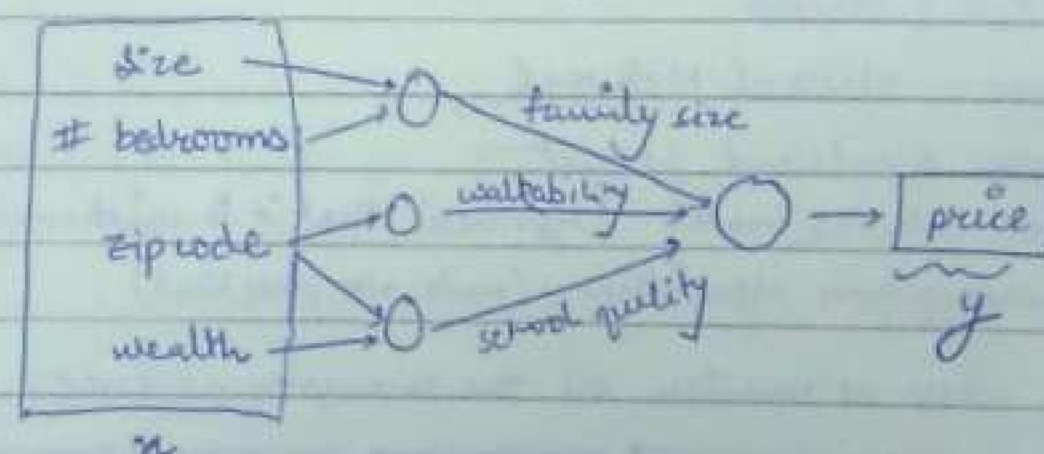
COURSERA COURSE

Course 1 (NN & DL)   DEEP LEARNING.ai

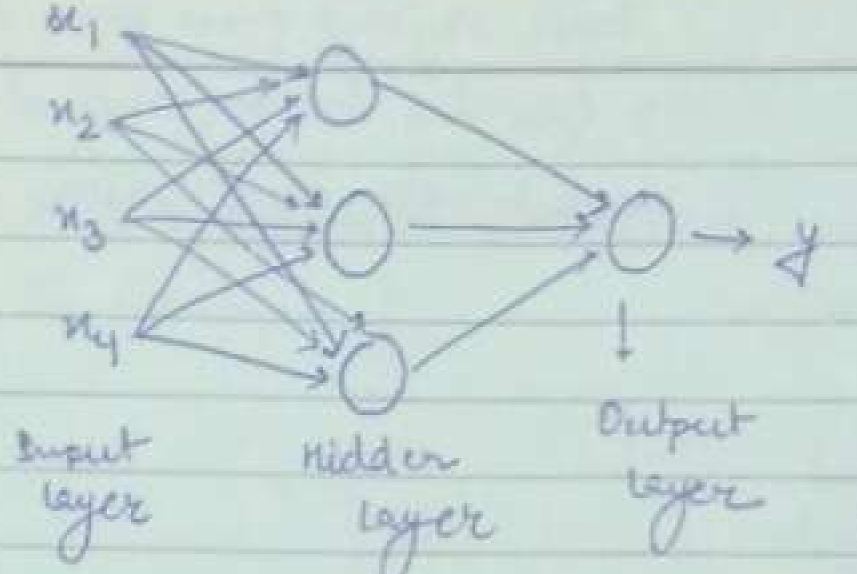
Input → ○ → Output
(Neuron)

Simplest Neural Network

All we need to feed in a neural network is input and output for entire training set. Rest everything is taken care of by itself. e.g.



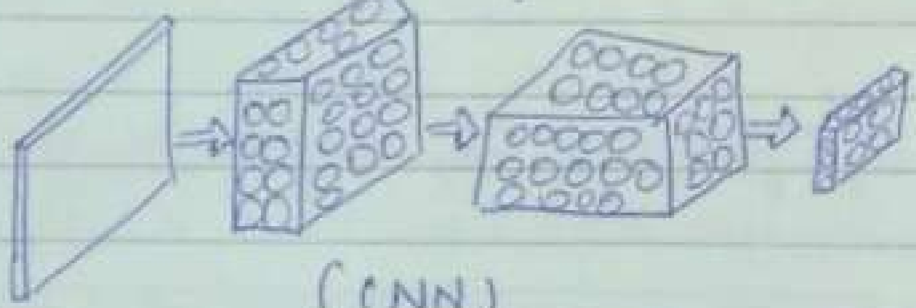
Neural network: very powerful and useful in supervised learning architectures.



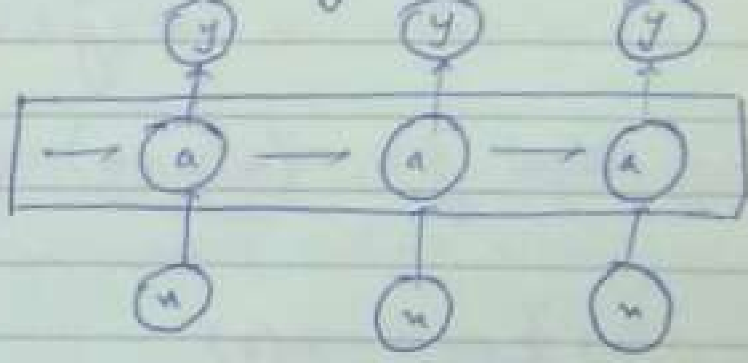
Every input layer feature is connected with every hidden layer feature.

