Deep Learning

what is Deep Learning ?

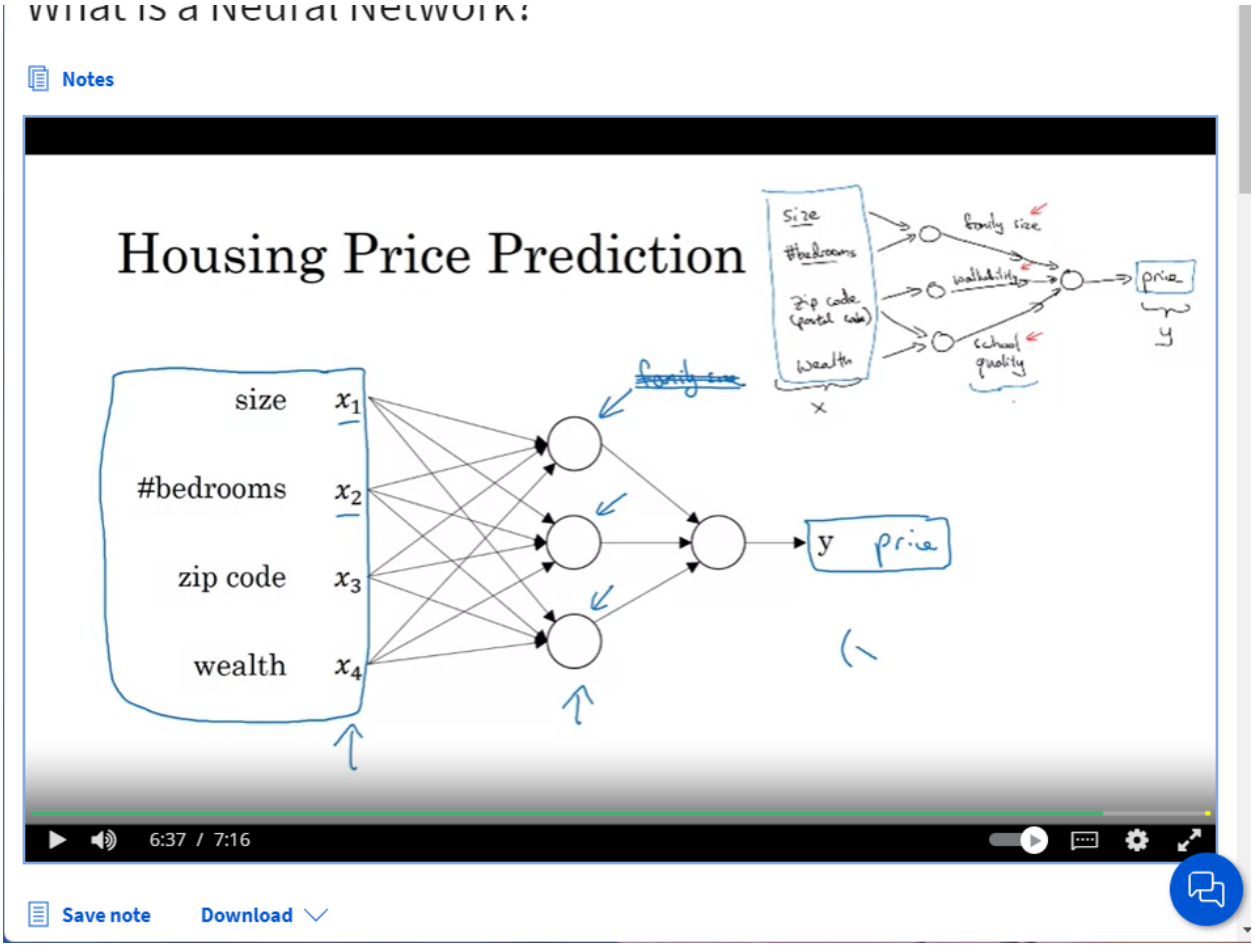
It is a process of training a neural network.

so what is Neural network ?

credit : Andrew NG,

Lets say we are computing the price of house based on size of house, then the non linear function or (linear function) can be considered as a neuron.

The collection of this neurons is called a neural network, Each and every neural network takes into all the features, but uses which ones it thinks as necessary.



All the neurons in the middle is coined as hidden layer.

Supervised Learning with neural networks

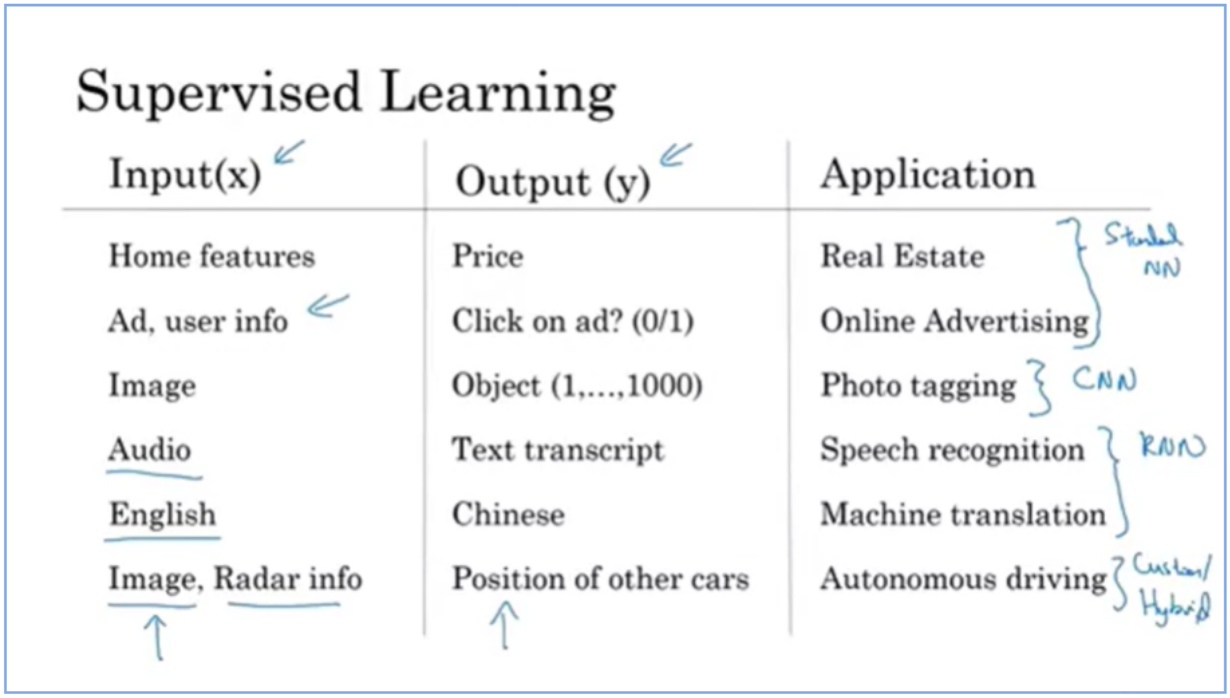
Deep Learning models works very well, but most of the hype around DL is bought out by Supervised learning.

They got well in interpreting structured as well as unstructured data, for example identifying cats from a group of dog pictures, which is unstructured data. rolling out specific ads to target audience (structured data)

for Real estate and Online advertising we use standard neural networks.

Supervised Learning with Neural Networks

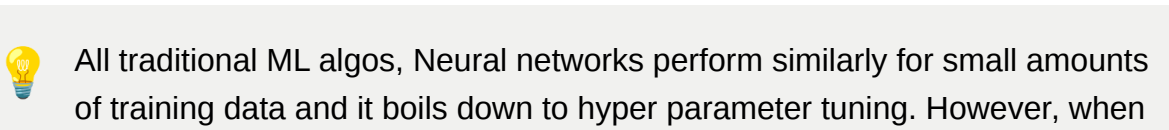
Notes



Save note Download

Why is Deep Learning Taking off ?

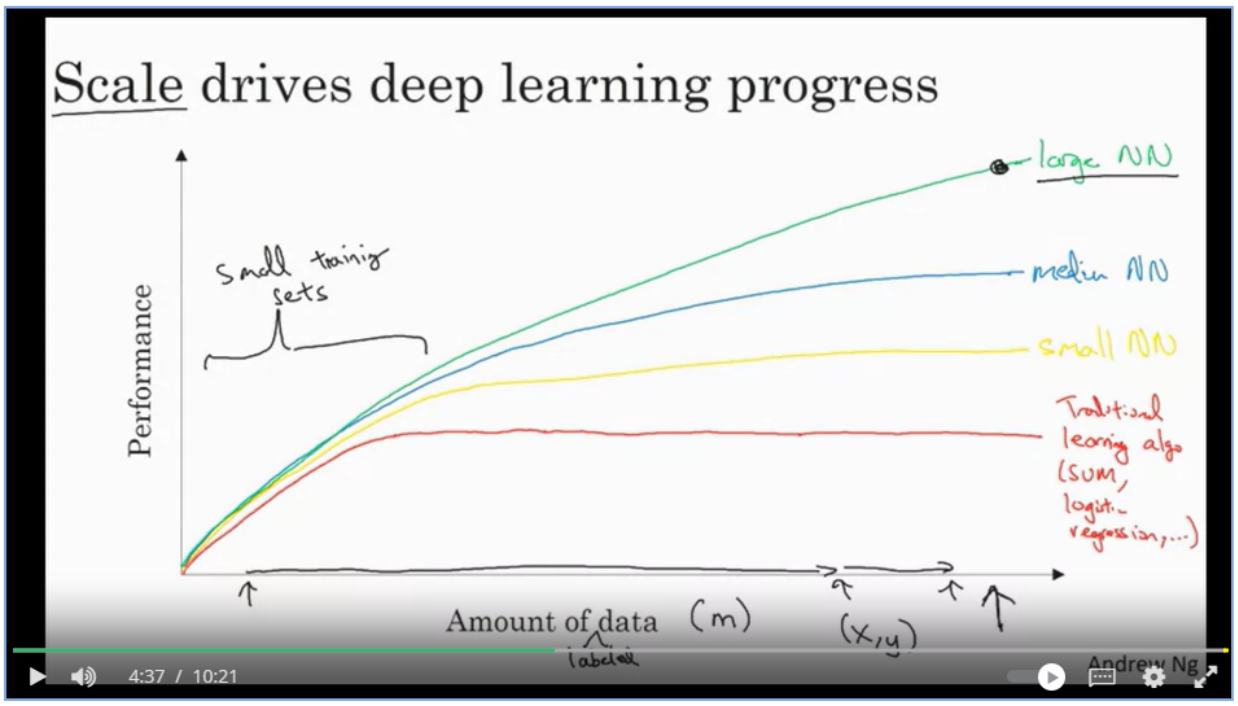
For all the traditional machine learning algorithms, the accuracy is almost a plateau, after some point in time, but the accuracy of neural networks explode, which makes the neural networks interesting.



All traditional ML algos, Neural networks perform similarly for small amounts of training data and it boils down to hyper parameter tuning. However, when we have large amounts of data, there is a significant increase in accuracy when neural networks are used.

for large data

traditional ml algos < small neural networks < medium NN < large NN



Scale is driving Deep Learning :

1. Algorithms :

one of the notable break through is using RELU function ?? instead of Sigmoid function ?? (It makes gradient Descent work much faster)

2. Computation :

It impacts how faster we can get the experimental result back

3. Data

Logistic Regression as a Neural Network :

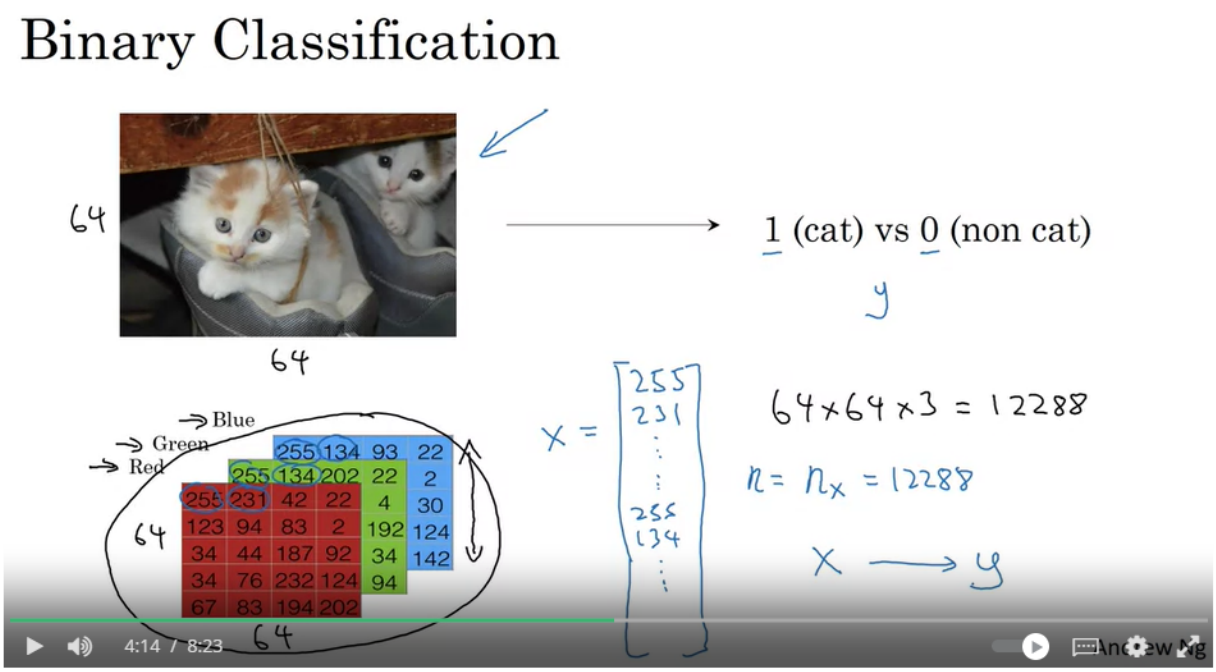
To build a classifier that says if the image is cat or dog, we need logistic regression.

The images are represented using 3 matrixes they are red, blue and green, (have dimensions each equal to the image). Every element has some value associated with it.

Now the X feature is aggregated as the each element of red, green and blue matrix placed one below another, this is the X feature, Nx is the number of rows.

Binary Classification

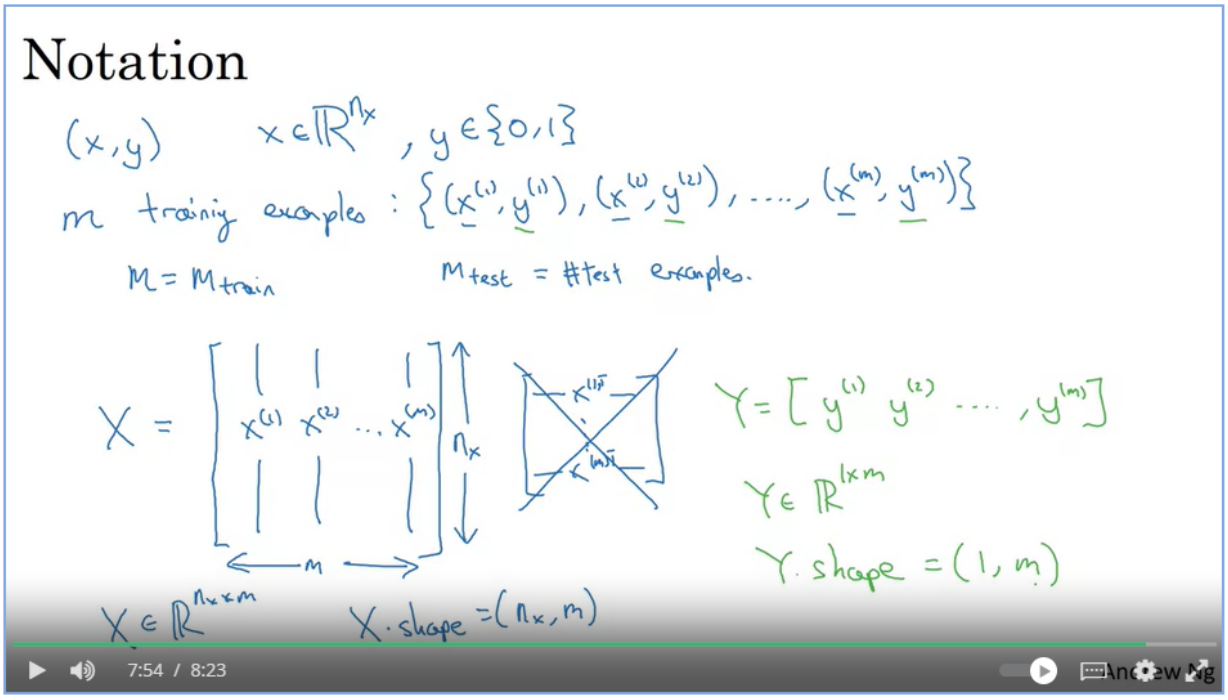
Notes



Save note Download

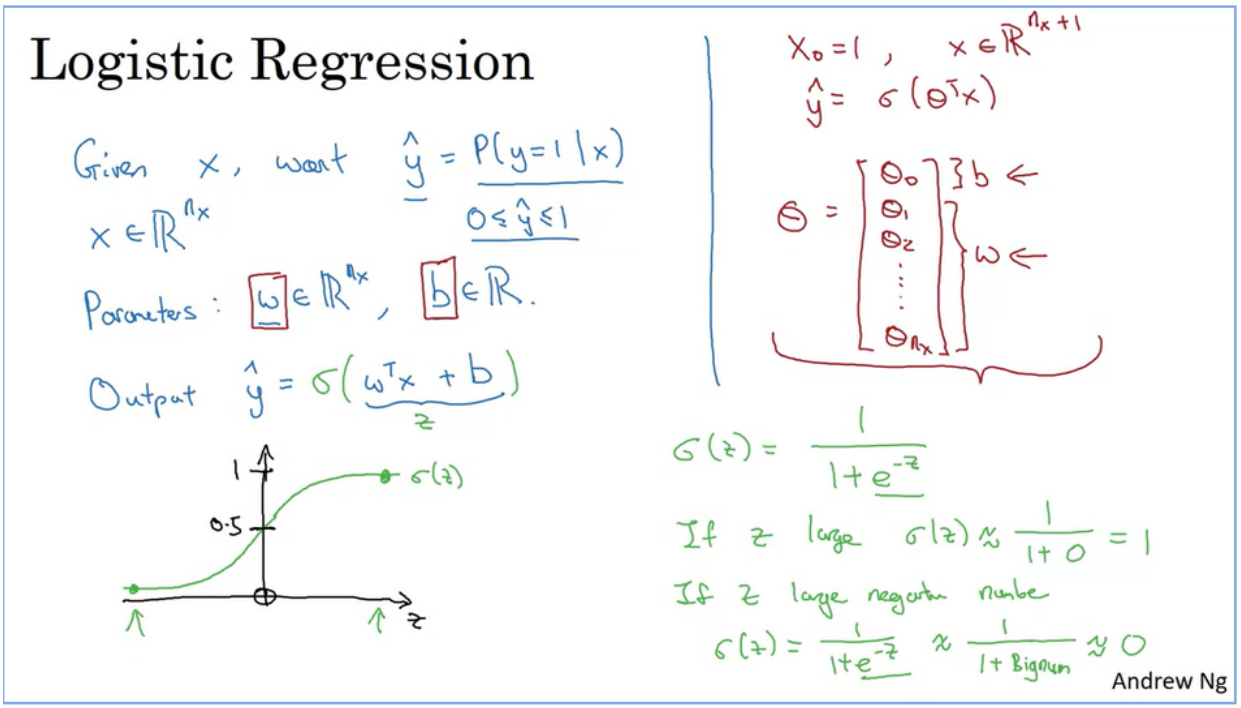
If there are m such training sets, then the training examples are stacked as below diagram named as X.

which also has Y (consisting of values 0 or 1).



