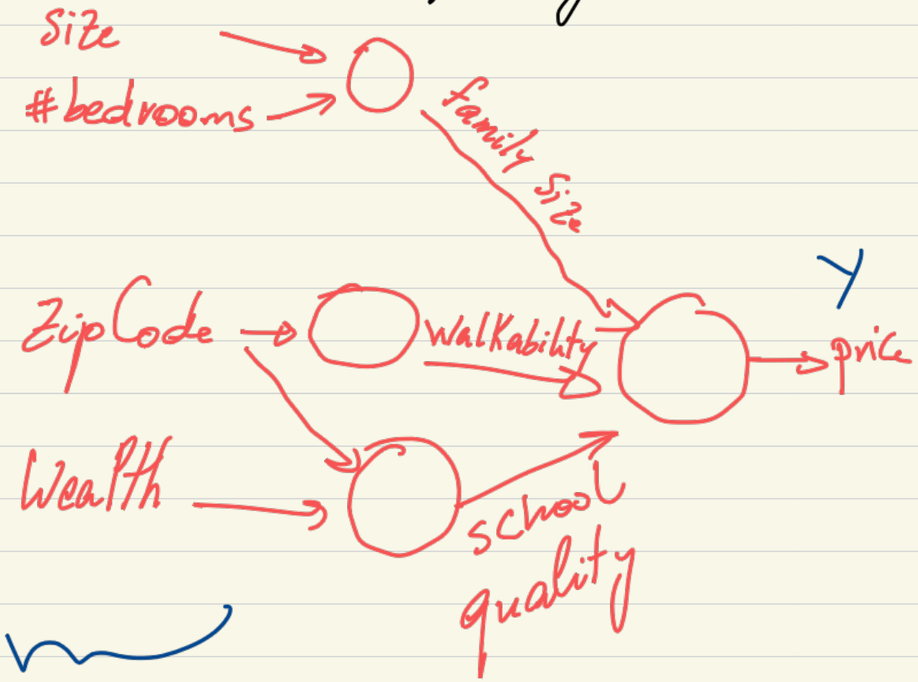
Deep learning specialisation by Andrew Ng on Coursera

Notes by Mina Rafla

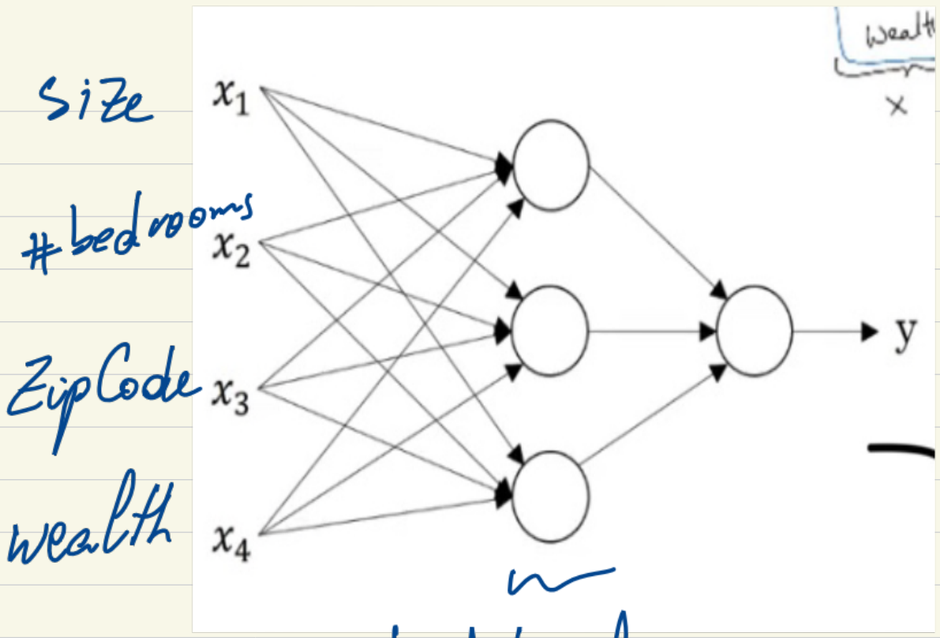
Course 1

Week 1

Let's say instead of predicting the price of a house only from the size, we can other features as # bedrooms. These 2 features tell us if the house fit the family size. The Zip Code can tell us the walkability. And may be size & wealth tell us the school quality



We just need to give 4 inputs & all the things in the middle is automatically generated.



It densily connected since each i/p feature is connected to each neuron in the hidden layer.

the NN decides the weights linking between each i/p feature & each hidden unit.
Giving enough data, NN can figure out function that accurately map X & Y.

In supervised learning, we have Input X & tries to estimate the price Y.

Many applications

- Online advertising to estimate ad clicks
- Photo tagging
- Speech Recognition
- Machine Translation
- Autonomous driving

Different types of data

- Structured Data (like tabular data)
- Unstructured Data → Audio, Images, Text

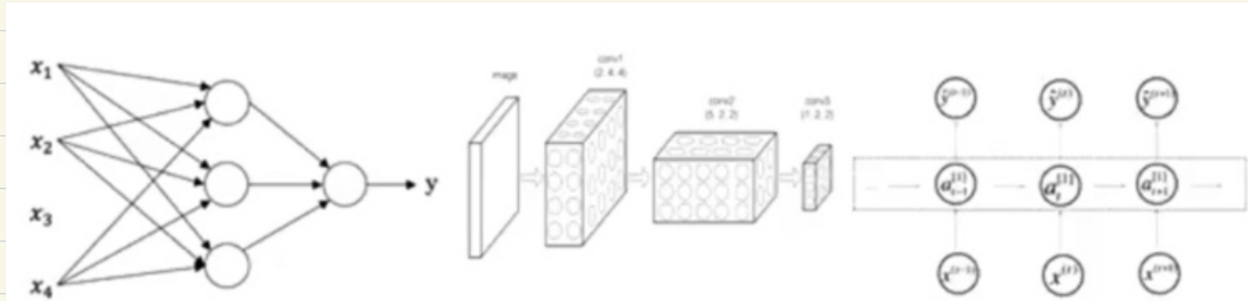
Depending on the application, a different type of NN can be used.

Photo tagging → CNN (Convolutional NN)

Speech Recognition → is a sequence data so RNN are used.

Machine Translation

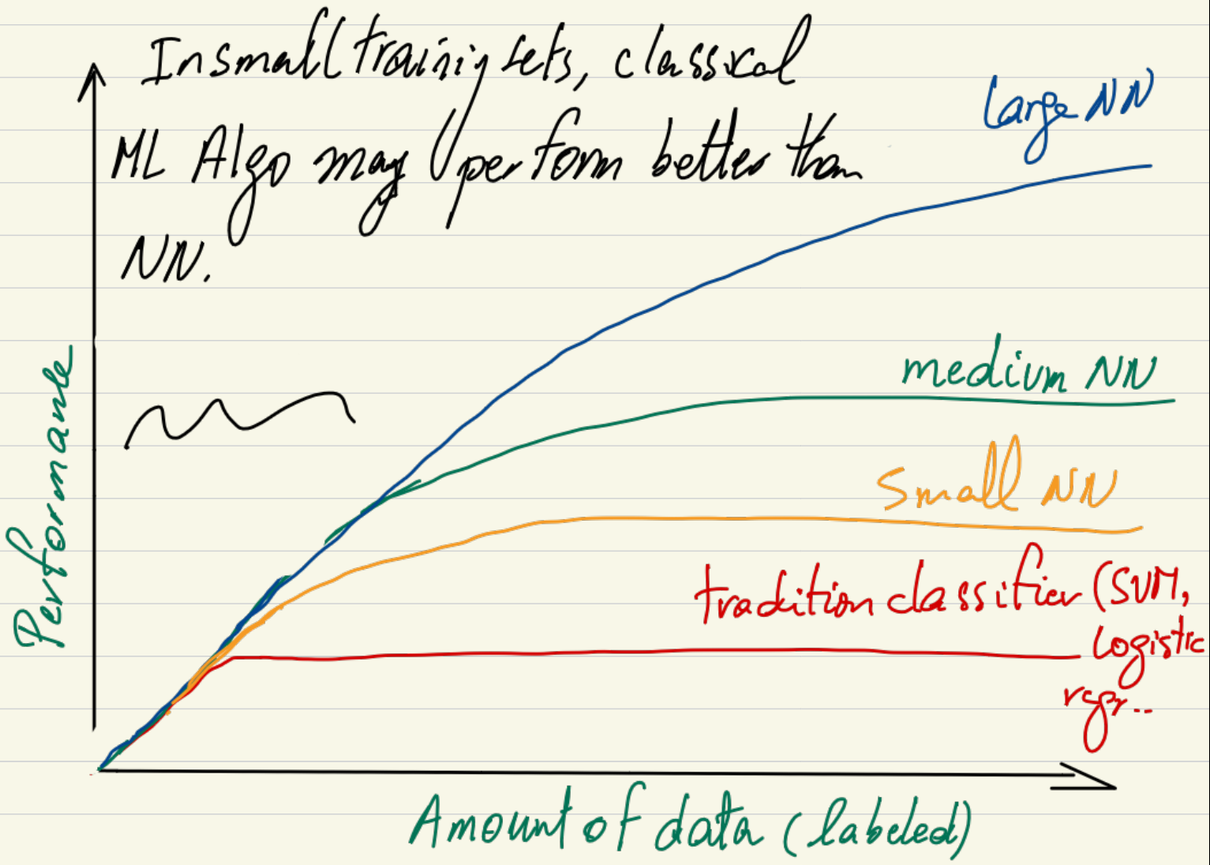
Autonomous driving → Custom or Hybrid NN



Standard NN | Convolutional NN | Recurrent NN

Why is deep learning taking off

Scale drives deep learning progress



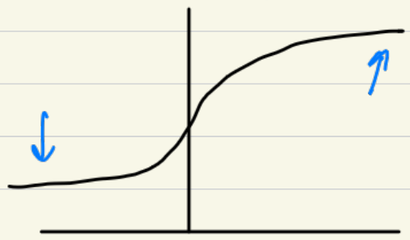
* We now have a fairly large amount of data!! because of digitilization

Scale drive Neural Network

- Data size
- Algorithms ↔ Computation (like GPUs & special Hardwares)
  - for example: size of the NN (hidden layes, hidden units)

for example:

When using Sigmoid function the learning may be very slow



Since the slope of the Sigmoid in the marked regions is nearly zero so the gradient becomes very small & the learning (with the gradient descent) becomes very slow

Unlike the ReLu function that made the gradient descent much faster. 