(Standard NN)

Note: for image data, we generally use CNN, and for sequence data like audio, we use RNN, for language also as the alphabets or words come one at a time and hence is sequence data. Radar or image input might need some custom or hybrid NN.



(CNN)

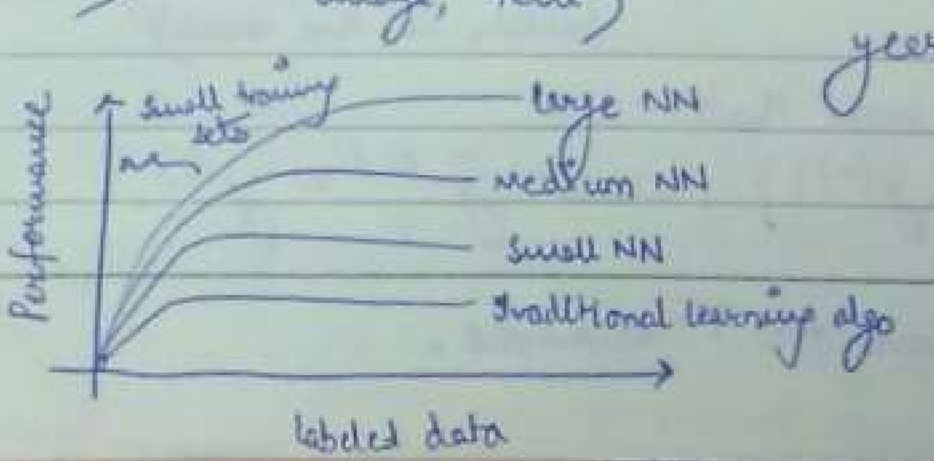


(RNN)

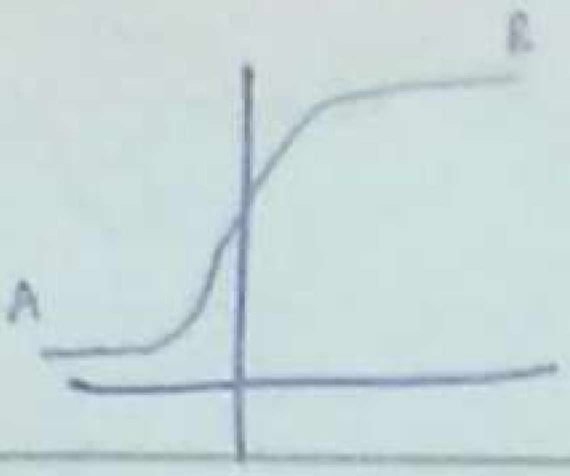
Supervised learning

- Structured data (in form of databases)
- Unstructured data (in form of audio, image, text)

Deep learning and NN has helped computers better understand unstructured data in recent years.

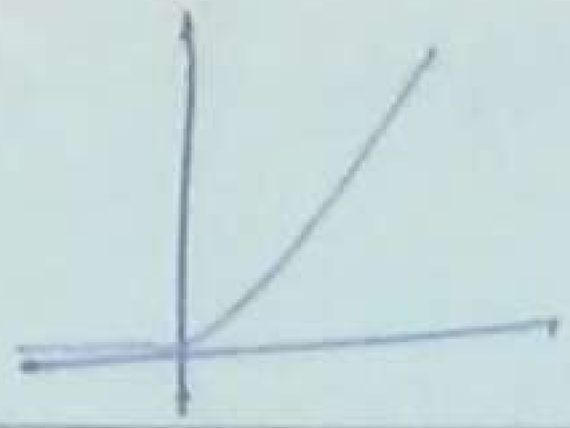


Large data and more complex NN led to the rise of deep learning.



sigmoid f^N

drawbacks:
Region A and B where the gradient is nearly zero, learning becomes really slow



Railer f^N
(value f^N of the rectified linear unit)
In this, gradient is 1 for all +ve values of input and is very less likely to gradually shrink to 0.

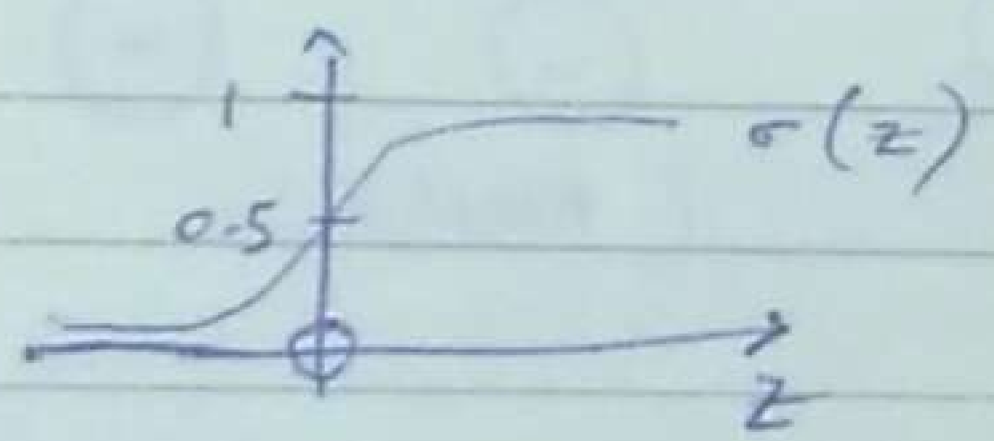
Hence, shifting from sigmoid to Relu has led to gradient descent work, much faster

