When you organize the computation of your neural network, usually we have forward propagation, followed by a back propagation step. Learn why in learning NN for forward propagation and backward propagation.

Logistic regression is an algorithm for binary classification. You have the problem as, the input is an image of cat, and output is 1(cat) or 0(not cat). We use y to denote the output label.

To store an image your computer stores three separate matrices , corresponding to red, blue, green colour channels of this image. So if the image size is 64X64, we would have 3 64X64 matrices.

To turn these pixel intensity values into a feature vector, what we are going to do is unroll all of these feature values into an input feature vector X, which is essentially just a one dimensional array containing the pixel values.

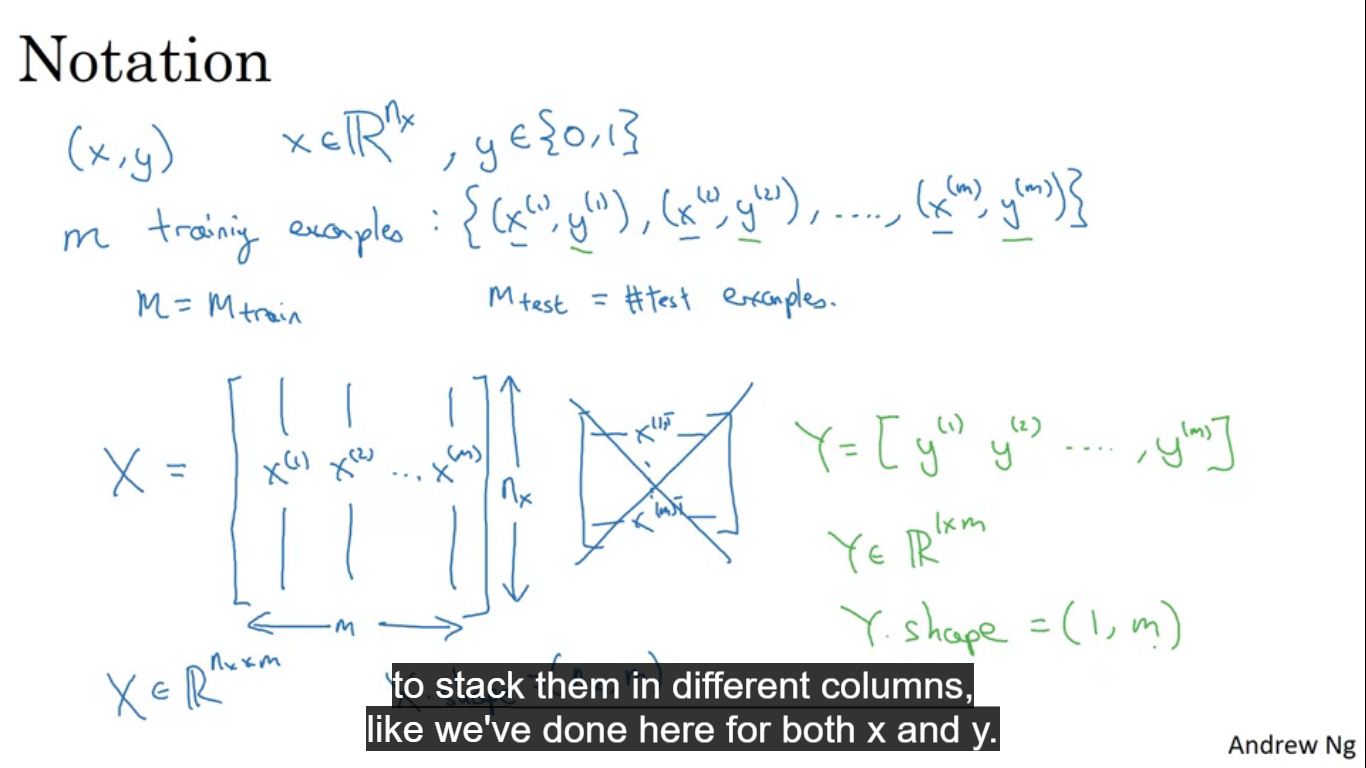
So total size of this feature vector X would be 64X64X3 = 12288 as that’s the total number we have in all of these matrices.

In binary classification, our goal is to learn the class of an image represented by feature vector X and predict whether the corresponding label is 0 or 1.

Notation

Single training example = (x,y) where x = n dimensional feature vector and y= label of the class.

Training sets comprise of lowercase m training examples.



**Logistic Regression**

Given an input feature X we want to find Y hat to be the probability of the chance that Y is equal to one given the input features X. So if X is an image we want to find out what is the chance that this is a cat picture.

Parameters of a logistic regression are, **W** which is an n dimensional vector like X, and **b** which is just a real number.

For linear regression:

Y = W(T)X + b

But this isn’t a good algorithm for binary classification

We want Y to be the chance/probability so Y should be between 0 and 1. Here it can be >1 or <0 which doesn’t make sense.

