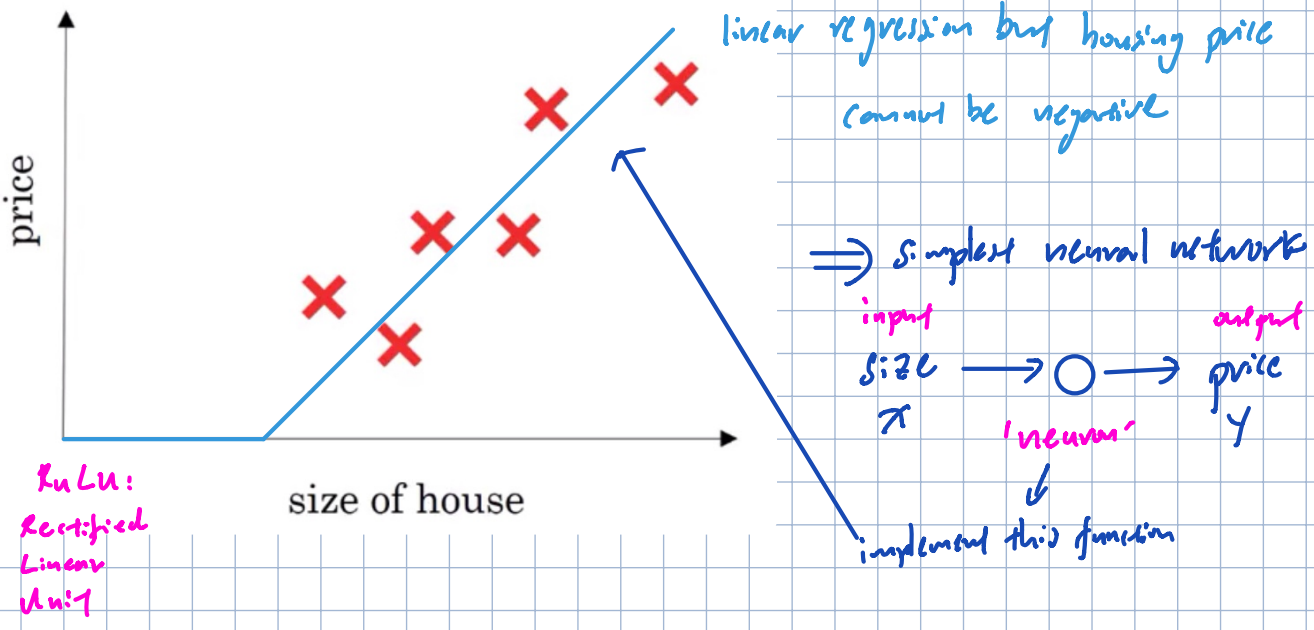
Course 1: Neural Network and Deep Learning

week 1 (Introduction)

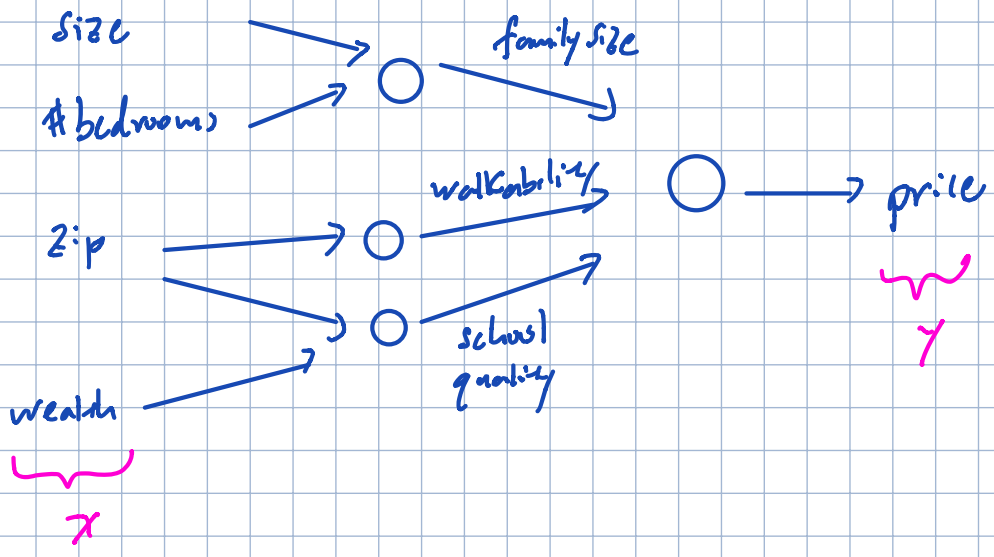
what is Neural Network ?