→ <u>logistic regression</u>:

Given X, we want $\hat{y} = P(y=1|x)$

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

Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z}) = \dfrac{1}{1+e^{-z}}$ (So, $0 \le \hat{y} \le 1$)



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

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \leftarrow (\text{bias}) \\ \\ \} w \leftarrow \\ \\ \end{matrix} \quad [x_0 = 1] \text{ added}$$

(won't use this notation anywhere further in the course)

The model needs to be trained such that:
Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

$x^{(i)}, y^{(i)}, z^{(i)}$ denote notations for i^{th} example.

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

* though not usually used in logistic regression as it makes gradient

descent to not work well. So, we define

L as $-(y\log\hat{y} + (1-y)\log(1-\hat{y}))$ that gives us an optimize problem that is convex.

[we need convex bcoz such problems only have global minima]

$$\text{if } y=1, \quad L(\hat{y},y) = -\log\hat{y}$$
$$\text{if } y=0, \quad L(\hat{y},y) = -\log(1-\hat{y})$$

if $y=1$, we want $-\log\hat{y}$ to be as small as possible, i.e $\hat{y}$ to be as large as possible and $\hat{y}$ is sigmoid i.e it can be maximum 1 which we actually want.

if $y=0$, we want $-\log(1-\hat{y})$ to be as small as possible, i.e $\hat{y}$ to be as small as possible, as $\hat{y}$ is sigmoid, $\hat{y}$ can be minimum 0 which is actually our target.

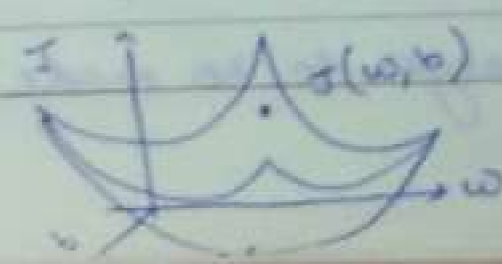
* Loss fn measures the error on a single training set, but cost fn i.e J measures how well we perform on entire training set.

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

cost of our parameters

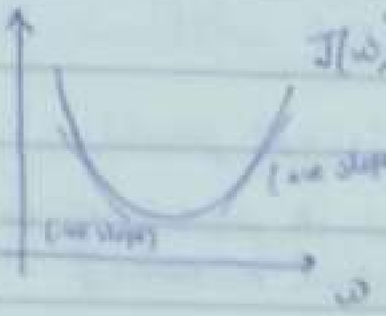
So, while training logistic regression model, we try to find w and b that minimize the overall cost fn J.

$\boxed{\text{logistic regression} \equiv \text{very very small NN}}$



convex fn, w and b are randomly initialized to some value and applying gradient descent, the lowermost point is achieved

- Gradient Descent : It moves in the dir^n of steepest slope as quickly downhill as possible. It takes a step in 1 Iteration and another in next itera^n and so on.



Repeat {

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

}

(bcoz if slope is +ve, ω will ↓ while coming down and if slope is -ve, ω will ↑)

for $J(\omega, b)$ →

$$\omega = \omega - \alpha \frac{\partial J(\omega, b)}{d\omega}$$

$$b = b - \alpha \frac{\partial J(\omega, b)}{db}$$

F/w propaga^n pass helps calculate the value of J and b/w propaga^n pass helps calculate the derivative of J and get final value of o/p.

$\frac{dJ}{da} = 3$ $a = 5$

$\frac{db}{dJ} = 6$ $b = 3$

$c = 2$

$$\boxed{u = bc} \rightarrow \boxed{v = a + u} \rightleftarrows \boxed{J = 3v}$$

(backpropaga^n)

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

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

$\frac{du}{db} = 2$

$\frac{dJ}{db} = 6$

16/3/18 logistic regression gradient decent

x_1, w_1, x_2, w_2, b →

$$\boxed{z = w_1 x_1 + w_2 x_2 + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{L(a, y)}$$

$$\frac{dL}{dz} = \frac{dL(a,y)}{dz} = \boxed{a - y}$$

$$\frac{dL(a,y)}{da} = \frac{-y}{a} + \frac{-y+1}{1-a} = \frac{a-y}{a(1-a)}$$

$$\frac{\partial L}{\partial w_1} = X_1 \cdot dz \qquad dw_2 = X_2 \cdot dz \qquad db = dz$$

$$\begin{cases} w_1 = w_1 - \alpha\, dw_1 \\ w_2 = w_2 - \alpha\, dw_2 \\ b = b - \alpha\, db \end{cases}$$

Through b/w propogation we get to know the changes to be made in w_1, w_2 and b for a closer output.

Note Vectorization techniques help us get rid of explicit full loops in our code to go through the entire training set.