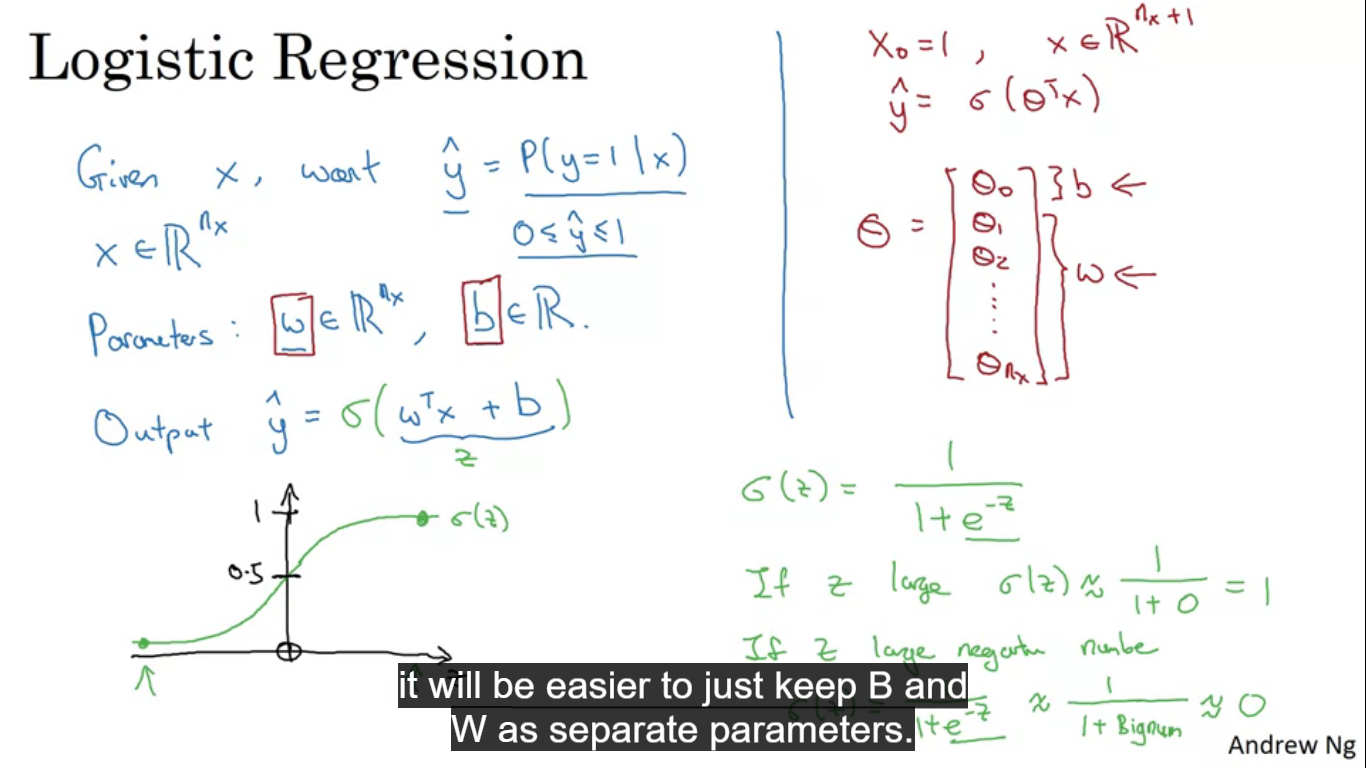
So our output is **Y\_pred = sigmoid(W(T)X + b)**

**Sigmoid(Z) = (1/(1+e^-Z))**

If Z is very large, then e^-Z will be close to zero, then sigmoid(Z) = 1

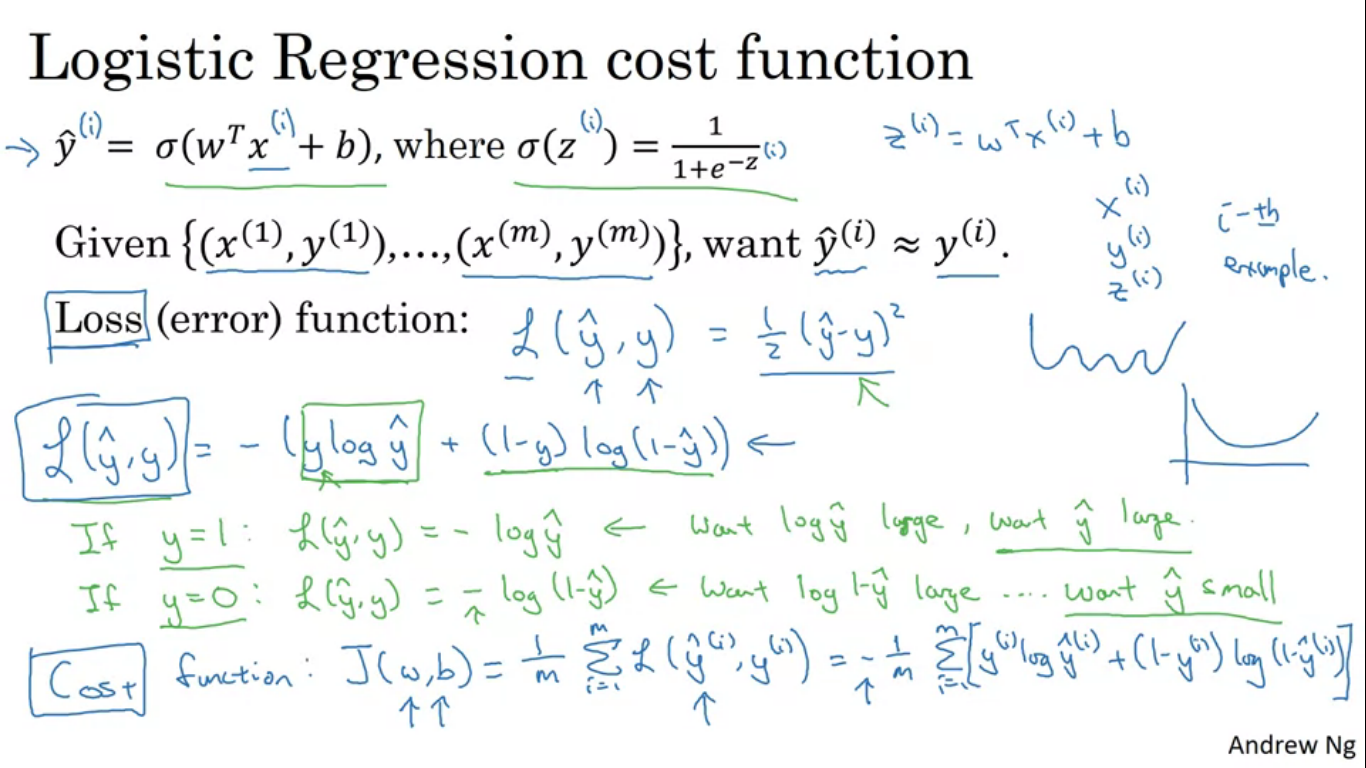
Conversely if Z is very small , or a very large negative number , then e^-z will be very big, then sigmoid(Z) = 0

