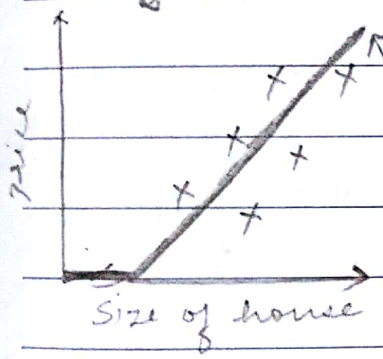
DEEP LEARNING (deeplearning.ai / Andrew Ng)
SPECIALIZATION

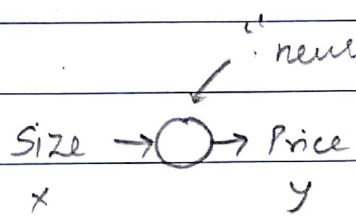
COURSE 1: Neural Networks and Deep Learning

Week 1:

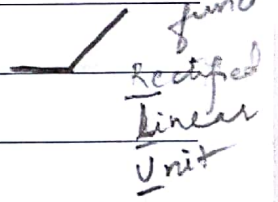
- What is neural network ?

Housing Price Prediction





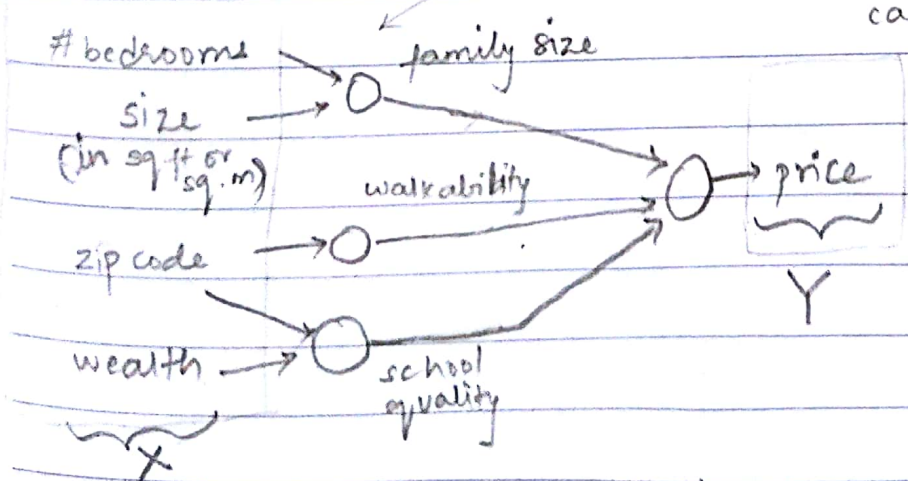
the neuron implements this function

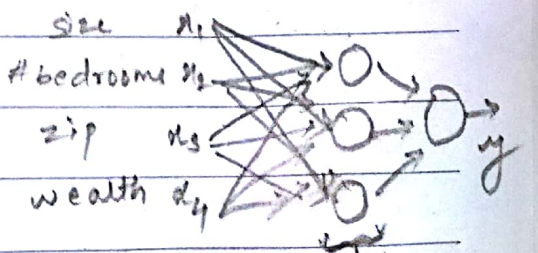


The neuron inputs the size, computes this linear function, takes a max of zero, & then outputs the estimated price

Stacking many neurons together to get a neural network. eg:

(determine whether or not a house can fit your family's family size)





Each of the neurons can be a ReLU, or some other slightly non-linear function.

Supervised Learning with Neural Networks

You have i/p (x) & want to predict the o/p (y), e.g:

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

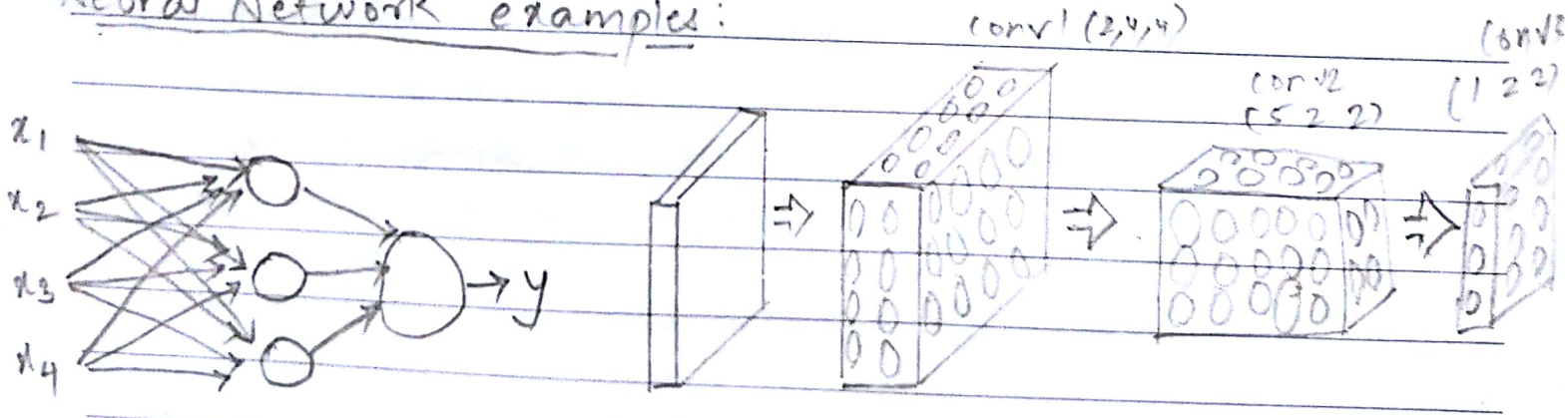
Real Estate, Online Ad. } Standard Neural N/w

Photo tagging - Convoluted NN (CNN)

Speech & Machine Transl": Recurrent NN (RNN)

Autonomous driving - Custom / Hybrid NN

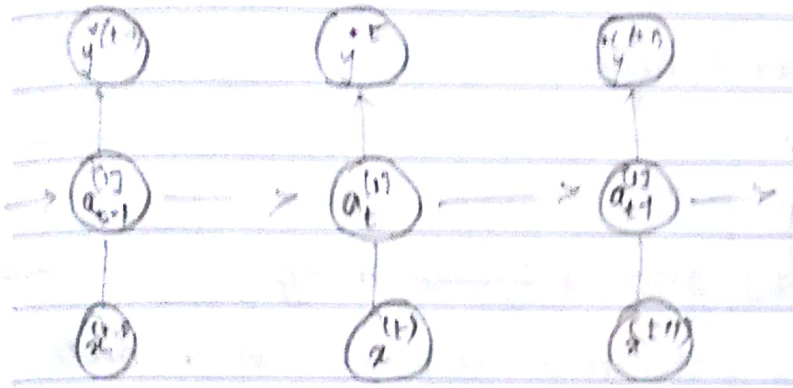
Neural Network examples:



Standard NN

Convolutional NN

1939 13
पौष / PAUSHA
शक / Shaka
3
जनवरी / JANUARY
बुधवार / WEDNESDAY
2018



Recurrent N.N

→ Supervised learning has been used for both –

Structured Data

Size	#Bedrooms	...	Price
2104	3		400
⋮			
3000	4		540

Unstructured Data

Audio, Image, Text

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
⋮	⋮		⋮
27	71244		1

Week 2 : Neural Networks Basics

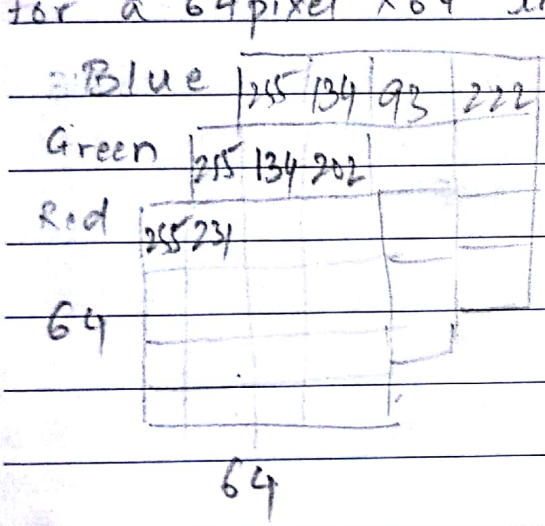
Logistic Regression as a Neural N/w :

Binary Classification :

In binary classification the goal is, given an i/p, predict to which of two classes this i/p belongs to. (eg) Given a picture of a cat, we want to classify it between cat (1) or non cat (0)

Image is stored in 3 separate matrices corresponding to red, green, and blue color channels of this image

For a 64 pixel × 64 image,



× = (feature vector)

255
231
⋮
255
134
⋮
255
134

Total dimension of this vector =

$64 \times 64 \times 3 = 12288$ (R,G,B)

$n = n_x = 12288$

So, our goal is to learn a classifier that can i/p an image represented by a feature vector x and predict whether the corresponding label y is 1 (cat) or 0 (non-cat)

Notation :-

Single training example is represented as

(x, y) where $x \in \mathbb{R}^{n_x} \Rightarrow x$ - dimentional feature vector

$y \in \{0, 1\} \Rightarrow$ label.

