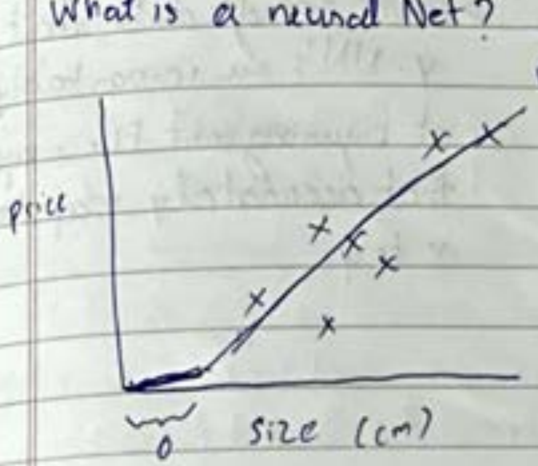
Deep learning Specialization

Course-1: Neural Networks and Deep learning

week-1

Deep Learning: → Training very large NN's

What is a neural Net?



Size $\xrightarrow{}$ O $\xrightarrow{}$ Price

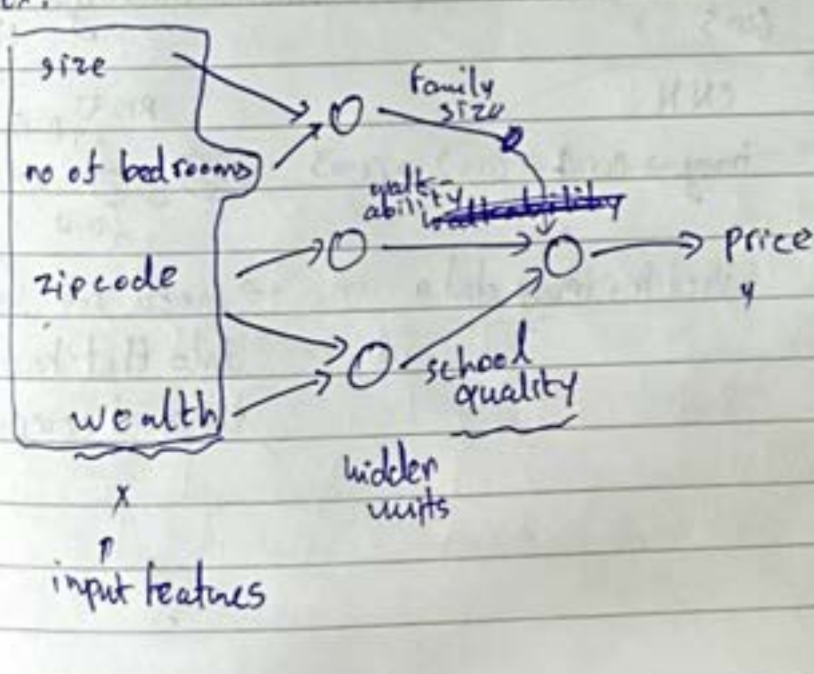
x (input) — neuron — y

- Neuron ~~this~~ implements the function we drew on the left.
- The neuron ~~computes~~ inputs the size and computes the linear function, takes max of 0 and then outputs the price

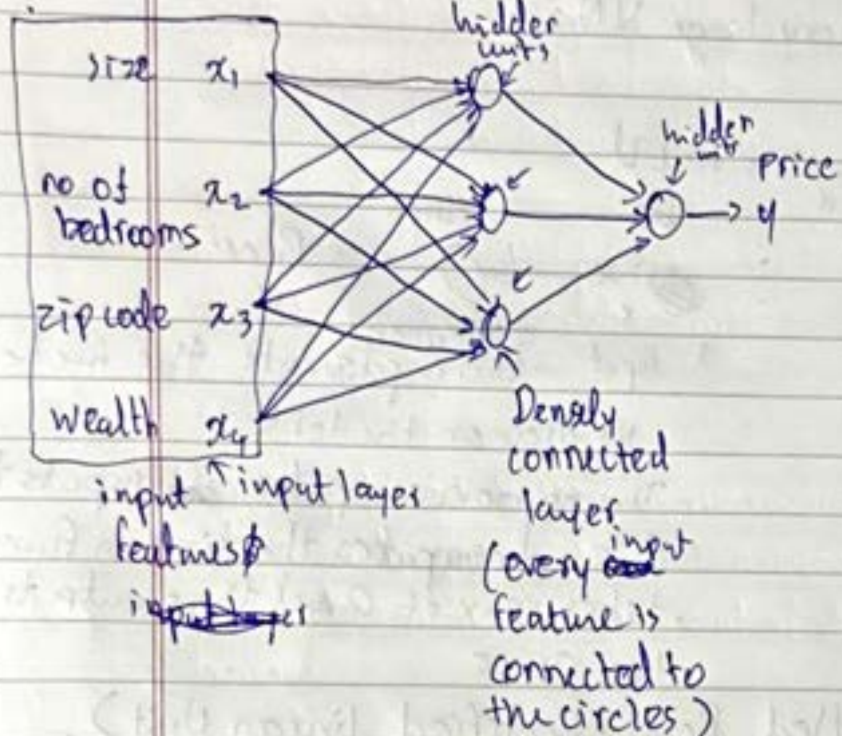
→ this function is called ReLU (Rectified linear Unit), the fn goes to 0 then takes off as a straight line. Rectify means taking max of 0 which is why we get a shape like this.

→ A single neuron NN is a very small NN, a large NN is formed by ~~stacking~~ taking many neurons and stacking them toghether.

ex.



Denotation



Hidden units take in input (x_1, x_2, x_3, x_4)

→ Given enough data about x and y. NN's are remarkably good at figuring out the ~~mapping~~ function that accurately map from x to y.

Supervised learning with NN

input (x)	output (y)	Application	
Home features	Price	Real Estate	Standard NN
Ad, user info	Click on ad? (0/1)	Online Advertising	Standard NN
Image	Object	Photo Tagging	CNN
English	Chinese	Machine Translation	RNN
Image, Radar Info	Position of other cars	Autonomous driving	Custom / Hybrid

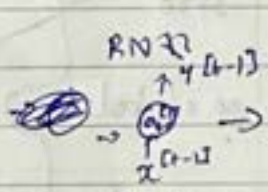
Standard NN

CNN

image → conv1 → conv2 → conv3

- Used for image data

RNN



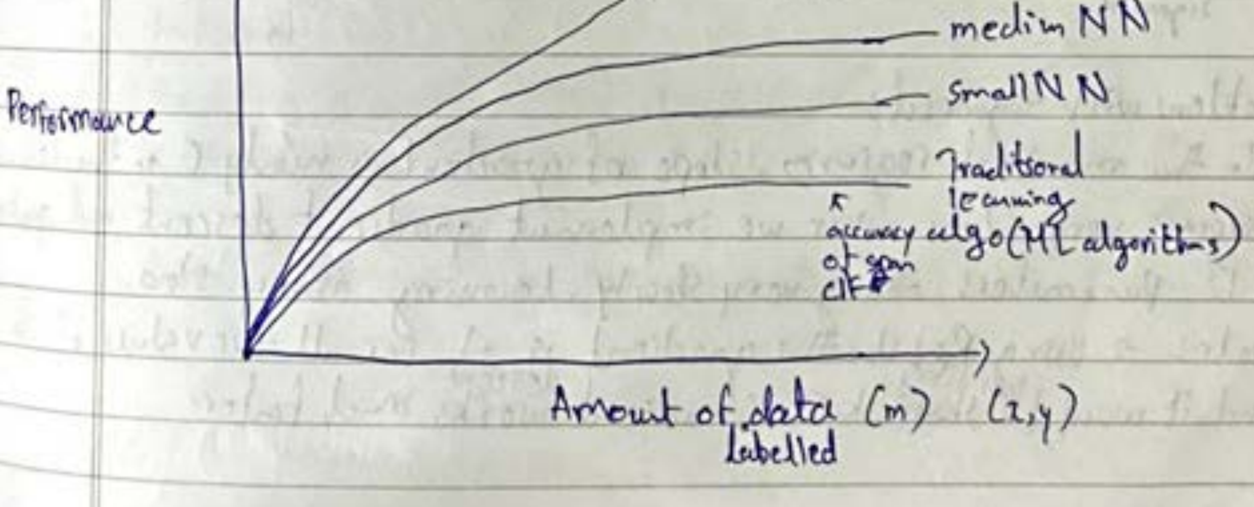
→ Good for 1d seq data that has a temporal component

Supervised Learning
we have applications of ML to both:
- Structured data
- Unstructured data

Structured data:
→ Databases of data
→ ex in housing price prediction, we have database or columns that tell us size of house, no of bedrooms
→ ex. whether or not you click on an ad, you might have info about the age, info about the ad and labels your trying to predict

Unstructured data:
→ Refers to audio, text, images, where you might want to recognise
• ~~you might~~ the text or image
→ features can be pixel values of image or individual words in a piece of text.
→ With rise of NN's and DL, computers are much better in interpreting unstructured data.

Why is Deep learning taking off?
Scale drives the deep learning process



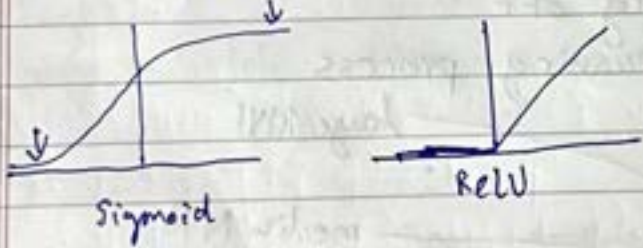
→ If we want a high level of performance, we need 2 things:
- You need to train a big enough NN to take advantage of the huge amount of data
- We need a lot of data, ~~scale~~

→ Scale has driven DL process. Scale refers to size of NN, just a neural network, a lot of hidden units, a lot of parameters, a lot of connections.

→ Best way to get better performance in a net:
- Train a bigger net or throw more data it only works up to a point because we run out of data or the net is too big to train.

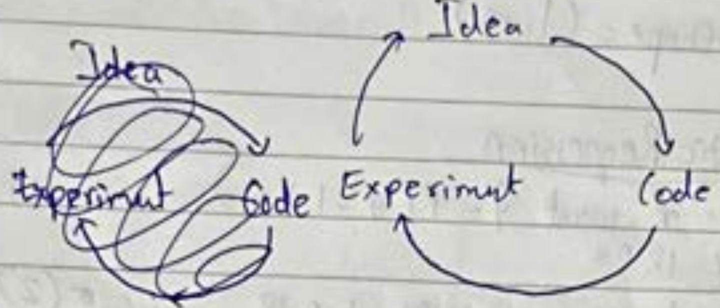
Scale drives DL progress:
→ Data
→ Computation
→ Algorithms

→ Breakthrough of NNs ex, switching from sigmoid to ReLU.



