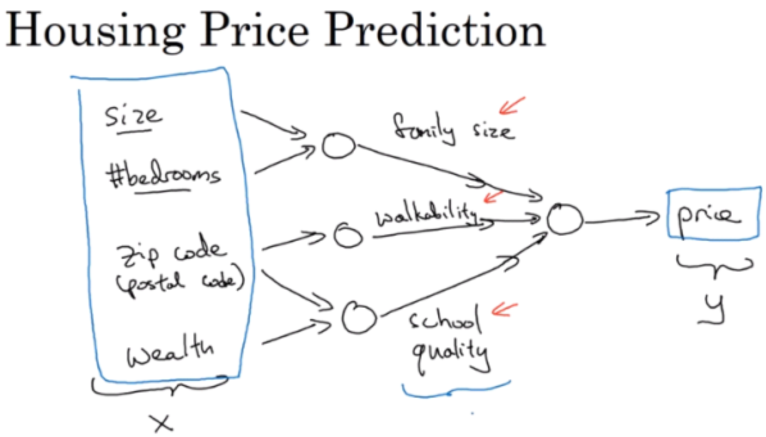
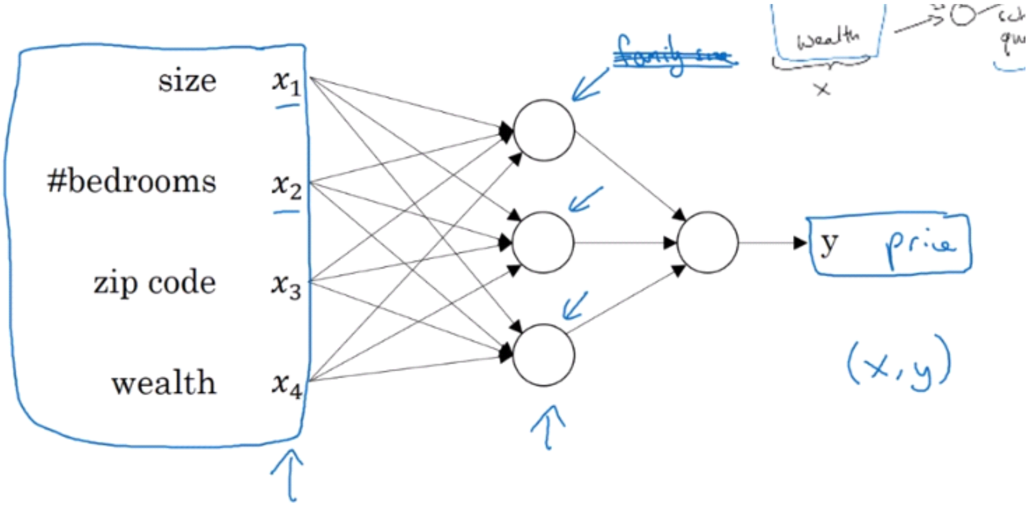
Week 1

Friday, December 21, 2018 9:10 PM



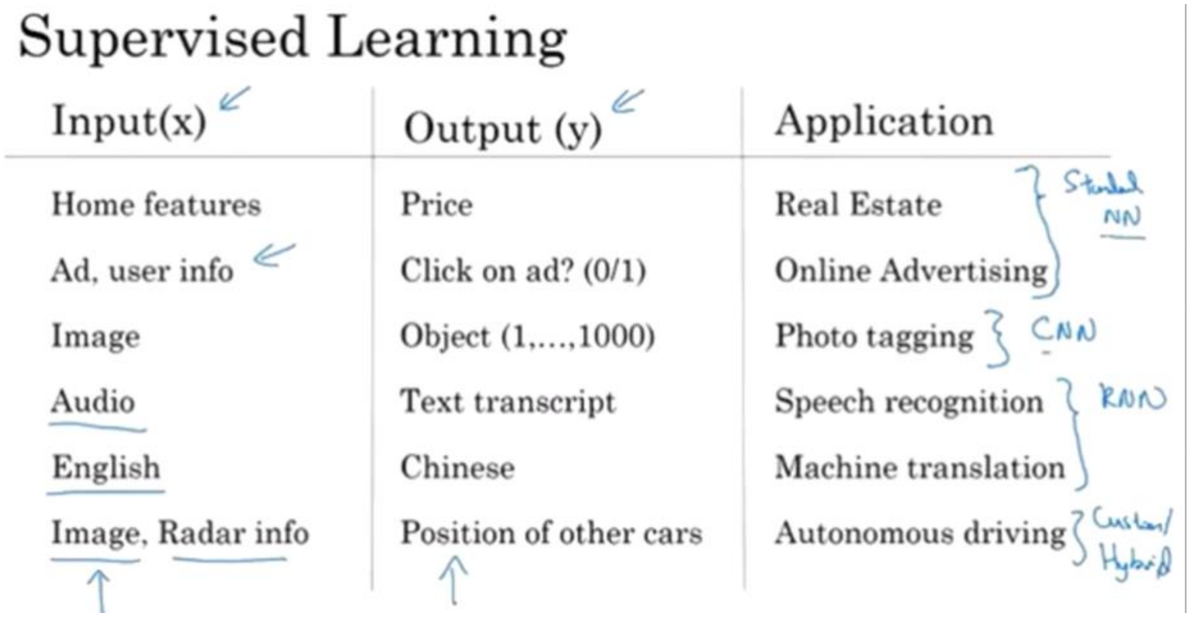
So neural network has multiple layers, with one input, n number of hidden and one output layer
And the hidden layer determine how complex is the neural network, as well as which features to give more importance.

Like above we gave features to it and now it's the neural network responsibility to determine the values in hidden layer.



So this is the actual neural network

Mostly neural network work with supervised learning

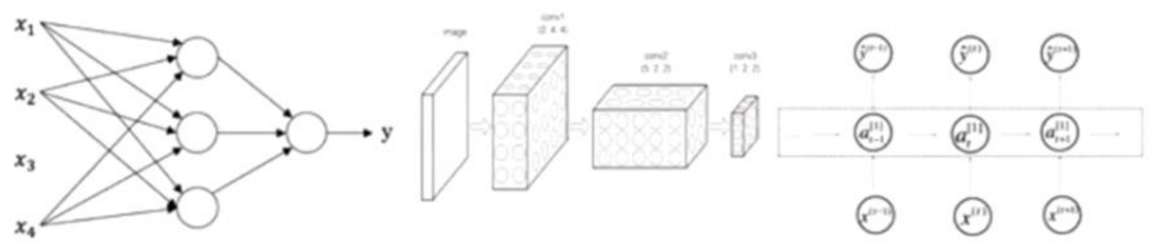


Supervised Learning

Input(x)	Output (y)	Application	
Home features	Price	Real Estate	Standard NN
Ad, user info	Click on ad? (0/1)	Online Advertising	
Image	Object (1,…,1000)	Photo tagging	CNN
Audio	Text transcript	Speech recognition	RNN
English	Chinese	Machine translation	
Image, Radar info	Position of other cars	Autonomous driving	Custom/ Hybrid

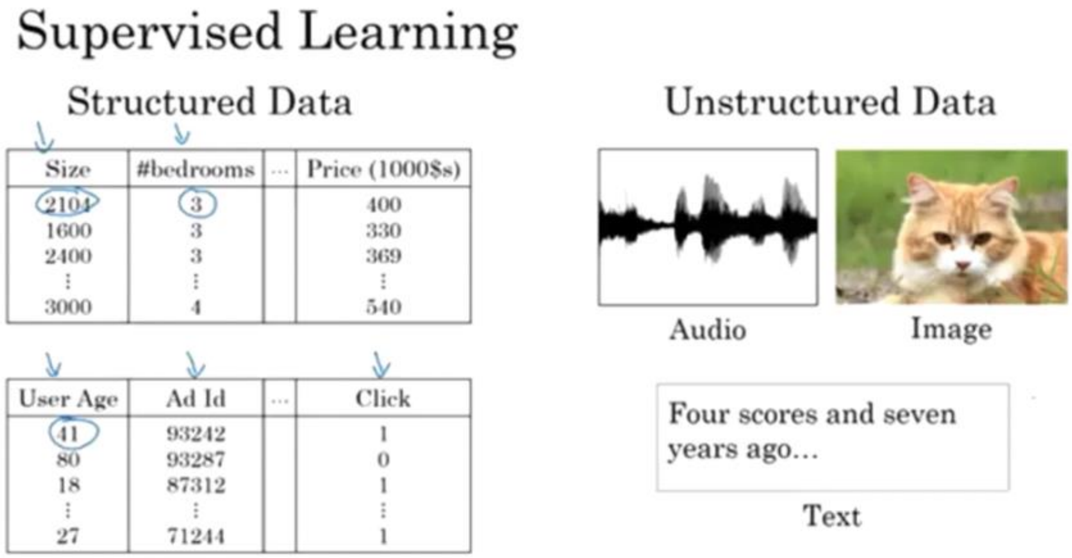
Above are some application of deep learning. And along with that the type of neural network mostly used by that application

Neural Network examples



Standard NN Convolutional NN Recurrent NN

So above are just a brief structure of NN



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…

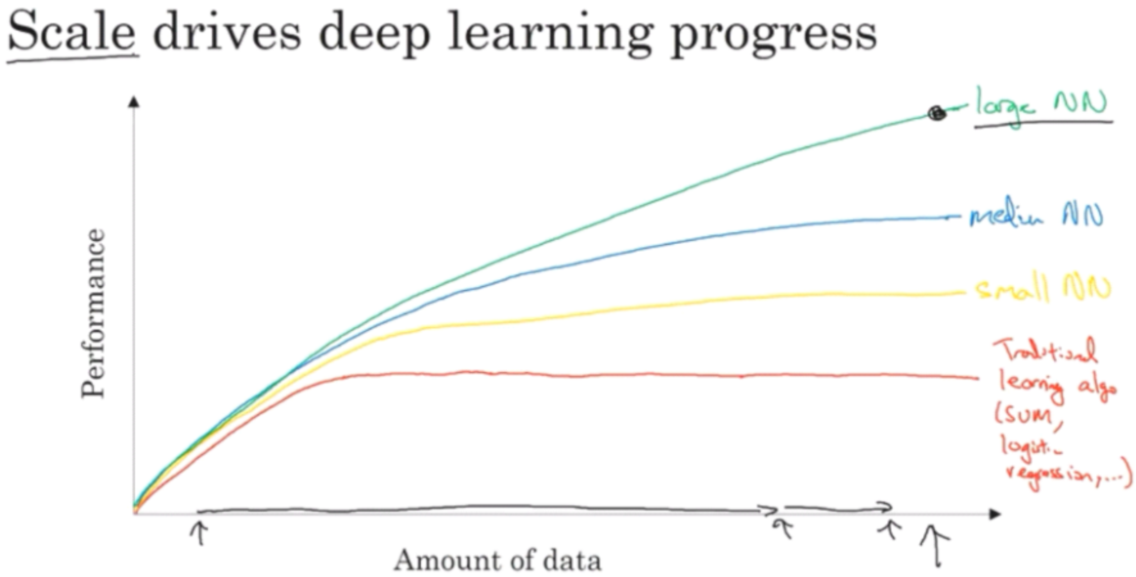
Text

Structured as data in our database
And the unstructured are stored in the form of audio, or image, or text. Here the information is the values of pixel in image, or the count of words in text

Unstructured was diff for computer to understand

NN are also working on unstructured data, for eg advertisement and in time series etc

WHY NOW?

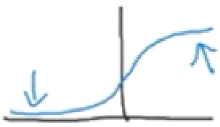


So for the traditional learning algo like SVM and regression the performance eventually platues after certain amount of data is trained. But for the NN, the more complex and more data a NN can have, the more well it will be able to perform. So with the amount of data we have in todays world, NN makes sense.