When we implement logistic regression, our job is to learn parameters W and b, so that Y becomes a good estimate of chance of Y being equal to 1.



To train the parameters W and b of the logistic regression model, we need to define a cost function.

Superscript parenthesis **i** represent data associated with the **ith** training example.



Loss or error function could be = **0.5(Y\_pred – Y\_true)** , we could use this but in logistic regression people don’t usually do this, because when it comes to learning parameters , the optimization problem becomes non convex, making gradient descent not able to find out global optimum.

Loss function actually used = **-(Y\_true log(Y\_pred) + (1-Y\_true)log(1-Y\_pred))**

If we were using squared error then we want squared error as small as possible and with this logistic regression loss function we also want it to be as small as possible.

* If Y\_true = 1, then the loss function is = **- log(Y\_pred)** to be as small as possible

So log(Y\_pred) to be large , so Y\_pred to be large, but Y\_pred is sigmoid function can never be >1 , so Y\_pred ~~1 as well.

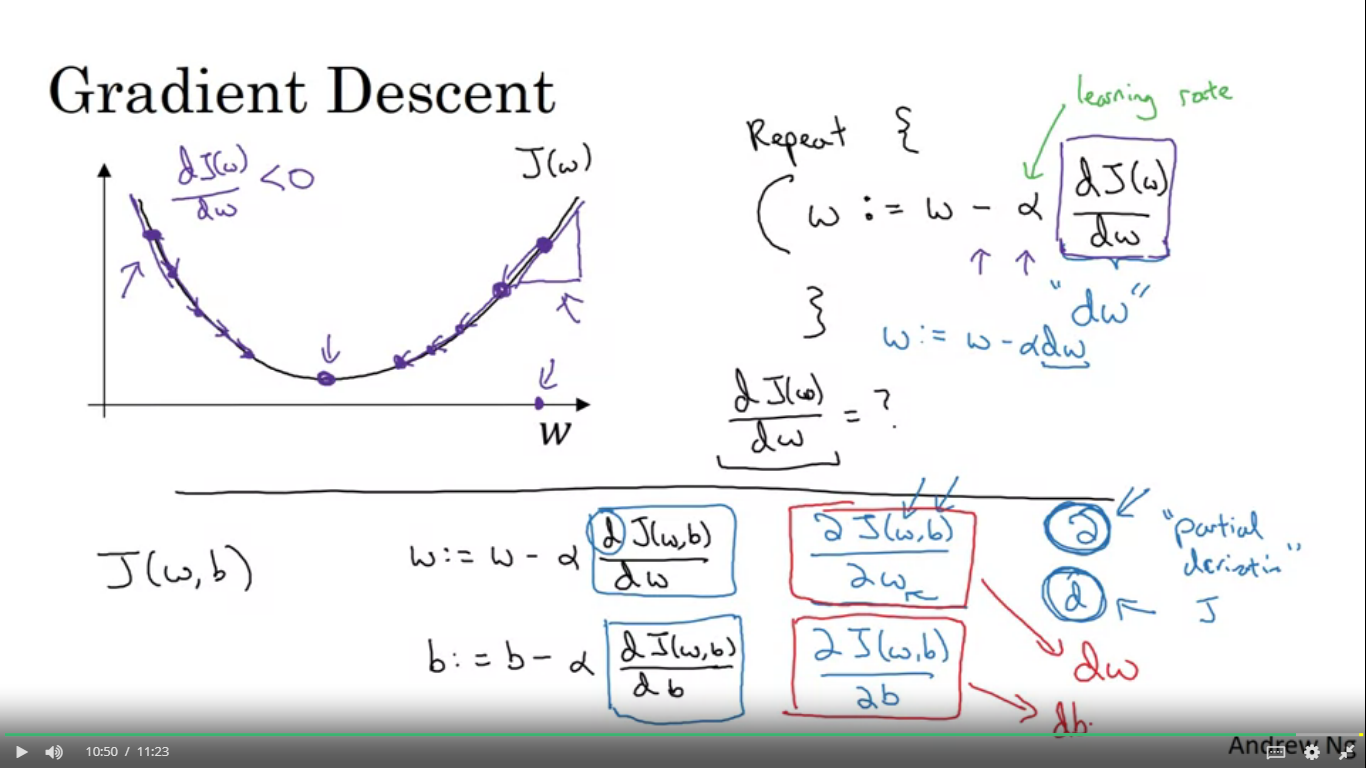
* If Y\_true = 0, then the loss function is = -log(1-Y\_pred) to be as small as possible

So log(1-Y\_pred) be large, so 1-Y\_pred be large so Y\_pred needs to be as small as possible. But can never be < 0, so Y\_pred~~0 as well.

A lot of functions exist like the one. Also previously we have defined it on a single training example, we want to define it on complete training dataset , so cost function is

**J(W,b) = (1/m)sum of loss function applied to each of training example(mentioned above)**

**Gradient Descent Algorithm**

****

Gradient descent algorithm to train and find W and b. Cost function measures how well your parameters W and b are doing on that training set.

