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

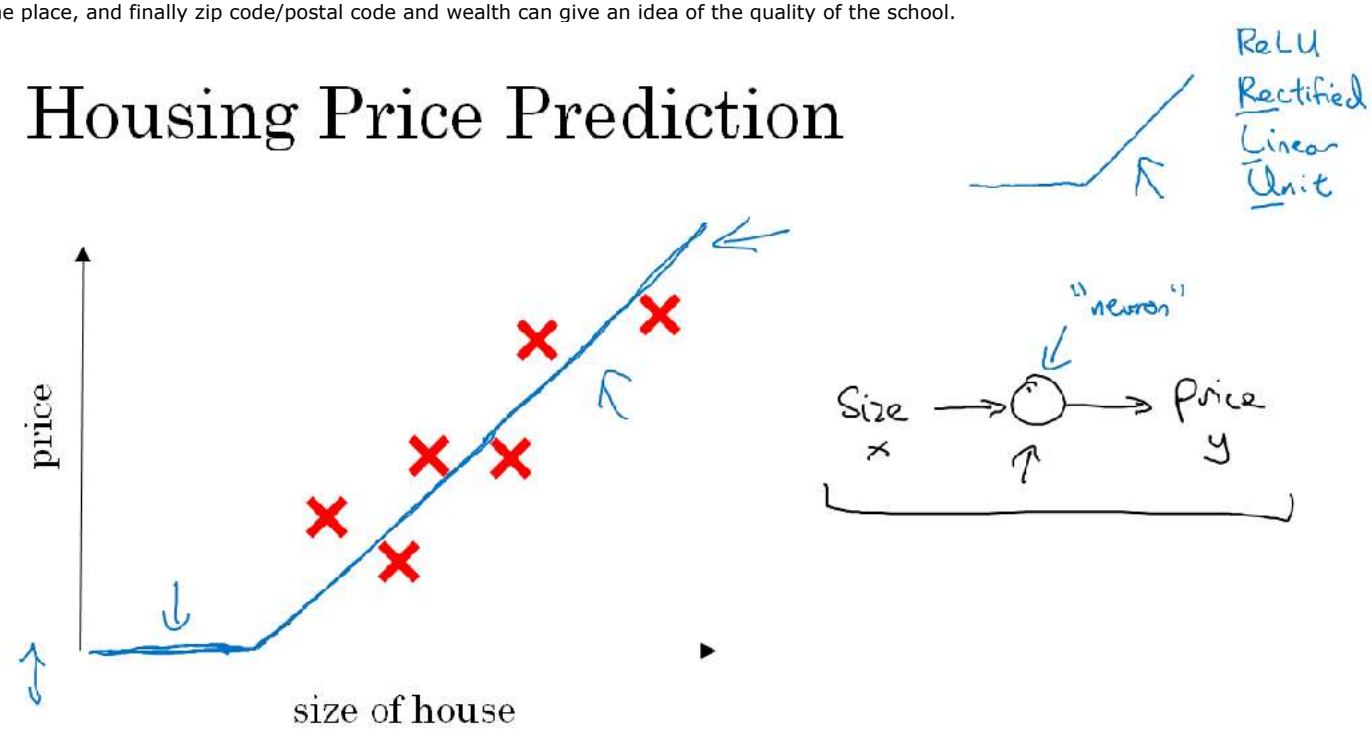
This is the simplest possible NN which can be solved by liner regression

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

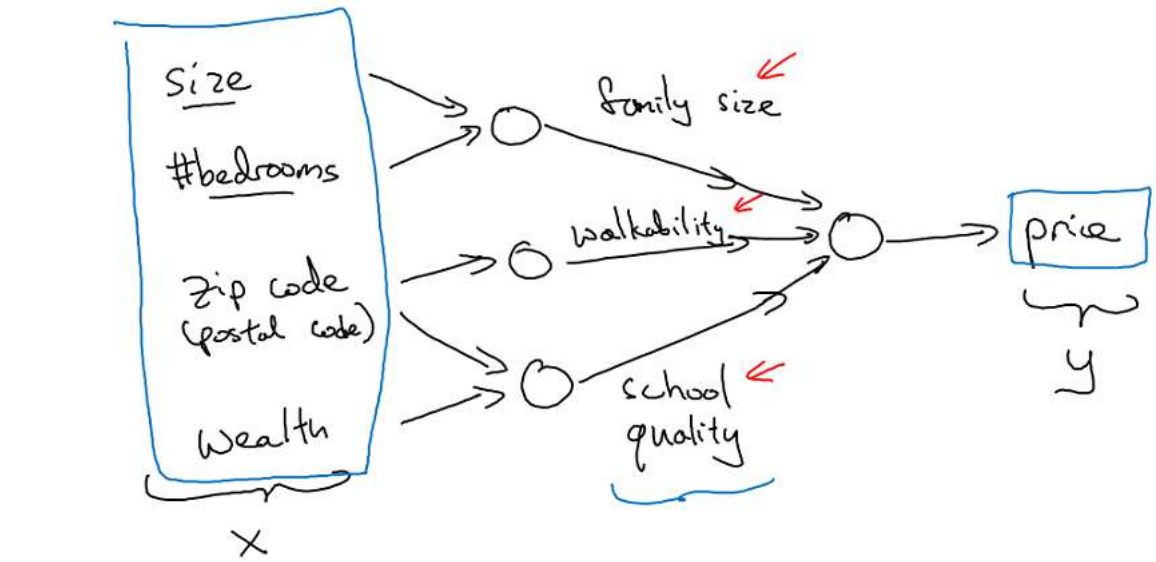
Housing price prediction advance (with more than 1 neuron)

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.



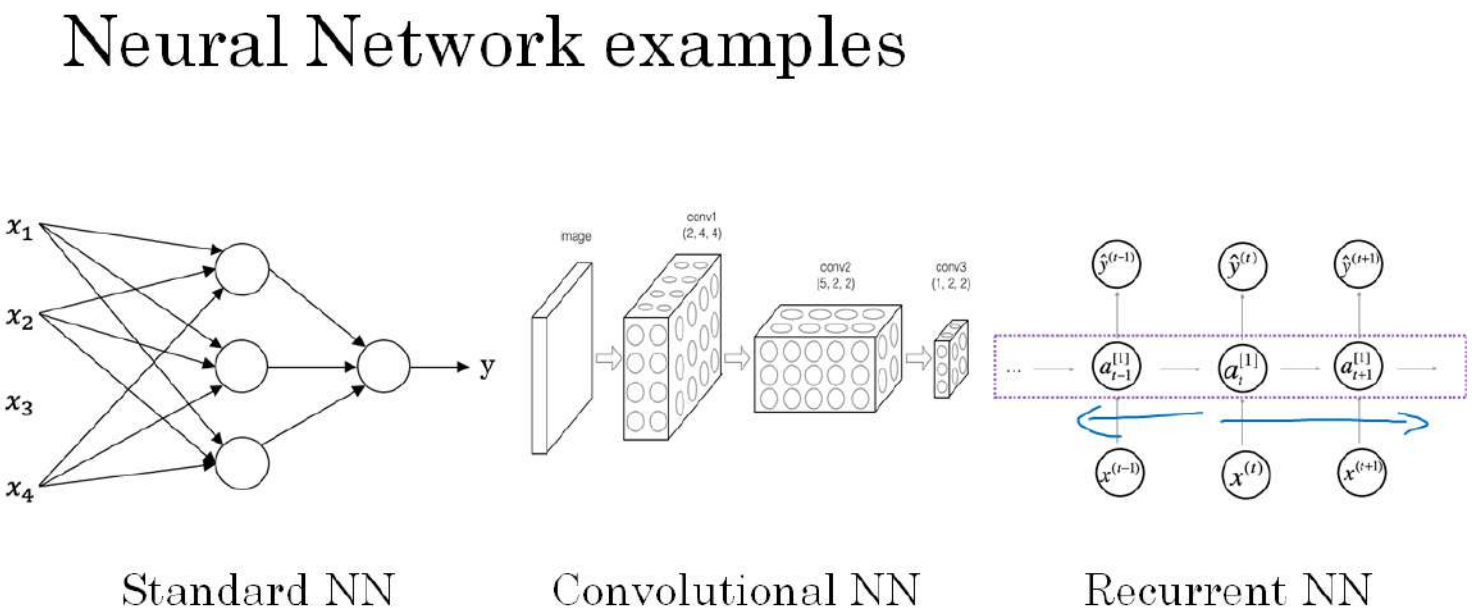
So it is found that NN are more useful in supervised learning setup where we have x input and map to y output as we saw in housing price prediction problem.

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

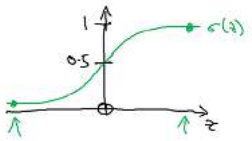
Logistic regression is the algorithm for binary classification. Ex: You have an image input, the binary classification will tell either its cat(1) or non-cat(0) this output label is called y. The input image is stored in computer in 3 separate matrices (RGB) as per size of the image, means that if you 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:

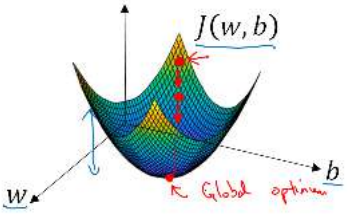
Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)})\right]$

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.



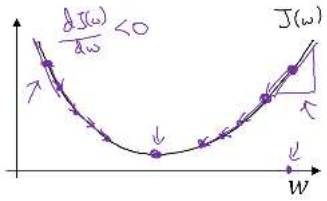
For the purpose of illustration lets say that there is some function J(w) we want to minimize. To make this easier to draw we are going to ignore b for now, just to make this a 1-D plot instead of higher dim plot.

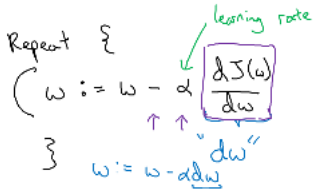
Repeat {
w := w - alpha dJ(w)/dw
}

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

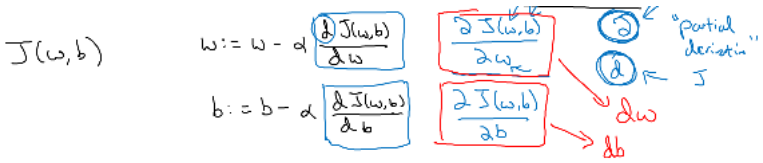
alpha here is the learning rate and controls how bigger step we take on each iteration of gradient descent. dJ(w)/dw is the derivative (slope of a function at a point), the update of the change you want to make to the parameter w. When we'll write code variable dw will be used for derivative term.

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.



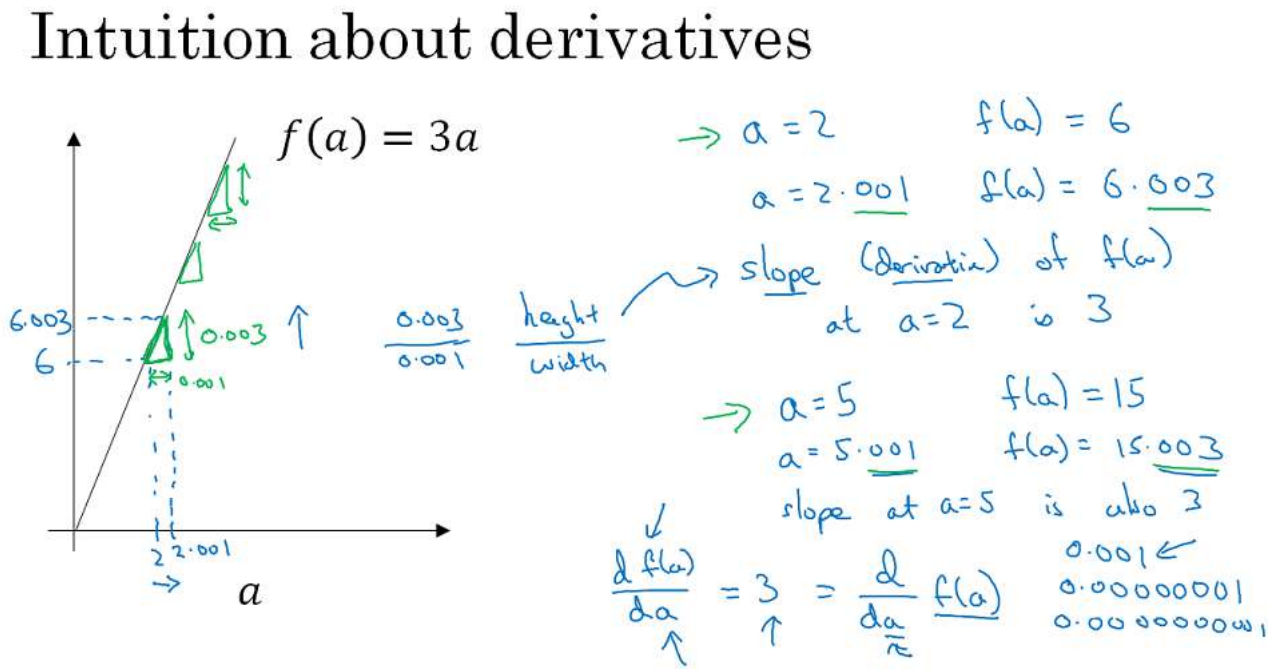


Overall intuition is now is that **dJ(w)/dw** represents the slope of the function and we want to know slope of the function at the current setting of the parameters so that we can the take the step of steepest descent. Now for both the parameter w and b **J(w,b)** we need to update the parameter b too 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

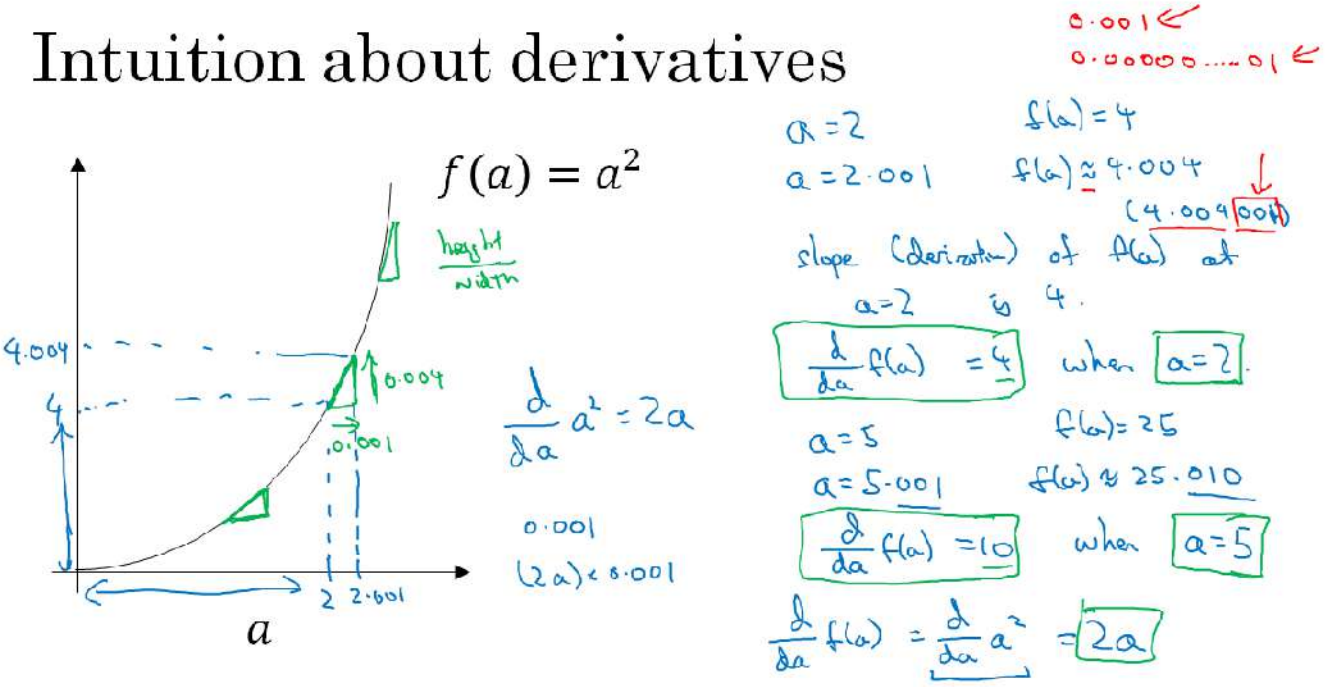
Let's first check the diagram below:



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.

