of the pairs you want to equalize. So the full algorithm is a bit more complicated than we have seen here, we can take a look at the paper for the full details.

So to summarize, that reducing or eliminating bias of our learning algorithms is a very important problem because these algorithms are being asked to help with or to make more and more important decisions in society. In this section we have seen just one set of ideas for how to go about trying to address this problem, but this is still a very much an ongoing area of active research by many researchers.

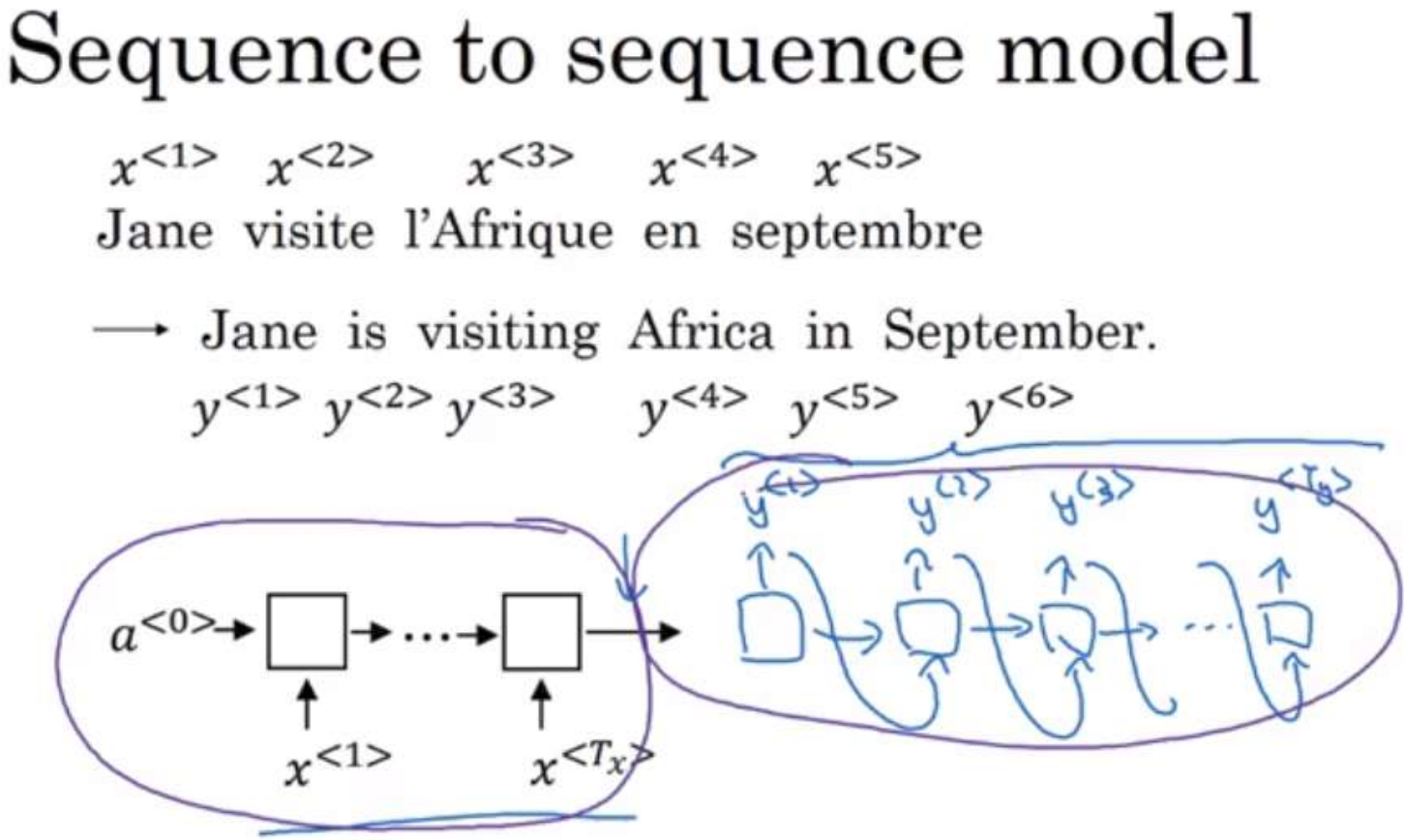
Week 3: Sequence models & Attention mechanism

Sequence models can be augmented using an attention mechanism. This algorithm will help your model understand where it should focus its attention given a sequence of inputs. This week, you will also learn about speech recognition and how to deal with audio data.

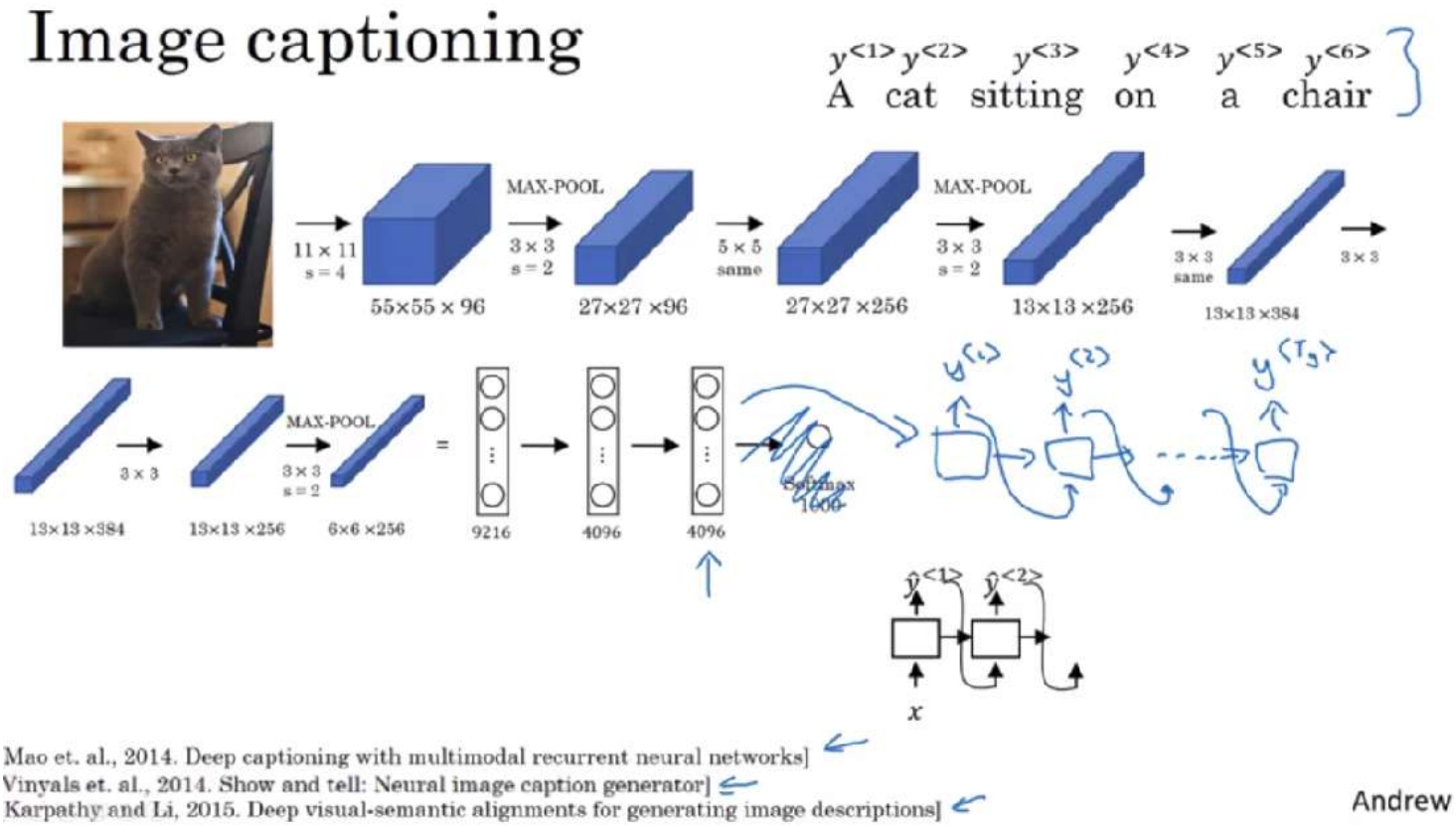
Various sequence to sequence architectures
Object Localization
Basic Models

In this week, you hear about sequence-to-sequence models, which are useful for everything from machine translation to speech recognition. Let's start with the basic models and then later this week you, hear about beam search, the attention model, and we'll wrap up the discussion of models for audio data, like speech. Let's get started.

Let's say you want to input a French sentence like Jane visite l'Afrique en septembre, and you want to translate it to the English sentence, Jane is visiting Africa in September. As usual, let's use $x^{<1>}$ through x, in this case $x^{<5>}$, to represent the words in the input sequence, and we'll use $y^{<1>}$ through $y^{<6>}$ to represent the words in the output sequence. So, how can you train a new network to input the sequence x and output the sequence y? Well, here's something you could do, and the ideas I'm about to present are mainly from these two papers due to Sutskever, Oriol Vinyals, and Quoc Le, and that one by Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwen, and Yoshua Bengio.



First, let's have a network, which we're going to call the encoder network be built as a RNN, and this could be a GRU and LSTM, feed in the input French words one word at a time. And after ingesting the input sequence, the RNN then offers a vector that represents the input sentence. After that, you can build a decoder network which I'm going to draw here, which takes as input the encoding output by the encoder network shown in black on the left, and then can be trained to output the translation one word at a time until eventually it outputs say, the end of sequence or end the sentence token upon which the decoder stops and as usual we could take the generated tokens and feed them to the next [inaudible] in the sequence like we 're doing before when synthesizing text using the language model. One of the most remarkable recent results in deep learning is that this model works, given enough pairs of French and English sentences. If you train the model to input a French sentence and output the corresponding English translation, this will actually work decently well and this model simply uses an encoder network, whose job it is to find an encoding of the input French sentence and then use a decoder network to then generate the corresponding English translation.



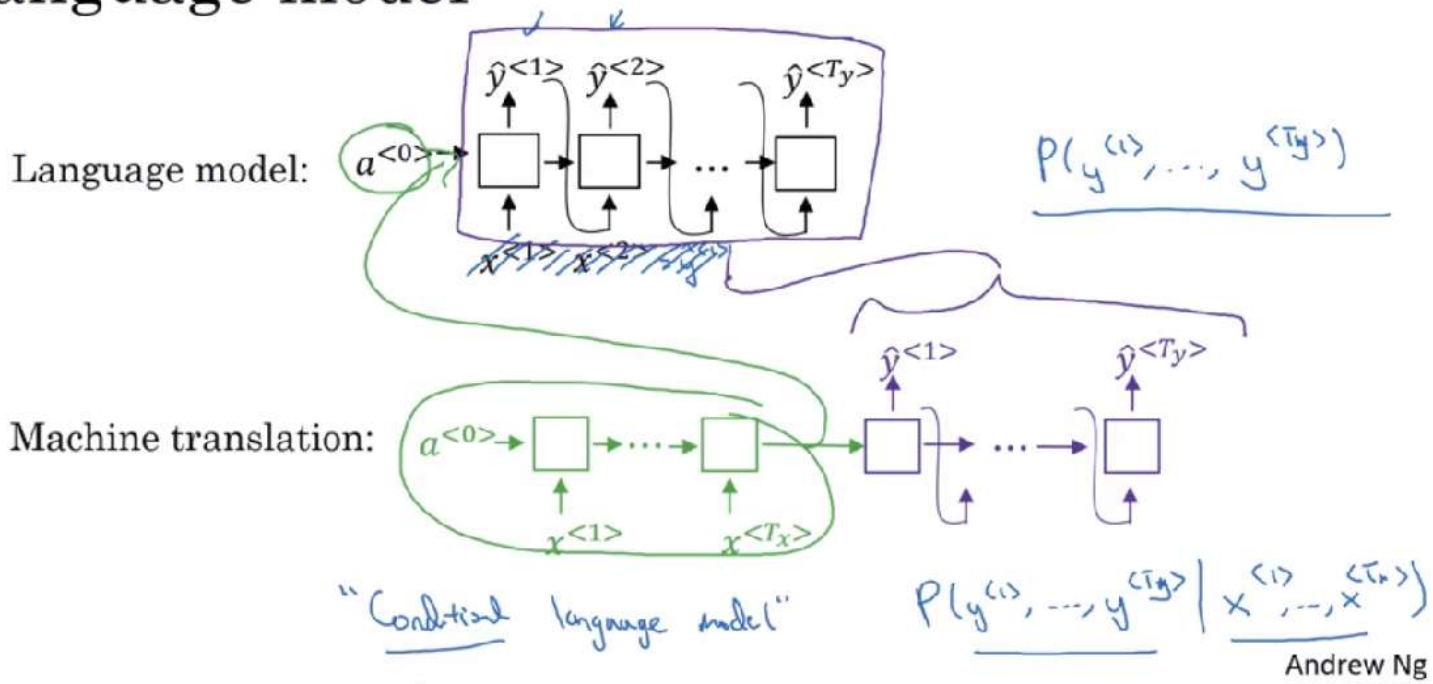
An architecture very similar to this also works for image captioning so given an image like the one shown here, maybe wanted to be captioned automatically as a cat sitting on a chair. So how do you train a new network to input an image and output a caption like that phrase up there? Here's what you can do. From the earlier course on [inaudible] you've seen how you can input an image into a convolutional network, maybe a pre-trained AlexNet, and have that learn an encoding or learn a set of features of the input image. So, this is actually the AlexNet architecture and if we get rid of this final Softmax unit, the pre-trained AlexNet can give you a 4096-dimensional feature vector of which to represent this picture of a cat. And so this pre-trained network can be the encoder network for the image and you now have a 4096-dimensional vector that represents the image. You can then take this and feed it to an RNN, whose job it is to generate the caption one word at a time. So similar to what we saw with machine translation translating from French to English, you can now input a feature vector describing the input and then have it generate an output sequence or output set of words one word at a time and this actually works pretty well for image captioning, especially if the caption you want to generate is not too long. As far as I know, this type of model was first proposed by Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan Yuille, although it turns out there were multiple groups coming up with very similar models independently and at about the same time. So two other groups that had done very similar work at about the same time and I think independently of Mao et al were Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan, as well as Andrej Karpathy and Fei-Fei Yi. So, you've now seen how a basic sequence-to-sequence model works, or how a basic image-to-sequence or image captioning model works, but there are some differences between how you would run a model like this, so generating a sequence compared to how you were synthesizing novel text using a language model. One of the key differences is, you don't want a randomly chosen translation, you maybe want the most likely translation, or you don't want a randomly chosen caption, maybe not, but you might want the best caption and most likely caption. So let's see in the next section how you go about generating that.