Course 1

Week 2

Logistic regression as a neural network
Basic Classification

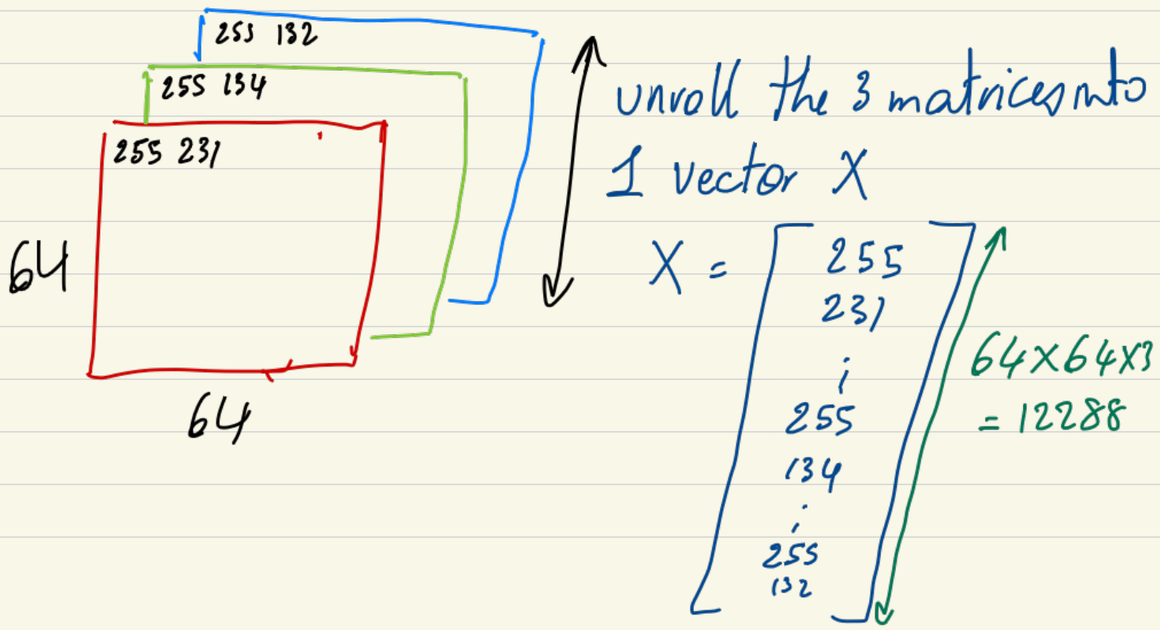
Logistic Regression is an algo for binary classification

64 → 1 (Cat)
0 (non cat)

64 x 64

How to present an i/p image in a computer?

↳ Three matrices → Red / Green / Blue



Since the image is 64 x 64 pixels so the i/p X will have a size $n_x = 64 \times 64 \times 3 = 12288$

$n = n_x = 12288$ → the size of the feature vector

So the goal is to learn a classifier that has as i/p our feature vector X & outputs Y (1 or 0) ⇒ Binary classification

Notation for the rest of the Course:

a single training example is presented by a pair

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

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

$M = M_{train}$ # of train examples M_{Test} = # of test examples

$X \in \mathbb{R}^{n_x \times m}$

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

Examples are stacked in columns

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

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

Logistic regression

Given an i/p vector $\underline{X}$ (for example an image)

We want to estimate $\hat{y} = P(Y=1/X)$ ↳ What is the chance that x is a cat picture ??

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

Parameters of a logistic regression

↳ $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output $\hat{y} = w^T x + b$ ← One solution that doesn't work as in linear Regression

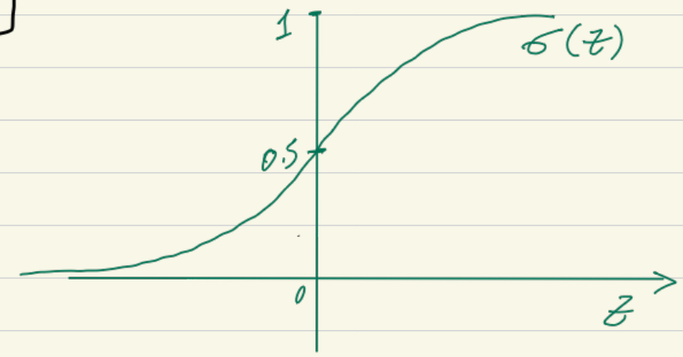
I want $\hat{y}$ to be between 0 & 1

In logistic Regression

$\hat{y} = \sigma(w^T x + b)$ → sigmoid function

Logistic Regression is used for classification

Sigmoid function



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

If z very large:

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

If z is very Small

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

When implementig Logistic Regression we try to find

$z = w^T x + b$ // So we try to find params w & b

Logistic regression cost function

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

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

We have m training examples

Of Course $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$

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

Notations here

Loss (Error) function:

↳ One thing I could do, Squared Error (bad idea)

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

In logistic Regression we don't do this since the optimization Problem becomes non-Convex, so we can fall into local optimum

↓

← Loss fn

In logistic Regression we use other loss function

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

Intuition:

We want the loss as Small as possible

If $y = 1$ ⇒ $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ⇒ So we want $\log \hat{y}$ to be large so $\hat{y}$ large (=1)

If $y = 0$ ⇒ $\mathcal{L}(\hat{y}, \hat{y}) = -\log(1-\hat{y})$

So we want $\log(1-\hat{y})$ large

so $\hat{y}$ we want it Small as possible

(=0)

Note that $0 \le \hat{y} \le 1$

Cost function: to show how the algo is doing on the entire dataset $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Expanding the Cost Function

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

$$-\frac{1}{m}\sum y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log 1-\hat{y}^{(i)}$$

the average of all loss functions for all the examples.

Loss function is applied on a single training example

Cost function is the cost of the parameters

Logistic Regression is a very Small Neural Network

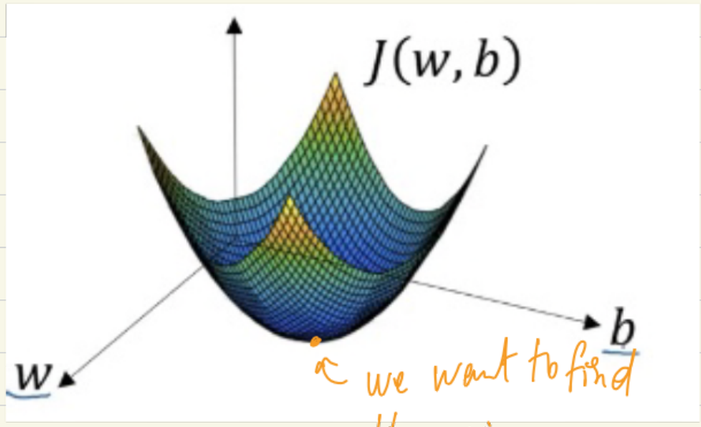
Gradient descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

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

$$-\frac{1}{m}\sum y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log 1-\hat{y}^{(i)}$$

We want to find w, b that minimizes $J(w,b)$



$J(w,b)$ is a surface

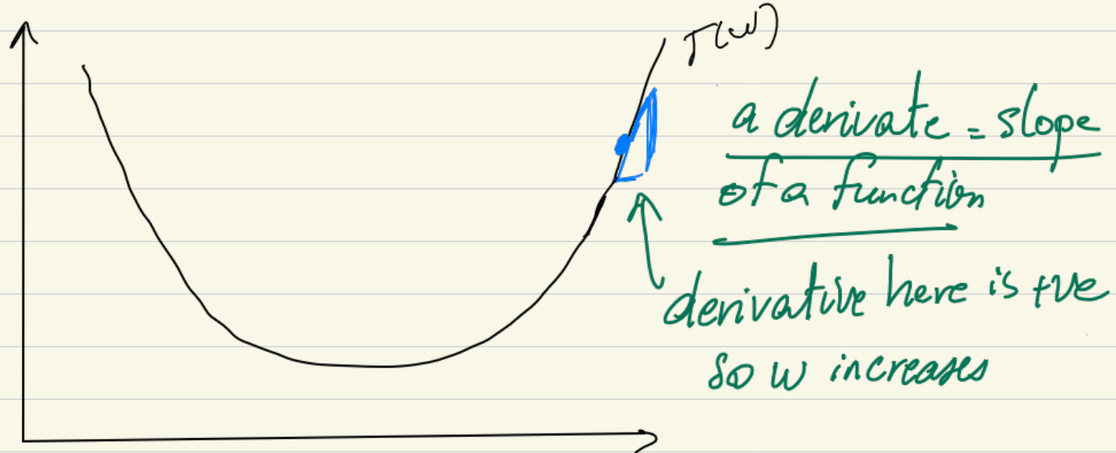
w can be multidimensional

we want to find this min

$J(w,b)$ is a convex function ∪ contrary to non-convex functions that contain several local optima

To find a good value for the params we initialize w, b to some initial value.

In Logistic Regression, no matter what the value initial is I should find the minimum cost function since it's a convex function



a derivate = slope of a function

derivative here is +ve so w increases

Repeat {

$$w := w - \alpha \frac{dJ(w)}{d(w)}$$

learning rate

Controls how big the steps I am taking

}

$$\frac{dJ(w)}{d(w)} = ?$$

The updates of the parameters are then

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

$$b := b - \alpha \frac{dJ(w,b)}{db}$$

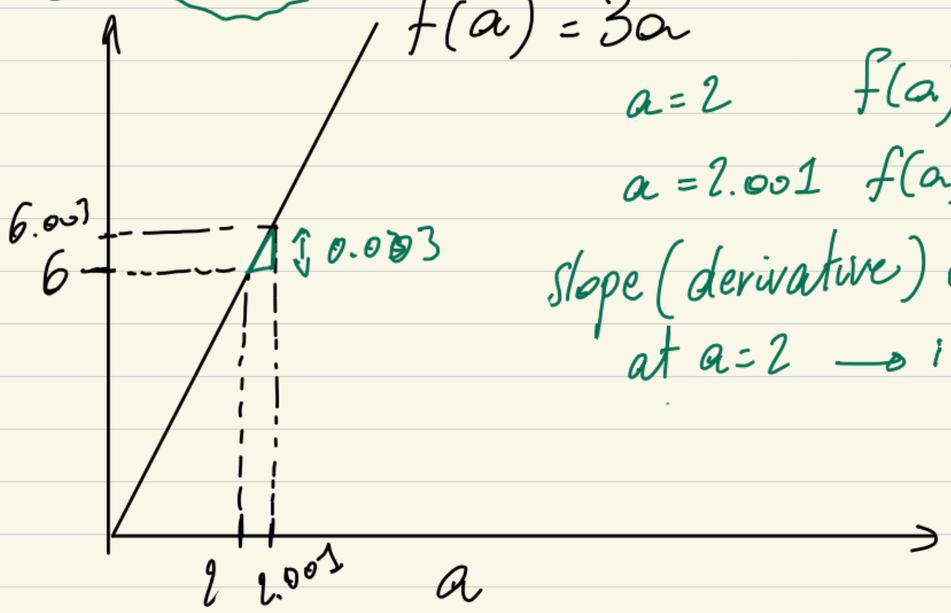
To write the symbol of the derivative of a function of 2 or more variables ⇒ we write

$$\frac{\partial J(w,b)}{\partial w}$$ ← partial derivative Symbol

$$\frac{\partial J(w,b)}{\partial b}$$

Derivatives

Intuition of derivatives



$a = 2 \quad f(a) = 6$

$a = 2.001 \quad f(a) = 6.003$

slope (derivative) of $f(a)$ at $a = 2 \longrightarrow$ is 3

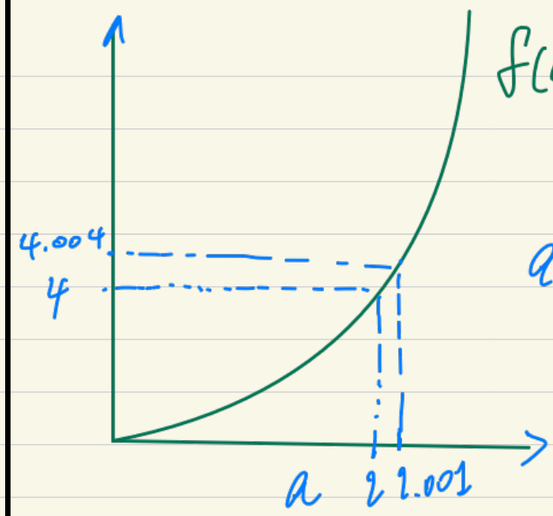
So slope $= \frac{\text{height}}{\text{width}} = \frac{0.003}{0.001}$

if $a = 5 \quad f(a) = 15$

if $a = 5.001 \quad f(a) = 15.003$

If I push a little to the right $f(a)$ goes up by 3.