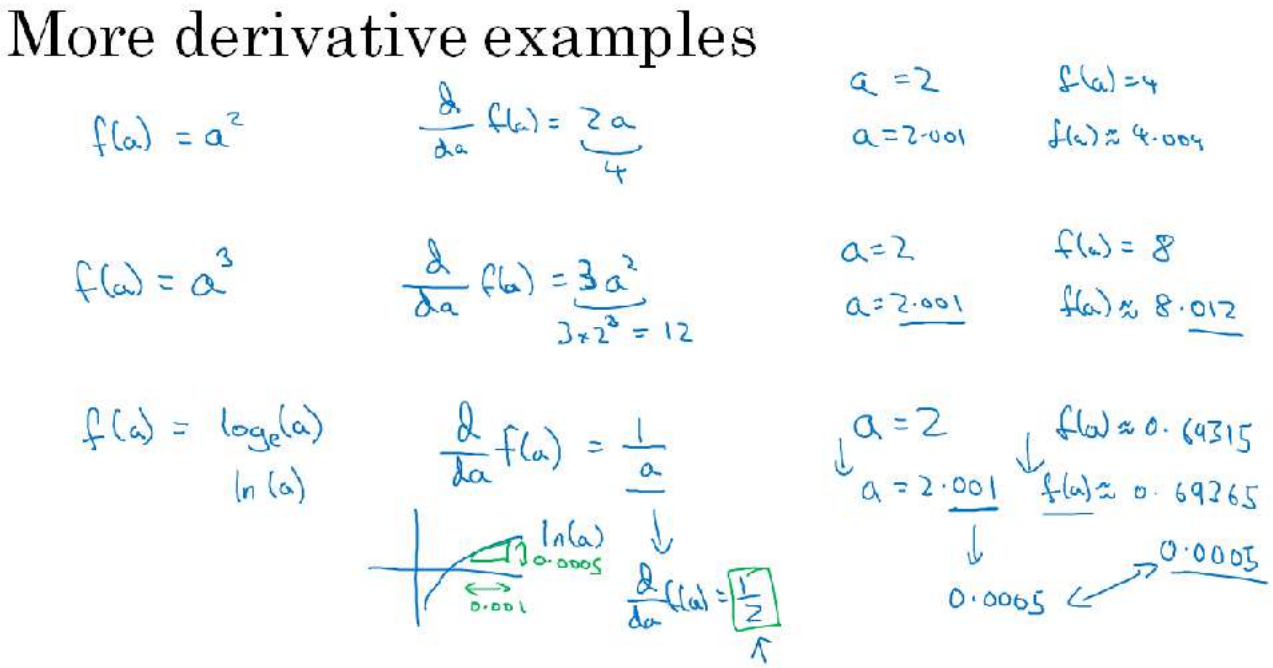
More derivative examples

Let's first check the diagram below:



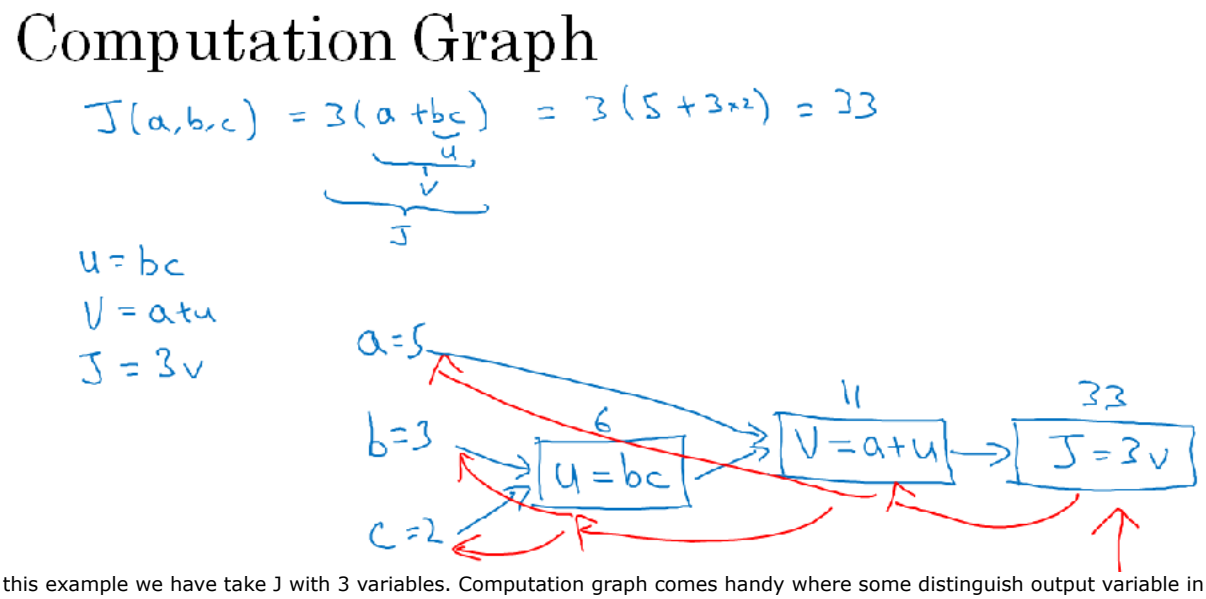
Let's go step by step and discuss the intuition about derivatives. In this example we'll see that slope of the function can be different at the different point at the function. The plotted function is f(a) = a^2 for a = 2, f(a) = 4, now slightly nudge val of a = 2.001, f(a) =4.004. If we plot this (as 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

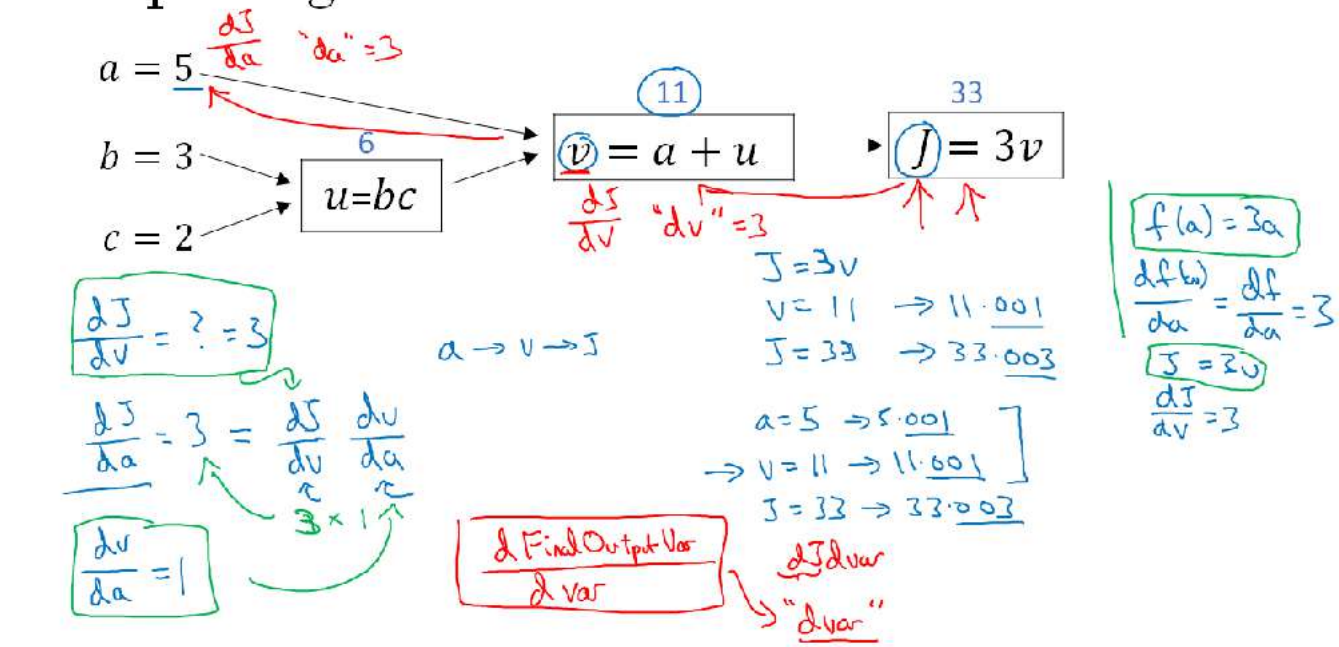
We firmly say that computation of neural network is organized in terms of two steps: a forward pass or forward propagation step which computes the output of the neural network followed by a backward pass or backward propagation which is used to compute the gradient or derivatives. The computation graph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



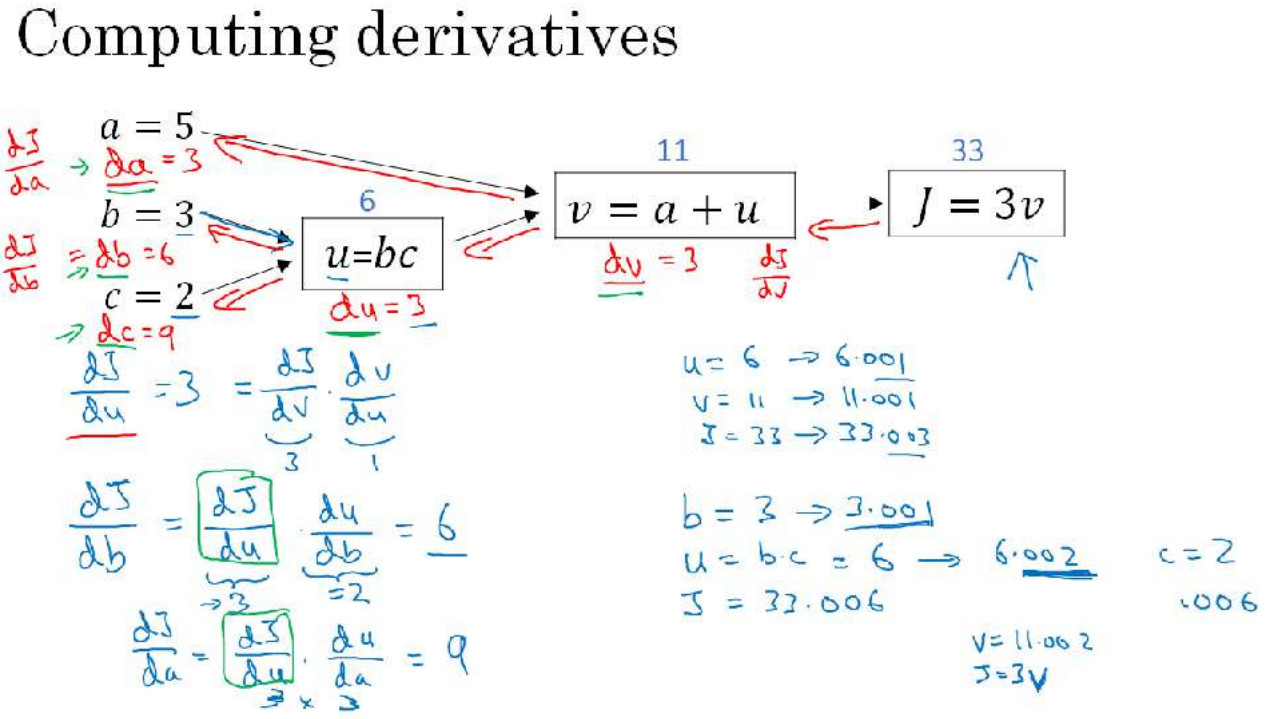
In this example we have take J with 3 variables. Computation graph comes handy where some distinguish output variable in this case to be optimized. In case of Logistic regression it is the cost function which we are trying to minimize. In this example we have also seen that with left to right pass we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

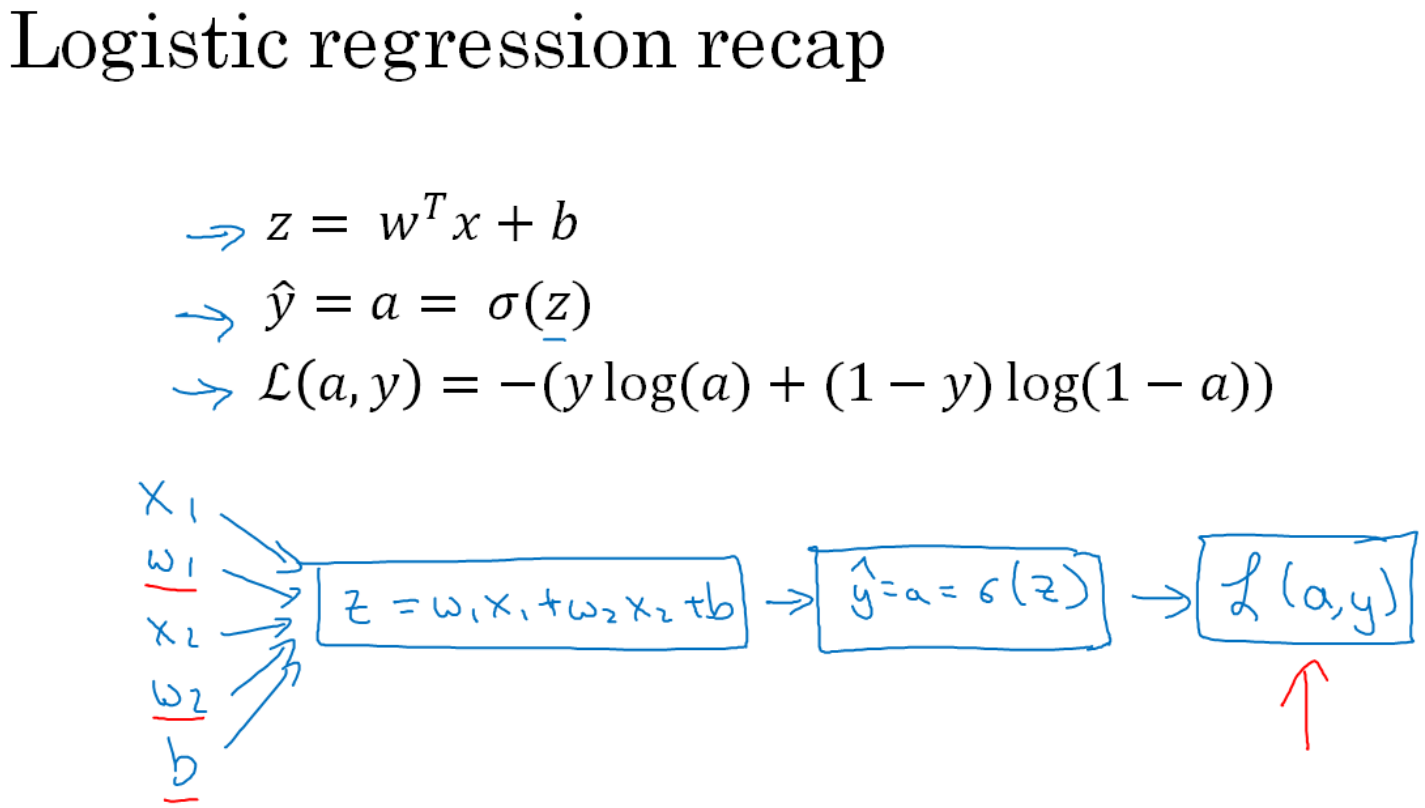


So the key take away from these example is that when computing all the derivatives, the most efficient way is to do is the the right to left computation (red arrows in graph) so first is to compute derivative with respect to v (dJ/dv or dv) and that will help in computing derivative with respect to a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

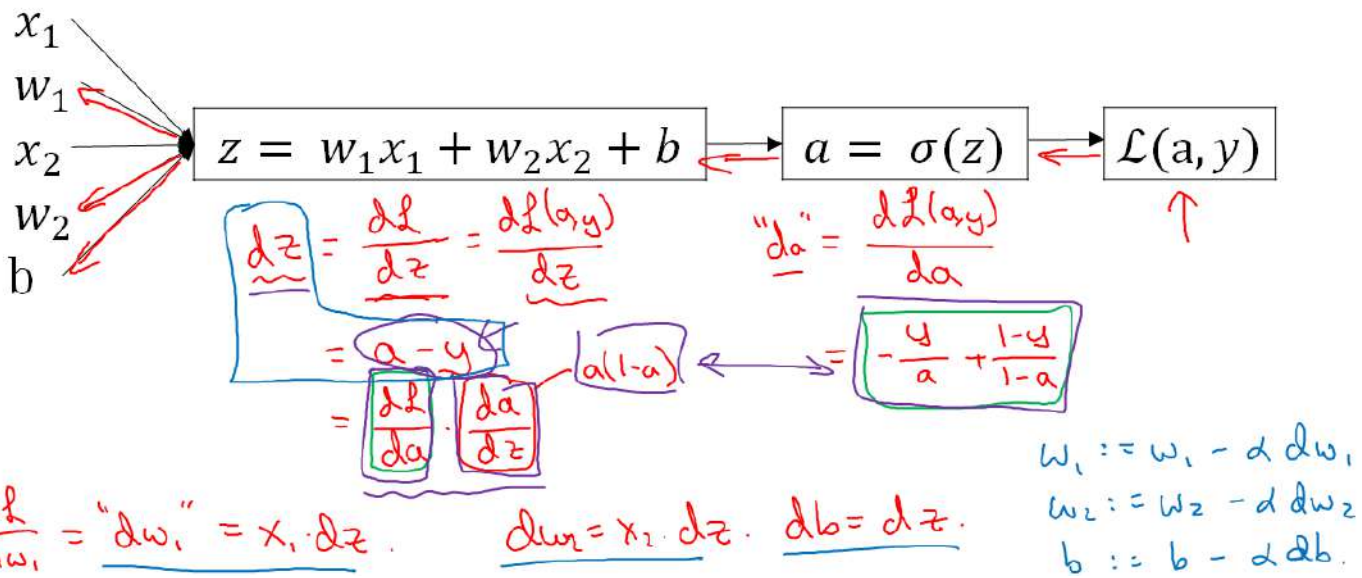
Let's check a diagram first and recap:



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:

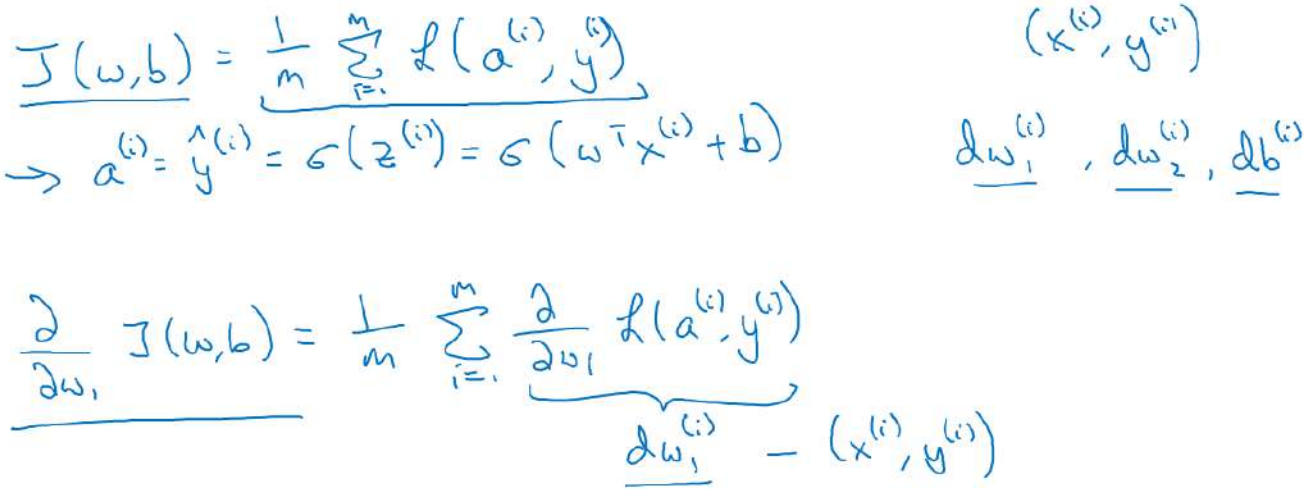
Logistic regression derivatives



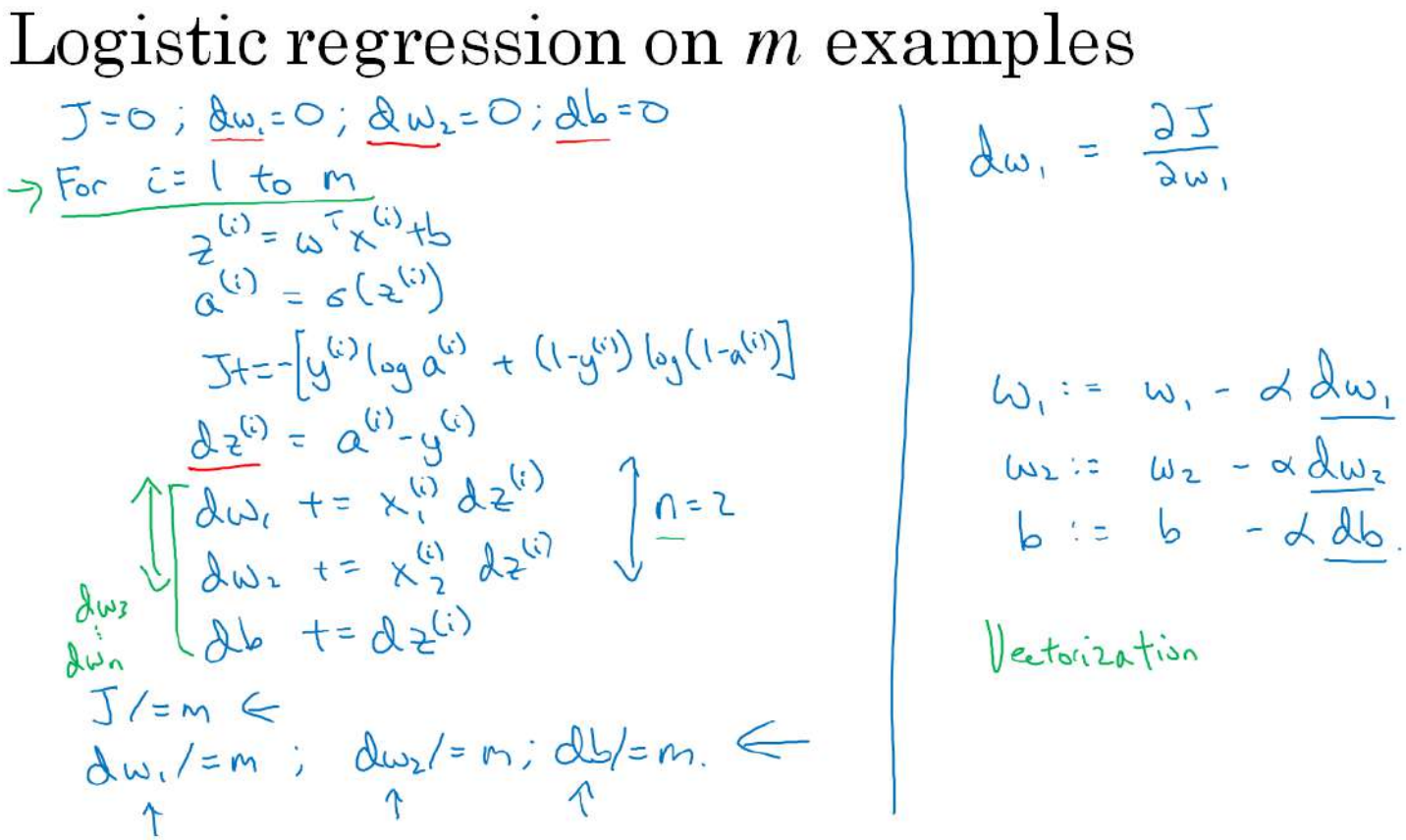
We've described the four propagation steps of how you actually compute the loss on a single training example, now let's talk about how you can go backwards to compute the derivatives. As shown in diagram, what we want to do is compute derivatives with respect to this loss, the first thing we want to 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.

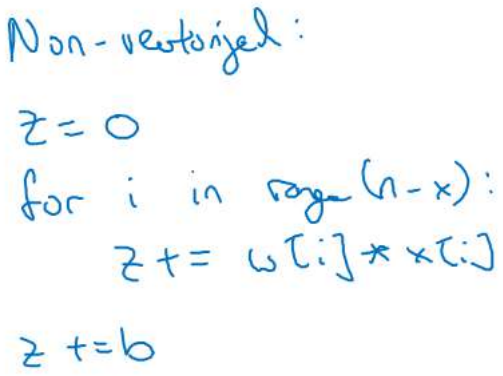


See the diagram above. Let's initialize J equals 0 on dw1=0, dw2=0, db=0 and what we're going to do is use a **for loop** over the training set and compute the derivatives to respect each training example and then add them up. Loop over 1 to M, where M is the number of training examples we compute $z^{(i)} = w^T x + b$, 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.

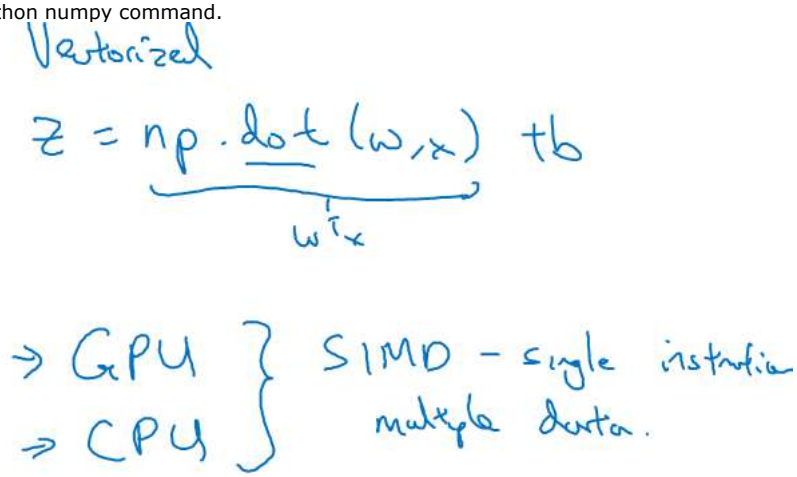
Unit 2 - Python and Vectorization
Vectorization
Vectorization is basically the art of getting rid of explicit for loop in your code. In the deep learning era, you often find yourself training on relatively large data sets, because that's when deep learning algorithms tend to shine. And so, it's important 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:

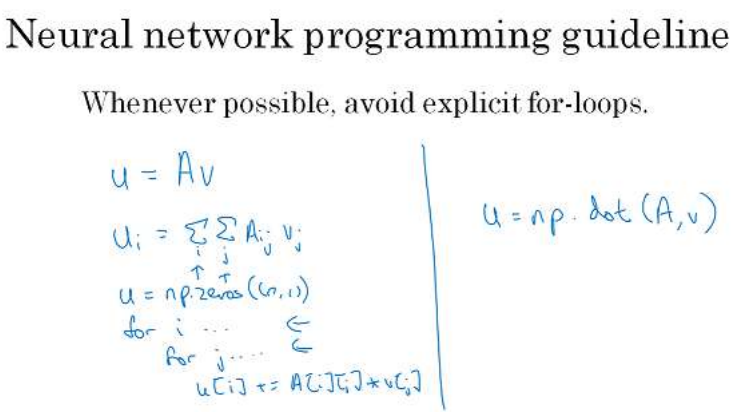
```
250286.989866
Vectorized version: 1.50275523040771484 ms
250286.989866
For loop: Non-Vectorized version: 474.29513931274414 ms
```

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

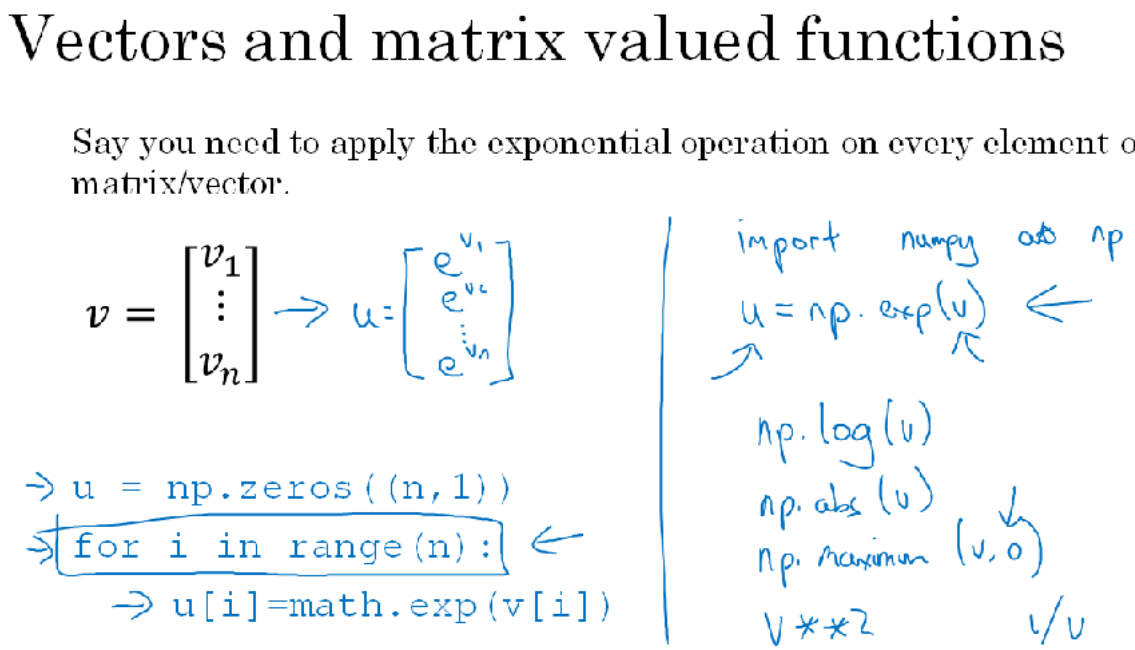
In last section we saw a example of how vectorization, by using built in functions and by avoiding explicit for loops, allows you to speed up your code significantly. Let's look at a few more examples.

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.



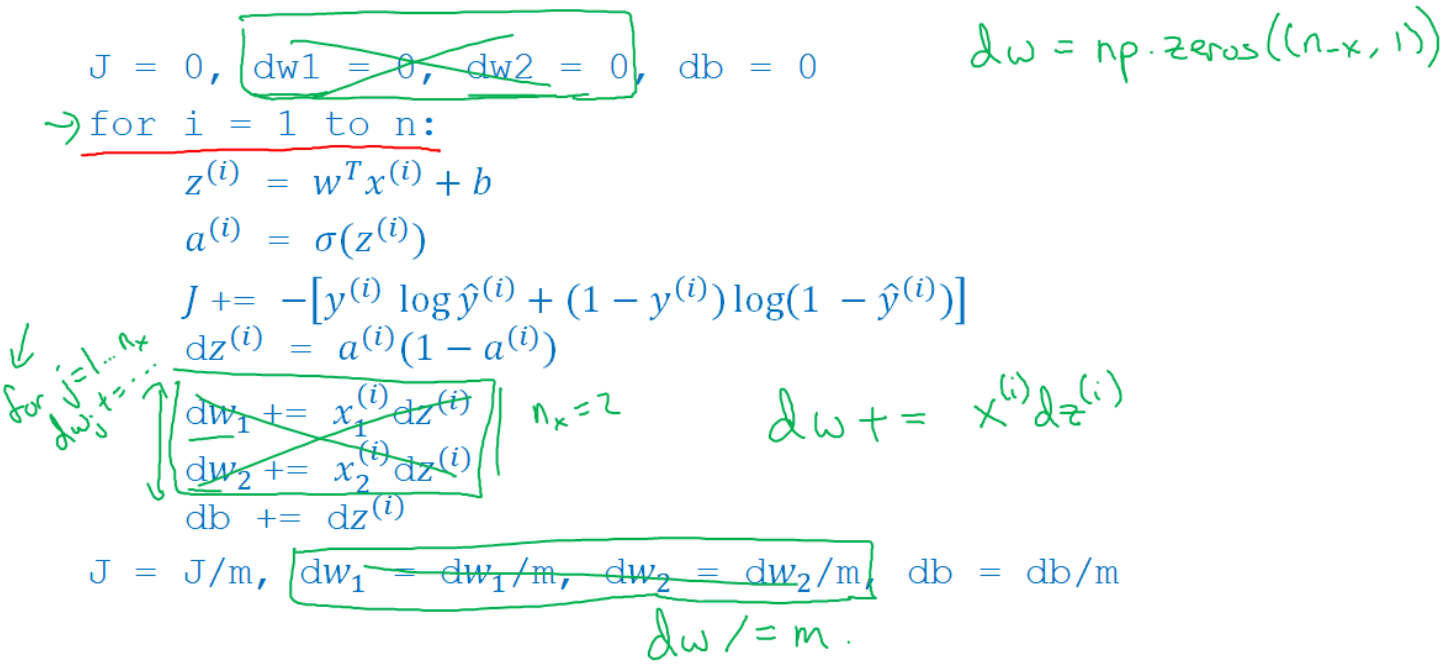
See in diagram a example which computes a vector u as the product of the matrix A, and another vector in non-vectorized implementation we have to use two for-loops, looping over both i and j (see left side of the diagram). In the vectorized implementation we just say u equals np dot (A,v) implementation 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

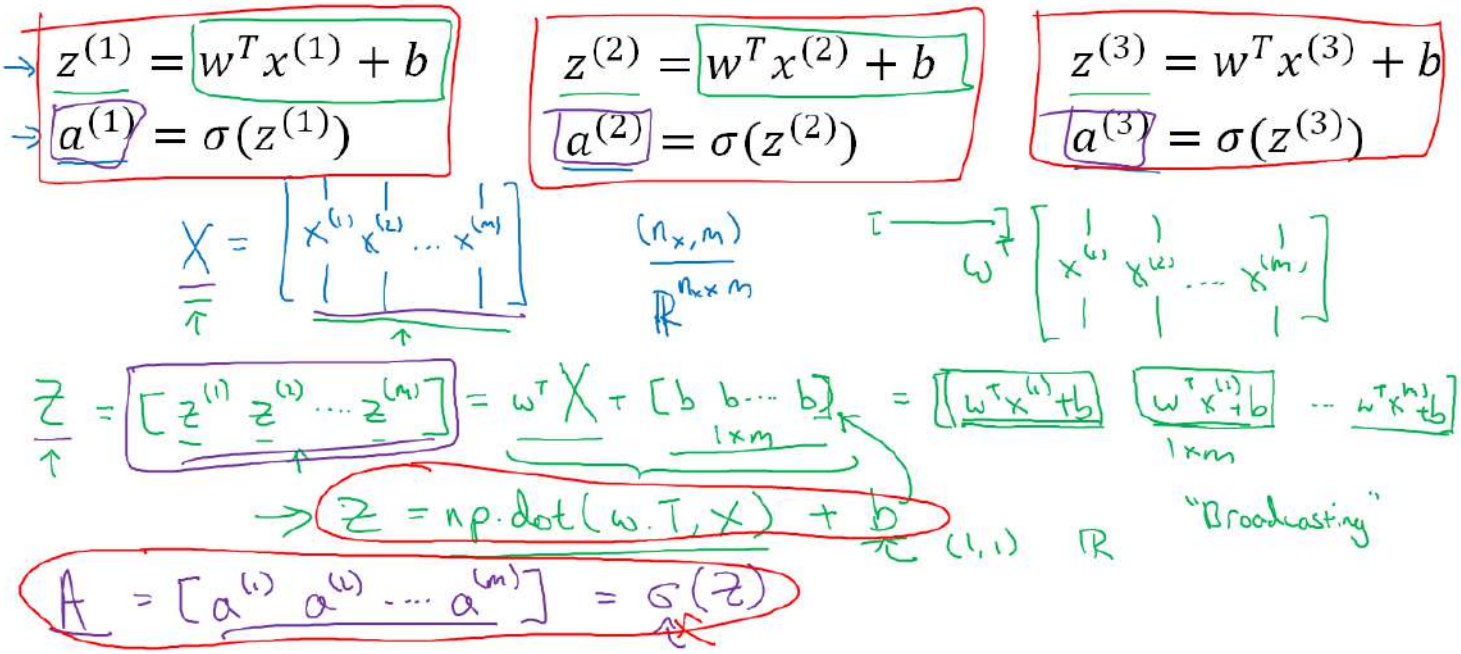


Looking into our code for computing the derivatives for logistic regression, and we had two for-loops. So in our example we had nx equals 2, but if you had more features than just 2 features then you'd need have a for-loop over dw1, dw2, dw3, and so on. Here instead of second for loop over the individual 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