To find W and b, we have to find the values of W and b that makes the cost function as small as possible. The cost function J is a convex function (single local minima)

To find the values, what we do is to initialise W and b to some initial value (usually 0)

Gradient descent starts at that initial point and then takes a step in the steepest downhill direction till you converge to the global optimum.

Alpha is the learning rate that controls how big a step we take on each iteration , te derivative is the change or the update you want to make to the parameter W

**W := W – alpha\*(dJ(W)/dW)**

**dJ(W,b)/dW** is represented by **dW**

**dJ(W,b)/db** is represented by **db**

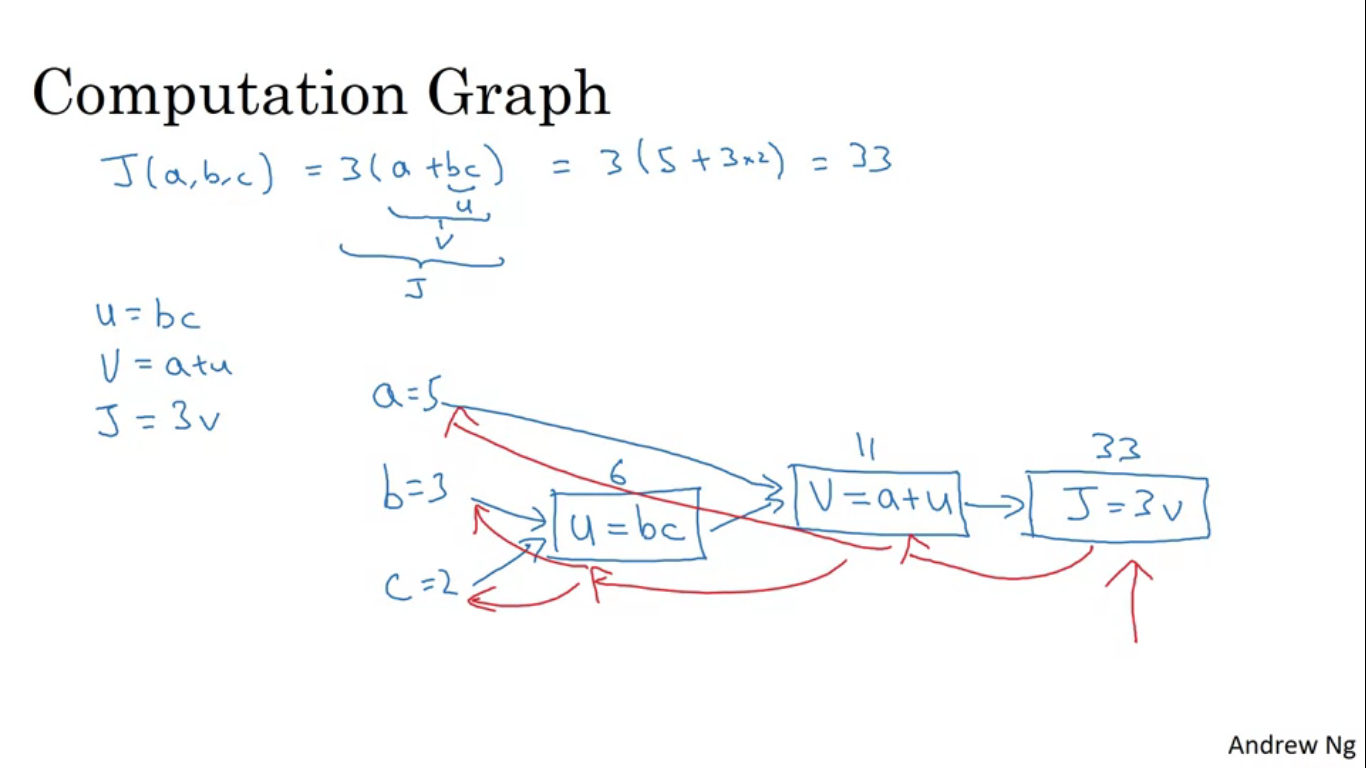
Definition of a derivative is the slope of a function at a point, slope is height / width of the triangle generated by the tangent at the point. Also check for the sign of the slope dW.

Here we wrote our function if only W was our parameter, but in real we have a cost function containing W and b

**Intuitive Understanding of derivatives**

Already have this from classes 11 and 12

**Computation graph** tells us why we have done forward and backward propagation.



Lets say we have a function J(a,b,c) = 3(a+bc)