slope of $f(a) = \frac{d\,f(a)}{da} = 3 = \frac{d}{da} f(a)$



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

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

because $f(a)$ goes 4 times higher when pushed to the right

slope (derivative) of $f(a)$ at $a = 2$ is 4

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

$a = 5 \quad f(a) = 25$

$a = 5.001 \quad f(a) = 25.010$

$\frac{d}{da} f(a) = 10$ when $a = 5$

$\frac{d}{da} f(a) = \frac{d}{da} a^2 = \boxed{2a}$

$w_i := w - \alpha \frac{dJ}{dw}$

similar

If we have a value "a", we push it slightly to the right new $f(a)$ = old $f(a) + \frac{d}{da} f(a) \times$ distance pushed to right

More derivative examples:

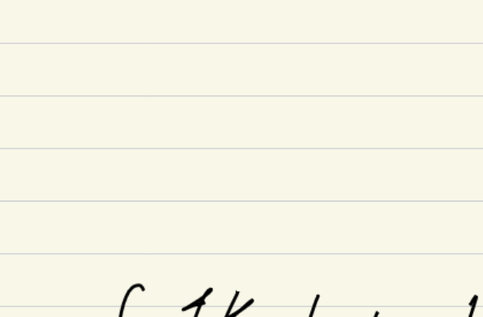
$f(a) = a^2 \quad \frac{d}{da} f(a) = 2a$

$f(a) = a^3 \quad \frac{d}{da} f(a) = 3a^2$

if J knudged a to the right by 0.001 f(a) will goup by $3a^2$

Let's say that $f(a) = \log(a)$ ($\ln a$ or $\log_e a$)

$\frac{d}{da} f(a) = \frac{d}{da} \log a = \frac{1}{a}$

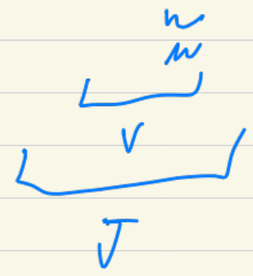


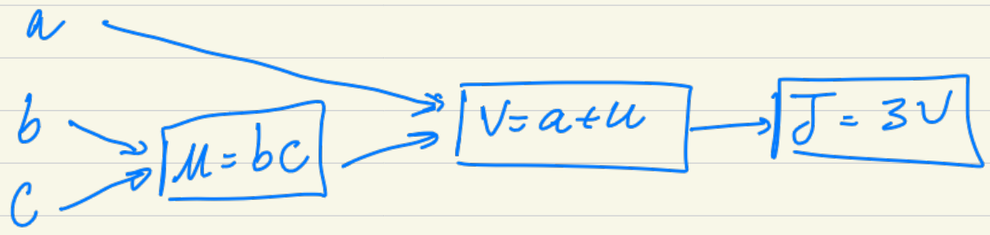
if $a = 2 \rightarrow f(a) = \log(2)$

$a = 2.001 \rightarrow f(a) = \log(2) + \frac{1}{a} * 0.001$

Computation graph

$J(a,b,c) = 3(a+b+c)$



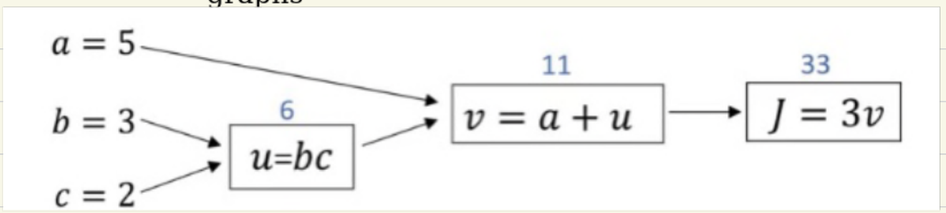


In order to compute J we go from left to right

to calculate derivatives we go from right to left

This will be clear in the next couple of videos.

Derivatives with a computation graphs



$\frac{dJ}{dv} = ?$ (If we take the value of v and change it how will the value of J change?)

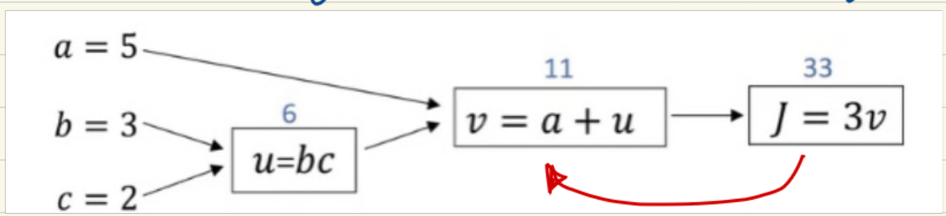
$\frac{dJ}{dv} = 3$

J = 3v

v = 11 → 11.001

J = 33 → 3.003

In the terminology of back propagation, to compute derivates of J we have to go back in the computation graph



$\frac{dJ}{da} = ?$ ⇒ a = 5 → 5.001, v = 11 → 11.001, J = 33 → 33.003 so $\frac{dJ}{da} = 3$

If I change a that will change V & that will change J

This is called the chain rule

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

$\frac{dJ}{da} = 3 \cdot 1 = 3$

If J change "a" by 0.001 V changes with the same amount

So $\frac{dV}{da} = 1$

$\frac{dJ}{du} = \frac{dJ}{dV}\frac{dV}{du} = 3 * 1 = 3$

$\frac{dJ}{db} = \frac{dJ}{dV}\frac{dV}{du}\frac{du}{db}$ ⇐ Chain Rule

$= 3 * 1 * 2 = 6$

b = 3 → 3.001

u = b.c = 6 → 6.002

Take home message → To take derivatives it's better to go from right to left (back propagation)

Logistic regression gradient decent

How to do gradient descent for logistic regression?

Logistic Regression Recap

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

$$\hat{y} = a = \sigma(z)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

Let's Say we have only 2 features x_1 & x_2

To compute derivatives of the loss:

$$* \frac{d\mathcal{L}(a,y)}{da} = -\frac{y}{a} + \frac{(1-y)}{(1-a)} *$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da}\frac{da}{dz} = \left(\frac{-y}{a} + \frac{1-y}{1-a}\right)\frac{da}{dz}$$

Not Mentioned by Andrew Ng:

To calculate the derivative of the Sigmoid function

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

$$\frac{da}{dz} = \sigma'(z) = \frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z))$$

Reminder that: $\boxed{\frac{d}{dz} e^z = e^z}$

quotient rule for derivatives: $f(x) = \frac{g(x)}{h(x)}$

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{(h(x))^2}$$

$$\frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] = \frac{(0)(1+e^{-z}) - \left((-e^{-z})(1)\right)}{(1+e^{-z})^2} =$$

$$\frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{(1+e^{-z})} \cdot \frac{e^{-z}}{(1+e^{-z})} =$$

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

$$\frac{1}{1+e^{-z}}\left[\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right] =$$

$$\sigma(z)\left[1-\sigma(z)\right] =$$

$$\boxed{a\left[1-a\right]}$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da}\frac{da}{dz} \quad \leftarrow a(1-a) = \boxed{a-y}$$

$$\boxed{-y/a + \frac{1-y}{1-a}}$$

$$\boxed{\frac{d\mathcal{L}}{dw_1} = x_1 \frac{d\mathcal{L}}{dz}} \qquad \boxed{\frac{d\mathcal{L}}{dw_2} = x_2 \frac{d\mathcal{L}}{dz}} \qquad \boxed{\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{dz}}$$

Gradient descent on m examples

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

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

What was shown in the previous lesson was how to calculate derivates for 1 training example.

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

Logistic Regression on m examples:

$J=0$, $dw_1=0$, $dw_2=0$, $db=0$ ← init

for $i=1$ to m:

forward:

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

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

$$J \mathrel{+}= -\left(y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)}) \right)$$ ← loss fn

backward:

$$\frac{d\mathcal{L}}{dz^{(i)}} \mathrel{+}= a^{(i)} - y^{(i)}$$

$$\frac{d\mathcal{L}}{dw_1} \mathrel{+}= x_1^{(i)} \frac{d\mathcal{L}}{dz^{(i)}}$$

$$\frac{d\mathcal{L}}{dw_2} \mathrel{+}= x_2^{(i)} \frac{d\mathcal{L}}{dz^{(i)}}$$

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

$J \mathrel{/}= m$

$dw_1 \mathrel{/}= m$; $dw_2 \mathrel{/}= m$; $db \mathrel{/}= m$

After the previous algorithm, to implement one step of the gradient descent we do

$$w_1 := w_1 - \alpha\, dw_1$$ ← $\frac{d\mathcal{L}}{dw_1}$

$$w_2 := w_2 - \alpha\, dw_2$$ ← $\frac{d\mathcal{L}}{dw_2}$

$$b := b - \alpha\, db$$ ← $\frac{d\mathcal{L}}{db}$

Vectorization

Vectorization => enables to get rid of for loops in your code

What is Vectorization?

In logistic Regression, we need to compute:

$z = w^T x + b$ Where $w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

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

Non-Vectorized Implementation:

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

Vectorized implementation

```
z = np.dot(w, x) + b
```

More Vectorization examples.

matrix, vector

$u = Av$

$u_i = \sum_j A_{ij} v_j$

Non Vectorized Implementation

```
u = np.zeros(n, 1)
for i ...
  for j ...
    u[i] += A[i][j] * v[j]
```

Vectorized Imp

```
u = np.dot(A, v)
```

Another example: Say you need to apply the exponential operation on every element of a matrix/Vector

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

Non Vectorized implementation

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

Vectorized imp

```
u = np.exp(v)
```

Logistic Regression derivatives

```
J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = w^T x(i) + b
    a(i) = σ(z(i))
    J += -[y(i) log a(i) + (1 - y(i)) log(1 - a(i))]
    dz(i) = a(i) - y(i)
    dw1 += x1(i) dz(i)
    dw2 += x2(i) dz(i)
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m, db = db/m
```

If I had more than 2 features I would do a for loop over the weights w1 w2 b

To eliminate this second for loop

$dw = np.zeros((n_x, 1))$ ← for the initialization

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

will eliminate the line marked above

Vectorising logistic regression

Forward propagation step in a logistic Regression:

$z^{(1)} = w^T x^{(1)} + b$, $a^{(1)} = \sigma(z^{(1)})$ — for 1st training example

$z^{(2)} = w^T x^{(2)} + b$, $a^{(2)} = \sigma(z^{(2)})$ — for 2nd training example

$z^{(3)} = w^T x^{(3)} + b$, $a^{(3)} = \sigma(z^{(3)})$ — for 3rd training example

The question now is how to do it without an explicit for loop

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

Examples are stacked by columns. # of features. # examples

Let's define $1 \times m$ vector $Z = [z^{(1)} z^{(2)} \cdots z^{(m)}]$

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

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

$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w^T x^{(m)} + b]$

$1 \times m$ vector

$$Z = [z^{(1)} z^{(2)} \dots z^{(m)}] = w^T X + [b\ b\ b \dots b]$$

$$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \dots w^T x^{(m)} + b]$$

In python, this implemented in:

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

b is just a real number ⟸ added to a vector using "Broadcasting"

So now we implemented "Z", how do we implement "a".