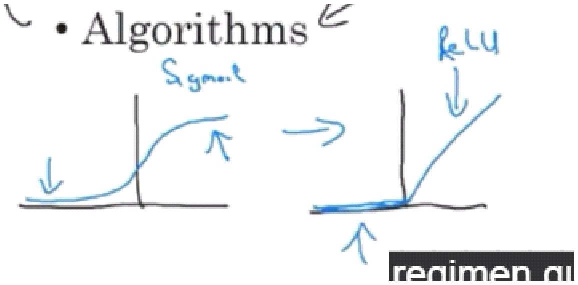
Some more context to the above, by amount of data we mean labeled data. And also if you don’t have much data, meaning we are in the starting of the x axis then there might be chance that SVM performs better than NN. But it is the right part of the graph where the amount of data is huge where NN make more sense.

Algorithms have also revolutionized the deep learning process. And mostly the new algo have improved the speed of the NN. And one of the changes is instead of using sigmoid activation function we are now using ReLU

Why this is



The arrows in the sigmoid are the area where gradient is nearly zero, and as we use gradient descent, this affect the time in learning
Where as for ReLU gradient is equal to 1 for all positive values

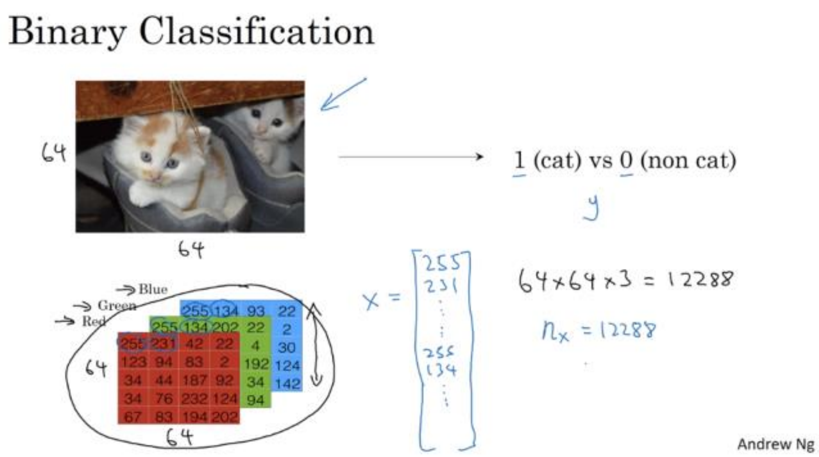


- Algorithms

regimen gu

Week 2

Friday, December 21, 2018 10:40 PM



Binary Classification

1 (cat) vs 0 (non cat)

y

$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$

$64 \times 64 \times 3 = 12288$

$n_x = 12288$

Andrew Ng

So here we are using 64x64 image. Also the image is colored so we have three sets of 64x64. Now the total number of features will be 12288.

Notation:

Single training example is (x,y) where x is real number and y is either zero or one.
We will have m training examples represented as below:

$$(x, y) \qquad x \in \mathbb{R}^{n_x}, \; y \in \{0, 1\}$$

$$m \text{ training examples}: \{(\underline{x}^{(1)}, y^{(1)}), (\underline{x}^{(2)}, y^{(2)}), \ldots, (\underline{x}^{(m)}, y^{(m)})\}$$

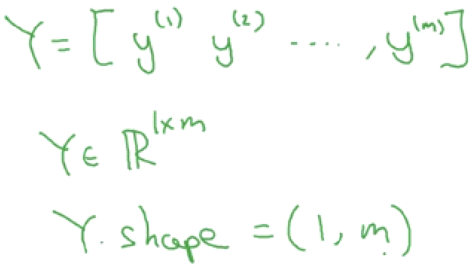
$$M = M_{train} \qquad M_{test} = \#\text{test examples.}$$

So we will be stacking our training example as below,
So each column will be a representing an example
Number of rows will be number of feature

$$X = \underbrace{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}}_{m} \updownarrow n_x \qquad \cancel{\begin{bmatrix} — x^{(1)T} — \\ \vdots \\ — x^{(m)T} — \end{bmatrix}}$$

As seen in right there might be a chance that other represent X as row for each example. But Andrew says that left representation is more helpful

$$X.\text{shape} = (n_x, m)$$

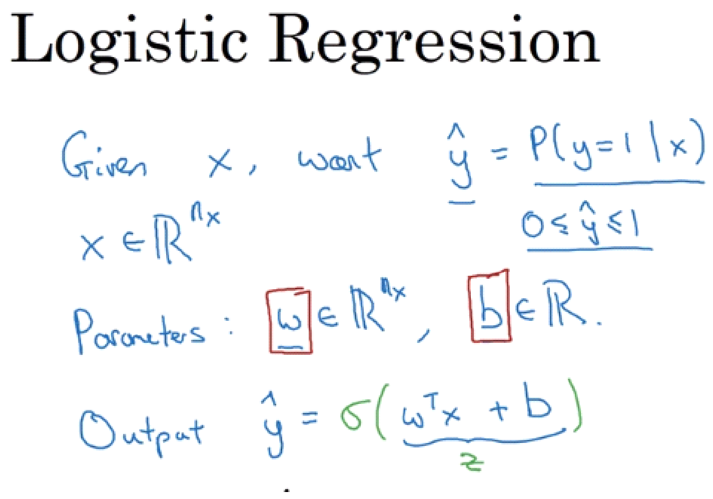


$$Y = [y^{(1)}\ y^{(2)} \dots , y^{(m)}]$$

$$Y \in \mathbb{R}^{1 \times m}$$

$$Y.\text{shape} = (1, m)$$

Logistic Regression:



Logistic Regression

Given x, want $\hat{y} = P(y=1 \mid x)$

$0 \leq \hat{y} \leq 1$

$x \in \mathbb{R}^{n_x}$

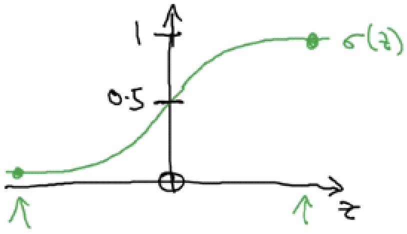
Parameters : $\boxed{w} \in \mathbb{R}^{n_x}$, $\boxed{b} \in \mathbb{R}$.

Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$

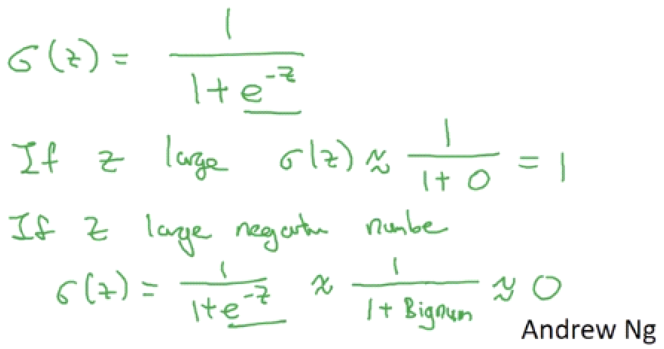
Here we have two parameter of the equation w and b

W, an n_x dimensional vector, and b, a real number.

Instead of using plain linear regression we will use below sigmoid curve



Below is the equation of sigmoid curve



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

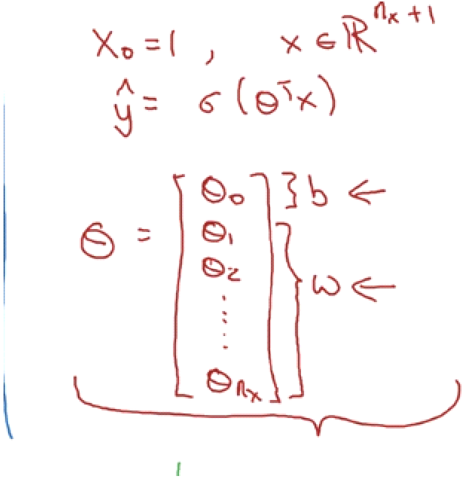
If z large $\sigma(z) \approx \frac{1}{1+0} = 1$

If z large negative number

$$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{Bignum}} \approx 0$$

Andrew Ng

One thing, eariler in the Machine learning course we used to work on the b and w at the same time using an x0. but here we will not



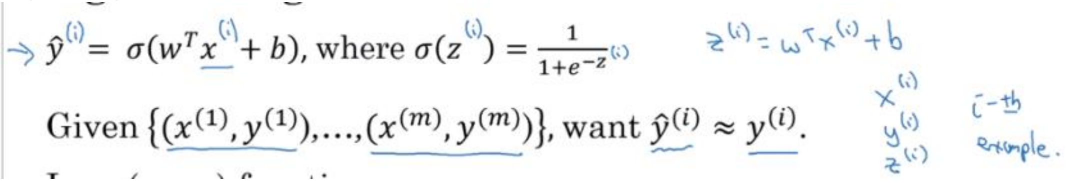
$X_0 = 1 \quad , \quad x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$

$$\underbrace{\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \leftarrow \\ \left. \begin{matrix} \\ \\ \\ \\ \end{matrix} \right\} w \leftarrow \end{matrix}}$$

We will not do this
We will keep theta 0 and other theta separate

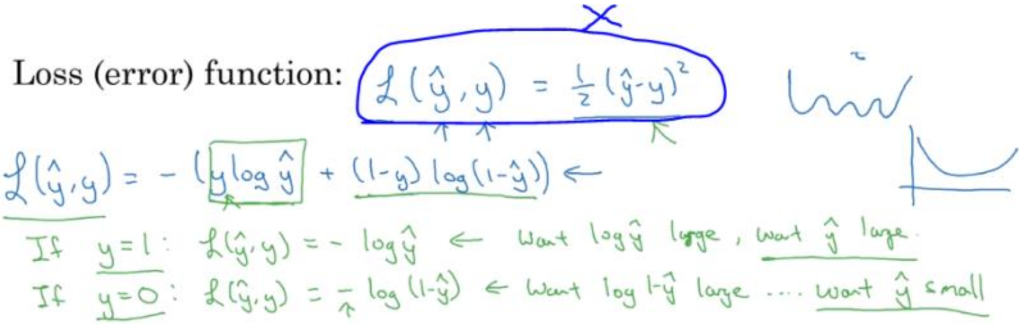
Logistic regression cost function:



$\rightarrow \hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ $\quad z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$. $\quad x^{(i)}$ $y^{(i)}$ $z^{(i)}$ i-th example.

Here we have i as the number of example



Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

$\mathcal{L}(\hat{y}, y) = -\left(\boxed{y \log \hat{y}} + (1-y)\log(1-\hat{y})\right) \leftarrow$

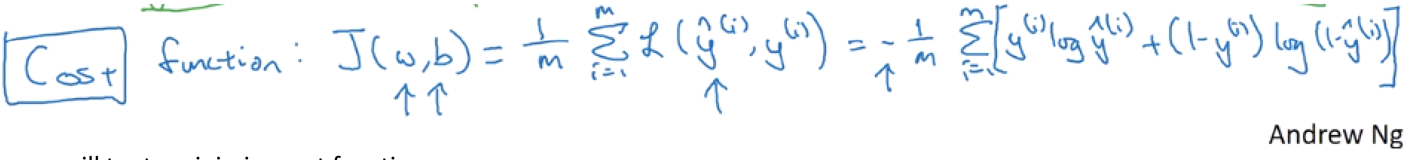
If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ Want $\log \hat{y}$ large, want $\hat{y}$ large.

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y}) \leftarrow$ Want $\log 1-\hat{y}$ large .... want $\hat{y}$ small

So we have a loss function which mean squared error. But in logistic regression it doesn't reach global optima. So for that we use the log cost function.

Also if y=1 then the loss function needs log of y hat to be large and hence y hat will be large
And vice versa for y-0

Now using this loss function we ill compute cost of the Logistic regression by applying this to all the examples. That will be cost function of logistic regression
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.