U = bc

V = a+U

J = 3V

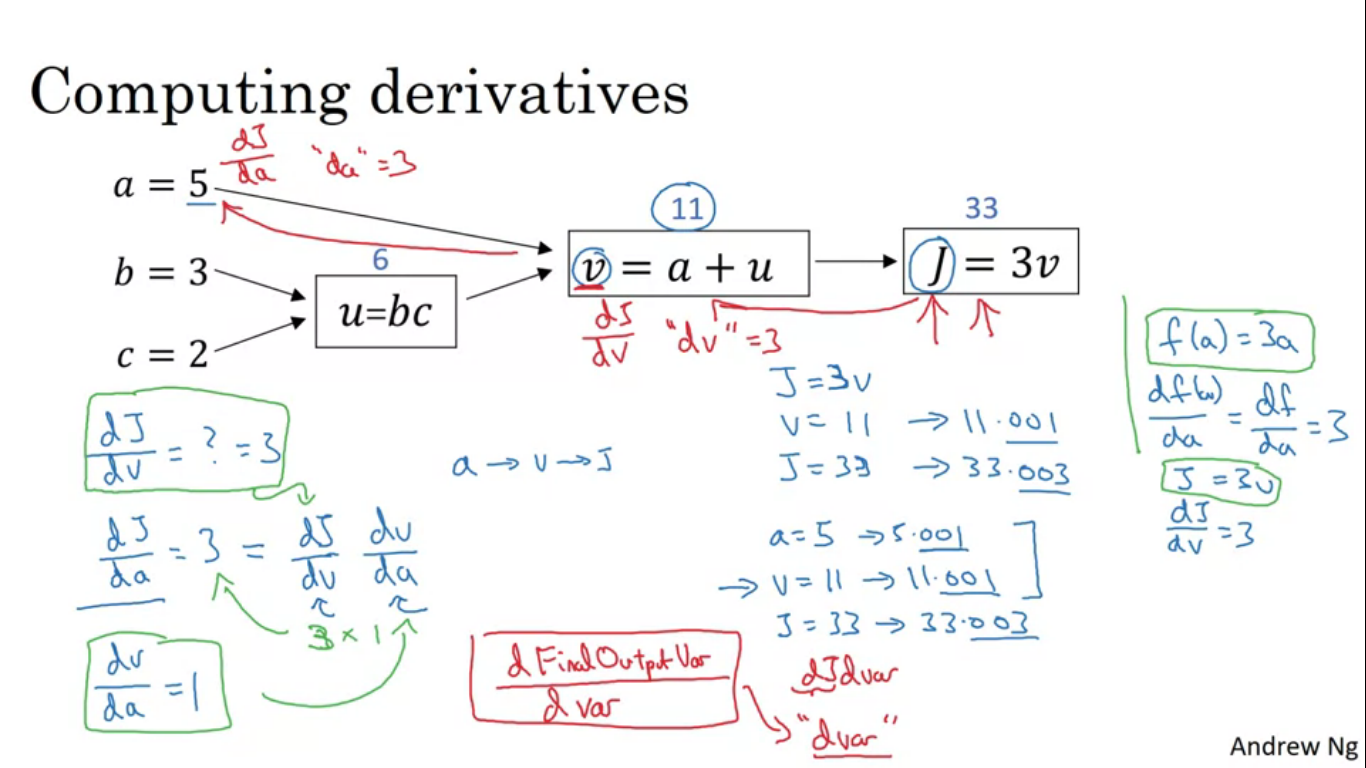
Computation graph comes in handy when there is some distinguished output variable that you want to optimize.

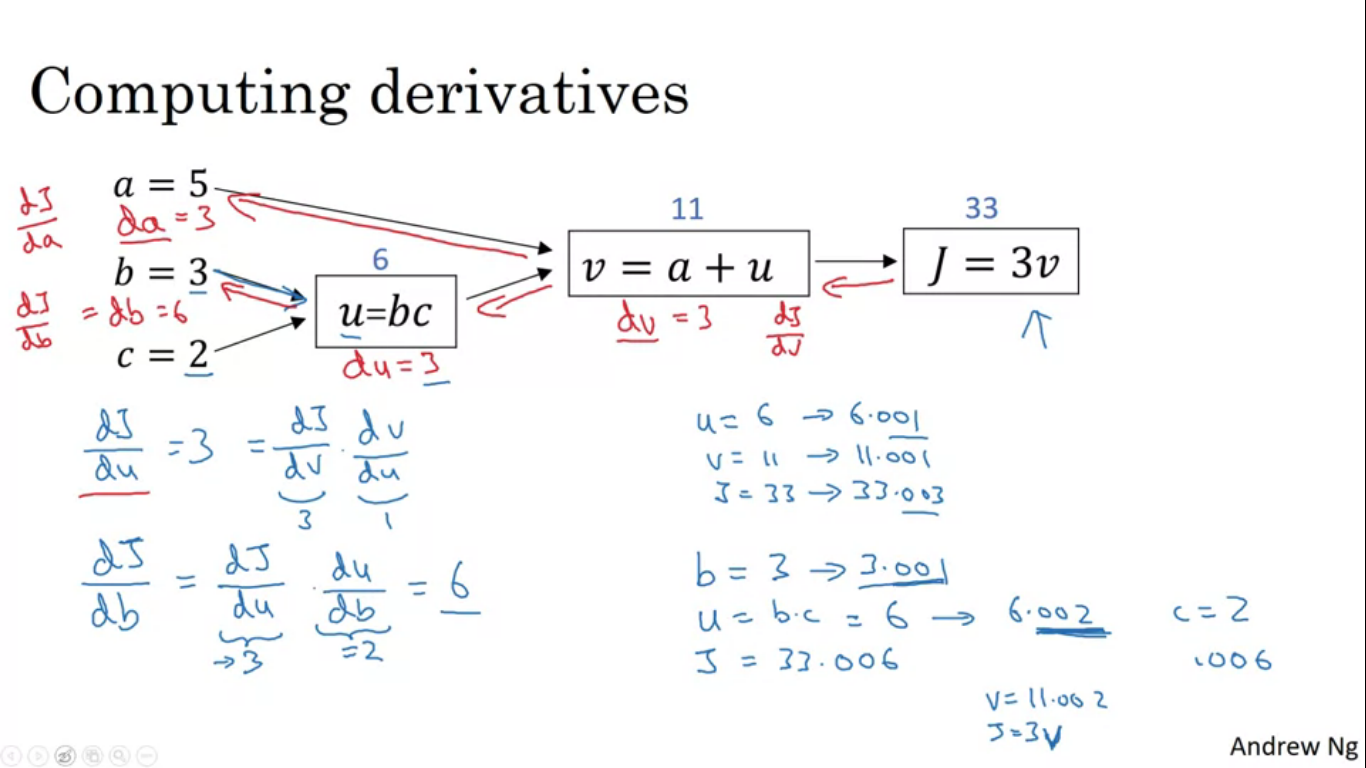
If we want to find the derivative of this final output variable , we find out dJ/dV, and we have done one step of back propagation.