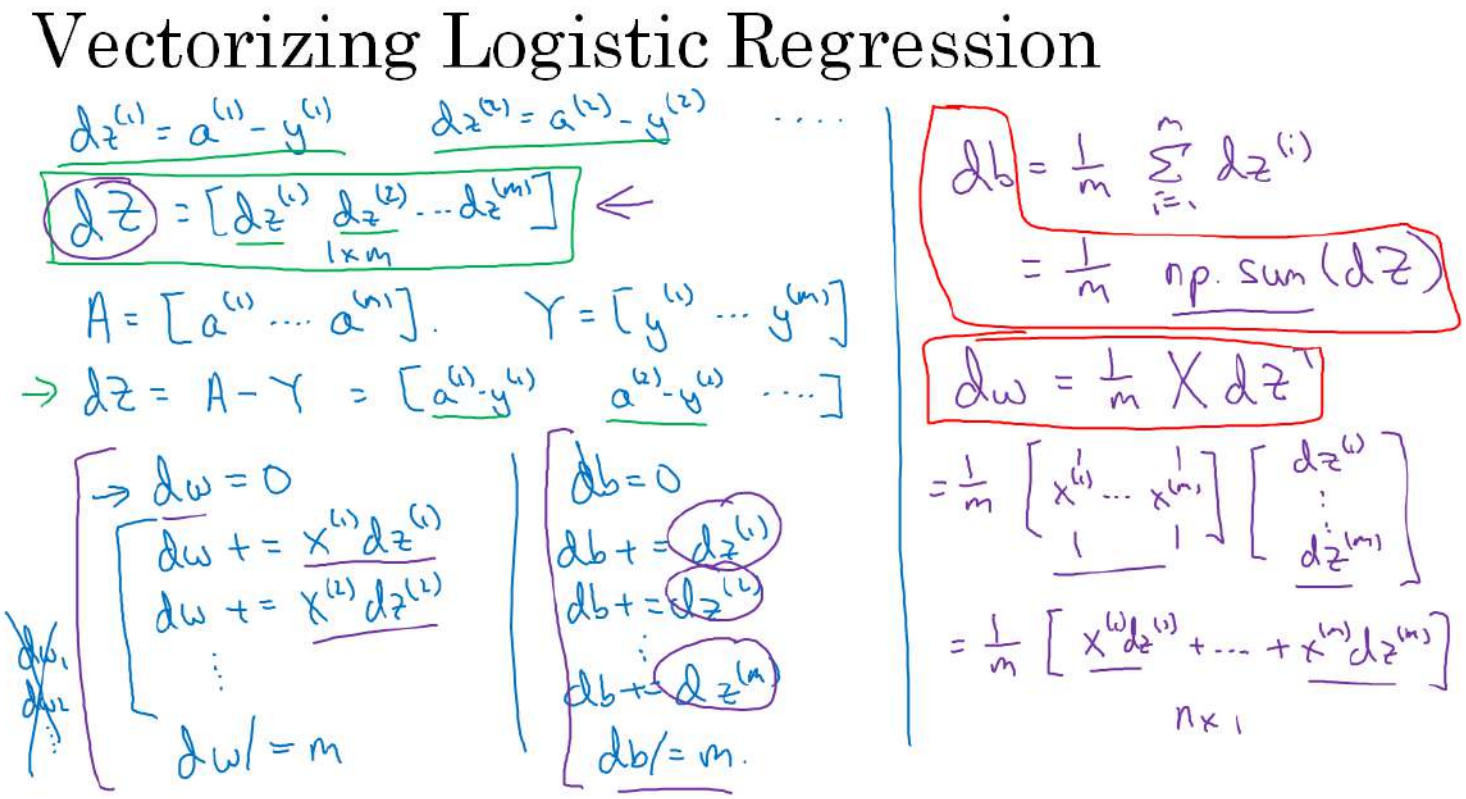


Let's first examine the forward propagation steps of logistic regression. So, if you have M training examples, then to make a prediction on the first example, you need to compute compute z, then compute the activations. Then to make a prediction on the second training example, third example and so on (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))
```

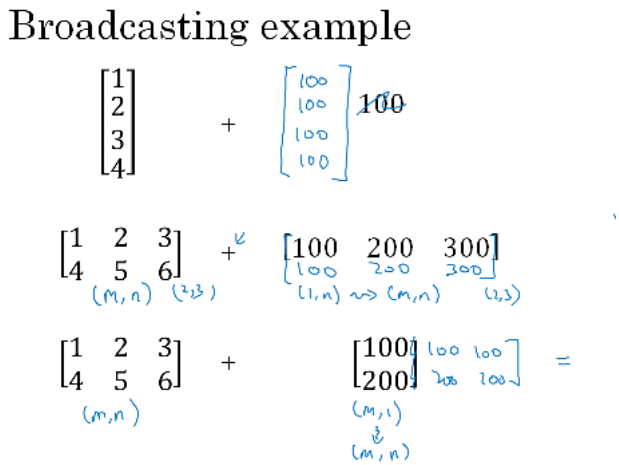
(3,4) / (1,4)

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.

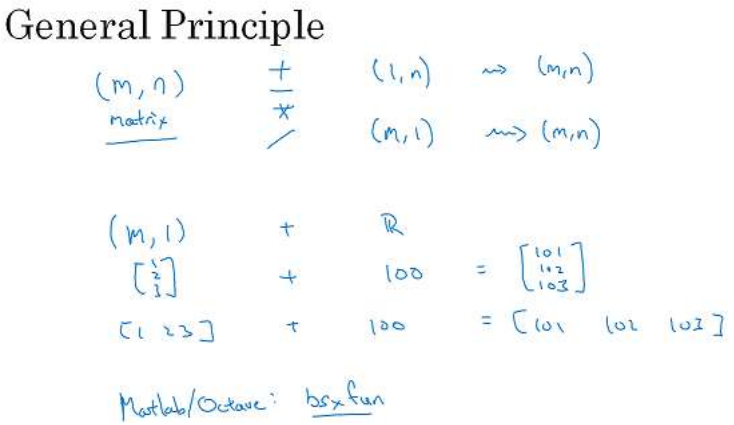
```
import numpy as np
A = np.array([56.0,0.0,4.4,68.0],
             [1.2,104.0,52.0,8.0],
             [1.8,135.0,99.0,0.9])
print (A)
cal = A.sum(axis=0)
print (cal)
percentage = 100*A/cal/reshape(1,4)
print (percentage)
```

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

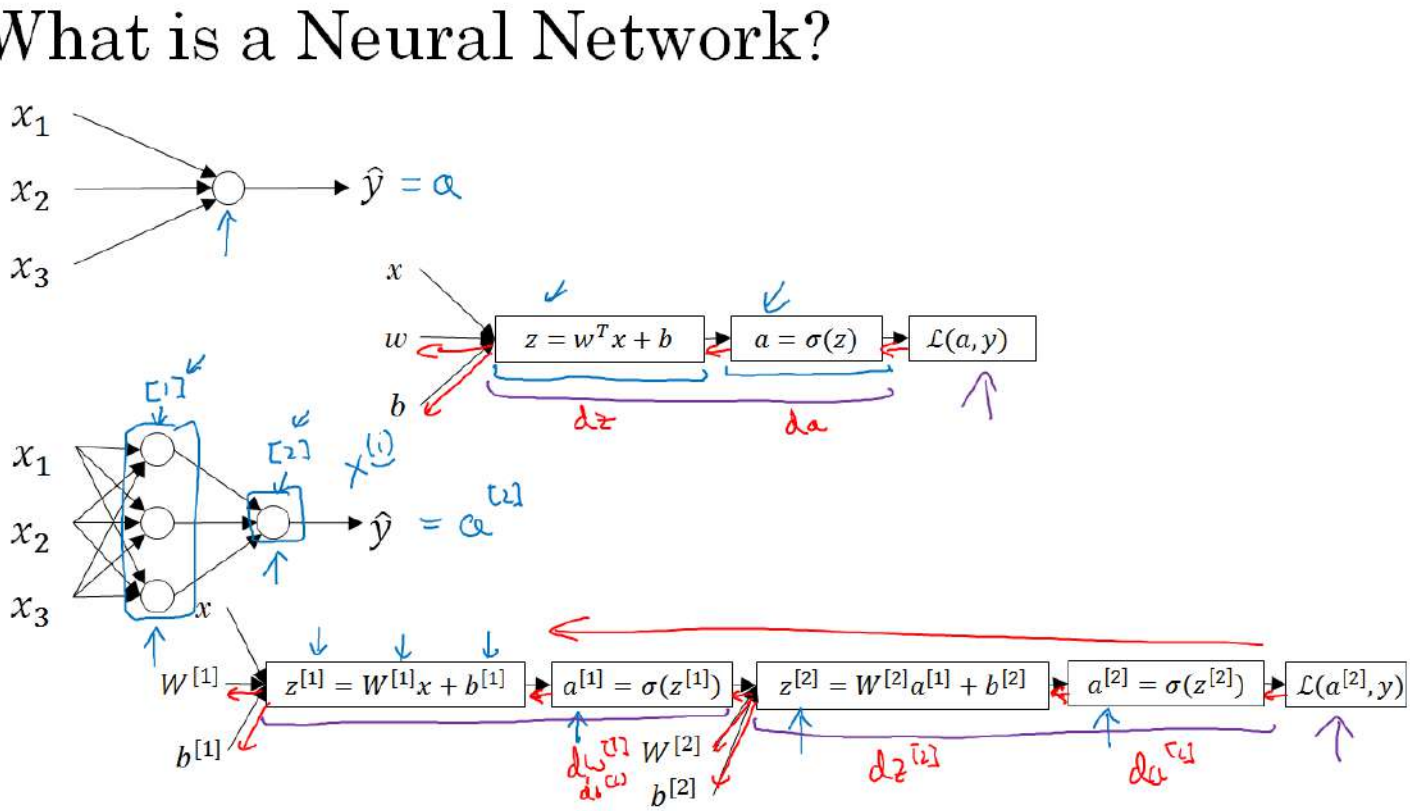
- Understand hidden units and hidden layers
- Be able to apply a variety of activation functions in a neural network.
- Build your first forward and backward propagation with a hidden layer
- Apply random initialization to your neural network
- Become fluent with Deep Learning notations and Neural Network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

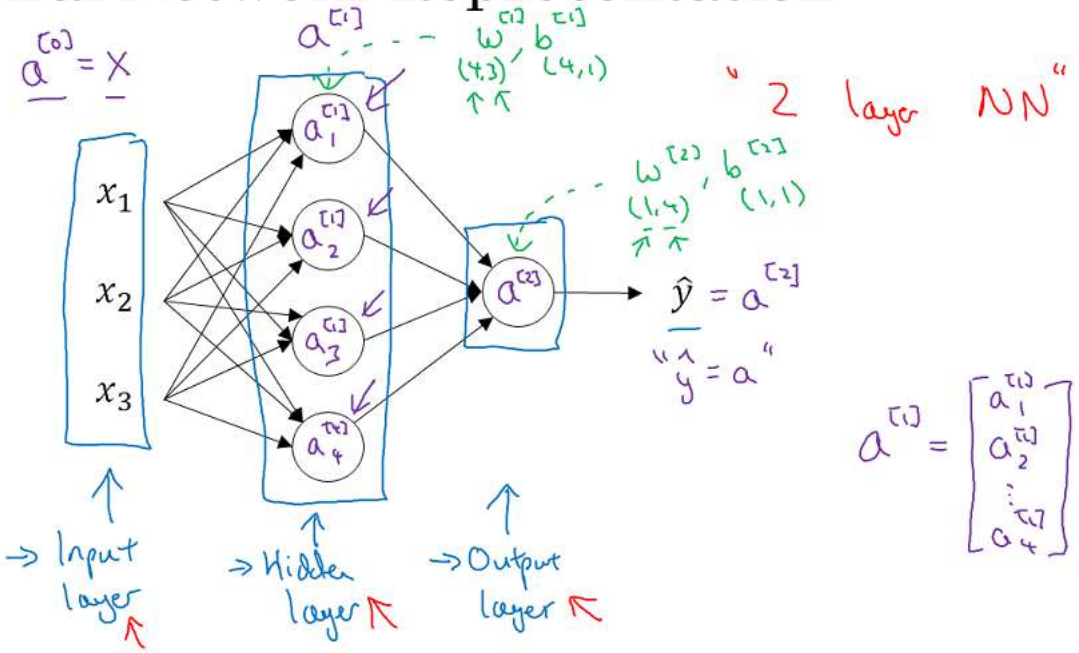


This diagram shows a quick overview of how you implement neural network. In last section we had talked about logistic regression and we saw how this model corresponds to the computation graph (as shown in diagram) where you put the input features **x** and parameters **w** and **b**, that allows you 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.

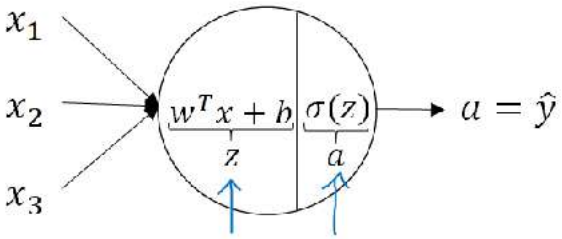
Let's introduce few more notation. Previously, we were using the vector x to denote the input features and the alternative notation for the values of the input features will be $\mathbf{a^{[0]}}$ and the term **a** also stands for activations and it refers to the values that different layers of the neural 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

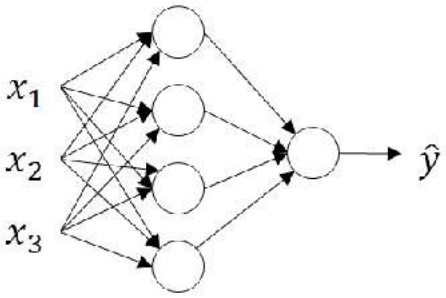
in the last section we saw what a single hidden layer neural network looks like. In this section we'll go through the details of exactly how this neural network compute outputs. This is a 2-steps of computation first you compute z and in second you compute the activation as a sigmoid function of z. Check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

Neural Network Representation

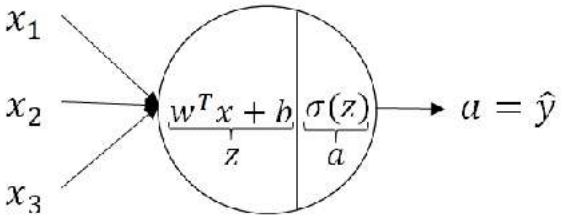


$z = w^T x + b$

$a = \sigma(z)$

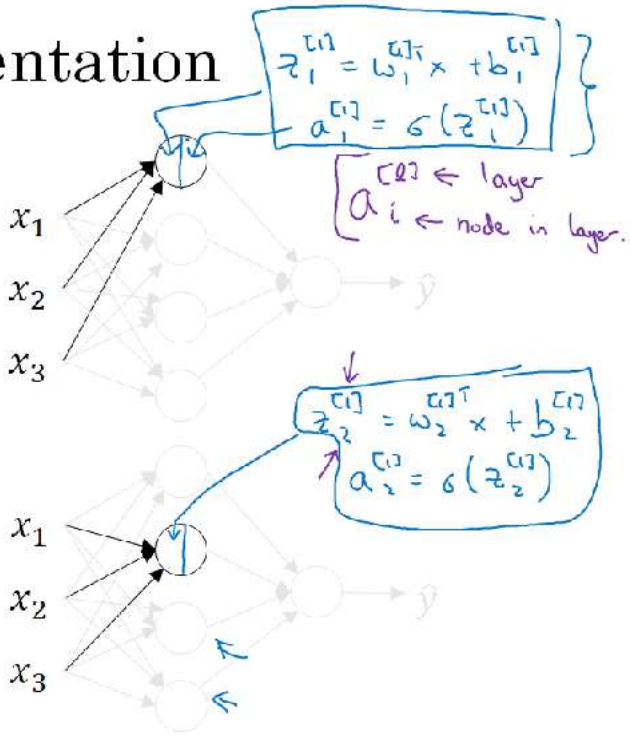


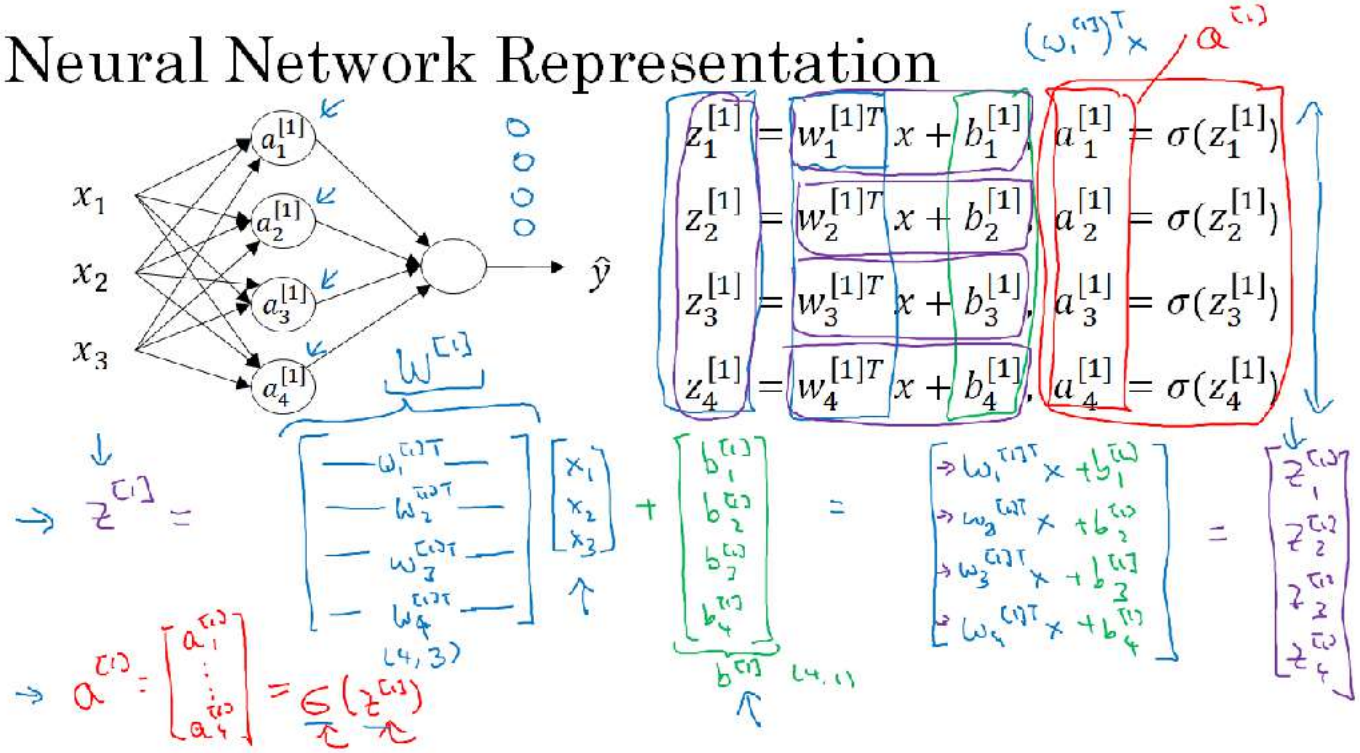
Neural Network Representation

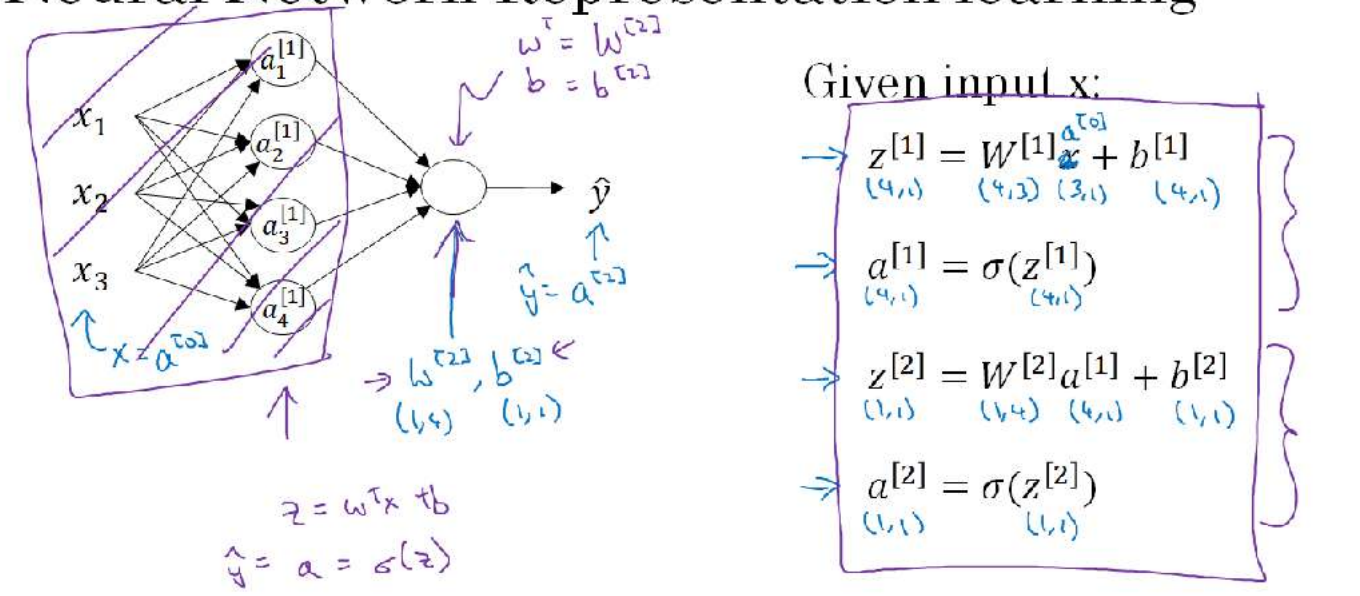


$z = w^T x + b$

$a = \sigma(z)$





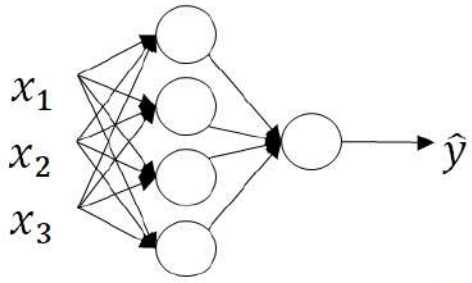


So looking into diagram, we can see that to compute the output of this neural network all you need is 4 four lines of code (right side the last image)

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)

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

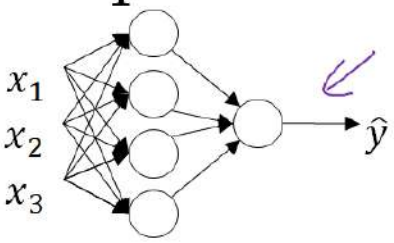
$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$

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
```