Logistic Regression

To build a model, we need a function.

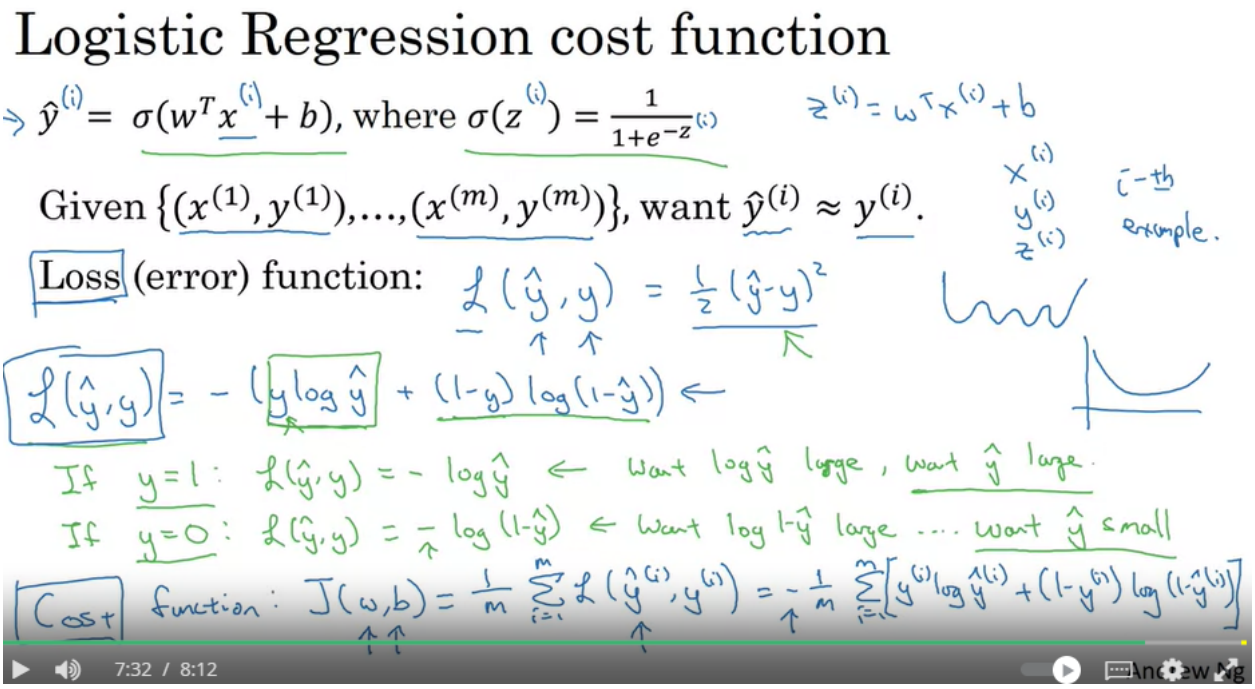


the function is outlined in the blue line beside output.

Since the output W^(T).X + B, would be be a continous variable, which we do not want, We require it to be between 0 and 1, for that purpose we apply sigmoid to the value, which converts the values between 0 and 1.

> We could use the notation (theta), but that notation is computationally expensive (in RED LINES), the only difference is that the b intercept is modelled into the matrix, with (THETA_0) representing B in our output.

Logistic Regression Cost Function :

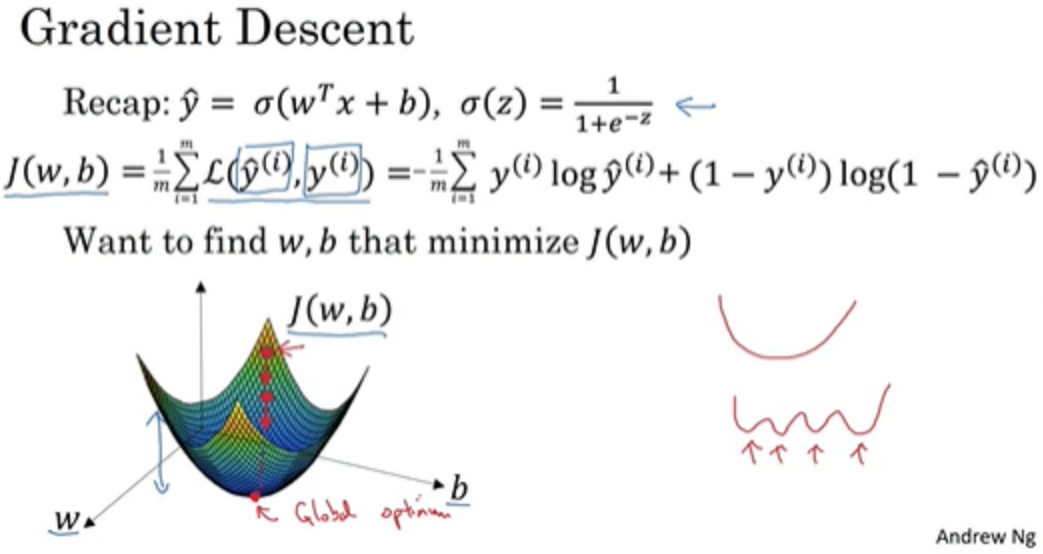


Use of traditional squared error, would give us a curve, for which finding gradient descent is not that easy. So we use another loss function (here applying gradient descent is much easier).

Gradient Descent :

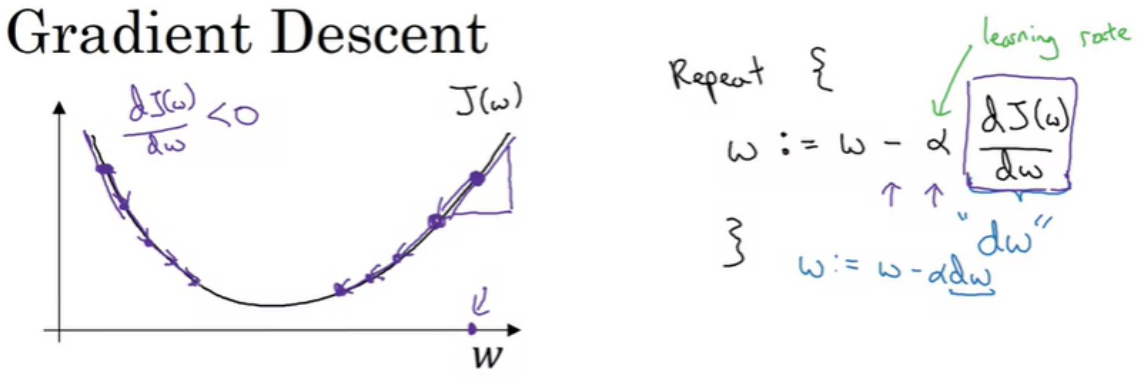
We have to minimize the cost function, the above cost function is observed to look like a convex curve, which has only one global minima, where ever we initialize the point, we would converge to a global minima.

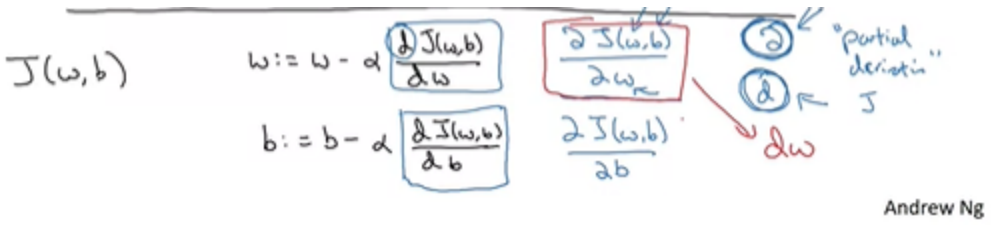
Intuition : we plot the cost function, in hopes of finding the variables x, y and (all other remaining dimensions), which gives us the global minima, as it can be considered as a best fit for our equation.



Note : There might be a lot of dimensions associated (w axis, b axis, so on), for this example, lets assume we only have one dimension.

We initialize at a particular point, and then converge down to a global minima, at each iteration we remove (learning rate * (derivative at that point))

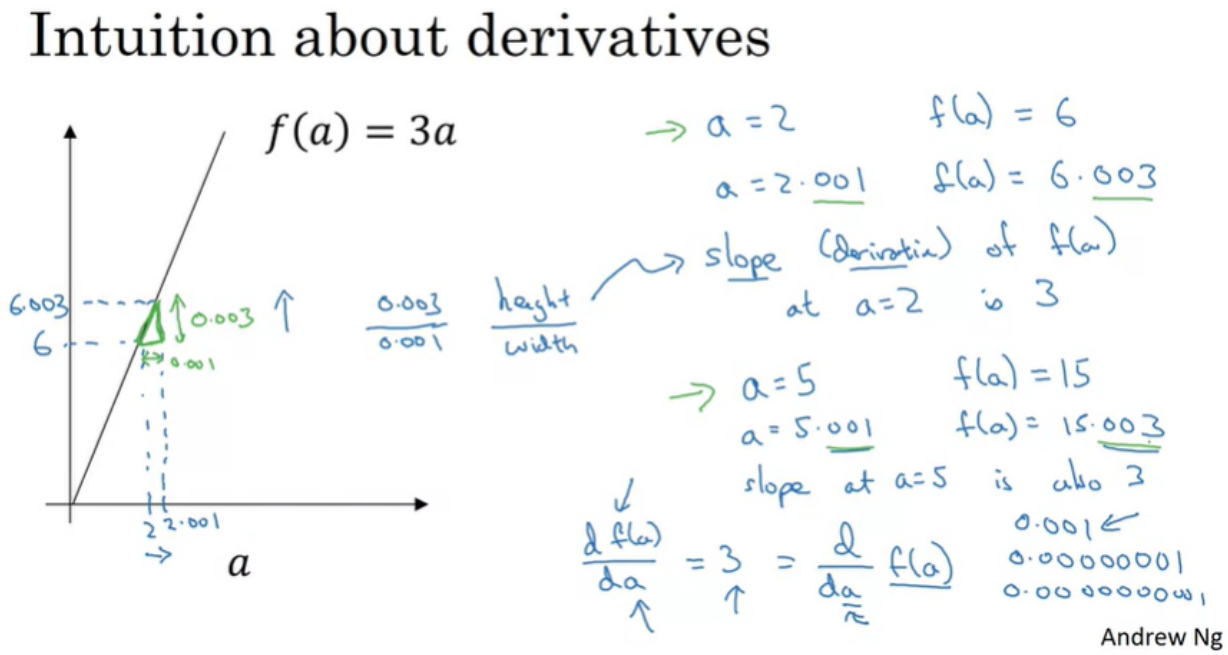




In the above figure, we are updating the value of “w”, by subtracting the learning rate * (the slope of the curve with respect to variable “w”) and vice versa for “b”.

Derivatives :

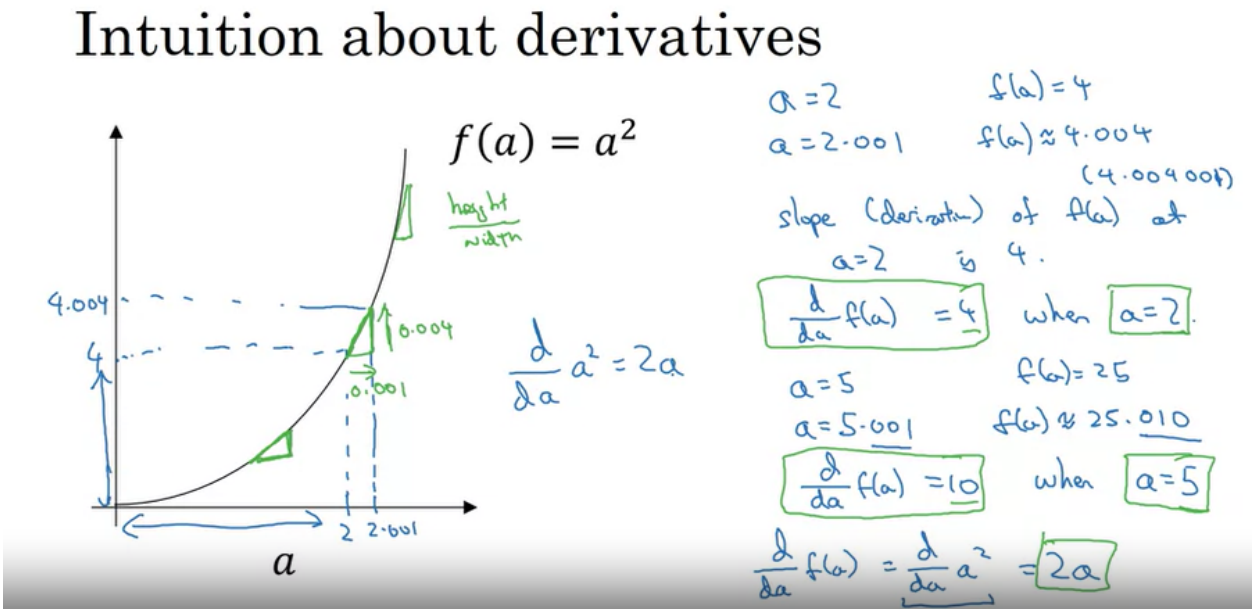
Derivative can be described as change in f(a), for a change in a.



Note : the slope of line is same all around the points, However it is different when it comes to non-linear graphs

Now, lets jump to some more examples.

Note : The derivative of a function at a specific point represents the rate of change of the function at that point. It is a measure of how much the function changes as you make an <u>infinitesimally</u> small change in the input variable.

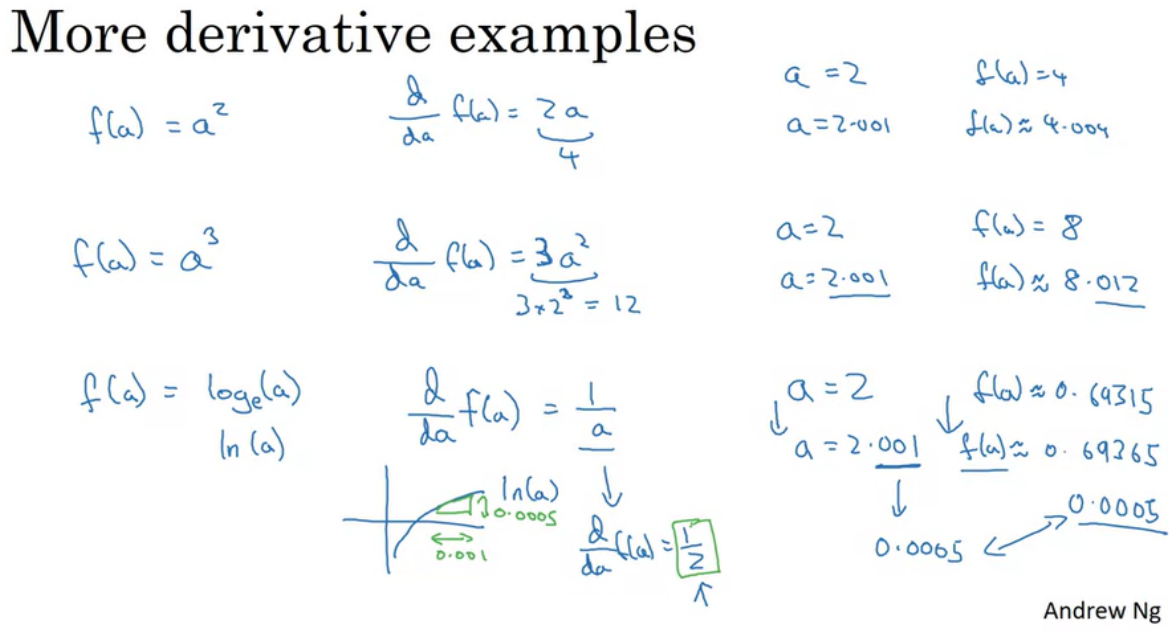


When you have a function (f(a) = a^2), you can calculate its derivative by looking at how the function changes as the input variable (a) changes. The derivative tells you the rate of change of the function.