Picking the most likely sequence

There are some similarities between the sequence to sequence machine translation model and the language models that you have worked within the first week of this course, but there are some significant differences as well. Let's take a look. So, you can think of machine translation as building a conditional language model. Here's what I mean, in language modeling, this was the network we had

built in the first week and this model allows you to estimate the probability of a sentence. That's what a language model does and you can also use this to generate novel sentences, and sometimes when you are writing x1 and x2 here, where in this example, x2 would be equal to y1 or equal to y and one is just a feedback. But x1, x2, and so on were not important. So just to clean this up for this slide, I'm going to just cross these off. X1 could be the vector of all zeros and x2, x3 are just the previous output you are generating. So that was the language model.

Machine translation as building a conditional language model



The machine translation model looks as follows, and I am going to use a couple different colors, green and purple, to denote respectively the coded network in green and the decoded network in purple and you notice that the decoded network looks pretty much identical to the language model that we had up there. So what the machine translation model is, is very similar to the language model, except that instead of always starting along with the vector of all zeros, it instead has an encoded network that figures out some representation for the input sentence, and it takes that input sentence and starts off the decoded network with representation of the input sentence rather than with the representation of all zeros. So, that's why we call this a conditional language model, and instead of modeling the probability of any sentence, it is now modeling the probability of, say, the output English translation, conditions on some input French sentence. So in other words, you're trying to estimate the probability of an English translation. Like, what's the chance that the translation is "**Jane is visiting Africa in September**," but conditions on the input French censors like, "**Jane visite I'Afrique en septembre**." So, this is really the probability of an English sentence conditions on an input French sentence which is why it is a conditional language model. Now, if you want to apply this model to actually translate a sentence from French into English, given this input French sentence, the model might tell you what is the probability of difference in corresponding English translations. So, x is the French sentence, "Jane visite l'Afrique en septembre." And, this now tells you what is the probability of different English translations of that French input. And, what you do not want is to sample outputs at random. If you sample words from this distribution, p of y given x, maybe one time you get a pretty good translation, "Jane is visiting Africa in September." But, maybe another time you get a different translation, "Jane is going to be visiting Africa in September. " Which sounds a little awkward but is not a terrible translation, just not the best one. And sometimes, just by chance, you get, say, others: "In September, Jane will visit Africa." And maybe, just by chance, sometimes you sample a really bad translation: "Her African friend welcomed Jane in September."

Finding the most likely translation

Jane visite l'Afrique en septembre.

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

English

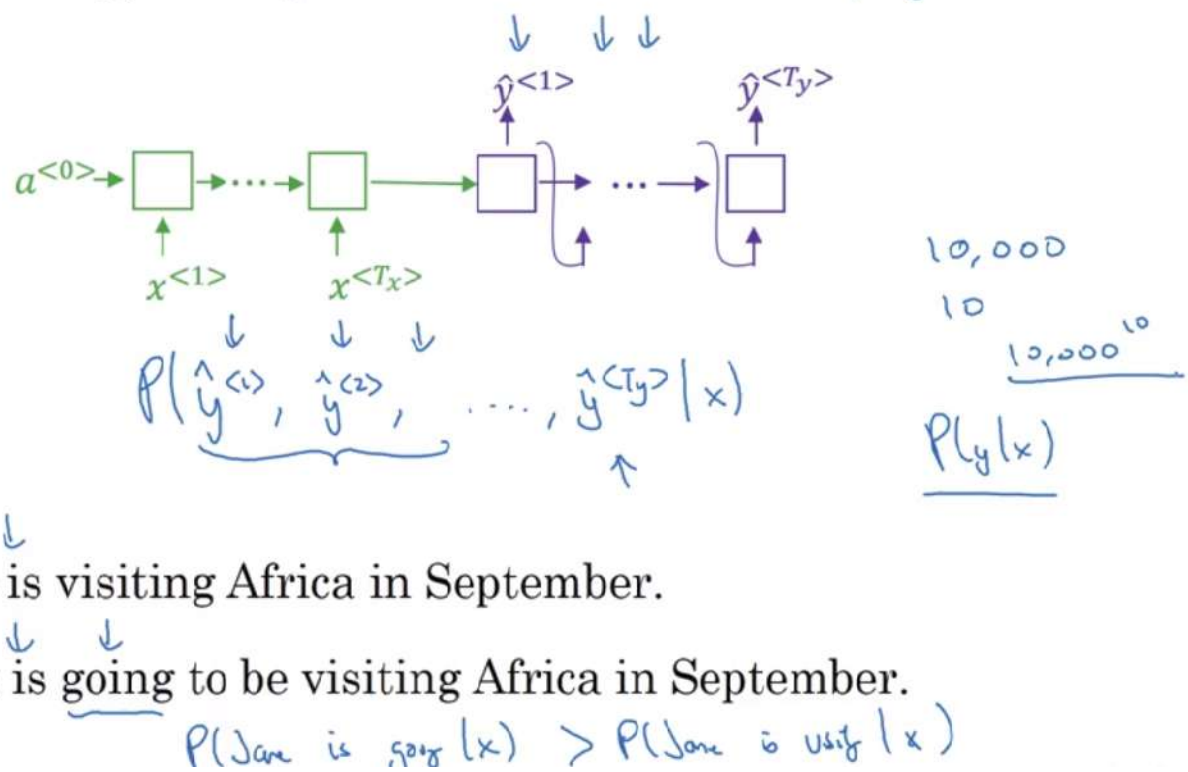
French

- → 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.

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

So, when you're using this model for machine translation, you're not trying to sample at random from this distribution. Instead, what you would like is to find the English sentence, y, that maximizes that conditional probability. So in developing a machine translation system, one of the things you need to do is come up with an algorithm that can actually find the value of y that maximizes this term over here. The most common algorithm for doing this is called **beam search**, and it's something you'll see in the next section. But, before moving on to describe beam search, you might wonder, why not just use **greedy search**? So, what is greedy search? Well, greedy search is an algorithm from computer science which says to generate the first word just pick whatever is the most likely first word according to your conditional language model. Going to your machine translation model and then after having picked the first word, you then pick whatever is the second word that seems most likely, then pick the third word that seems most likely. This algorithm is called greedy search.

Why not a greedy search?



→ Jane is visiting Africa in September.

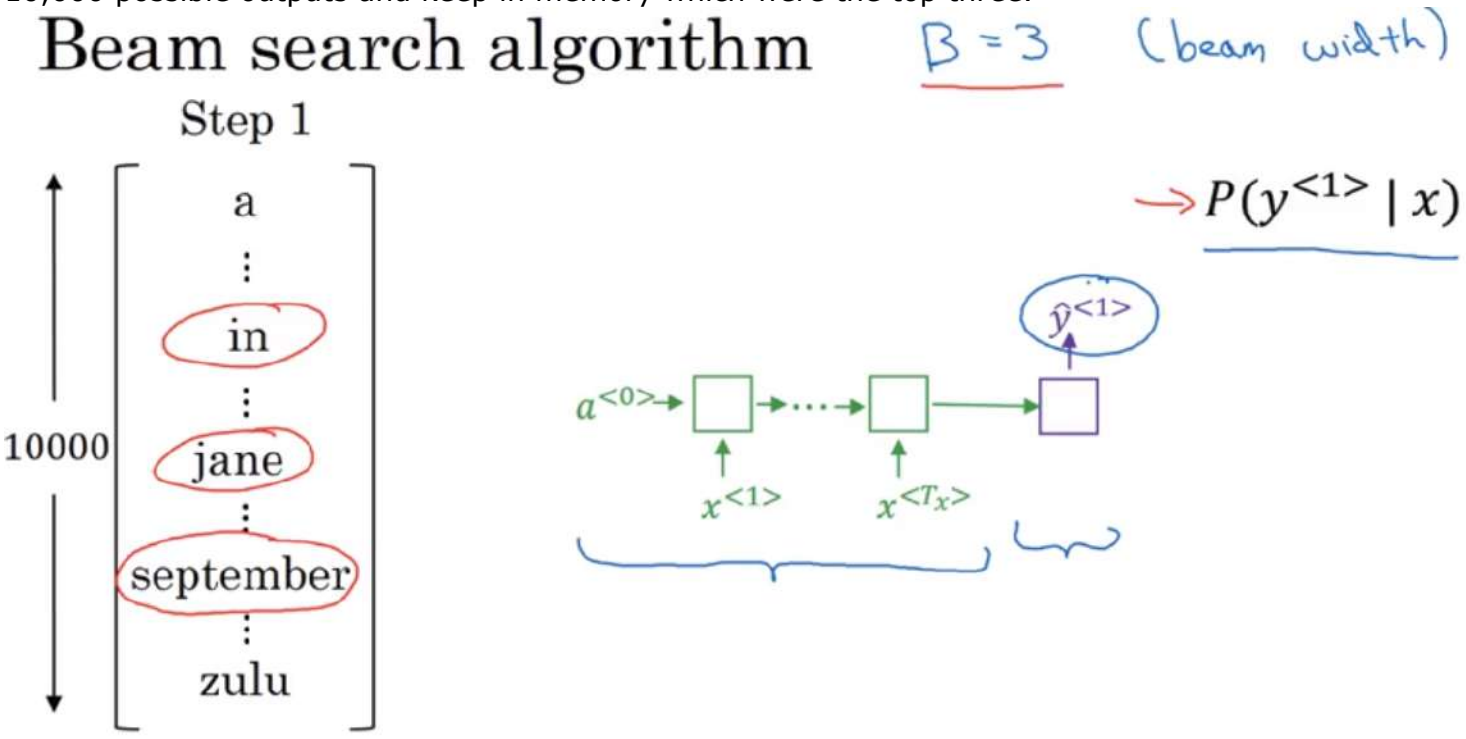
→ Jane is going to be visiting Africa in September.

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

And, what you would really like is to pick the entire sequence of words, y1, y2, up to yTy, that's there, that maximizes the joint probability of that whole thing and it turns out that the greedy approach, where you just pick the best first word, and then, after having picked the best first word, try to pick the best second word, and then, after that, try to pick the best third word, that approach doesn't really work. To demonstrate that, let's consider the following two translations. The first one is a better translation, so hopefully, in our machine translation model, it will say that p of y given x is higher for the first sentence. It's just a better, more succinct translation of the French input. The second one is not a bad translation, it's just more verbose, it has more unnecessary words. But, if the algorithm has picked "Jane is" as the first two words, because "going" is a more common English word, probably the chance of "Jane is going," given the French input, this might actually be higher than the chance of "Jane is visiting," given the French sentence. So, it's quite possible that if you just pick the third word based on whatever maximizes the probability of just the first three words, you end up choosing option number two. But, this ultimately ends up resulting in a less optimal sentence, in a less good sentence as measured by this model for p of y given x. I know this was may be a slightly hand-wavey argument, but, this is an example of a broader phenomenon, where if you want to find the sequence of words, y1, y2, all the way up to the final word that together maximize the probability, it's not always optimal to just pick one word at a time and, of course, the total number of combinations of words in the English sentence is exponentially larger. So, if you have just 10,000 words in a dictionary and if you're contemplating translations that are up to ten words long, then there are 10000 to the tenth possible sentences that are ten words long. Picking words from the vocabulary size, the dictionary size of 10000 words. So, this is just a huge space of possible sentences, and it's impossible to rate them all, which is why the most common thing to do is use an approximate search out of them and, what an approximate search algorithm does, is it will try, it won't always succeed, but it will to pick the sentence, y, that maximizes that conditional probability and, even though it's not guaranteed to find the value of y that maximizes this, it usually does a good enough job. So, to summarize, in this section, you saw how machine translation can be posed as a conditional language modeling problem. But one major difference between this and the earlier language modeling problems is rather than wanting to generate a sentence at random, you may want to try to find the most likely English sentence, most likely English translation. But the set of all English sentences of a certain length is too large to exhaustively enumerate. So, we have to resort to a search algorithm. So, with that, let's go onto the next section where you'll learn about beam search algorithm.

Beam Search

In this section, we'll learn about the beam search algorithm. In the last section, we remember how for machine translation given an input French sentence, we don't want to output a random English translation, we want to output the best and the most likely English translation. The same is also true for speech recognition where given an input audio clip, you don't want to output a random text transcript of that audio, you want to output the best, maybe the most likely, text transcript. **Beam search is the most widely used algorithm to do this** and in this section, we'll see how to get beam search to work for yourself. Let's just try Beam Search using our running example of the French sentence, "Jane, visite l'Afrique en Septembre". Hopefully being translated into, "Jane, visits Africa in September". The first thing Beam search has to do is try to pick the first words of the English translation, that's going to operate. So here I've listed, say, 10,000 words into vocabulary and to simplify the problem a bit, I'm going to ignore capitalization. So I'm just listing all the words in lower case. So, in the first step of Beam Search, I use this network fragment with the coalition in green and decoalition in purple, to try to evaluate what is the probability of that for a square. So, what's the probability of the first output y, given the input sentence x gives the French input. So, whereas greedy search will pick only the one most likely words and move on, Beam Search instead can consider multiple alternatives. So, the Beam Search algorithm has a parameter called B, which is called the beam width and for this example I'm going to set the beam width to be with the 3 and what this means is Beam search will cause that not just one possibility but consider three at the time. So in particular, let's say evaluating this probability over different choices the first words, it finds that the choices in, Jane and September are the most likely three possibilities for the first words in the English outputs. Then Beam search will store away in computer memory that it wants to try all of three of these words, and if the beam width parameter were said differently, the beam width parameter was 10, then we keep track of not just three but of the ten, most likely possible choices for the first word. So, to be clear in order to perform this first step of Beam search, what you need to do is run the input French sentence through this encoder network and then this first step will then decode the network, this is a softmax output overall 10,000 possibilities. Then you would take those 10,000 possible outputs and keep in memory which were the top three.