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

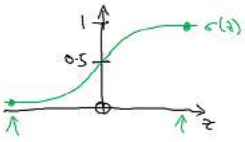
$A^{[0]} \quad X = a^{[0]} \quad x^{(i)} = a^{[0](i)}$

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

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

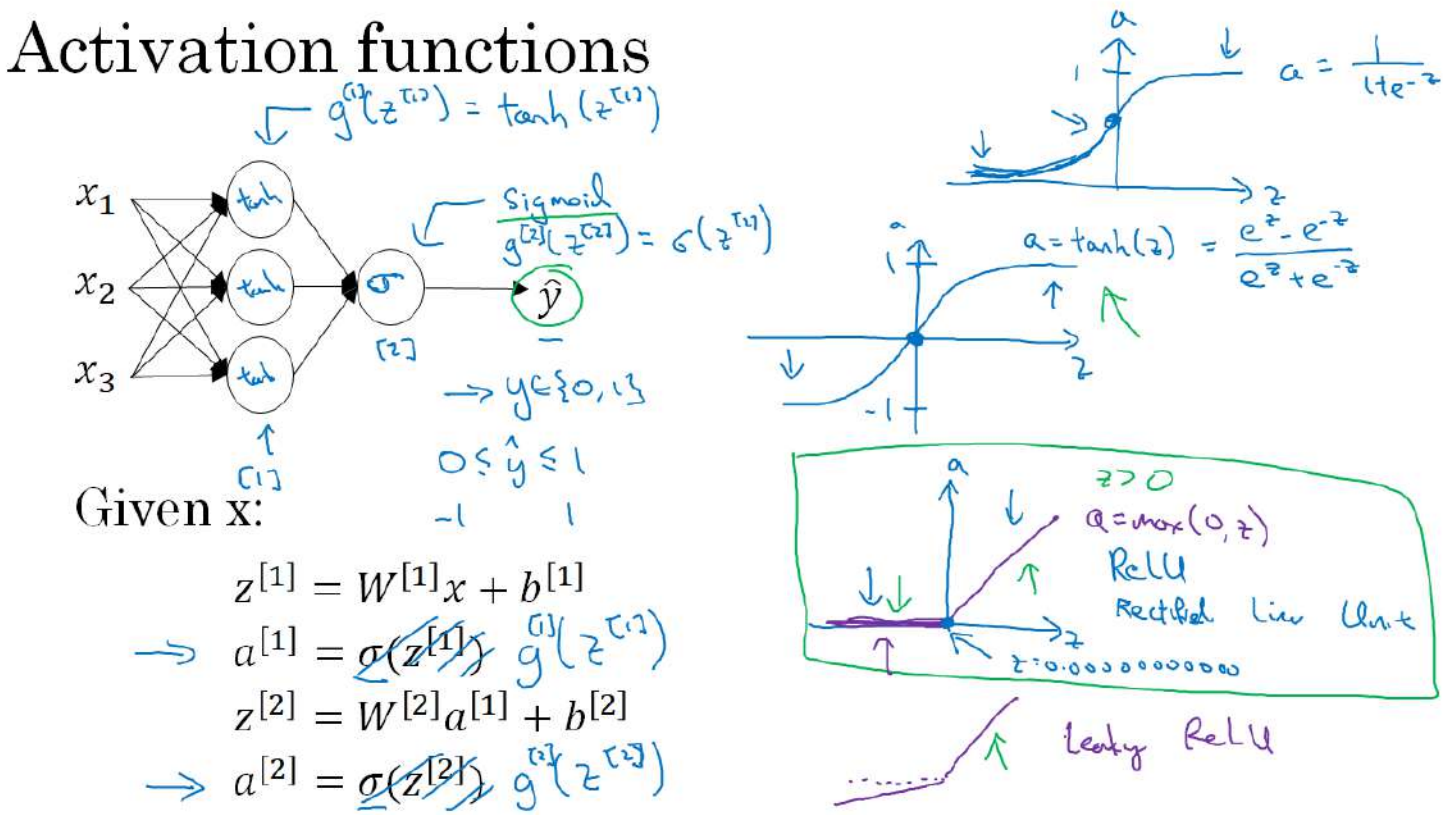


So in more general case we can have a different function **g** of **z** where **g** could be a nonlinear function that may not be the **sigmoid** function. As we have seen **sigmoid** function goes between 0 & 1, an activation function that almost always works better than the **sigmoid** function is 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.

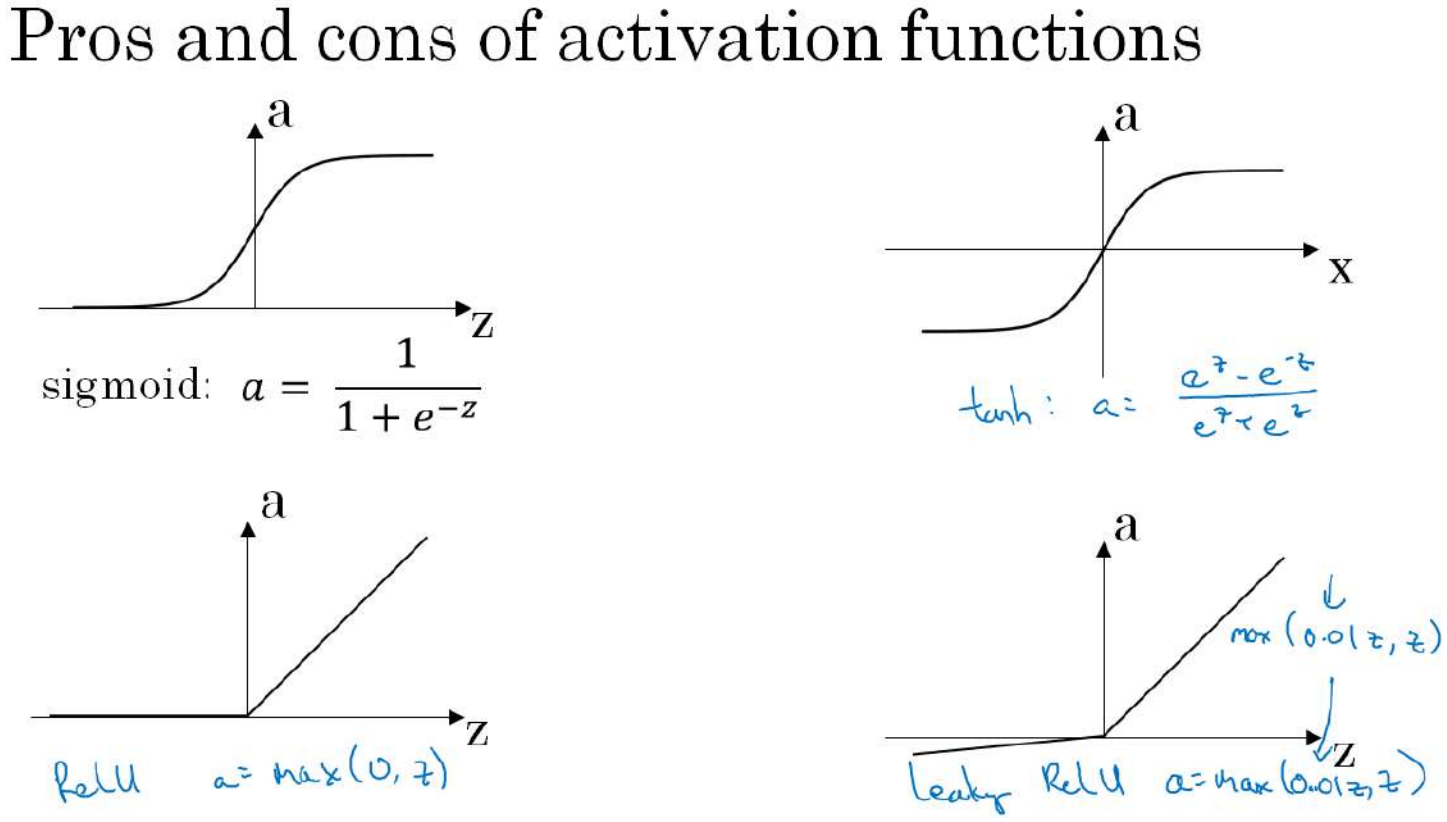
Takeaway from the current understanding is that no need to use the **sigmoid** activation as **tanh** function is almost always strictly superior the one exception is for the output layer because if **y** is either 0 or 1 then it makes sense for **yhat** to be a number that you want to output between 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:

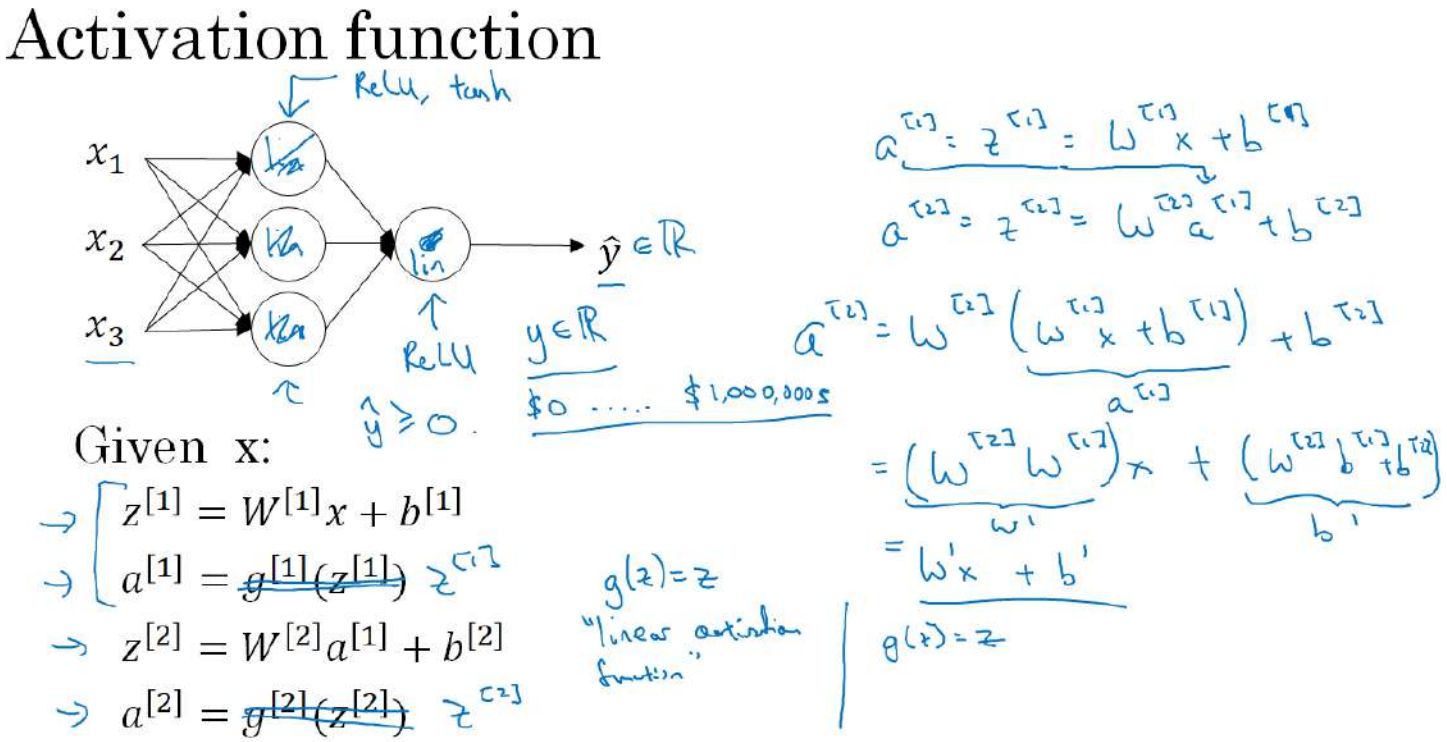


First is the **sigmoid** activation function, never use this except for the **output layer** if you are doing **binary classification** or maybe almost never use this and the reason never use this is because the **tanh** is pretty much strictly superior. The default most commonly used activation function 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.