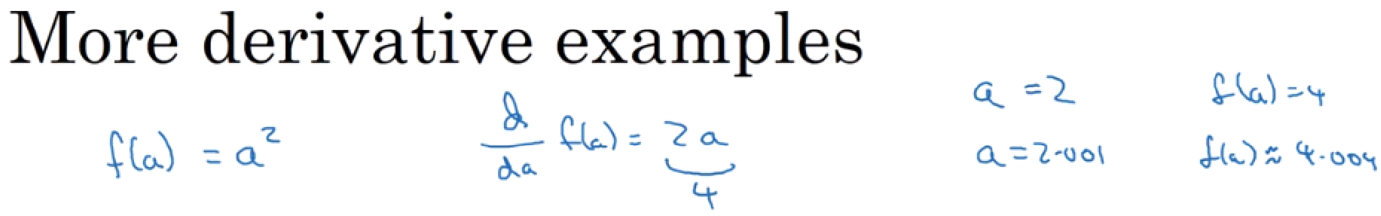
$\boxed{\text{Cost}}$ function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log \hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$

Andrew Ng

So we will try to minimize cost function

For gradient descent we usually initialize the values as zero. As the cost function is convex meaning it has only one global minima we don't need to use random intialization

Derivative:
It is the slope of the graph at any given point



More derivative examples

$f(a) = a^2$

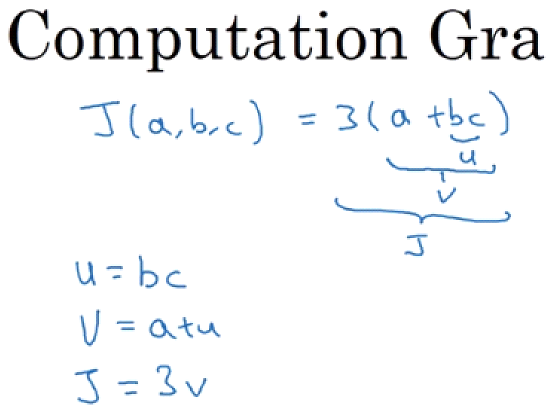
$\frac{d}{da} f(a) = \underbrace{2a}_{4}$

$a = 2$ $f(a) = 4$

$a = 2.001$ $f(a) \approx 4.004$

Derivative d/da(f(a) here is that when we change the value of "a" a littile bit, how will it affect f(a)
Here we check that when I increase a by 0.001, what change is there in f(a).
So f(a) became 4.004 so the increase 4 time in f(a) when a is increased. So derivative is 4.
Computation Graph:

Lets consider a simple example:



Computation Graph

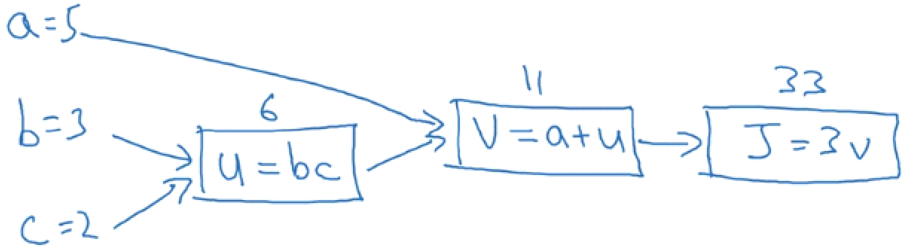
$J(a,b,c) = \underbrace{3(\underbrace{a+\underbrace{bc}_{u}}_{v})}_{J}$

$u = bc$

$v = a+u$

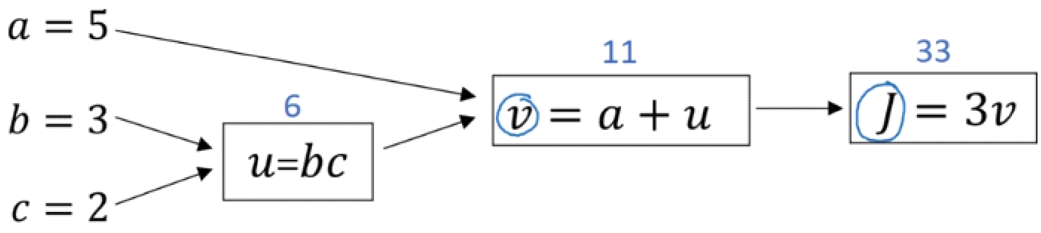
$J = 3v$

So u v and ja re the three steps



And above is the graph for that
This is a left to right pass, now there is also right to left pass which is more natural for computing derivative

Derivatives with a Computation Graph



$$\frac{dJ}{dv} = ?$$

Now we same computation graph

$$\frac{dJ}{dv} = ?$$

The above is that if we change the value of v a little, how will it affect value of J

$$J = 3v$$
$$v = 11 \rightarrow 11.\underline{001}$$
$$J = 33 \rightarrow 33.\underline{003}$$

$$\frac{dJ}{da} =$$

$$a = 5 \rightarrow 5.\underline{001}$$
$$v = 11 \rightarrow 11.001$$
$$J = 33 \rightarrow 33.\underline{003}$$

So we get the derivative as 3.

$$\frac{dJ}{da} = 3 = \frac{dJ}{dv}\frac{dv}{da}$$

Now the thing is, derivate of j with respect to a is equal to product of
Derivative of J wrt v and derivative of v wrt a

$$a \rightarrow v \rightarrow J$$

So this is the chain rule in calculus

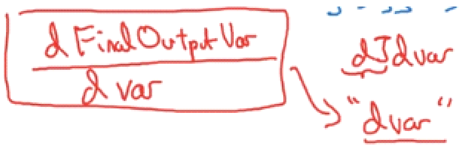
$$\boxed{\frac{dJ}{dv} = ? = 3}$$

$$\underline{\frac{dJ}{da}} = 3 = \frac{dJ}{dv}\frac{dv}{da}$$

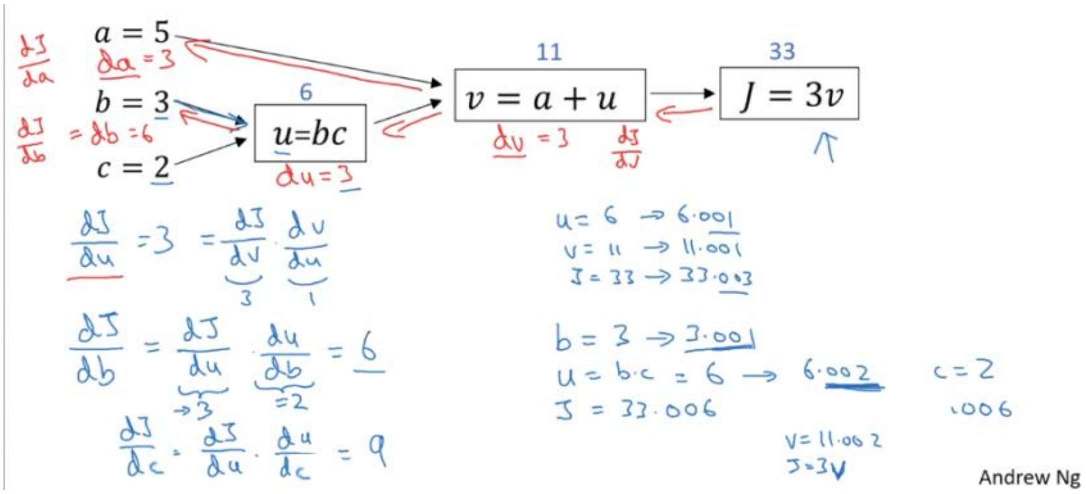
$$3 \times 1$$

$$\boxed{\frac{dv}{da} = 1}$$

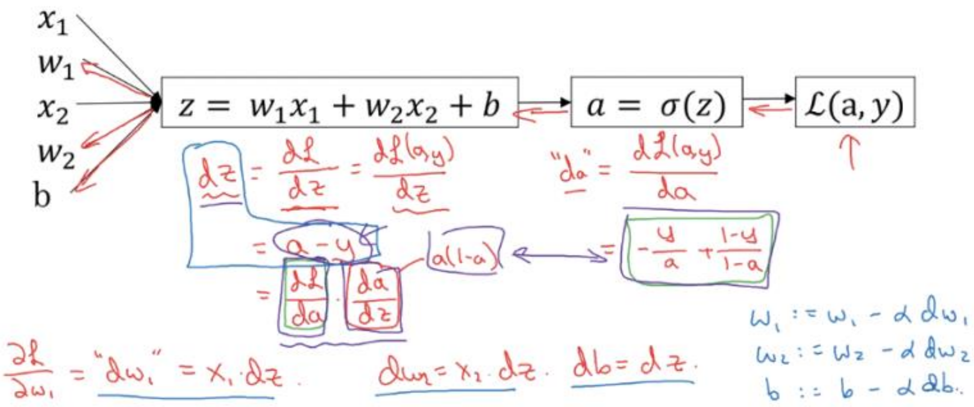
So here we get the derivative of final varible wrt to other variable



So we will in python just use the dvar to represent derivative of final output variable wrt to var
'dvar=The derivative of a final output variable with respect to various intermediate quantities.



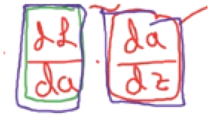
So for computing derivative it is efficient to move from right to left



Now here we will use derivative for the logistic reg and gradient descent.
Above is the equation.

So we can calculate dz and da using calculus

da =-(y/a)+(1-y)/(1-a)
dz=a-y
So the above dz is multiplication of



Where da/dz is a(1-a)
And we have da above
Simplifying we get dz as a-y

So the final dw1=x1.dz
Dw2=x2.dz
Db=dz

The above derivative computation is for just one example.
Lets now consider for m examples

```
$J=0$ ; $dw_1=0$ ; $dw_2=0$ ; $db=0$
For $i=1$ to $m$
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$
    $dz^{(i)} = a^{(i)} - y^{(i)}$
    $dw_1 += x_1^{(i)} dz^{(i)}$          $\updownarrow n=2$
    $dw_2 += x_2^{(i)} dz^{(i)}$
    $db += dz^{(i)}$
$J /= m$
$dw_1 /= m$ ; $dw_2 /= m$ ; $db /= m$.
```

$$w_1 := w_1 - \alpha \underline{dw_1}$$

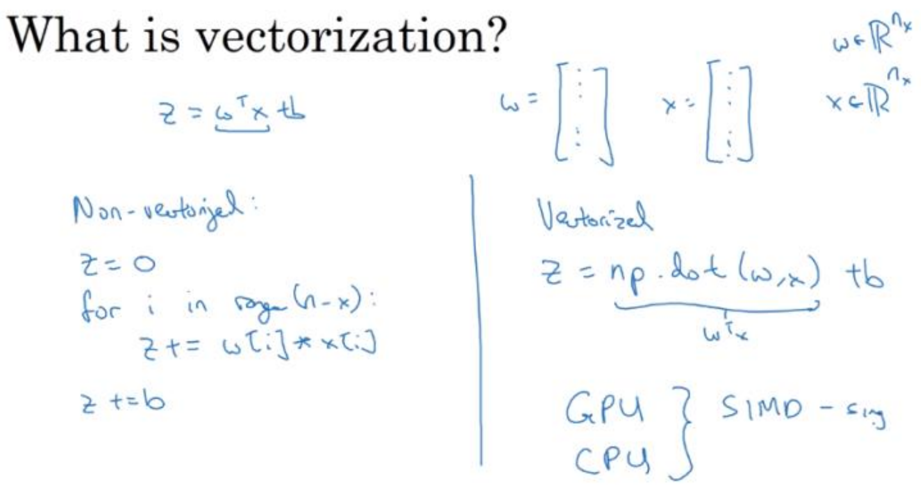
$$w_2 := w_2 - \alpha \underline{dw_2}$$

$$b := b - \alpha \, db.$$

Weakness with this is the for loop
Before deeplearning it was not a major issue with **vectorization** of the above code is not done. But now we need to improve the performance.

So here in the code we have two for loop, one is visible another one is when updating the w parameter. Here we just have two, but in real life there are many.

Vectorization



What is vectorization?