1. When (a) is 2, (f(a)) is 4, and when (a) is 2.001 **(in reality it is much smaller than 0.001, (close to 0))**, (fa) is approximately 4.004. To calculate the derivative, you divide the change in f(a) (0.004) by the change in (a) (0.001). This ratio, 0.004/0.001, gives you approximately 4, which matches the derivative 2a (according to the formula d(a^2) = 2*a), where (a) is 2, and 2a is 4. This shows that the derivative accurately reflects the rate of change at (a = 2).

2. Now, when (a) is 2, (f(a)) is 4, and when (a) is 4, \(f(a)\) is 16. Again, you can calculate the derivative by dividing the change in (f(a)) (16 - 4 = 12) by the change in (a) (4 - 2 = 2). This ratio, 12/2, gives you 6, which **does not** match the derivative 2a, where \(a\) is 2, and 2a is 4. Because derivatives **define how much the function changes as you make an <u>infinitesimally</u> small change in the input variable, Here number increase of 2 units is really large, <u>so the derivative formula does not work here</u>.**

some more examples :

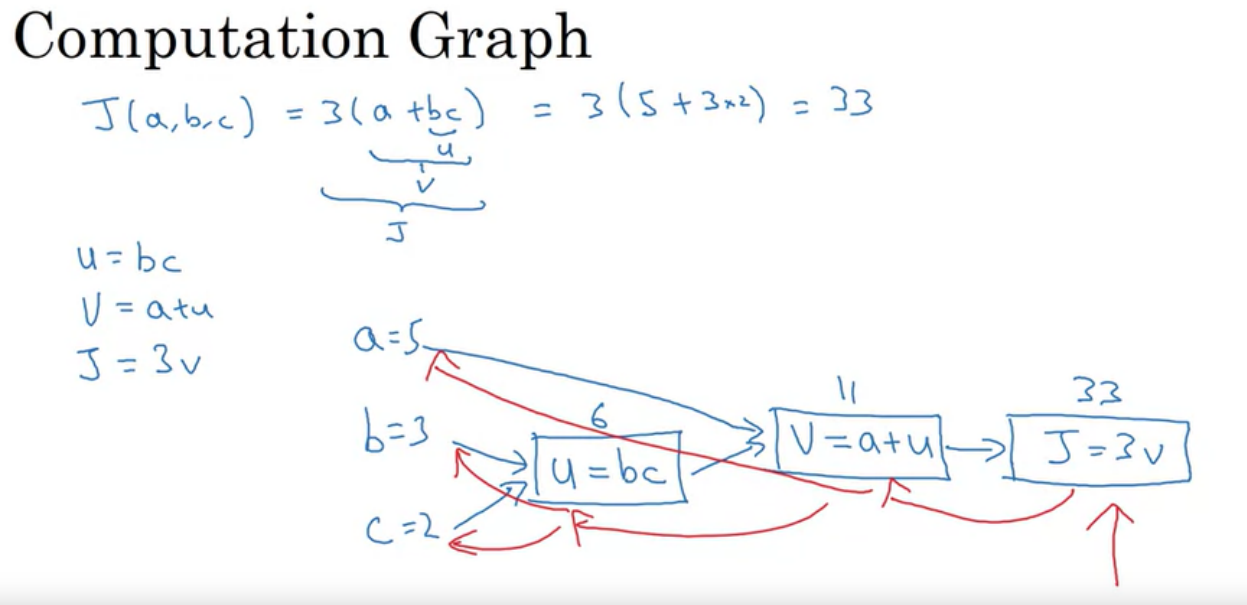


Computation graph :

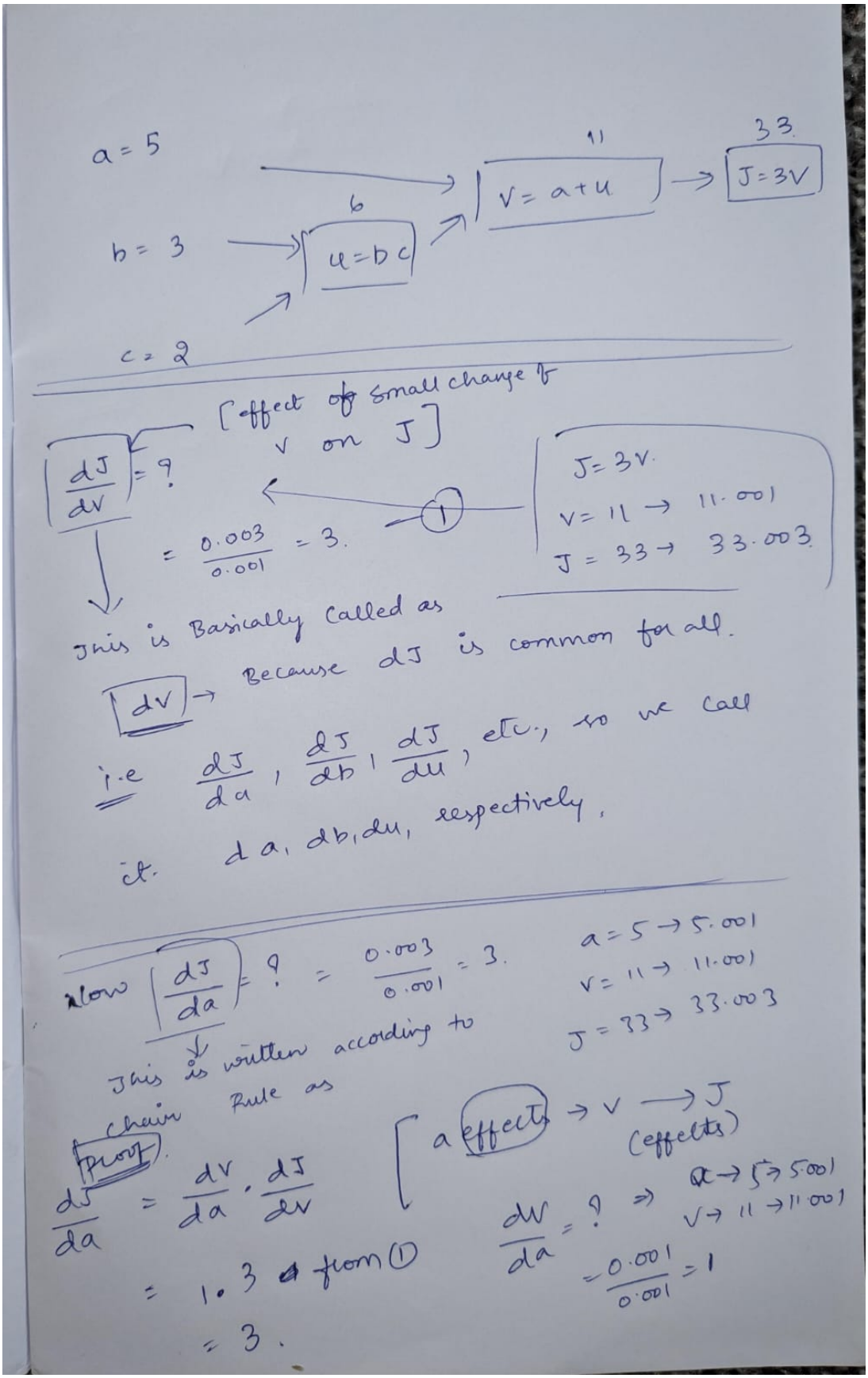
This diagram below states that

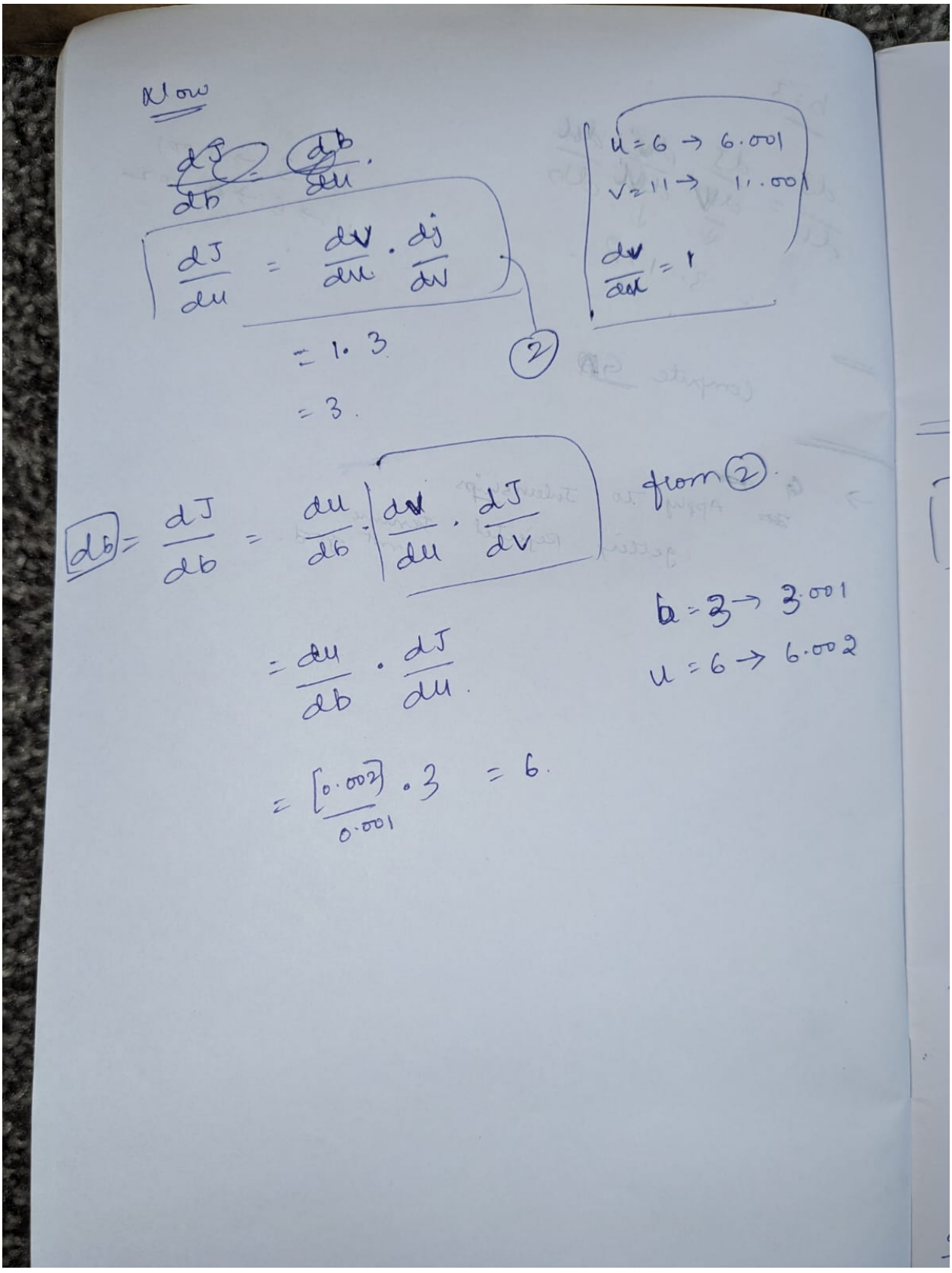
1. blue arrow depicts how the computation is done from left to right

2. Updation of weights (using derivatives is done from right to left)



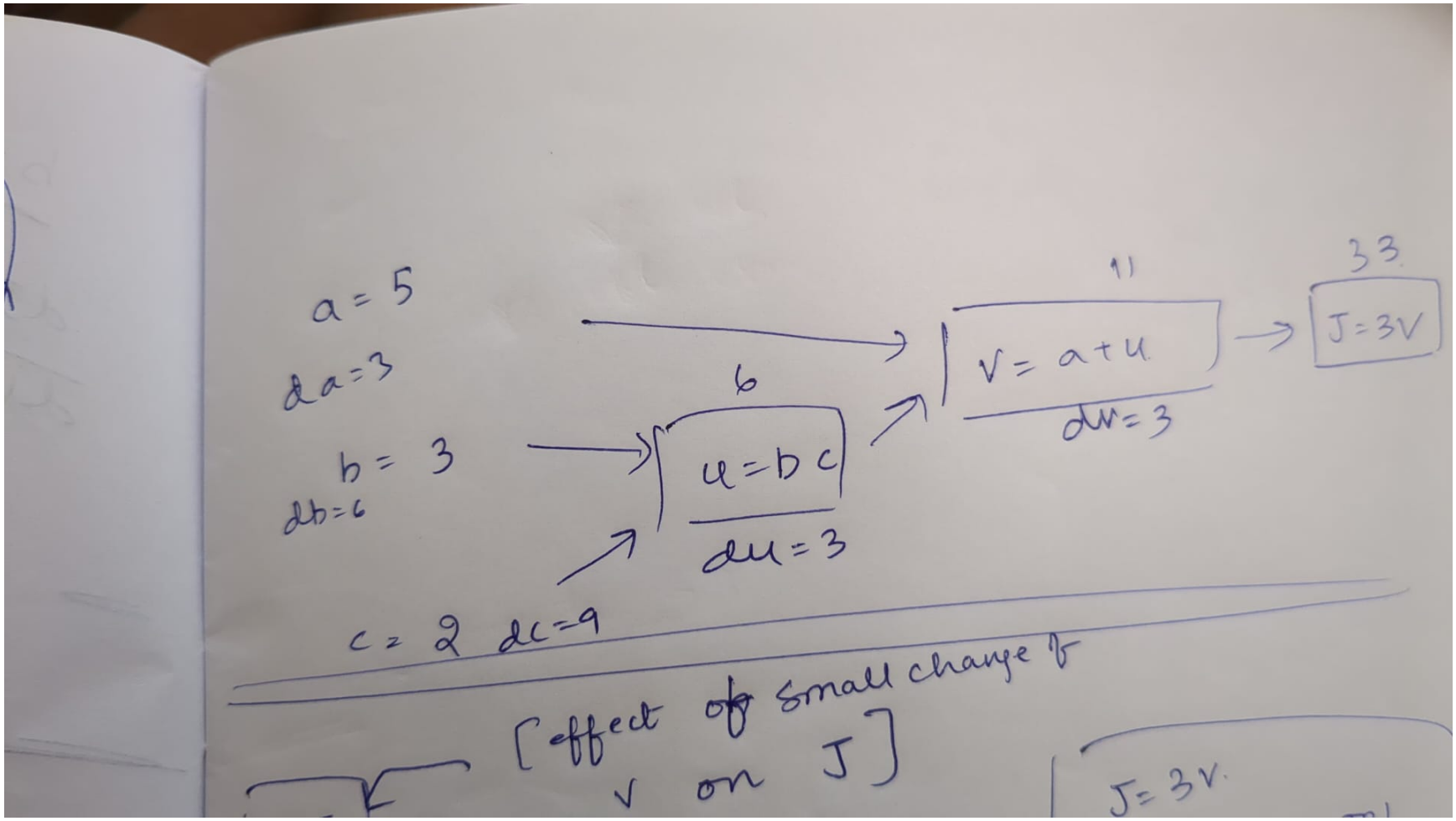
Derivatives with a Computation graph :


a = 5
b = 3
c = 2
6
11
33
u = bc
V = a + u
J = 3V
[effect of small change of V on J]
dJ/dV = ?
= 0.003/0.001 = 3.
J = 3V
V = 11 → 11.001
J = 33 → 33.003
This is Basically called as
dV → Because dJ is common for all.
i.e dJ/da, dJ/db, dJ/du, etc., so we call
it. da, db, du, respectively.
Now dJ/da = ? = 0.003/0.001 = 3.
This is written according to chain Rule as
Proof.
dJ/da = dV/da . dJ/dV
= 1.3 from (1)
= 3.
a effects → V → J
(effects)
dJ/da = ? ⇒
a → 5 → 5.001
V → 11 → 11.001
= 0.001/0.001 = 1

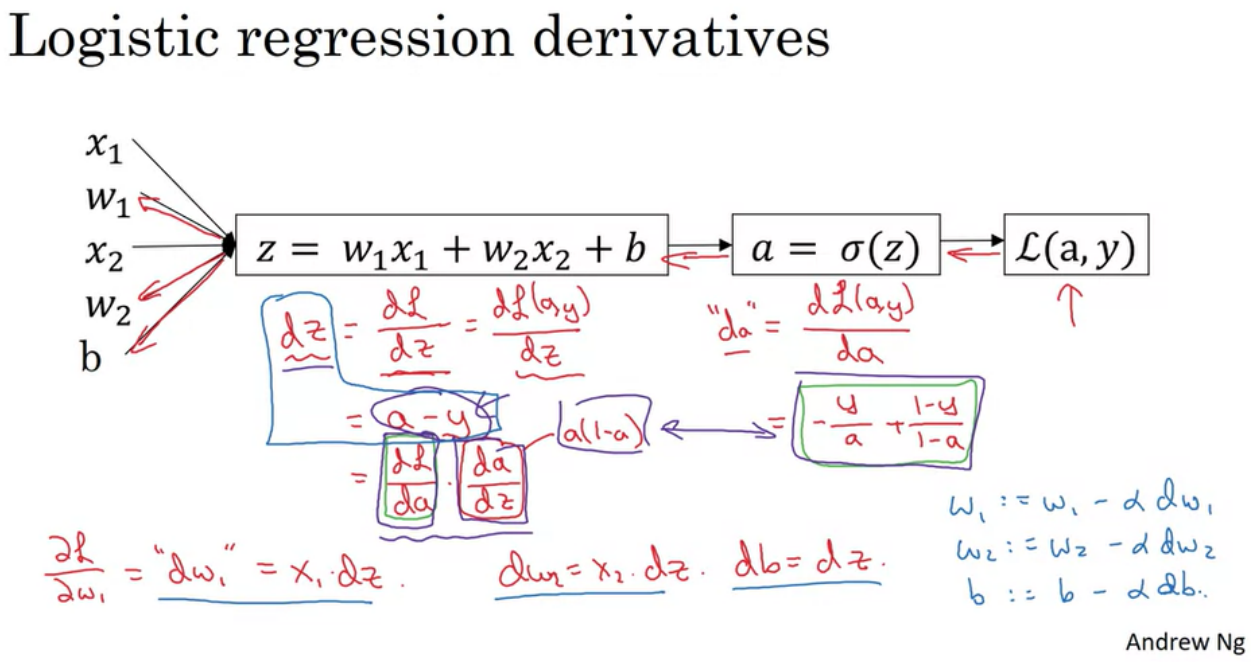

Now
$\frac{dJ}{db} = \frac{dv}{du} \cdot \frac{du}{db}$
$\frac{dJ}{du} = \frac{dv}{du} \cdot \frac{dJ}{dv}$
$= 1 \cdot 3$
$= 3$
(2)
$u = 6 \rightarrow 6.001$
$v = 11 \rightarrow 11.001$
$\frac{dv}{du} = 1$
$db = \frac{dJ}{db} = \frac{du}{db} \cdot \frac{dv}{du} \cdot \frac{dJ}{dv}$
from (2).
$= \frac{du}{db} \cdot \frac{dJ}{du}$
$= \frac{[0.002]}{0.001} \cdot 3 = 6$
$b = 3 \rightarrow 3.001$
$u = 6 \rightarrow 6.002$

dj/dv means : the effect of dj on small change in dv

similarly dc = 9


a = 5
da = 3
b = 3
db = 6
c = 2 dc = 9
u = b c
du = 3
6
v = a + u
dv = 3
11
J = 3V
33
[effect of small change of v on J]
J = 3v.