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

$m = m_{train}$ $\quad m_{test}$ = no. of test examples.

$$X = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots \\ | & | & | & | \end{bmatrix} \quad n_x \quad \longleftarrow m \longrightarrow$$

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

In python, command to find shape of matrix

X.shape = (n_x, m)

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \quad ; \quad Y \in \mathbb{R}^{1 \times m}$$

python, Y.shape = $(1, m)$

Logistic Regression :

Given X, we want a prediction of the probability that the class is either one or zero

Given X, want $\hat{y} = P(y=1|X)$ $\quad X \in \mathbb{R}^{n_x}$

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

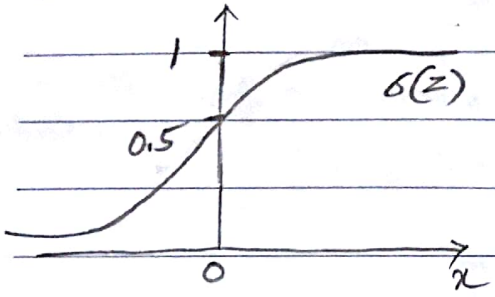
weights & biases

Output $\hat{y} = w^T x + b$. This is good for linear regression but not for binary classification since here we want $0 \leq \hat{y} \leq 1$ (Probability). But $w^T x + b$ can be much higher than one or it can even be negative. A probability can neither be negative or greater than one.

So, in logistic regression, our output is going to be the sigmoid function applied to that linear equation.

$\therefore$ output $\hat{y} = \sigma(w^T x + b)$ | $w \in \mathbb{R}^{n_x}$ & b is a real number

Sigmoid func^n:



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

If z is large^{+ve}, $e^{-z} \simeq 0$ $\therefore \sigma(z) = \frac{1}{1+0} = 1$

If z large negative, $e^{-z} \simeq \infty$ $\therefore \sigma(z) = \frac{1}{1+\infty} = 0$

So, while implementing logistic regression, we try to learn parameters w & b, so that $\hat{y}$ becomes a good estimate of the chance of Y being equal to one.

1939 7 पौष / PAUSHA शक / Shaka जनवरी / JANUARY रविवार / SUNDAY

In programming, we use $x_0 = 1$, then $X \in \mathbb{R}^{n_x+1}$

$$\hat{y} = \sigma(\theta^T X) \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \\ \} w \end{matrix}$$

← we use them separately

No need to worry about this!

Logestic Regression Cost Function:

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

To train the parameters w and b, we need to define a cost function.
Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

Loss (error) function: To measure how good our output $\hat{y}$ is when the true label is y.

Traditionally, the most used loss function would be:

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

However, this function may not be convex, making us get stuck in local minima. For this reason, our loss function is going to be instead:

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

Since we want loss func^n to be small,

→ If $y=1$, loss function will be $-\log \hat{y}$. We want $\log \hat{y}$ to be large $\Rightarrow \hat{y}$ should be large. Since $\hat{y}$ is sigmoid function, it can never be greater than one. So we want $\hat{y}$ to be close to one as well.

→ If $y=0$, then loss function will be $-\log(1-\hat{y})$. Since we want loss func^n to be small, we want $\log(1-\hat{y})$ to be large or $1-\hat{y}$ to be large. This implies that our $\hat{y}$ must be very small. Or we want our $\hat{y}$ to be as close to zero as possible.

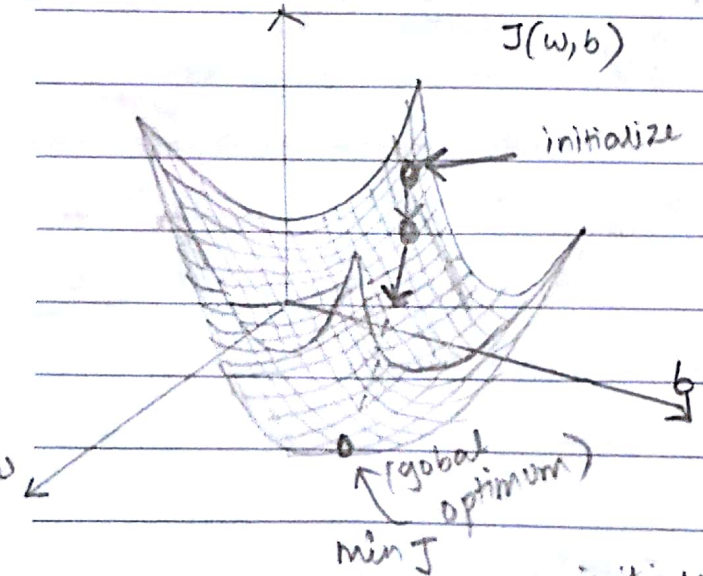
The loss function was defined with respect to a single training example. Now, we define cost function, which measures how well you're doing on entire training set.
So, cost funct^n is avg of sum of loss fun^n applied to all m training examples

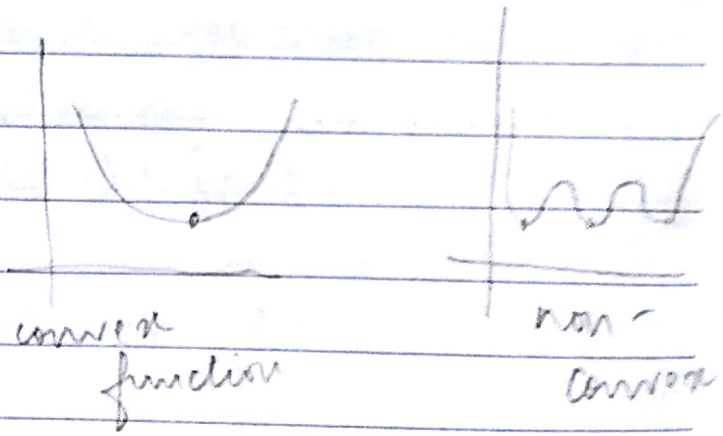
$$\text{Cost Function : } J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

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

This is going to be cost function, whereas the loss function is referred to the single example. Our goal is then to find (w,b) that minimise J(w,b).

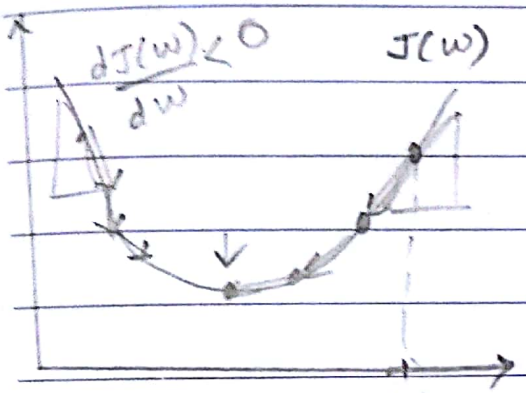
Gradient Descent:





Initialize some value to w & b. Usually initialized to 0. Random initialization works but ppl. don't usually do that for logistic regression. But since this funct^n is convex, you should get to the same point or roughly the same pt. irrespective of where you initialize.

Gradient descent will start from initial value for J and makes step in the direction of the biggest gradient (steepest direction in the curve), until converging to the global optimum (or close by).



Ignoring 'b' for now, just to get 1-D plot. We are going to repeatedly update the value of w.

Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$ ← learning rate (how big a step we take on each iteration of gradient descent)

} until convergence

The gradient's meaning is the slope of the curve. Therefore assuming that we're starting from the right side of the curve (ipt: w), its gradient is going to be positive (slope > 0), ∴ the G.D step is going to move our pt. in the opposite direction of the slope, which is towards the center, where the minimum is. The same applies if we start with a J that is in the left side of the curve (slope < 0 → step in the opposite direction, towards the center).

On right side of slope: $w := w - \alpha \underbrace{(+ve)}_{\text{slope}}$. GD moves left

Left — : $w := w - \alpha \underbrace{(-ve)}_{\text{slope}}$. GD moves right

All these work if J is a funct^n of w only.

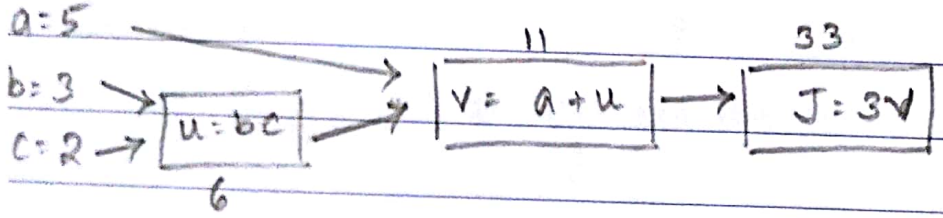
If J also depends on b, i.e $J = J(w, b)$, then we have two updates:

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

• 'δ' implies partial derivative.