Problem with sigmoid:
→ In the marked regions, slope of gradient is nearly 0 so the learning becomes very slow when we implement gradient descent and gradient is 0, parameters change very slowly, learning is very slow
• Soln:- → Using ReLU. The gradient is +1 for all +ve values of i/p and it would less likely shrink 0. Gradient descent works much faster.

→ Fast computation is very important, training process of net is very intuitive.

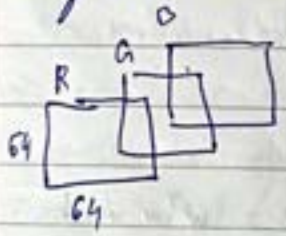


Week-2

Neural Network Basics

Binary Classification:

- we might have an image of cat, we want to classify whether its cat (1) or not cat (0) → y.
- An img is represented as 3 seperate matrices for Red (R), Green (G), Blue (B) color channels. ex, if we have 64 × 64 (height × width) img we have 3 R, G, B matrices of 64 × 64.



unroll into feature vec →

$$\underset{\text{feature vector}}{x} = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ \vdots \\ \vdots \\ 255 \\ \vdots \\ \vdots \end{bmatrix} \begin{matrix} \text{unroll} \\ \} \text{Red} \\ \\ \} \text{Green} \\ \\ \} \text{Blue} \end{matrix}$$

total dimension of vector $x = 64 \times 64 \times 3 = 12288$

$n = n_x = 12288$ n, n_x: dimensions of x

Notation

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

we will use m to denote a train ex: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})\}$

$m = m_{train}, m_{test}$

Putting all train ex in compact notation

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

$\longleftrightarrow m$

m : columns

n_x : rows

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

x.shape : (n_x, m)

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

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

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

Logistic Regression

Given x, want $\hat{y} = P(y=1 \mid x)$ $\quad z = w^T x + b$

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

Parameters : $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ $\quad \sigma(z) = \frac{1}{1+e^{-z}}$

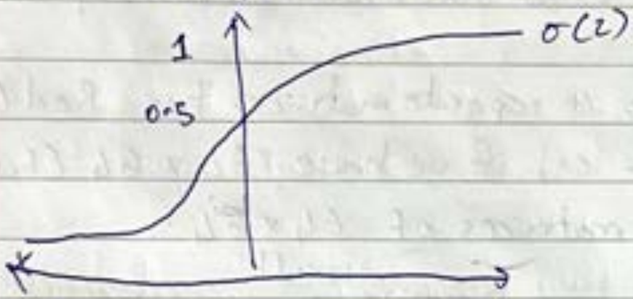
Output : $\hat{y} = \underbrace{\sigma}_{\text{sigmoid}}(\underbrace{w^T x + b}_{z})$

if z is large :

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

if z is a large negative no

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



Logistic Regression cost function

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

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

Loss (error) function : $L(\hat{y}, y) = -(y \log(\hat{y}) + (1-y)\log(1-\hat{y}))$

if $y=1$: $L(\hat{y}, y) = -\log(\hat{y})$ ← want $\log \hat{y}$ large, want $\hat{y}$ large

if $y=0$: $L(\hat{y}, y) = -\log(1-\hat{y})$ ← want $\log(1-\hat{y})$ large... want $\hat{y}$ small

- Loss function was defined wrt single training ex

Cost function : measures how are you doing on the entire train set

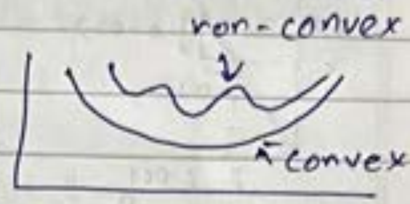
$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

$$= \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)}) \right]$$

→ Cost function is the cost of training the parameters (w, b). So in → ~~Using the~~ training the logistic regression model we will try finding parameters (w, b), that minimise overall cost function

Gradient Descent (GD)

* Recap on logistic regression,



$z = w^T x + b$

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

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

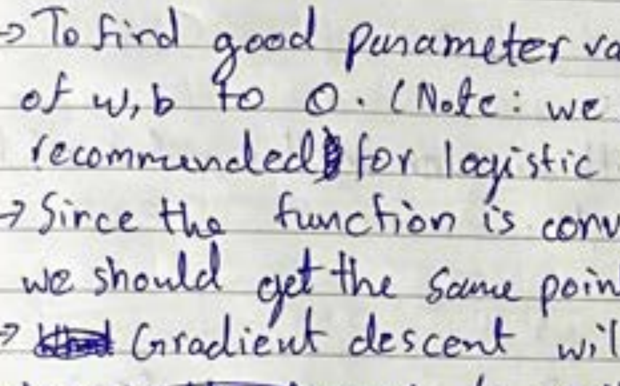
→ Find w, b that minimise $J(w,b)$
- $J(w,b)$ is a convex function

→ To find good parameter values we initialize the values of w, b to 0. (Note: we can use random initialization its not recommended) for logistic regression.

→ Since the function is convex, no matter where we initialize, we should get the same point or roughly the same point.

→ Gradient descent will start at the initial point and takes a step ~~in the steepest~~ downhill ~~direction~~ in the direction of steepest descent as quickly down as possible. This is one iteration of GD

→ After a few iterations we converge, close to the global optimum.



We repeatedly carry out this update:

repeat {

$w := w - \alpha \frac{\partial J(w,b)}{\partial w}$

$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$

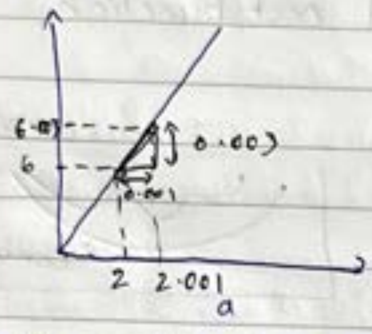
denotation in code

$\partial J(w,b)/\partial w \rightarrow dw \quad \partial J(w,b)/\partial b \rightarrow db$

α: is the learning rate it controls how big a step we take in each step of gradient descent

Derivatives

Intuition about derivatives :



$f(a) = 3a$ $\quad a = 2 \quad f(2) = 3(2) = 6$

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

- slope (derivative) of $f(a)$

$$\text{slope} = \frac{\text{height}}{\text{width}} = \frac{0.003}{0.001} = 3$$

at $a = 2$, slope $= 3$

⇒ slope $= 3$ denotes the fact that when we increase a by 0.001 ~~to the right~~.

⇒ The amount at $f(a)$ goes up 3 times as high as you increased it in the horizontal direction

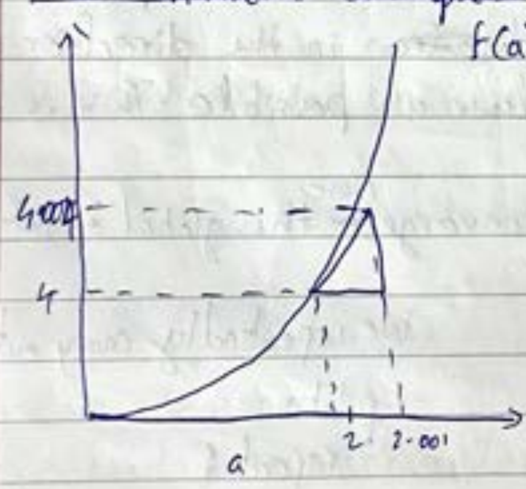
$a = 5 \quad f(a) = 15$

$a = 5.001, \; f(a) = 15.003$

$$\text{slope at } a = 5 = \frac{0.003}{0.001} = 3$$

→ Derivatives are defined with an even smaller value of how much you increase a to the right.

More derivative examples



$f(a) = a^2$

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

$a = 2.001 \quad f(a) = 4.004001$

slope at $a = 2$ is 4

$$\text{slope} = \frac{0.004}{0.001} = 4$$

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

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

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

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

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

More derivative examples

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

$a = 2.001$, $f(a) = 4.001$

$f(a) = a^3$, $\frac{d}{da} f(a) = 3a^2$, $a = 2$, $f(a) = 8$

$a = 2.001$, $f(a) = 8.012$

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

Computation Graph

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

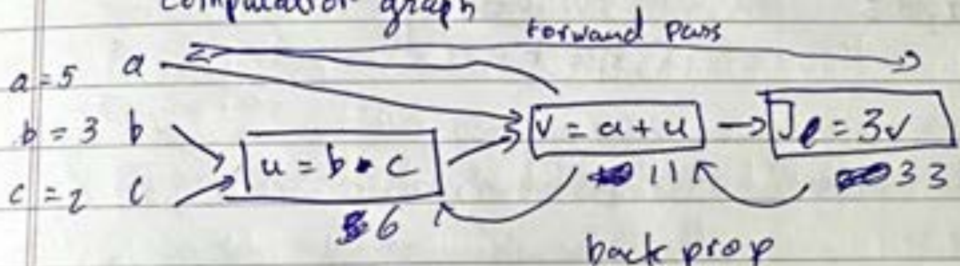
Forward pass : Used to compute output of neural net.

Back propagation : Used to compute gradients or derivatives

$u = bc$

$v = a + u$

$j = 3v$

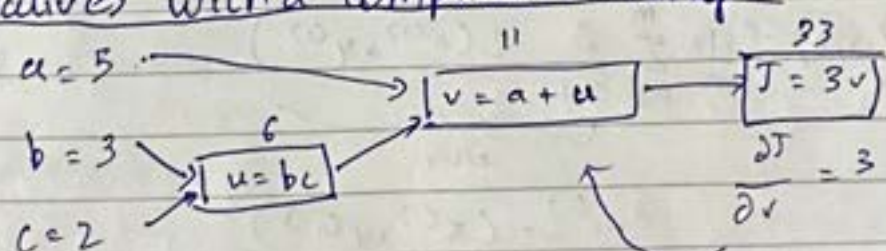


inputs : u, v, j

forward pass : left to right we compute value of j

- In order to compute derivates we go from right to left

Derivatives with a computation Graph



$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$

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

$= 6$

$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc}$

$= 3 \cdot b$

$= 9$

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

$\frac{\partial J}{\partial v} = 3$ $\frac{dJ}{da} = 3$ $\frac{dJ}{db} = 6$

$\frac{dJ}{dc} = 9$

$\frac{dJ}{du} = \frac{dJ}{\partial v} \cdot \frac{\partial v}{du} = 3 \cdot 1 = 3$ from $v = a + u$ $\frac{dv}{da} = 1$, $\frac{dv}{du} = 1$

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

~~Derivatives with a Compdat~~

Logistic Regression : Gradient Descent

$z = w^T x + b$

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

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

dz, da → are code variable names

$$x_1, w_1, x_2, w_2, b \rightarrow \boxed{z = w_1x_1 + w_2x_2 + b} \rightarrow \boxed{\hat{y} = a = \sigma(z)} \rightarrow \boxed{L(a,y)}$$

$da = \frac{dL(a,y)}{da}$ → da denotes the code representation

$$dz = \frac{dL}{da} \cdot \overset{a(1-a)}{\frac{da}{dz}}$$

$$= a - y$$

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

$$\frac{dL}{dw_1} = dw_1 = x_1 \cdot dz$$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

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

$$\frac{dL}{dw_2} = dw_2 = x_2\, dz$$

$$db = dz$$

Gradient Descent on m train examples

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

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

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

$$\partial w_1^{(i)} - (x^{(i)}, y^{(i)})$$

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$

for $i=1$ to m

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

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

$J += -[y^{(i)} \log(a^{(i)}) + (1-y^{(i)}) \log(1-a^{(i)})]$

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

$dw_1 += x_1^{(i)} dz^{(i)}$

$dw_2 += x_2^{(i)} dz^{(i)}$

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

$J /= m$

$dw_1 /= m,\ dw_2 /= m,\ db /= m$

→ Disadvantage with the above implementation:
- We are using 2 for loops to implement logistic regression.
  - ⇒ 1st for loop to iterate over all training ex
  - ⇒ 2nd for loop, the no of times we iterate is equal to the no of features (dw_1, dw_2)
  - → This a problem because when we use bigger datasets, its important to implement your algorithms without explicitely using for loops is important and it helps scale to larger datasets.