Finding the derivatives for Logistic Regression

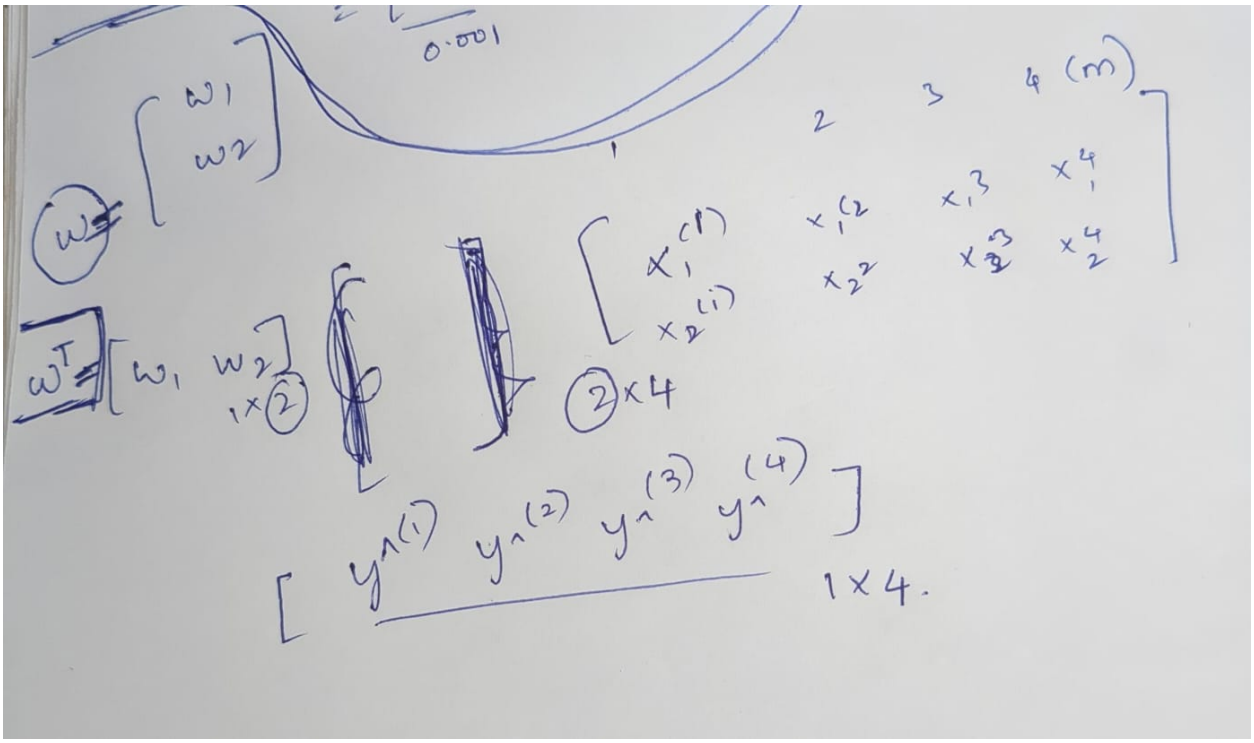


Note : The above example has only 2 features and only one training example.

L(a,y) is the loss function, a is the predicted values, Loss function is same it is discussed above. apply derivative formulas lead to the above values.

Gradient Descent on m Examples

Note : here we have m examples, 2 features each



This is the sort of example considered

Logistic regression on m examples

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

$$\rightarrow a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

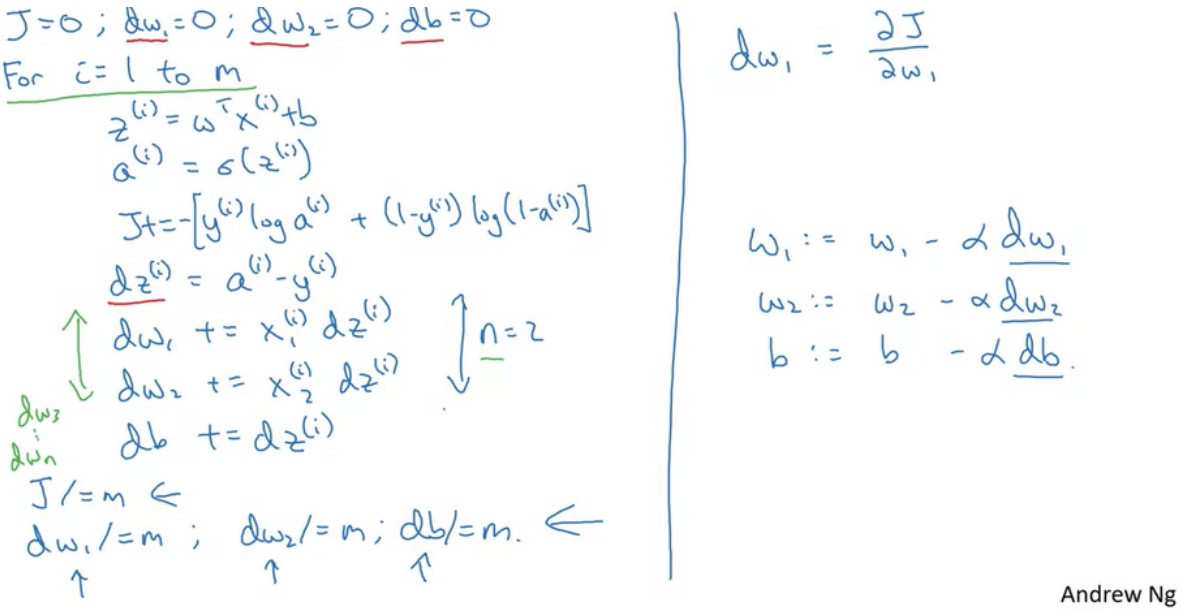
$(x^{(i)}, y^{(i)})$

$dw_1^{(i)}$, $dw_2^{(i)}$, $db^{(i)}$

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

dw1 value is computed as above. in the same way we calculate dw2 and db

Algorithm to find gradient descent on M Examples



note : we will compute dw1, dw2 and db for each example and take average.

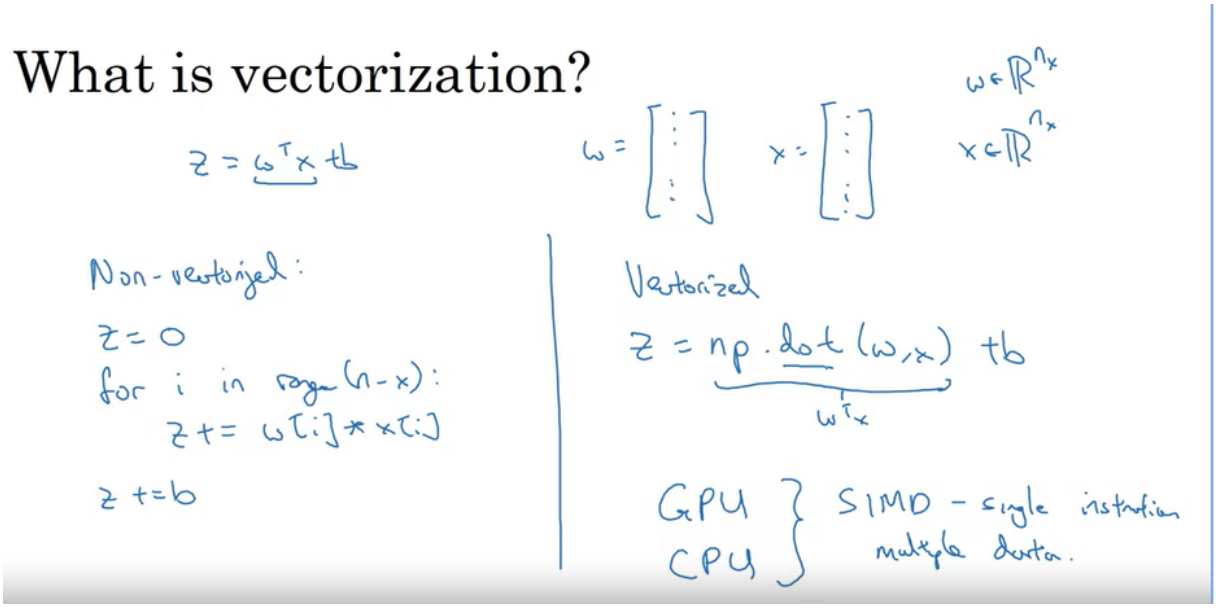
note : all the formulas above are computed and explained in the Finding the derivatives for Logistic Regression section above.

and then update w1:= w1 - (learning rate)*dw1, vice versa for all.

we can observe that in the above example there is a for loop, in reality, if we have more features, then more associated (w's), which makes the use of another "for" loop for weights (w's). This makes our DL model work slow, so we use vectorization.

Vectorization

The use of for-loops drastically decreases the efficiency of the program, so while training the DL model, we might have to wait for a long amount of time just for it to train.



But we can get over with for-loops **sometimes(not in all scenarios)** using vectorization technique.

For a glimpse see the time taken to execute code using for-loop and vectorization below:

In [1]:
```
import numpy as np
```

In [15]:
```
a = np.random.rand(1000000)
b = np.random.rand(1000000)
```

In [16]:
```
import time

tic = time.time()
x = np.dot(a,b)
toc = time.time()

print("vectorized time", str(1000*(toc - tic)), "ms")
print(x)

x = 0
tic = time.time()
for i in range(1000000):
    x += a[i]*b[i]
toc = time.time()

print("Time using for loop", str(1000*(toc - tic)), "ms")
print(x)
```

```
vectorized time 1.1518001556396484 ms
249930.35482235448
Time using for loop 337.20874786376953 ms
249930.35482235378
```

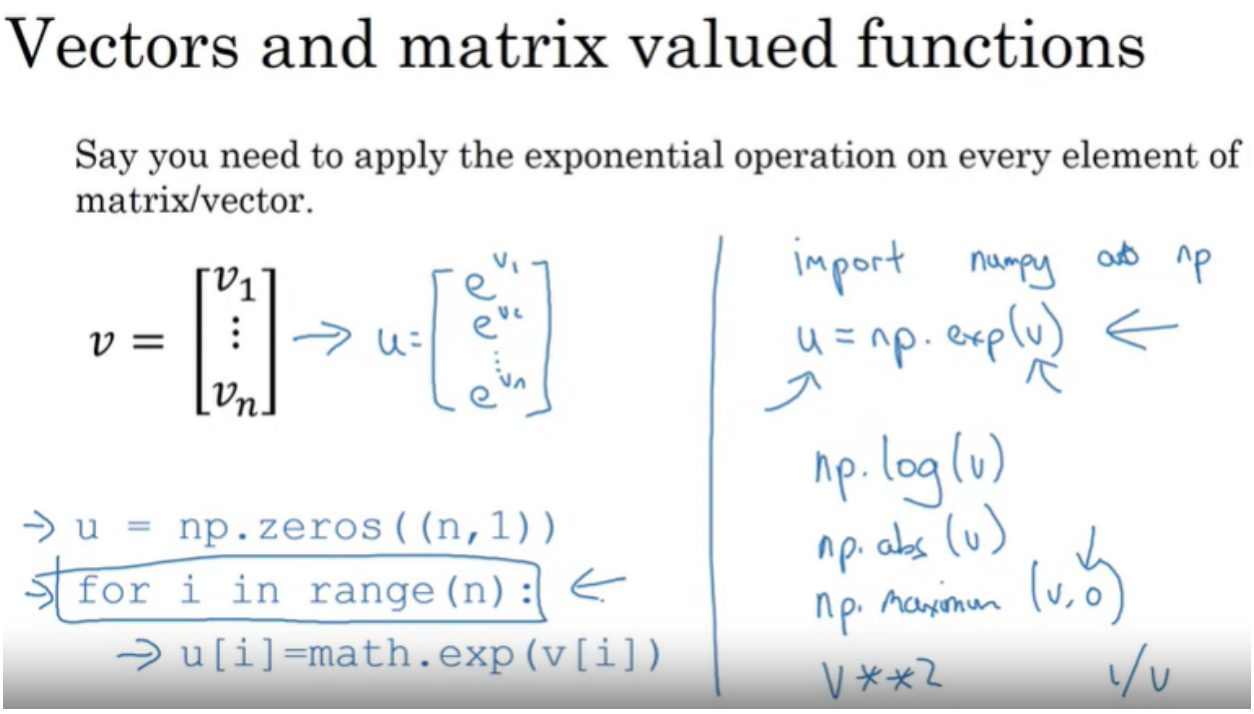
Vectorization was about 300 more faster than for loop.

credit **Andrew-NG (transcript_week2_course1)** :

But all the demos I 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.function or other functions that don't require you explicitly implementing a for loop. It enables Phyton 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

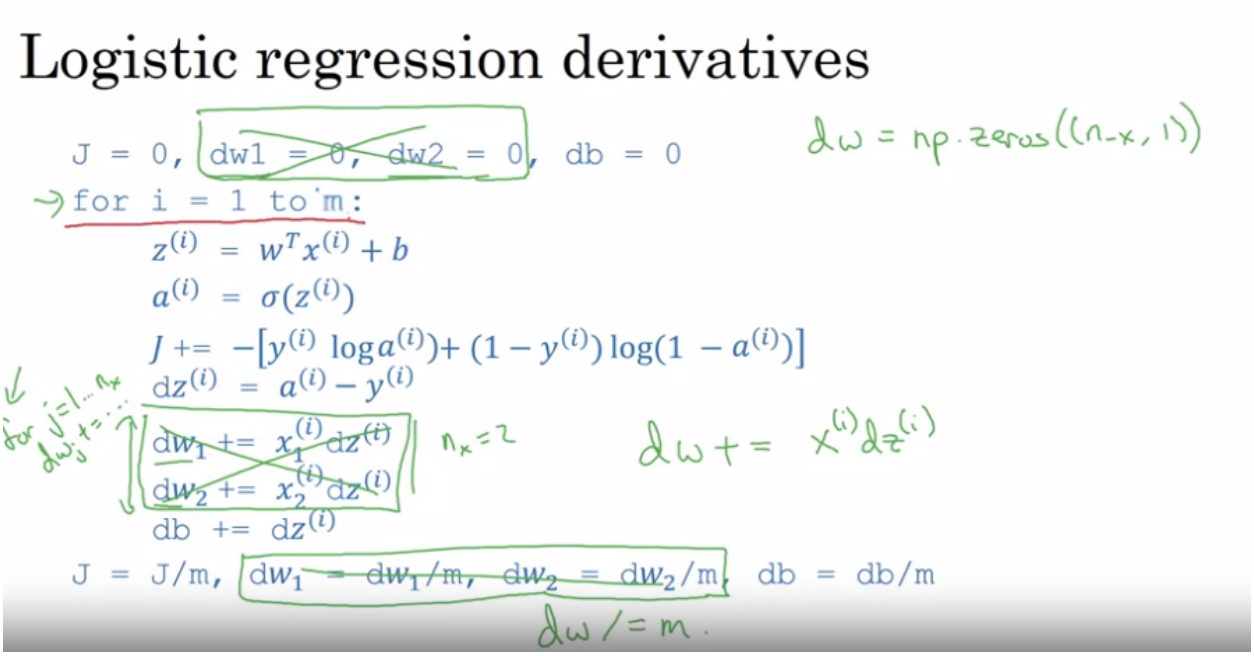
explicit for loops. Let's go onto the next video to see some more examples of vectorization and also start to vectorize logistic regression.

Neural Network programming guideline: Whenever possible, avoid explicit for-loops.



So, in the above figure we can see that left side is non-vectorized, and right hand side is vectorized.

From this Algorithm(finding gradient descent on m examples)



lets try to get rid of one for loop. In this figure dw1, dw2 are just two weights **[but in reality there could be a lot of them (equal to the number of features), (Here, for simplicity we have just taken 2 features each of m training examples)].** So this weights will require a for loop. Lets try to get rid of this specific one.

visualizing $dw += x^{(i)} \cdot dz^{(i)}$

w is a $(n_x \times 1)$ matrix

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \end{bmatrix}_{n_x \times 1}$$

dw has the same dimensions as w.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$ ← (X looks something like this).