e.g. housing price prediction

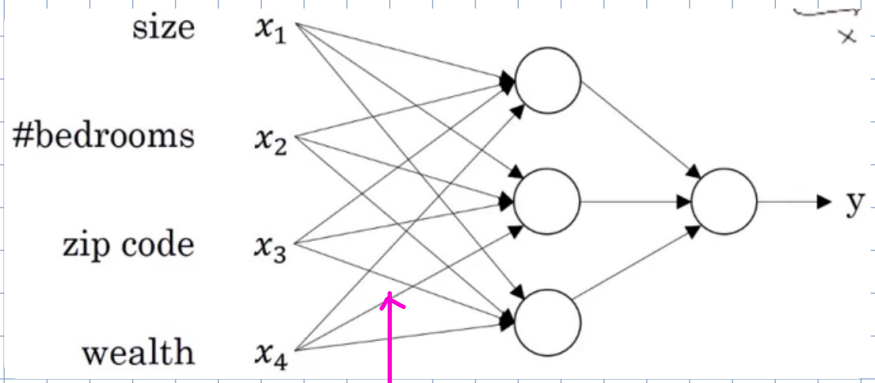


All this neuron does is it inputs the size (x), computes this linear function, takes a max of 0, and then outputs the estimated price.

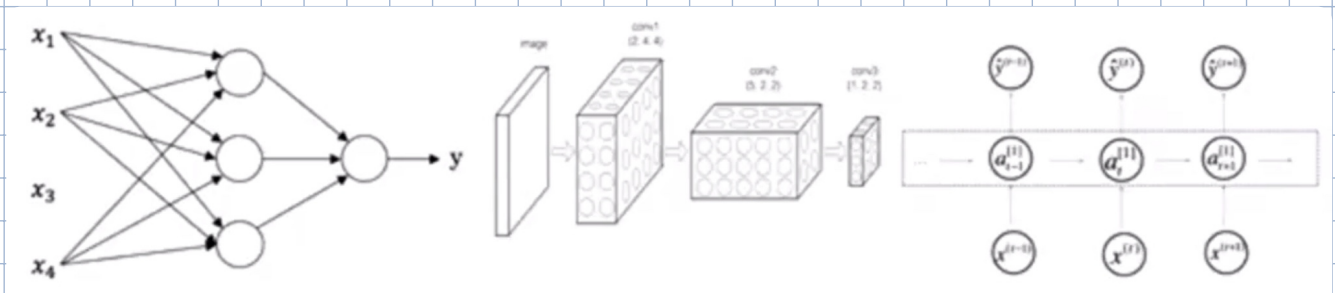
Sorry now we have multiple features: size, no. of bedrooms etc.



A neural network with 4 inputs



Neural Network examples



Standard NN | Convolutional NN | Recurrent NN

↓ image etc

↓ time series sequence

Structured Data vs Unstructured Data

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

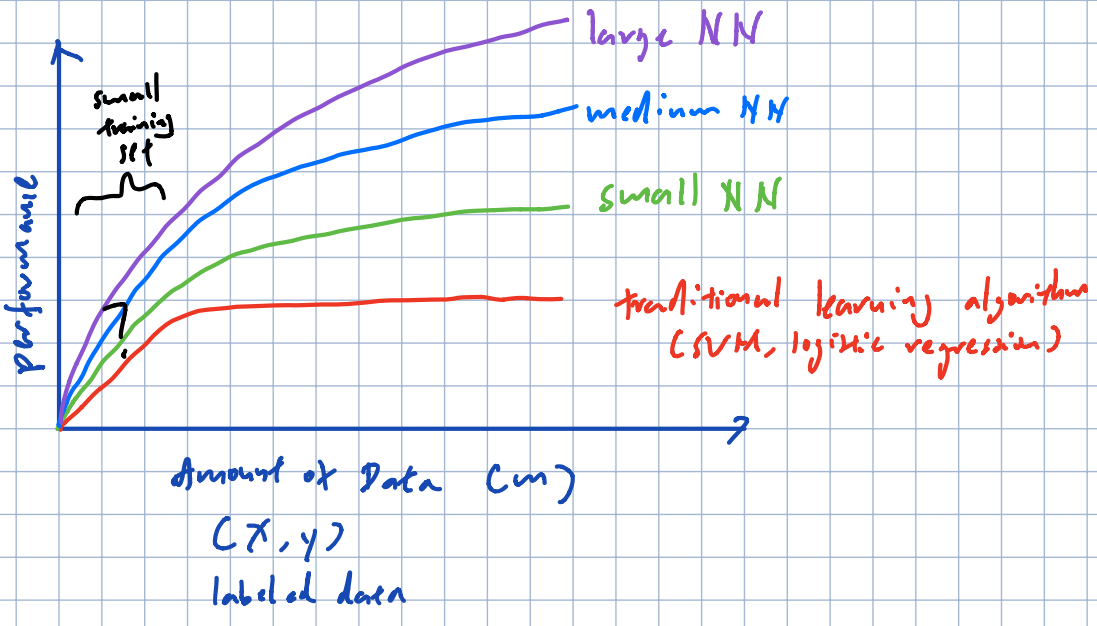
Unstructured Data

Audio | Image

Four scores and seven years ago...