→ Solution: Vectorization, this speeds up your code and we get rid of for loops.

Python and Vectorization

~~Vectorization in logistic regression~~

ex, $z = w^T x + b$ $\qquad w = \begin{bmatrix} \vdots \end{bmatrix},\ x = \begin{bmatrix} \vdots \end{bmatrix} \qquad w \in \mathbb{R}^{n_x},\ x \in \mathbb{R}^{n_x}$

Non vectorized	Vectorized

`z=0`
`for i in range(n-x):`
`    z += w[i] * x[i]`
`z += b`
	$z = \underbrace{np.dot(w, x)}_{w^T x} + b$

-> Scalable DL implementations are done on GPU
-> GPU and CPU have parallelization instructions
-> They are called SIMD instructions (Single instruction multiple data)
-> SIMD: using built in functions or other functions such np.dot. we dont have to explicitely implement a for loop. It enables Python numpy to take advantage of parallelism to do the computation faster. These work great on GPU compared to CPU.
->

More Vectorization examples

NN programming guideline: avoid explicit for loops whenever possible

ex $u = Av$

$u_i = \sum_j A_{ij} v_j$

vectorized
u = np.dot(A, v)

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

Say you need to apply the exponential operation on every element of a matrix/vector.

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

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

vectorized
u = np.exp(v)
we can apply similar operations:
np.log(v)
np.dot(v)
np.maximum(v, 0)

classmate

Date ____
Page ____

Logistic Regression derivatives

$J=0$, $\boxed{dw1=0,\ dw2=0,}$ $db=0$

instead of initializing $dw_1, dw_2 = 0$ we can get rid of it and make dw a vector

dw = np.zeros((nx,1))

→ for i=1 to m:

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

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

we want to eliminate the 2nd for loop

$J += -[y^{(i)}\log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})]$

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

for i=1 to n_x dw gets updated

$dw_1 += x_1^{(i)} dz^{(i)}$

$dw_2 += x_2^{(i)} dz^{(i)}$

$n_x = 2$ instead of doing this we can do

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

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

$J = J/m$, $\boxed{dw_1 = dw_1/m,\ dw_2 = dw/m,}$ $db = db/m$

we can have: $dw /= m$

Vectorizing Logistic Regression

Forward propagation in non-vectorized form

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

$a^{(1)} = \sigma(z^{(1)})$ $\quad a^{(2)} = \sigma(z^{(2)})$ $\quad a^{(3)} = \sigma(z^{(3)})$

→ In order to carry the 4 propogation steps i.e, to compute preds on m train ex, there is a way to do it without using a single for loop

matrix of train i/p

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

First we will compute $z^{(1)}, z^{(2)}, \ldots$ in one step.

- Construct (1,m) matrix (row vector) while computing, $z^{(1)}, z^{(2)}, \ldots$

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

classmate

Date

Page

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

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

$$z = np.dot(w.T, X) + \underset{\mathbb{R}(1,1)}{b}$$

$\Rightarrow$ note that b is a real no, not a vector. When we add real no to vector Python automatically takes real no b and expands it to $(1 \times m)$ row vector. This operation is called Broadcasting in Python

Next we want to compute

$A = [a^{(1)} \; a^{(2)} \ldots a^{(m)}]$

Vectorizing Logistic Regressions Gradient Descent

$dz^{(1)} = a^{(1)} - y^{(1)}$, $dz^{(2)} = a^{(2)} - y^{(2)} \ldots$

$$\boxed{dZ = \underset{1 \times m}{[dz^{(1)} \; dz^{(2)} \ldots dz^{(m)}]}}$$

$\Leftarrow$ we can compute this at the same time with 1 line of code

$A = [a^{(1)} \ldots a^{(m)}]$ $\quad Y = [y^{(1)} \ldots y^{(m)}]$

$dZ = A - Y = [a^{(1)} - y^{(1)} \; a^{(2)} - y^{(2)} \ldots]$

From the prev implementation we got rid of 1 for loop but we still have,

$$\left.\begin{array}{ll} \text{for } j \to 1 \text{ to } n_x & \\ dw = 0 & db = 0 \\ dw \mathrel{+}= x^{(1)} dz^{(1)} & db \mathrel{+}= dz^{(1)} \\ dw \mathrel{+}= x^{(2)} dz^{(2)} & db \mathrel{+}= dz^{(2)} \\ \vdots & \\ dw \mathrel{/}= m & db \mathrel{/}= m \end{array}\right\}$$

nonvectorized

vectorized:

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

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

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

for $i=1$ to m:

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

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

$J += -[y^{(i)} \log(a^{(i)}) + (1-y^{(i)}) \log(1-a^{(i)})]$

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

$\left.\begin{array}{l} dw_1 += x_1^{(i)} dz^{(i)} \\ dw_2 += x_2^{(i)} dz^{(i)} \end{array}\right\} dw^{(i)} += x^{(i)} * dz^{(i)}$

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

$J /= m, db /= m, dw_1 /= m, dw_2 /= m$

for i in range(1000):

$Z = w^T x + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

$w := w - \alpha dw$

$b := b - \alpha db$

Broadcasting in Python

ex, Calories of carbs, proteins, fats in 100g of different food:

$$\begin{array}{c|cccc} & \text{Apples} & \text{Beef} & \text{Eggs} & \text{Proteins} \\ \hline \text{Carb} & 56.0 & 0.0 & 4.4 & 68.0 \\ \text{Protein} & 1.2 & 104.0 & 52.0 & 8.0 \\ \text{Fat} & 1.8 & 135.0 & 99.0 & 0.9 \end{array} \rightarrow A_{(3,4)} \quad (3,4)$$

Calculate the percentage of calories from carbs, proteins and fats if take apples. Can you do this without a for loop?

total no of calories = $56 + 1.2 + 1.8 = 59$

% of cal from carb = $(56/59) \times 100 = 94.91\%$

proteins = $(1.2/59) \times 100 = 2.03\%$

fats = $(1.8/59) \times 100 = 3.05\%$

Soln:- We will compute the column sum (we get total no of calories from each food)

- Next will divide each of 4 cols by their corresponding sum

Ans to the question

```
col_sum = A.sum(axis = 0)   # axis = 0, Python sums down vertically
                            #        1, sums horizontally
percentage = (A / col_sum.reshape(1,4)) * 100
```

$\underbrace{A}_{(3,4)} / \underbrace{\text{col\_sum.reshape}(1,4)}_{(1,4)}$

Broadcasting

$= (3,4) / (1,4)$

more examples,

ex₁,

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$$

Python will take 100 and auto expand it to a (4,1) vector

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

This type of broadcasting works for both col and row vectors

ex₂,

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(1,n)}{\begin{bmatrix}100 & 200 & 300\end{bmatrix}}$$

Python will copy matrix m times and turns it into (m,n) matrix

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

(m,n) ~ (2,3)   (1,n) ⇝ (m,n) (2,3).

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \begin{bmatrix}100\\200\end{bmatrix}$$

(m, 1) ⇝ Python copies n times horizontally ⇝ (m,n)

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(m,n)}{\begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix}} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$

General Principle to Broadcasting:

(m, n) matrix   + − * / row vec (1, n) ⇝ (m, n)

col vec (m,1) ⇝ (m, n)

(m, 1) + Real no ℝ

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + 100 = \begin{bmatrix}101 & 102 & 103\end{bmatrix}$$

A note on numpy vectors

Note:

generates 5 random no with $\mu=0, \sigma=1$

When you use np.random.randn(5), → Don't use

- we get a rank 1 vector and its shape is (5,)
  - ↳ This is not a row or column vector
  - ↳ a.transpose will be the same as a
  - ↳ dot product between a and a.transpose gives us a no. instead of matrix

Use

a = np.random.randn(5,1) → column vector

a = np.random.randn(1,5) → row vector

when we are not sure about dimensions of one of the vectors use assert statement.

assert(a.shape == (5,1))

Important functions in numpy

~~Explanation of logistic Regression cost function~~

- reshape: np.reshape((m,n)) reshapes an array
- norm: ~~np.linalg.~~norm Normalizes an array either row-wise or column wise. Normalizing data leads to better performance because gradient descent converges faster after normalization.
- While normalizing $\frac{x}{\|x\|}$ divide each row of vector by norm

$$x = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \end{bmatrix}$$

then $\|x\|$ = np.linalg.norm(x, axis=1, keepdims=True) = $\begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix}$

$$\|x\| = \begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix}$$

$$x_{norm} = \frac{x}{\|x\|} = \begin{bmatrix} 0 & 3/5 & 4/5 \\ 2/\sqrt{56} & 6/\sqrt{56} & 4/\sqrt{56} \end{bmatrix}$$

Note: we can divide matrices by diff sizes - Broadcasting

keepdims = True will broadcast correctly against original x

axis = 1, we will get the norm row-wise

axis = 0, we will get col wise norm

ord : type of norm (degree for root)

np.absolute : computes abs value for each element
np.sum: computes element wise sum

during the exercise
Note: when they ask to flatten array of shape (209, 64, 64, 3) [h w c]
to flatten this we are using:
$$X = X.\text{reshape}(X.\text{shape}[0], -1).T$$
-1 tells numpy to automatically calculate 2nd dimension which is: $64 \times 64 \times 3 = 12288$

transpose will change shape to (12288, 209)