Derivatives with a computation Graph:

Let $J(a,b,c) = 3(a+bc)$:



Now we calculate $\frac{dJ}{dv} = ?$ ... if we change 'v', how does the 'J' cha

J = 3v. v is currently 11 ∴ J = 33.

v = 11 → 11.001
J = 33 → 33.003
} J goes up 3 times. ∴ $\frac{dJ}{dv} = 3$

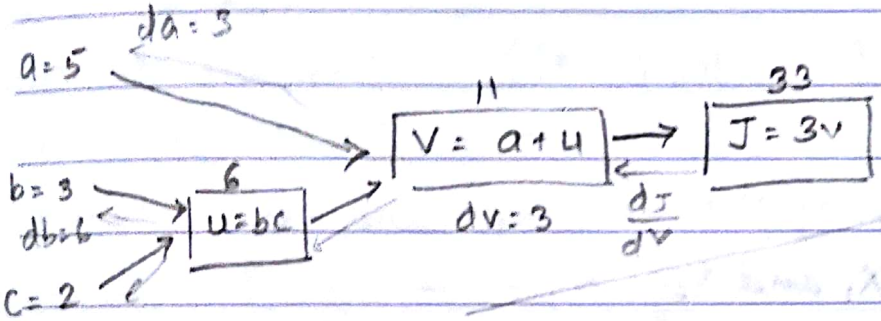
$\frac{dJ}{da} = ?$

a = 5 → 5.001
v = 11 → 11.001
J = 33 → 33.003
} J goes up 3 times ∴ $\frac{dJ}{da} = 3$

So changing a changes v which further changes J

So $\frac{dJ}{da} = \frac{dJ}{dv} \frac{dv}{da}$ ∴ $\frac{dv}{da} = 1$

To easily implement this in python, we'll use the variable names as follows: $\frac{dJ}{da} = da$; $\frac{dJ}{dv} = dv$; $\frac{dJ}{db} = db$ and so on.



$$\frac{dJ}{du} = 3 = \underbrace{\frac{dJ}{dv}}_{3} \cdot \underbrace{\frac{dv}{du}}_{1}$$

$$\frac{dJ}{db} = db = \underbrace{\frac{dJ}{du}}_{3} \cdot \underbrace{\frac{du}{db}}_{2}$$

$b = 3 \rightarrow 3.001$

$u = b \cdot c = 6 \rightarrow 6.002$

$J = 33 \rightarrow 33.006$

$$\frac{du}{db} = \frac{6.002}{3.001} = 2$$

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

$$\rightarrow \frac{dJ}{dc} = \underbrace{\frac{dJ}{du}}_{3} \cdot \underbrace{\frac{du}{dc}}_{3}$$

$c = 3 \rightarrow 3.001$

$u = bc = 9 \rightarrow 9.003$

$$\frac{du}{dc} = \frac{0.003}{0.001} = 3$$

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

Logestic Regression Gradient Descent:

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

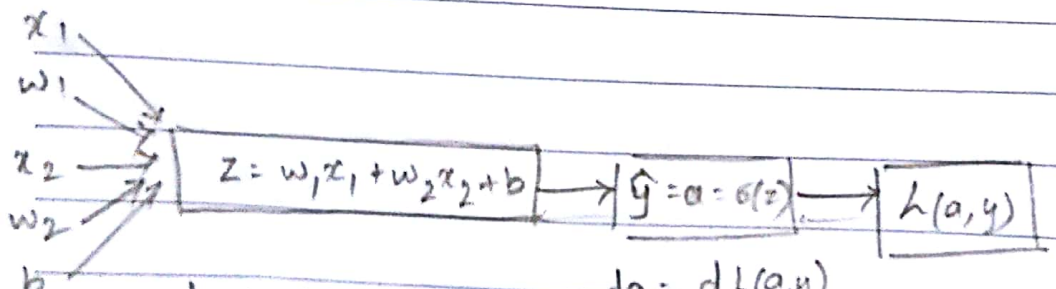
$$\hat{y} = a = \sigma(z)$$

$$L(a, y) = -(y \log a + (1-y) \log(1-a)) \quad \text{// for one example}$$

Let's write this out as a computation graph and let's assume we have two features x_1 and x_2.

So, to get z, we'll need inputs w_1, w_2 & b in addition to feature values x_1, x_2.

1939 24
पौष / PAUSHA
शक / Shaka
14
जनवरी / JANUARY
रविवार / SUNDAY



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

$$dz = \frac{dL(a,y)}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} \qquad a(1-a)$$

$$\rightarrow \quad dz = a - y$$

Then the final step is to go back & compute how much you need to change w_1 & w_2.

$\frac{\partial L}{\partial w_1} : dw_1 = x_1 dz$; $dw_2 = x_2 dz$ & $db = dz$

then $w_1 := w_1 - \alpha dw_1$ $\quad b := b - \alpha db$

$w_2 := w_2 - \alpha dw_2$

Gradient Descent on m examples:

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

It turns out that the derivative of the loss w.r.t a certain weight w_1 for all the examples is going to be simply the avg. of the derivative of the cost for each variable w.r.t a certain weight w_1.

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})}_{dw_1^{(i)} \text{ — } (x^{(i)}, y^{(i)})}$$

Let's formalise all this in a concrete algorithm.

$J=0$; $dw_1=0$; $dw_2=0$; $db=0$

For $i=1$ to m

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

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

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

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

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

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

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

} for $n=2$ (another for loop over all the features)

} inside the for loop

$J /= m$; //since we are computing avgs.

$dw_1 /= m$; $dw_2 /= m$; $db /= m$

$w_1 = w_1 - alpha * dw_1$

$w_2 = w_2 - alpha * dw_2$

$b = b - alpha * db$

dw_1 & dw_2 do not have superscript (i) as these are accumulators to sum over the entire training set.

$dw_1 = \frac{\partial J}{\partial w_1}$

..... till here we have implemented one step of gradient descent. We'll need to run this multiple times in order to take multiple steps of gradient descent.

We'll use 2 for loops and thus the code runs less efficiently. So, we use a technique called vectorization that allows you to get rid of these for loops.

Python and Vectorization:

$z = w^T x + b$

non-vectorized:

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

vectorized: (in python or numpy)

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

$w^T x$

more examples:

whenever possible, avoid explicit for-loops.

I) $u = A \cdot v$ then matrix multiplicat^n is done as $u_i = \sum_j A_{ij} v_j$

Non vectorised: (N V)

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

Vectorized. (python)

(V)

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

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

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

Non-vect:

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

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

more examples:

```
np.log(v); // element-wise log
np.abs(v); // element-wise abs
np.maximum(v,0) // element wise max w.r.t 0
v**2 ; // Square of each element
1/v
```

→ Let's use vectorization to compute derivatives of our logestic regression

Non vectorized form:

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

for $i=1$ to m:

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

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

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

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

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

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

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

If $n_x > 2$, then we require another for loop

for $j = 1 \ldots n_x$

$dw_j\ += \ldots$

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

vectorizing our second for-loop: ←

→ our dw will be a vector now -

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

then,

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

and $dw\ /= m$

Now, let's try to get rid of the main for loop as well.

Vectorizing logistic Regression: (forward propagation, ie computing 'z' & 'a')

for m training examples, we need to compute

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

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

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

X is a (n_x, m) matrix

Now, let's compute $z^{(1)}, z^{(2)}, z^{(3)} \ldots$ all in one line of code

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

$1 \times m$ matrix

$[\text{——}]$ row vector $\quad \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} + [b \; b \; \cdots \; b]$

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

$1 \times m \qquad 1 \times m$

In python,

```
Z = np.dot(w.T, X) + b
    (1,1)
```

python automatically adds 'b' to each element of the matrix $w^T x$. This is called 'broadcasting'

Next, we'll compute $a^{(1)}, a^{(2)} \ldots a^{(m)}$ all at the same time.

$A = [a^{(1)} \; a^{(2)} \; \cdots \; a^{(m)}] = \text{sigmoid}(Z)$

Vectorizing Logistic Regression's Gradient Output :

Previously, we computed the derivatives for each training example's prediction as $dz^{(1)} = a^{(1)} - y^{(1)}$, $dz^{(2)} = a^{(2)} - y^{(2)}$ and so on.