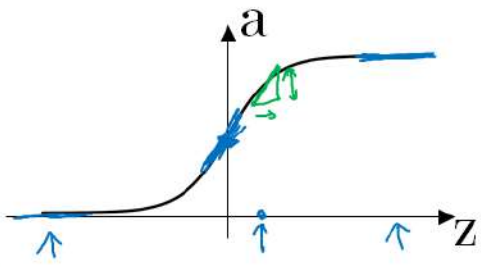
In the diagram for a given **x**, we have 4 forward propagation equations for the neural network. One of thing we can try is to get rid of the function **z** and set $\mathbf{a^{[1]} = z^{[1]}}$ or alternatively we could say that **g(z) = z** sometimes this is called the **linear activation function** 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

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

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

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

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

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

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

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

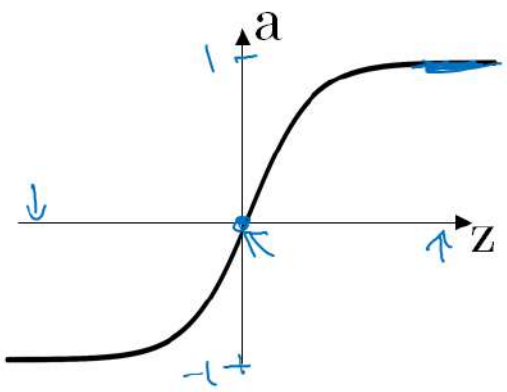
$z = 0$ $\quad g(z) = \frac{1}{2}$

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

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

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

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

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

$g'(z) \approx 0$

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

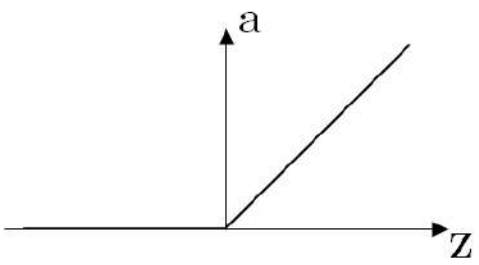
$g'(z) \approx 0$

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

$g'(z) = 1$

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

ReLU and Leaky ReLU



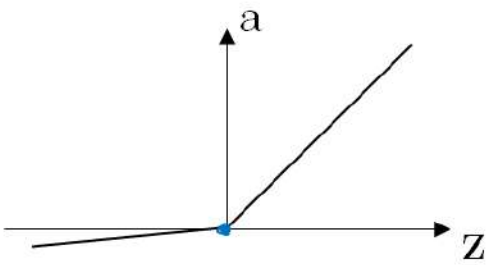
ReLU

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

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

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

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

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

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

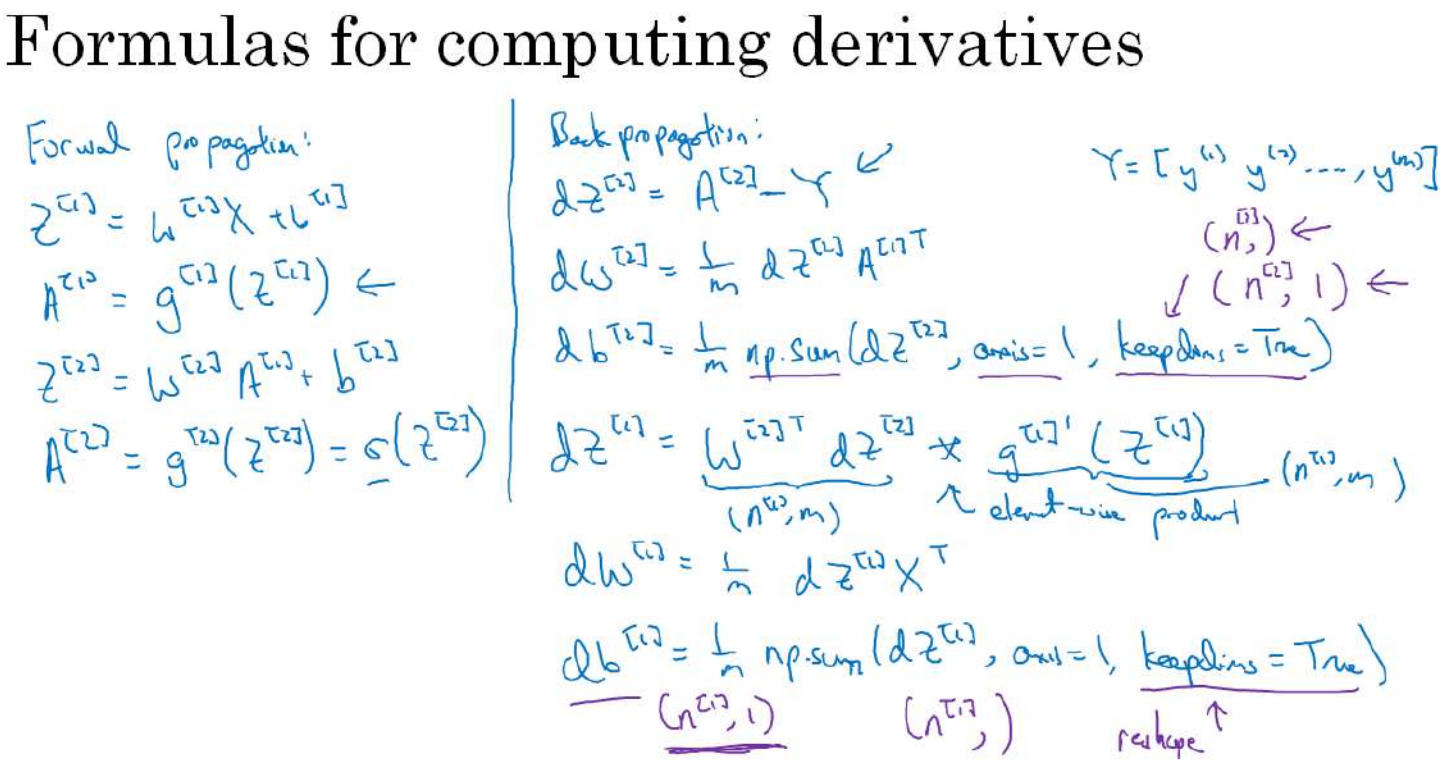
$W^{[2]} := \ldots$ $b^{[2]} := \ldots$

}

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

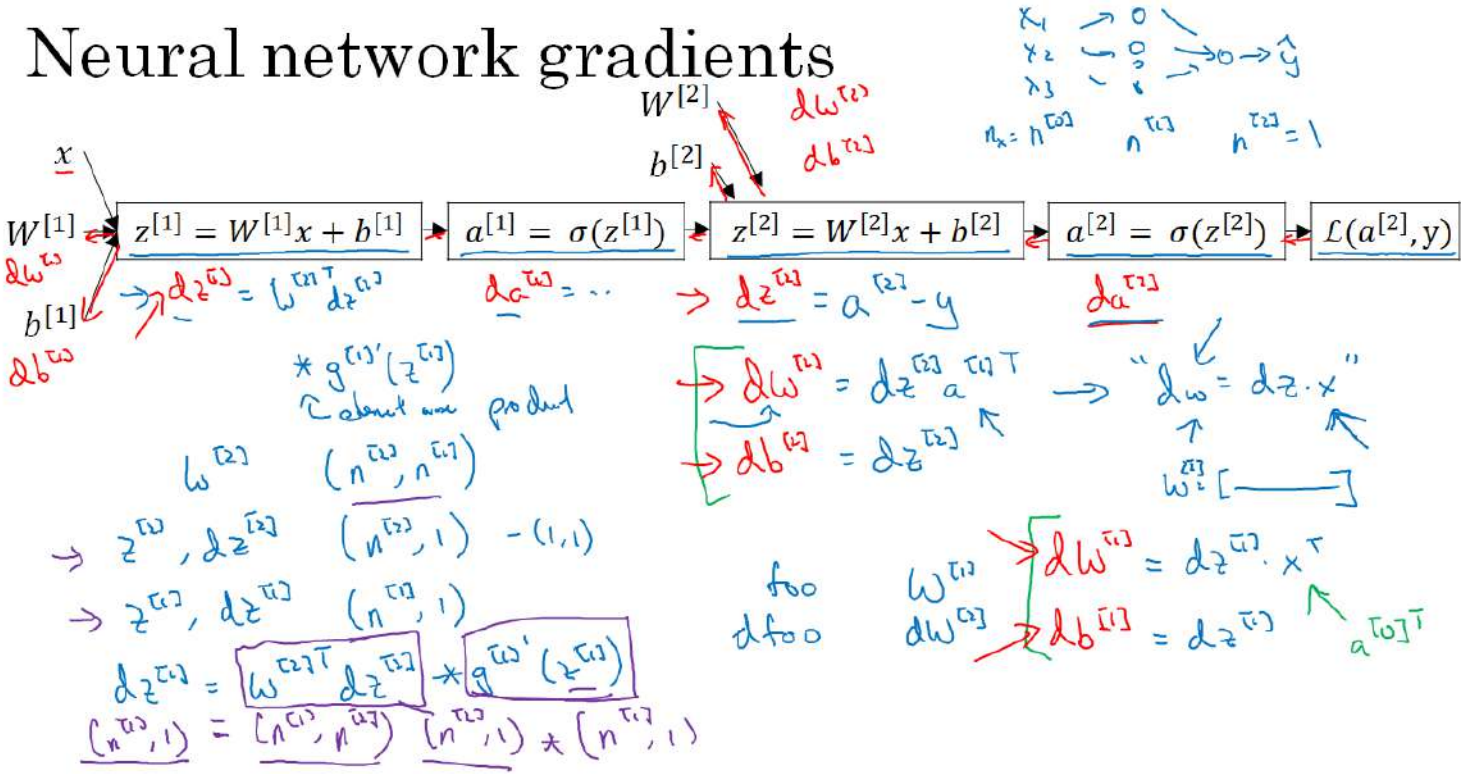
need to compute the derivative so you need to compute **Dw [1]** which is the derivative of the cost function with respect to the parameter w1, then you need to compute another variable which is going to call **Db [1]** which is the derivative or the slope of your cost function with respect to the variable **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.



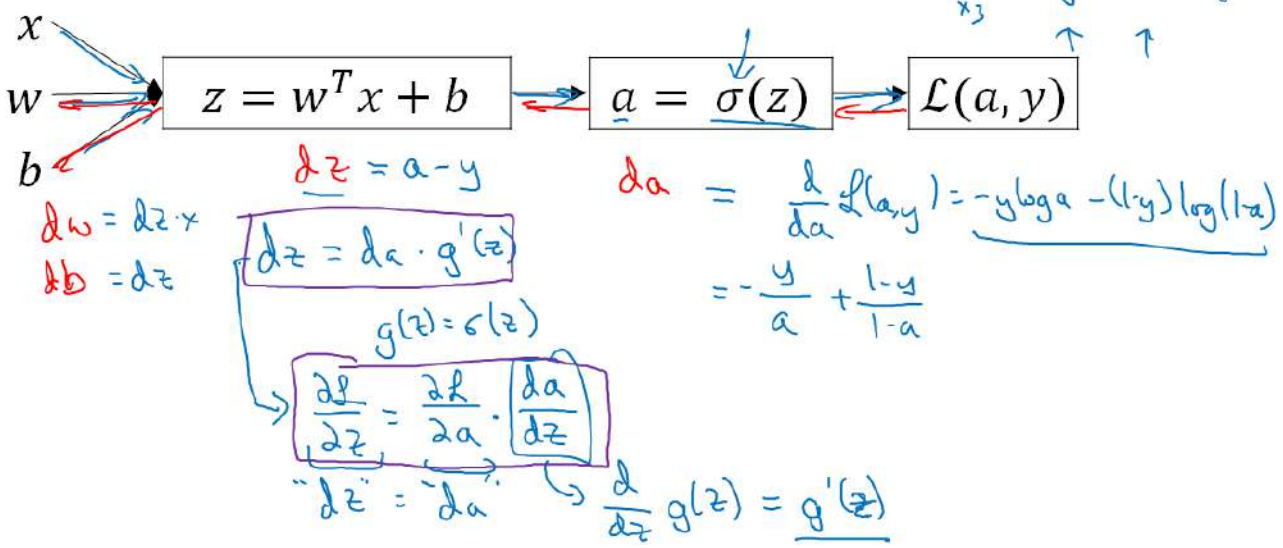
if we assume we're doing binary classification then this activation function really should be the **sigmoid** function. So that's the forward propagation or the left-to-right forward computation for our neural network, next let's compute the derivatives so this is the back propagation step it computes **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

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

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

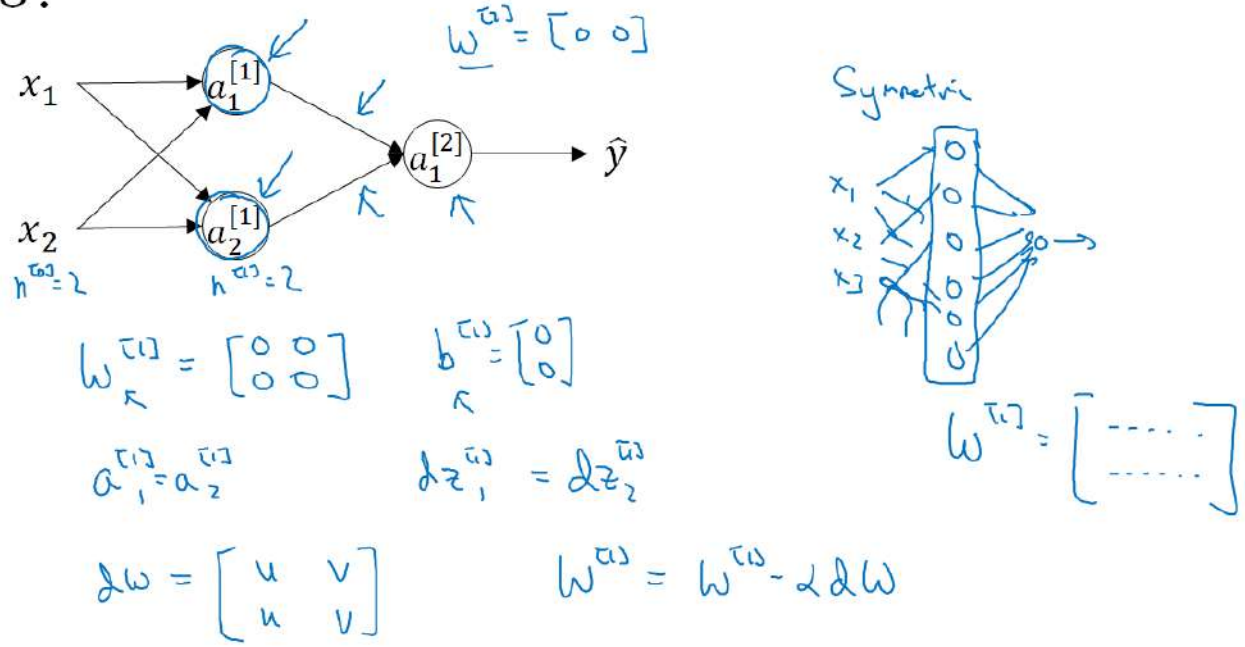
Random initialization

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:

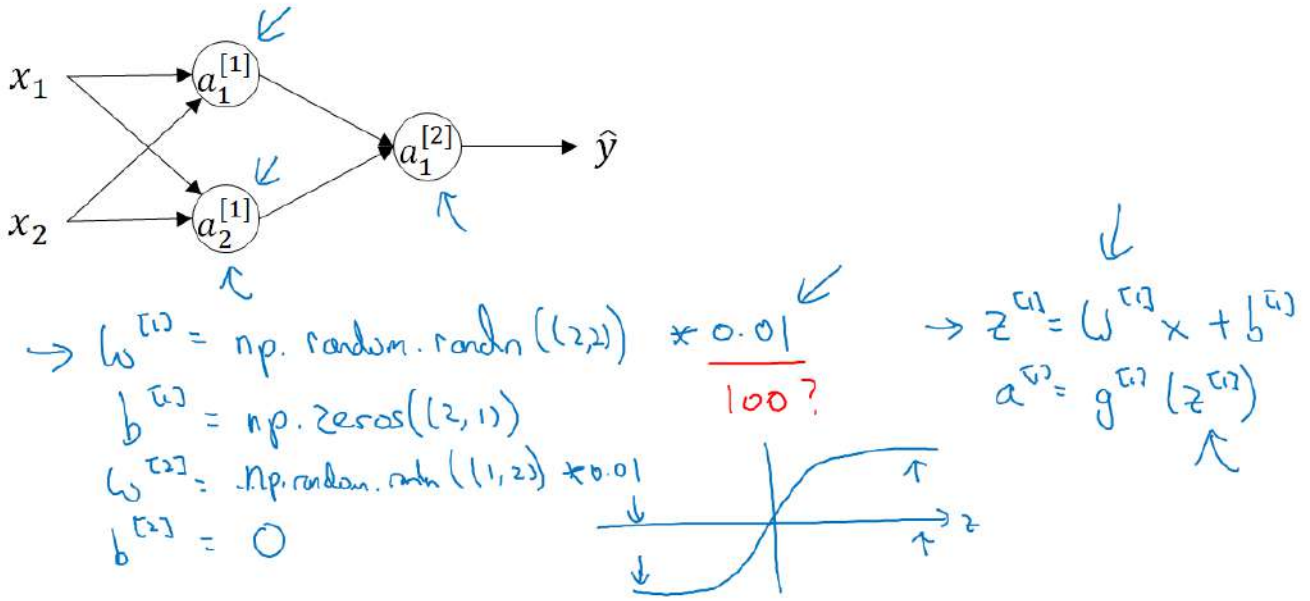
What happens if you initialize weights to zero?



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.