$n_x = m$

So lets recreate w's dimensions

$dw = np.zeros(n_x, 1)$

now $x^{(i)}$ looks like $\begin{bmatrix} feature1 \\ feature2 \\ \vdots \end{bmatrix}_{2 \times 1}$

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

$a^{(i)} = \sigma(z^{(i)})$ & $z^{(i)} = w^T x^{(i)} + b$

predicted value.

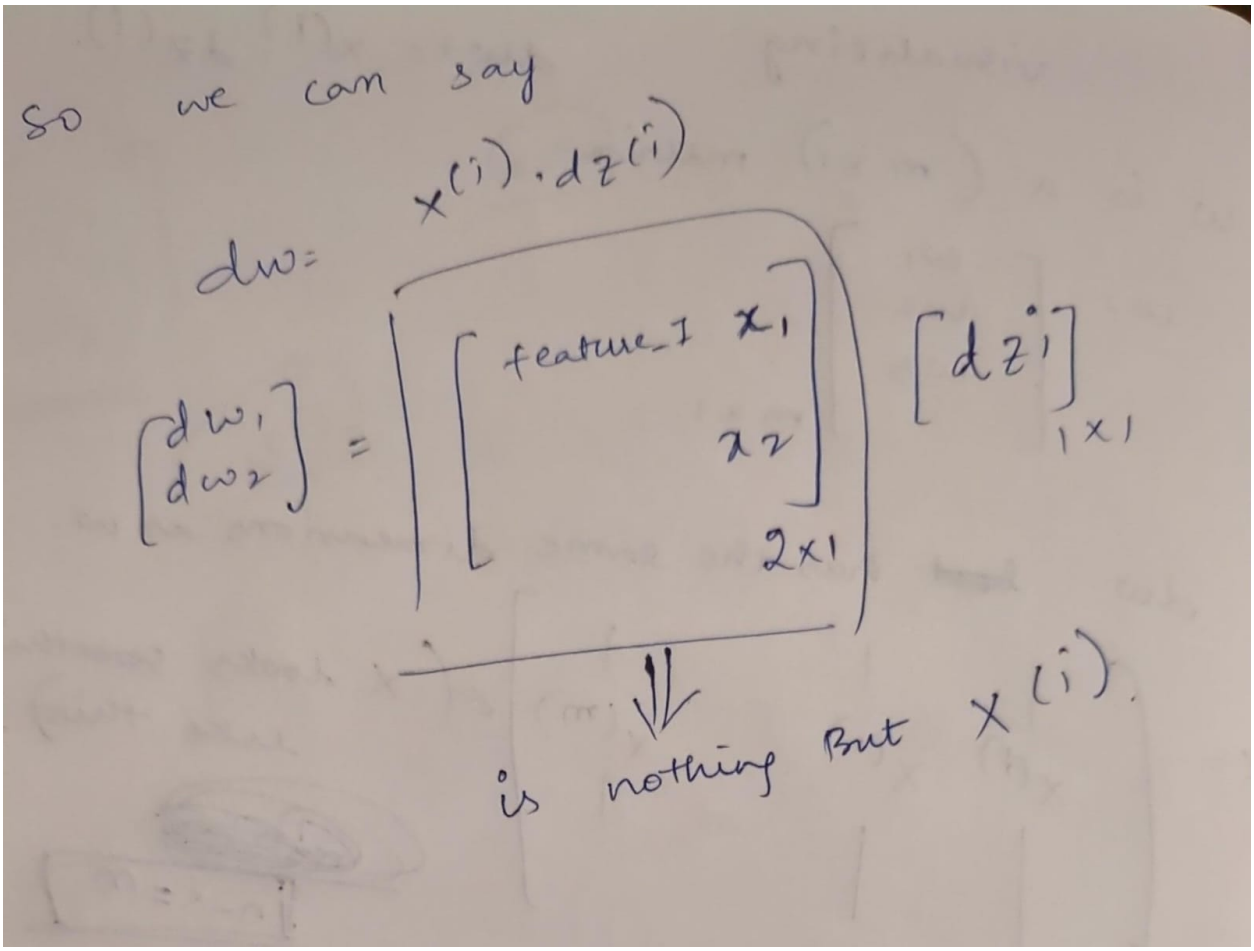
~~Did the sigmoid earlier~~

when we apply sigmoid function of $z^{(i)}$ we get $a^{(i)}$.

$a^{(i)}$ is (1,1) shape matrix.

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

will result in (1,1) shape matrix.

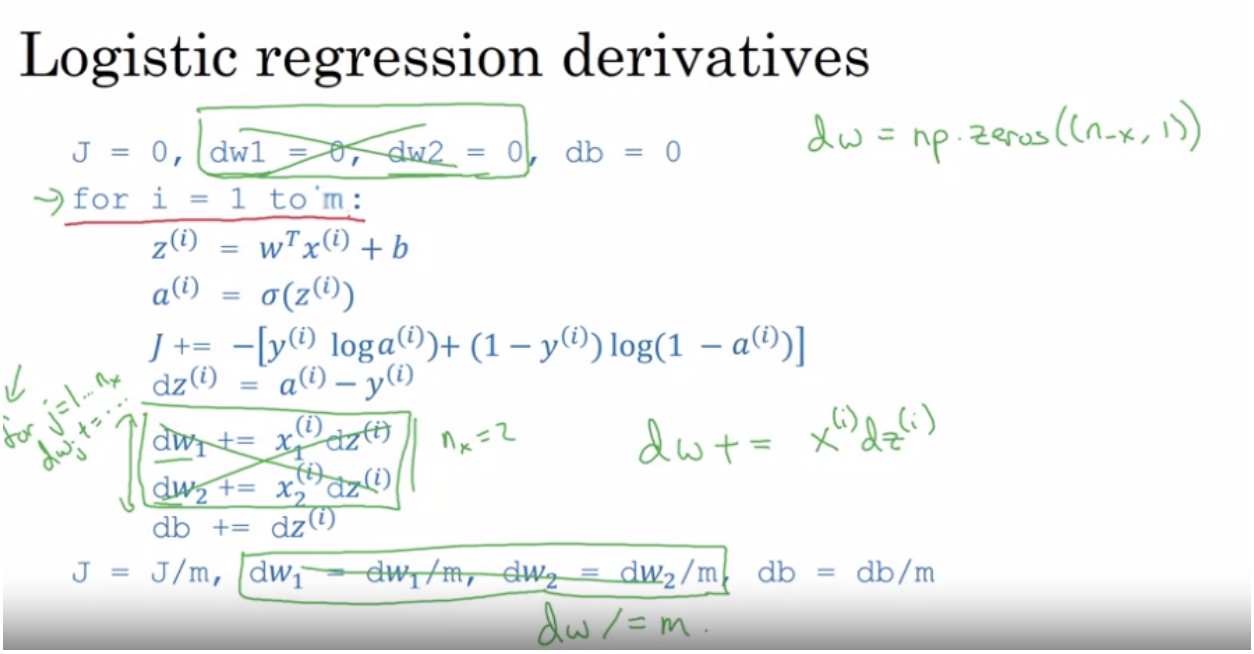


and then we are left with one thing, divide each element of dw with m.

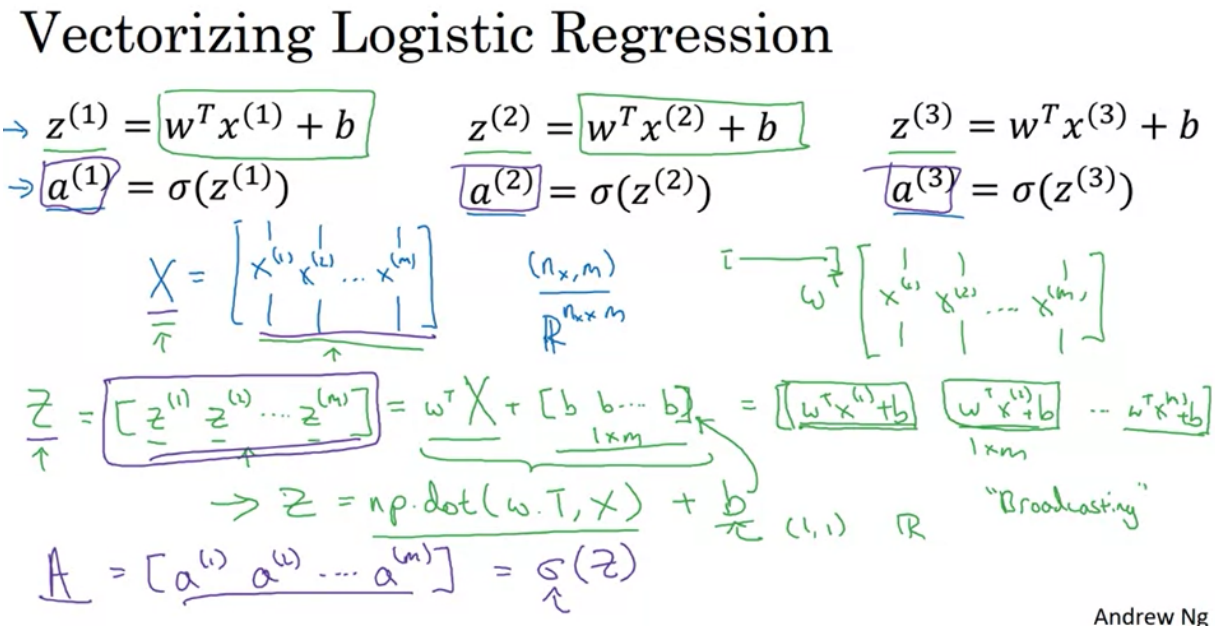
for doubts refer again:

1.
2.

Vectorizing the entire Gradient descent :

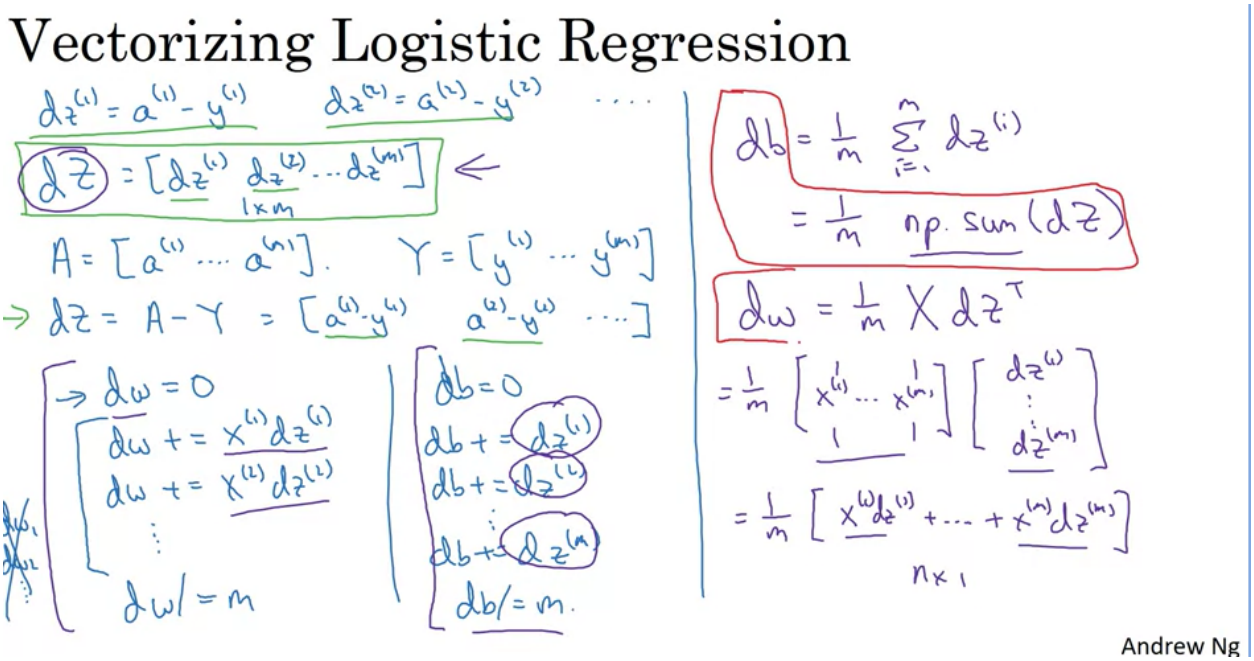


But from the above code snippet, we can see that, there is still a for loop, which effects our execution time, lets try to get a code without for-loop.



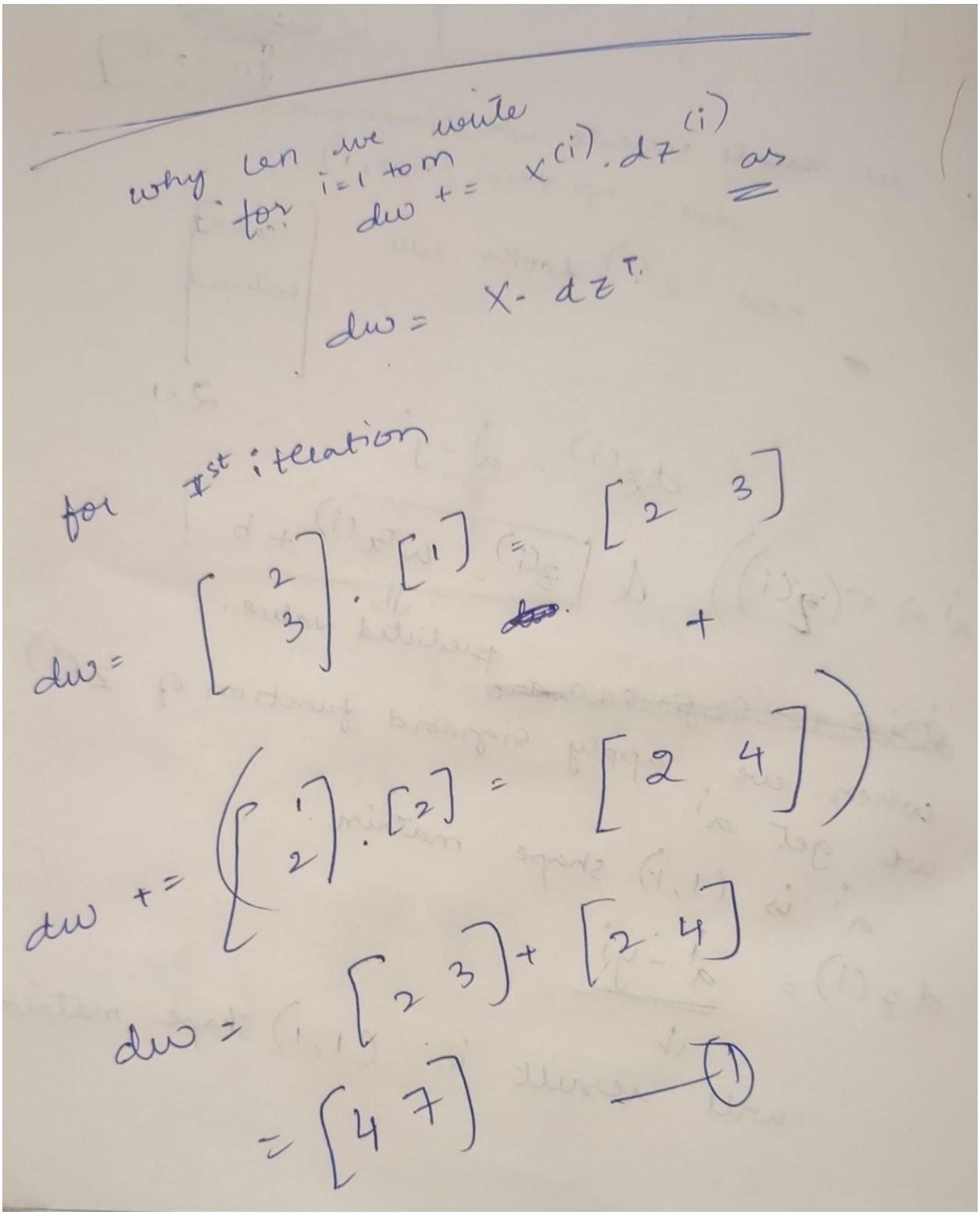
1. We are computing the predicted values, in each iteration (z^(i)), this can be vectorized as in the above figure.
2. computing the sigmoid in each iteration can as well be vectorized, sigma(z)

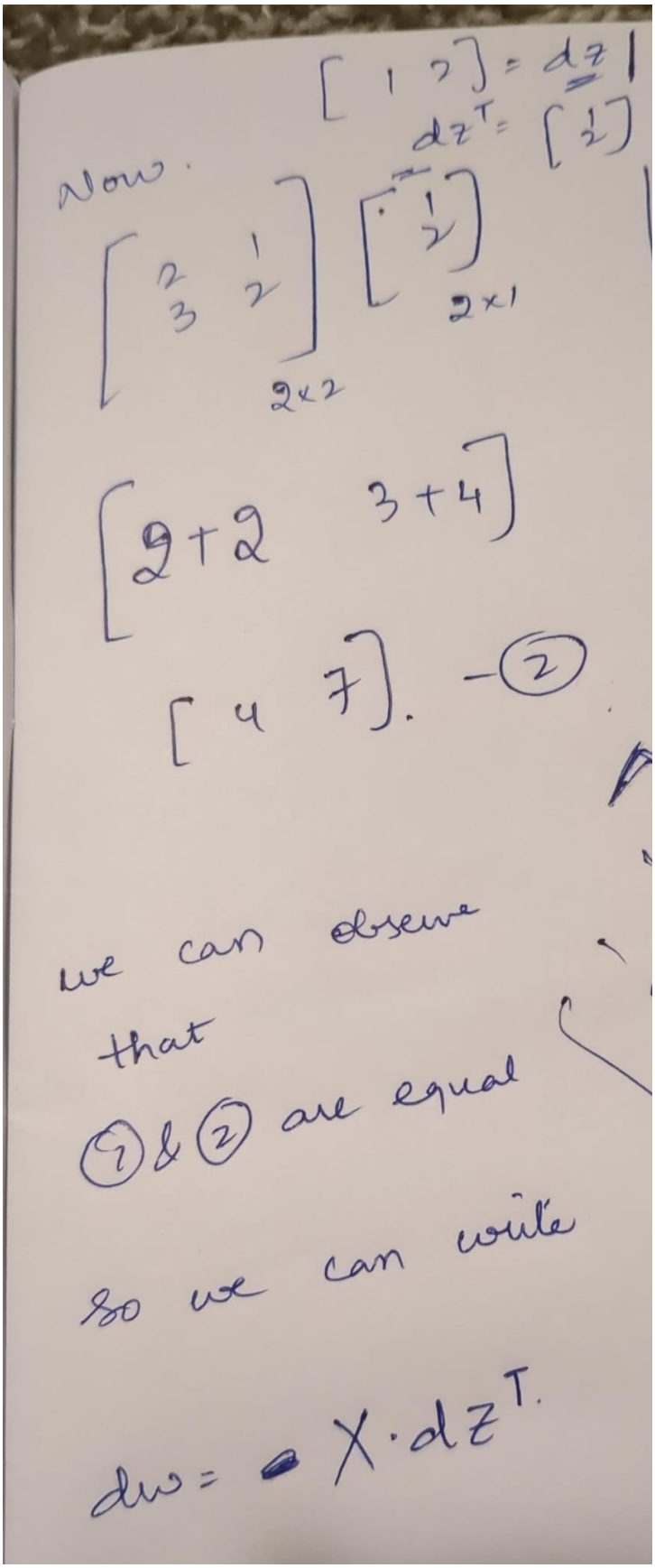
we still have to find vectorized calculation for dz, dw and db.