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

$$dZ = \underset{1 \times m}{\left[dz^{(1)} \quad dz^{(2)} \quad \dots \quad dz^{(m)} \right]} = A - Y$$

$$= \left[a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \quad \dots \right]$$

in our prev. page,

$A = \left[a^{(1)} \quad \dots \quad a^{(m)} \right]$

& $Y = \left[y^{(1)} \quad \dots \quad y^{(m)} \right]$

He had another for loop where we calculated :

1939 1 माघ / MAGHA
शक / Shaka
21
जनवरी / JANUARY
रविवार / SUNDAY

$dw = 0$

$dw += x^{(1)} dz^{(1)}$

$dw += x^{(2)} dz^{(2)}$

$\vdots$

$dw /= m$

$db = 0$

$db += dz^{(1)}$

$db += dz^{(2)}$

$\vdots$

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

$db /= m$

Vectorising the second for - loop:

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

In python

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

$$dw = \frac{1}{m} . X . dz^T \quad *$$

$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

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

$n \times 1$

$\Rightarrow \frac{1}{m} * X * dz^T$

putting it all together:

A = 1/(1+np.exp(-z))

Non-vectorised

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

for i = 1 to m:

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

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

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

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

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

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

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

$J = J/m, \; dw_1 = dw_1/m, \; dw_2 = dw_2/m$

$db = db/m$

Vectorised:

$Z = w^T X + b$

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

$A = sigmoid(z)$

$dZ = A - Y$

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

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

$w = w - \alpha dw$

$b = b - \alpha db$

with this we have just implemented a single elevation of gradient descent for logistic regression

But we will need to call this multiple times for multiple iterations of gradient descent. So we will need a 'for' loop. There may not be a way to get rid of this for loop

Broadcasting in Python:

↳ is a technique used to speed up computation.

(eg)

	Apples	Beef	Eggs	Potatoes
carbs	56.0	0.0	4.4	68.0
Proteins	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

Our goal is to calculate the percent of calories from C, P, & F for each of the 4 foods

A: 59 cal

so. carbs = $\frac{56}{59}$ = 94.9% .. and so on for each of the 4 foods

Can we do this without a for-loop? Yes!

→ let the above matrix be A (3×4)

① With 1 line of code, we'll calculate sum of each columns.

② — " — , — " — divide each of the value of columns with their corresp. sum

python:

```
import numpy as np
A = np.array([[56.0, 0.0, 4.4, 68.0], [1.2, 104.0, 52.0, 8.0], [1.8, 135.0, 99.0, 0.9]])
```

③ `cal = A.sum(axis=0)` // calculate sum of all columns

`print(cal)` // output: [59. 239. 155.4 76.9]

④ `percentage = 100*A/cal.reshape(1,4)` // divide by 1×4 matrix

`print(percentage)` // o/p: [[94.91525424 0. 2.83140283 88.42652796]
[. . . .]
[. . . .]]

→ cal = A.sum(axis=0)

axis = 0 .. to sum vertically; axis = 1 .... sum horizontally

→ percentage = 100 * A / cal.reshape(1, 4)

since our 'cal' is already a 1×4 matrix, the 'reshape' part is a bit redundant. We have divided a (3×4) matrix by a (1×4) matrix.

More examples of broadcasting:

→ $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 = ?$. Python will auto expand '100' into a (4×1) vector as well: $\begin{bmatrix}100\\100\\100\\100\end{bmatrix}$

The the result is $\begin{bmatrix}101\\102\\103\\104\end{bmatrix}$.

→ Similarly, if we add:

$\underset{(A)}{\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}} + \underset{(B)}{\begin{bmatrix}100&200&300\end{bmatrix}}$. Python will expand matrix B as $\begin{bmatrix}100&200&300\\100&200&300\end{bmatrix}$

Then result is $\begin{bmatrix}101&102&103\\104&105&106\end{bmatrix}$

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

⇒ **General Principle:** (python converts it into)

(m, n) matrix	$\pm$	(1, n)	⇝ (m, n)
	$*$ /	(m, 1)	⇝ (m, n)
or (m, 1) (1, m)	$\pm$ $*$ /	$\mathbb{R}$ $(1\times1)_{\text{matrix}}$	⇝ (m, 1)

A note on python/numpy vectors:

```
import numpy as np
a = np.random.randn(5)
print(a)            // o/p: [0.5 ... -0.2 ... 0.95 ... -0.82 ... -1.42 ...]
print(a.shape)      // o/p: (5,) . so this is neither a row vector
                            or a column vector.
print(a.T)          // the o/p of this ends up looking the same as
                       print(a). ie [0.5 ... -0.2 ... 0.95 ... -0.82 ... -1.42]
print(np.dot(a, a.T)) // o/p: 4.0657.. now we may think that this
                         should result in a matrix but we get a
                         number instead.
```

transpose →

So, to solve this we should do:

```
a = np.random.randn(5,1)  → // o/p will be a 5x1 matrix
print(a)
print(a.T)          // o/p will be a 1x5 matrix.
print(np.dot(a,a.T)) // o/p will be a outer product of 2 matrices... a
                        5x5 matrix
```

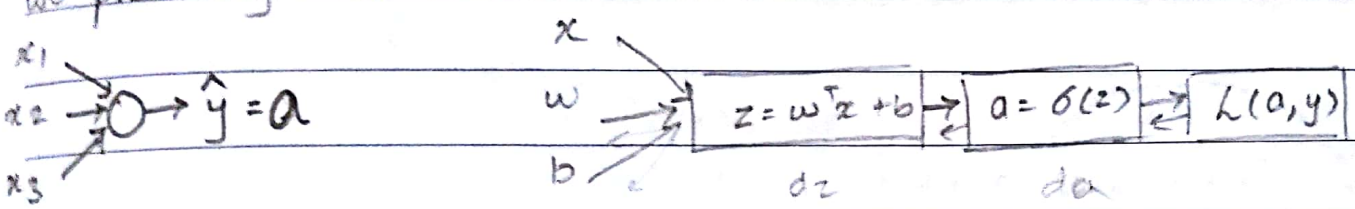
In detail,

```
→ a = np.random.randn(5)   . this is basically a 'rank 1 array'.
So a.shape = (5,)  ——→
never
use
this
```

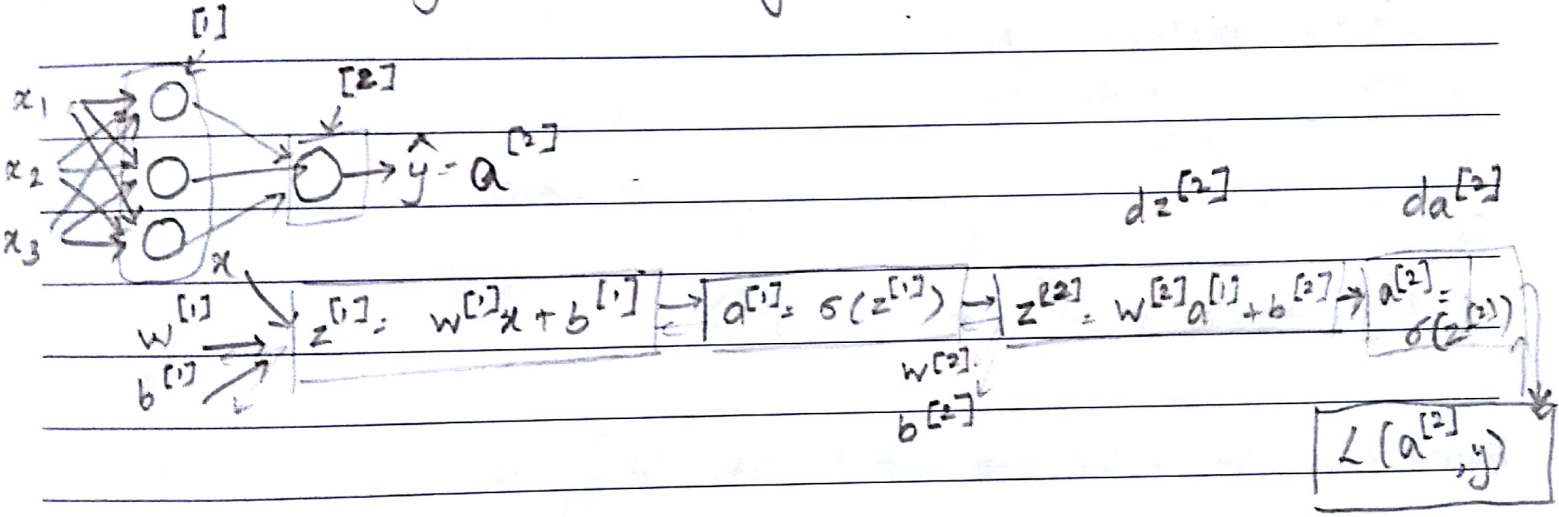
```
a = np.random.randn(5,1) → a.shape = (5,1) ... column vector
assert(a.shape == (5,1))  ← check what this does
a = a.reshape((5,1))  // we can use reshape to convert a
                         rank matrix to a (5x1) or
                         (1x5) matrix
```

Week 3: Shallow Neural Networks:

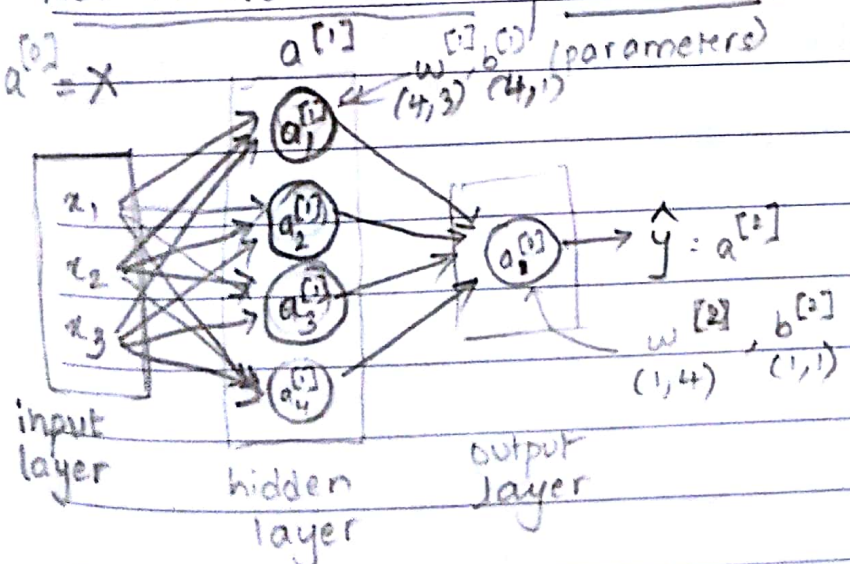
we previously had;



with 1 hidden layer:- (or 2 layer neural n/w)



Neural Network Representation:

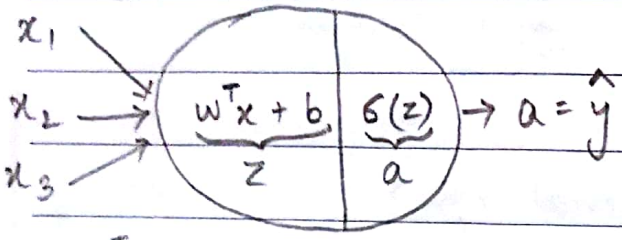


$a^{[0]}$ = activations of the i/p layer

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

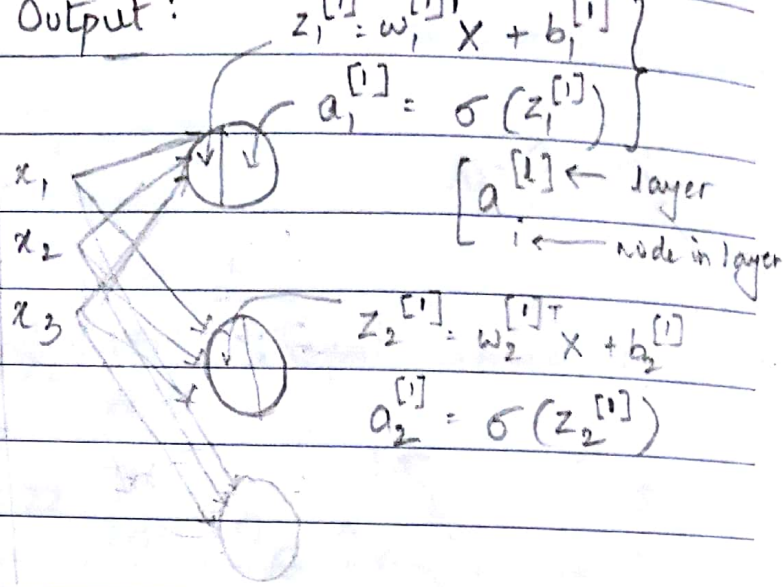
2 layer N.N (1-hidden layer)
1- o/p layer

Computing a Neural Network's Output:



$z = w^T x + b$ } a single sigmoid neuron does these two things

$a = \sigma(z)$ }



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

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

$\vdots$

Doing this with a for loop will be very inefficient. So, we vectorize.

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b \\ \vdots \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$(4,3)$ $W^{[1]}$ $(4,1)$ $b^{[1]}$

then we calculate 'a' as

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

So, given input x:

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

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

Second layer (output layer):

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

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

Vectorizing across multiple examples:

for a 2 layer N.N :

$x \longrightarrow a^{[2]} = \hat{y}$. This is for one training example.

If there are m training examples, then

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

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

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

$a^{[2](i)}$ ← example i, layer 2

So, for all training examples,

for i = 1 to m,

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

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

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

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

} Non-vectorized form

1939 | 15
माघ / MAGHA
शक / Shaka
4
फरवरी / FEBRUARY
रविवार / SUNDAY

In vectorized form, the computation will be.

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ ← m eg's; ↓ diff. no. features

(n_x, m)

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

$A^{[0]}$

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

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

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

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

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

→ diff. training examples

↓ hidden layers

Explanation for vectorized implementation:

justification for vectorized implementation —

$z^{[1](1)} = w^{[1]} x^{(1)} + b^{[1]}$ → for 1st training example

$z^{[1](2)} = w^{[1]} x^{(2)} + b^{[1]}$ → for 2nd training example

$w^{[1]} = \begin{bmatrix} — \\ — \end{bmatrix}$; $w^{[1]} x^{(1)} = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ column vector — so are $w^{[1]} x^{(2)}$ & $w^{[1]} x^{(3)}$

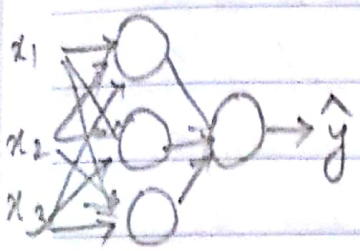
Our $X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix}$, then $w^{[1]} X = \begin{bmatrix} | & | & | \\ w^{[1]}x^{(1)} & w^{[1]}x^{(2)} & w^{[1]}x^{(3)} \\ | & | & | \end{bmatrix}$

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

$+b^{[1]} \quad +b^{[1]} \quad +b^{[1]}$

So, $w^{[1]} x^{(i)} = z^{[1](i)}$

Activation functions:



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

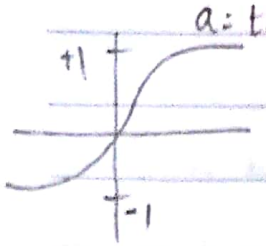
$a^{[1]} = \sigma(z^{[1]})$

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

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

Sigmoid function is one activation function. We can have more:

1) tanh (almost always works better than sigmoid):

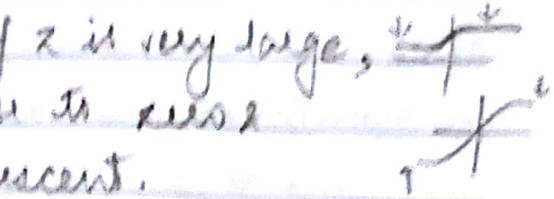


$= \frac{e^z - e^{-z}}{e^z + e^{-z}} = g(z)$

↳ because it has a zero mean not 0.5 like sigmoid

sigmoid can be used in binary classification at the output layer as in B.C, our $y \in \{0,1\}$. So it makes sense to have sigmoid where $0 \le \hat{y} \le 1$ instead of tanh, where $-1 \le \hat{y} \le 1$.

Disadvantage of sigmoid & tanh is that if z is very large, or very small, then the slope is close to zero so this can slow down gradient descent.



2) ReLU

derivative is 1 so long as z is positive and " is 0 ← z is negative

when z=0, slope is not defined.

But in computer, we get derivative as 0.0000...

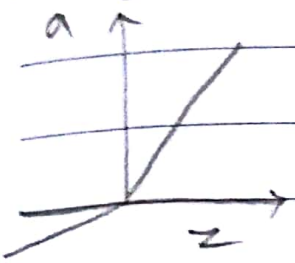
Some thumb rules :-

- When output is 0 or 1, sigmoid funct^n is natural choice for output layer and for other layers ReLU is default choice.
- When not sure what to use, use ReLU
- Advantage of ReLU is that for a lot of space of z, the derivative of slop of activation function is very different from zero. So in practice using ReLU, your NN will often learn much faster than tanh or sigmoid as slope of the function doesn't much go to zero which slows down learning. Though half of the range of z, the slope of ReLU is zero but in practice, enough of your hidden units will have z greater than 0 so learning can still be fast
- You can try all the activation functions & choose the best.

Pros & cons

1) Sigmoid : Never use except in o/p layer of binary classification

3) Leaky ReLu:



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

works better than ReLU

→ Why do you need non-linear activation functions?

Given x:

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

$a^{[1]} = g^{[1]}(z^{[1]})$ ... so instead of this let's say we have $g(z) = z$

then $g(z)$ is a linear activation function. or identity activation function i.e it outputs whatever is the input

So $\hat{y}$ is a linear function of your input features x.