* We define vector $[a^{(1)} a^{(2)} \dots a^{(m)}] = A$

$$= \sigma(Z)$$

Victor rising logistic regression gradient decent

In this lesson, we will see how to use vectorization for gradient's calculations.

When we saw the gradient's calculation:

* $\frac{d\mathcal{L}}{dz^{(1)}} = a^{(1)} - y^{(1)}$ $\frac{d\mathcal{L}}{dz^{(2)}} = a^{(2)} - y^{(2)}$ ...

↳ denoted by $dz^{(1)}$

We define $dZ = [dz^{(1)}, \dots dz^{(m)}]$ ⇒ $1 \times m$ vector

From the prev lesson: $A = [a^{(1)} a^{(2)} \dots a^{(m)}]$ ⇒ $1 \times m$ vector

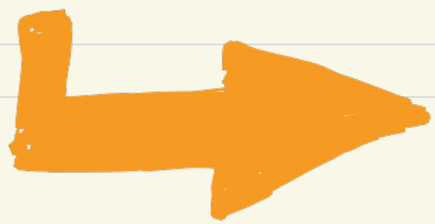
$Y = [y^{(1)} \dots y^{(m)}]$

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

So with one line of code we can do this calculation

In a previous lesson, we were able to get rid of one for loop (over the set of features for one training example.

But we still have to get rid of a for loop over the training examples.



Reminder From a previous Lesson

```
J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = w^T x(i) + b
    a(i) = σ(z(i))
    J += -[y(i) log a(i) + (1 - y(i)) log(1 - a(i))]
    dz(i) = a(i) - y(i)
    dw1 += x1(i) dz(i)
    dw2 += x2(i) dz(i)
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m, db = db/m
```

If I had more than 2 features I would do a for loop over the weights w1 w2 b

To eliminate this Second for loop

$dw = np.zeros((n_x, 1))$ ← for the initialization

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

will eliminate the line marked above

What we do basically with db is

$$db = \frac{1}{m}\sum_{i}^{m} dz^{(i)}$$

In python

$$= \frac{1}{m} np.Sum(dz)$$

What about dw?

$$dw = \frac{1}{m} X dz^T = \frac{1}{m}\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}\begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix} =$$

$n_x \times 1$

$$\frac{1}{m}\left[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}\right] =$$

In python

So let's put everything together:

Inefficient Python implementation

] In the previous lesson we got rid of the for loop over the features

How to implement logistic Regression Gradient Descent efficiently in Python.

↳ $Z = w^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

$w := w - \alpha\, dw$

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

This is only one iteration of Gradiat Descent

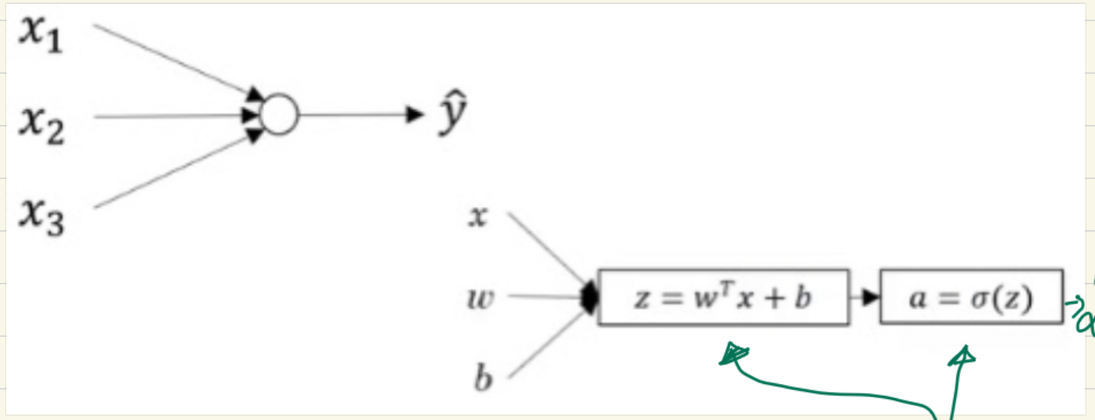
For multiple iterations ⇒ I will need a for loop !!!

Course 1

Week 3

Neural networks overview

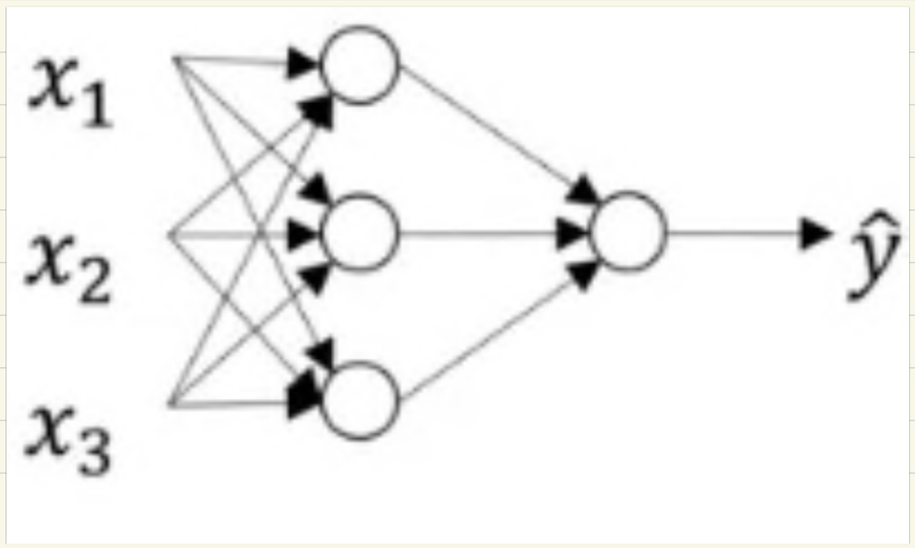
A logistic Regression is like a Small Neural Netw.



In logistic Regression we have two steps of calculations

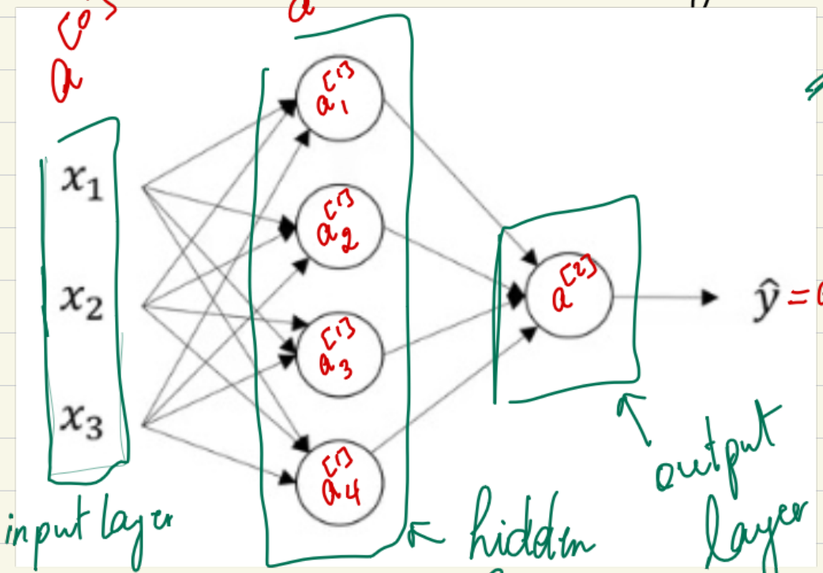
In Neural Nets, more Calculations are needed.

* will get clearer in the next lessons *



Neural network representation

A neural network with Si le hidden layer



2 layer NN

Some new notations:

Input Layer $X = a^{[0]}$ ⇒ "a" for activation

Hidden Layer "$a^{[1]}$" & each node will be called $a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}$

Output Layer "$a^{[2]}$" => which is a number

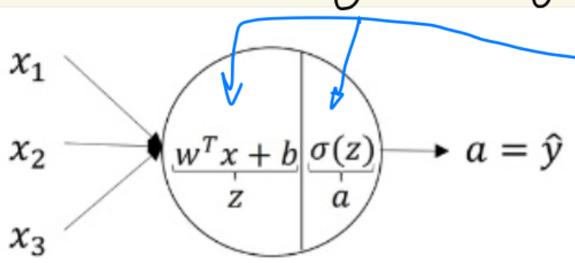
Note that: the hidden layer $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$

- the hidden layer has params associated with it noted $w^{[1]}$, $b^{[1]}$
  - $b^{[1]}$ → (4, 1) vector
  - $w^{[1]}$ → (4, 3) matrix
    - 3 nodes in the i/p layer
    - 4 nodes in hidden layer

When we count layers in NN, we don't count the i/p layer

Computing in neural networks output

The circle in a logistic Regression has 2 steps of computations.