To find dJ/da = 3 if you change a then you change V and in turn which changes J

Final output variable which we really care about is J, a lot of computation will be for finding derivative of final output variable with various intermediate variables.

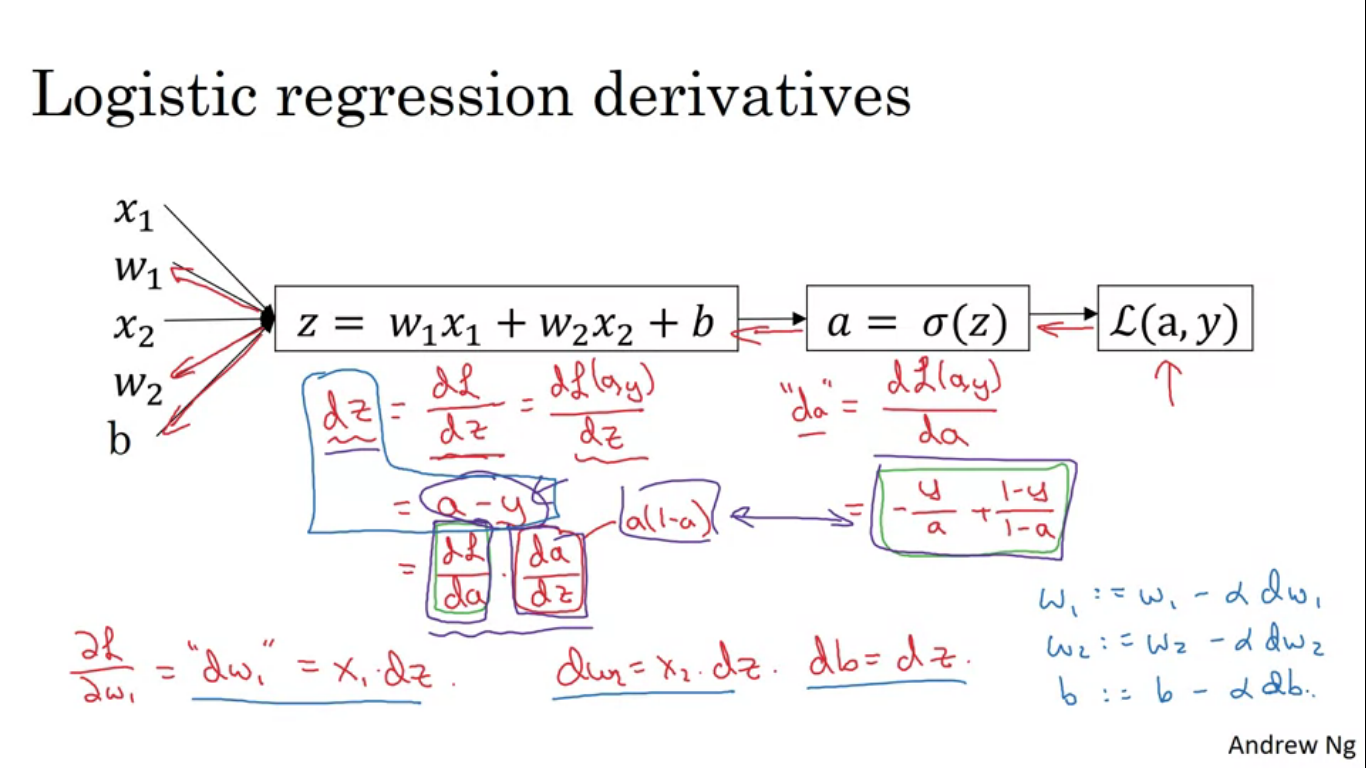
dvar represents the derivative of the final output variable we care about with respect to various intermediate quantities.





**To compute derivatives to implement logistic regression**

For loop se algorithm runs less efficiently , to counter this problem, there are techniques of vectorisation which we would use.

****

**Vectorization** is an art of getting rid of explicit for loops in our code. We deal with very large datasets and we don’t want our code to run for a very long time to get the result.

In logistic regression we needed to compute

Z = W(T)X + b , W and X are very large Nx dimensional vectors.

Non Vectorized implementation

Z = 0

For i in range(n-x):

Z+=W[i]\*X[i[

Z+=b

This is a very slow implementation, vectorized implementation is much faster.

In python , numpy

We calculate

Z = np.dot(W,X) +b this computes W(T)X+b

Both CPU and GPU support parallelization and support SIMD instructions = Single input multi data

Rule of thumb - whenever possible avoid using explicit for loops.

Matrix multiplication – U – AV

u(i) = sum(Aij Vi)

for loop me bahut karna padega and it will be slow too.

Non Vectorized Version

U=np.zeroes((n,1))

For i...

For j....