Text

Scale drives deep learning progress



(x, y)
labeled data

week 2 (basics)

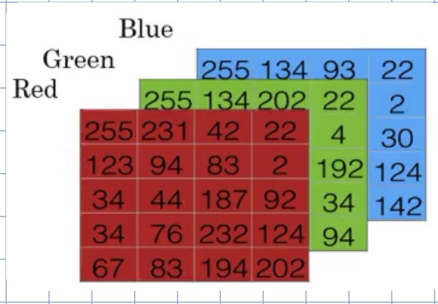
If we have a training set of m training examples, we might use a for loop, step through m training examples.

For neural network, usually want to process an entire training set, w/o using an explicit for loop to loop over the entire training set.

Binary Classification

y

→ 1 (cat) vs 0 (non cat)



unroll →

feature vector

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

if image is 64x64

$n_x = 64 \times 64 \times 3$

$= 12288$

Notation:

(x, y) : a single training example

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

entire training set

m: no. of training examples: $\{(x^{(1)}, y^{(1)}), \dots (x^{(m)}, y^{(m)})\}$

↳ m_{train} : no. of training set examples

↳ m_{test} : no. of test set examples

compact representation of training examples:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x \quad \text{i.e. } X \in \mathbb{R}^{n_x \times m}$$

m

$$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}] \quad \text{i.e. } Y \in \mathbb{R}^{1 \times m}$$

Logistic Regression

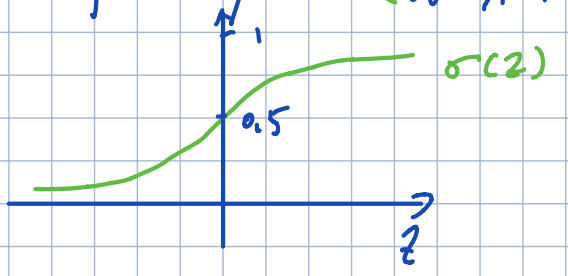
Given an input feature vector X, want output prediction $\hat{y}$, which is an estimate of y.

Formally: $\hat{y} = P(y=1|X)$

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

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

↳ output $\hat{y} = \sigma(w^T x + b)$



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

If z is large, $\sigma(z) \approx 1$

z is small, $\sigma(z) \approx 0$
(or large -ve)

sometimes, the intercept, b, is incoporated in feature

i.e. $x_0 = 1$, $x \in \mathbb{R}^{n+1}$

↳ $\hat{y} = \sigma(\theta^T x)$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \\ \\ \\ \} w \end{matrix}$$

For neural network, it is easier to separate these parameters.

Logistic Regression Cost Function

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

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

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

Loss (error) function:

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

Generally Don't use this → optimisation problem becomes non-convex

i.e. gradient descent will find local optima (i.e. ∪∩∪)

option 2: (actual loss function for logistic regression)

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

for optimisation problem, we want $\mathcal{L}$ as small as possible

If $y=1$, $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ⟵ want $\log \hat{y}$ large

↳ $\hat{y}$ large (can never exceed 1)

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

↳ $\hat{y}$ small

(can never below 0)

* Loss function was defined w.r.t. a single training example

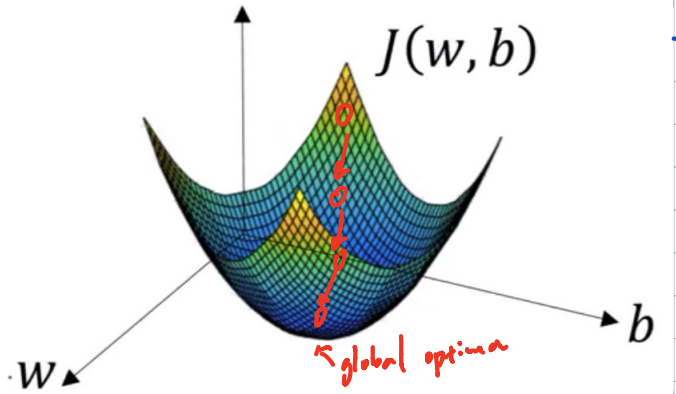
* Cost function is w.r.t the entire training set

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

$\hat{y}^{(i)}$: predict using a set of w, b

Gradient Descent

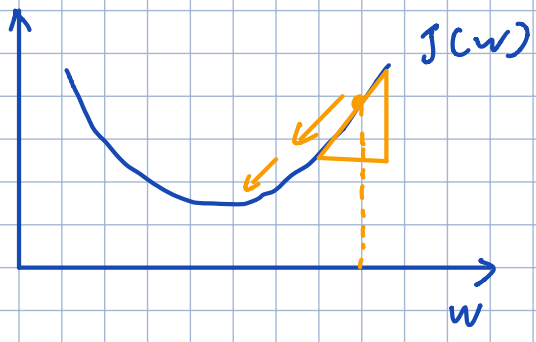
From the above cost function, we want to find w, b that minimise $J(w, b)$