Let's go into the second step of Beam search. Having picked in, Jane and September as the three most likely choice of the first word, what Beam search will do now, is for each of these three choices consider what should be the second word, so after "in" maybe a second word is "a" or maybe as Aaron, I'm just listing words from the vocabulary, from the dictionary or somewhere down the list will be September, somewhere down the list there's visit and then all the way to z and then the last word is zulu. So, to evaluate the probability of second word, it will use this new network fragments where

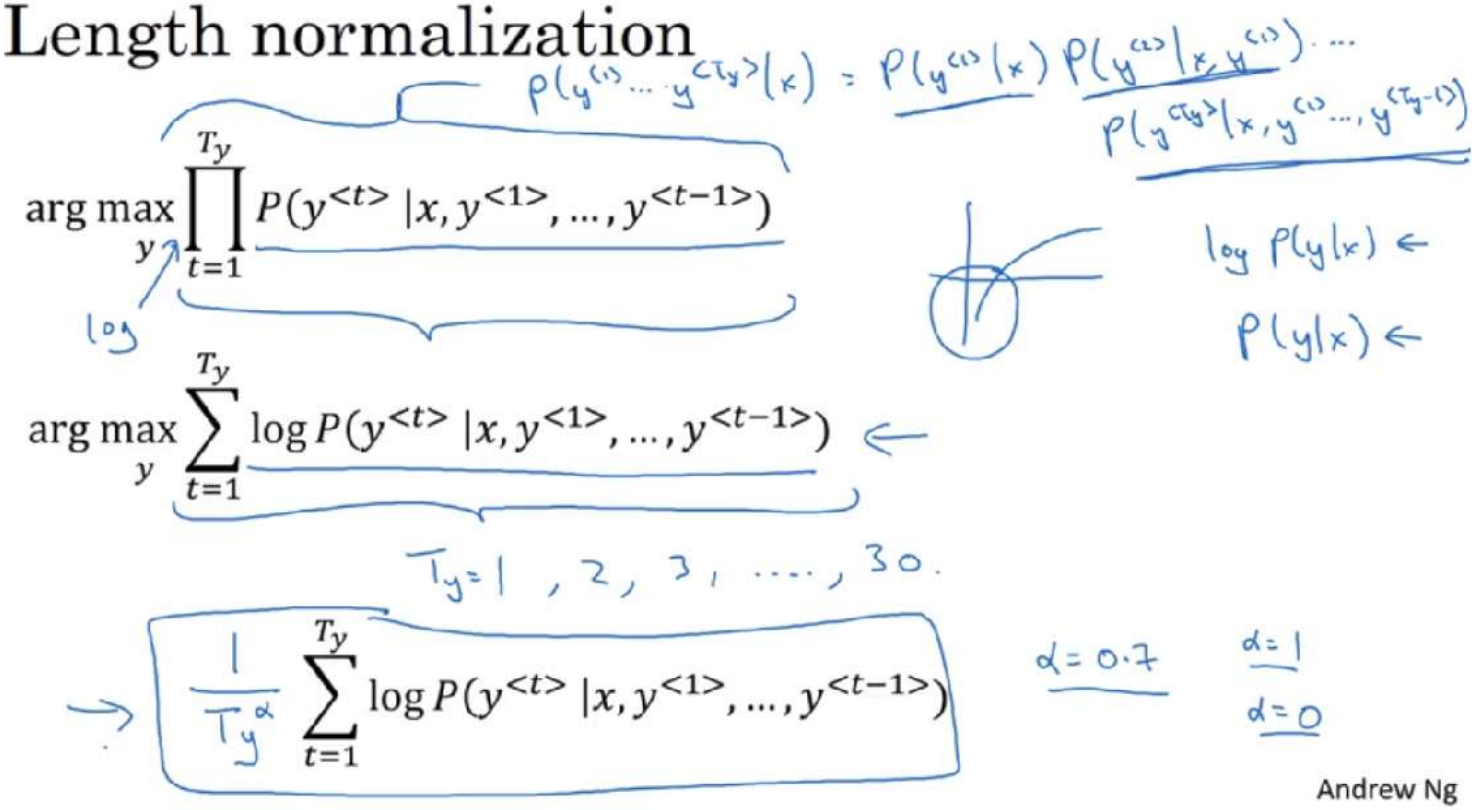
is coder in green and for the decoder portion when trying to decide what comes after in. Remember the decoder first outputs, yhat$^{<1>}$. So, I'm going to set to this y hat one to the word "in" as it goes back in. So there's the word **"in"** because it decided for now. That's because It trying to figure out that the first word was "in", what is the second word, and then this will output I guess yhat$^{<2>}$ and so by hard wiring y hat one here, really the inputs here to be the first words "in" this time were fragment can be used to evaluate whether it's the probability of the second word given the input french sentence and that the first words of the translation has been the word "in". Now notice that what we also need help out in this second step would be assertions to find the pair of the first and second words that is most likely it's not just a second where is most likely that the pair of the first and second whereas the most likely and by the rules of conditional probability. This can be expressed as P of the first words times P of probability of the second words. Which you are getting from this network fragment and so if for each of the three words you've chosen "in", "Jane," and "September" you save away this probability then you can multiply them by this second probabilities to get the probability of the first and second words. So now you've seen how if the first word was "in" how you can evaluate the probability of the second word. Now at first it was "Jane" you do the same thing. The sentence could be "Jane a"," Jane Aaron", and so on down to "Jane is", "Jane visits" and so on. And you will use this in your network fragments let me draw this in as well where here you will hardwire, yhat$^{<1>}$ to be Jane. And so with the First word yhat$^{<1>}$'s hard wired as Jane than just the network fragments can tell you what's the probability of the second words to me. And given that the first word is "Jane". And then same as above you can multiply with P(y$^{<1>}$) to get the probability of Y1 and Y2 for each of these 10,000 different possible choices for the second word. And then finally do the same thing for September although words from a down to Zulu and use this network fragment. That just goes in as well to see if the first word was September. What was the most likely options for the second words. So for this second step of beam search because we're continuing to use a beam width of three and because there are 10,000 words in the vocabulary you'd end up considering three times 10000 or thirty thousand possibilities because there are 10,000 here, 10,000 here, 10,000 here as the beam width times the number of words in the vocabulary and what you do is you evaluate all of these 30000 options according to the probably the first and second words and then pick the top three. So with a cut down, these **30,000 possibilities down to three again down the beam** width rounded again so let's say that 30,000 choices, the most likely were in September and say Jane is, and Jane visits sorry this bit messy but those are the most likely three out of the 30,000 choices then that's what Beam's search would memorize away and take on to the next step being surge. So notice one thing if beam search decides that the most likely choices are the first and second words are in September, or Jane is, or Jane visits. Then what that means is that it is now rejecting September as a candidate for the first words of the output English translation so we're now down to two possibilities for the first words but we still have a beam width of three keeping track of three choices for pairs of y$^{<1>}$, y$^{<2>}$before going onto the third step of beam search. Just want to notice that because of beam width is equal to three, every step you instantiate three copies of the network to evaluate these partial sentence fragments and the output and it's because of beam width is equal to three that you have three copies of the network with different choices for the first words, but these three copies of the network can be very efficiently used to evaluate all 30,000 options for the second word. So just don't instantiate 30,000 copies of the network or three copies of the network to very quickly evaluate all 10,000 possible outputs at that softmax output say for Y2. Let's just quickly illustrate one more step of beam search. So said that the most likely choices for first two words were in September, Jane is, and Jane visits and for each of these pairs of words which we should have saved the way in computer memory the probability of y$^{<1>}$, y$^{<2>}$ given the input X given the French sentence X. So similar to before, we now want to consider what is the third word. So in September a? In September Aaron? All the way down to is in September Zulu and to evaluate possible choices for the third word, you use this network fragments where you Hardwire the first word here to be in the second word to be September and so this network fragment allows you to evaluate what's the probability of the third word given the input French sentence X and given that the first two words are in September and English output and then you do the same thing for the second fragment. So like so and same thing for Jane visits and so beam search will then once again pick the top three possibilities may be that things in September. Jane is a likely outcome or Jane is visiting is likely or maybe Jane visits Africa is likely for that first three words and then it keeps going and then you go onto the fourth step of beam

search hat one more word and on it goes and the outcome of this process hopefully will be that adding one word at a time that Beam search will decide that. Jane visits Africa in September will be terminated by the end of sentence symbol using that system is quite common. They'll find that this is a likely output English sentence and you'll see more details of this yourself. So with a beam of three being searched considers three possibilities at a time. Notice **that if the beam width was said to be equal to one, say cause there's only one, then this essentially becomes the greedy search algorithm which we had discussed in the last section but by considering multiple possibilities say three or ten or some other number at the same time beam search will usually find a much better output sentence than greedy search**. You've now seen how Beam Search works but it turns out there's some additional tips and tricks from our partners that help you to make beam search work even better. Let's go onto the next section to take a look.

Refinements to Beam Search

In the last section, you saw the basic beam search algorithm. In this section, we'll learn some little changes that make it work even better. Length normalization is a small change to the beam search algorithm that can help you get much better results. Here's what it is. Beam search is maximizing this probability and this product here is just expressing the observation that $P(y^{<1>})$ up to $P(y^{<Ty>})$, given x, can be expressed as $P(y^{<1>})$ given x times $P(y^{<2>})$, given x and y1 times dot dot dot, up to I guess p of y Ty given x and y1 up to y t1-1. Maybe this notation is a bit more scary and more intimidating than it needs to be, but this is that probabilities that you see previously. Now, if you're implementing these, these probabilities are all numbers less than 1. Often they're much less than 1 and multiplying a lot of numbers less than 1 will result in a tiny, tiny, tiny number, which can result in numerical underflow. Meaning that it's too small for the floating part representation in your computer to store accurately. So in practice, instead of maximizing this product, we will take logs and if you insert a log there, then log of a product becomes a sum of a log, and maximizing this sum of log probabilities should give you the same results in terms of selecting the most likely sentence y. So by taking logs, you end up with a more numerically stable algorithm that is less prone to rounding errors, numerical rounding errors, or to really numerical underflow and because the log function, that's the logarithmic function, this is strictly monotonically increasing function, maximizing P(y) and because the logarithmic function, here's the log function, is a strictly monotonically increasing function, we know that maximizing log P(y) given x should give you the same result as maximizing P(y) given x. As in the same value of y that maximizes this should also maximize that. So in most implementations, you keep track of the sum of logs of the probabilities rather than the protocol of probabilities. Now, there's one other change to this objective function that makes the machine translation algorithm work even better. Which is that, if you referred to this original objective up here, if you have a very long sentence, the probability of that sentence is going to be low, because you're multiplying as many terms here. Lots of numbers are less than 1 to estimate the probability of that sentence. And so if you multiply all the numbers that are less than 1 together, you just tend to end up with a smaller probability and so this objective function has an undesirable effect, that maybe it unnaturally tends to prefer very short translations. It tends to prefer very short outputs. Because the probability of a short sentence is determined just by multiplying fewer of these numbers are less than 1 and so the product would just be not quite as small and by the way, the same thing is true for this. The log of our probability is always less than or equal to 1. You're actually in this range of the log. So the more terms you have together, the more negative this thing becomes. So there's one other change to the algorithm that makes it work better, which is instead of using this as the objective you're trying to maximize, one thing you could do is normalize this by the number of words in your translation. And so this takes the average of the log of the probability of each word. And this significantly reduces the penalty for outputting longer translations. And in practice, as a heuristic instead of dividing by Ty, by the number of words in the output sentence, sometimes you use a softer approach. We have Ty to the power of alpha, where maybe alpha is equal to 0.7. So if alpha was equal to 1, then yeah, completely normalizing by length. If alpha was equal to 0, then, well, Ty to the 0 would be 1, then you're just not normalizing at all. And this is somewhat in between full normalization, and no normalization, and alpha's another hyper parameter you have within that you can tune to try to get the best results and have to admit, using alpha this way, this is a heuristic or this is a hack. There isn't a great theoretical justification for it, but people have found this works well.

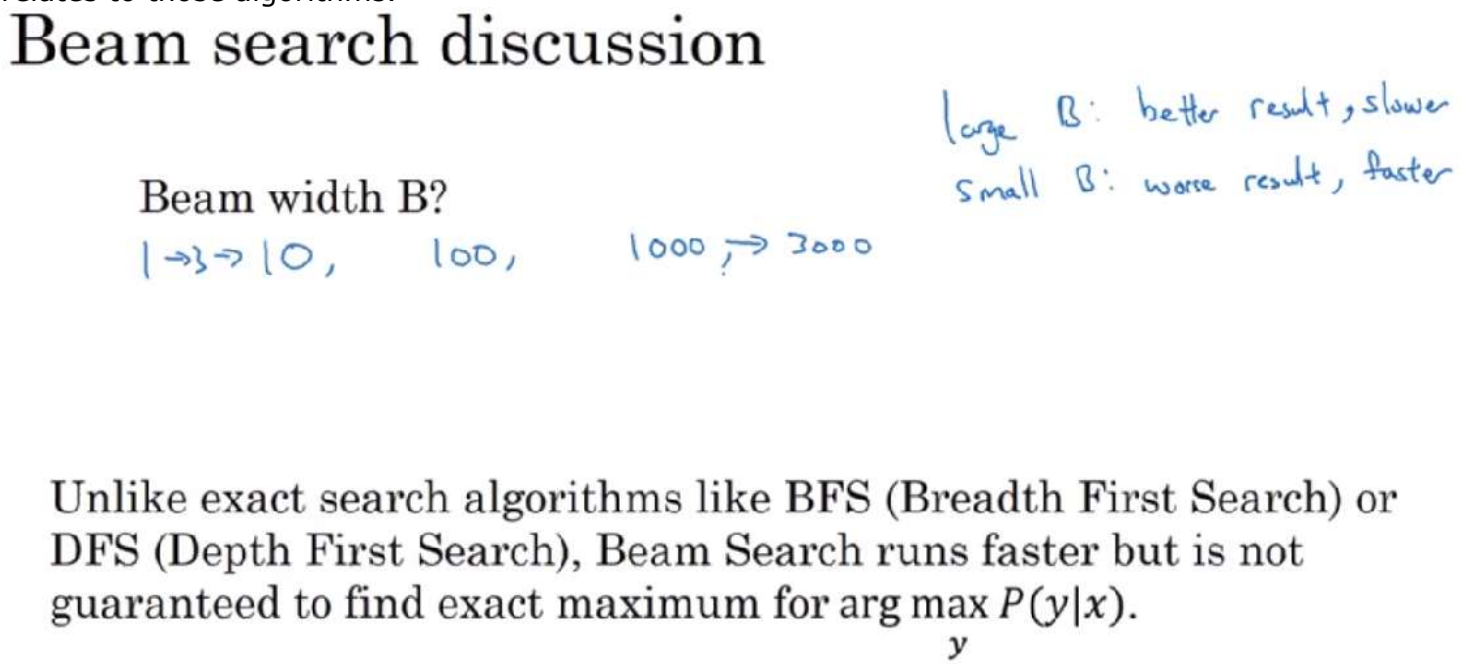
People have found that it works well in practice, so many groups will do this. And you can try out different values of alpha and see which one gives you the best result.