$z = \underbrace{w^T x} + b$

$w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $w \in \mathbb{R}^{n_x}$ $x \in \mathbb{R}^{n_x}$

Non-vectorized:

```
z = 0
for i in range(n-x):
    z += w[i] * x[i]
z += b
```

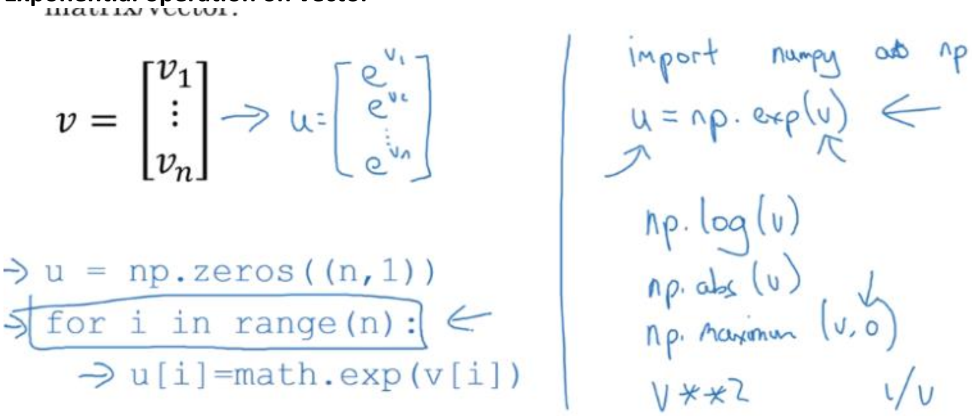
Vectorized

$z = \underbrace{np.dot(w,x)}_{w^Tx} + b$

GPU } SIMD - sing
CPU }

So for the two np array w and x we just need to use np.dot
Np.dot can be used for matrix as well

Exponential operation on vector

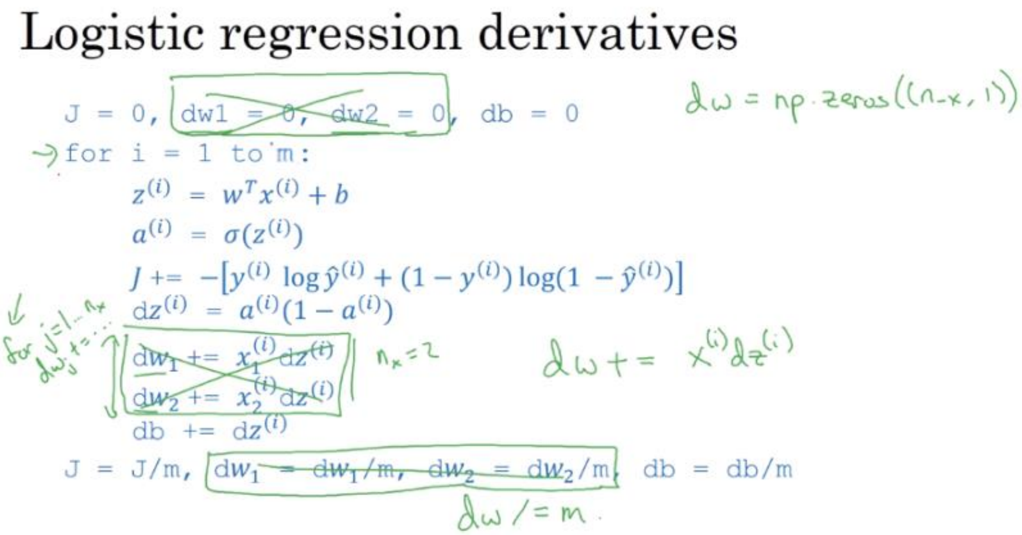


matrix/vector:

$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$

```
u = np.zeros((n,1))
for i in range(n):
    u[i]=math.exp(v[i])
```

```
import numpy as np
u = np.exp(v)
np.log(v)
np.abs(v)
np.maximum(v,0)
v**2        1/v
```



Logistic regression derivatives

```
J = 0, ~~dw1 = 0, dw2 = 0~~, db = 0          dw = np.zeros((n-x, 1))
for i = 1 to m:
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J += -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$
    $dz^{(i)} = a^{(i)}(1 - a^{(i)})$
    ~~$dw_1 += x_1^{(i)} dz^{(i)}$~~   $n_x = 2$      $dw += x^{(i)} dz^{(i)}$
    ~~$dw_2 += x_2^{(i)} dz^{(i)}$~~
    $db += dz^{(i)}$
J = J/m, ~~$dw_1 = dw_1/m$, $dw_2 = dw_2/m$~~, db = db/m
         dw /= m.
```

for $j = 1 \ldots n_x$
$dw_j += \ldots$

Note: In the above picture, y hat is nothing but a
So in the above code we have eliminated one for loop, we first initialized dw as a zero array of length as x vector.
And then used

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

Further vectorization: below is for FORWARD PROPOGATION

$$\rightarrow \underline{z^{(1)}} = w^T x^{(1)} + b \qquad \underline{z^{(2)}} = w^T x^{(2)} + b \qquad \underline{z^{(3)}} = w^T x^{(3)} + b$$

$$\rightarrow \underline{a^{(1)}} = \sigma(z^{(1)}) \qquad \underline{a^{(2)}} = \sigma(z^{(2)}) \qquad \underline{a^{(3)}} = \sigma(z^{(3)})$$

So for each example we need to calculate the above.

Now we need to vectorize it:

Remember we stacked the example vector such that the colums will be representing each example and the rows will be each feature.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \qquad \underline{(n_x, m)} \quad \mathbb{R}^{n_x \times m}$$

This vector is of shape (nx,m) where nx is the number of feature and m is # of examples

$$[z^{(1)}\ z^{(2)} \cdots z^{(m)}] = \underline{w^T X} + \underbrace{[b\ b \cdots b]}_{1\times m}$$

Now our z vector is dot product of w transform and X , with summing the b vector which is just a 1xm matrix with each value as b

Now the product will be as below

$$\underset{w^T}{[\text{———}]} \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$[w^T x^{(1)} \qquad w^T x^{(2)} \qquad \cdots \ w^T x^{(m)}]$$

So w will be multiplied by each example of X
Then summing b we will have

$$[z^{(1)}\ z^{(2)} \cdots z^{(m)}] = \underline{w^T X} + \underbrace{[b\ b \cdots b]}_{1\times m} = [\underline{w^T x^{(1)} + b} \quad \underbrace{w^T x^{(2)} + b}_{1\times m} \ \cdots \ \underline{w^T x^{(m)} + b}]$$

So this is the python code where b is simply broadcasted to the vector

$$Z = \underline{\text{np.dot}(w.T, X)} + \underset{\mathbb{R}}{b}$$

Now **BACKWARD PROPOGATION:**

$$\underline{dz^{(1)} = a^{(1)} - y^{(1)}} \qquad dz^{(2)} = a^{(2)} - y^{(2)}$$

$$dZ = [\underline{dz^{(1)}}\ \underset{1\times m}{dz^{(2)}} \dots dz^{(m)}]$$

$$A = [a^{(1)} \dots a^{(m)}]. \qquad Y = [y^{(1)} \dots y^{(m)}]$$

$$dZ = A - Y = [\underline{a^{(1)}-y^{(1)}} \quad a^{(2)}-y^{(2)} \quad \dots]$$

So earlier we vectorized calculating dw and db
So what we did earlier was instead of looping over all the w parameter we used vector form of dw

Now we will further simplify it

$$db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)}$$
$$= \frac{1}{m}\ \underline{np.sum}(dZ)$$

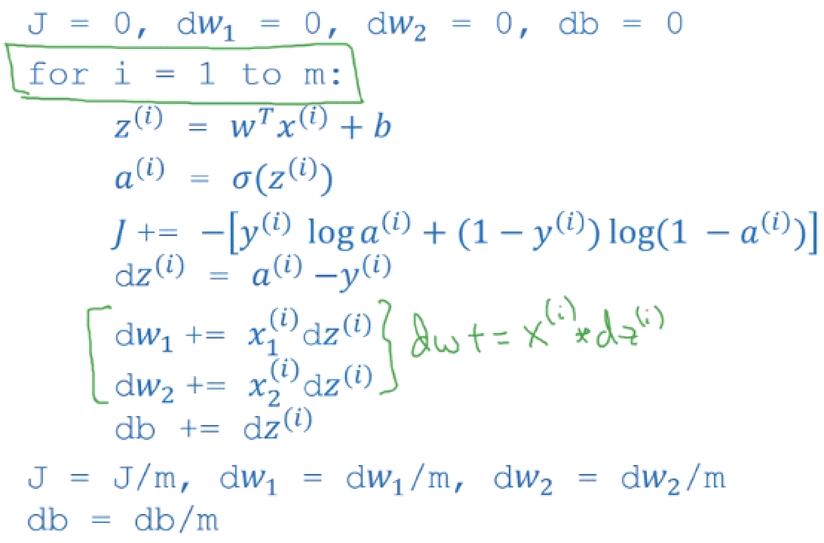
Db is sum of dx divided by m

For dw

$$dw = \frac{1}{m} X\, dZ^T$$
$$= \frac{1}{m}\underline{\begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix}} \begin{bmatrix} dz^{(1)} \\ \vdots \\ \underline{dz^{(m)}} \end{bmatrix}$$
$$= \frac{1}{m}\underset{n\times 1}{[\underline{x^{(1)}dz^{(1)}} + \dots + \underline{x^{(m)}dz^{(m)}}]}$$

I had doubt here, how it happened, Solved it in copy

OLD representation:



```
J = 0, dw_1 = 0, dw_2 = 0, db = 0
for i = 1 to m:
        $z^{(i)} = w^T x^{(i)} + b$
        $a^{(i)} = \sigma(z^{(i)})$
        $J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$
        $dz^{(i)} = a^{(i)} - y^{(i)}$
        $dw_1 += x_1^{(i)} dz^{(i)}$        $dw += x^{(i)} * dz^{(i)}$
        $dw_2 += x_2^{(i)} dz^{(i)}$
        $db += dz^{(i)}$
J = J/m, dw_1 = dw_1/m, dw_2 = dw_2/m
db = db/m
```

New form

$Z = w^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

$w := w - \alpha dw$

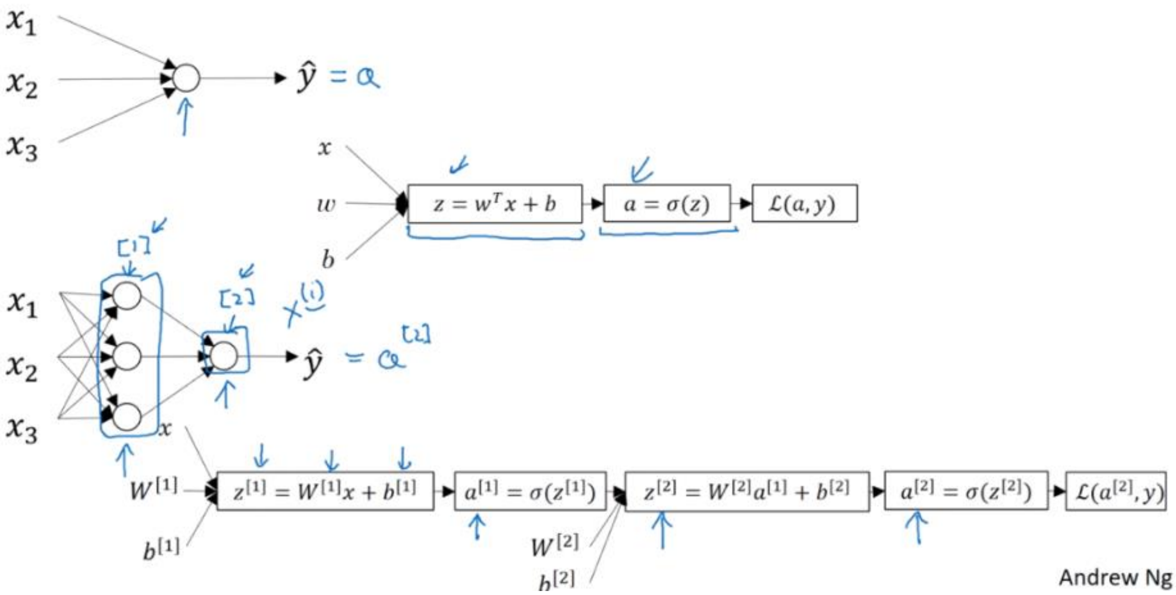
$b := b - \alpha db$

Broadcasting in Python

Week 3

Sunday, December 23, 2018 12:37 AM

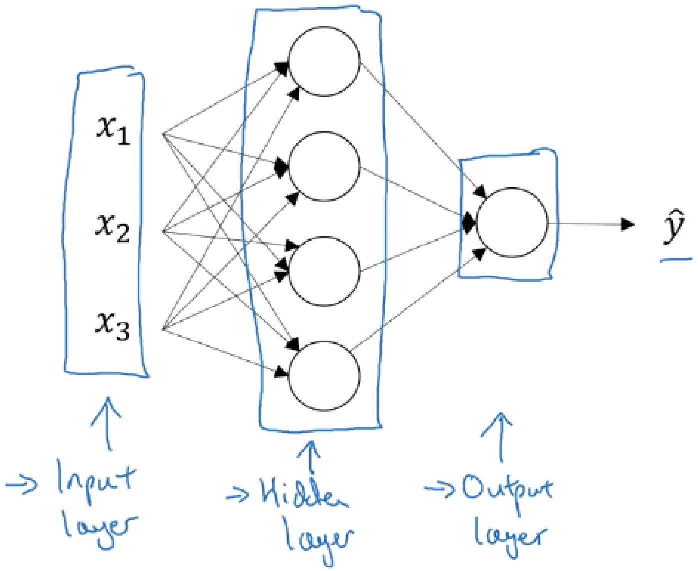
Overview:
So now each node will calculate z and a