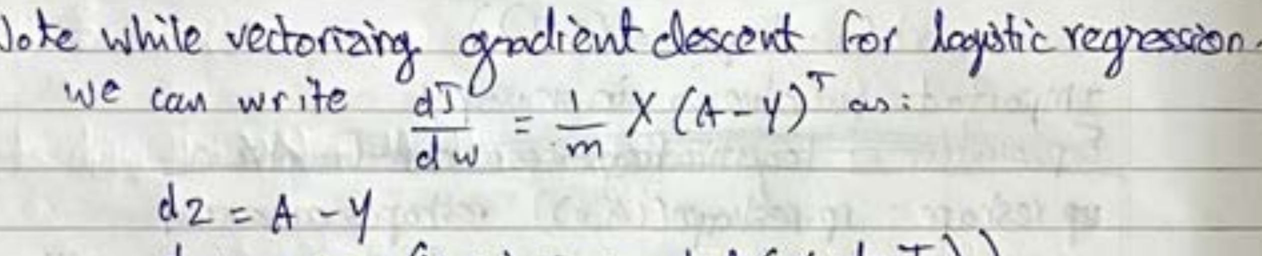
np.zeros(shape = (m,n)) : you get (m,n) arr of zeros

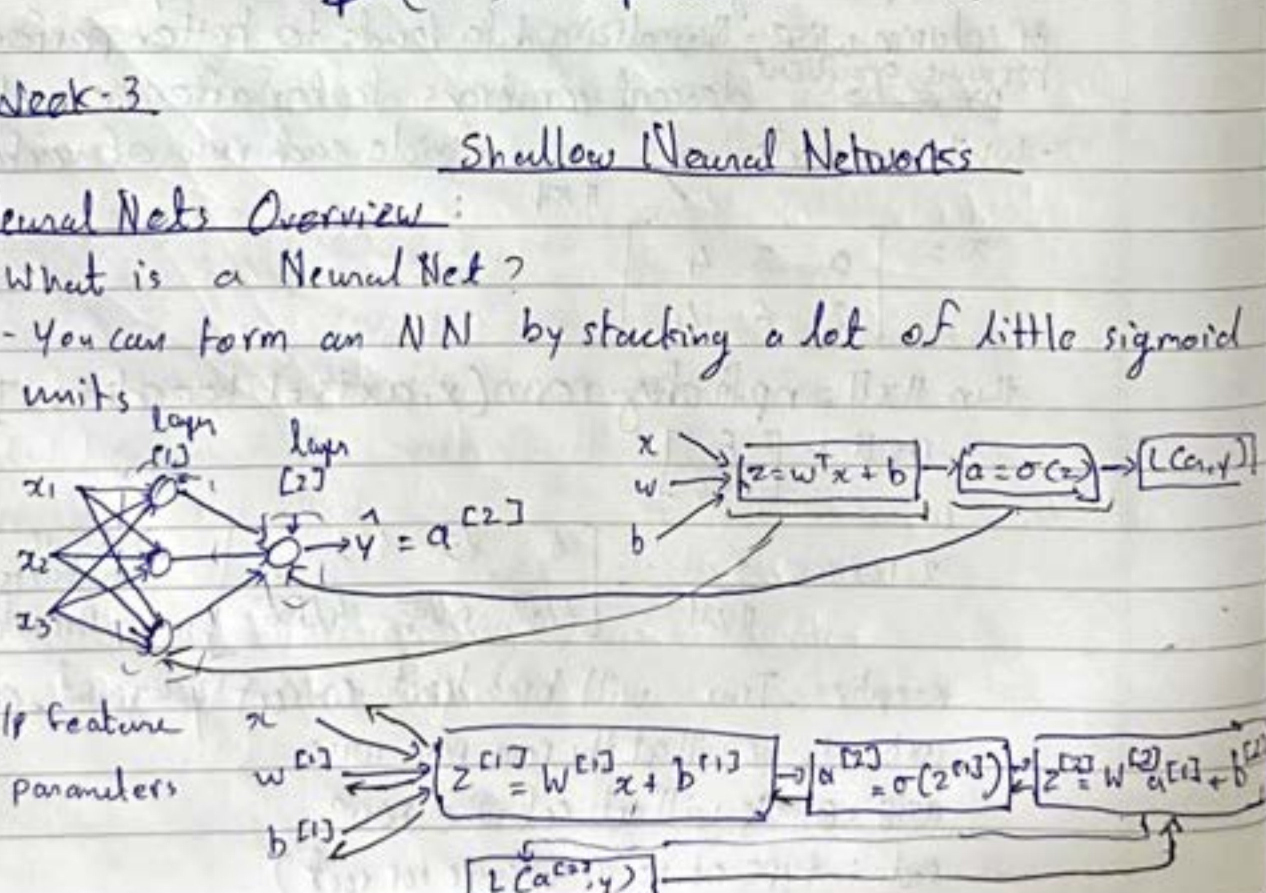
Note while vectorizing gradient descent for logistic regression.
we can write $\frac{dJ}{dw} = \frac{1}{m} X (A-Y)^T$ as:
$$dz = A - Y$$
$$dw = (1/m) * np.dot(x, dz.T))$$

<u>Week-3</u>

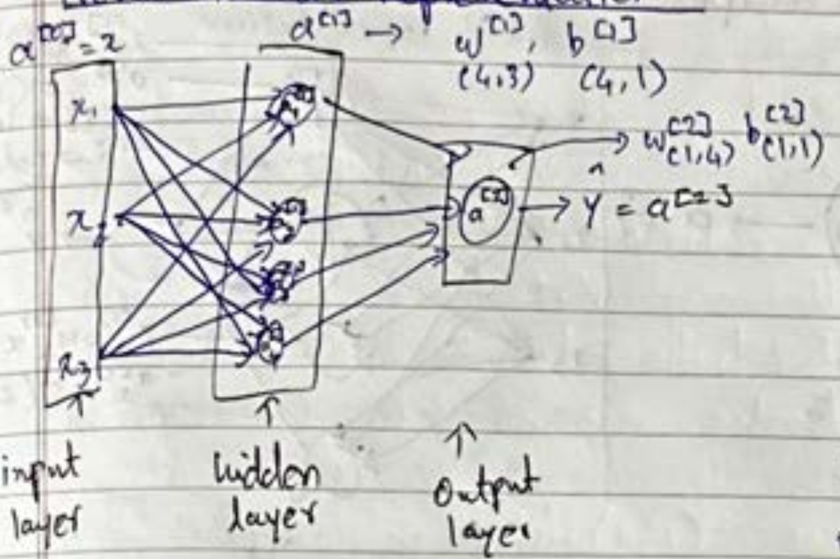
<u>Shallow Neural Networks</u>

<u>Neural Nets Overview</u>:
What is a Neural Net?
- You can form an NN by stacking a lot of little sigmoid units





Neural Network Representation



in logistic regression we used $\hat{y} = a$ because we only had one output. In NN we will use superscript square brackets to distuingish the layer it came from

Note: This is a 2 layer net. we dont count i/p layer, so we only have hidden and output layer.

representing the hidden layer

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix}$$ ← in this we got 4 hidden units in the first layer

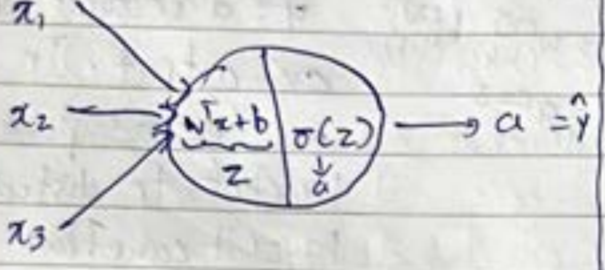
→ Hidden and output layers have parameters associated with them. Hidden layer is associated with param w, b.
$a^{[1]}$ has $w^{[1]}_{(4,3)}$, $b^{[1]}_{(4,1)}$, w is a $(4,3)$ matrix and b is $(4,1)$ vector

$w^{[1]} \rightarrow (4, 3)$ → We have 3 i/p features in input layer
↳ we have 4 nodes in hidden layer

→ Output layer $a^{[2]}$ is associated with parameters $w^{[2]}_{(1,4)}$, $b^{[2]}_{(1,1)}$
$w^{[2]} \rightarrow (1,4)$ dims $(1,4)$ because we got 1 node and 4 i/p features from prev layer (hidden)

Computing an NN's output : One hidden layer NN

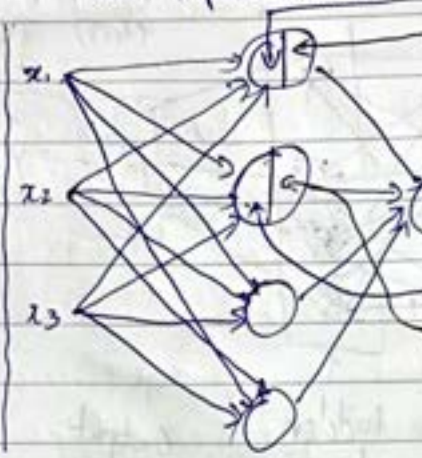
Logistic Regression



$z = w^Tx + b$

$a = \sigma(z)$

NN representation



$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}$

$a_1^{[1]} = \sigma(z_1^{[1]})$

$[l] \rightarrow$ layer

$a :\rightarrow$ node in layer

$z_2^{[1]} = w_2^{[1]}x + b_2^{[1]}$

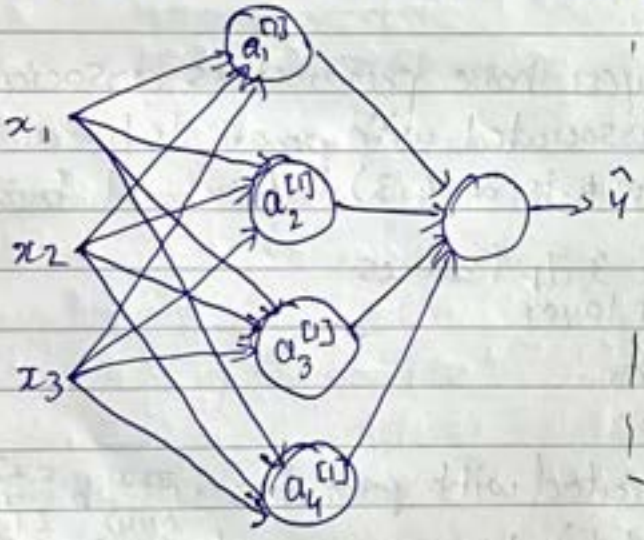
$a_2^{[1]} = \sigma(z_2^{[1]})$

- In a hidden layer node there are 2 steps of computation:

$z = w^Tx + b$

$a = \sigma(z)$

Lets draw the NN with the notation



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

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

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

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

We will vectorize these eqns because doing it with a for loop is inefficient

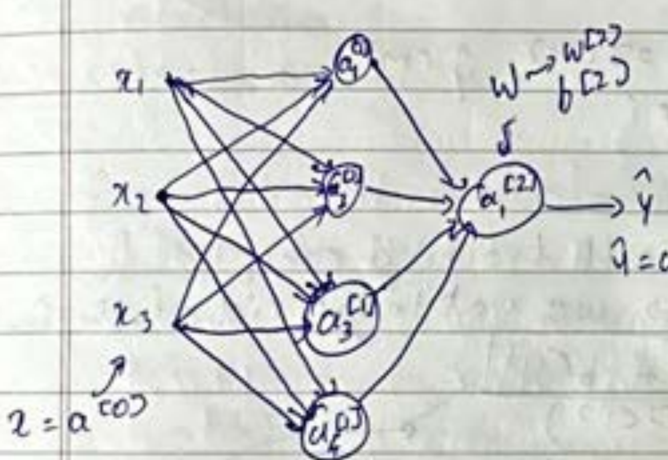
- → We will start by computing z as a vector
- → Stack the w's into matrix