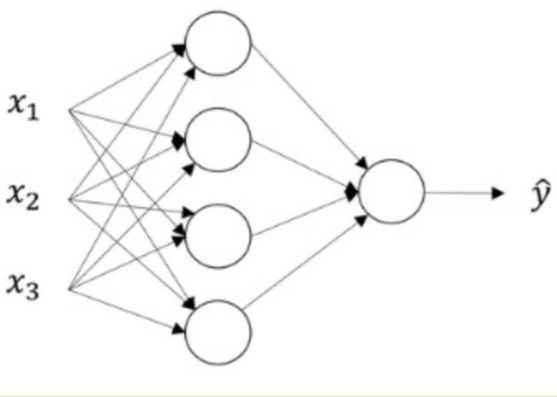
$z = w^T x + b$

$a = \sigma(z)$

A NN does this a lot more

For Example For a neural network like the following:

The Same is done for each node in the hidden layer

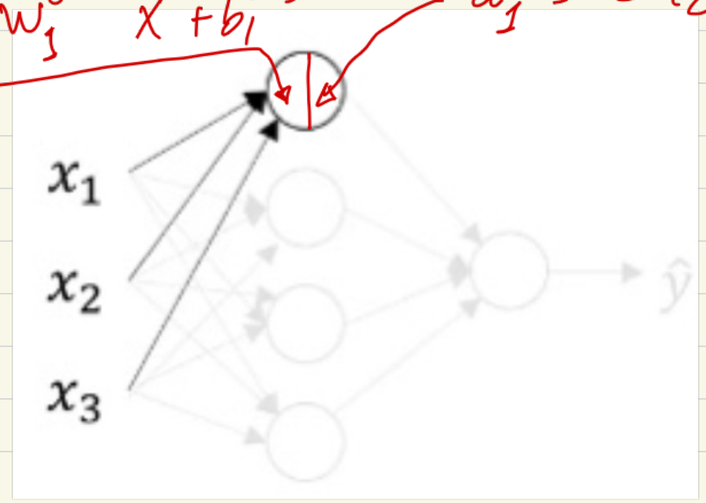


let's take a look on what happens in a single node in a hidden layer

$z_1^{[1]} = w_1^{[1]T} X + b_1^{[1]}$ $\quad a_1^{[1]} = \sigma(z_1^{[1]})$

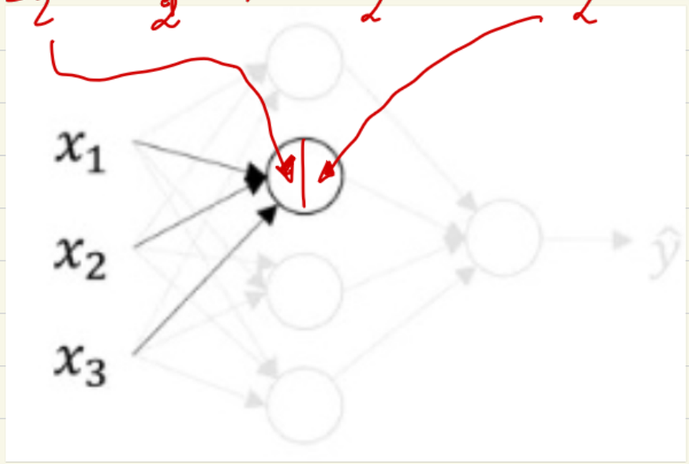
$[l]$ ← layer

a_i ← node



Same for the second in the hidden layer

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



Etc for all the other nodes

To do this using vectors & matrices (to avoid for loops in Python)

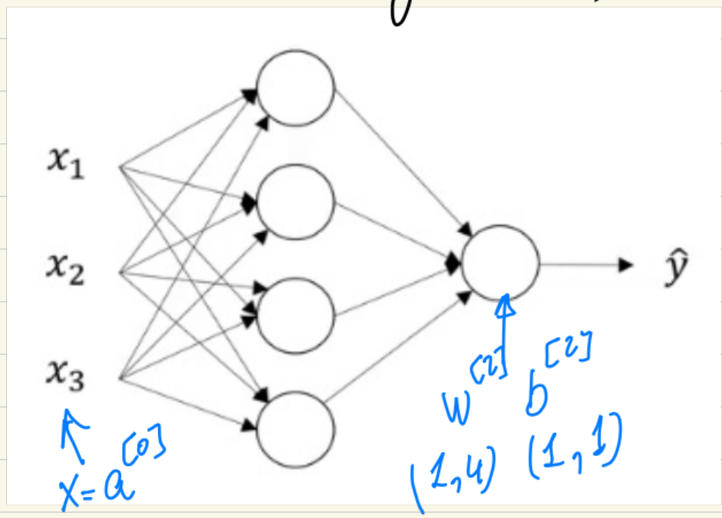
We end up with:

$$\underbrace{\begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix}}_{W^{[1]}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{X} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]}\ (4\times 1)\ \text{vector}} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]}x + b_2^{[1]} \\ w_3^{[1]}x + b_3^{[1]} \\ w_4^{[1]}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$$

Similarly,

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

So given an i/p X:



$$z^{[1]} = W^{[1]}X + b^{[1]} \quad (4,1) = (4,3)(3,1) + (4,1)$$

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

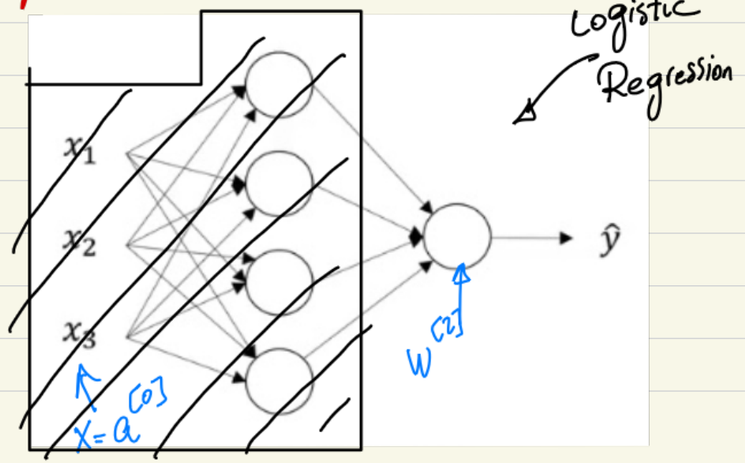
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \quad (1,1) = (1,4)(4,1) + (1,1)$$

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

Remember that we said the input layer X is $a^{[0]}$

So in the equation we can replace X by $a^{[0]}$

If we take the figure above & cover up the left part like this, we end up with a structure like a logistic Regression



Vectorising across multiple examples

One hidden Layer

These are our four equations from the previous lesson. They tell given an i/p vector x how to get a prediction "a2"

$$z^{[1]} = W^{[1]} x + b^{[1]} \quad (4,1) = (4,3)(3,1) + (4,1)$$

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

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

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

If I have m feature example. we should repeat these equations

$x \longrightarrow a^{[2]} = \hat{y}$

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

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

$\vdots$

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

Round bracket (i) means the i^{th} example.

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

A non vectorized implementation will consist of a for loop to loop on each example.

```
for i = 1 to m,
    z[1](i) = W[1] x(i) + b[1]
    a[1](i) = σ(z[1](i))
    z[2](i) = W[2] a[1](i) + b[2]
    a[2](i) = σ(z[2](i))
```

Recall we define matrix $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x # of features, m # examples)

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

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

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

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

Basically what we have done is to use Capital X matrix instead of lower case $x^{(i)}$ by stacking $x^{(i)}$ in columns. Same thing can be done with Z

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

Same thing can be done with the A matrix

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

Remember from a previous lessons that

$Z^{[1](1)}$ is $(4, 1)$ matrix where 4 was the number of nodes. So Capital Case Z matrix

Contains $\begin{bmatrix} | & | & & | \\ Z^{[1](1)} & Z^{[1](2)} & \cdots & Z^{[1](m)} \\ | & | & & | \end{bmatrix}$ ↕ hidden units

* a number of columns = number of example
& # of rows = number of nodes in the layer 1 *

Remember the node number was node $Z_1^{[1](1)}$ → first node

Explanation for vectorised implementation

Justification for vectorized implementation

$$Z^{[1](1)} = W^{[1]} x^{(1)} + b^{[1]}_{=0} \leftarrow \text{First training Exaple}$$

$$Z^{[1](2)} = W^{[1]} x^{(2)} + b^{[1]}_{=0} \leftarrow \text{Second training Exaple}$$

$$Z^{[1](3)} = W^{[1]} x^{(3)} + b^{[1]}_{=0} \leftarrow \text{Third exaple}$$

for the moment we assumbe $b^{[1]} = 0$

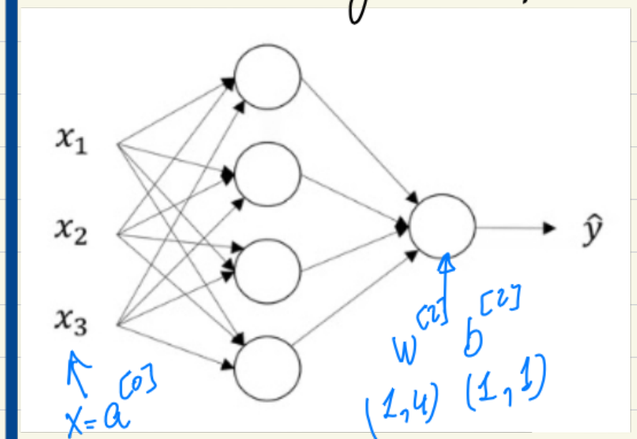
↳ for Simplification

$$W^{[1]} = \begin{bmatrix} — \\ — \\ — \\ — \end{bmatrix}$$

So $W^{[1]} x^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ⇒ Column vector

To Know Why it's a column vector ⇒ Check next slide

So given an i/p X:



$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$(4,1) \quad (4,3)\ (3,1)\ (4,1)$

$$a^{[1]} = \sigma(Z^{[1]})$$
$(4,1) \quad (4,3)$

$$Z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$(1,1) \quad (1,4)\ (4,1)\ (1,1)$ — a value

$$a^{[2]} = \sigma(Z^{[2]})$$
$(1,1) \quad (1,1)$

(The example from a previous lesson)

$W^{[1]}$ here is (4,3) matrix — 3 because of the i/p features, 4 because of the # nodes

$x^{(1)}$ is a vector (3,1) — Multiplying the ⟶ will result in a Column vector.

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

So now if we consider the training set Capital X

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \Rightarrow \text{multiplying it by } W^{[1]}$$

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

$$= \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$$

Assuming $b^{[1]} = 0$ ⟸

We can also add $b^{[1]}$ won't change anything

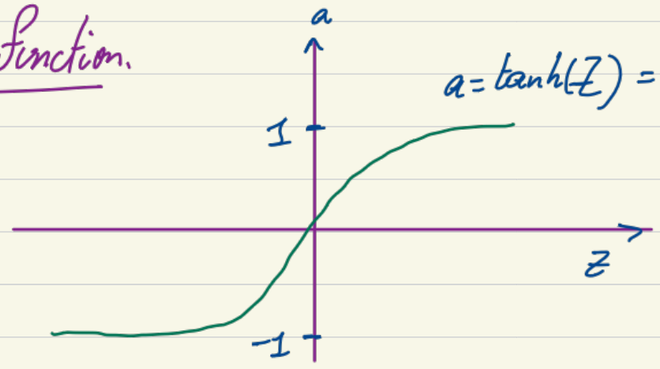
Activation function

One of the choices when designing a NN is to choose the activation function. So far we have been using the Sigmoid function

$a = \frac{1}{1+e^{-z}}$ ⇒ Sigmoid function

↳ the sigmoid goes between 0 & 1

↳ Another activation function that works better most of the time is the tanh function.



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

* The tanh function is a shifted version of the Sigmoid function

It work's better than the sigmoid because it has a mean of zero

So the mean of the activations coming out of the hidden layer is close to Zero. ⇒ We will see more in the second course

In the hidden layer tanh is almost always superior than the Sigmoid fn.