So just to wrap up how you run beam search, as you run beam search you see a lot of sentences with length equal 1, a lot of sentences with length equal 2, a lot of sentences with length equals 3 and so on, and maybe you run beam search for 30 steps and you consider output sentences up to length 30, let's say. And so with beam with a 3, you will be keeping track of the top three possibilities for each of these possible sentence lengths, 1, 2, 3, 4 and so on, up to 30. Then, you would look at all of the output sentences and score them against this score and so you can take your top sentences and just compute this objective function onto sentences that you have seen through the beam search process. And then finally, of all of these sentences that you validate this way, you pick the one that achieves the highest value on this normalized log probability objective. Sometimes it's called a normalized log likelihood objective and then that would be the final translation, your outputs.

Finally, a few implementational details, how do you choose the beam width B? The larger B is, the more possibilities you're considering, and does the better the sentence you probably find. But the larger B is, the more computationally expensive your algorithm is, because you're also keeping a lot more possibilities around. All right, so finally, let's just wrap up with some thoughts on how to choose the beam width B. So here are the pros and cons of setting B to be very large versus very small. If the beam width is very large, then you consider a lot of possibilities, and so you tend to get a better result because you are consuming a lot of different options, but it will be slower. And the memory requirements will also grow, will also be compositionally slower. Whereas if you use a very small beam width, then you get a worse result because you're just keeping less possibilities in mind as the algorithm is running. But you get a result faster and the memory requirements will also be lower. So in the previous section, we used in our running example a beam width of three, so we're keeping three possibilities in mind. In practice, that is on the small side. In production systems, it's not uncommon to see a beam width maybe around 10, and I think beam width of 100 would be considered very large for a production system, depending on the application. But for research systems where people want to squeeze out every last drop of performance in order to publish the paper with the best possible result. It's not uncommon to see people use beam widths of 1,000 or 3,000, but this is very application, that's why it's a domain dependent. So I would say try other variety of values of B as you work through your application. **But when B gets very large, there is often diminishing returns**. So for many applications, I would expect to see a huge gain as you go from a beam width of 1, which is very greedy search, to 3, to maybe 10. But the gains as you go from 1,000 to 3,000 in beam width might not be as big and for those of you that have taken maybe a

lot of computer science courses before, if you're familiar with computer science search algorithms like **BFS, Breadth First Search, or DFS, Depth First Search**. The way to think about beam search is that, unlike those other algorithms which you have learned about in a computer science algorithms course, and don't worry about it if you've not heard of these algorithms. But if you've heard of Breadth First Search and Depth First Search then unlike those algorithms, which are exact search algorithms. Beam search runs much faster but does not guarantee to find the exact maximum for this arg max that you would like to find. If you haven't heard of breadth first search or depth first search, don't worry about it, it's not important for our purposes. But if you have, this is how beam search relates to those algorithms.



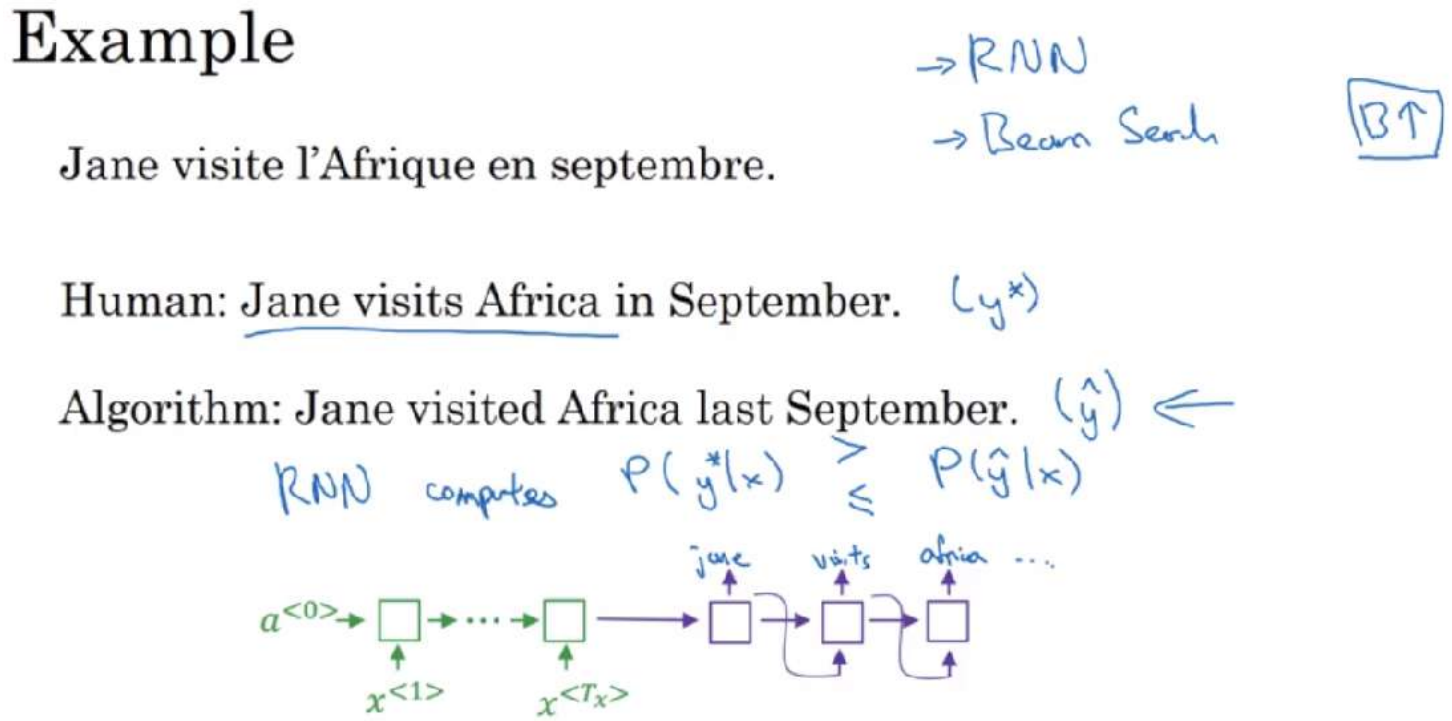
So that's it for beam search, which is a widely used algorithm in many production systems, or in many commercial systems. Now, in the circles in the sequence of courses of deep learning, we talked a lot about error analysis. It turns out, one of the most useful tools I've found is to be able to do error analysis on beam search. So you sometimes wonder, should I increase my beam width? Is my beam width working well enough? And there's some simple things you can compute to give you guidance on whether you need to work on improving your search algorithm. Let's talk about that in the next section.

Error analysis in beam search

In the third course of this sequence of five courses, we saw how error analysis can help you focus your time on doing the most useful work for your project. Now, beam search is an approximate search algorithm, also called a heuristic search algorithm and so it doesn't always output the most likely sentence. It's only keeping track of B equals 3 or 10 or 100 top possibilities. So what if beam search makes a mistake? In this section, we'll learn how error analysis interacts with beam search and how we can figure out whether it is the beam search algorithm that's causing problems and worth spending time on or whether it might be your RNN model that is causing problems and worth spending time on. Let's take a look at how to do error analysis with beam search.

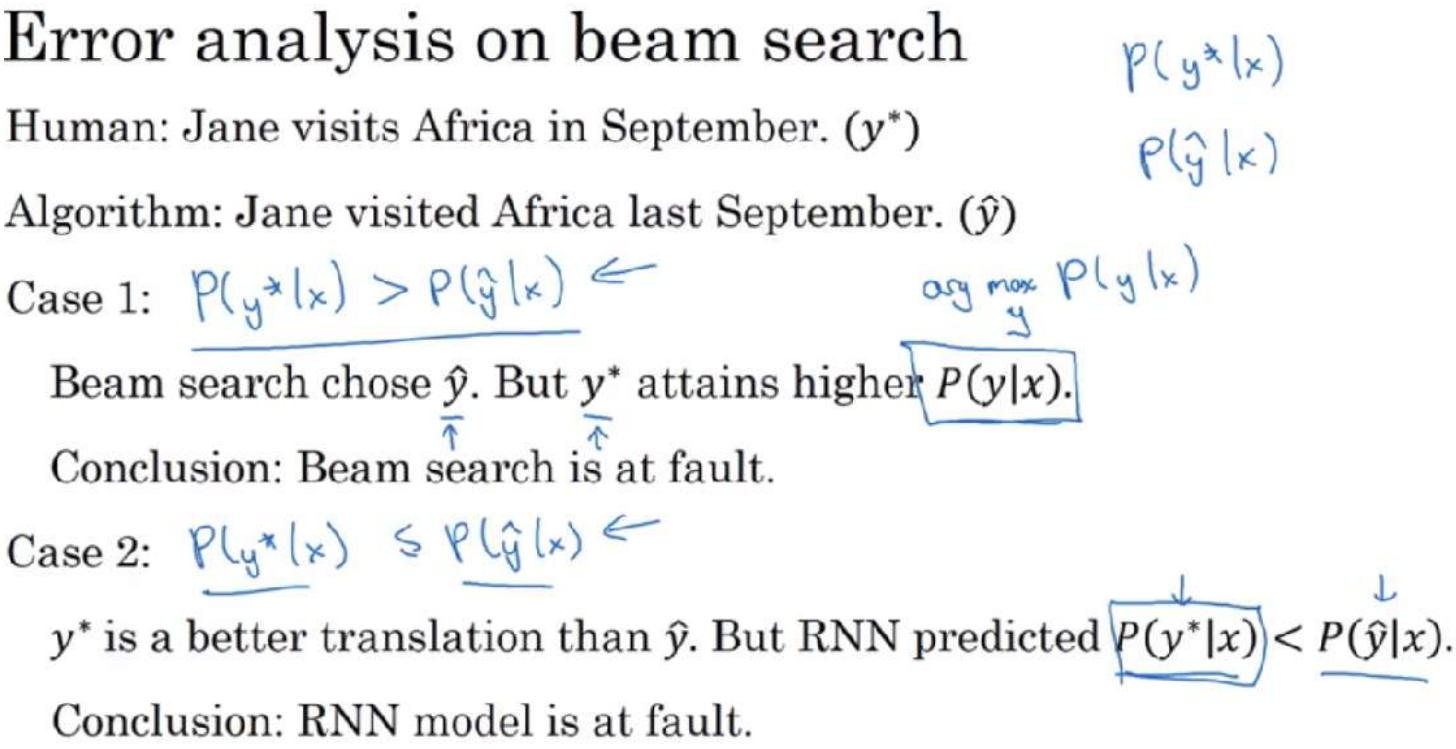
Let's use this example of Jane visite l'Afrique en septembre. So let's say that in your machine translation dev set, your development set, the human provided this translation and Jane visits Africa in September, and I'm going to call this y*. So it is a pretty good translation written by a human. Then let's say that when you run beam search on your learned RNN model and your learned translation model, it ends up with this translation, which we will call yhat, Jane visited Africa last September, which is a much worse translation of the French sentence. It actually changes the meaning, so it's not a good translation. Now, your model has two main components. There is a neural network model, the sequence to sequence model. We shall just call this your RNN model. It's really an encoder and a decoder and you have your beam search algorithm, which you're running with some beam width b. And wouldn't it be nice if you could attribute this error, this not very good translation, to one of these two components? Was it the RNN or really the neural network that is more to blame,

or is it the beam search algorithm, that is more to blame? And what you saw in the third course of the sequence is that it's always tempting to collect more training data that never hurts. So in similar way, it's always tempting to increase the beam width that never hurts or pretty much never hurts. But just as getting more training data by itself might not get you to the level of performance you want. In the same way, increasing the beam width by itself might not get you to where you want to go. But how do you decide whether or not improving the search algorithm is a good use of your time? So just how you can break the problem down and figure out what's actually a good use of your time. Now, the RNN, the neural network, what was called RNN really means the encoder and the decoder. It computes P(y given x). So for example, for a sentence, Jane visits Africa in September, you plug in Jane visits Africa. Again, I'm ignoring upper versus lowercase now, right, and so on and this computes P(y given x). So it turns out that the most useful thing for you to do at this point is to compute using this model to compute P(y* given x) as well as to compute P(yhat given x) using your RNN model and then to see which of these two is bigger. So it's possible that the left side is bigger than the right hand side. It's also possible that P(y*) is less than P(yhat) actually, or less than or equal to, right? Depending on which of these two cases hold true, you'd be able to more clearly describe this particular error, this particular bad translation to one of the RNN or the beam search algorithm being had greater fault. So let's take out the logic behind this.