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

→ We will stack a's toghter

$$a = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_1^{[4]} \end{bmatrix} \rightarrow \sigma(z^{[1]})$$

Clear representation of each layer and the dimensions of each layer



Given input x

$$\underset{(4,1)}{z_1^{[1]}} = \underset{(4,3)}{W^{[1]}} \underset{(3,1)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}}$$

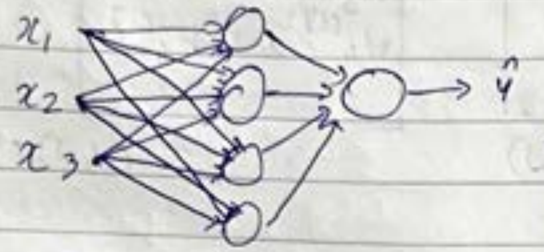
$$\underset{(4,1)}{a^{[1]}} = \sigma(\underset{(4,1)}{z^{[1]}})$$

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

→ We can think of the last unit as being analogous to logistic regression

Vectorizing across multiple examples

Aim: compute the outputs for all examples in NN at the same time



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

* These eqns tell how given ~~these eqn~~ an input feature x, you can use them to generate $a2 = y$ hat for a single train ex.

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

if we have m train ex we need to repeat the process

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

$a^{[2](i)} \rightarrow$ example i

$[2] \rightarrow$ layer 2

Computing predictions for all training examples

for $i = 1$ to m: $\rightarrow$ we wat to get rid of the for loop

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

X is matrix of train ex stacked in columns

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

(n_x, m)

classmate

Date ____
Page ____

We can do it similarly for Z, a as we did for X,

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

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

horizontal index corresponds to diff train ex, when we sweep left to right through train ex

vertical index corresponds to diff nodes in NN, as we scan down we index into hidden unit no.

$Z^{[1]} = W^{[1]}X + b^{[1]}$ $X \to A^{[0]}$, $x^{(i)} = a^{[0]}$, $x^{(i)} = a^{[0](i)}$ $\to W^{[1]}A^{[0]} + b^{[1]}$

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

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

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

To simplify,

train ex

hidden units

This notation is used for matrix X, Z as well

Explanation for vectorized implementation

Justification for vectorized implementation:

Forward propagation calculation for a few ex

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

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

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

col vector

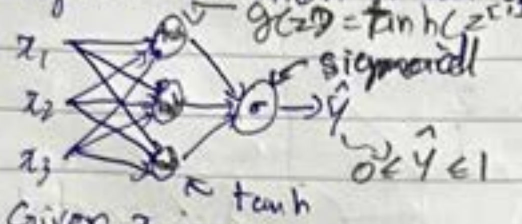
$$W^{[1]} \underbrace{\begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix}}_{X} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix}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

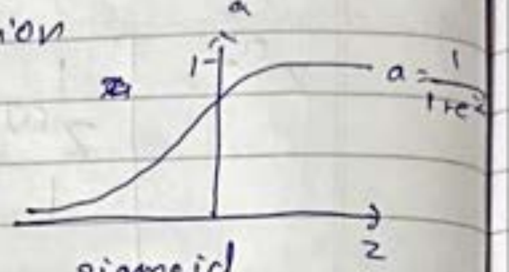
$W^{[1]}x^{(i)} = z^{[1](i)}$

$Z^{[1]} = W^{[1]}X + b^{[1]}$ - this line allows you vectorize all m examples at the same time

Activation Functions

→ Sigmoid function is an activation function





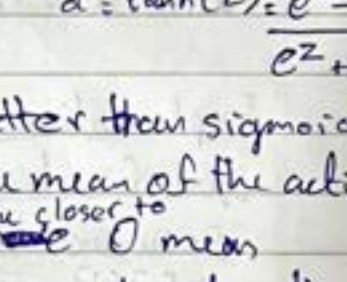
Given x:

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

→ We can have a different function $g(z)$ where g could be non-linear function that may not be a sigmoid function. A tangent or hyperbolic tangent (tanh) almost always works better than sigmoid.

→ tanh goes between -1 and +1.

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



→ If hidden units were tanh ~~it almost~~ its better than sigmoid because with values between -1 and 1, the mean of the activations that come out of your hidden ~~units~~ layers are closer to 0 mean

→ When training a learning algorithm, you might center the data and have your data have 0 mean using tanh instead of sigmoid

→ The effect of centering your data so, that the mean of your data is close to 0 rather than maybe 0.5; this makes learning for the next layer a little easy.

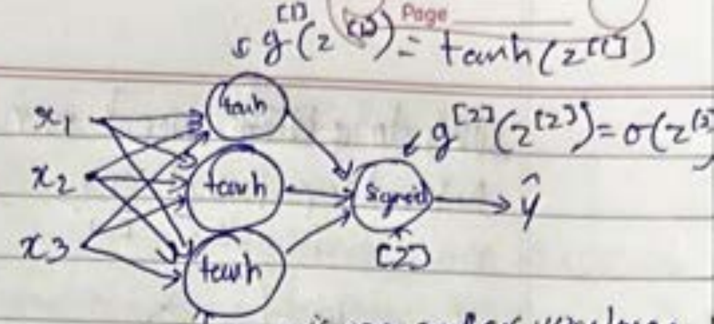
→ tanh is almost strictly superior.

→ One exception is for the output layer where it will be sigmoid because the output should be $0 \le \hat{y} \le 1$, while binary classification

→ We can have tanh activation fn for hidden units and a sigmoid fn for output layer.

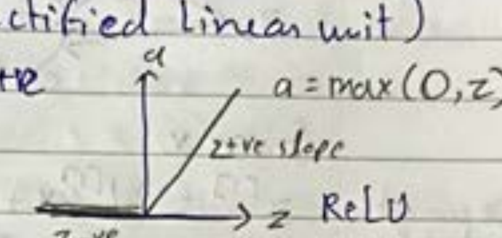
→ Sometimes activation functions can be different for different layers

$g^{[1]}(z^{[1]}) = \tanh(z^{[1]})$
$g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$



→ Downside of both sigmoid and tanh is if z is very small or very large, the gradient of derivative of the slope is very small. If z is very small it ends up being close to 0 and it slows down gradient descent. ~~An another~~

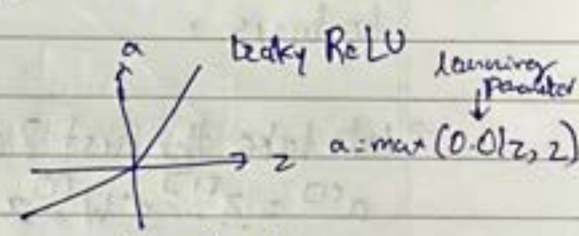
→ Another popular choice is ReLU (Rectified Linear unit)
- ~~then~~ The derivative is 1 as long as ~~slope~~ z is +ve
- Derivative is 0. when z is -ve



Rule of thumb for choosing activations func.
- if output is bet 0 and 1, we are using Binary classification then sigmoid is the choice for o/p layer.
- For all other units ReLU is the default choice. Even for hidden unit

- Disadvantage of ReLU: Derivative is equal to zero when z is -ve

Leaky ReLU:
- When z is -ve, instead of being 0 it takes a small gradient for -ve z values

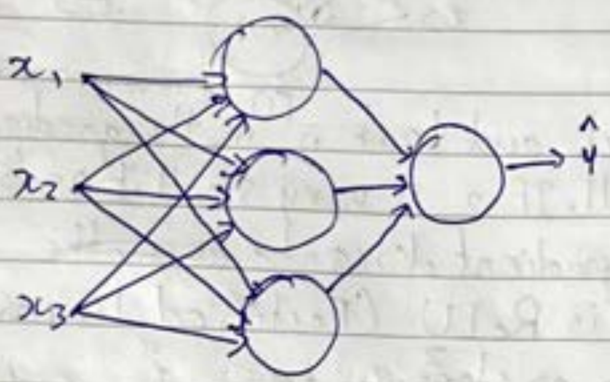


Note: Using either is fine. ~~Usually we ca~~ Use ReLU

Advantage of ReLU and Leaky ReLU: → A lot of space for z, the ~~derivative~~ slope of activation fn. is very different from 0.
→ Using ReLU the NN learns faster than tanh or sigmoid. The main reason is that there is less effect of slope of fn going down to 0 that slows down learning

Note: if your not sure about which activation fn to use try them all and evaluate them on a holdout validation set or development set.

Why do we need non-linear activation func ?
Activation functions



Given x

We can get rid of g and set $a^{[1]} = z^{[1]}$

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

$a^{[1]} = \cancel{g^{[1]}(z^{[1]})}\ z^{[1]}$ (we can say $g(z) = z$. This is called the linear activation fn / identity activation fn. Since it outputs whatever was the input)

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

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

- What if $a^{[2]} = z^{[2]}$?

⇒ The model is computing y or $\hat{y}$ as a linear function of the input features x

⇒ Lets take the first 2 eqns:

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

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

subs ① in ②

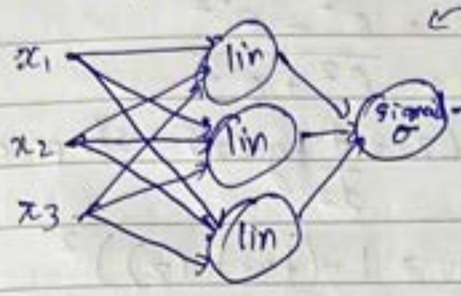
$$a^{[2]} = W^{[2]}\underbrace{(W^{[1]}x + b^{[1]})}_{a^{[1]}} + b^{[2]}$$

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

$$= W'x + b'$$

⇒ If we use a linear activation fn or alternatively if we don't have an activation fn, then no matter how many layers the NN has all its doing is computing a linear activation function. So we might as well not have any hidden layers:

→ Lets say we had :



This model is no more expressive than a standard logistic regression without any hidden layer.

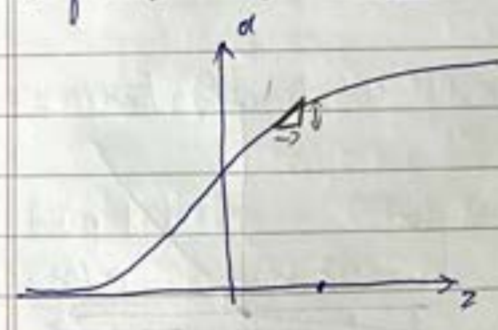
→ Imp: Linear hidden layer is more or less useless because the composition of 2 linear functions is itself a linear function.