Non-vectorized implementation

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

Vectorized implementation

$$z = \underbrace{np.dot(\omega, x)}_{\omega^T X} + b$$

(much faster than non-vectorized implementation)

GPU, CPU } single instruc", multiple data.

Note Always avoid using for loops whenever possible.

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

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

vectorized.

```
import numpy as np
u = np.exp(v)
```

- vectorizing logistic regression:

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

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

vectorized representa"

So, $Z = \omega^T X + b$

i.e $[z^1 \; z^2 \; ... \; z^m] = \omega^T X + \underset{(1\times m)}{[b \; b \; ... \; b]}$

$= [\omega^T x^{(1)} + b \;\; \omega^T x^{(2)} + b \; .. \; \omega^T x^{(m)} + b]$

$A = [a^1 \; a^2 \; ... \; a^m] = \sigma(Z)$ { no looping needed }.

$dZ = A - Y$

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

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

Q. Calories from carbs, Proteins, fats in 100g of different foods:

$$\begin{array}{l} \\ \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} \begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \\ \end{array}$$

$$\begin{array}{l} \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} = \begin{array}{c} A \\ (3,4) \end{array}$$

Calculate % of calories from carb, protein, fats without using a for loop.

Ans. cal = A.sum(axis=0) // python will sum vertically.

>> [59. 239. 155.4 76.9]

%age = 100* A / cal.reshape(1,4)

(not necessary in this case as matrix is already (1,4))

→ Broadcasting example:

i) $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \equiv \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \underbrace{\begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}}_{\text{(auto expansion)}}$ (in python) $= \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$

ii) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}$

iii) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix}$

General principle: $(m,n) \begin{smallmatrix} + \\ - \\ * \\ / \end{smallmatrix} (1,n) \approx (m,n) \begin{smallmatrix} + \\ - \\ * \\ / \end{smallmatrix} (m,n)$

$(m,n) \begin{smallmatrix} + \\ - \\ * \\ / \end{smallmatrix} (m,1) \approx (m,n) \begin{smallmatrix} + \\ - \\ * \\ / \end{smallmatrix} (m,n)$

Note a = np.random.randn(5). Here 'a' is a rank 1 array in python and is neither a column nor a row vector

a = a.T and np.dot(a,a.T) is a single number

a.shape = (5,) ← (Imp)

while, $\boxed{a = np.random.randn(5,1)}$ is a $(5,1)$ column vector and now a^T is a $(1,5)$ row vector and np.dot(a, a.T) is $(5,5)$ vector.

Note vector is contained in two square brackets - [[. .]]
array " " " 1 " " - [. . .] }**

To avoid making rank 1 arrays, rather make vectors or matrices as their behaviour is easy to understand.

If sometimes, we end up with a rank 1 array, we can always reshape it to a vector as a = a.reshape((5,1)).

assert (a.shape == (5,1)) {use such statements to assert anything}

COURSE 1 (Week 2)
Programming Assignment FAQ
B, D, B, C, D, A, C, D, A, B

Note In ML or DL we don't use math library because it takes real number inputs. In ML we generally deal with arrays and so numpy library is needed.

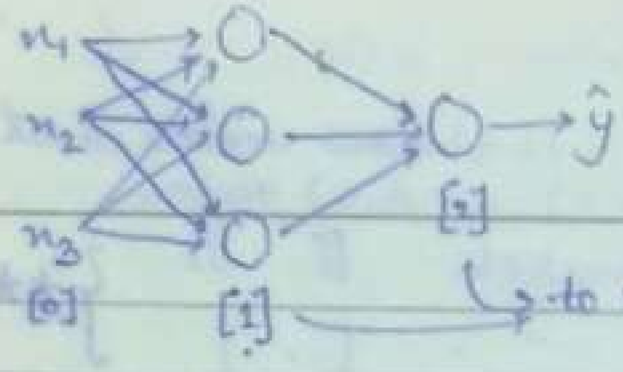
Note gradient of sigmoid fn w.r.t input $x = \sigma(x)(1-\sigma(x))$

Note $\text{softmax}(x) = \text{softmax}([x_1 \; x_2 \; \cdots \; x_n]) = \left[\frac{e^{x_1}}{\sum_j e^{x_j}} \; \frac{e^{x_2}}{\sum_j e^{x_j}} \; \cdots \; \frac{e^{x_n}}{\sum_j e^{x_j}}\right]$

Neural Network:

$x_1, x_2, x_3 \rightarrow \bigcirc \rightarrow \hat{y} = a$

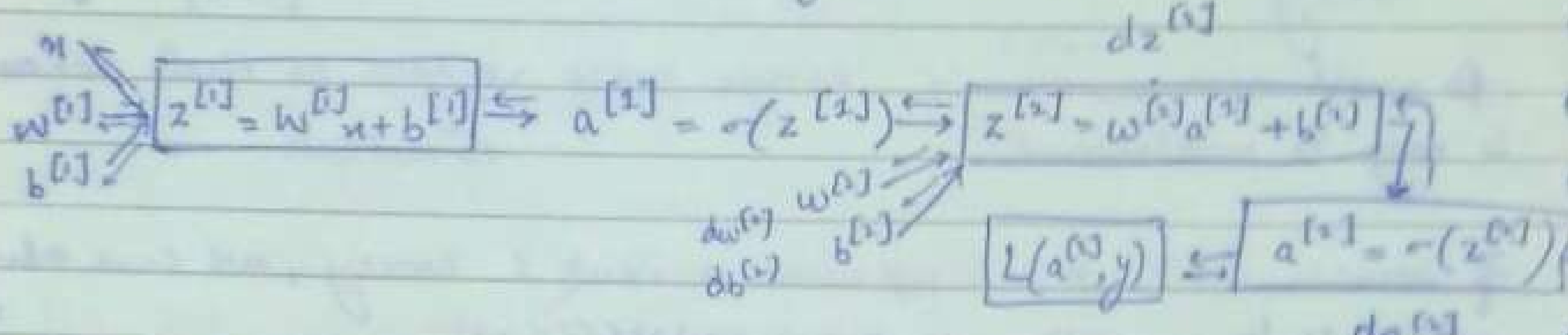
$x, w, b \rightarrow \boxed{z = w^T x + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{L(a,y)}$



$\hookrightarrow$ to represent layer 1 and 2.

Note $X = a^{[0]}$

{combination of many sigmoid fn give a neural network?}



So, backward calculations are done in neural network for updating weights and biases.

[0] : Input layer (input features)

[1] : Hidden layer (Things in hidden layer are not seen in the training set)

[2] : Output layer

$$a^{[1]} = \begin{bmatrix} a^{[1]}_1 \\ a^{[1]}_2 \\ a^{[1]}_3 \end{bmatrix}$$ (for hidden layer in above case)

We don't count the i/p layer in NNs and so the above example is a 2-layered NN. The hidden layer has parameters w and b associated with it ($w^{[1]}$, $b^{[1]}$)

$w^{[1]} \to (3,3)$, $b^{[1]} \to (3,1)$ → as we have 3 i/p feature and 3 hidden layer nodes.

Similarly o/p layer also has $w^{[2]}$, $b^{[2]}$ associated with it.

$w^{[2]} \to (1\times3)$, $b^{[2]} \to (1\times1)$ {as 1 o/p layer nodes and the layer 1 has 3 o/p features to layer 2.}

Now, vectorized implementation of this can be written as:

Given i/p x:

$$\left.\begin{aligned} z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \end{aligned}\right\} \text{output of layer 1}$$

$$\left.\begin{aligned} z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) = \hat{y} \end{aligned}\right\} \text{output of layer 2.}$$

Now, we need to compute outputs on all the examples on pretty much all at the same time. If we have $\underline{m}$ training examples instead of 1,

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

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

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

$$\vdots$$

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

So, $a^{[2](i)}$ — (i) refers to training eg. i; $[2]$ refers to the 2nd layer.

So, for i = 1 to m:

$$\left.\begin{aligned} z^{[1](i)} &= W^{[1]} x^{(i)} + b^{[1]} \\ a^{[1](i)} &= \sigma(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2]} a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) \end{aligned}\right\} \xrightarrow{\text{vectorizing}} \text{this, we get:}$$

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

So,

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

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

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

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

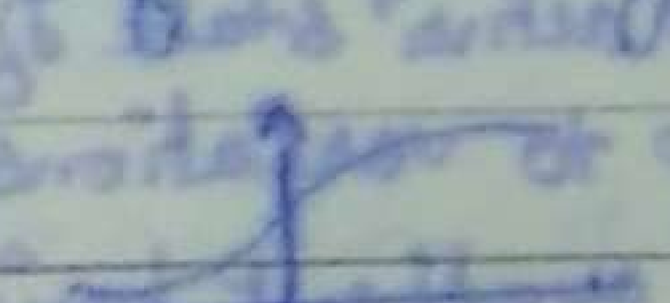
$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix} = W^{[1]} X + b^{[1]}$$

vertically, we have nodes in the layer (i.e the different hidden units)
horizontally is the no. of training examples.

Note $X = A^{[0]}$

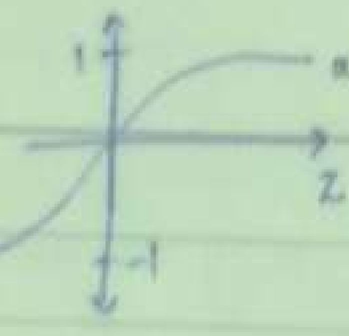
⇒ Activa^n fn [of which sigmoid is just an example]

Sigmoid fn:



activa^n fn almost always work better than sigmoid fn and can be non linear

$\boxed{a = \tanh(z)}$ instead of $\sigma(z)$ is superior activa^{n} f^{n} for sure



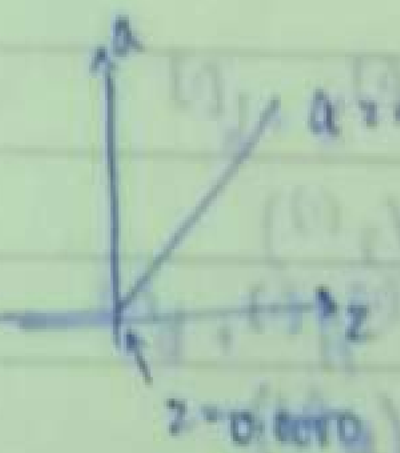
$a = \tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ (mathematically a shifted version of the sigmoid f^{n})

advantage.