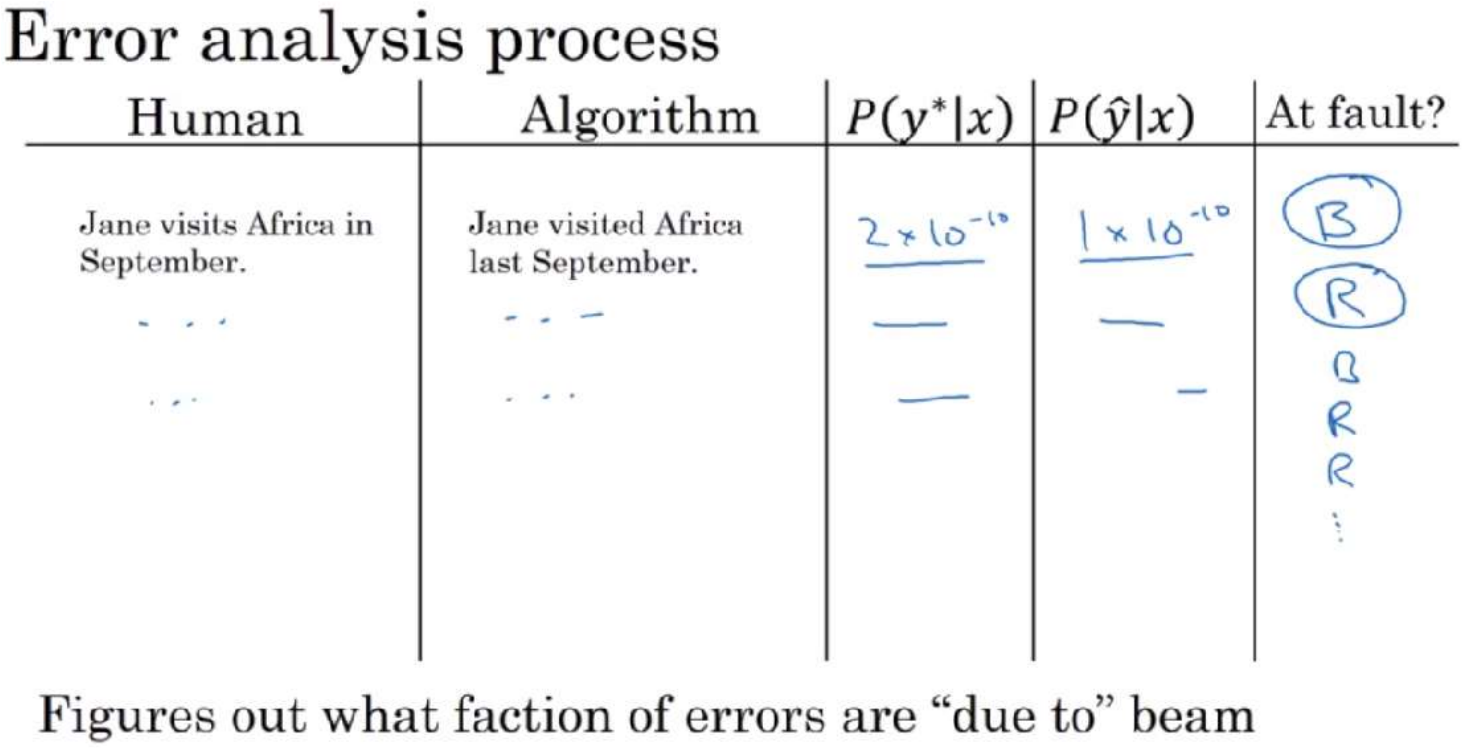


Here are the two sentences from the previous slide. And remember, we're going to compute P(y* given x) and P(y-hat given x) and see which of these two is bigger. So there are going to be two cases. In case 1, P(y* given x) as output by the RNN model is greater than P(y-hat given x). What does this mean? Well, the beam search algorithm chose yhat, right? The way you got y-hat was you had an RNN that was computing P(y given x) and beam search's job was to try to find a value of y that gives that arg max. But in this case, y* actually attains a higher value for P(y given x) than the y-hat. So what this allows you to conclude is beam search is failing to actually give you the value of y that maximizes P(y given x) because the one job that beam search had was to find the value of y that makes this really big. But it choose yhat, the y* actually gets a much bigger value. So in this case, you could conclude that beam search is at fault. Now, how about the other case? In case 2, P(y* given x) is less than or equal to P(y-hat given x), right? And then either this or this has gotta be true. So either case 1 or case 2 has to hold true. What do you conclude under case 2? Well, in our example, y* is a better translation than y-hat. But according to the RNN, P(y*) is less than P(yhat), so saying that y* is a less likely output than yhat. So in this case, it seems that the RNN model is at fault and it might be worth spending more time working on the RNN. There's some subtleties pertaining to length normalizations that I'm glossing over. And if you are using some sort of length normalization, instead of evaluating these probabilities, you should be evaluating the optimization objective that takes into account length normalization. But ignoring that complication for now, in this

case, what this tells you is that even though y* is a better translation, the RNN ascribed y* in lower probability than the inferior translation. So in this case, I will say the RNN model is at fault. So the error analysis process looks as follows. You go through the development set and find the mistakes that the algorithm made in the development set and so in this example, let's say that P(y* given x) was 2 x 10 to the -10, whereas, P(yhat given x) was 1 x 10 to the -10. Using the logic from the previous slide, in this case, we see that beam search actually chose y-hat, which has a lower probability than y*.



So I will say beam search is at fault. So I'll abbreviate that B. And then you go through a second mistake or second bad output by the algorithm, look at these probabilities and maybe for the second example, you think the model is at fault. I'm going to abbreviate the RNN model with R. And you go through more examples and sometimes the beam search is at fault, sometimes the model is at fault, and so on and through this process, you can then carry out error analysis to figure out what fraction of errors are due to beam search versus the RNN model. And with an error analysis process like this, for every example in your dev sets, where the algorithm gives a much worse output than the human translation, you can try to describe the error to either the search algorithm or to the objective function, or to the RNN model that generates the objective function that beam search is supposed to be maximizing. And through this, you can try to figure out which of these two components is responsible for more errors. And only if you find that beam search is responsible for a lot of errors, then maybe is we're working hard to increase the beam width. Whereas in contrast, if you find that the RNN model is at fault, then you could do a deeper layer of analysis to try to figure out if you want to add regularization, or get more training data, or try a different network architecture, or something else and so a lot of the techniques that you saw in the third course in the sequence will be applicable there. So that's it for error analysis using beam search. I found this particular error analysis process very useful whenever you have an approximate optimization algorithm, such as beam search that is working to optimize some sort of objective, some sort of cost function that is output by a learning algorithm, such as a sequence-to-sequence model or a sequence-to-sequence RNN that we've been discussing in these lectures. So with that, I hope that you'll be more efficient at making these types of models work well for your applications.



Bleu Score

One of the challenges of machine translation is that, given a French sentence, there could be multiple English translations that are equally good translations of that French sentence. So how do you evaluate a machine translation system if there are multiple equally good answers, unlike, say, image recognition where there's one right answer? You just measure accuracy. If there are multiple great answers, how do you measure accuracy? The way this is done conventionally is through something called the BLEU score. Let's say you are given a French sentence Le chat est sur le tapis. And you are given a reference, human generated translation of this, which is the the cat is on the mat. But there are multiple, pretty good translations of this. So a different human, different person might translate it as there is a cat on the mat and both of these are actually just perfectly fine translations of the French sentence. What the BLEU score does is given a machine generated translation, it allows you to automatically compute a score that measures how good is that machine translation and the intuition is so long as the machine generated translation is pretty close to any of the references provided by humans, then it will get a high BLEU score. BLEU, stands for bilingual evaluation, Understudy. So in the theater world, an understudy is someone that learns the role of a more senior actor so they can take over the role of the more senior actor, if necessary and motivation for BLEU is that, whereas you could ask human evaluators to evaluate the machine translation system, the BLEU score is an understudy, could be a substitute for having humans evaluate every output of a machine translation system. So the BLEU score was due to Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. This paper has been incredibly influential, and is, actually, quite a readable paper. So I encourage you to take a look if you have time. So, the intuition behind the BLEU score is we're going to look at the machine generated output and see if the types of words it generates appear in at least one of the human generated references and so these human generated references would be provided as part of the depth set or as part of the test set.

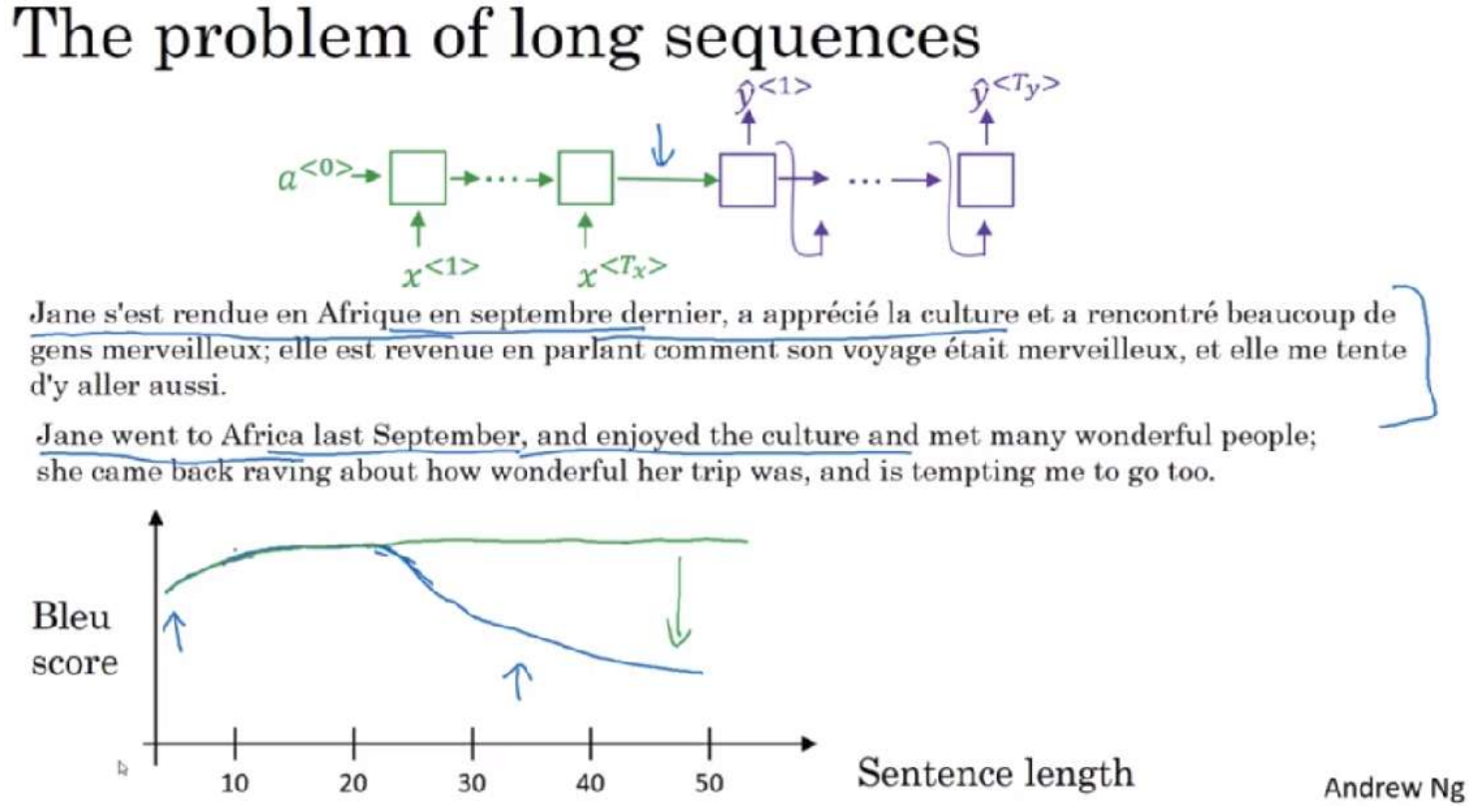
There is a great importance of having a single real number evaluation metric. Because it allows you to try out two ideas, see which one achieves a higher score, and then try to stick with the one that achieved the higher score. So the reason the BLEU score was revolutionary for machine translation was because this gave a pretty good, by no means perfect, but pretty good single real number evaluation metric and so that accelerated the progress of the entire field of machine translation. This section gave us a sense of how the BLEU score works. In practice, few people would implement a BLEU score from scratch. There are open source implementations that you can download and just use to evaluate your own system. But today, BLEU score is used to evaluate many systems that generate text, such as machine translation systems, as well as the example I showed briefly earlier of image captioning systems where you would have a system, have a neural network

generated image caption and then use the BLEU score to see how much that overlaps with maybe a reference caption or multiple reference captions that were generated by people. So the BLEU score is a useful single real number evaluation metric to use whenever you want your algorithm to generate a piece of text and you want to see whether it has similar meaning as a reference piece of text generated by humans. This is not used for speech recognition, because in speech recognition, there's usually one ground truth and you just use other measures to see if you got the speech transcription on pretty much, exactly word for word correct. But for things like image captioning, and multiple captions for a picture, it could be about equally good or for machine translations. There are multiple translations, but equally good. The BLEU score gives you a way to evaluate that automatically and therefore speed up your development.

Attention Model Intuition

For most of this week, we've been using a Encoder-Decoder architecture for machine translation. Where one R and N reads in a sentence and then different one outputs a sentence. There's a modification to this called the Attention Model, that makes all this work much better. The a**ttention algorithm, the attention idea has been one of the most influential ideas in deep learning**.

Let's take a look at how that works. Get a very long French sentence like this. What we are asking this green encoder in your network to do is, to read in the whole sentence and then memorize the whole sentences and store it in the activations conveyed here. Then for the purple network, the decoder network till then generate the English translation. Jane went to Africa last September and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too. Now, the way a human translator would translate this sentence is not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch. Instead, what the human translator would do is read the first part of it, maybe generate part of the translation. Look at the second part, generate a few more words, look at a few more words, generate a few more words and so on.