→ Unless there is non-linearity ~~there~~ is nothing interesting we are computing as you go deeper in the network.

→ We can use linear activation only if we are doing regression problem. We can use this in the output layer.

<u>Derivatives of Activation functions</u>

<u>Sigmoid activation function</u>



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

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

$\frac{d}{dz}(g(z))$ = slope of $g(z)$ at z

$$= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \quad - ②$$

subs ① in ②

$$= g(z)\,(1 - g(z))$$

$$= a(1-a)$$

if $z = 10$, $g(z) \approx 1$

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

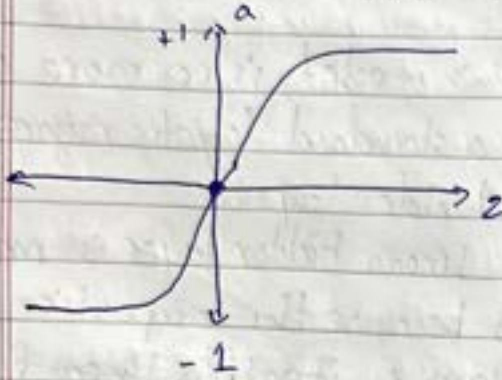
if $z = -10$, $g(z) \approx 0$

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

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

$\frac{d}{dz}(g(z)) = \frac{1}{2} \cdot \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

tanh activation function :



$g(z) = \tanh(z)$

$= \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

$g'(z) = \dfrac{d}{dz} g(z)$ = slope of $g(z)$ at z

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

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

if $z = 10$, $\tanh(z) \approx 1$

$g'(z) \approx 0$

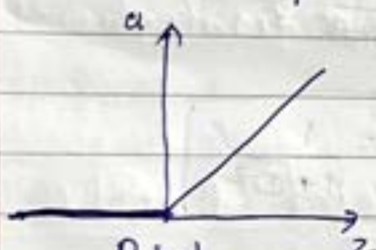
$z = -10$, $\tanh(z) \approx -1$

$g'(z) \approx 0$

$z = 0$, $\tanh(z) = 0$

$g'(z) = 1$

ReLU and Leaky ReLU:



ReLU

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

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

Leaky ReLU

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

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

Ex.

classmate
Date
Page

Gradient Descent for Neural Networks

Implementing Gradient Descent for an hidden layer

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

$(n^{[1]}, n^{[0]})\ (n^{[1]},1)\ (n^{[2]}, n^{[1]})\ (n^{[2]},1)$ input features, hidden units, output unit

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

Gradient descent:

Repeat {

Compute predictions $(\hat{y}^{(i)}, i = 1 \dots m)$

$dW^{[1]} = \frac{dJ}{dW^{[1]}}, \; db^{[1]} = \frac{dJ}{db^{[1]}}, \dots$

$W^{[1]} -= \alpha \frac{dJ}{dW^{[1]}}, \; b^{[1]} -= \alpha \frac{dJ}{db^{[1]}}$

$W^{[2]} -= \alpha \frac{dJ}{dW^{[2]}}, \; b^{[2]} -= \alpha \frac{dJ}{db^{[2]}}$

Formulas to compute 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 Prop

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

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

sum horizontally

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

prevents python from outputting one rank arrays where dimension are (n,). Setting it to True will output (n,1)

$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * g^{[1]'}(Z^{[1]})$ $(n^{[1]}, m)$

elementwise product

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

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

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

Backpropagation intuition (Optional)

Forward pass on computation graph for logistic regression

Computing gradients

~~Logistic~~ Logistic regression

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

$dz = a - y$

$\boxed{dz = da \cdot g'(z)}$

$g(z) = \sigma(z)$

$dw = dz \cdot x$

$db = dz$

$\frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz}$

$dz = da$

$\boxed{\frac{da}{dz} = \frac{d}{dz} g(z) = g'(z)}$

$da = \frac{-y}{a} + \frac{1-y}{1-a}$

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

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

Neural network gradients:

$x, W^{[1]}, b^{[1]} \rightarrow \boxed{z^{[1]} = W^{[1]} x + b^{[1]}} \rightarrow \boxed{a^{[1]} = \sigma(z^{[1]})} \rightarrow \boxed{z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}} \rightarrow \boxed{a^{[2]} = \sigma(z^{[2]})} \rightarrow \boxed{L(a^{[2]}, y)}$

$W^{[2]}$ $dW^{[2]}$ $db^{[2]}$ $b^{[2]}$

$dW^{[2]} = \frac{dL}{dW^{[2]}} = \frac{dL}{dz^{[2]}} \cdot \frac{dz}{dW}$

$= (a - y)\, x$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dW^{[2]}$

$dz^{[1]}$ $da^{[1]}$

$dz^{[2]} = a - y$

$da^{[2]} = \frac{-y}{a} + \frac{1-y}{1-a}$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

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

$z^{[2]}, dz^{[2]}$ $(n^{[2]}, 1) - (1, 1)$

$z^{[1]}, dz^{[1]}$ $(n^{[1]}, 1)$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

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

$dW^{[1]} = dz^{[1]} x^T$ $a^{[0]T}$

$db^{[1]} = dz^{[1]}$

$dz^{[2]} = \frac{dL}{dz^{[2]}} = \frac{dL}{da} \cdot \frac{da}{dz}$

$= \left(\frac{-y}{a} + \frac{1-y}{1-a}\right) a(1-a)$

$= a(1-a) \cdot \frac{-y}{a} + \frac{a(1-a)(1-y)}{(1-a)}$

$= -y(1-a) + a(1-y)$

$= -y + ay + a - ay$

$dz^{[2]} = a - y$

$dW^{[2]} = dz^{[2]} a^{[1]T}$ dw is row vector that's we are using $a^{[1]}$

$da^{[2]} = \frac{dL}{da^{[2]}} = -(y \log a + (1-y)\log(1-a))$

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

Summary of gradient descent:

Normal

$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

$dz^{[1]} = W^{[2]T} dz^{[2]} \cdot g^{[1]\prime}(z^{[1]})$

$dW^{[1]} = dz^{[1]} x^T$

$db^{[1]} = dz^{[1]}$

Vectorized implementation

$dZ^{[2]} = A^{[2]} - Y$

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

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

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

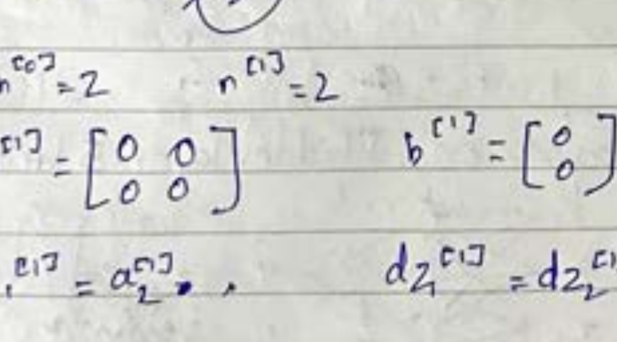
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

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

Random Initialization

→ In logistic regression we could ~~initial~~ initialize the weights to parameters to 0. In NN's initializing weight parameters to 0 won't work.

What happens if we initialize the weights to zero?



→ Initializing bias terms to 0 is fine. But not ok to do it for weights

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

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

Both activations are the same

$a_1^{[1]} = a_2^{[1]}$, $\qquad dz_1^{[1]} = dz_2^{[1]}$

Hidden units are completely identical when we compute

$dz_1^{[1]} = dz_2^{[1]}$ (outgoing wts are symmetrical)

$W^{[2]} = [0\ 0]$

After a few iterations of training the hidden units compute the same function.

$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ every row takes on the same value. so we perform a weight update.

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

- After every iteration 1st row equals 2nd row
- By the proof of induction if we initialized all ~~in~~ the values w to 0 then both hidden units start off computing the same function and both hidden units have the same influence on output unit.
- 2 hidden units are still symmetric
- By induction no matter how long we train the N.N, the hidden units compute the same function.
- In this case we dont need more than one hidden unit.
- If we have a large N.N. with 3 features and many hidden units, if initialized weights to 0, all hidden units are symmetric. No matter how long we run gradient descent they compute the same function

⇒ We want different hidden units to compute different functions. Solution: Initialize parameters randomly and multiply with small no.

$W^{[1]}$ = np.random.randn((2,2)) * 0.01 we small no because we wants weights to have small random values

$b^{[1]}$ = np.zeros((2,1)) (b doesnt have the symmetry breaking problem)

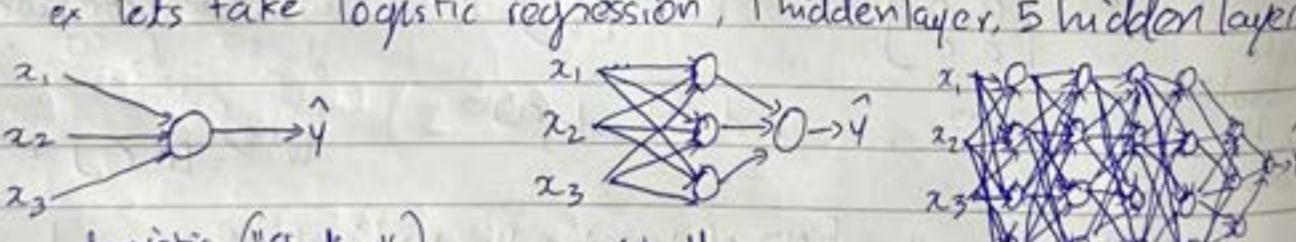
→ When we train with just one hidden layer its relatively shallow NN without too many hidden layers.