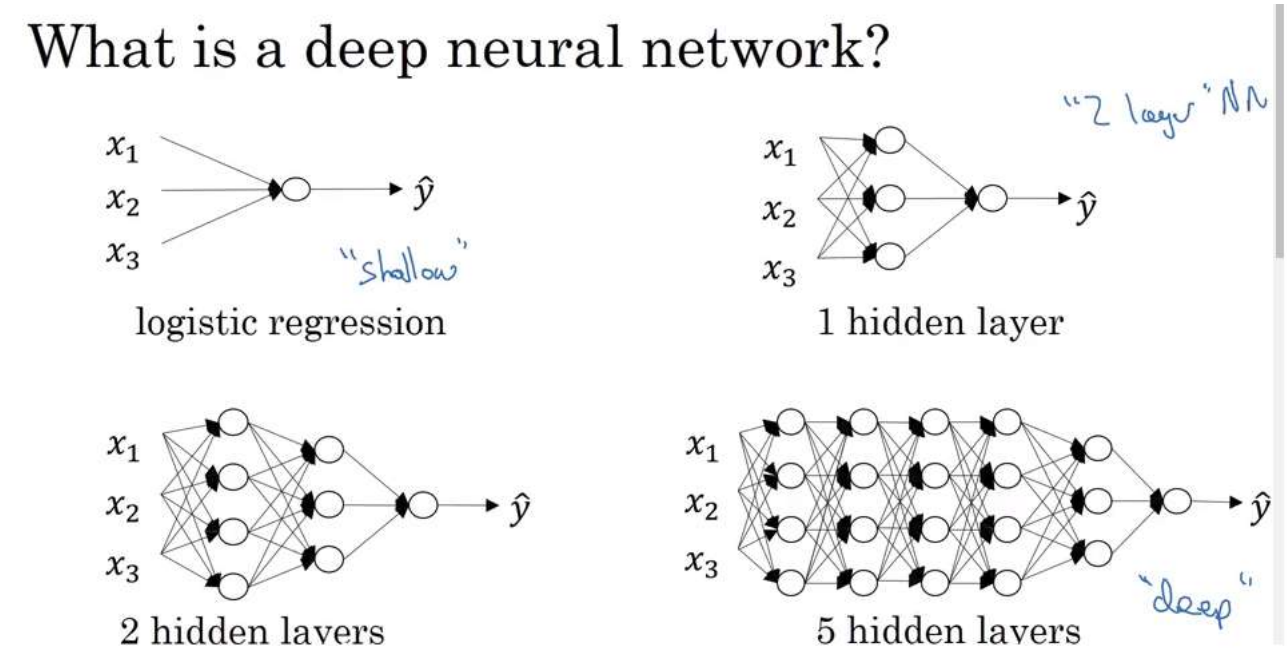
Week 3: Deep Neural Networks (Multiple layers)

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

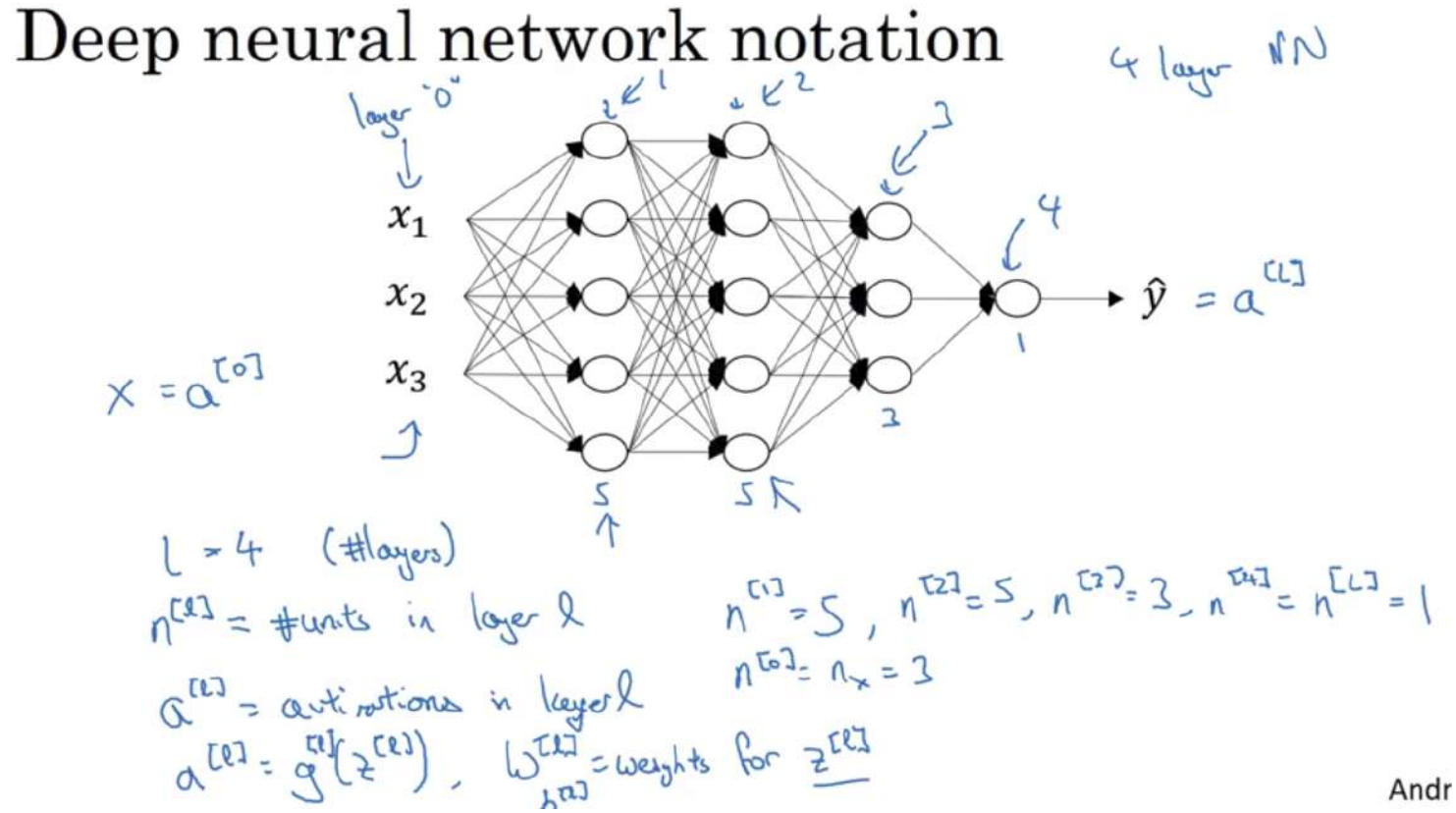
Deep L-layer Neural Network

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.

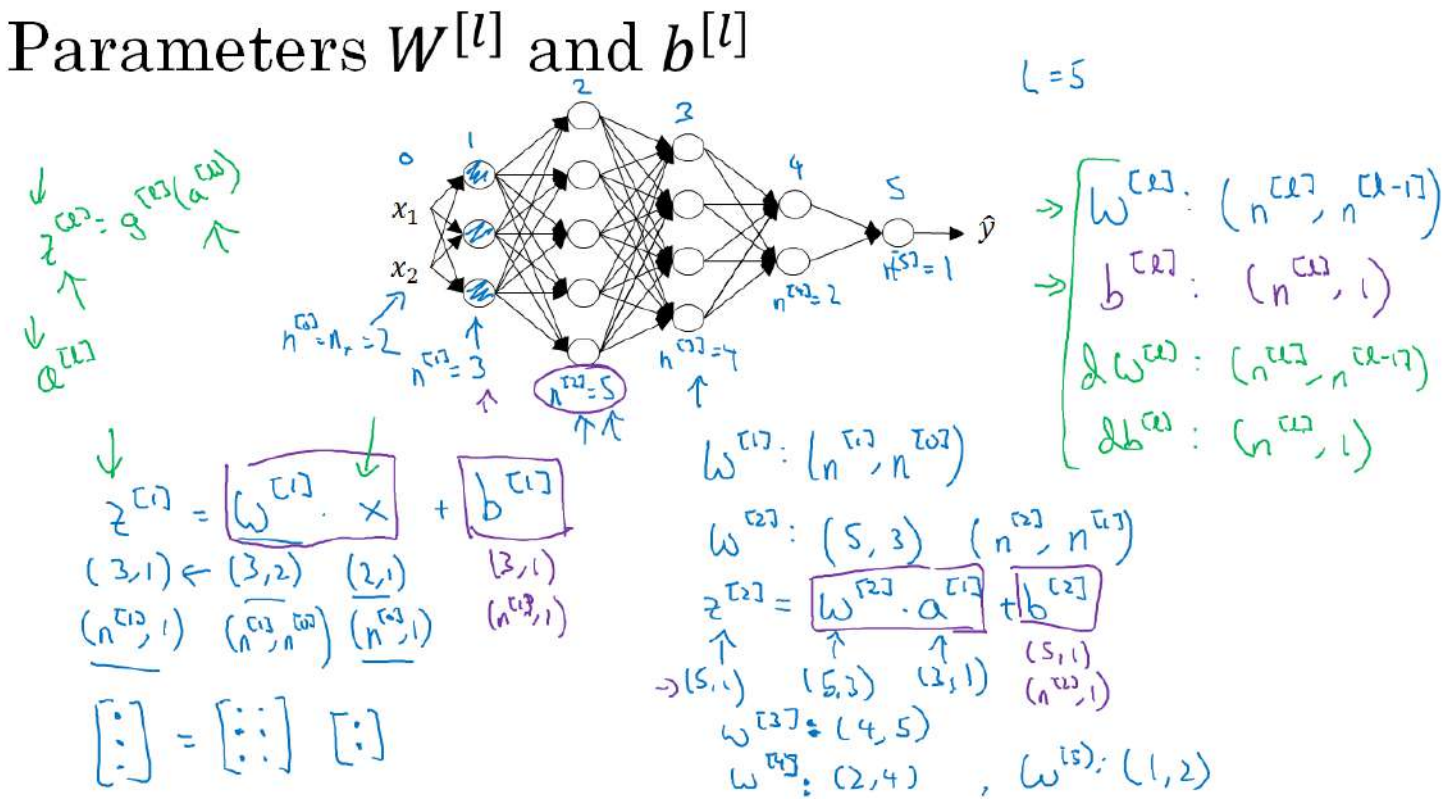
Forward Propagation in a Deep Network

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

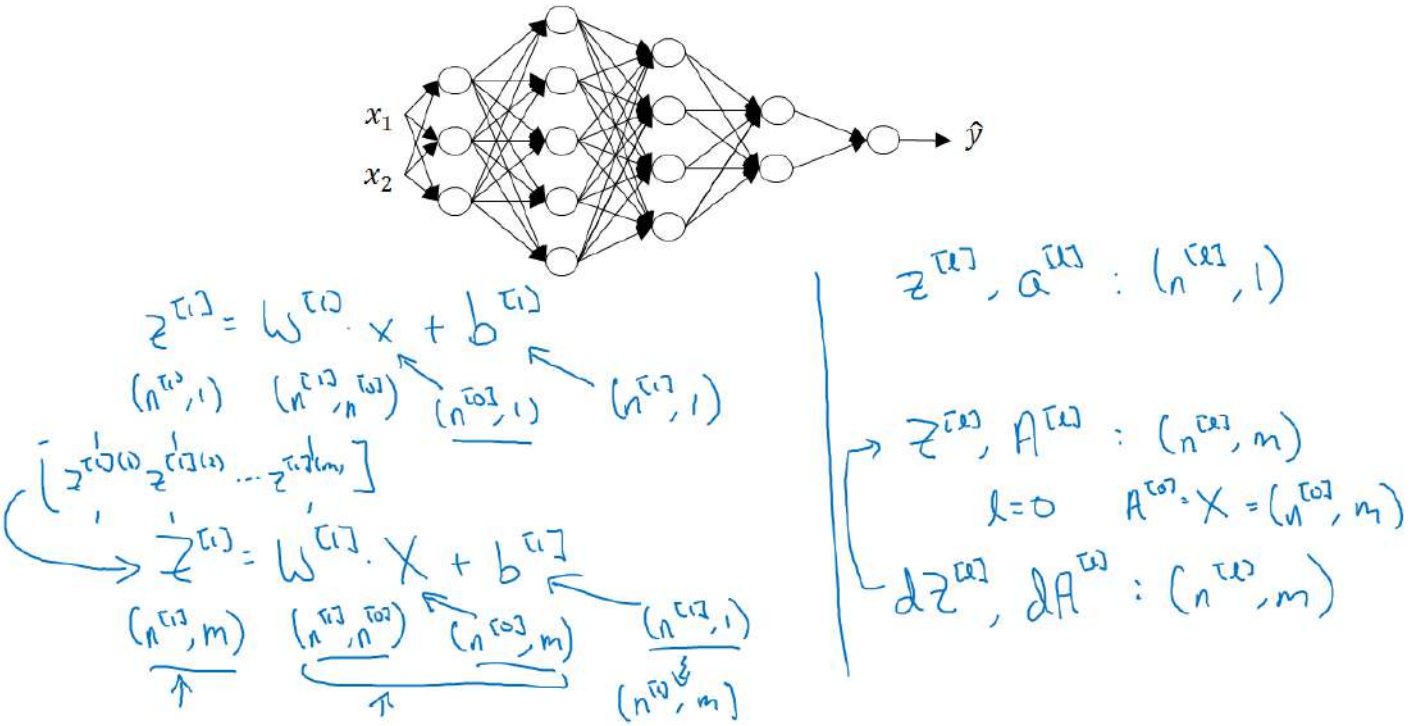
When implementing a deep neural network, one of the debugging tools people often use to check the correctness of the code.



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



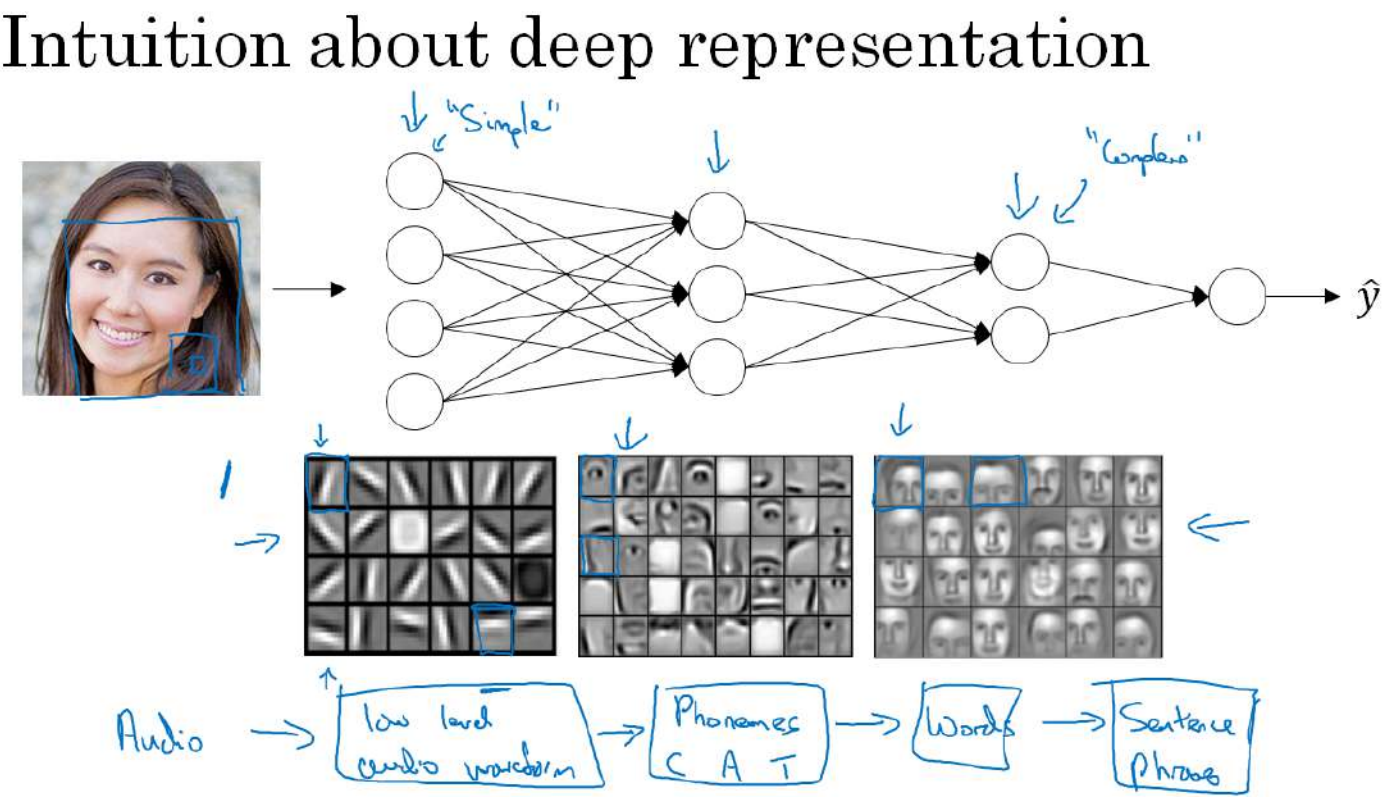
Check the above diagram. Now if you're implementing backpropagation then the dimensions of **dw** should be the same as dimension of **w** similarly **db** should be the same dimension as **b**. Now the other key set of quantities whose dimensions to check are these **z**, **x**, as well as $\mathbf{a^{[l]}}$ 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