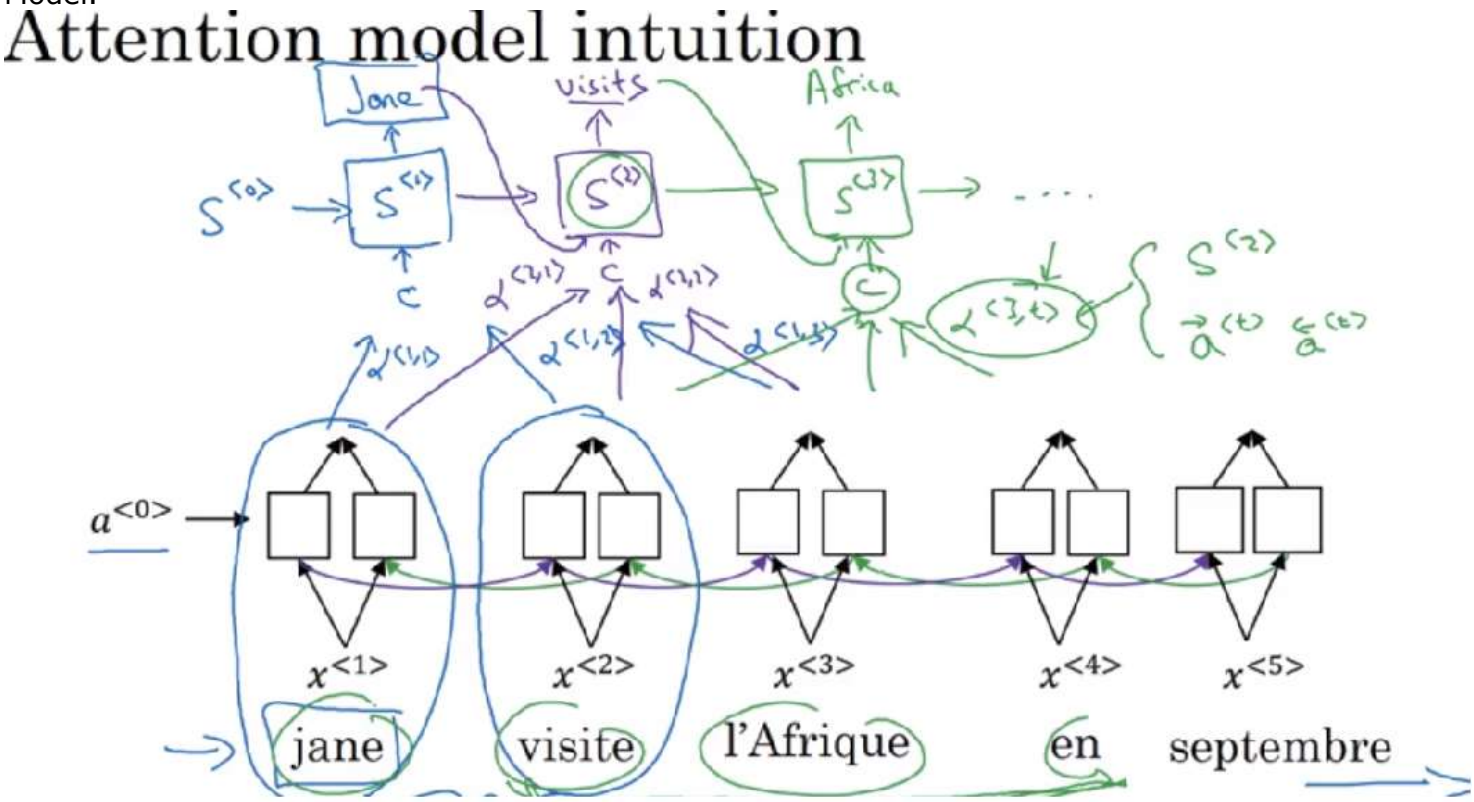


You kind of work part by part through the sentence, because it's just really difficult to memorize the whole long sentence like that. What you see for the Encoder-Decoder architecture above is that, it works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down. The Bleu score might look like this as the sentence that varies and short sentences are just hard to translate, hard to get all the words, right? Long sentences, it doesn't do well on because it's just difficult to get in your

network to memorize a super long sentence. In this and the next section, you'll see the Attention Model which translates maybe a bit more like humans might, looking at part of the sentence at a time and with an Attention Model, machine translation systems performance can look like this, because by working one part of the sentence at a time, you don't see this huge dip which is really measuring the ability of a neural network to memorize a long sentence which maybe isn't what we most badly need a neural network to do. In this video, I want to just give you some intuition about how attention works and then we'll flesh out the details in the next video.

The Attention Model was due to Dimitri, Bahdanau, Camcrun Cho, Yoshe Bengio and even though it was obviously developed for machine translation, it spread to many other application areas as well. This is really a very influential, I think very seminal paper in the deep learning literature. Let's illustrate this with a short sentence, even though these ideas were maybe developed more for long sentences, but it'll be easier to illustrate these ideas with a simpler example. We have our usual sentence, Jane visite l'Afrique en Septembre. Let's say that we use a R and N, and in this case, I'm going to use a bidirectional R and N, in order to compute some set of features for each of the input words and you have to understand it, bidirectional R and N with outputs Y1 to Y3 and so on up to Y5 but we're not doing a word for word translation, let me get rid of the Y's on top. But using a bidirectional R and N, what we've done is for each other words, really for each of the five positions into sentence, you can compute a very rich set of features about the words in the sentence and maybe surrounding words in every position. Now, let's go ahead and generate the English translation. We're going to use another R and N to generate the English translations. Here's my R and N note as usual and instead of using A to denote the activation, in order to avoid confusion with the activations down here, I'm just going to use a different notation, I'm going to use S to denote the hidden state in this R and N up here, so instead of writing A1 I'm going to right S1 and so we hope in this model that the first word it generates will be Jane, to generate Jane visits Africa in September. Now, the question is, when you're trying to generate this first word, this output, what part of the input French sentence should you be looking at? Seems like you should be looking primarily at this first word, maybe a few other words close by, but you don't need to be looking way at the end of the sentence. What the Attention Model would be computing is a set of attention weights and we're going to use Alpha one, one to denote when you're generating the first words, how much should you be paying attention to this first piece of information here. And then we'll also come up with a second that's called Attention Weight, Alpha one, two which tells us what we're trying to compute the first work of Jane, how much attention we're paying to this second work from the inputs and so on and the Alpha one, three and so on, and together this will tell us what is exactly the context from denoter C that we should be paying attention to, and that is input to this R and N unit to then try to generate the first words. That's one step of the R and N, we will flesh out all these details in the next video. For the second step of this R and N, we're going to have a new hidden state S two and we're going to have a new set of the attention weights. We're going to have Alpha two, one to tell us when we generate in the second word. I guess this will be visits maybe that being the ground trip label. How much should we paying attention to the first word in the french input and also, Alpha two, two and so on. How much should we paying attention the word visite, how much should we pay attention to the free and so on. And of course, the first word we generate in Jane is also an input to this, and then we have some context that we're paying attention to and the second step, there's also an input and that together will generate the second word and that leads us to the third step, S three, where this is an input and we have some new context C that depends on the various Alpha three for the different time sets, that tells us how much should we be paying attention to the different words from the input French sentence and so on. So, some things I haven't specified yet, but that will go further into detail in the next video of this, how exactly this context defines and the goal of the context is for the third word is really should capture that maybe we should be looking around this part of the sentence. The formula you use to do that will defer to the next video as well as how do you compute these attention weights. And you see in the next video that Alpha three T, which is, when you're trying to generate the third word, I guess this would be the Africa, just getting the right output. The amounts that this R and N step should be paying attention to the French word that time T, that depends on the activations of the bidirectional R and N at time T, I guess it depends on the fourth activations and the, backward activations at time T and it will depend on the state from the previous steps, it will depend on S two, and these things together will influence, how much you pay attention to a specific word in the input French sentence. But we'll flesh out all these details in the next video. But the key intuition to take

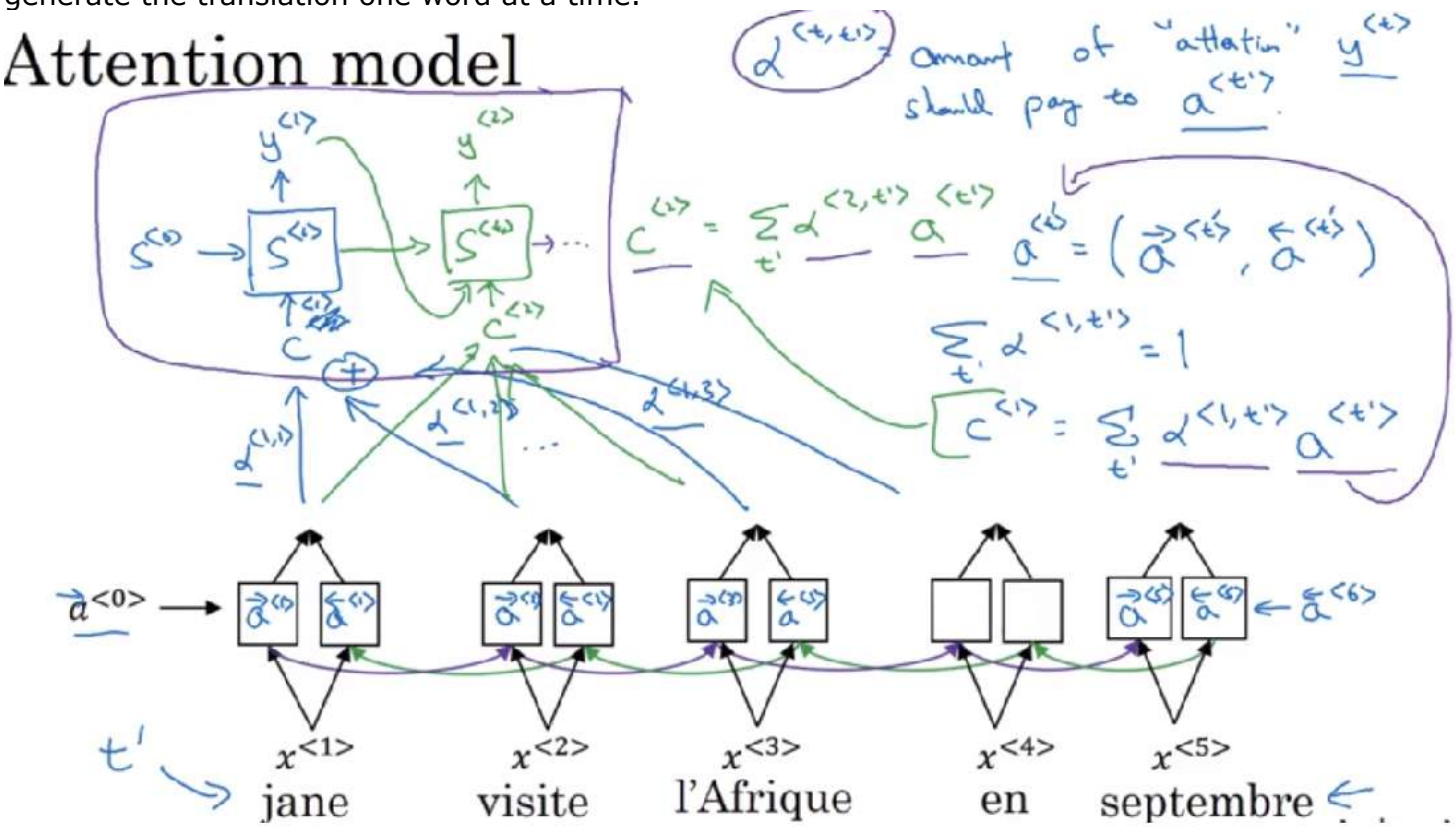
away is that this way the R and N marches forward generating one word at a time, until eventually it generates maybe the EOS and at every step, there are these attention weighs. Alpha T.T. Prime that tells it, when you're trying to generate the T, English word, how much should you be paying attention to the T prime French words. And this allows it on every time step to look only maybe within a local window of the French sentence to pay attention to, when generating a specific English word. I hope this video conveys some intuition about Attention Model and that we now have a rough sense of, maybe how the algorithm works. Let's go to the next video to flesh out the details of the Attention Model.



Attention Model

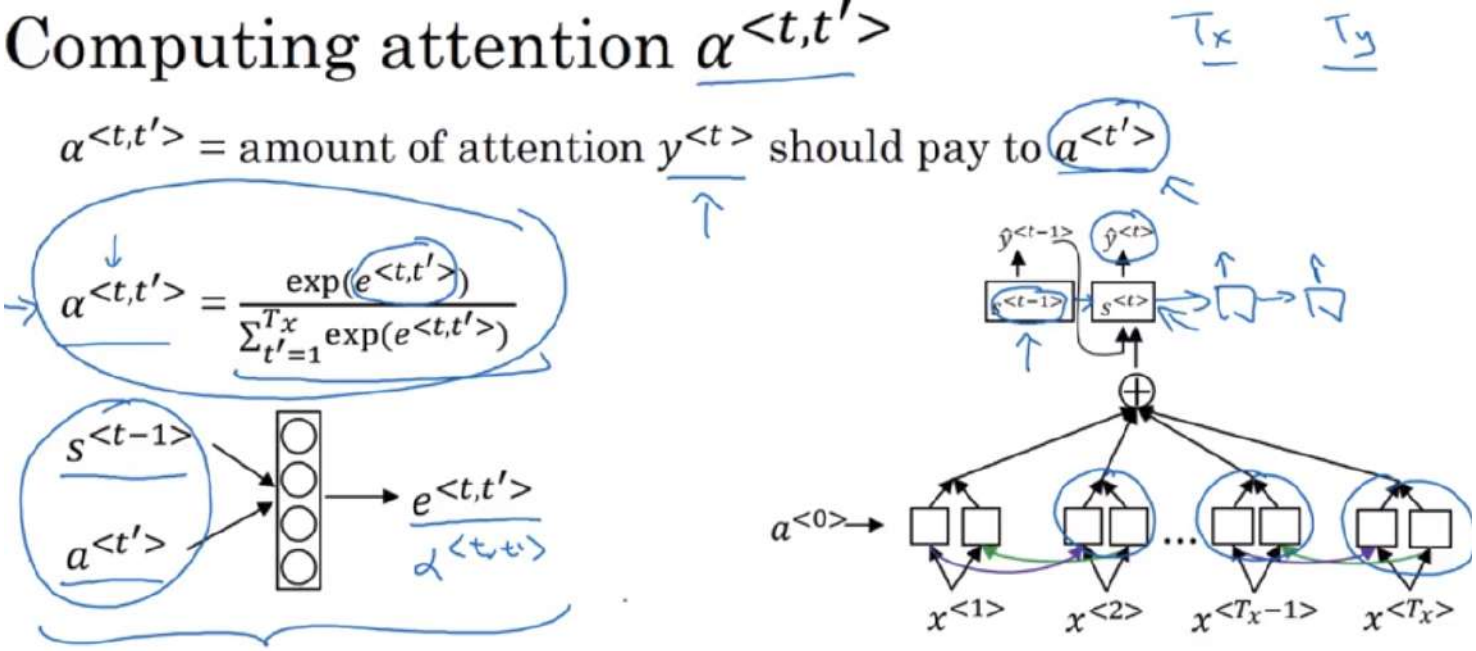
In the last video, you saw how the attention model allows a neural network to pay attention to only part of an input sentence while it's generating a translation, much like a human translator might. Let's now formalize that intuition into the exact details of how you would implement an attention model. So same as in the previous video, let's assume you have an input sentence and you use a bidirectional RNN, or bidirectional GRU, or bidirectional LSTM to compute features on every word. In practice, GRUs and LSTMs are often used for this, with maybe LSTMs be more common. And so for the forward occurrence, you have a forward occurrence first time step. Activation backward occurrence, first time step. Activation forward occurrence, second time step. Activation backward and so on. For all of them in just a forward fifth time step a backwards fifth time step. We had a zero here technically we can also have I guess a backwards sixth as a factor of all zero, actually that's a factor of all zeroes. And then to simplify the notation going forwards at every time step, even though you have the features computed from the forward occurrence and from the backward occurrence in the bidirectional RNN. I'm just going to use a of t to represent both of these concatenated together. So a of t is going to be a feature vector for time step t. Although to be consistent with notation, we're using second, I'm going to call this t_prime. Actually, I'm going to use t_prime to index into the words in the French sentence. Next, we have our forward only, so it's a single direction RNN with state s to generate the translation. And so the first time step, it should generate y1 and just will have as input some context C. And if you want to index it with time I guess you could write a C1 but sometimes I just right C without the superscript one. And this will depend on the attention parameters so alpha_11, alpha_12 and so on tells us how much attention. And so these alpha parameters tells us how much the context would depend on the features we're getting or the activations we're getting from the different time steps. And so the way we define the context is actually be a way to some of the features from the different time steps waited by these attention

waits. So more formally the attention waits will satisfy this that they are all be non-negative, so it will be a zero positive and they'll sum to one. We'll see later how to make sure this is true. And we will have the context or the context at time one often drop that superscript that's going to be sum over t_prime, all the values of t_prime of this waited sum of these activations. So this term here are the attention waits and this term here comes from here. So alpha(t_prime) is the amount of attention that's yt should pay to a of t_prime. So in other words, when you're generating the t of the output words, how much you should be paying attention to the t_primeth input to word. So that's one step of generating the output and then at the next time step, you generate the second output and is again done some of where now you have a new set of attention waits on they to find a new way to sum. That generates a new context. This is also input and that allows you to generate the second word. Only now just this way to sum becomes the context of the second time step is sum over t_prime alpha(2, t_prime). So using these context vectors. C1 right there back, C2, and so on. This network up here looks like a pretty standard RNN sequence with the context vectors as output and we can just generate the translation one word at a time.