Week-4 (Deep Neural Networks)

Deep L layer NN

What is a deep NN?

ex lets take logistic regression, 1 hidden layer, 5 hidden layers



logistic ("shallow") regression

1 layer NN

1 hidden layer

(2 layer NN)

we dont include i/p layer

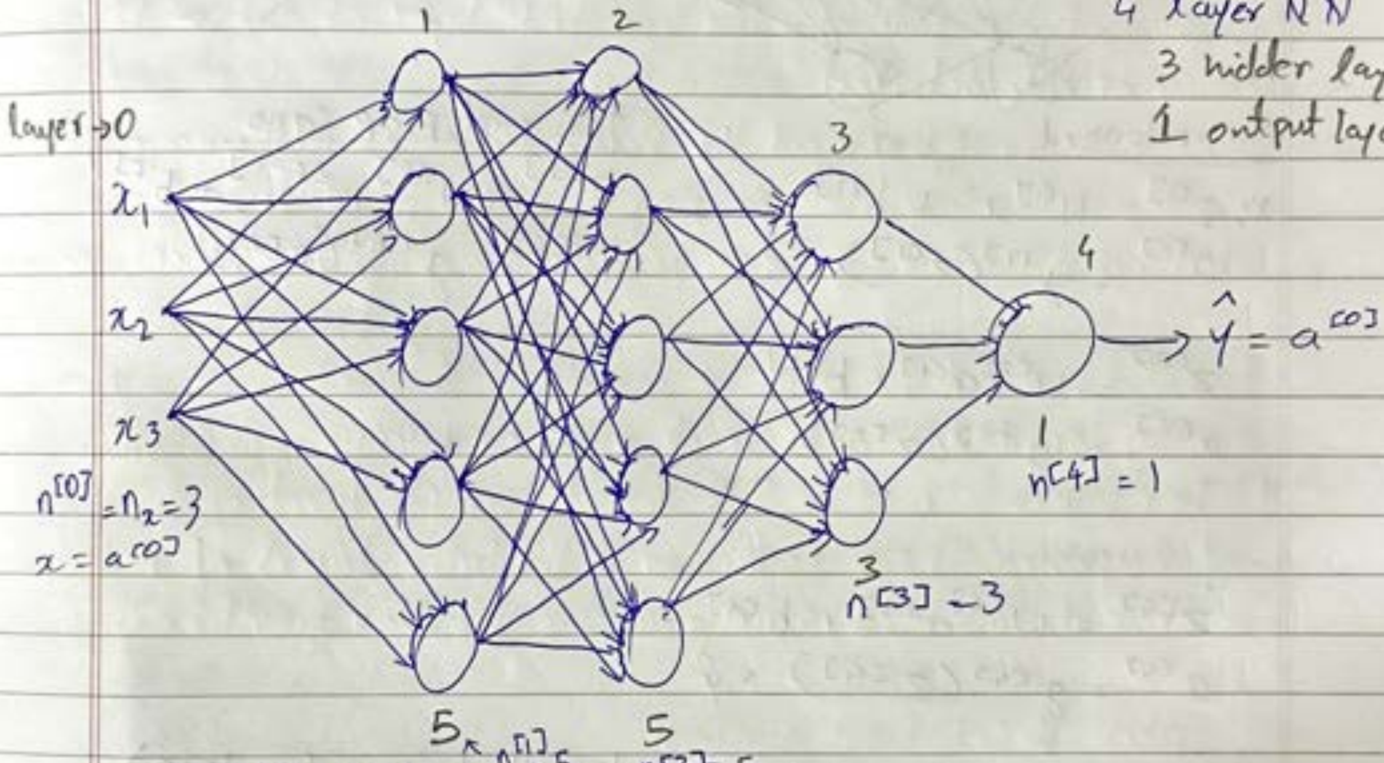
5 hidden layers

(Deep)

- Very deep NN can learn functions that shallower models are unable to.
- For any problem its hard to predict in advance how deep an NN should be. So it would be reasonable to try logistic regression, try 1 and then 2 hidden layer; ~~and view no of hidden~~
- The no of hidden layers is another hyperparameter that you could try a variety of values with and evaluate it on validation data, or on development set.

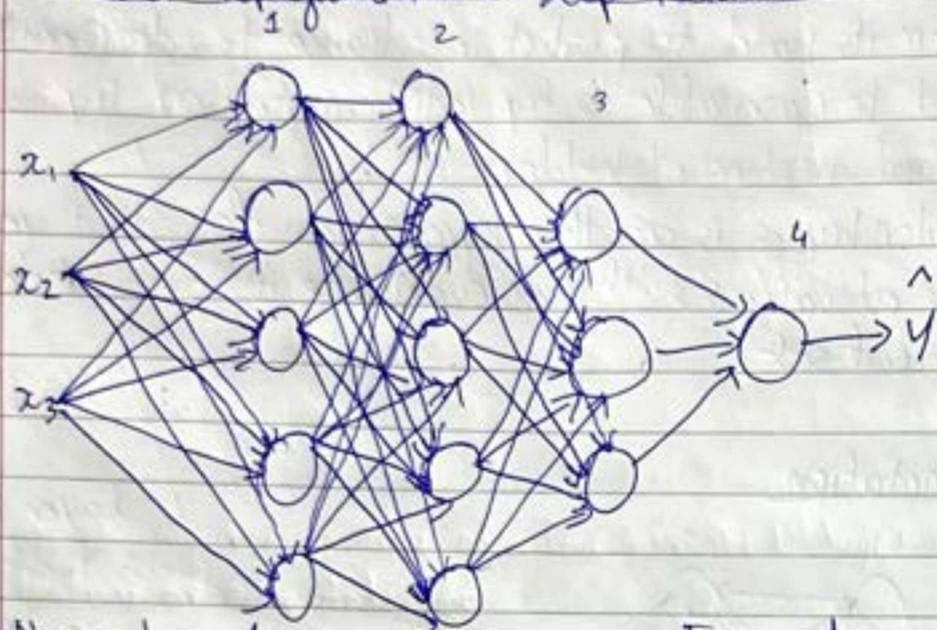
Deep NN notation

4 layer NN
3 hidden layers
1 output layer



$L = 4$ (no of layers)
$n^{[l]}$ = no of nodes in layer l
$a^{[l]}$ = activations in layer l
$a^{[l]} = g^{[l]}(z^{[l]})$
$w^{[l]}$ = weights for $z^{[l]}$
$b^{[l]}$ = bias

Forward Propogation in a Deep Network



Non vectorized

x: $z^{[1]} = W^{[1]} \overset{a^{[0]}}{x} + b^{[1]}$

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

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

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

⋮

$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$

$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$

Forward prop eqns

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

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

Vectorized implementation (We will have a for loop to compute activations for $l = 1$ to 4)

$X = A^{[0]}$

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

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

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

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

$\cancel{\hat{y} = g(z^{[4]}) = A^{[4]}}$

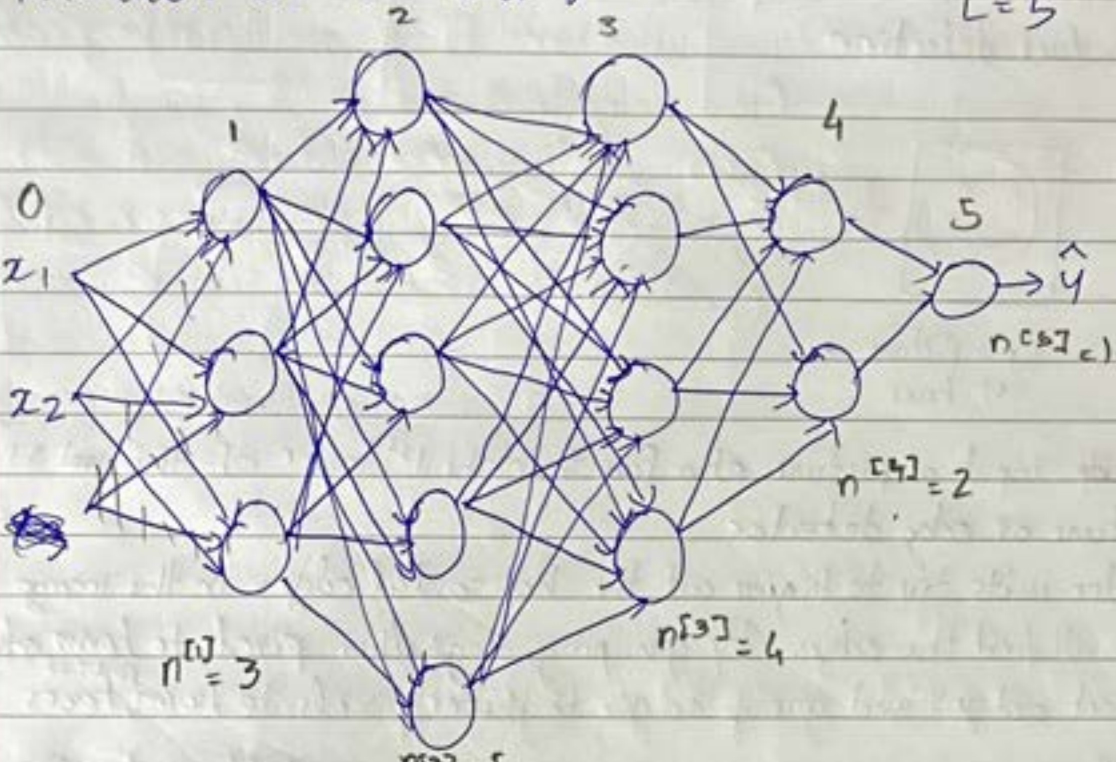
$\hat{y} = g(Z^{[4]}) = A^{[4]}$

classmate
Date
Page

Getting your matrix dimensions right

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

$L=5$ (ignore x_3)



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

$(3,1)\ (3,2)\ (2,1)\ (3,1)$

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

$\begin{bmatrix}\vdots\end{bmatrix} = \begin{bmatrix}\vdots & \vdots\end{bmatrix}\begin{bmatrix}\vdots\end{bmatrix}$

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

Note: a and z should have dimensions $(n^{[l]},1)$

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

$W^{[2]} : (5,3)\ (n^{[2]}, n^{[1]})$

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

$(5,1)\ (5,3)(3,1)\ (5,1)$

$W^{[3]} : (4,5)$

$W^{[4]} : (2,4)$

$W^{[5]} : (1,2)$

$\boxed{\begin{array}{l} W^{[l]} : (n^{[l]}, n^{[l-1]}) \\ b^{[l]} : (n^{[l]},1) \\ dW^{[l]} : (n^{[l]}, n^{[l-1]}) \\ db^{[l]} : (n^{[l]},1) \end{array}}$

<u>Vectorized implementation:</u>

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

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

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

$Z^{[1]} = W^{[1]} X + b^{[1]} \leftarrow (n^{[1]},1)$

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

$\downarrow$

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

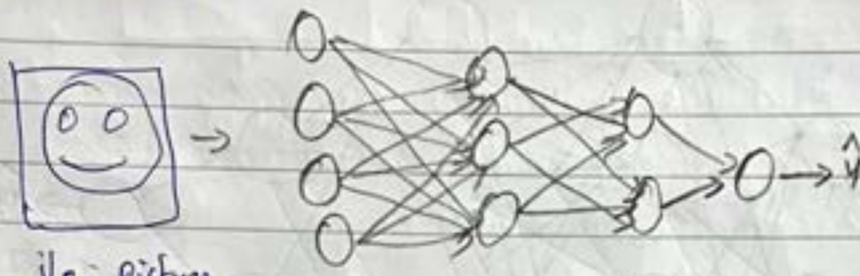
$z^{[l]}, a^{[l]} : (n^{[l]},1)$

$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$

$l=0,\ A^{[0]} = X = (n^{[0]}, m)$

$dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$

Why deep representations ?
Intuition about deep representation
ex, face detection



- We ~~if~~ input a picture of a face then the 1st layer of the ~~nn~~ NN can be a feature or edge detector
- Hidden units try to figure out the horizontal edges in the image
- We will find the edges by grouping toghether pixels to form edges. It can detect edges and group edges toghether pixels to form parts of faces ex, nose, , eye
- By putting a lot of edges it can detect ~~&~~ different parts faces. Then it can try to detect different types of faces.
- Earlier layers are of ~~a~~ NN detect simple functions like edges. and composing them toghether in the later layers of neuralnet so that it can learn more and more complex function.

ex, speech ~~Detection~~ recognition system
- i/p audio clip
- Lower level layers learn ~~the~~ detect the low level audio waveform features such as tone going up or down, pitch, etc
- By composing low level waveforms will learn to detect the basic units of sound.