→ a convex function J(w,b)

↳ reason to use this J for logistic regression

Say we have some function J(w) (ignore b for now)



Gradient Descent:

Repeat {

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

(learning rate)

}

till algorithm converges

For logistic regression, J(w, b)

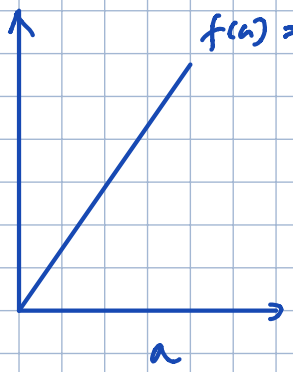
Repeat {

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

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

}

Derivatives



a = 2 , f(a) = 6

a = 2.001, f(a) = 6.003

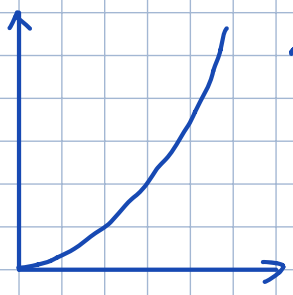
slope (derivatives) of f(a) at a = 2 is 3

$\frac{df(a)}{da} = 3$

a = 5 , f(a) = 15

a = 5.001 , f(a) = 15.003

slope at a = 5, is also 3



$f(a) = a^2$

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

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

slope at $a=2$ is 4 (i.e. $\frac{0.004}{0.001} = 4$)

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

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

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

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

$\frac{d}{da} a^2 = 2a$ ←

Computation Graph

say we have a function

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

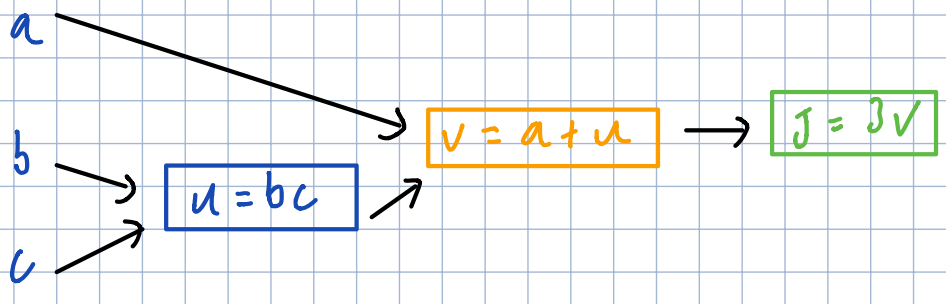
u, v, J

computation takes 3 steps:

$u = bc$

$v = a + u$

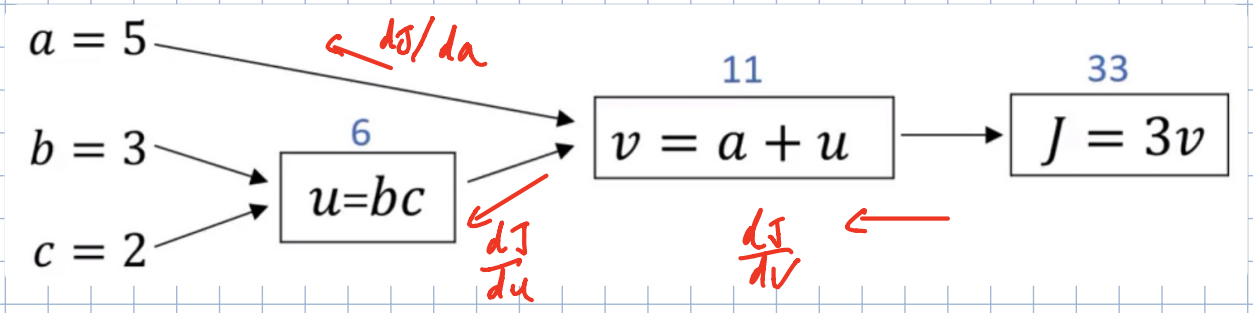
$J = 3v$



computation: Left → Right

derivative: Right → Left

Computing Derivatives



Say we want to compute: $\frac{dJ}{dv}$

i.e. if we change value of v by a little bit, how will value of J change?

Now: $J = 3v$

$v = 11 \longrightarrow$ if $v = 11.001$

$\hookrightarrow J = 33.003$ $\Big\}$ $\frac{dJ}{dv} = 3$ $\left(\because \frac{0.003}{0.001} = 3\right)$

How about $\frac{dJ}{da}$?

Now: $a = 5 \longrightarrow$ if $a = 5.001$

$\hookrightarrow v = a + u = 11.001$ $\Big\}$ $\frac{dJ}{da} = 3$

$J = 33.003$

$\Rightarrow \frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da}$ a cause v to change; v cause J to change

have $\frac{dv}{da} = 1$ $(\because a \uparrow 0.001,\ v \uparrow 0.001)$

How about $\frac{dJ}{du}$?