So now we will have multiple z and a
And note that we have a square bracket here which represent node layer

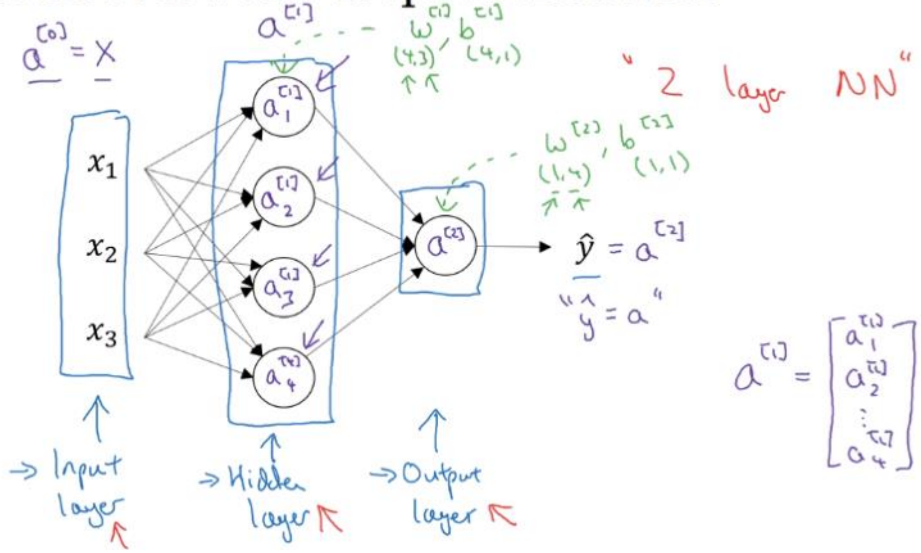
Also same derivative will be calculated from right to left

We will start with single hidden layer:



So here the **a** that we used earlier to represent the output is now called activation function

Neural Network Representation



Now the above fig has below items:

1. Now we are using superscript to show the layer number
2. Input layer is 0
3. The above is a 2 layer NN, input layer is not counted

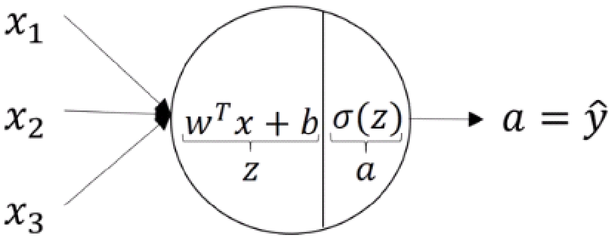
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix}$$

This is the vector from layer 1 of NN

Each layer has a matrix of **w** and a vector of b

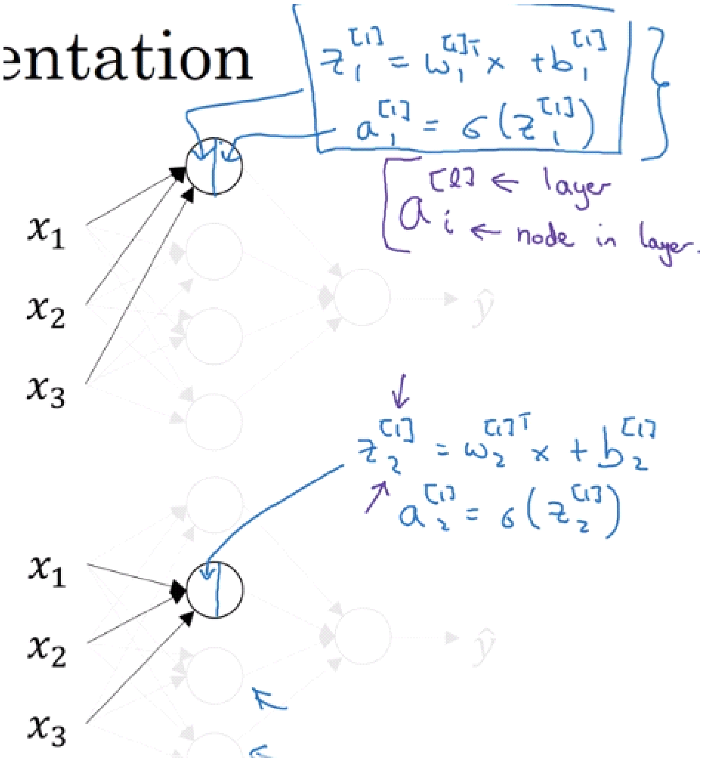
$$\underset{(4,3)}{w^{[1]}}, \underset{(4,1)}{b^{[1]}}$$

You can say each node is a logistic regression

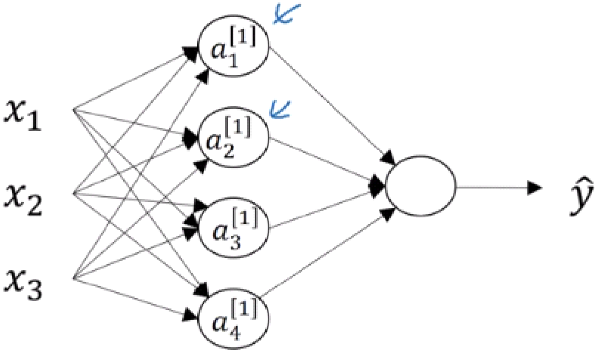


$$z = w^T x + b$$

$$a = \sigma(z)$$


entation
$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$
$a_1^{[1]} = \sigma(z_1^{[1]})$
$a_i^{[l]}$ ← layer
← node in layer.
$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$
$a_2^{[1]} = \sigma(z_2^{[1]})$
x_1
x_2
x_3
$\hat{y}$

So each layer is represented in superscript and node in that layer in subscript



$$z_1^{[1]} = \underline{w_1^{[1]T}}\, x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T}\, x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T}\, x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T}\, x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

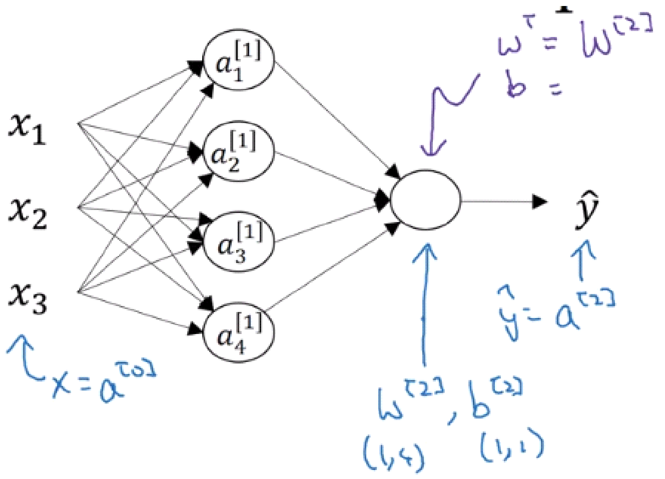
Now we will vectorize Z

$$\underset{(4,3)}{\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

So above we have parameter vector stacked horizontally

In the below we have whole equation for Z

$$z^{[1]} = \underset{(4,3)}{\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} \rightarrow w_1^{[1]T} x + b_1^{[1]} \\ \rightarrow w_2^{[1]T} x + b_2^{[1]} \\ \rightarrow w_3^{[1]T} x + b_3^{[1]} \\ \rightarrow w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$



Instead of x, we will now say a[0]

Given input x:

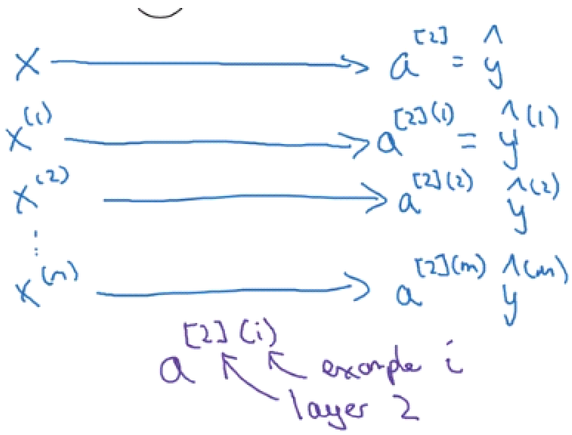
$$\rightarrow \underset{(4,1)}{z^{[1]}} = \underset{(4,3)}{W^{[1]}} \underset{(3,1)}{\overset{a^{[0]}}{\cancel{x}}} + \underset{(4,1)}{b^{[1]}}$$

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

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

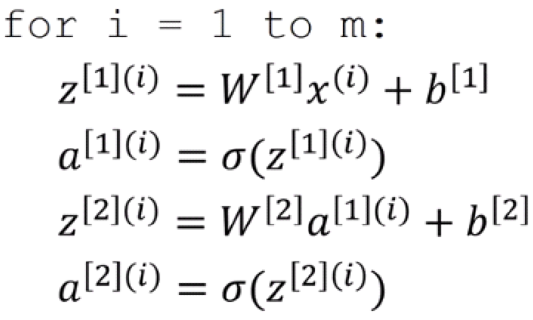
$$\rightarrow \underset{(1,1)}{a^{[2]}} = \sigma(\underset{(1,1)}{z^{[2]}})$$

So now we have vectorized the nodes of the NN,
Now we will look at **vectorization across multiple examples**


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

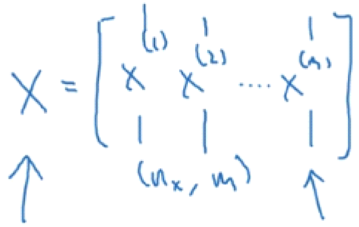
So now we have multiple X, and for each X we will have y which will have superscript in small bracket
And for the a, each example will have small bracket after the square bracket for node layer

This will be a for loop example

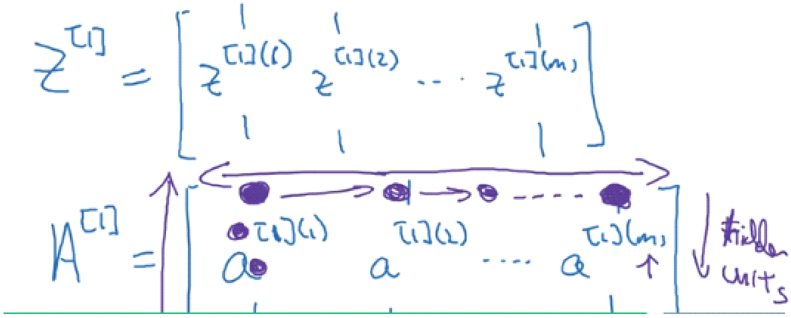


```
for i = 1 to m:
    $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
    $a^{[1](i)} = \sigma(z^{[1](i)})$
    $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
    $a^{[2](i)} = \sigma(z^{[2](i)})$
```