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

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

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

$= W'x + b'$

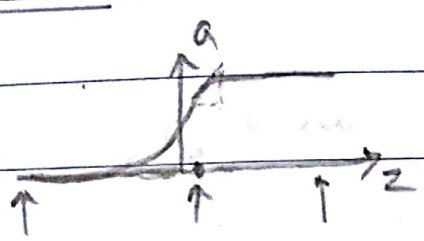
So o/p is just a linear funct^n of the i/p.

Whatever hidden layers you add, the activation will be always linear like logistic regression (So it is useless in a lot of complex problems).

You might use linear activation function in one place - in the o/p layer if the output is real nos. (like price of the house $0 to .... $10000... ; basically regression problem). But even in this case if the o/p value is non-negative you could use ReLU instead.

Derivatives of activation function:

Sigmoid function:



$$g(z) = \frac{1}{1+e^{-z}} \quad \text{then slope} = \frac{d}{dz} g(z) = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$$

$$\text{or } g'(z) = g(z)\,(1 - g(z))$$

Sanity check:

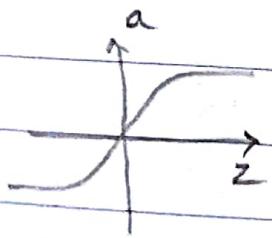
If $z = 10$, $g(z) \approx 1$. $\frac{d}{dz} g(z) \approx 1(1-1) = 0$ ... correct, as if z is large, slope is close to zero

$z = -10$, $g(z) \approx 0$. $\frac{d}{dz} g(z) \approx 0(1-0) = 0$ .... correct, as if z is very small, slope is close to zero

$z = 0$, then $g(z) = 1/2$ $\frac{d}{dz} = 1 \cdot (1-0.5) = 0.25$ ... correct slope.

Since $a = g(z) = \frac{1}{1+e^{-z}}$, then $g'(z) = a(1-a)$

tanh function:



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

$g(z) = \tanh(z)$ then $\frac{d}{dz} g(z) = 1 - (\tanh(z))^2$

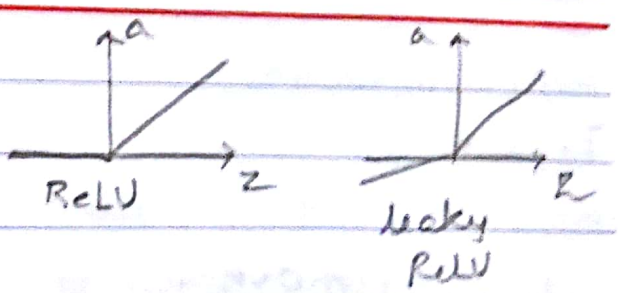
Sanity check: $z = 10$, $\tanh(z) \approx 1$; $g'(z) = 0$

$z = -10$, $\tanh(z) \approx -1$; $g'(z) = 0$

$z = 0$, $\tanh(z) \approx 0$; $g'(z) = 1$

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

→ ReLU and Leaky ReLU:



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

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

undefined if $z = 0$ .... but when implemented on computer you can set it to 1, no problem. So

$g'(z) = 1$ if $z \geq 0$

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

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

Gradient descent for Neural Networks:

1939 22
माघ / MAGHA
शक / Shaka
11
फरवरी / FEBRUARY
रविवार / SUNDAY

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

Input layer, $n_x = n^{[0]}$

Hidden layer, $n^{[1]}$

Output layer, $n^{[2]} = 1$

So, $w^{[1]} = (n^{[1]}, n^{[0]})$; $b^{[1]} = (n^{[1]}, 1)$; $w^{[2]} = (n^{[2]}, n^{[1]})$; $b^{[2]} = (n^{[2]}, 1)$

Cost function :

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$$

gradient descent :

Repeat {
- Compute predictions ($\hat{y}^{(i)}$ for $i = 1 \ldots m$)
- Compute derivatives ($dw^{[1]}$, $db^{[1]}$, ... ($dw^{[2]}$, $db^{[2]}$)
- update $w^{[1]} := w^{[1]} - \alpha . dw^{[1]}$
  $b^{[1]} := b^{[1]} - \alpha\, db^{[1]}$ } also for $w^{[2]}$ & $b^{[2]}$

}

This is one iteration of gradient descent. So repeat this until convergence

Formulas for computing derivatives :

Forward propagation steps :

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

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

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

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

* keepdims = True ensures that the o/p is a (n, 1) matrix else o/p is a (n,) matrix

Backpropagation Steps :

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

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

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

$dZ^{[2]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m) \text{ matrix}} \times \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]}, m) \text{ matrix}}$ ← element wise product

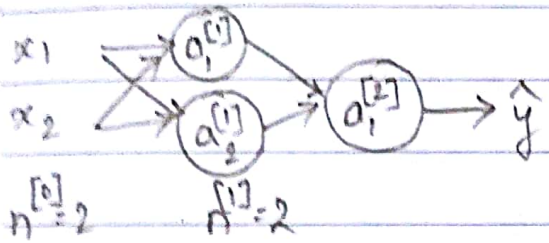
$dW^{[1]} = \frac{1}{m}\, dZ^{[1]} X^T$

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

↓ $(n^{[1]}, 1)$ vector

Random Initialization:

For logistic regression, it was ok to initialize the weights to zero. For N.N., to initialise the weights to zero and then applying gradient descent, it won't work. Let's see why.



$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ let's say we initialize with all zeros & $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

- Turns out, initialising b to zero is ok, but initializing w to all 0s is a problem.

With this formalization, for any example you give it, we have $a_1^{[1]} = a_2^{[1]}$ (as both are computing the exact same function.)

During backpropagation, $dz_1^{[1]} = dz_2^{[1]}$

And by proof of induction, it turns out that after every single iteration of training your two hidden units are still computing exactly the same function.

our dw looks like this:

$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ ... every row takes on the same value.

so when we perform: $W^{[1]} = W^{[1]} - \alpha dW$ then $W^{[1]} = \begin{bmatrix} \cdots \\ \cdots \end{bmatrix}$ ← first row is equal to the second row.

All hidden units become symmetric.

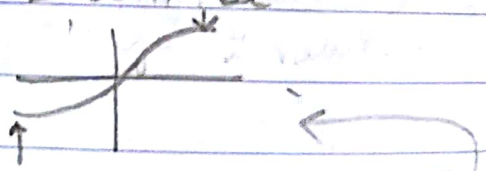
so, there is really no point in having more than one hidden unit as they are all computing the same thing.

Solution : initialize your parameters randomly.

You can set $w^{[1]} = np.random.randn((2,2)) * 0.01$ # because we want very small values.

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

Large values of weights imply that z will be very large or very small



In this case, we are more likely to end up at these parts, where the slope of the gradient is very small. Meaning that the gradient descent will be very slow.

Also, at o/p layer we can have $w^{[2]} = np.random.randn((1,2)) * 0.01$

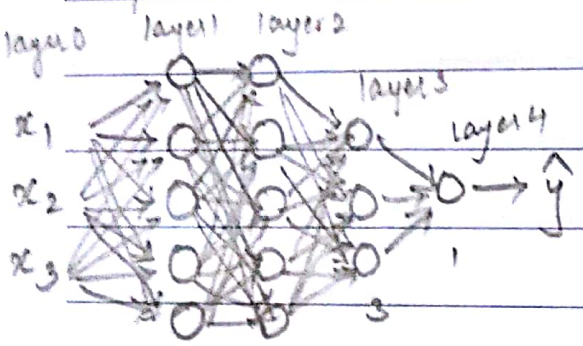
When working with very deep N.N, we might want to pick a different constant than 0.01.

Week 4 : Deep Neural Networks

Deep L-layer neural network :

Shallow N.N → one or 2 layers

Deep N.N → three or more layers



'4' layer N.N.

Notations :

L = no. of layers in the network,
here L = 4.

$n^{[l]}$ = no. of units in l

$n^{[1]} = 5$; $n^{[2]} = 5$; $n^{[3]} = 3$; $n^{[4]} = 1$
$= n_y^{[L]}$

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

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

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

$b^{[l]}$

$x = a^{[0]}$ $\quad a^{[L]} = \hat{y}$

Forward Propagation in Deep Neural Network :-

$a^{[0]}$

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

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

layer 2: $z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$

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

layer 4: $z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$

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

$= \hat{y}$

generally,

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

} all for a single training eg.