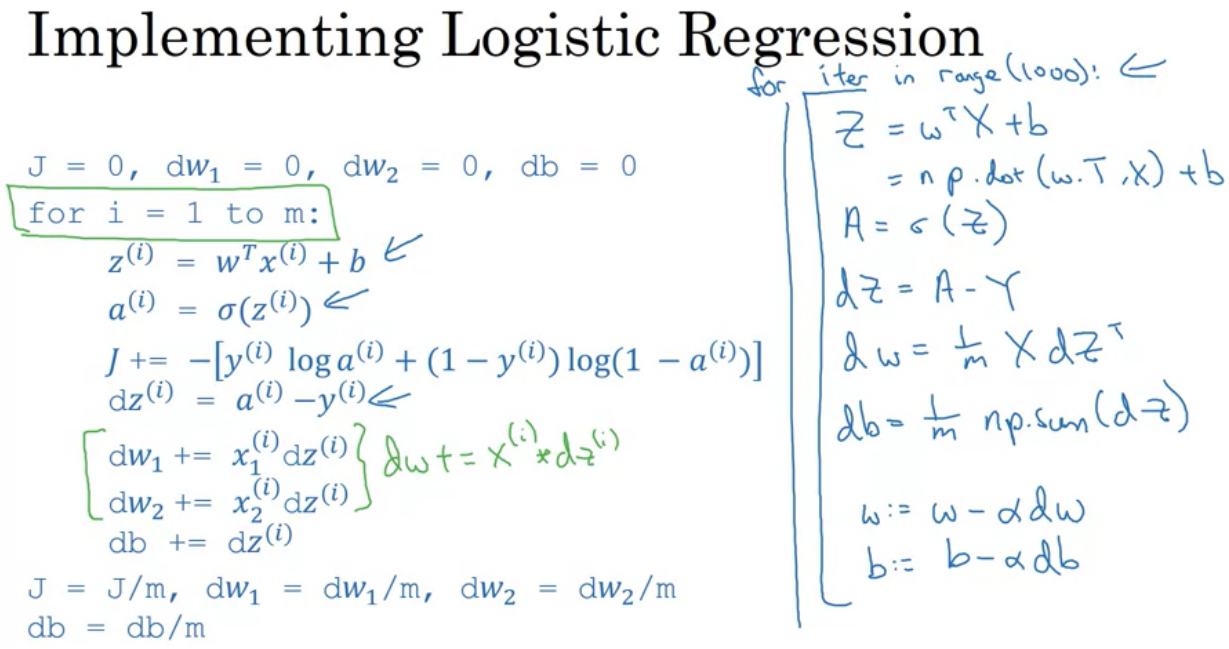


3. dz can be written as A - Y.
4. db as the average over m samples on the sum of dz.
5. for dw :

we are computing dw += x(i)*dx(i), every time, this can be vectorized as


why can we write
for i=1 to m
dw += x(i) . dz(i)
as
dw = X . dz^T
for 1st iteration
dw = [2 3] . [1] = [2 3] +
dw += ([1 2] . [2] = [2 4])
dw = [2 3] + [2 4]
= [4 7]
1

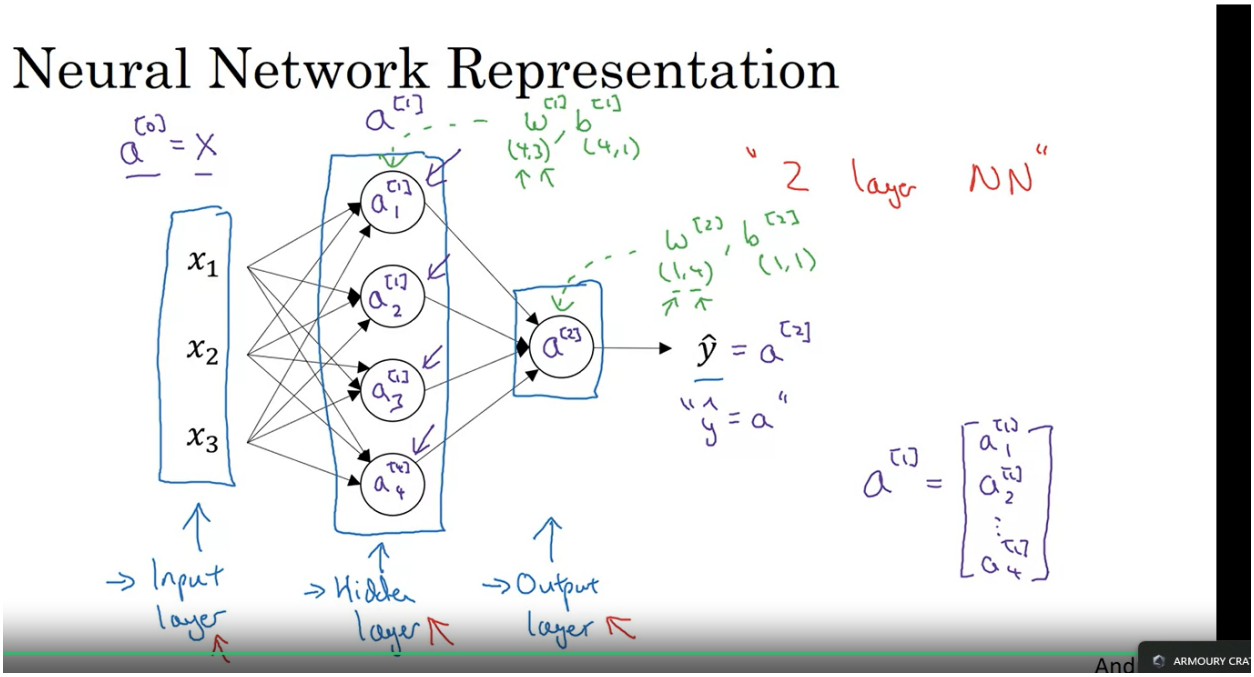

[1 2] = dz
dz^T = [1/2]
Now.
[2 1; 3 2] [1/2]
2x2
2x1
[2+2 3+4]
[4 7]. -(2)
we can observe that
(1) & (2) are equal
so we can write
dw = X · dz^T.



This is the highly optimized representation of the logistic regression to calculate derivatives in backward propagation.

Week 3 : Shallow Neural Networks

Neural Network Representation



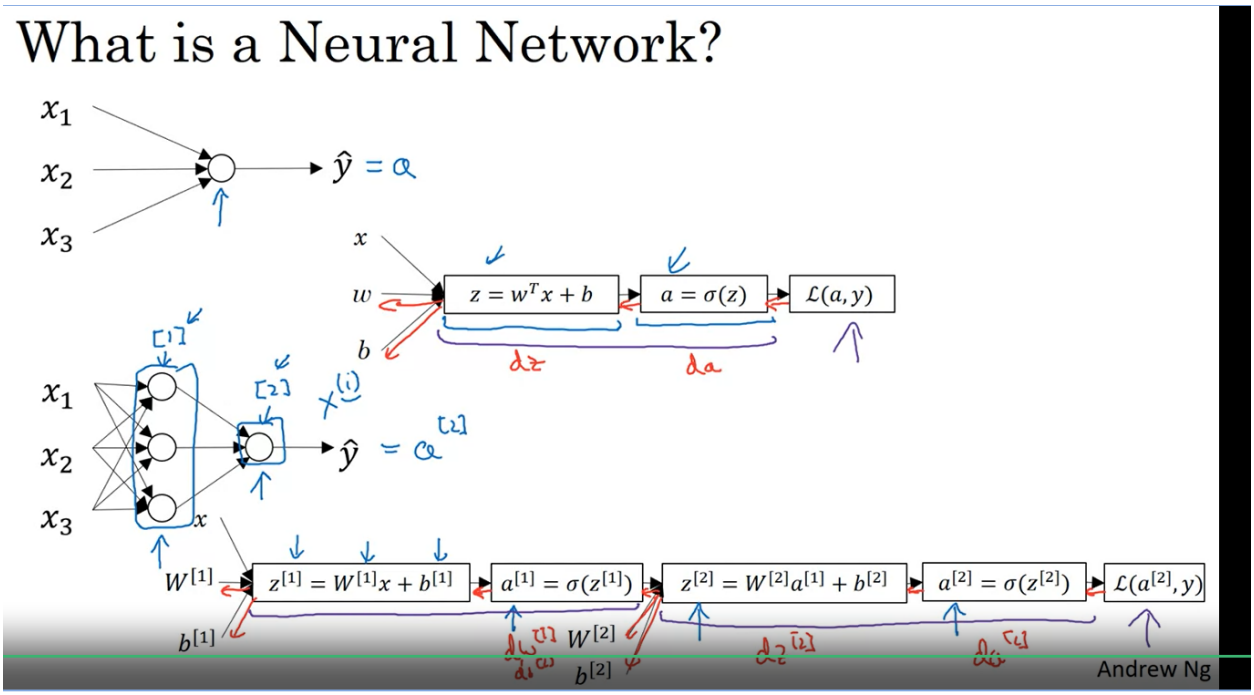
a[0], represents that it is the input that will be passed on to the other layer, a is also known as acitvation.

→ In general Terms, a means that, this input will be passed on the next layer.

→ The above image is a two layer neural network, (it should have been 3 layers), but by convention it is called 2 layered neural network.

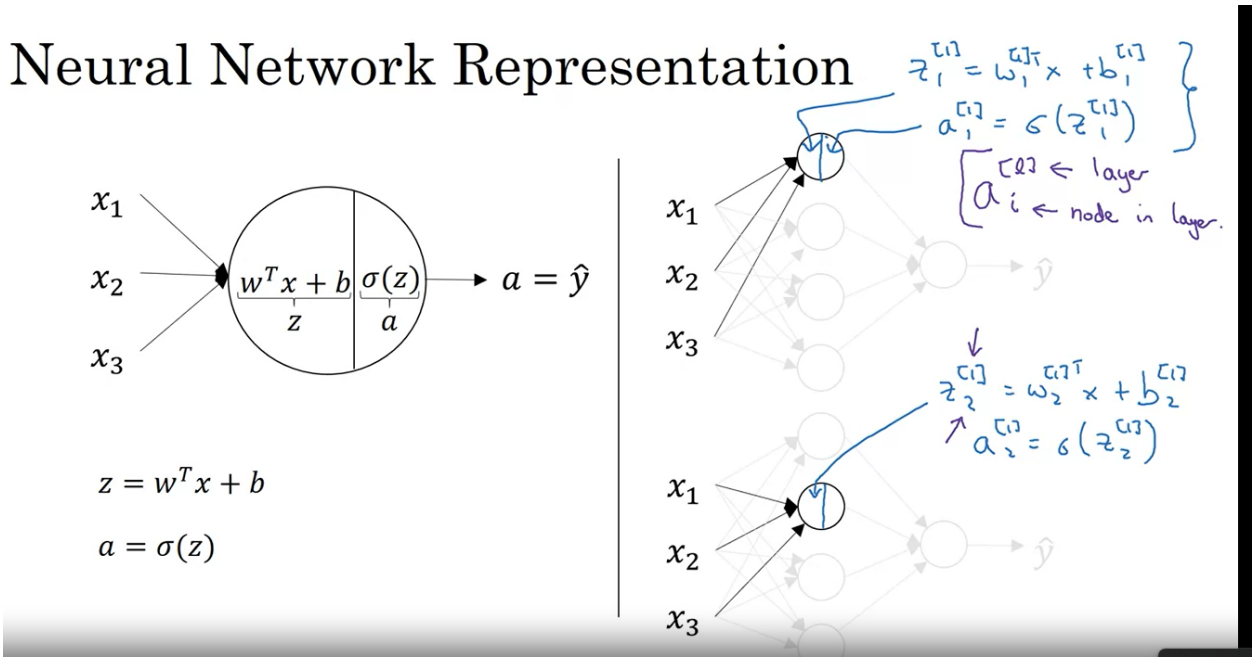
→ the first layer is called the input layer and the last one is obviously called the output layer.

→ a[1] and a[2] are associated with parameters, w and b.

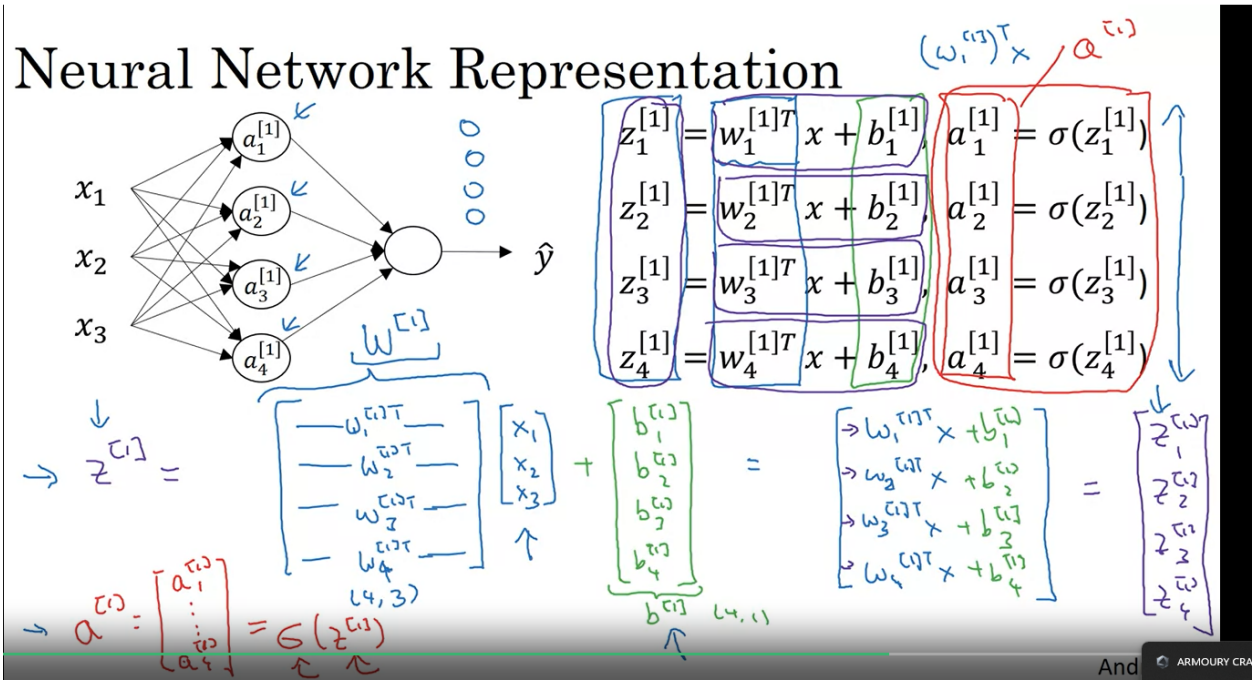


So, for each node in a[1], we calculate z and corresponding sigma value.

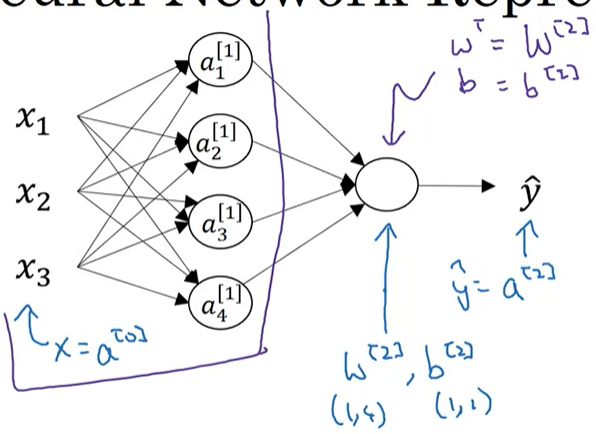
as shown in the figure above.



This gives a better and clear picture of what is going on.