The only exception is for the output layer

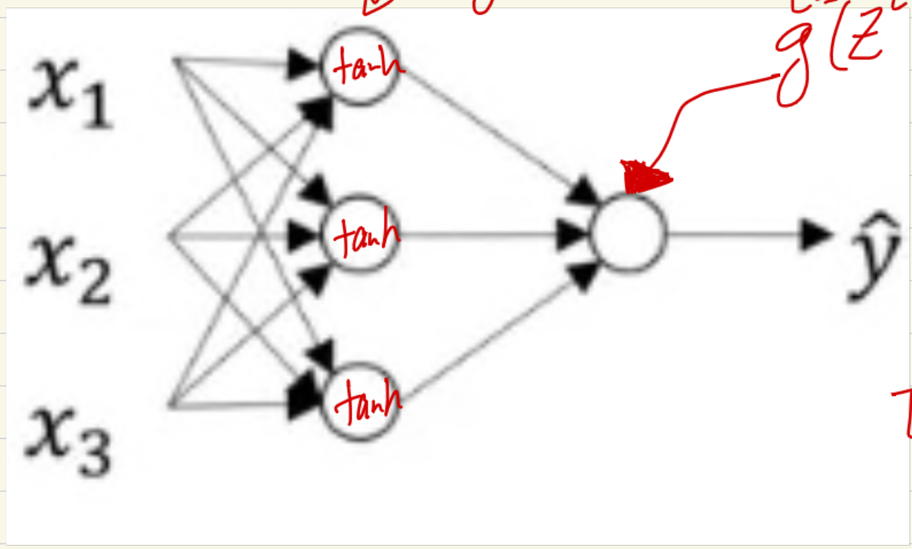
↳ if y is 0 or 1 ⇒ So the Sigmoid is used

or between 0 & 1

So we can use tanh for hidden layers & sigmoid for o/p layer.

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

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



Problem with Tanh & Sigmoid

Again the problem of tanh & Sigmoid is that when z is ver large or very small, the slope (gradient) is close to zero. One other choice is

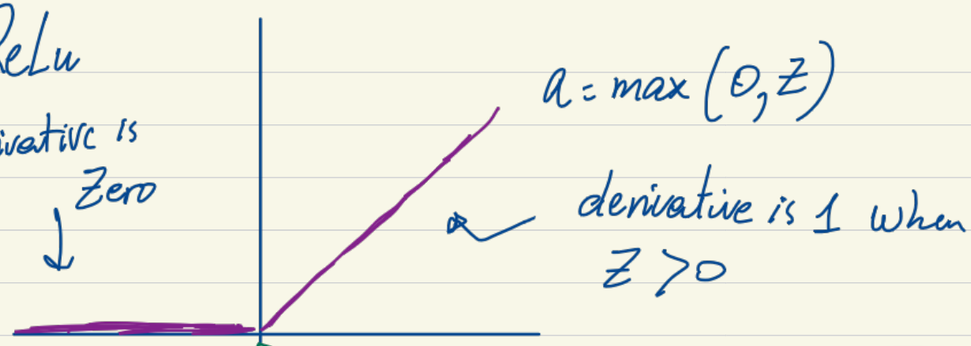
Rectified linear Unit Relu

ReLu

derivative is Zero

$a = \max(0, z)$

derivative is 1 when $z > 0$



Technically when $z = 0$, the slope is NOT DEFINED

but we never get exactly $z = 0$

A Rule of Thumb ⟹ If the output is 0 & 1 ↳ Binary classif.

The output layer ⟹ Sigmoid

for all other units ReLu is the default choice of activation function.

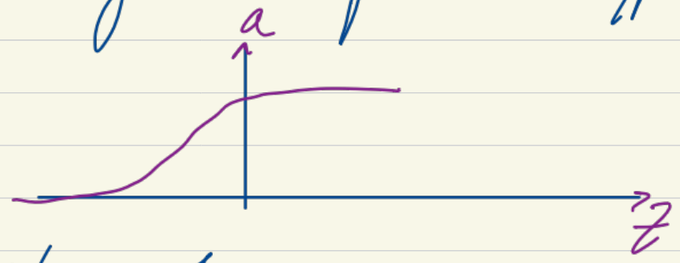
Disadv. of ReLu is the slope of 0 when z is −ve

So there is another version of the ReLu called Leaky ReLu

ReLu (or Leaky Relu) makes the Algo learns a lot faster than the sigmoid or tanh

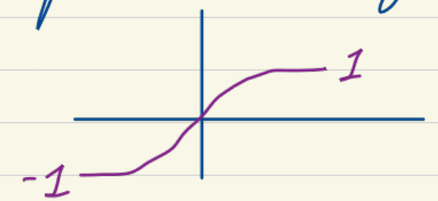
Pros & Cons of activation functions

- Never use Sigmoid except for the o/p layer



$$\text{Sigmoid} = \frac{1}{1 + e^{-z}}$$

- tanh is strictly superior than the sigmoid



$$\tanh: a = \frac{e^{z} + e^{-z}}{e^{z} + e^{-z}}$$

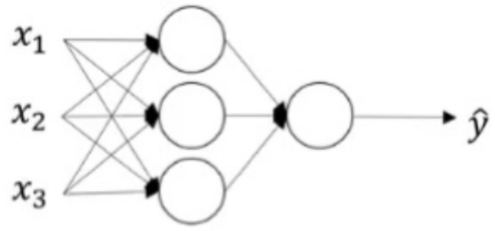
- We can always use tanh by default

↳ it has a Slope = 1

- We can also use Leaky ReLu

Why do you need nonlinear activation functions

* Activation function



Let's try to get rid of the function $g()$ & make it a linear fn

Given x:

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

$a^{[1]} = g^{[1]}(z^{[1]})$ → $Z^{[1]}$ → or $g(z) = z$ ↳ linear activation function

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

$a^{[2]} = g^{[2]}(z^{[2]})$ $Z^{[2]}$

In this case my NN is learning a linear function of my i/p. No matter how many layers I have !!

It turns out that in this case,

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

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

$= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} = W^{[2]}W^{[1]}x + W^{[2]}b^{[1]} + b^{[2]}$

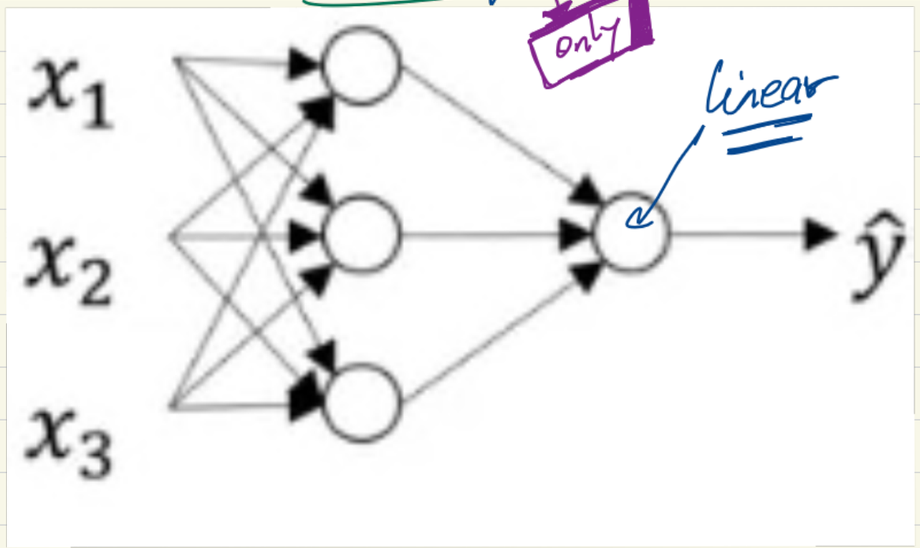
($a^{[1]}$)

$W'x + b'$

* So a Linear hidden Layer is useless *

One reason to use where I might use a linear activation fn is in Regression problems → where y takes a continuous value ⇒ a real number

↳ So I use it in my o/p layer → so my output can $\in]-\infty, +\infty[$



Derivatives of activation functions

When implement Backprop ⇒ I will need to calc the slope of activation function

Sigmoid function

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

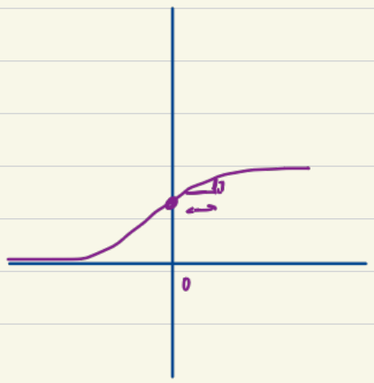
I proved it before in my notes.

So if $z = 10$

$g(z) \simeq 1$

$\frac{d}{dz} g(z) = 1(1-1) \simeq 0$

↳ Slope close to Zero **



if $z = 0$ $g(z) = 1/2$

$\frac{d}{dz} g(z) = \frac{1}{4}$ *

↳ $g'(z)$

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

if $z = 10$ $\tanh(z) \simeq 1$ $g'(z) \simeq 0$

if $z = -10$ $\tanh(z) = -1$ $g'(z) \simeq 0$

if $z = 0$ $\tanh(z) = 0$ $g'(z) \simeq 1$

Relu activation function

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

So technically it will be $z = 0.\ldots\ldots 1$.

(It won't be undefined)!!

Gradient descent for neural networks

Here we will see how to implement Gradient Descent for a NN with a single hidden Layer

Params: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \rightarrow (n^{[2]}, 1)$

$(n^{[1]}, n^{[0]})$ $(n^{[1]}, 1)$ $(n^{[2]}, n^{[1]})$

As a reminder:

$n_x = n^{[0]}$ i/p features

$n^{[1]}$ hidden units

$n^{[2]}$ output units = 1

Column vectors

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

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

(If I am doing binary classification I can use the same loss function

$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

To train the algo I will need to perform gradient descent:

first I have to initialize the parameters (Randomly)

Repeat {

Compute preds $(\hat{y}_i \quad i = 1 \ldots m)$ ← for all examples

$dW^{[1]} = \frac{dJ}{dW^{[1]}} \quad db^{[1]} = \frac{dJ}{db^{[1]}}$, ... ← Compute derivatives

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

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

$W^{[2]} = W^{[2]} - \alpha\, dW^{[2]}$

⋮

}

Formulas for Computing derivatives:

- Forward propagation

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

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

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

$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$ ← Sigmoid if I am doing binary classification

Back propagation

$dZ^{[2]} = A^{[2]} - Y$ → $Y = [y^{(1)}\ y^{(2)} \ldots y^{(m)}]$ ← These equations was seen in the LR part

↳ 1 x m vector

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

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

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

$(n^{[1]}, m)$; * element wise product ; $g^{[1]\prime}$ ↳ the derivative of the activation function in the hidden layer.

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

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

Course 1

Week 4

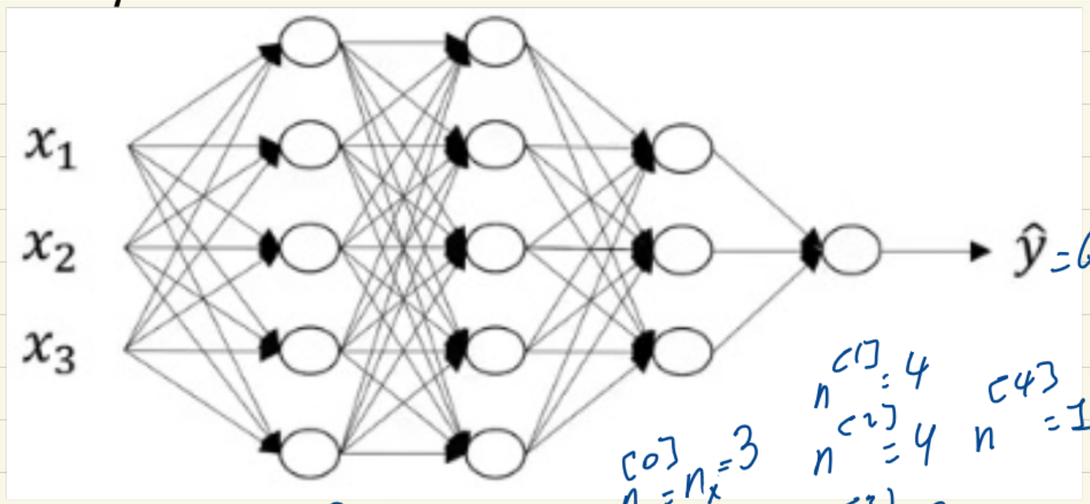
Deep L layer neural network

What is a deep neural network?