(mean of data comes closer to zero and hence easier to analyse) while in case of σ (sigmoid f^{n}) data is centered about around 0.5 which makes learning for the next layer a little bit hard.

Note activa^{n} f^{n} can be different for different layers. sigmoid f^{n} used in o/p layer when classifica^{n} is to be done, as the value varies b/w 0 & 1 and not -1 and 1 like the tanh f^{n}.

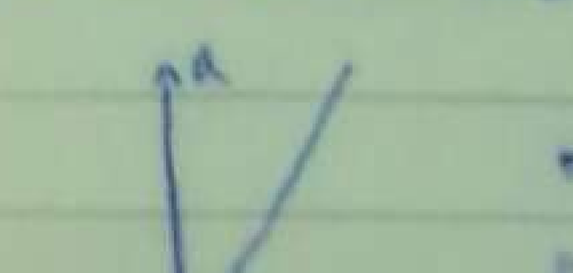
Drawback of these 2 activa^{n} f^{n} is that when z is very large or very small, the slope of the f^{n} ends up being close to zero & so this can slow down gradient decent.



= ReLU (Rectified Linear Unit)

[Network learns much faster & ReLU as compared to sigmoid or tanh]

for o/p layer, sigmoid is used by default in case of binary classifica^{n} only. In all other cases generally ReLU is used nowadays or tanh is also sometimes used.



= leaky ReLU. to avoid problem of dying ReLU.

bcoz in dying ReLU, those neurons which entered the 0 activa^{n} state of ReLU, will stop responding to variations in error/input bcoz of gradient being 0 in that state.

→ Need for non-linear activa^n f^n:

$a = g(z) = z$ (sometimes called linear activa^n f^n) or 'identity activa^n f^n.

⇓

This means our model is computing $\hat{y}$ has as a linear f^n of the input features i.e $a = z = wx + b$. So, no matter how many hidden layers are there in the NN, always a linear f^n of the i/p is calculated hence a hidden layer (linear) is more or less useless, bcoz even combina^n of linear f^n is a linear f^n, leading to no new f^n being made in the hidden layers to understand the i/p - o/p rela^n.

It might be okay to have a linear activa^n f^n if y is a regression variable, so one can use linear activa^n in the o/p layer if y is to be obtained in b/w $-\infty$ to $+\infty$ (i.e regression). e.g in housing prices example, ReLU can also be used bcoz all prices are +ve and lie b/w 0 to ∞.

⇒ Derivatives of activa^n f^n:

slope of $g(z)$ or activa^n f^n.

① Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$, So, $\frac{d}{dz} g(z) = \frac{1}{1+e^{-z}}\left(\frac{e^{-z}}{1+e^{-z}}\right) = g(z)(1-g(z))$

i) $z = 10 \Rightarrow \frac{dg(z)}{dz} \simeq 0$

ii) $z = -10 \Rightarrow$ " $\simeq 0$

iii) $z = 0 \Rightarrow$ " $\simeq 1/4$

② tanh: $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$, So, $g'(z) = 1-(\tanh(z))^2$

i) $z = 10$, $\tanh(z) \approx 1 \Rightarrow g'(z) \simeq 0$

ii) $z = -10$ $g'(z) \approx 0$

iii) $z = 0$ $g'(z) \approx 1$

③ ReLU: $g(z) = \max(0, z)$, So, $g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$

④ Leaky ReLU: $g(z) = \max(0.01z, z)$, So $g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$

⇒ Gradient decent for NN:

We will know equations in order to get back propogation of the gradient decent working.

e.g. $n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

$n^{[0]}$ ↓ i/p features

$n^{[1]}$ ↓ hidden layer units

$n^{[2]}$ ↓ o/p units

Parameters: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$ (for a single hidden layer)

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

Cost fn: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} \underbrace{L(\hat{y}, y)}_{\text{loss fn}}$ (for binary classification)

L is same as that used in logistic regression.

Gradient decent: (to learn parameters for NN)
↓
(to train the parameters)

Repeat {

Compute predic^n ($\hat{y}^{(i)}$, $\hat{y}^{(i)}$ for i=1 to m)

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...

$w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$

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

⋮

}

Eq:

Feed Propoga^n:

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

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

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

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

Back propog^n:

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

$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$

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

np.sum ↓ for summing matrix horizontally

(to ensure that o/p vectors are of form $(n,1)$ & not $(n,)$)

Back propoga^n contd...

$$dz^{[1]} = \underbrace{w^{[2]T} dz^{[2]}}_{(n^{[2]},m)} \underset{\text{element wise product}}{*} \underbrace{g^{[1]'}(z^{[1]})}_{(n^{[1]},m)} = da^{[1]} * g^{[1]'}(z^{[1]})$$