Vectorised way for the whole training set :-

$A^{[0]} = X$

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

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

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

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

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

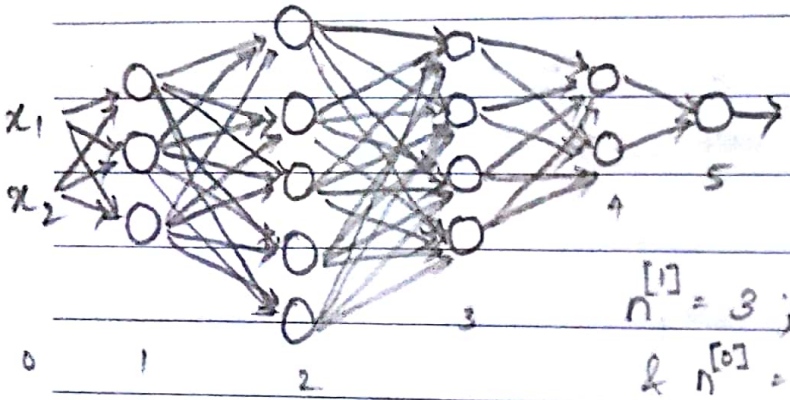
similarly $A = \begin{bmatrix} | & | & & | \\ | & | & & | \end{bmatrix}$

then,

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

while performing forward propagation, it is perfectly ok to have a for-loop to calculate activation function for layers 1, 2, 3 & 4.

Getting your matrix dimensions right: (best way is to use paper & pencil)



$L = 5$

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

$n^{[1]} = 3$; $n^{[2]} = 5$; $n^{[3]} = 4$; $n^{[4]} = 2$; $n^{[5]} = 1$

& $n^{[0]} = n_x = 2$

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

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

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

So, we need a matrix $W^{[1]}$ to be something that when we multiply an $(n^{[0]}, 1)$ matrix, we get an $(n^{[1]}, 1)$ vector. $z^{[1]}$ is $(3,1)$ & x is $(2,1)$. So $W^{[1]}$ will be $(3,2)$. or generally, $W^{[1]}$ is $(n^{[1]}, n^{[0]})$ matrix.

$W^{[1]} : (n^{[1]}, n^{[0]})$ ... generally $W^{[n]} : (n^{[1]}, n^{[l-1]})$

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

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

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

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

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

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

generally,

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

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

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

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

so z & a should have the same dimension

When we have a vectorized implementation, that looks at multiple examples at a time.

The dimensions of wb, dw, and db will stay the same. But the dimensions of z, a, as well as x will change a bit.

Previously we had,

$z^{[1]} = w^{[1]} x + b^{[1]}$ ← $(n^{[1]}, 1)$

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

In vectorised implementation:

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

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

$(n^{[1]}, 1)$ → $(n^{[1]}, m)$ matrix by python broadcasting.

$Z^{[1]}, a^{[1]} : (n^{[1]}, 1)$ ... previously,

now in vectorized,

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

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

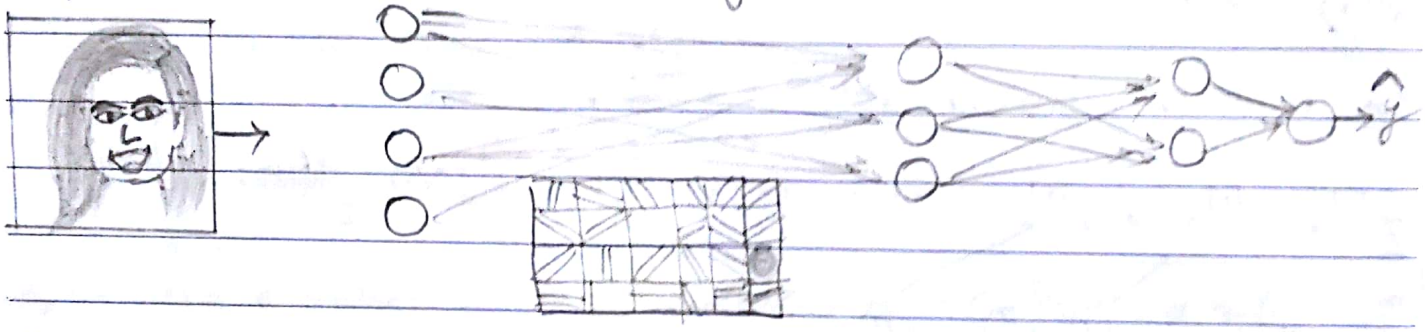
$dZ^{[1]}, dA^{[1]} : (n^{[1]}, m)$ .... same as Z & A.

Why deep representations?

Why deep networks work well?

First, what is deep network computing?

→ If you're building a system for face recognition, here's what a deep neural network could be doing.



The first layer of the N.N, you can think of as maybe being a feature detector or an edge detector. If I have 20 hidden units, then the hidden units are trying to figure out where the edges of what orientation in the image, some other hidden unit maybe trying to figure out where are the horizontal lines in the image, etc. We can think of the first layer of the neural n/w as looking the picture and try to figure out where are the edges in the picture

Now let's think about where are the edges in this picture by grouping together pixels to form edges. It can then detect the edges and group the edges together to form parts of faces. For eg, you might have a neuron trying to see if it's finding an eye or a diff neuron trying to find that part of the nose. And so by putting together lots of edges, it can start to detect diff. parts of faces.

And then finally, by putting together different parts of face, like an eye or a nose or a chin, it can then try to recognise or detect different types of faces. So intuitively, we can think of the earlier layers of the N.N as detecting simple functions like edges. And then composing them together in the later layers of a N.N. so that it can learn more & more complex functions. The edge detectors are looking in relatively small areas of an image, and facial detectors look at much larger areas of image. Main takeaway: find simpler things like edges; composing them together to find complex things & composing those together to find even more complex things.

In speech recognition, if you input an audio clip, the first level of a N.N might learn to detect low level audio wave features, such as is this tone going up? or down? is it white noise or slithering sound & what is the pitch. And then composing them, you'll learn to detect basic units of sound (phonemes). Then composing these together maybe learn to recognise words in the audio. Then compose those together in order to recognise entire phrases or sentences.

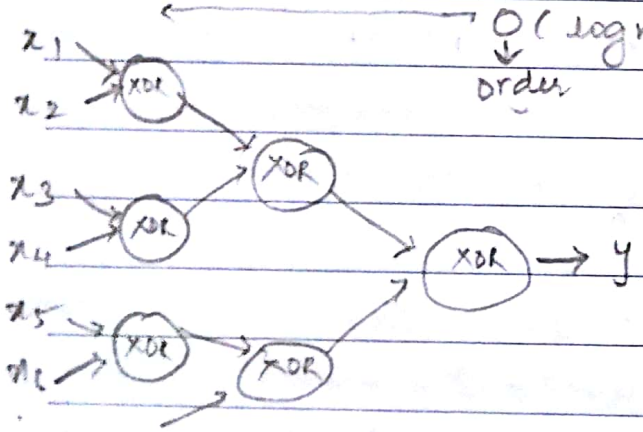
Circuit theory and deep learning:

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.

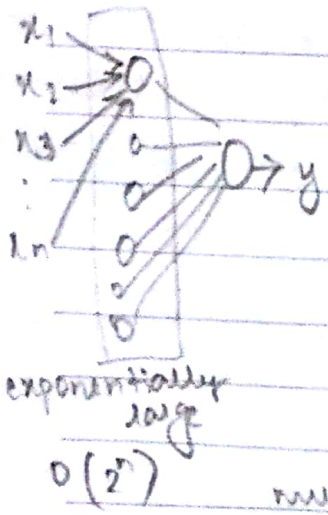
(eg): x_1 XOR x_2 XOR x_3 XOR x_4 .... XOR x_n = y

$\longleftarrow \quad O(\log n) \quad \longrightarrow$

order



Without a N.N. w.r.t. (only 1 hidden layer):

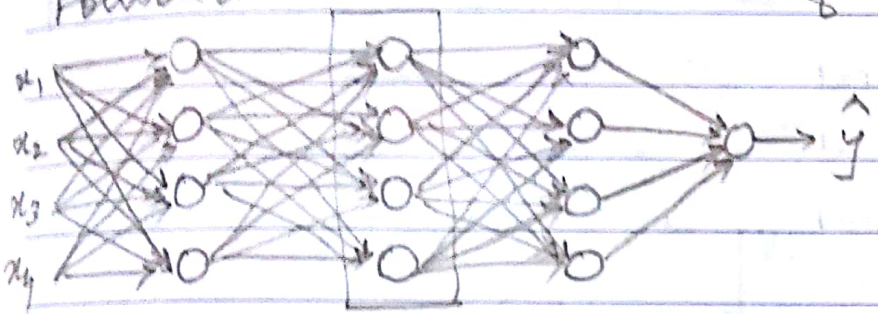


then in order to compute this XOR function, this hidden layer will need to be exponentially large because essentially, you need to exhaustively enumerate over 2^n possible configurations of the input bits that result in XOR being either 1 or 0.

So, there are mathematical functions, that are much easier to compute with deep N.N than with shallow networks.

Building blocks of deep neural networks

Forward and backward functions:



Let's pick one layer & look at the computations focussing on just that layer for now.