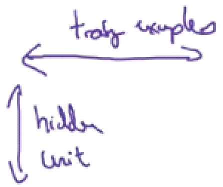
Now vectorized:


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

This is our training vector


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

This is Z and A
Where each column is an example and each row is for the hidden unit



$z^{[1](1)} = W^{[1]}\underline{x^{(1)}} + \cancelto{0}{b^{[1]}}$, $\underline{z^{[1](2)}} = W^{[1]}\underline{x^{(2)}} + \cancelto{0}{b^{[1]}}$, $\underline{z^{[1](3)}} = W^{[1]}\underline{x^{(3)}} + \cancelto{0}{b^{[1]}}$

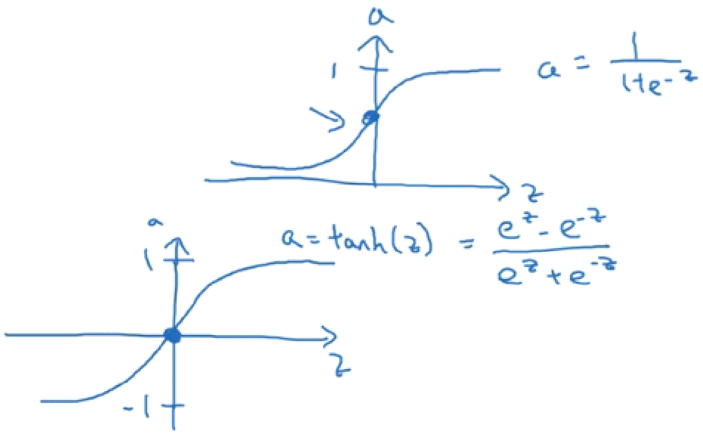
$W^{[1]}x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$ $W^{[1]}x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$ $W^{[1]}x^{(3)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$

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

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

We can also use other activation function instead of sigmoid

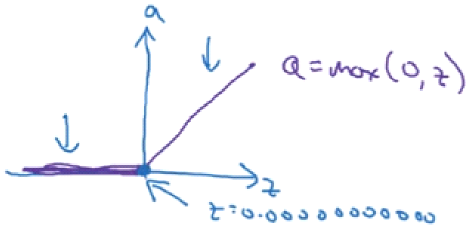
Like tanh.
So sigmoid is not used mostly and tanh is superior than sigmoid with one exception for the output layer, where sigmoid is preferred.



Above are the sigmoid and tanh
So tanh is in between 1 and -1, so that is why it is not preferred for the classification task. Where we need 0 or 1

tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.
As seen in lecture the output of the tanh is between -1 and 1, it thus centers the data which makes the learning simpler for the next layer.

So another issue is that when z is very large or small, it slows the gradient descent.
So instead tanh or sigmoid we can use ReLU,



For output sigmoid is used when classification. And for hidden now ReLU is preferred
Sigmoid outputs a value between 0 and 1 which makes it a very good choice for binary classification. You can classify as 0 if the output is less than 0.5 and classify as 1 if the output is more than 0.5.

If not sure try many activation function and see result

Why do we need non linear activation function:

Why not just use value of Z
So if you use linear activation function or simply Z, then the final is just a linear function of X and Y

$$\underbrace{a^{[1]} = z^{[1]}}\, = \underbrace{W^{[1]}x} + b^{[1]}$$

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

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

$$= \underbrace{(W^{[2]}W^{[1]})}_{W'}x + \underbrace{(W^{[2]}b^{[1]} + b^{[2]})}_{b'}$$

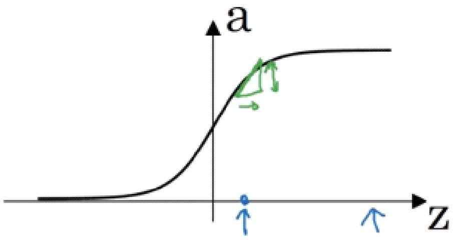
$$= W'x + b'$$

So in the end you just have a linear function.

Linear function can be used in output layer in case we have a regression problem. Or also ReLU when the regression problem don’t have any negative value

Derivatives of activation functions

Sigmoid:



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

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

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

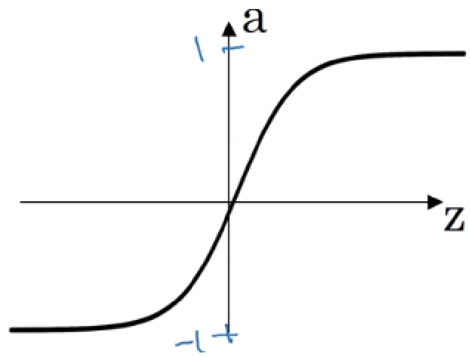
$$= \underline{g(z)\,(1-g(z))} \leftarrow$$

$$= \boxed{a\,(1-a)}$$

$$\Big|\, g'(z) = a(1-a)$$

g`(z) is the derivative

Tanh:

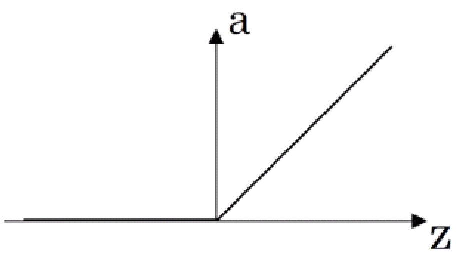


$$g(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = \frac{d}{dz} g(z) = \text{slope of } g(z) \text{ at } z$$
$$= \underline{1 - (\tanh(z))^2} \leftarrow$$

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

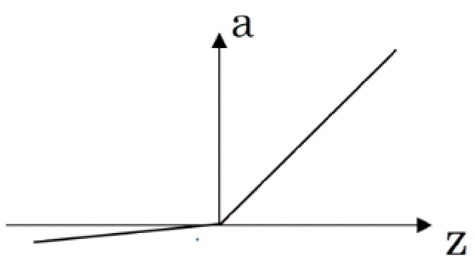
ReLU:



ReLU

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

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



Leaky ReLU

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

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

Gradient descent for neural networks

Parameters: $\underset{(n^{[1]}, n^{[0]})}{W^{[1]}}, \underset{(n^{[1]}, 1)}{b^{[1]}}, \underset{(n^{[2]}, n^{[1]})}{W^{[2]}}, \underset{(n^{[2]}, 1)}{b^{[2]}}$ $\quad n_x = n^{[0]}, n^{[1]}, \underline{n^{[2]} = 1}$

So above we have parameter and below that the shape of the matrix of those parameter.
So w[1] is of n[1],n[0]. That is the number of node in hidden layer, number of node in input layer

So w[2] is of n[2],n[1]. That is the number of node in output layer, number of node in hidden layer

So b[1] is of n[1],1. That is the number of node in hidden layer
So b[2] is of n[2],1. That is the number of node in output layer

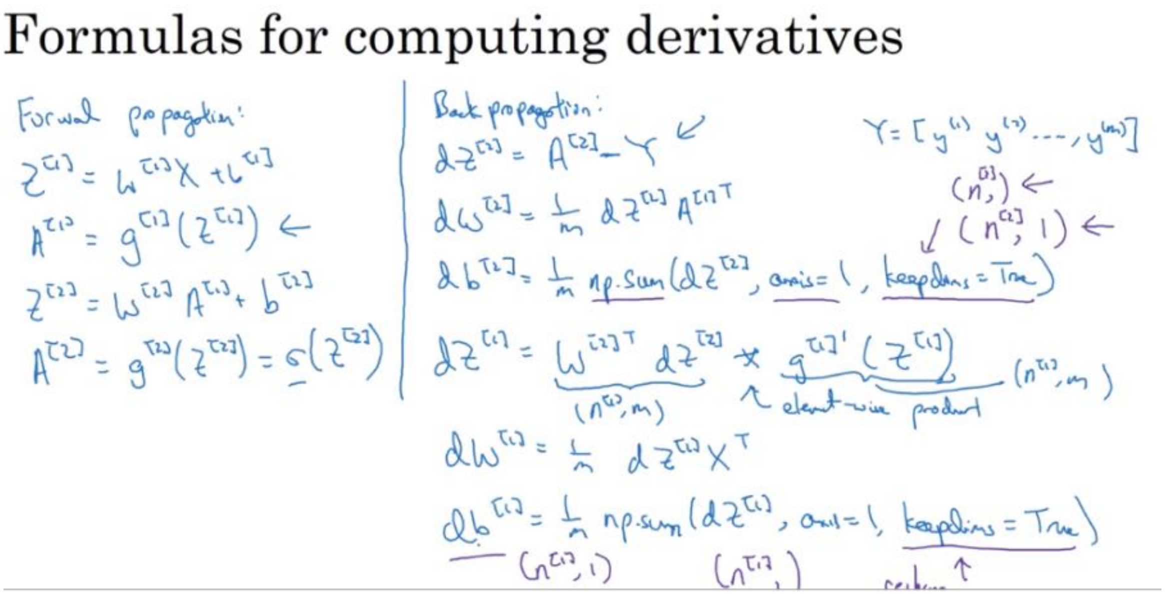
Cost function:

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

Gradient descent:

We should initialize parameter randomly

```
→ Repeat {
    → Compute predicts $(\hat{y}^{(i)}, i = 1 \ldots, m)$
      $dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}}, \ldots$
      $W^{[1]} := W^{[1]} - \alpha \, dW^{[1]}$
      $b^{[1]} := b^{[1]} - \alpha \, db^{[1]}$
```



Formulas for computing derivatives

Forward propagation:

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

$A^{[1]} = g^{[1]}(Z^{[1]})$ ←

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$

Back propagation:

$dZ^{[2]} = A^{[2]} - Y$ ↙ $Y = [y^{(1)}\ y^{(2)} \dots, y^{(m)}]$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$ $(n^{[2]},)$ ←

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

$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{}$ $(n^{[1]}, m)$

↑ element-wise product

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

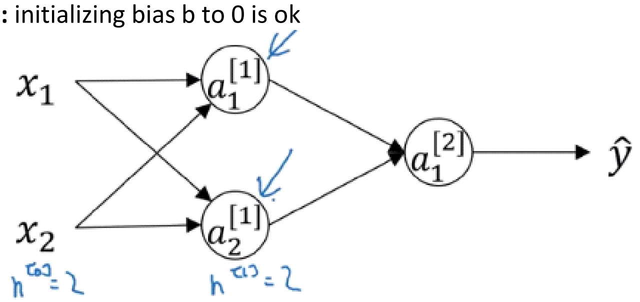
$db^{[1]} = \frac{1}{m}$ np.sum($dZ^{[1]}$, axis=1, keepdims=True)

$(n^{[1]}, 1)$ $(n^{[1]},)$ reshape ↑

Pending: Backpropagation intuition video.

Random Initialization:
In last week course when we initialized the parameter to zero for the logistic regression it worked fine. But not in the case of NN

Note: initializing bias b to 0 is ok



$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

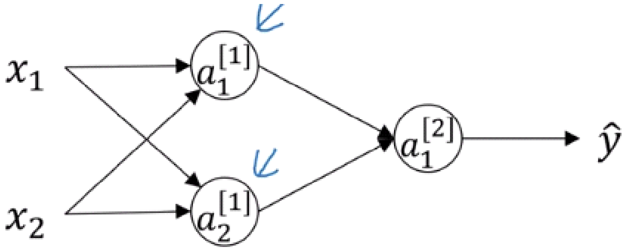
$a_1^{[1]} = a_2^{[1]}$

And also the derivative will be same

$dz_1^{[1]} = dz_2^{[1]}$

So this will create a symmetry in the nodes. Which will make the hidden units redundant .