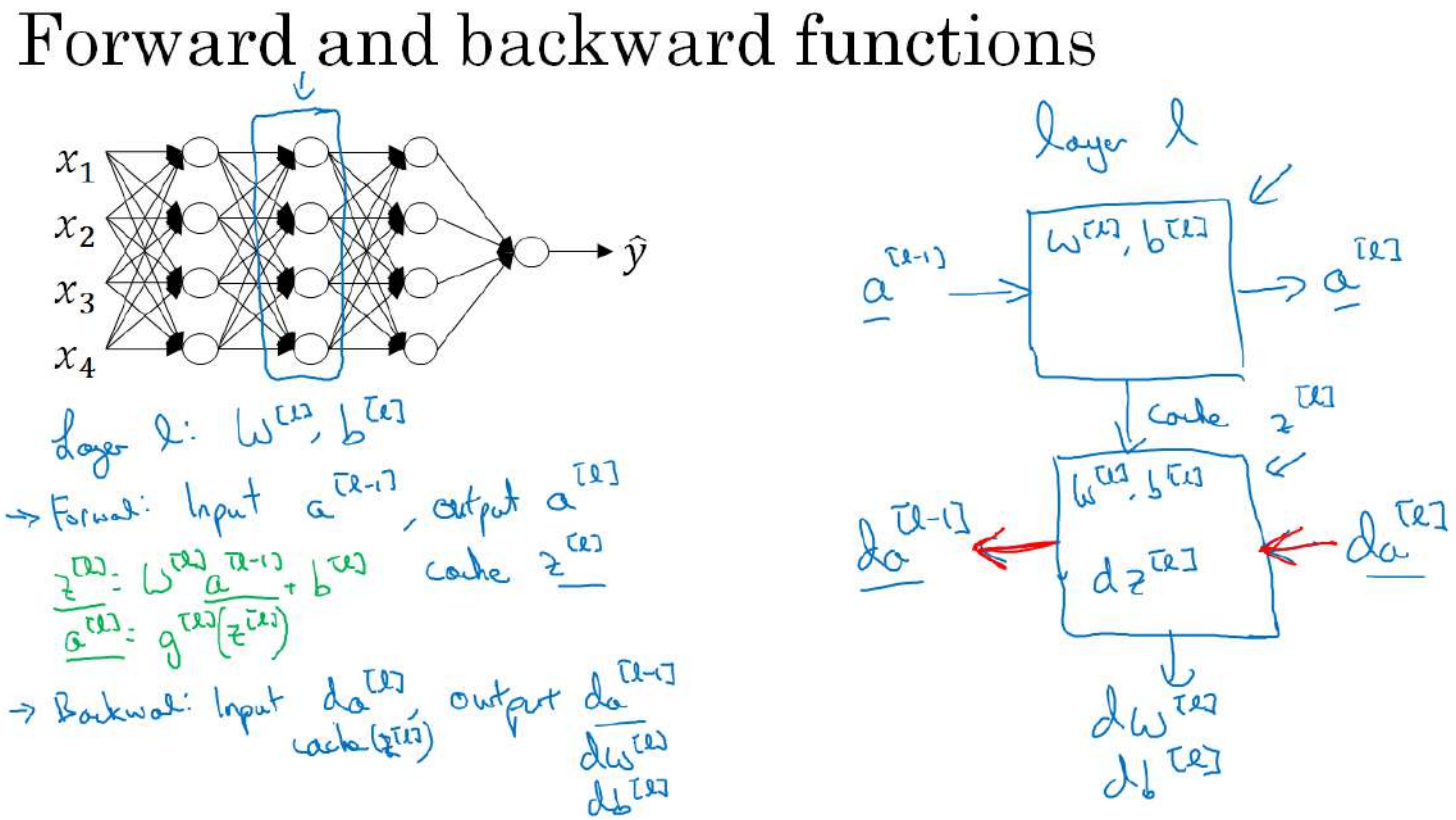
other types of data than images and face recognition as well.



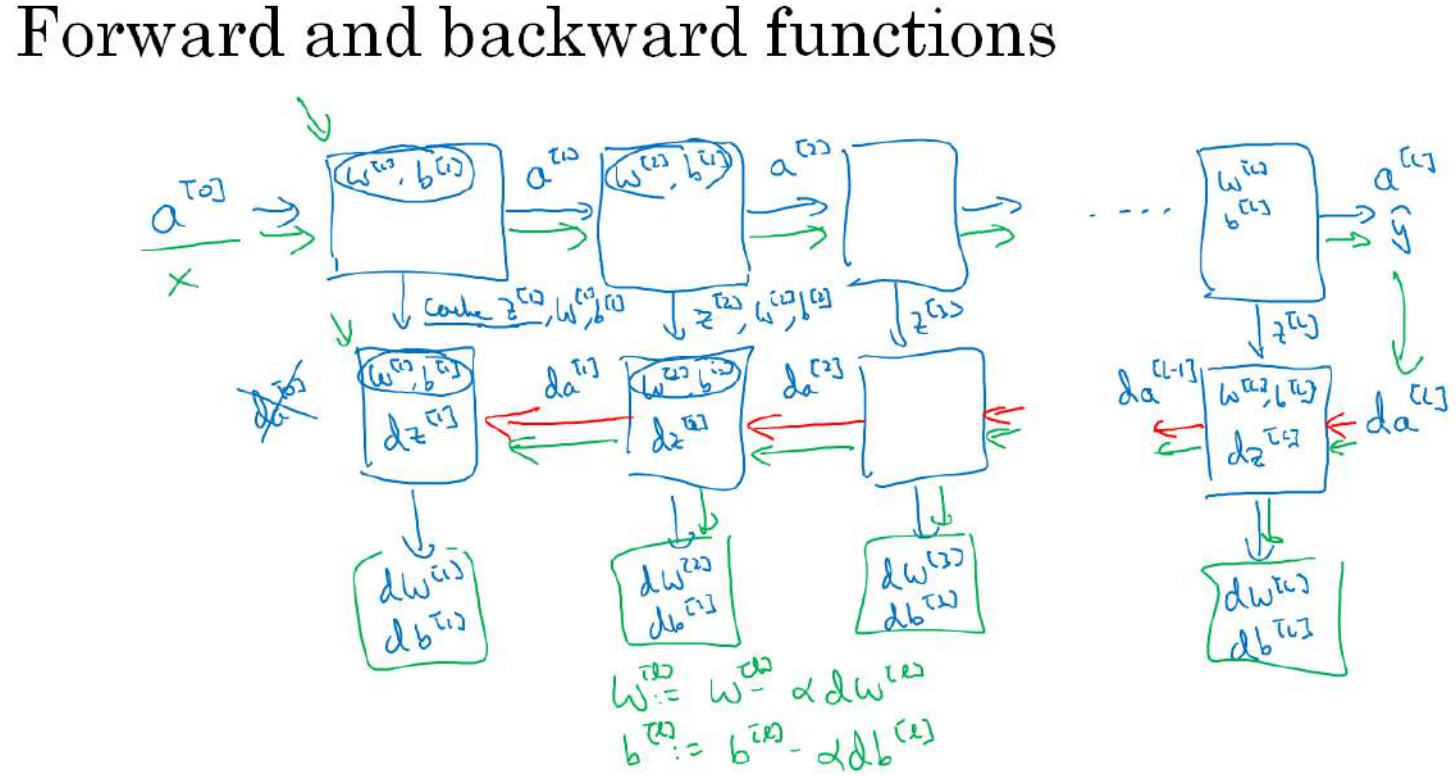
Another example, if you're trying to build a **speech recognition system**, it's hard to revisualize speech but if you input an audio clip then maybe the first level of a neural network might learnt to detect low level audio wave form features, such as is this tone is going up? Is it going down? Is it 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

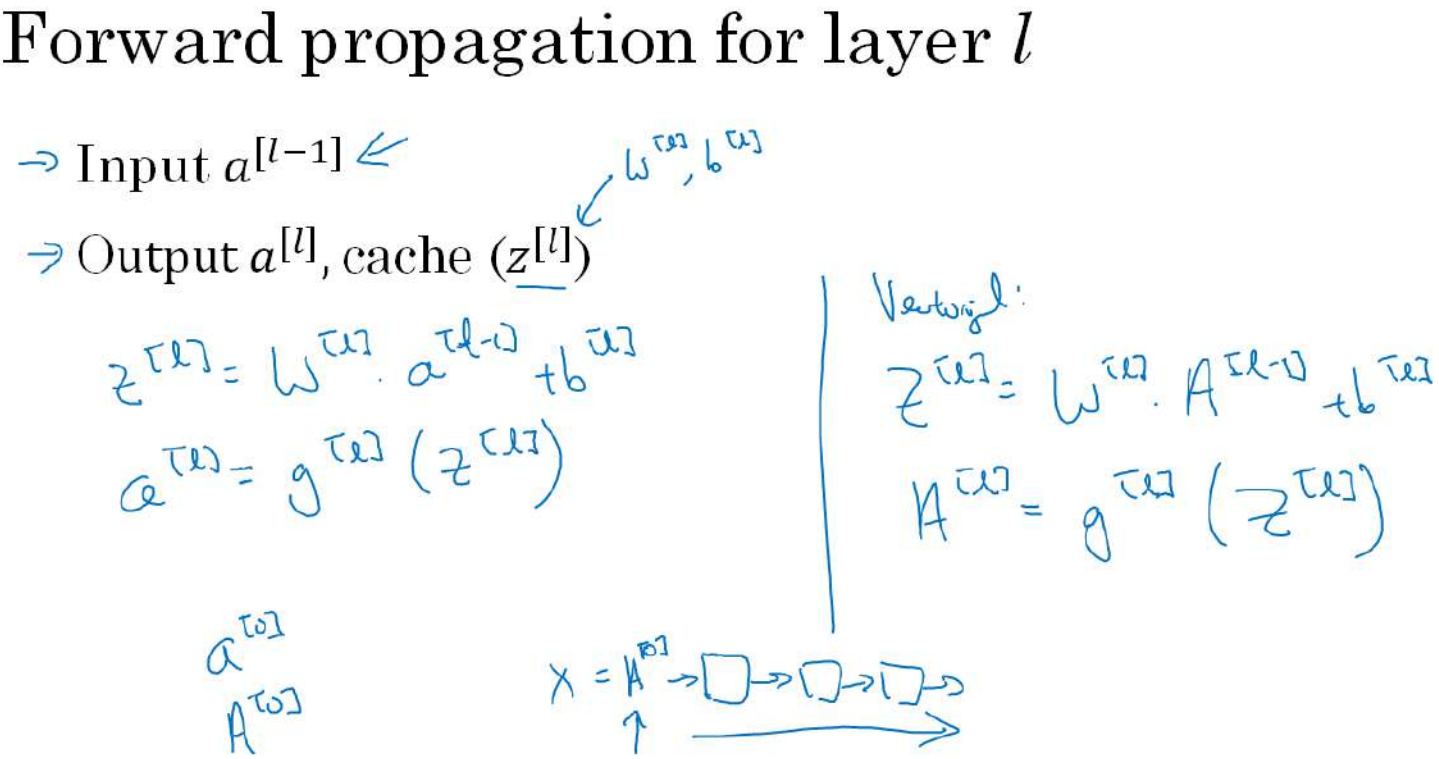


We have used NN with a few layers let's pick one layer and look at the computations focusing on just that layer for now check diagram for full flow of forward propagation nad backward propagation for that layer. This also show how we cache intermediate values for forward propagation and backward propagation.



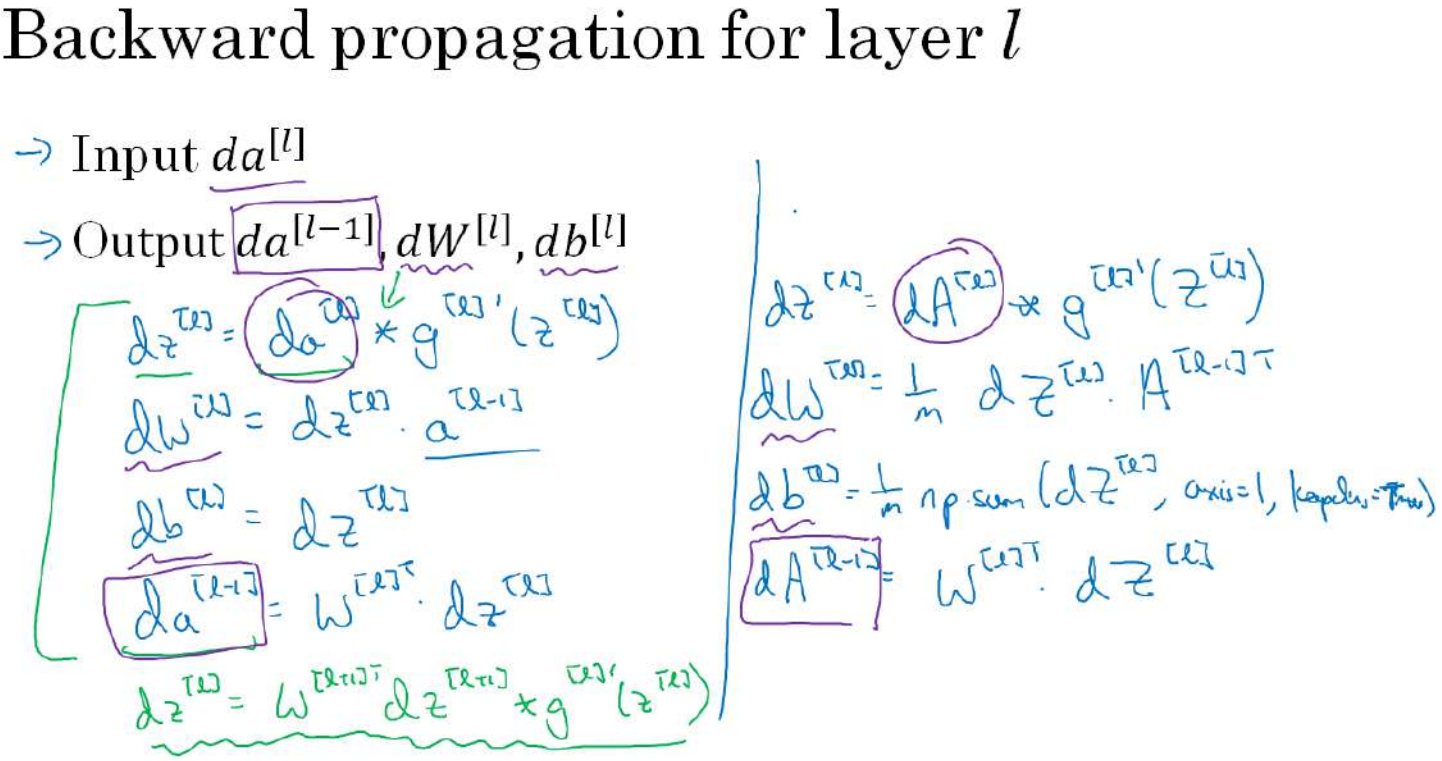
Forward and backward propagation

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.

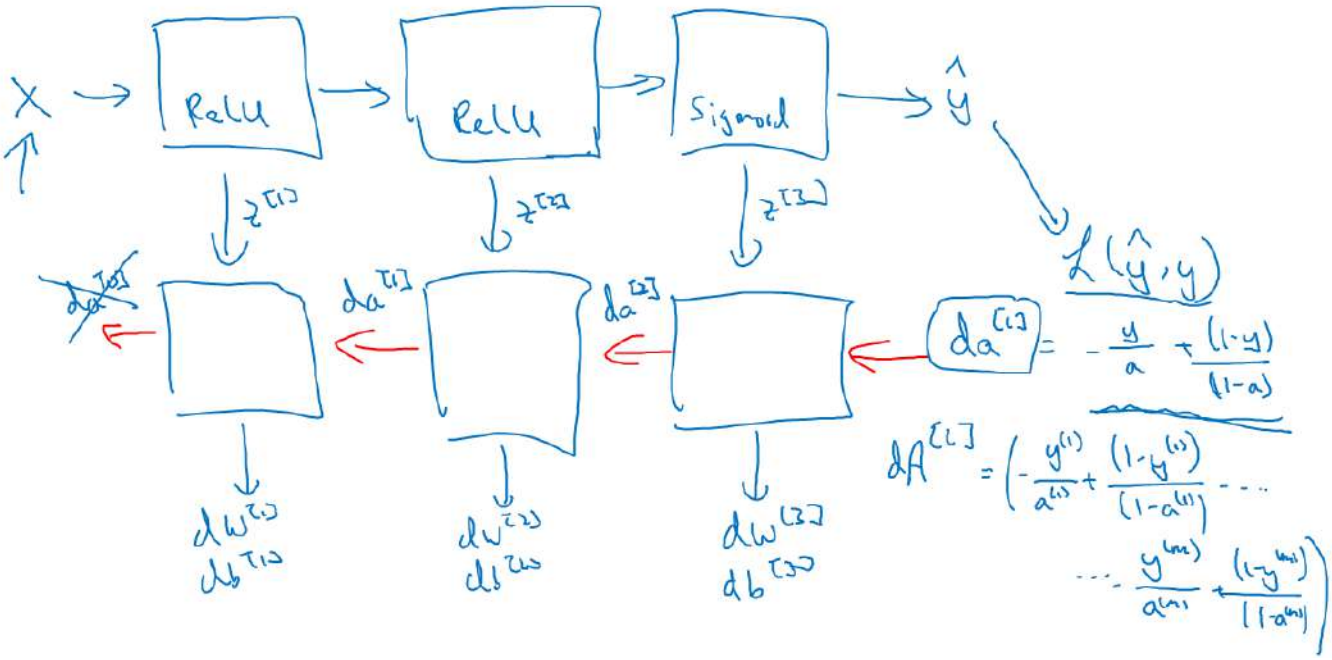
Now, Let's talk about the backward propagation step. See the diagram below.



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 **********