$u = 6 \longrightarrow$ if $u = 6.001$

$\hookrightarrow v = 11.001$

$\hookrightarrow J = 33.003$

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

($\frac{dJ}{dv} = 3$, $\frac{dv}{du} = 1$)

How about $\frac{dJ}{db}$?

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

($\frac{dJ}{du} = 3$, $\frac{du}{db} = 2$)

if $b = 3 \longrightarrow 3.001$

$\hookrightarrow u = b \times c = 6 \longrightarrow 6.002$

i.e. $\frac{du}{db} = 2$

Logistic Regression Gradient Descent (for 1 training example)

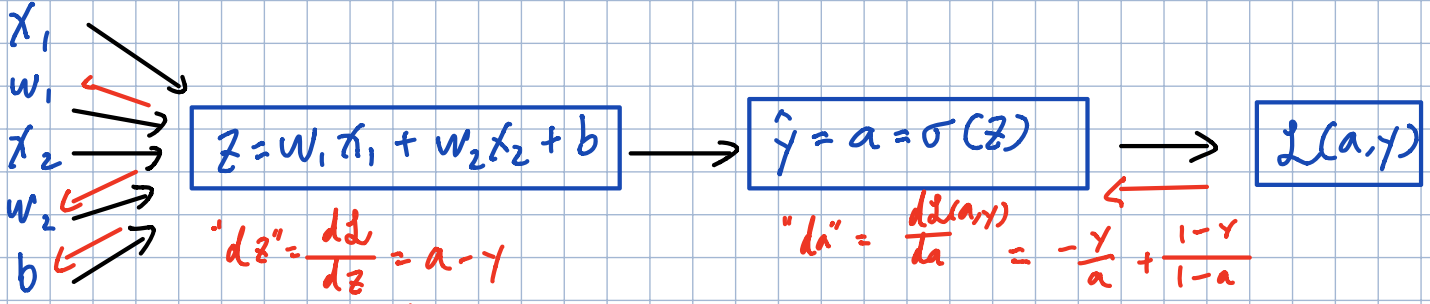
Recap:

$z = w^T x + b$

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

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

Says we have 2 features: x_1, x_2



In logistic regression, we want to modify w and b, in order to reduce this loss $\mathcal{L}$.

$\frac{\partial \mathcal{L}}{\partial w_1} =$ "dw_1" $= x_1 dz$ ($dz = \frac{d\mathcal{L}}{dz}$, $\frac{dz}{dw_1}$)

$dw_2 = x_2 \cdot dz$ ($\frac{d\mathcal{L}}{dz}$)

$db = dz$ ($\frac{d\mathcal{L}}{dz}$)

i.e. compute dz, then can get dw_1, dw_2, db

↳ $w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

Note: these are for 1 training example

i.e. they are actually

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$ for $(x^{(i)}, y^{(i)})$

Gradient Descent on m examples

Recall the cost function for logistic regression

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

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

For m training examples, the derivatives w.r.t the parameters $(w_1, w_2, \dots, b)$ is also going to be the average of derivates w.r.t to the parameters.

e.g.

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

for $(x^{(i)}, y^{(i)})$

The Algorithm (1 step of gradient descent)

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

For $i = 1$ to m:

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

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

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

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

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

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ } $n = 2$

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

$J /= m$

$dw_1 /= m;\ dw_2 /= m;\ db /= m;$ # compute average

Here:

$dw_1 = \frac{\partial J}{\partial w_1}$ (accumulators to sum over $dw_1^{(i)}$ for the entire training set)

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

Everything here is just for 1 step of gradient descent

* In practice, avoid using explicit for loop, use vectorisation.

Derivation of $\frac{dL}{dz} = a - y$ ($a = \hat{y}$)

By chain rule:

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

Step 1: $\frac{dL}{da}$

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

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

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

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

Step 2: $\frac{da}{dz}$

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

The derivatives of a sigmoid has the form

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

↳ since $a = \sigma(z)$

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

step 3: $\frac{dL}{dz}$

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

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

$$= a - y$$

Vectorisation

Recall in logistic regression, we need to compute

$$z = w^T x + b \qquad \text{where } w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$$

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

non-vectorised:

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

z += b
```

vectorised

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

$w^T x$

In neural network, whenever possible, avoid explicit for-loop.

e.g.

$u = Av$

vector — matrix — vector

by definition: $u_i = \sum_j A_{ij} v_j$

non-vectorisation:

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

vectorisation:

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

e.g. apply exponential on every element of a matrix / vector

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

non-vectorisation

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

vectorisation

$u = np.exp(v)$

Logistic Regression Derivatives (not really needed, see the one below)

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

dz here is $\frac{dL}{dz}$ (difference from previous slide)

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

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

$= a - y$

2 features also need for loop for large n

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

make dw a vector: $dw = np.zeros((n_x, 1))$

↳ $dw += x^{(i)} dz^{(i)}$

↳ $dw /= m$

Vectorising Logistic Regression

If we have m training examples, then to make predictions on the first example: predict on second example: third...

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

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

So this is the forward propagation for m training examples.

We first try to compute all the zs and as

Vectorisation:

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

i.e. $\mathbb{R}^{n_x \times m}$

construct: $Z = [z^{(1)}\ z^{(2)}\ \cdots\ z^{(m)}]$

$= w^T X + [b\ b\ \cdots\ b]$ ($1 \times m$)

w^T is a row vector $[\ \ ]$ (n_x)

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

To implement this: `Z = np.dot(w.T, X) + b`

`b`: just a number ↳ broadcast to $1 \times m$

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

Vectoring Logistic Regression's Gradient Computation

Perform gradient computation on m training examples at the same time.

for the first training example: $dz^{(1)} = \frac{dL}{dz^{(1)}} = a^{(1)} - y^{(1)}$

2nd training example: $dz^{(2)} = a^{(2)} - y^{(2)}$ ...

Define a new variable

$dZ = [dz^{(1)}\ dz^{(2)}\ \cdots\ dz^{(m)}]$

$1 \times m$

Recall we have computed

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

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

In the for loop, we still has to loop through m training examples for dw, db:

$dw = 0$
$dw \mathrel{+}= x^{(1)} dz^{(1)}$
$dw \mathrel{+}= x^{(2)} dz^{(2)}$
$\vdots$
$dw \mathrel{/}= m$

$db = 0$
$db \mathrel{+}= dz^{(1)}$
$db \mathrel{+}= dz^{(2)}$
$\vdots$
$db \mathrel{/}= m$

Here dw is already a vectors so no need loop through $dw_1, dw_2 \dots$ (for each feature)

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

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

adding up all the dw terms, for each features

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

$n \times 1$

Previously, inefficient non-vectorized version

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

get rid of this for loop

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

on all m training example

$Z = w^T X + b$
$= \text{np.dot}(w.T, X) + b$
$A = \sigma(Z)$
$dZ = A - Y$
$dw = \frac{1}{m} X dZ^T$
$db = \frac{1}{m}\,\text{np.sum}(dZ)$
$\hookrightarrow w := w - \alpha\, dw$ — 1 iteration!
$b := b - \alpha\, db$

Broadcasting in Python

Calories from Carbs, Proteins, Fats in 100g of different foods:

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

$= A$ (3,4)

↓

59cal : carbs: $56/59 \approx 95\%$

Say we want to calculate % of calories from Carbs, Protein and fats for each of the 4 foods.

i.e. Sum up each col to get total number of calories in each food.

then divide through out the matrix

```
cal = A.sum(axis = 0)
```

0 ↓ 1 →

actually here it is redundant

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

(3×4)/(1×4)

e.g.

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

e.g.

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

(m×n) (1,n) → (m,n) copy

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

e.g.

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

(m,n) (m,1) → (m,n)

$$= \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$

General Principle

If we have a (m,n) matrix:

(m,n) $+ - \times /$ $(m,1) \rightsquigarrow (m,n)$, $(1,n) \rightsquigarrow (m,n)$, then apply the operation element wise

Numpy Vectors

```
a = np.random.randn(5)
```

↳ a.shape = (5,) ⇒ rank 1 array } Do not use

```
a = np.random.randn(5,1)
b = np.random.randn(1,5)
```

} consisting and easy to understand

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