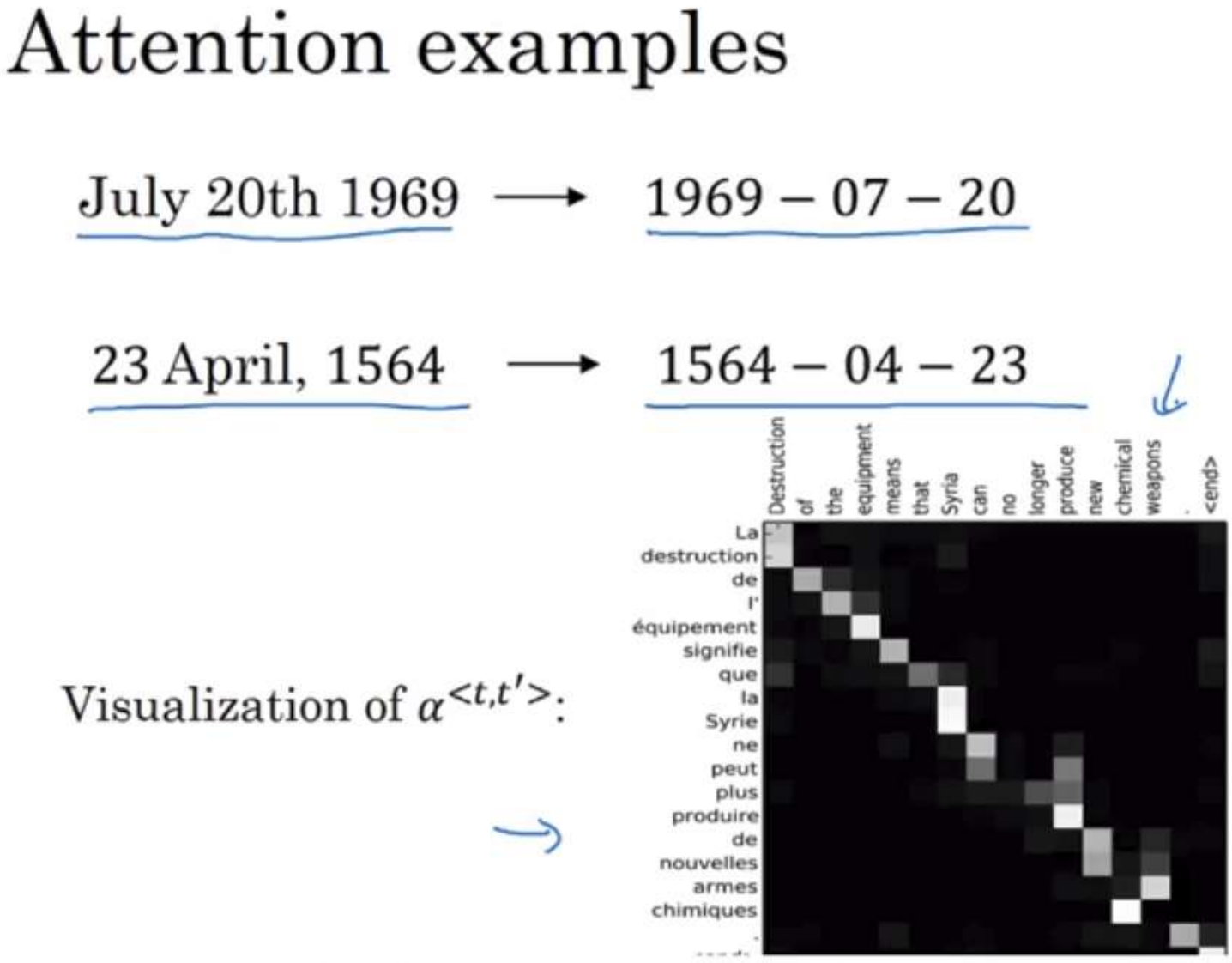


We have also define how to compute the context vectors in terms of these attention ways and those features of the input sentence. So the only remaining thing to do is to define how to actually compute these attention waits. Let's do that on the next slide. So just to recap, alpha(t, t_prime) is the amount of attention you should paid to a(t_prime) when you're trying to generate the t th words in the output translation. So let me just write down the formula and we talk of how this works. This is formula you could use the compute alpha(t, t_prime) which is going to compute these terms e(t, t_prime) and then use essentially a soft pass to make sure that these waits sum to one if you sum over t_prime. So for every fix value of t, these things sum to one if you're summing over t_prime. And using this soft max prioritization, just ensures this properly sums to one. Now how do we compute these factors e. Well, one way to do so is to use a small neural network as follows. So s t minus one was the neural network state from the previous time step. So here is the network we have. If you're trying to generate yt then st minus one was the hidden state from the previous step that just fell into st and that's one input to very small neural network. Usually, one hidden layer in neural network because you need to compute these a lot. And then a(t_prime) the features from time step t_prime is the other inputs. And the intuition is, if you want to decide how much attention to pay to the activation of t_prime. Well, the things that seems like it should depend the most on is what is your own hidden state activation from the previous time step. You don't have the current state

activation yet because of context feeds into this so you haven't computed that. But look at whatever you're hidden stages of this RNN generating the upper translation and then for each of the positions, each of the words look at their features. So it seems pretty natural that alpha(t, t_prime) and e(t, t_prime) should depend on these two quantities. But we don't know what the function is. So one thing you could do is just train a very small neural network to learn whatever this function should be. And trust that obligation trust wait and descent to learn the right function. And it turns out that if you implemented this whole model and train it with gradient descent, the whole thing actually works. This little neural network does a pretty decent job telling you how much attention yt should pay to a(t_prime) and this formula makes sure that the attention waits sum to one and then as you chug along generating one word at a time, this neural network actually pays attention to the right parts of the input sentence that learns all this automatically using gradient descent. Now, one downside to this algorithm is that it does take quadratic time or quadratic cost to run this algorithm. If you have tx words in the input and ty words in the output then the total number of these attention parameters are going to be tx times ty. And so this algorithm runs in quadratic cost.



Although in machine translation applications where neither input nor output sentences is usually that long maybe quadratic cost is actually acceptable. Although, there is some research work on trying to reduce costs as well. Now, so far up in describing the attention idea in the context of machine translation. Without going too much into detail this idea has been applied to other problems as well. So just image captioning. So in the image capturing problem the task is to look at the picture and write a caption for that picture. So in this paper set to the bottom by Kevin Chu, Jimmy Barr, Ryan Kiros, Kelvin Shaw, Aaron Korver, Russell Zarkutnov, Virta Zemo, and Andrew Benjo they also showed that you could have a very similar architecture. Look at the picture and pay attention only to parts of the picture at a time while you're writing a caption for a picture. So if you're interested, then I encourage you to take a look at that paper as well. And you get to play with all this and more in the programming exercise. Whereas machine translation is a very complicated problem in the prior exercise you get to implement and play of the attention while you yourself for the date normalization problem. So the problem inputting a date like this. This actually has a date of the Apollo Moon landing and normalizing it into standard formats or a date like this and having a neural network a sequence, sequence model normalize it to this format. This by the way is the birthday of William Shakespeare.



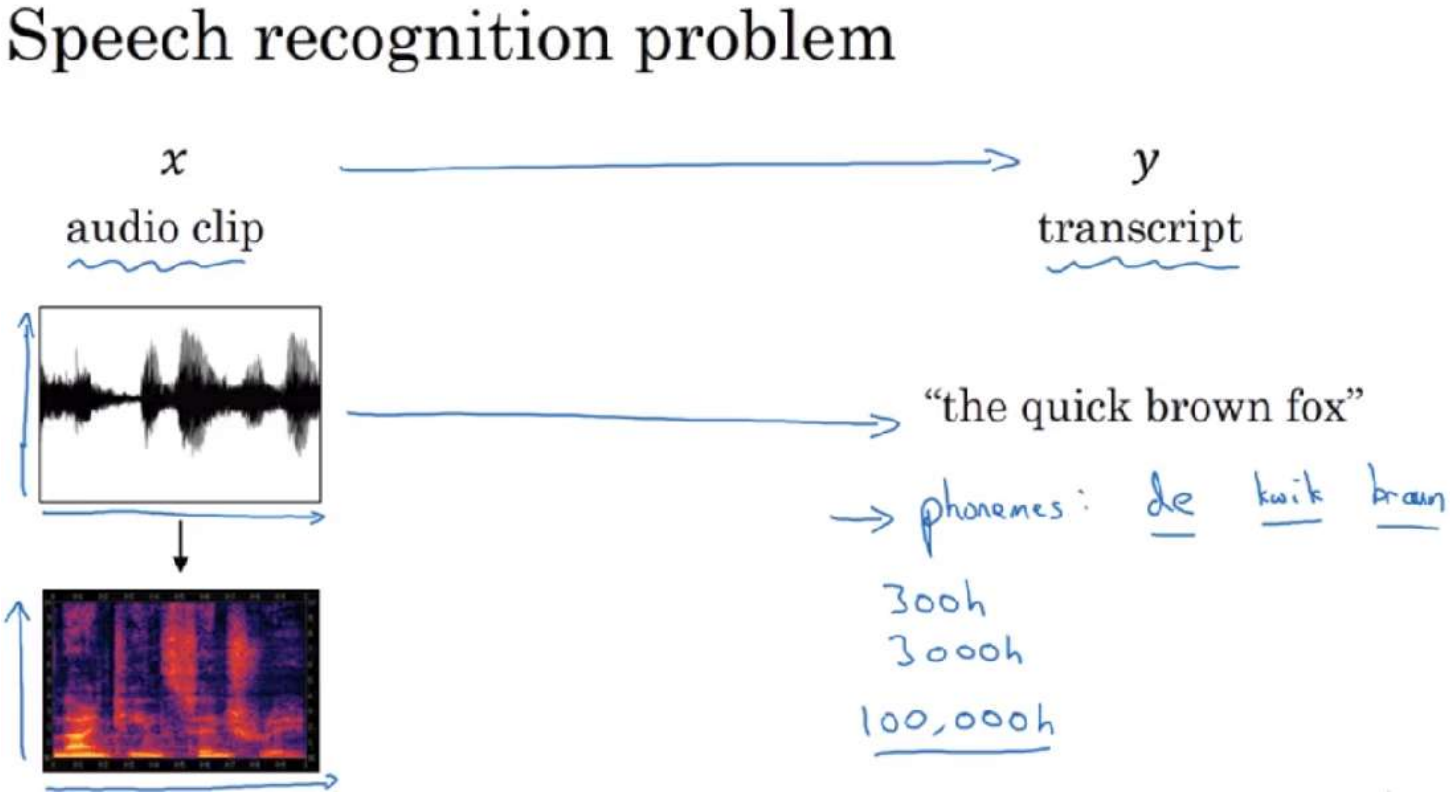
Also it's believed to be. And what you see in prior exercises as you can train a neural network to input dates in any of these formats and have it use an attention model to generate a normalized format for these dates. One other thing that sometimes fun to do is to look at the visualizations of the attention waits. So here's a machine translation example and here were plotted in different colors. the magnitude of the different attention waits. I don't want to spend too much time on this but you find that the corresponding input and output words you find that the attention waits will tend to be high. Thus, suggesting that when it's generating a specific word in output is, usually paying attention to the correct words in the input and all this including learning where to pay attention when was all learned using propagation with an attention model.