~~Summary: Deep NN with multiple hidden layers will learn lower level simple features and then later might to be able to have earlier layers ~~ ~~and learn these the low~~

Deep NN ~~--~~ having multiple hidden layers. in the earlier layers they learn lower level simple features and in the deeper layers put together the simpler things learned like specific words or phrases or sentences

Why deep networks work well?

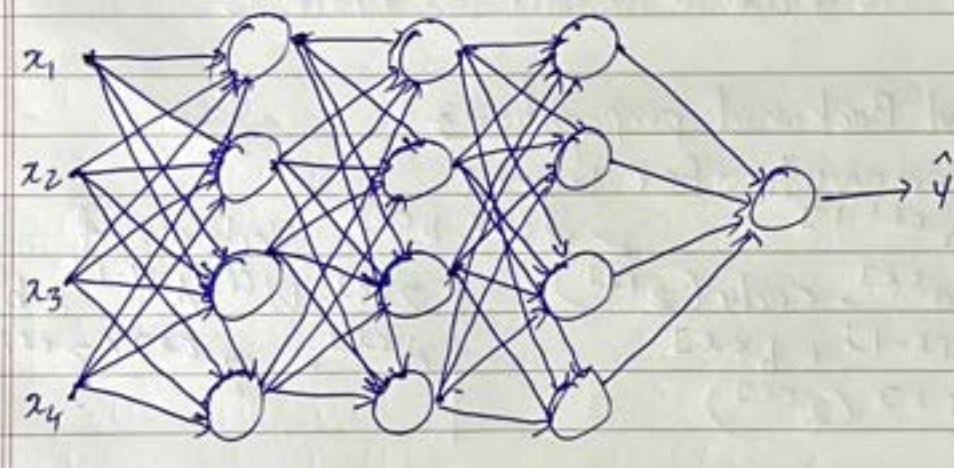
- This comes circuit theory

Circuit theory and deep learning:

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

Building Blocks of Deep Neural Networks:

Forward and Backward functions:



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

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

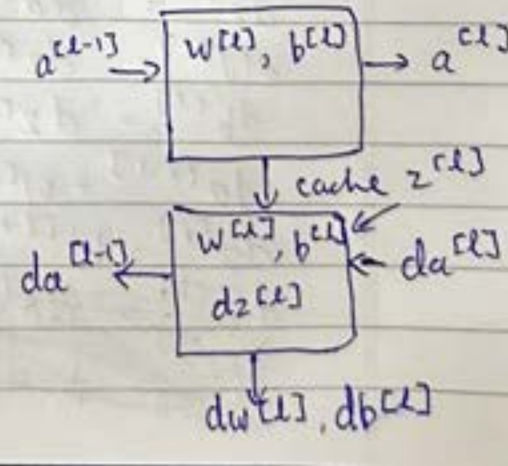
$z^{[l]} : w^{[l]} a^{[l-1]} + b^{[l]}$ cache $z^{[l]}$

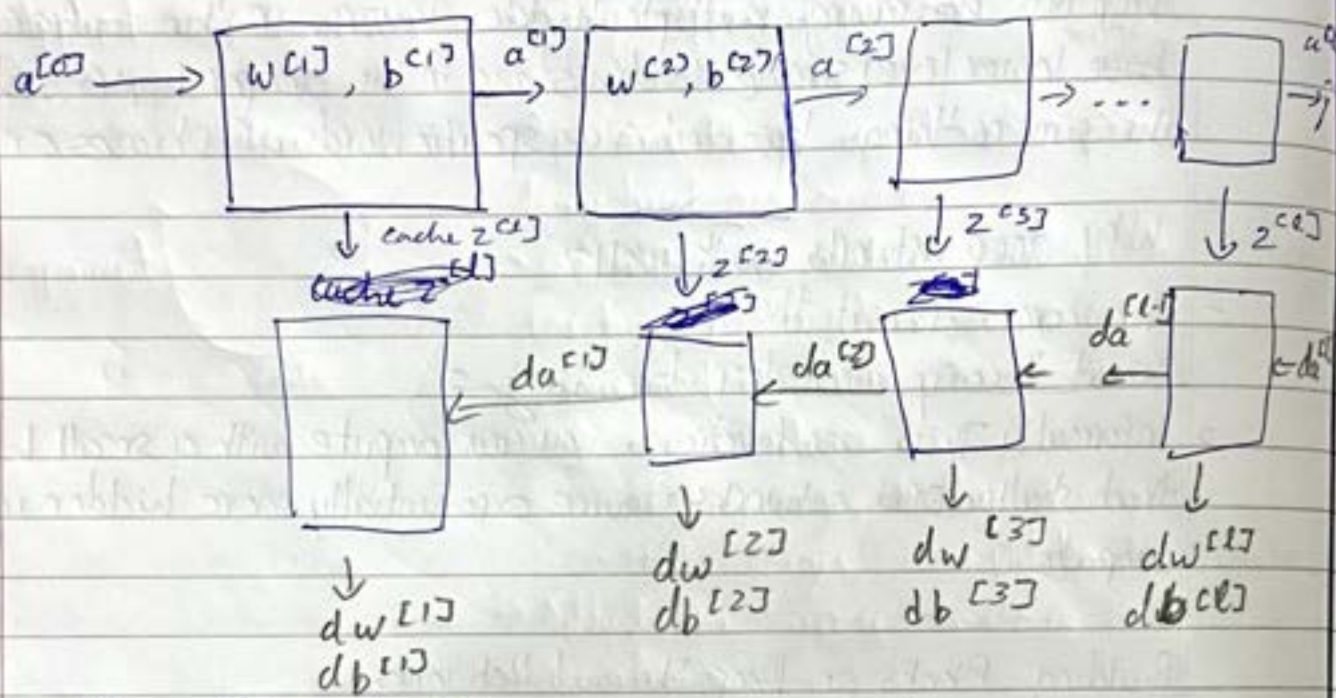
$a^{[l]} : g^{[l]}(z^{[l]})$

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

→ cache $(z^{[l]})$ $dw^{[l]}$

$db^{[l]}$





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

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

Forward and Backward propogation:

Forward propogation for layer l:

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

Output: $a^{[l]}$, cache $(z^{[l]})$ ← $w^{[l]}, b^{[l]}$

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

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

Vectorized

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

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

Backward propogation for layer l.

Input: $da^{[l]}$

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

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

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

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

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

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

Vectorized

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

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

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

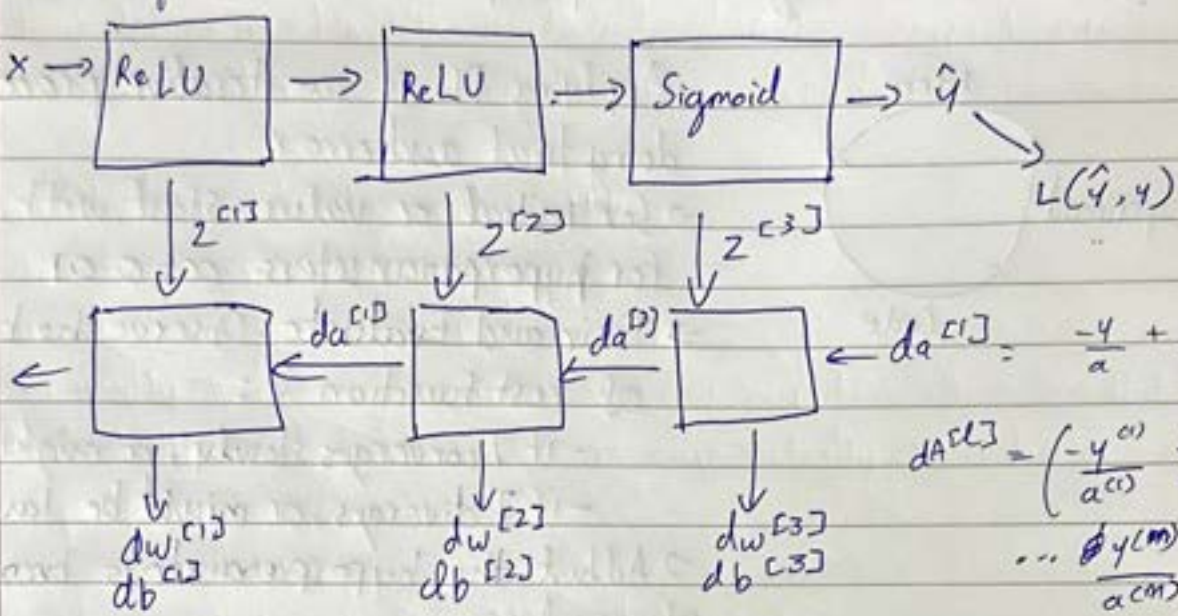
$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

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

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

db

Summary



$da^{[l]} = \frac{-y}{a} + \frac{1-y}{1-a}$

$dA^{[l]} = \left(\frac{-y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{1-a^{(1)}} \dots \dots \frac{-y^{(m)}}{a^{(m)}} + \frac{(1-y^{(m)})}{1-a^{(m)}}\right)$

Parameters vs Hyperparameters

What are hyperparameters?

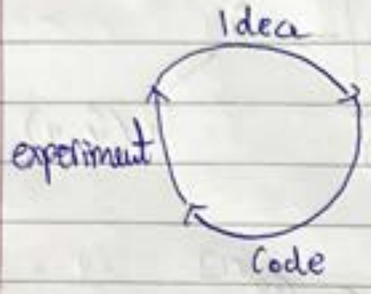
We have parameters : $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots$

Hyperparameters : → learning rate α
→ no of iterations of gradient descent
→ hidden layers L
→ no of hidden units $n^{[1]}, n^{[2]}, \dots$
→ Choice of activation function

→ Hyperparameters are parameters that control w, b. They determine the final value of w, b.

→ other hyperparameters like momentum term, mini batch size ad various regularization parameters.

Applied deep learning is a very emperical process



Applying DL is an iterative process by doing trial and error :

⇒ Set initial α value. Start with guesses for hyperparameters. $\alpha = 0.01$

⇒ Train and Evaluate : Observe the behavior of cost function J :

- if J converges slowly, α might be too sm
- if J diverges, α might be large

⇒ Adjust the hyperparameters based on observations

What does this have to do with the brain?