So we need to initialize with random so that we don't have this symmetry:



$$\rightarrow w^{[1]} = \text{np.random.randn}((2,2)) * \underset{100?}{\underline{0.01}}$$

$$b^{[1]} = \text{np.zero}((2,1))$$

$$w^{[2]} = \ldots$$

$$b^{[2]} = 0$$

So we can use above initialization

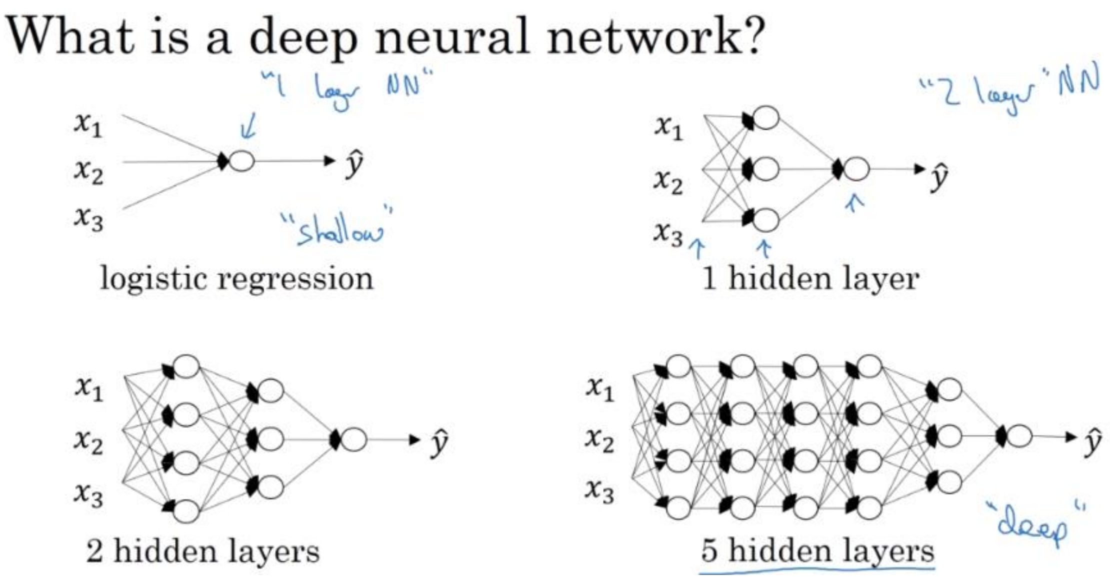
Now one thing why we multiplied the w with 0.01

So if w is large then we might end up with Z being large and if Z is large then it will take more time for the gradient descent to converge.

Week 4

Sunday, December 23, 2018 7:10 AM

What is a deep neural network



What is a deep neural network?

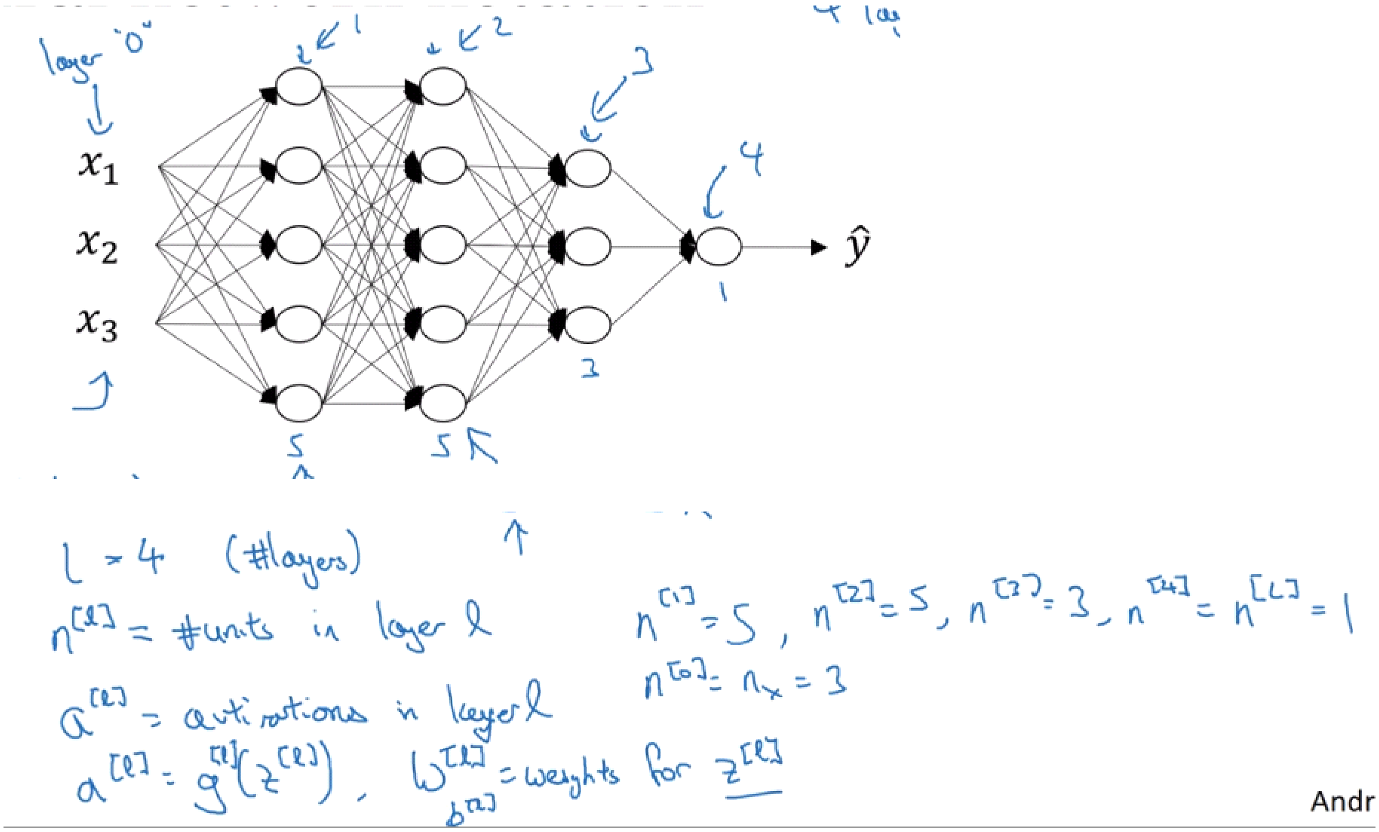
logistic regression

1 hidden layer

2 hidden layers

5 hidden layers

L to denote number of layer



$L = 4$ (#layers)

$n^{[l]}$ = #units in layer l

$n^{[1]} = 5,\ n^{[2]} = 5,\ n^{[3]} = 3,\ n^{[4]} = n^{[L]} = 1$

$n^{[0]} = n_x = 3$

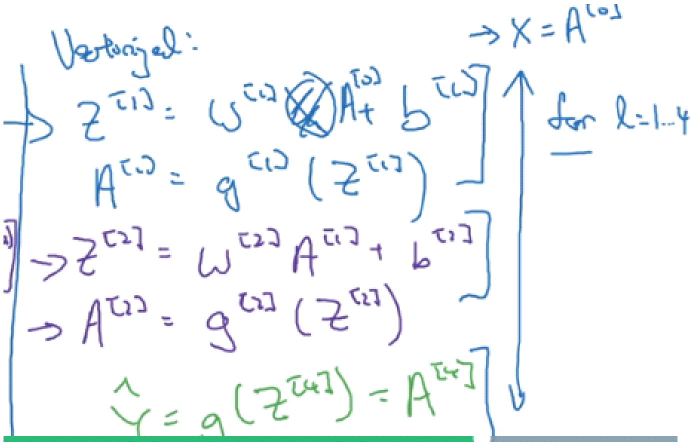
$a^{[l]}$ = activations in layer l

$a^{[l]} = g^{[l]}(z^{[l]})$, $W^{[l]}$ = weights for $z^{[l]}$

$b^{[l]}$

Andr

Forward propagation:



Vectorized:

$\rightarrow X = A^{[0]}$

$$\rightarrow Z^{[1]} = W^{[1]} \cancel{X} A^{[0]} + b^{[1]}$$

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

$$\rightarrow Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$\rightarrow A^{[2]} = g^{[2]}(Z^{[2]})$$

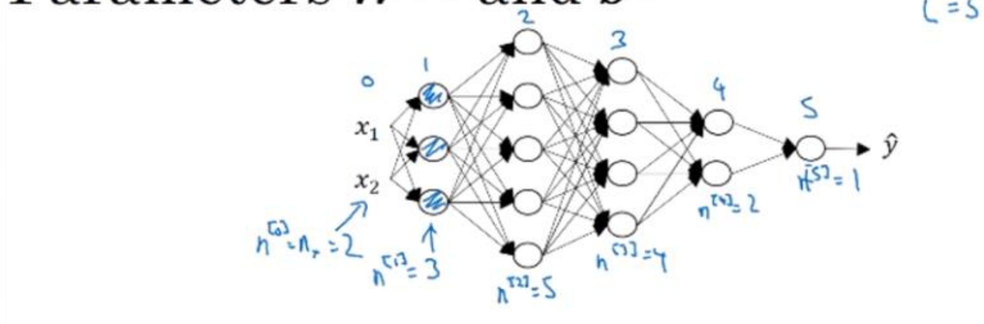
$$\hat{Y} = g(Z^{[4]}) = A^{[4]}$$

for $l = 1 \ldots 4$

So the above for loop cannot be avoided. This loop over all the layer in NN

Getting your matrix dimensions right

Parameters $W^{[l]}$ and $b^{[l]}$



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

$$W^{[l]} : (n^{[l]}, n^{[l-1]})$$

$$b^{[l]} : (n^{[l]}, 1)$$

dw and db will be same dimension as w and b

Z and A has same dimension
And dz and da will have same dimension as Z and A
Now for the m examples
Forward

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

$$(n^{[1]},1) \quad (n^{[1]},n^{[0]}) \quad (n^{[0]},1) \quad (n^{[1]},1)$$

$$\left[z^{[1](1)} \; z^{[1](2)} \; \cdots \; z^{[1](m)} \right]$$

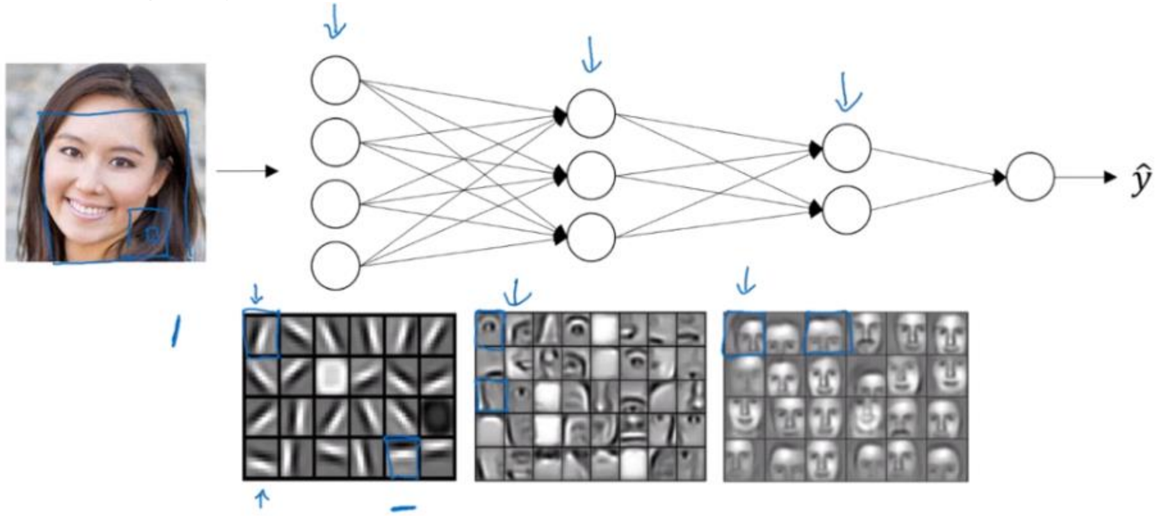
$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]}$$

$$(n^{[1]},m) \quad (n^{[1]},n^{[0]}) \quad (n^{[0]},m) \quad (n^{[1]},1)$$

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

Why deep representations?