Neural Network Representation learning



Given input x:

$$z^{[1]}_{(4,1)} = W^{[1]}_{(4,3)} a^{[0]}_{(3,1)} + b^{[1]}_{(4,1)}$$

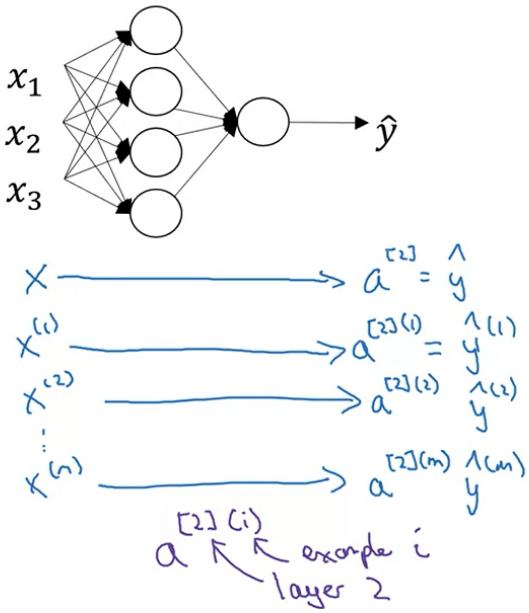
$$a^{[1]}_{(4,1)} = \sigma(z^{[1]}_{(4,1)})$$

$$z^{[2]}_{(1,1)} = W^{[2]}_{(1,4)} a^{[1]}_{(4,1)} + b^{[2]}_{(1,1)}$$

$$a^{[2]}_{(1,1)} = \sigma(z^{[2]}_{(1,1)})$$

Until this point, we were focusing on features of one sample, now lets consider we have more than one examples

Vectorizing across multiple examples



$$\left.\begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \end{aligned}\right\} \leftarrow$$

→ for i = 1 to m,

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

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

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

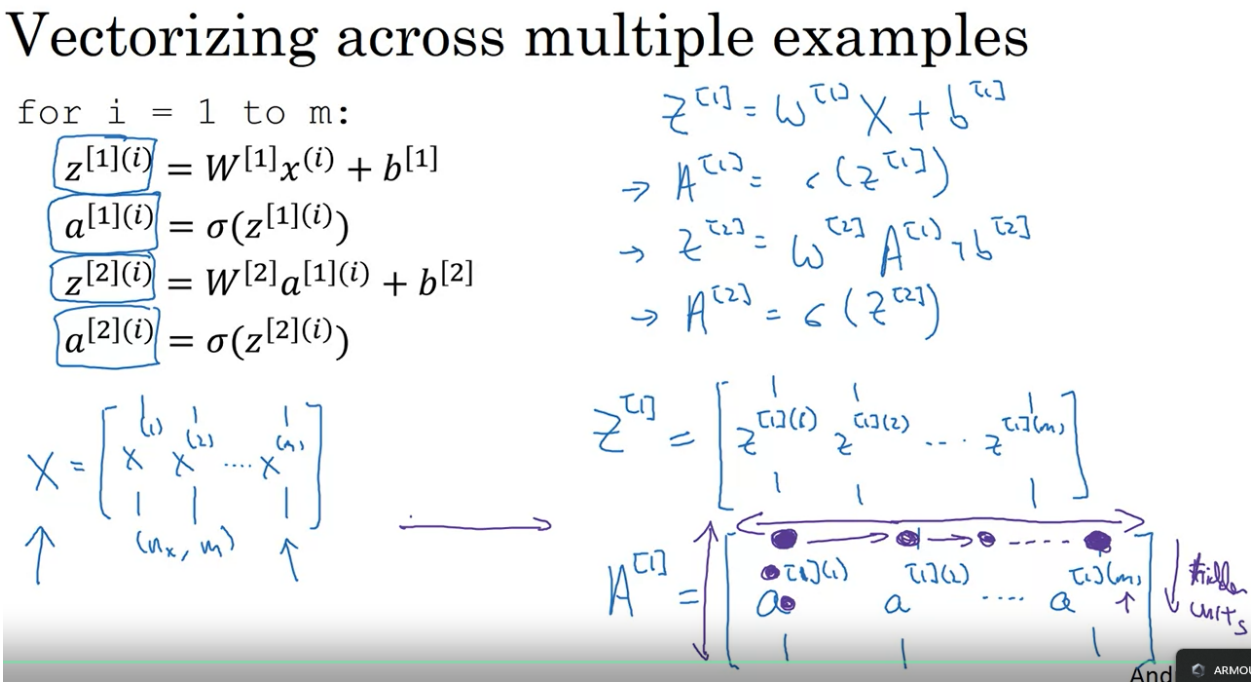
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

And

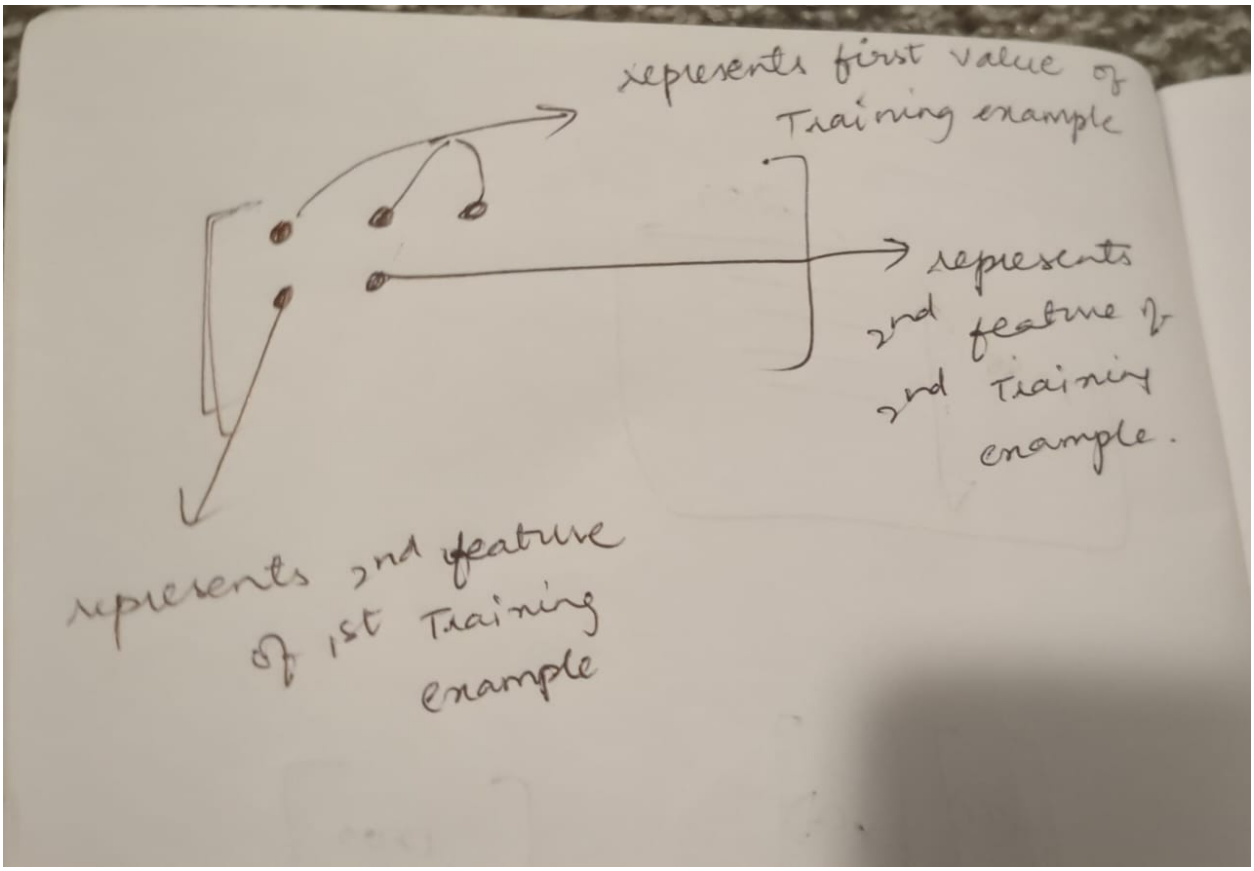
This is how it looks like, when there are multiple examples, we can observe that X(i) matrix represents features at sample i, and a[2](i) represents predicted value at 2nd layer, that is the output, and i is that particular index.

and the **CODE** to the right, is how can we write code for multiple training examples.

Absolutely we need to vectorize it,



CODE to the right is how you vectorize it.



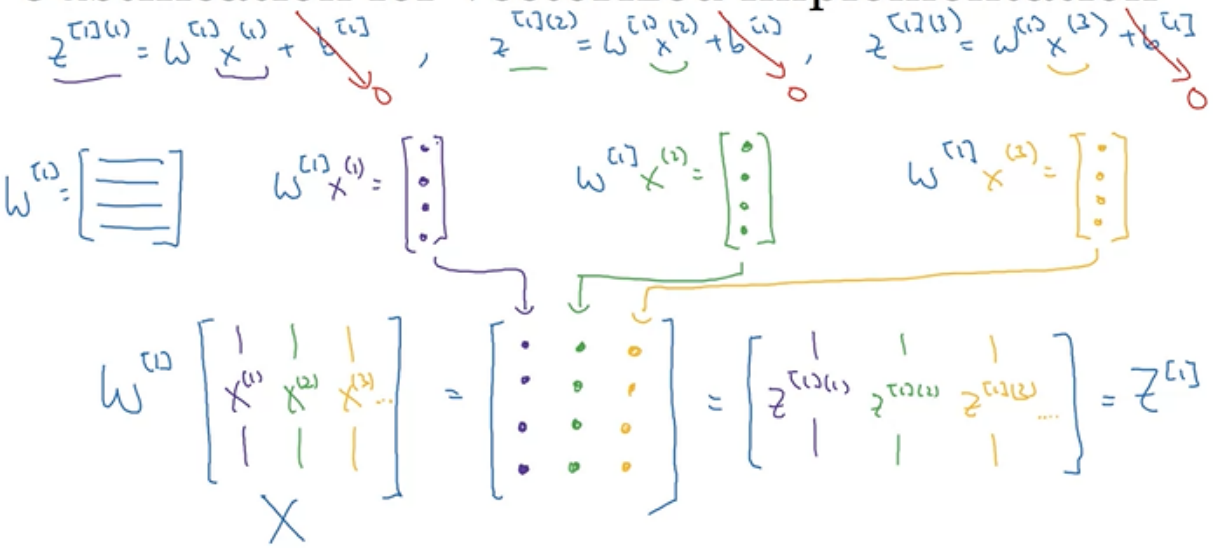
note :

a[i], z[i], in here [] → represents values at that particular layer

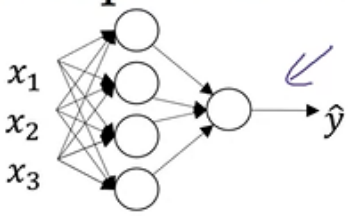
a(i), z(i), in here () → represents the value at that particular sample.

Justification for the Vectorized Implementation :