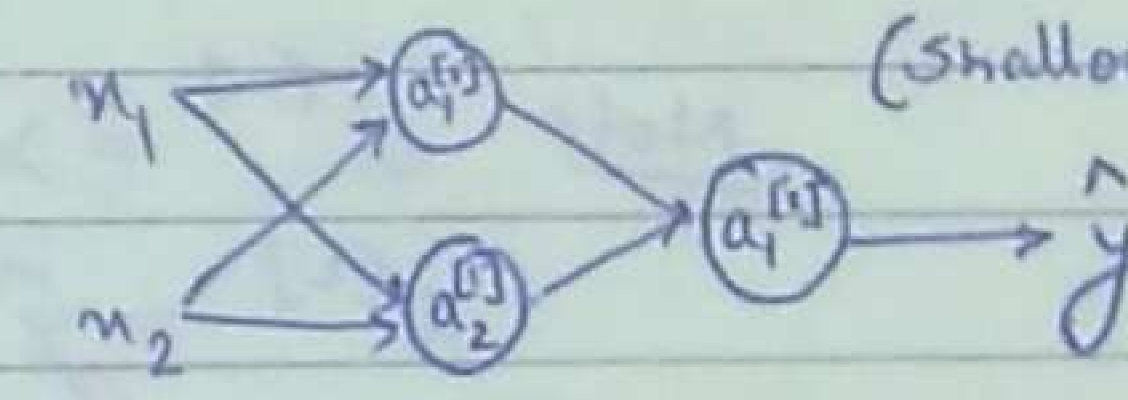
$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$

$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T \quad , \quad dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$

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

Note: Initializing parameters not to zero but randomly turns out to be very important for training neural network. Initializing to zero would work for logistic regression but not for NN, while applying Gradient Decent bcoz:

e.g. (Shallow NN)



$n^{[0]} = 2 \quad n^{[1]} = 2$

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

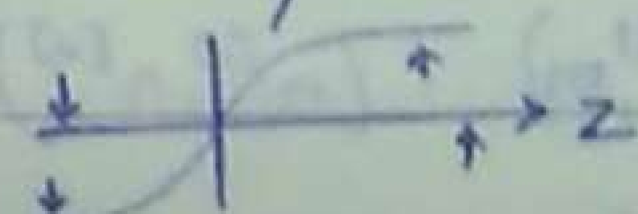
If this is the case, $a_1^{[1]} = a_2^{[1]}$ bcoz both the hidden units are calculating exactly the same f^n.
Also, $dz_1^{[1]} = dz_2^{[1]}$ while back propoga^n. As $a_1^{[1]} = a_2^{[1]}$, outgoing weights also seem to be equal. i.e $w_2 = [0\ 0]$. So, after every itre^n we will see, both the hidden units are computing exactly the same f^n, and are symmetric. Hence, there is no point having more than 1 hidden unit with weights initialized to zero.

Solu^n is to initialize the parameters randomly.

$w^{[1]} = np.random.randn((2,2)) * 0.01$

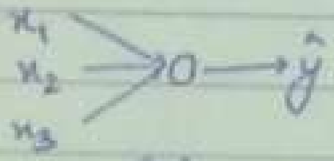
$b^{[1]} = np.zero((2,1))$

Note: bcoz b doesn't have symmetry problem, as long as z is initialized randomly.
we multiplied $w^{[1]}$ by 0.01 bcoz we initialize weights with very very small values (random) bcoz otherwise $a^{[1]} = g(z^{[1]})$ will have very small gradient or slope.



as $z^{[1]}$ is very large or very small for large $w^{[1]}$, meaning learning will be very slow for less $da^{[1]}$, as long as tanh or sigmoid f^n are used in the NN.

* Deep NN: (to solve percep^n problems in real world).

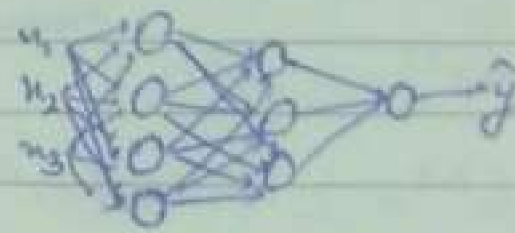


logistic regression
(very shallow model)

1 layer NN

1 hidden layer
(shallow NN)

2-layer model



2 hidden layer
(Deep NN)

3-layer model

(bcoz we don't include i/p layer as a layer)

No of layers = L

No. of units/nodes in layer l = $n^{[l]}$

Note $a^{[0]} = x$

$a^{[L]} = \hat{y}$

activa^n in layer l = $a^{[l]} = g^{[l]}(z^{[l]})$

$w^{[l]}$ = weights for computing $z^{[l]}$.

general fwd propoga^n algo for DNN: (considering vectorial representa^n)

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

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

→ computing activa^n for all layers. $l = 1$ to L {for loop needed to calculate all activa^n}

Note $\underset{(n^{[1]},m)}{Z^{[1]}} = \underset{(n^{[1]},n^{[0]})}{W^{[1]}} \overset{(n^{[0]},m)}{X} + \underset{(n^{[1]},1)}{b^{[1]}}$

(and similarly for next layers)