U[i] = A[i][j] \* V[i]

Vectorized Version

U = np.dot(A,V)

Another Example

A vector V jispe exponential operator on every element

Non vectorized

U = np.zeros((n,1))

For i in range(n):

U[i] = math.exp(v[i])

Vectorized approach

Numpy has a function by which we can compute this by a single call to a single function

U = np.exp(v)

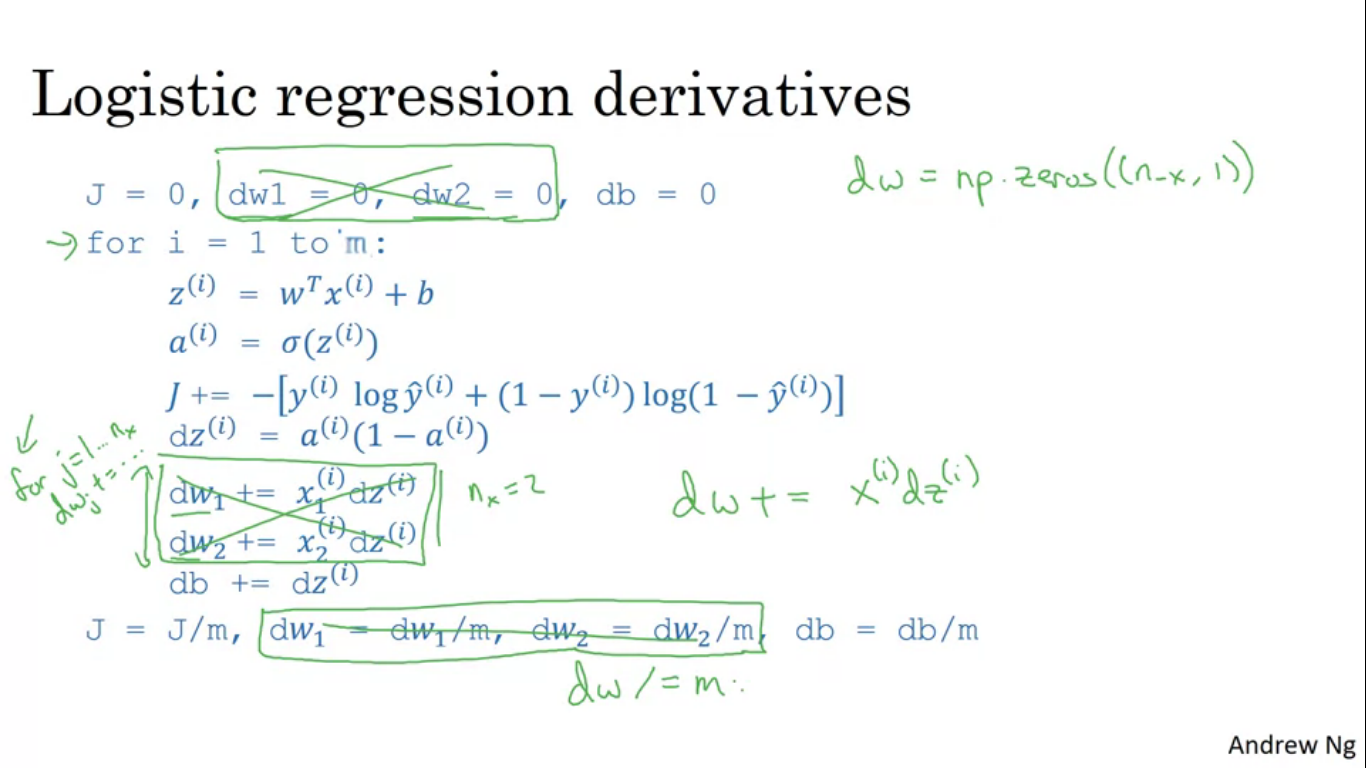
Numpy functions

* Np.log (v)– computes element wise log
* Np.abs(v) – computes absolute value
* Np.maximum(v,0) – computes element wise max to find max with 0
* V\*\*2 – gives element wise square of each element

Now in the logistic regression we had two for loops one in starting and another in adding of dw1,dw2 etc.

To get rid of the second for loop, dw1,dw2 etc gets replaced by dw vector . dw = np.zeros((nx,1))

Next done on images Removal of Inside For Loop



Next we remove the outside For Loop also

Logistic regression forward propagation

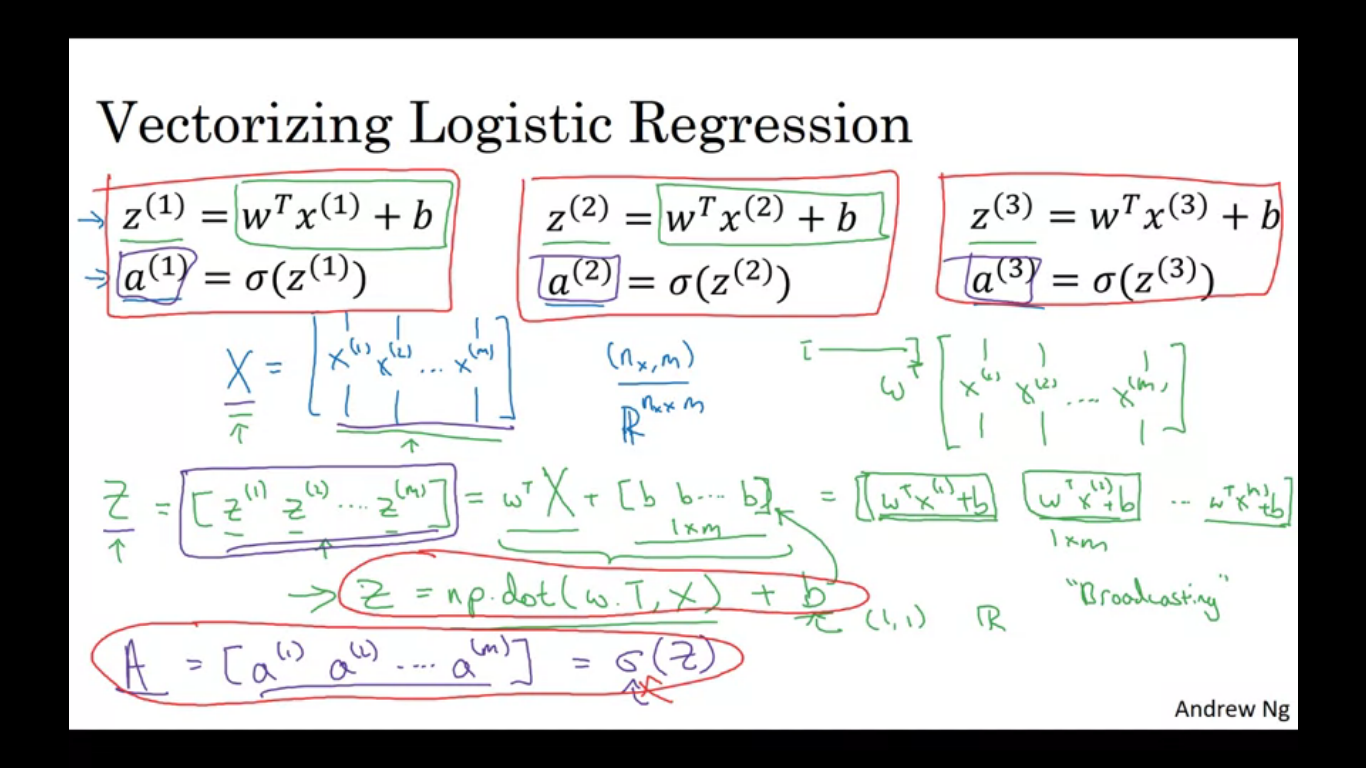
We defined a matrix X to be our training inputs stacked together in columns with dimensions (nx,m),