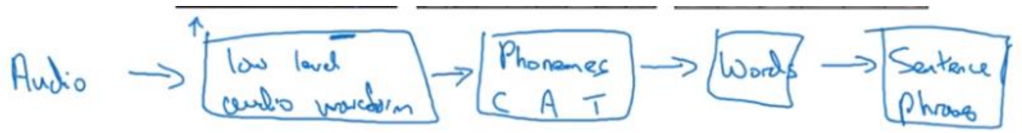
For the best performance the NN has to have many hidden layers:
So considering example of face detection



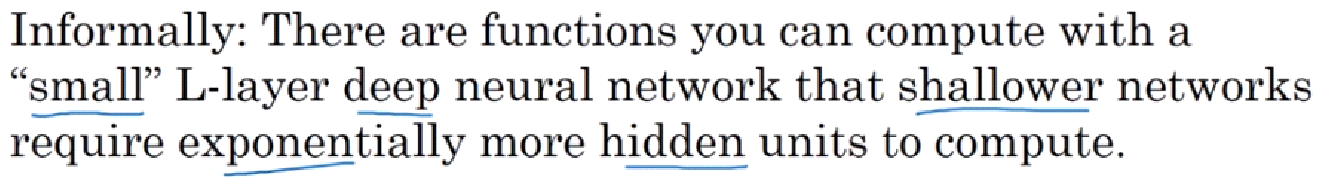
So the first layer will detect the edges in the image,
Second use the data of those egdes and make parts of face
Then last will be able to recognize the face

Note that the first layer will work on small part of image, second will on more area,
And third on the largest

It goes from simple to complex
For audio



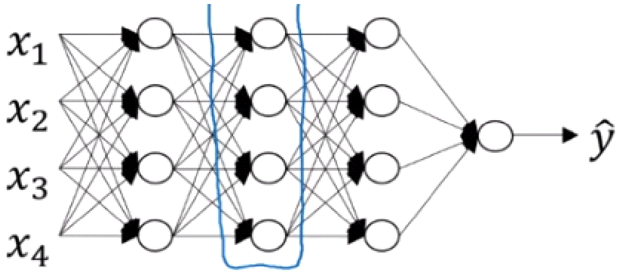
Another reason is:



Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

Building blocks of deep neural networks

Asd



Layer l: $W^{[l]}$, $b^{[l]}$

Forward: Input $a^{[l-1]}$, output $a^{[l]}$

$\underline{z^{[l]}} = W^{[l]} \underline{a^{[l-1]}} + b^{[l]}$ cache $\underline{z^{[l]}}$

$\underline{a^{[l]}} = g^{[l]}(z^{[l]})$

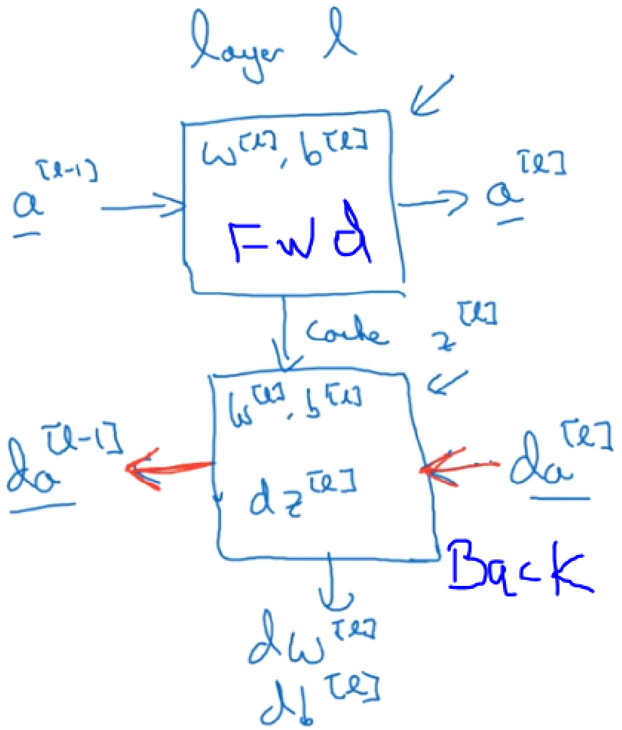
→ Backward: Input $da^{[l]}$, output $\underline{da^{[l-1]}}$

cache $(z^{[l]})$

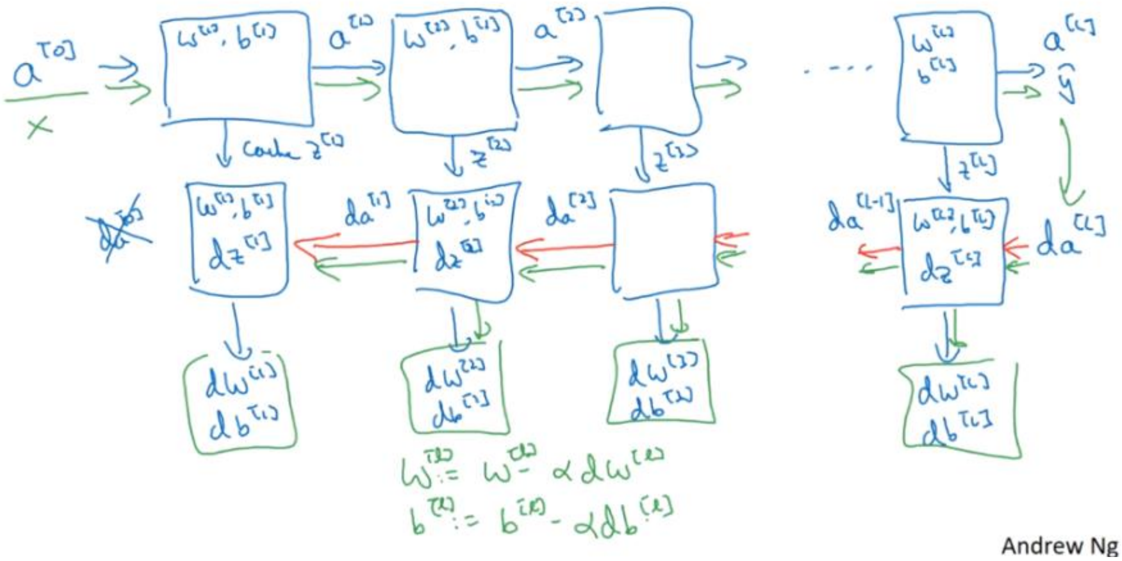
$dW^{[l]}$

$db^{[l]}$

So basically	



Andrew



Andrew Ng

Forward propagation for layer l

→ Input $a^{[l-1]}$ ←

→ Output $a^{[l]}$, cache $(z^{[l]})$ ↙ $W^{[l]}, b^{[l]}$

Vectorized:

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$

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

Backward propagation for layer l

→ Input $da^{[l]}$

→ Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$

$$\underline{dz^{[l]}} = \underline{da^{[l]}} * g^{[l]\prime}(z^{[l]})$$

$$dW^{[l]} = dz^{[l]} \cdot \underline{a^{[l-1]}}$$

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

$$\underline{da^{[l-1]}} = W^{[l]T} \cdot dz^{[l]}$$

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

Vectorized

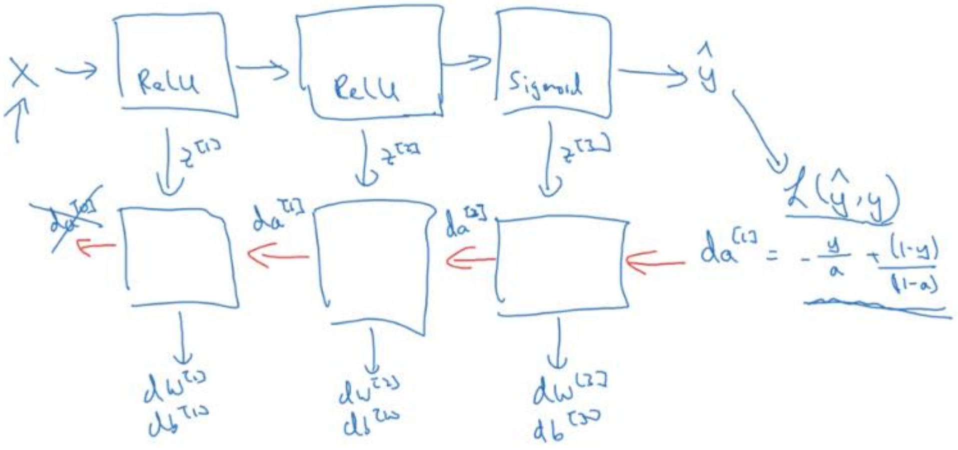
$$dZ^{[l]} = \boxed{dA^{[l]}} * g^{[l]\prime}(Z^{[l]})$$

$$\underline{dW}^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$

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

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

Flow chart



Parameter vs Hyperparameter

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}$ ...

Hyperparameters: learning rate α

#iterations

#hidden layer L

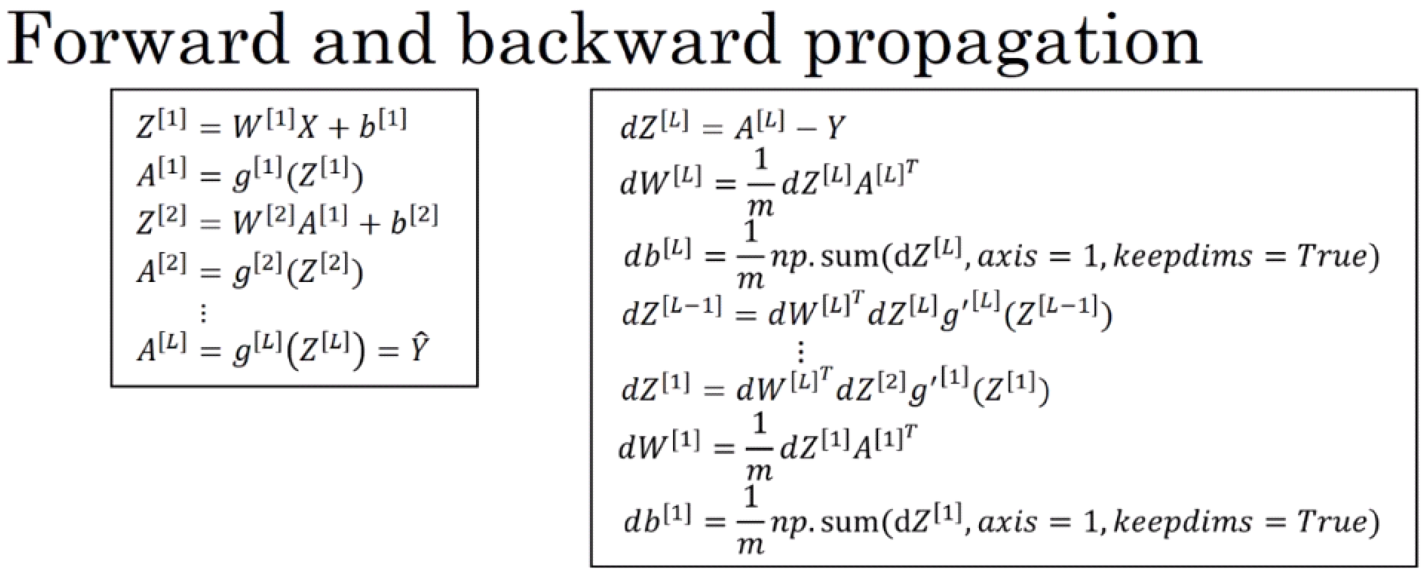
hidden units $n^{[1]}, n^{[2]}, \ldots$

choice of activation function

So the hyperparameter will determine how the final value of parameter will be

Applied deep learning is a very empirical process

Hyper parameter can change overtime



Forward and backward propagation

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) \\
&\vdots \\
A^{[L]} &= g^{[L]}\left(Z^{[L]}\right) = \hat{Y}
\end{aligned}
$$

$$
\begin{aligned}
dZ^{[L]} &= A^{[L]} - Y \\
dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]^T} \\
db^{[L]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{d}Z^{[L]}, axis = 1, keepdims = True) \\
dZ^{[L-1]} &= dW^{[L]^T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\
&\vdots \\
dZ^{[1]} &= dW^{[L]^T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]^T} \\
db^{[1]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{d}Z^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$