↳ We have more than 1 hidden layer

* A Logistic Regression is a shallow NN!! *

Deep Neural Network notation



$L = 4$ # of layers

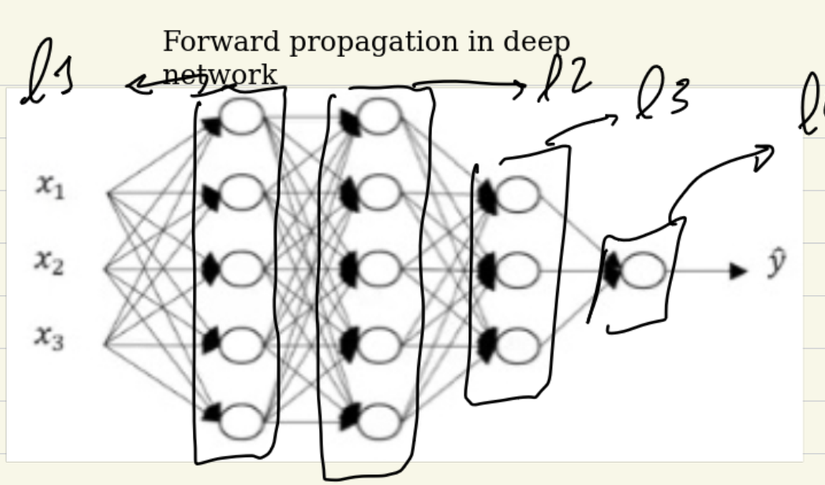
$n^{[l]}$ = # of units (nodes) in layer l

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

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

$a^{[0]} = x$ & $y = a^{[L]}$

Forward propagation in deep network



* How to do forward propagation for a single example?

x: $z^{[1]} = W^{[1]} x + b^{[1]}$ → $x = a^{[0]}$ → the activation of layer 0

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

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

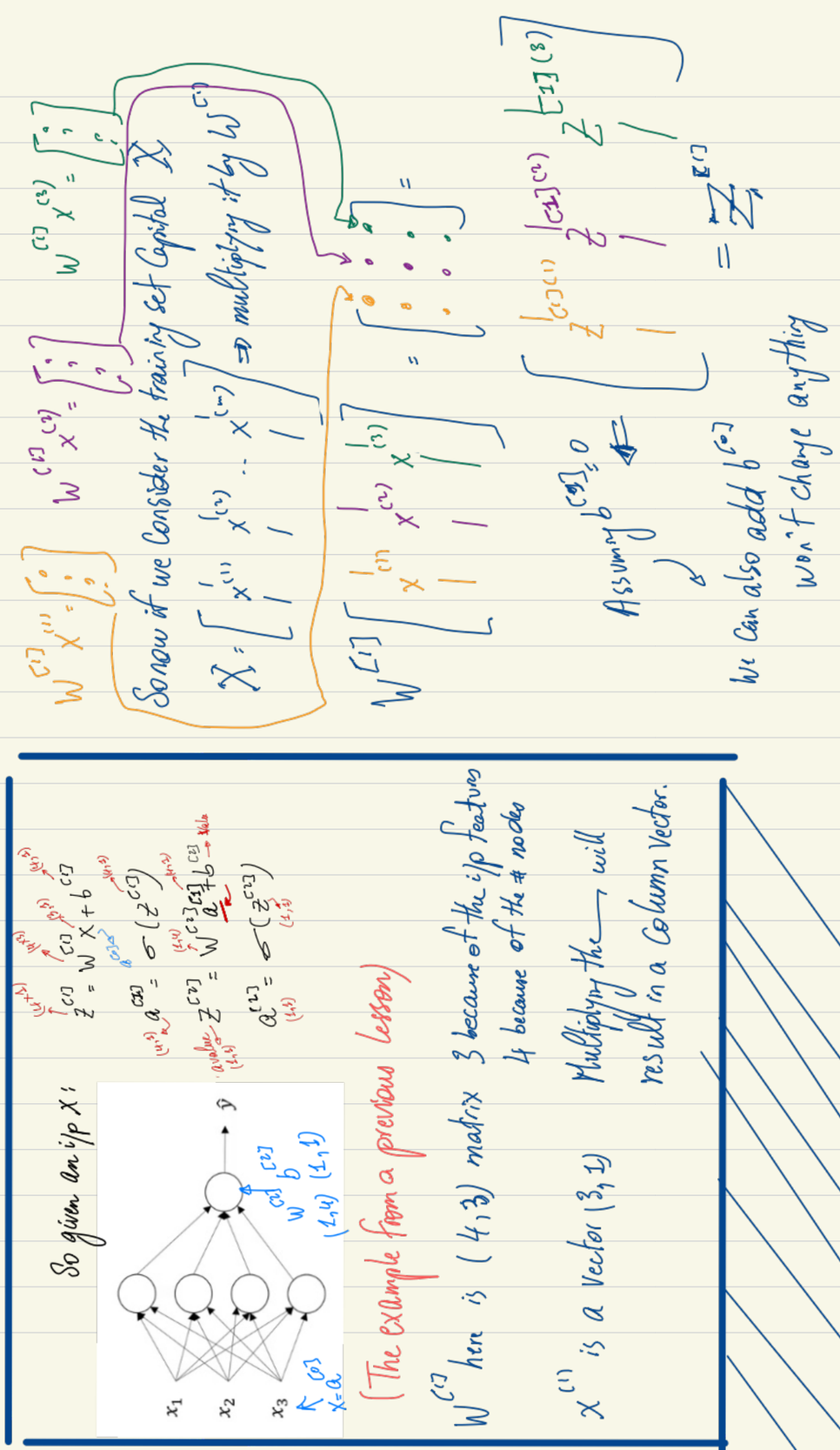
$a^{[2]} = g^{[2]}(z^{[2]})$

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

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

Vectorization: (here we will use capital letters)

A reminder for a previous lesson is in the next page.



(The example from a previous lesson)

$W^{[1]}$ here is $(4,3)$ matrix 3 became of the i/p features
4 became of the # nodes

$x^{(1)}$ is a vector $(3,1)$ Multiplying the — will result in a Column vector.

When doing Vectorization:

$$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\hat{Y} = g(Z^{[4]}) = A^{[4]}$$

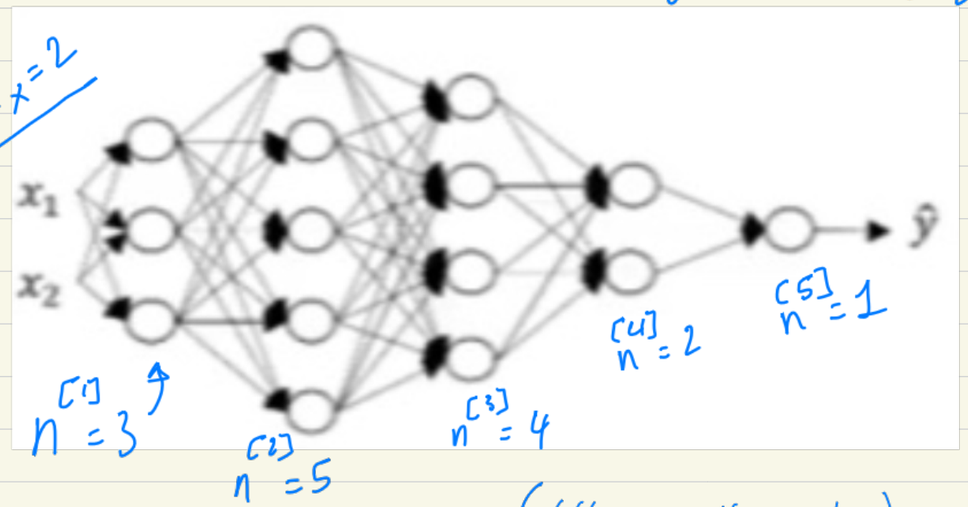
we have a for loop on the layers!! for l: 1...4

When doing forward prop it's okay to use an explicit for loop

To have a bug free implementation we have to think somehow systematically about the dimension of matrices!!

Getting your matrix dimensions right

$L=5$ # of layers (not counting the i/p layer as usual)



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

(let's ignore the bias term for now)

$Z^{[1]}$: $(3,1)$ $(n^{[1]},1)$ = $W^{[1]}$: $(3,2)$ $(n^{[1]},n^{[0]})$ X: $(2,1)$ $(n^{[0]},1)$

Mon generally: $W^{[l]} = (n^{[l]}, n^{[0]})$

$$\begin{bmatrix} \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

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

Accordingly: $* Z^{[l]} = (n^{[l]}, 1) *$

What is the dimensions of the bias term $b^{[l]}$?

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

Note that when doing backpropagation $dW^{[l]}$ have the same dimensions of $W^{[l]}$

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

Let's continue with the dimensions of other terms:

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

$a^{[l]}$ have the same dimensions of $Z^{[l]}$

Let's look at vectorized implementation to consider several examples at the same time:

For only one example: $Z^{[1]} = W^{[1]} X + b^{[1]}$

$Z^{[1]}$: $(n^{[1]},1)$ = $W^{[1]}$: $(n^{[1]},n^{[0]})$ X: $(n^{[0]},1)$ → $b^{[1]}$: $(n^{[1]},1)$

For multiple examples we consider terms in Capital letters

↳ In next Page

Let's look at vectorized implementation to consider several examples at the same time:

For only one example: $Z^{[1]} = W^{[1]} X + b^{[1]}$ → $(n^{[1]}, 1)$

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

For multiple examples (we consider terms in capital letters):

* we stack the terms of all examples together

↳ $Z^{[1]} = \begin{bmatrix} z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \end{bmatrix}$ $(n^{[1]}, m)$

$Z^{[1]} = W^{[1]} X + b^{[1]}$ → the same as for only 1 example

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

↓ ↓ ↓ ↓

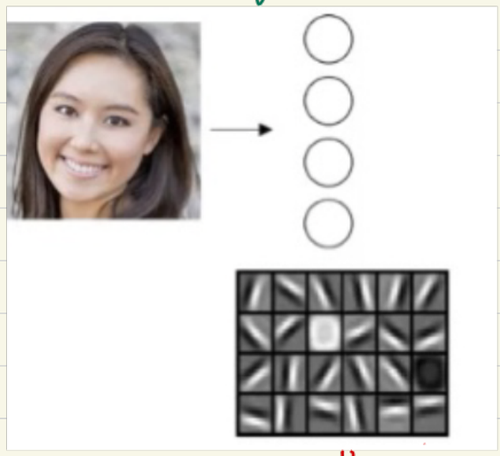
it changes | doesn't change | all the examples | doesn't change. It is added through Python broadcasting.