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

forward prop : Input $a^{[l-1]}$; output $a^{[l]}$

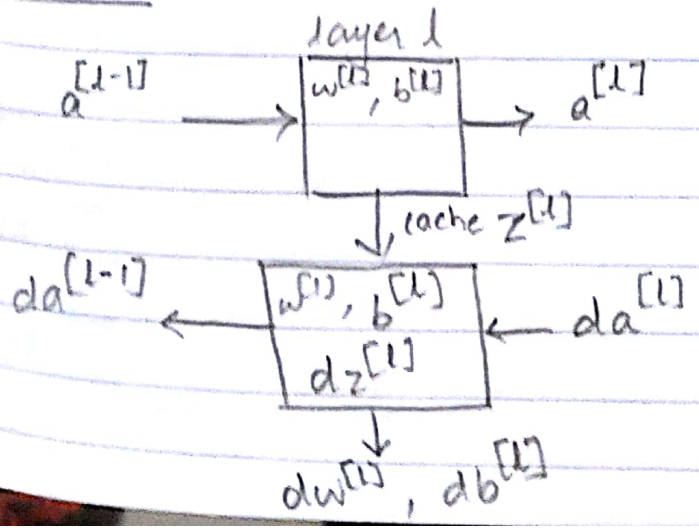
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \quad \& \quad a^{[l]} = g^{[l]}(z^{[l]})$$

We also store the value $z^{[l]}$ (or cache) as it will be useful for back prop step later.

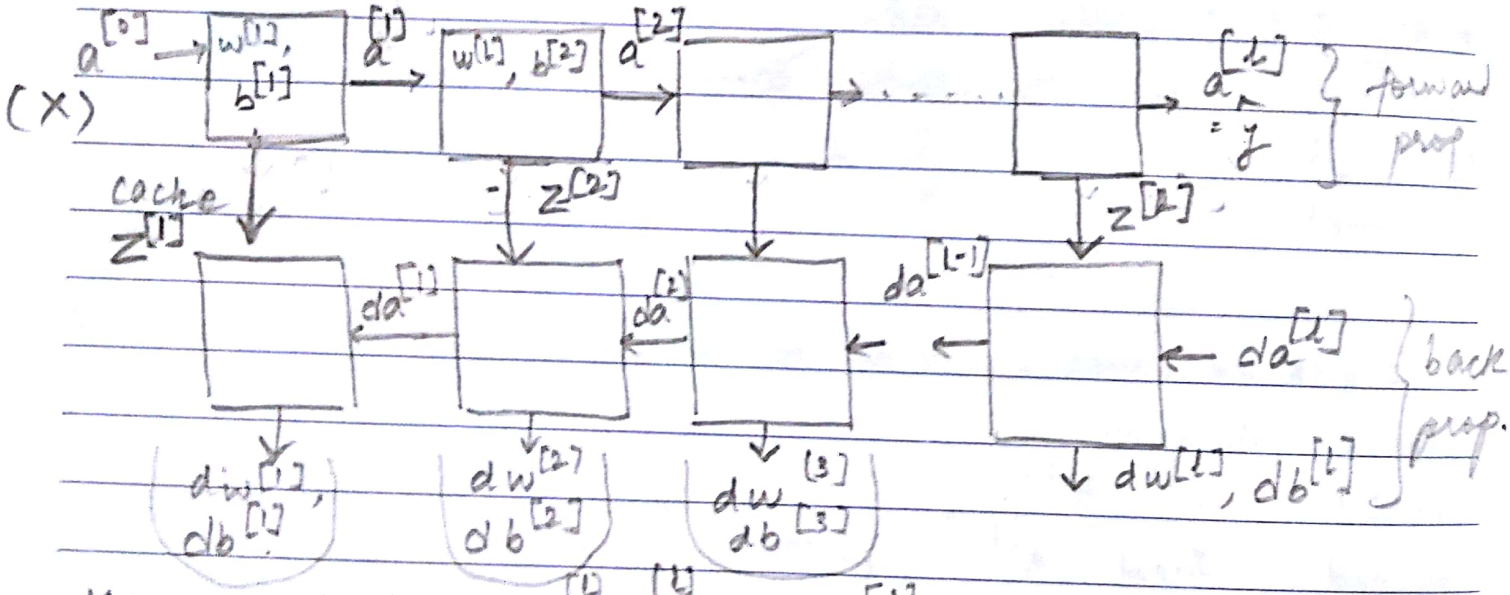
Back prop : Input : $da^{[l]}$; output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

cache : $z^{[l]}$

Summarising



If you can implement these two functions (fwd & back prop), then the basic computation of the NN will be as follows:



then we calculate $w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$

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

This is just one iteration of gradient descent for your N.N.

Though it is written cache $z^{[1]}, z^{[2]} \ldots$ but actually while programming, we cache $z^{[1]}, w^{[1]}, b^{[1]}, z^{[2]}, w^{[2]}, b^{[2]}$, etc..

Forward and Backward Propagation:

We previously saw the basic blocks of implementing a deep N.N - a fwd prop for each layer, and a corresponding back prop step. Let's see how you actually implement these steps. Starting with fwd prop.

Forward propagation for layer l:

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

Output $a^{[l]}$, cache $(z^{[l]})$

eq^{n}s

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

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

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

vectorized

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

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

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

Backward propagation for layer l

Input $da^{[l]}$

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

element-wise product

eq^{n}s

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]}) \quad -①$$

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

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

$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]} \quad -②$$

putting ② in ①, we get

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

vectorized,

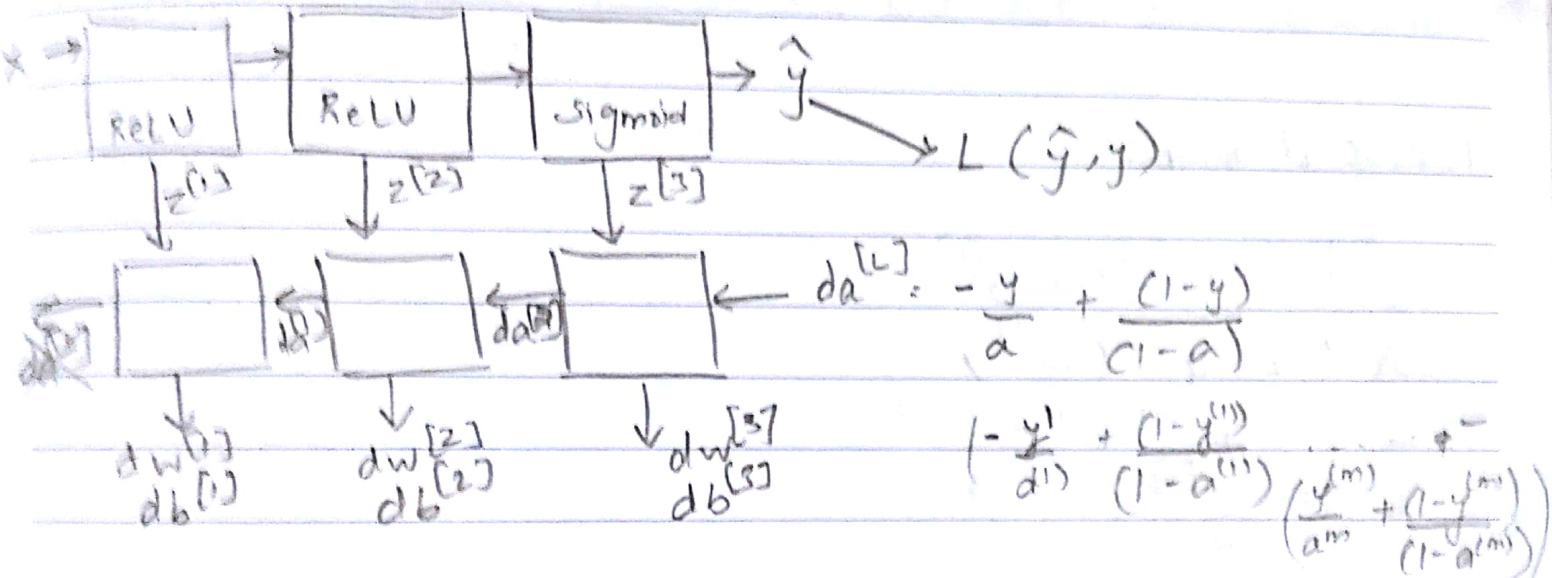
$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$

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

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

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

Summary :



Parameters vs Hyperparameters :

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

Hyper parameters : learning rate, α
iterations of gradient descent
hidden layers, L
hidden units $n^{[1]}, n^{[2]}, \ldots$
choice of activation function

} all these control our parameters W & b

There are more hyper parameters that we'll learn later.

Forward and Backward propagation equations :

Forward prop:

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

1939 13
फाल्गुन / FALGUN
शक / Shaka
4
मार्च / MARCH
रविवार / SUNDAY

Backward prop:

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