Speech Recognition - Audio data

Speech recognition

One of the most exciting developments were sequence-to-sequence models has been the rise of very accurate speech recognition. We're nearing the end of the course, we want to take just a couple of videos to give you a sense of how these sequence-to-sequence models are applied to audio data, such as the speech. So, what is the speech recognition problem? You're given an audio clip, x, and your job is to automatically find a text transcript, y. So, an audio clip, if you plot it looks like this, the horizontal axis here is time, and what a microphone does is it really measures minuscule changes in air pressure, and the way you're hearing my voice right now is that your ear is detecting little changes in air pressure, probably generated either by your speakers or by a headset. And some audio clips like this plots with the air pressure against time. And, if this audio clip is of me saying, "the quick brown fox", then hopefully, a speech recognition algorithm can input that audio clip and output that transcript. And because even the human ear doesn't process raw wave forms, but the human ear

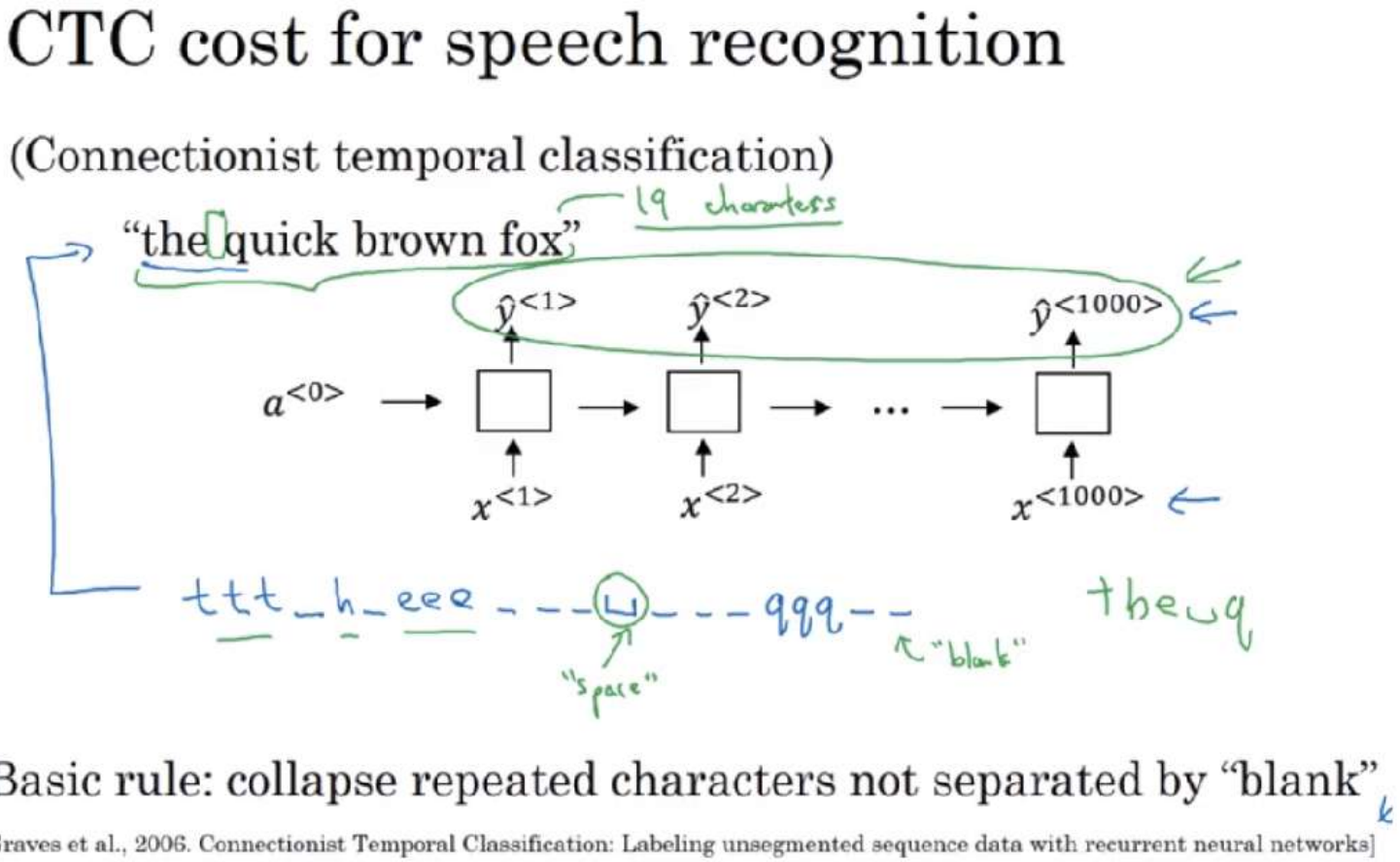
has physical structures that measures the amounts of intensity of different frequencies, there is, a common pre-processing step for audio data is to run your raw audio clip and generate a spectrogram. So, this is the plots where the horizontal axis is time, and the vertical axis is frequencies, and intensity of different colors shows the amount of energy. So, how loud is the sound at different frequencies? At different times? And so, these types of spectrograms, or you might also hear people talk about false back outputs, is often commonly applied pre-processing step before audio is pass into in the running algorithm. And the human ear does a computation pretty similar to this pre-processing step. So, one of the most exciting trends in speech recognition is that, once upon a time, speech recognition systems used to be built using phonemes and this where, I want to say hand-engineered basic units of cells.



So, the quick brown fox represented as phonemes. I'm going to simplify a bit, let say, "The" has a "de" and "e" sound and Quick, has a "ku" and "wu", "ik", "k" sound, and linguist used to write off these basic units of sound, and try the Greek language down to these basic units of sound. So, brown, this aren't the official phonemes which are written with more complicated notation, but linguists use to hypothesize that writing down audio in terms of these basic units of sound called phonemes would be the best way to do speech recognition. But with end-to-end deep learning, we're finding that phonemes representations are no longer necessary. But instead, you can built systems that input an audio clip and directly output a transcript without needing to use hand-engineered representations like these. One of the things that made this possible was going to much larger data sets. So, academic data sets on speech recognition might be as a 300 hours, and in academia, 3000 hour data sets of transcribed audio would be considered reasonable size, so lot of research has been done, a lot of research papers that are written on data sets there are several thousand voice. But, the best commercial systems are now trains on over 10,000 hours and sometimes over a 100,000 hours of audio. And, it's really moving to a much larger audio data sets, transcribe audio data sets were both x and y, together with deep learning algorithm, that has driven a lot of progress is speech recognition.

So, how do you build a speech recognition system? In the last section, we're talking about the attention model. So, one thing you could do is actually do that, where on the horizontal axis, you take in different time frames of the audio input, and then you have an attention model try to output the transcript like, "the quick brown fox", or what it was said. One other method that seems to work well is to use the CTC cost for speech recognition. CTC stands for **Connectionist Temporal Classification** and is due to Alex Graves, Santiago Fernandes, Faustino Gomez, and Jürgen Schmidhuber. So, here's the idea. Let's say the audio clip was someone saying, "the quick brown

fox". We're going to use a neural network structured as shown in diagram below with an equal number of input x's and output y's, and we have drawn a simple of what uni-directional for the RNN for this, but in practice, this will usually be a bidirectional LSTM and bidirectional GRU and usually, a deeper model. But notice that the number of time steps here is very large and in speech recognition, usually the number of input time steps is much bigger than the number of output time steps. So, for example, if you have 10 seconds of audio and your features come at a 100 hertz so 100 samples per second, then a 10 second audio clip would end up with a thousand inputs. Right, so it's 100 hertz times 10 seconds, and so with a thousand inputs. But your output might not have a thousand alphabets, might not have a thousand characters. So, what do you do? The CTC cost function allows the RNN to generate an output like this ttt, there's a special character called the blank character, which we're going to write as an underscore here, h_eee___, and then maybe a space, we're going to write like this, so that a space and then ___ qqq__. And, this is considered a correct output for the first parts of the space, quick with the Q, and the basic rule for the CTC cost function is to collapse repeated characters not separated by "blank". So, to be clear, I'm using this underscore to denote a special blank character and that's different than the space character. So, there is a space here between the and quick, so I should output a space. But, by collapsing repeated characters, not separated by blank, it actually collapse the sequence into t, h, e, and then space, and q, and this allows your network to have a thousand outputs by repeating characters allow the times. So, inserting a bunch of blank characters and still ends up with a much shorter output text transcript. So, this phrase here **"the quick brown fox"** including spaces actually has 19 characters, and if somehow, the newer network is forced upwards of a thousand characters by allowing the network to insert blanks and repeated characters and can still represent this 19 character upwards with this 1000 outputs of values of Y.

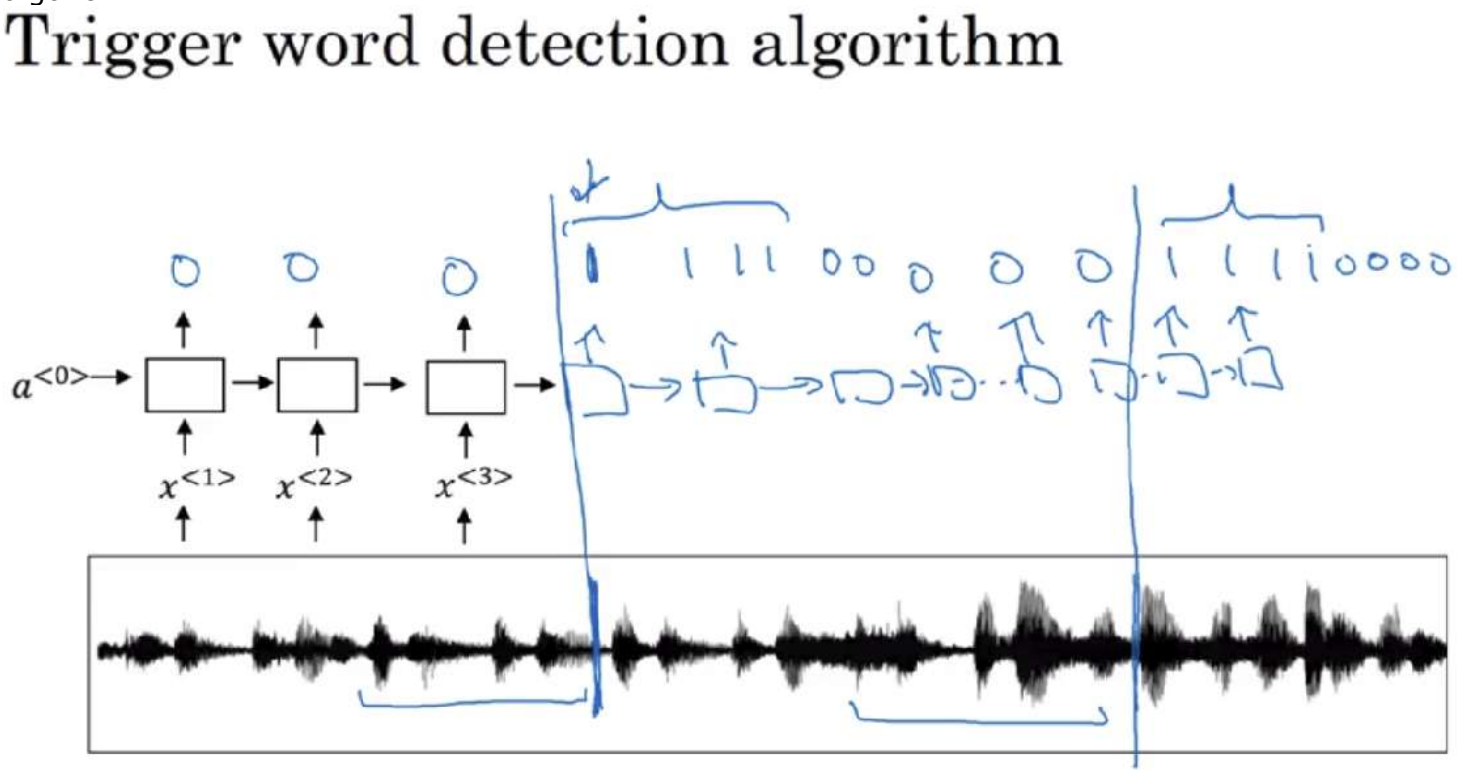


So, this paper by Alex Grace, as well as by those deep speech recognition system, which I was involved in, used this idea to build effective Speech recognition systems. So, I hope that gives you a rough sense of how speech recognition models work. Attention like models work and CTC models work and present two different options of how to go about building these systems. Now, today, building effective where production skills speech recognition system is a pretty significant effort and requires a very large data set. But, what I like to do in the next video is share you, how you can build a trigger word detection system, where keyword detection system which is actually much easier and

can be done with even a smaller or more reasonable amount of data. So, let's talk about that in the next video.

Trigger Word Detection

We've now learned so much about deep learning and sequence models that we can actually describe a trigger word system quite simply but when the rise of speech recognition have been more and more devices you can wake up with your voice and those are sometimes called **trigger word detection** systems so let's see how you can build a trigger word system examples of triggering systems include Amazon echo which is broken out what that word alexa or how Apple Siri working out with hey Siri and Google home woken up with ok Google so stands the trigger word detection that if you have say an Amazon echo in your living room you can walk the living room and just say Alexa what time is it and have it wake up I'll be triggered by the words of alexa and answer your voice query so if you can build a trigger word detection system maybe you can make your computer do something by telling it computer activate one of my friends also works on turning on an offer particular lamp using a trigger word kind of as a fun project but what I want to show you is how you can build a trigger word detection system now the trigger word detection literature is still evolving so there actually isn't a single universally agreed on algorithm for trigger word detection yet the literature on trigger word detection algorithm is still evolving so there isn't wide consensus yet on what's the best algorithm for trigger word detection so I'm just going to show you one example of an algorithm.



You can use now you've seen our ends like this and what we really do is take an audio clip maybe compute spectrogram features and that generates features x1 x2 x3 audio features x1 x2 x3 that you pass through an RNN and so all that remains to be done is to define the target labels Y so if this point in the audio clip is when someone just finished saying the trigger word such as hey Siri or okay Google then in the training sets you can set the target labels to be zero for everything before that point and right after that to set the target label of one and then if a little bit later on you know the trigger word was set again and the trigger was said at this point then you can again set the target label to be one right after that now this type of labeling scheme for an RNN you know could work actually this won't actually work reasonably well one slight disadvantage of this is it creates a very imbalanced training set so if a lot more zeros than ones so one other thing you could do that it's getting a little bit of a hack but could make them all the little bit easy to train is instead of setting only a single time step to output one you can actually make an output a few ones for several times or for a fixed period of time before reverting back to zero so and that thumb slightly evens out the ratio of ones to zeros but this is a little bit of a hack but if this is when in the audio clipper trigger where

the set then right after that you can set the toggle able to one and if this is the trigger words say the game then right after that just when you want the RNN to output one but so I think you should feel quite proud of yourself we've learned enough about the learning that it just takes one picture at one slide to this to describe something as complicated as trigger word detection and based on this I hope you'd be able to implement something that works and allows you to detect trigger words and get it to work maybe even make it do something fun in your house that I'm like turn on or turn off you could do something like a computer when you're when someone else says they trigger words on this is the last technical section of this course and to wrap up in this course on sequence models you learned about RNN's including both GRU and LSTM and then in the second week you learned a lot about word embeddings and how they learn representations of words and then in this week you learned about the attention model as well as how to use it to process audio data.

**********END OF COURSE **********