Why deep representations ?

Why deep networks are better than shallow nets?

Intuition about deep representation

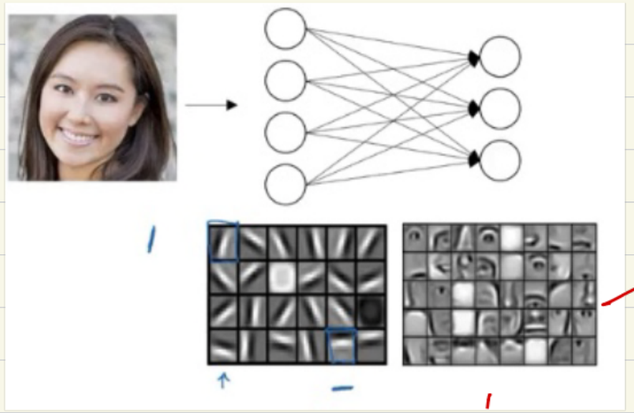
↳ if I am building a system for face detection, the first layer can be an edge detector



edge detection by 24 hidden units

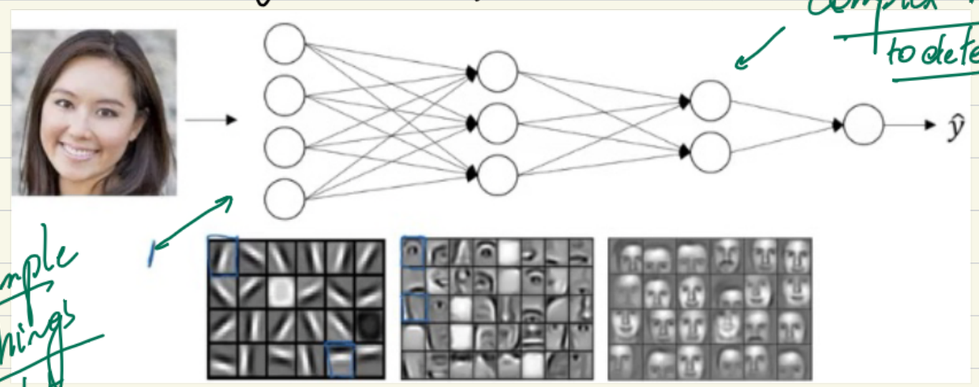
↳ A second layer may be used to group detected edges from previous layer to find parts of faces

↳ example in next page



group detected edges to form parts of faces (eyes, nose, etc--)

Finally, last layer may be useful to group parts of faces to recognize bigger parts of faces



edge dectors look at smaller parts of images

facial defeetors look at bigger parts of images

(will make more sense in the CNN Course)

This hierarichal schema can also be applied on different types of data!! (Audio for example)

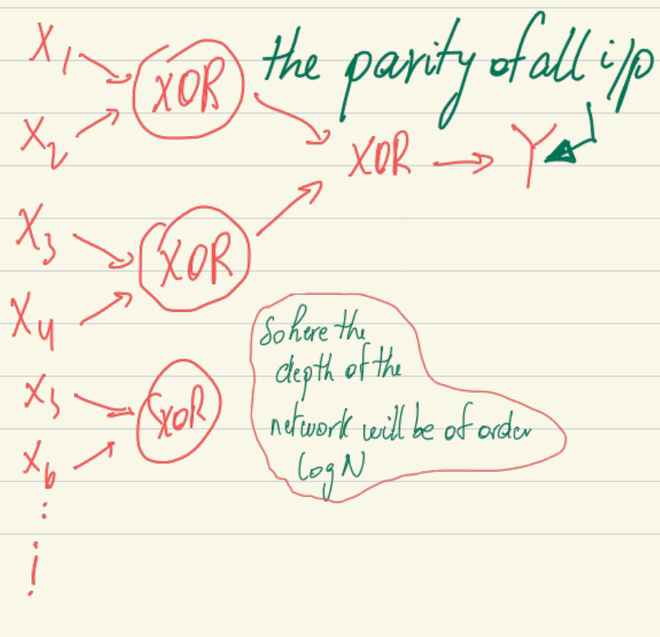
Circuit Theory & deep learning

> 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.

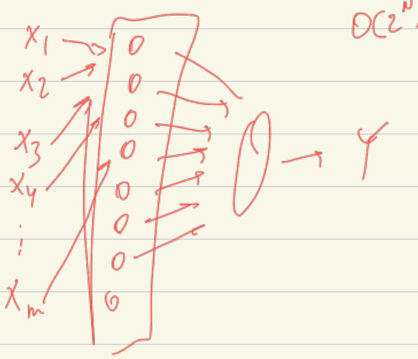
# of hidden layers is small!!

Let's say we want to compute XOR of all my i/p features:

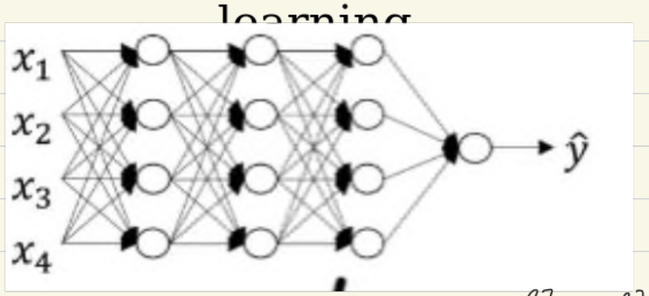
X_1 XOR X_2 X_2 XOR X_3 --



If I am forced to compute the same function with only one layer!! The # of neurons will 2^{N-1} $O(2^N)$



Building blocks of deep learning



Layer l; we have parameters: $W^{[l]}$, $b^{[l]}$

Forward: i/p: $a^{[l-1]}$, output: $a^{[l]}$

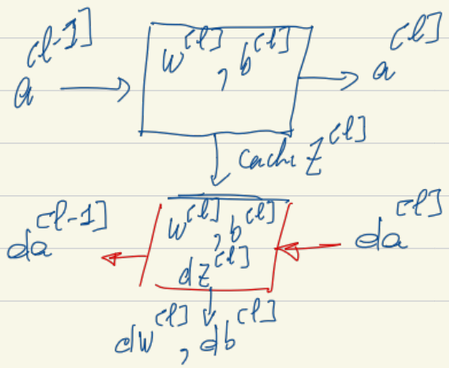
$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ — it will be useful to cache $Z^{[l]}$ for backprop.

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

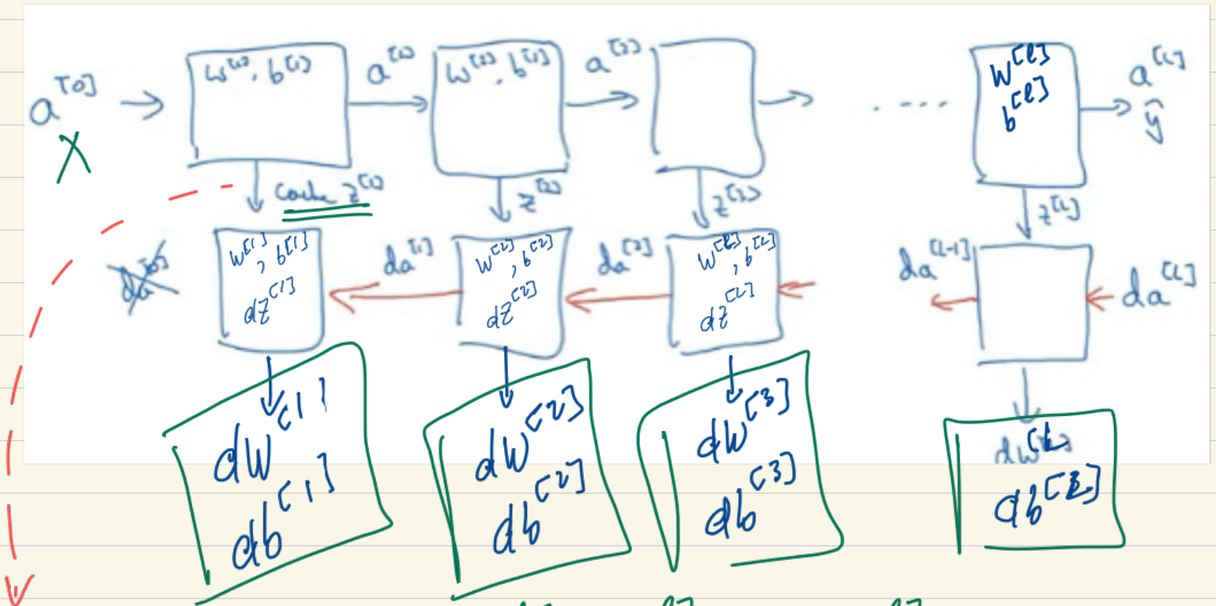
Backprop: Input: $da^{[l]}$ + $Z^{[l]}$ output: $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

So This the basic structure for implementing forward & backward steps

Layer l:



To have a fuller overview:



we store in the cache $Z^{[l]}$, $W^{[l]}$, $b^{[l]}$

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

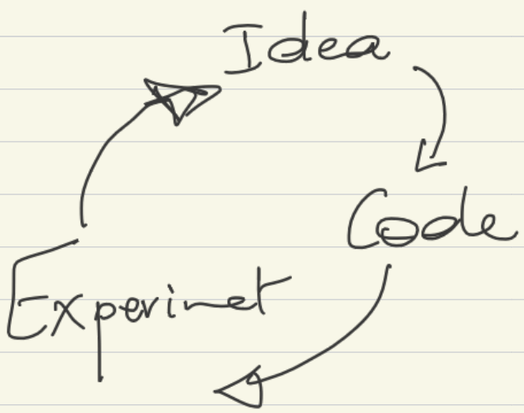
$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$

Train / Dev / Test sets

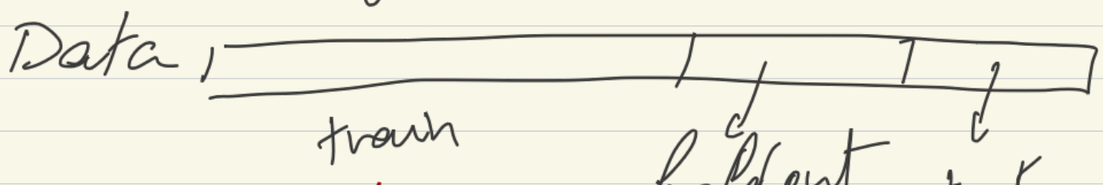
Applied ML is highly iterative Process

- # layers
- # hidden units
- learning rates
- activation fn

choices of several params.



Traditionally:



train my model

"dev set"

tune my hyperparams.

Ratios can be 60% 20% 20%

Big Data → we can increase the ratio of training data 98% 1% 1%

* Mismatched train/test distro is another problem
  * Make sure we don't have a deployment bugs!!
    - make sure
    - the Rule of thumb that dev & test sets come from the same distro.
* We can also not have a test set (& it's okay :)