```
a = a.reshape((5,1))
```

convert to proper vector/matrix

Explanation of logistic regression cost function

To recap:

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

Interpret $\hat{y} = p(y=1|x)$

Another way of saying: if $y=1$, then $p(y|x) = \hat{y}$

if $y=0$, then $p(y|x) = 1-\hat{y}$

↳ $p(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$

If $y=1$, $p(y|x) = \hat{y}^{1}(1-\hat{y})^{0} = \hat{y}$

If $y=0$, $p(y|x) = 1-\hat{y}$

Log function is a strictly monotonically increasing function, so maximise $\log p(y|x)$ is similar to maximise $p(y|x)$:

$$\log p(y|x) = \log \hat{y}^{y}(1-\hat{y})^{(1-y)} = y\log\hat{y} + (1-y)\log(1-\hat{y})$$
$$= -\mathcal{L}(\hat{y}, y)$$

minimise $\mathcal{L}$ ⇒ maximise $\log p(y|x)$

Cost on m examples

The probability of all the labels in the training set:

$$p(\text{labels in training set}) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$

↳ same as maximising

$$\log p(\text{labels in training set}) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$
$$= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)})$$
$$= -\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Principle of maximum likelihood estimation

$$= -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

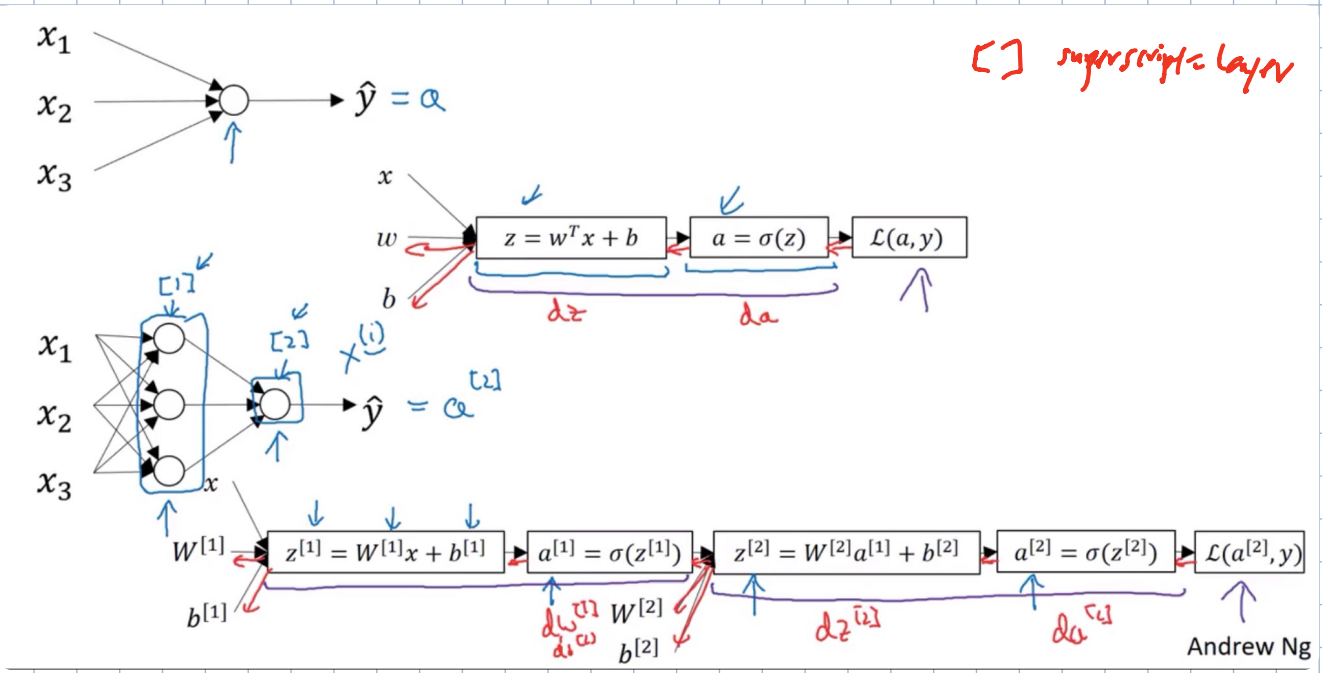
carry out maximum likelihood on logistic regression model $\log p(y|x)$

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