→ broadcasting will lead this to $(n^{[1]},m)$

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

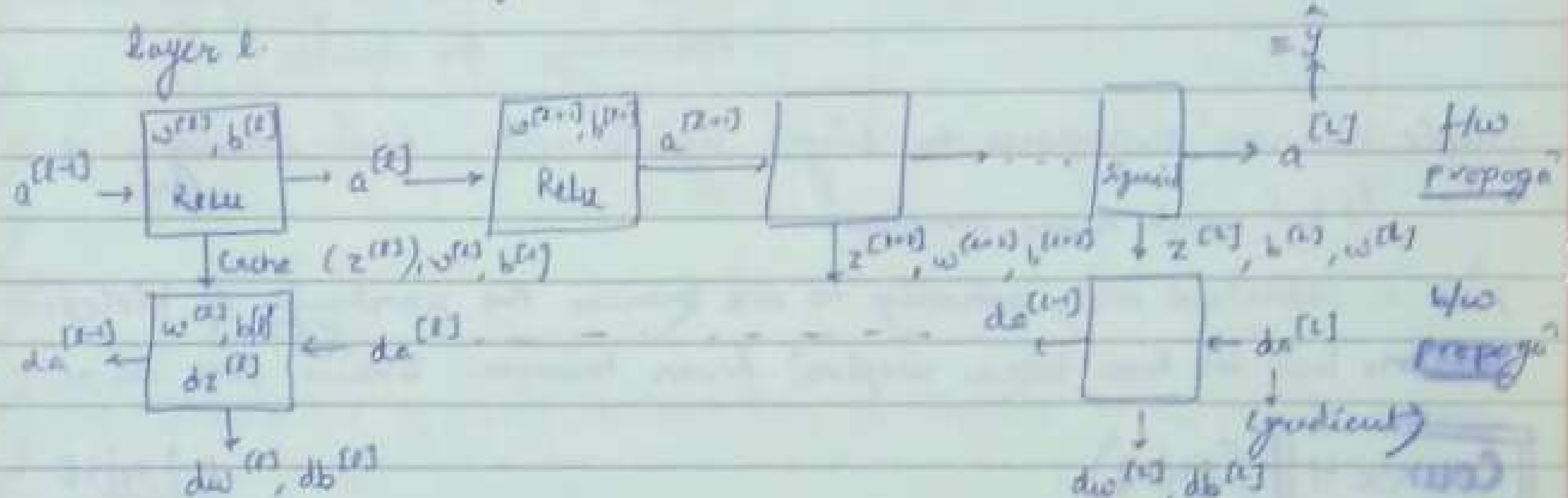
$db^{[1]} = (n^{[1]}, m)$

for m input training examples.

→ Why deep representations?

Ans Earlier layers of NN detect simple fⁿ and they are composed together in the later layers of NN, so as to learn more & more complex fⁿ. This type of simple to complex heirarchial representaⁿ is used in images, speech recognition.

If we try to compute a fⁿ with lesser hidden layers then we will need to have exponentially more hidden units in it.



Through this b/w step, we get all derivative values and w can get updated as $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$
and $b^{[l]} = b^{[l]} - \alpha db^{[l]}$. This comprises of 1 iteraⁿ of gradient descent for NN.

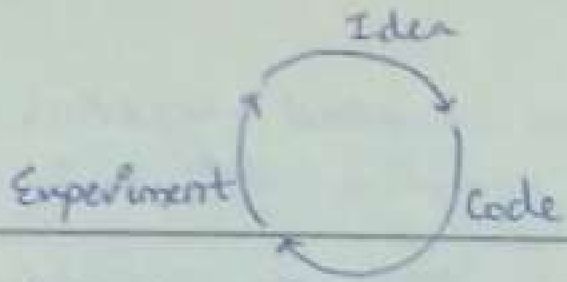
cache: pass info from one to the other so that it can be used during back propagaⁿ.

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots$

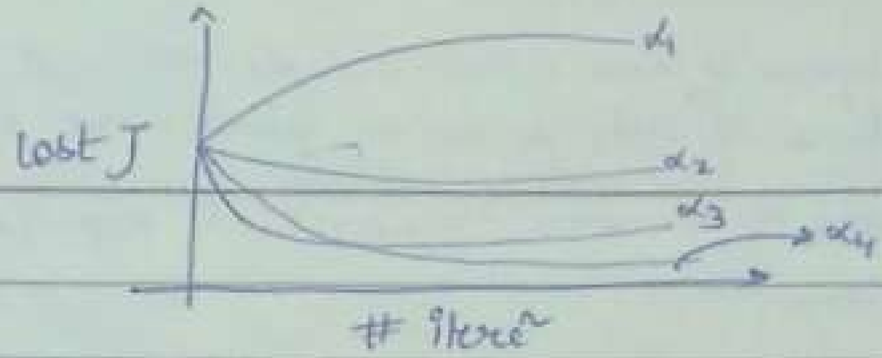
Hyperparameters: Learning rate $\underline{\alpha}$, # hidden units $(n^{[1]}, n^{[2]}, n^{[3]}, \ldots n^{[L]})$
iterations, # hidden layer L, choice of activaⁿ fⁿ.

Hyperparameters need to be given to the learning algorithm as they finally control the parameters W and B

More hyperparameters: momentum term, mini batch size, regularizaⁿ parameter



Applied DL is a very empirical process.



(α_4 Selected as cost values significantly c every iterⁿ)

In similar way other hyperparameter values are chosen, by trying and evaluating the results.

Single neuron analogous to single logistic unit. This analogy is
↳ but much more complex.
just a selective one, actually no one knows the working of biological neuron, but DL has taken inspiraⁿ from human brain for sure.