To perform the steps, numpy command is

**Z = np.dot(W.T,X) + b**

Next find a way to compute **a**

**A = sigmoid(Z) to find sigmoid done in programming assignment**

****

**Logistic regression backward propagation (vectorizing logistic regression gradient computation)**

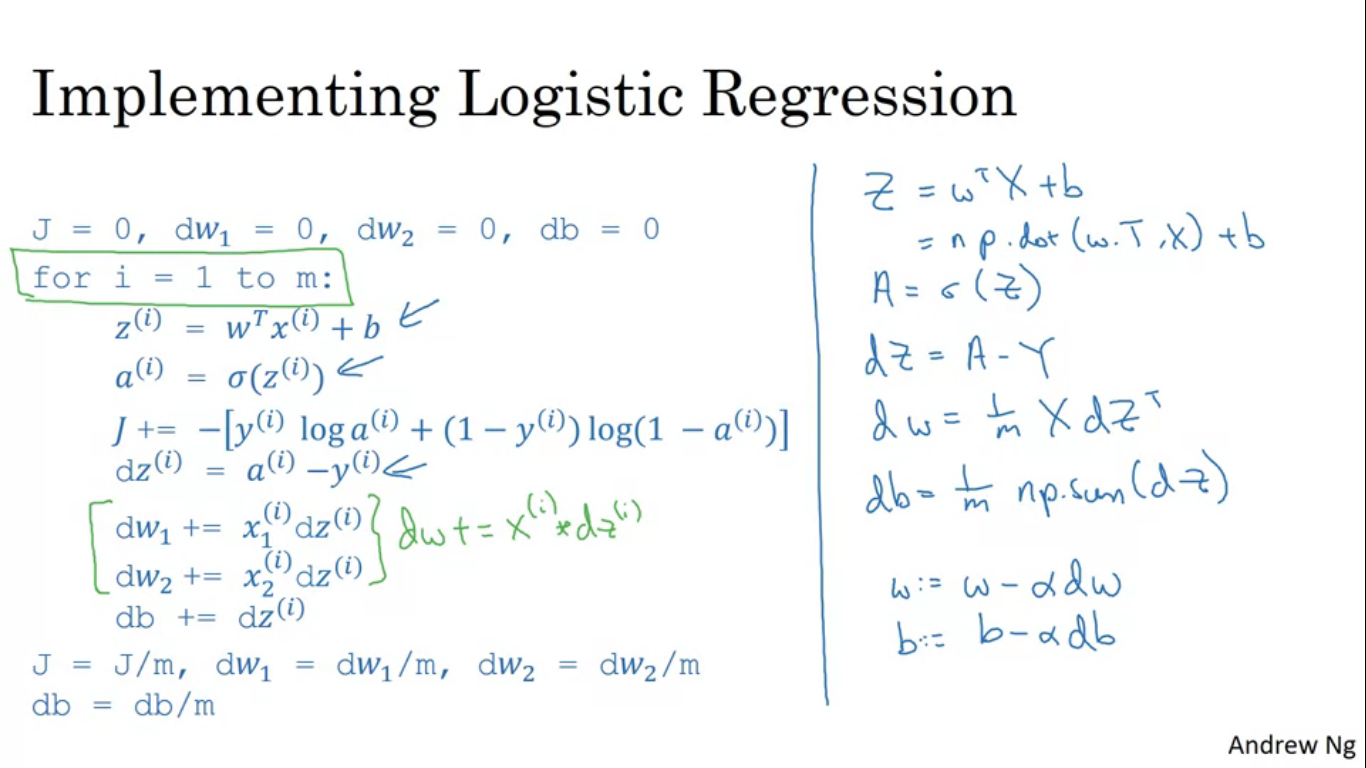
**DZ = A-Y**

**DB = (1/m)np.sum(DZ)**

**DW = (1/m) np.dot(X,DZ.T)**

****

**Complete Logistic regression Vectorized(IMP)**



**Broadcasting in Python**

You have a matrix representing four foods and have a value of carbs, proteins and fats. You need to find the percentage of each of the carbs, proteins and fats for all the foods.

So what we want is to sum up each of the four columns and find the ratio of the values to the sum.

ab+ac-b-c

a(b+c)-1(b+c)

(a-1)(b+c)