Justification for vectorized implementation



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
```

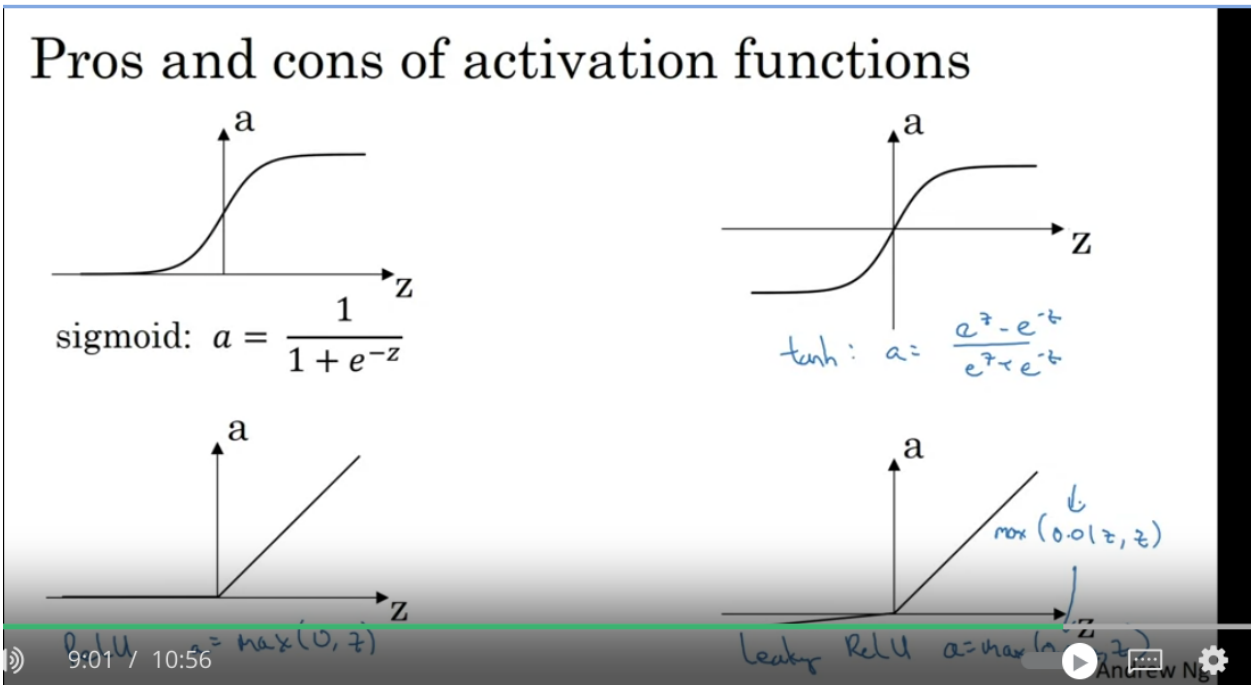
$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

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

Andrew Ng

Thus can be written as above.

Activation Functions



these are the different activation functions available,

→ do not use sigmoid except for output layer of the logistic regression.

→ tanh almost always works well than sigmoid, so use it whenever possible. but we have to use the sigmoid in the output layer of logistic regression because, because we expect values from 0 to 1.'

→ RELU is the best alternative possible, because for sigmoid, at extreme values the slope almost reaches 0, which makes the learning a lot slower, but for Rectified LInear Unit (RELU), the slope is 1 for all values greater than 0, which makes the learning faster.

If you observe clearly the slope at point zero is infinity, and the probability of output getting 0 is really low, hence we trust RELU.

The curve for sigmoid activation and tanh activation reaches near zero at some point, making learning slower, but why ?

NOTE : just in case, if you are wondering what slope has to do with activation functions, remember in backward propagation we calculate how the values of output changes by the small change in the input, which is nothing but slope right, if the curve we choose has low slope then definitely the gradient descent will be slower.

Why do you need non-linear activation functions ?

The below link explains why we need non-linear activation functions ?

https://www.v7labs.com/blog/neural-networks-activation-functions#3-types-of-neural-networks-activation-functions

when we use the linear activation function

g(z) = z

g(z) is also called as identity activation function, provides the output as it is.

NOTE : Honestly, not intuitivly clear why to use activation function.

→ they are used to notice important patterns

→ if you dont use activation functions, basically at the end, what we are seeing is just a linear regression model of machine learning.

Linear Activation Function:

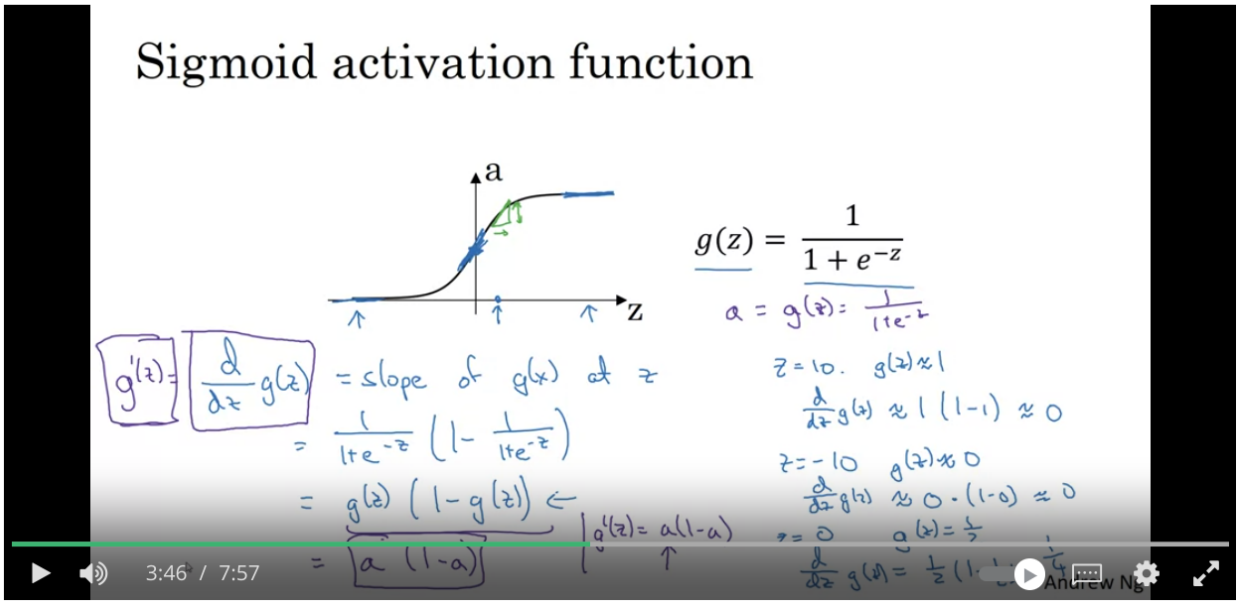
- In the case of a purely linear activation function, the derivative is a constant, and it's always 1. When backpropagating through multiple layers, this constant gradient can cause the gradients to either vanish or explode, making it difficult for the network to effectively learn and adapt.

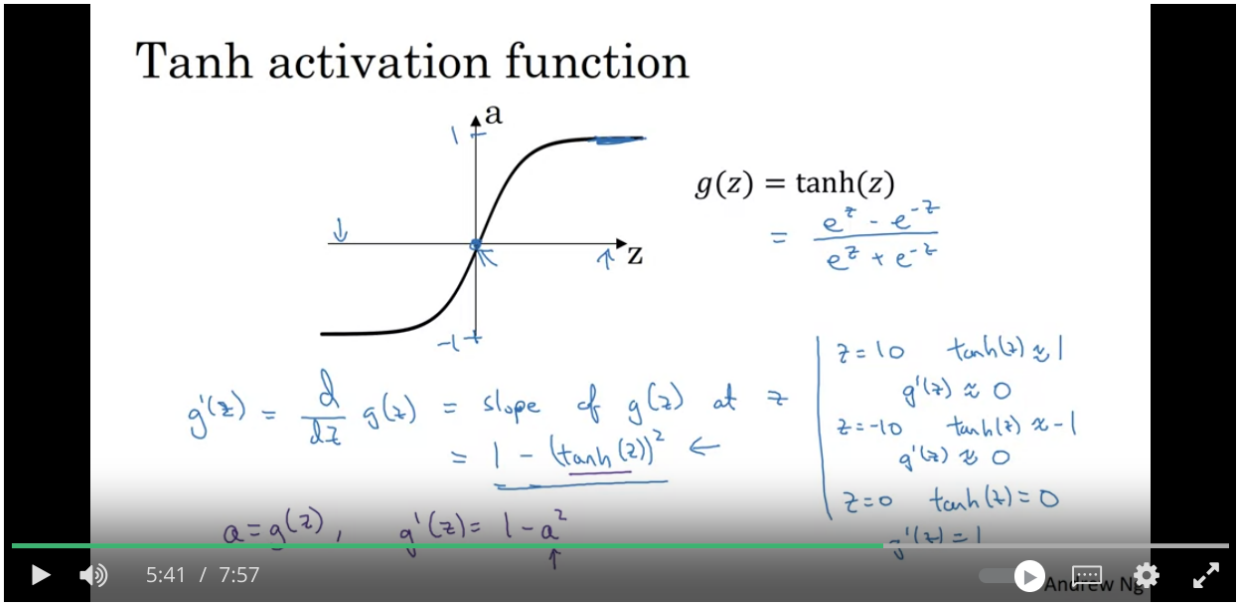
ReLU to the Rescue:

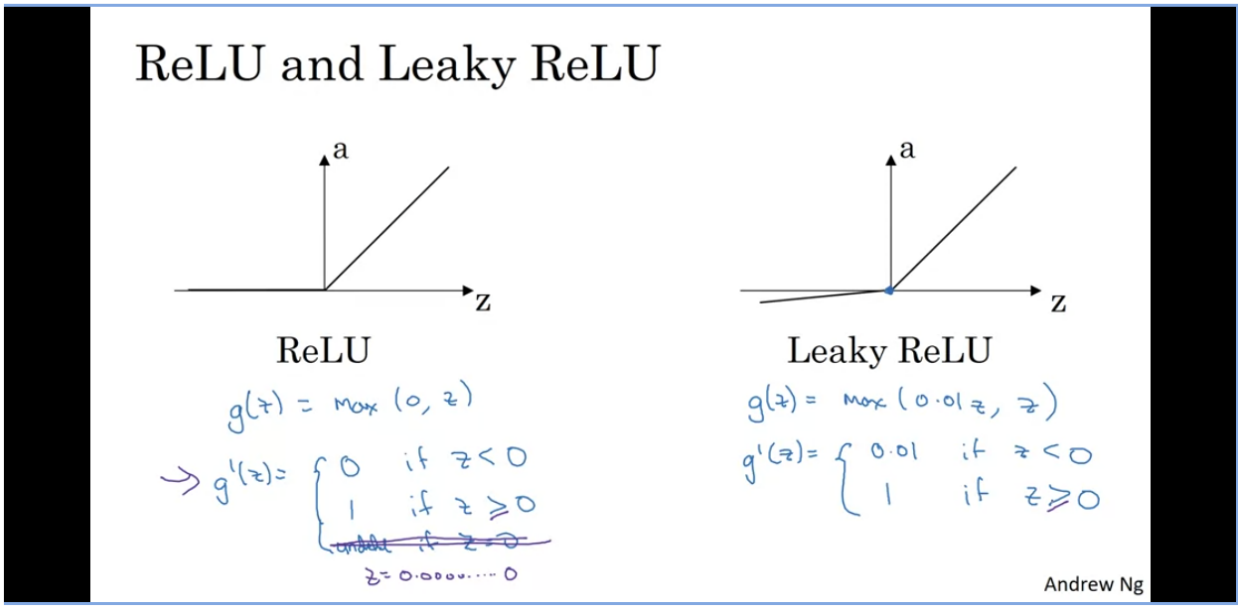
- ReLU, by setting negative values to 0, introduces non-linearity and helps mitigate the vanishing gradient problem. For positive values, the gradient is a constant 1, allowing for effective gradient propagation and weight updates. This makes ReLU well-suited for deep networks, where the ability to effectively learn and adapt through many layers is crucial.

While ReLU helps address the vanishing gradient problem, it's not without its challenges, such as the potential for neurons to become "dead" (always outputting zero for any input less than or equal to zero). There are variations of ReLU, like Leaky ReLU and Parametric ReLU, designed to handle some of these issues. The choice of activation function is an essential consideration in designing and training neural networks.

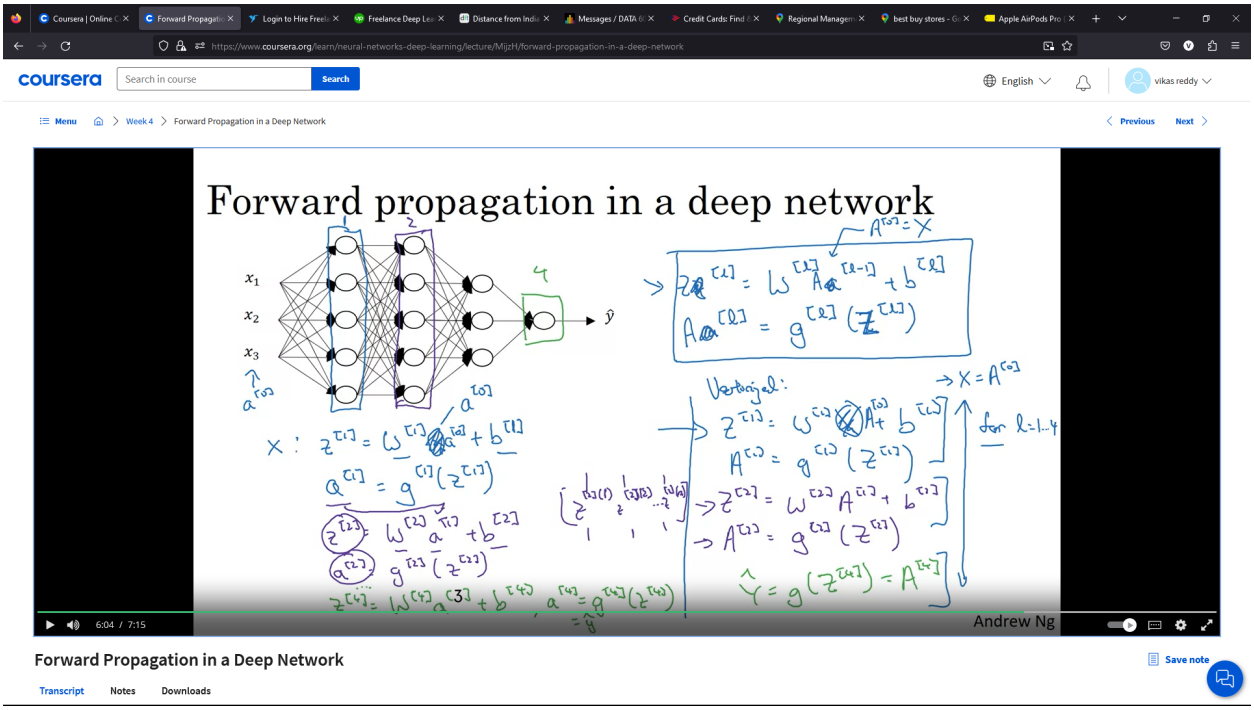
Derivatives for Activation Functions


Sigmoid activation function
a
z
$g(z) = \frac{1}{1+e^{-z}}$
$a = g(z) = \frac{1}{1+e^{-z}}$
$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(x)$ at z
$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$
$= g(z)\left(1 - g(z)\right)$
$= a(1-a)$
$g'(z) = a(1-a)$
$z = 10$. $g(z) \approx 1$
$\frac{d}{dz} g(z) \approx 1\,(1-1) \approx 0$
$z = -10$ $g(z) \approx 0$
$\frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$
$z = 0$ $g(z) = \frac{1}{2}$
$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$
3:46 / 7:57
Andrew Ng


Tanh activation function
a
z
1
-1
$g(z) = \tanh(z)$
$= \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$
$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(z)$ at z
$= 1 - (\tanh(z))^2$
$a = g(z)$, $g'(z) = 1 - a^2$
$z = 10$ $\tanh(z) \approx 1$
$g'(z) \approx 0$
$z = -10$ $\tanh(z) \approx -1$
$g'(z) \approx 0$
$z = 0$ $\tanh(z) = 0$
$g'(z) = 1$
5:41 / 7:57
Andrew Ng



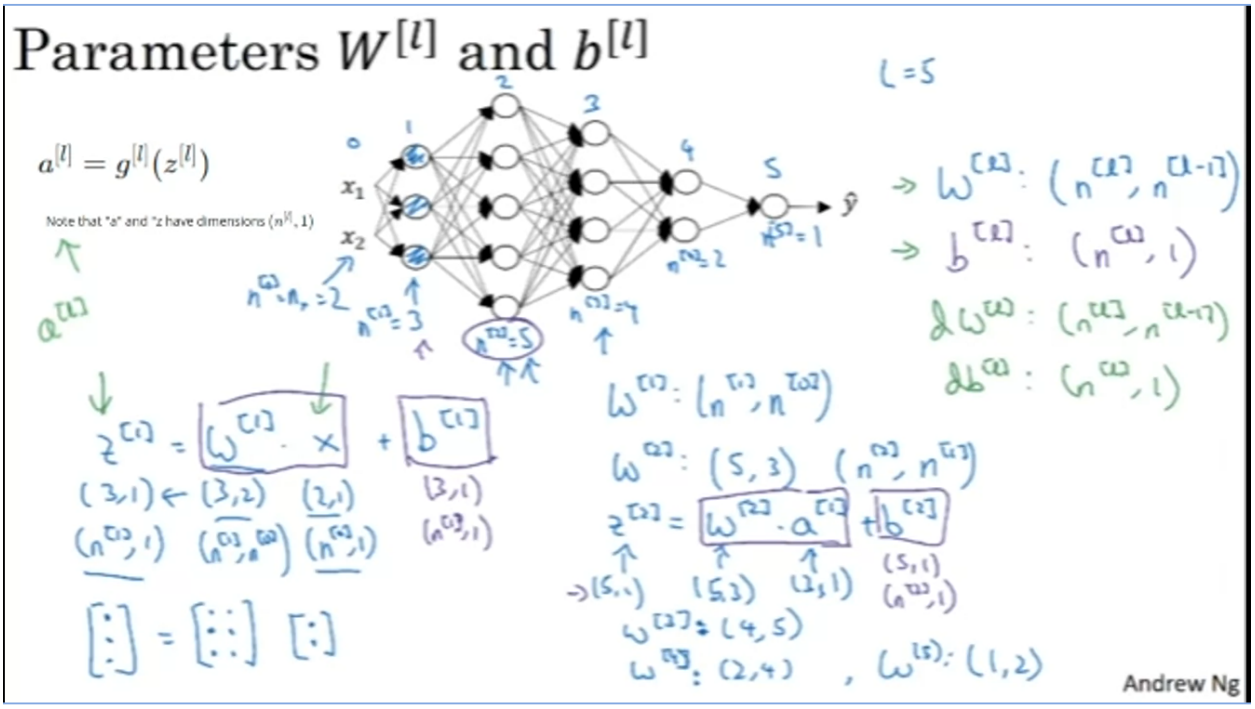
Deep Layer neural Network



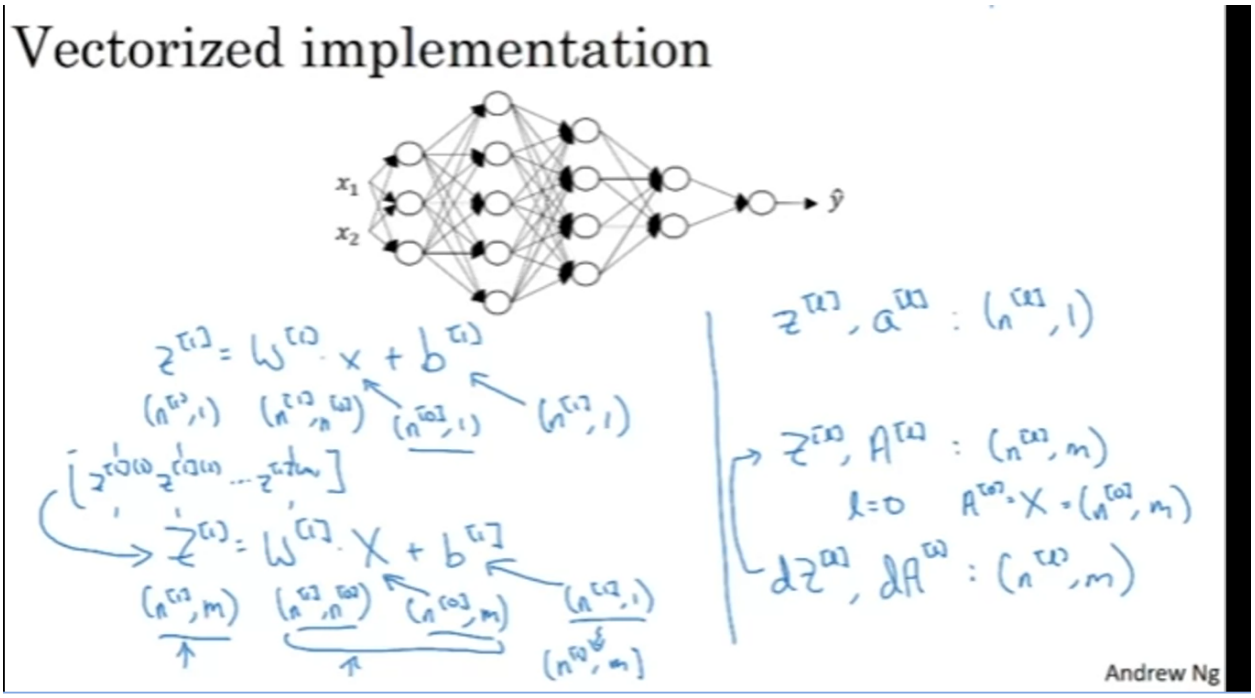
left part of the image, shows implementing the neural network for single example, and the right part shows the way to implement neural network for multiple examples.

There is a need for using for loop, to calculate the Z, and A using for loop, for each and every layer. There is no known way to vectorize it currently.

Getting the Dimensions Right

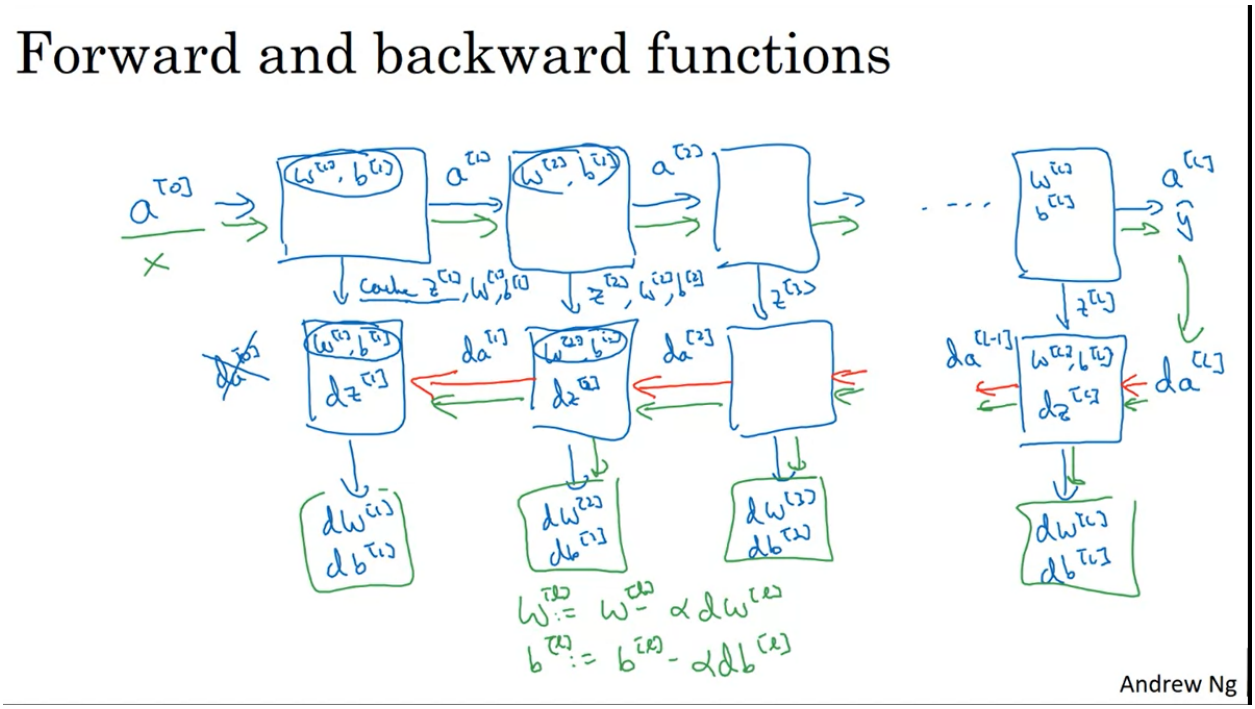


These are the dimensions for the single example



Dimensions for multiple examples

Building blocks of neural networks



In each and every step, there is a forward propagation and backward propagation step going on, this is the visual representation.