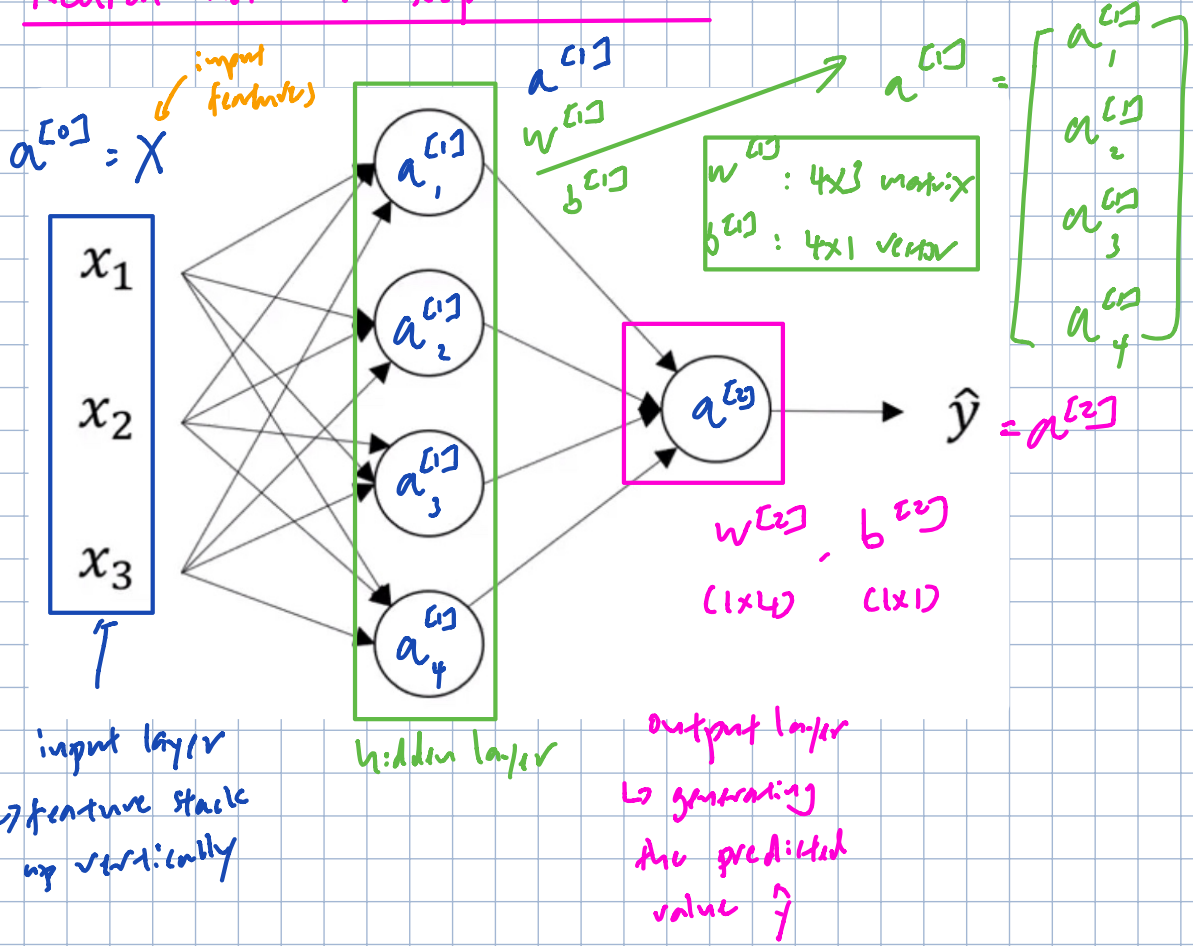
week 3 (Shallow Neural Network)

Neural Network Overview

[] superscript: layer



Neural Network Representation



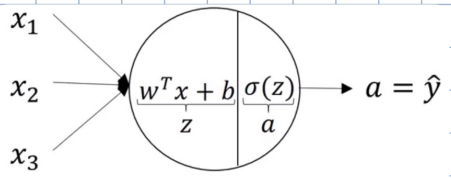
→ The term "hidden layer" refers to the fact that in the training set, the true values for these nodes are not observed.

→ The above is called a 2-layer NN, because we usually don't count input layer. Input layer is sometimes called layer 0.

→ The hidden layer and output layer will have parameters associated with them.

↳ w, b

Computing a NN's output



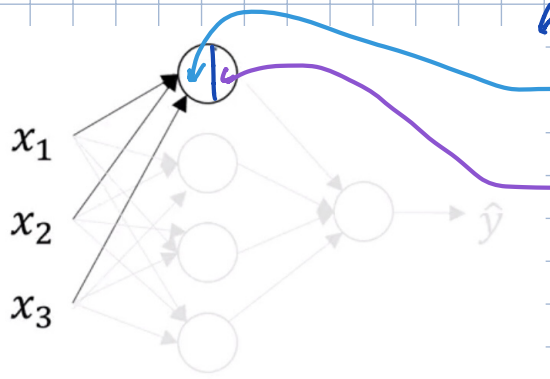
← logistic regression

$z = w^T x + b$

$a = \sigma(z)$

In NN, we repeat this calculation many times.

Neural network



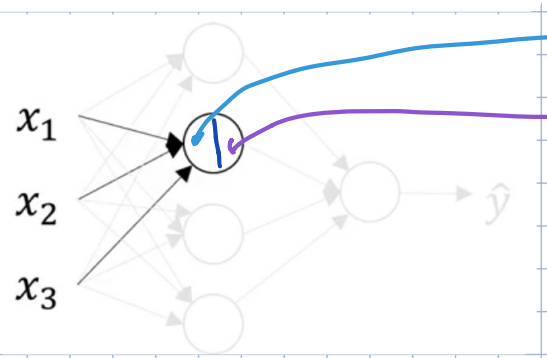
Lets focus on the first node in the hidden layer

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

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

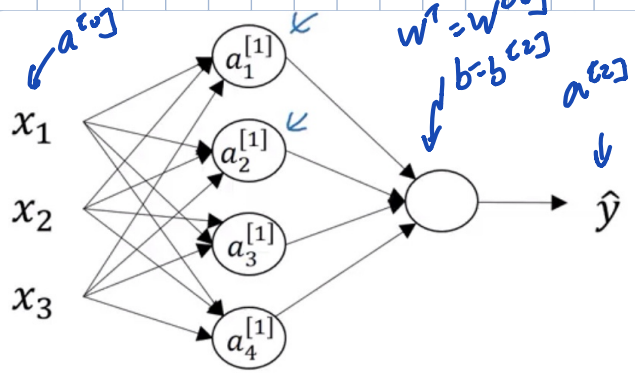
notation: $a_i^{[l]}$ ← layer, ← node in that layer

Second node in layer 1



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

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



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

Compute $z^{[1]}$ in vectorized manner:

$$\begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]} x + b_3^{[1]} \\ w_4^{[1]} x + b_4^{[1]} \end{bmatrix}$$

$(4,3)$

$W^{[1]}$

$b^{[1]}$

$$= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$$

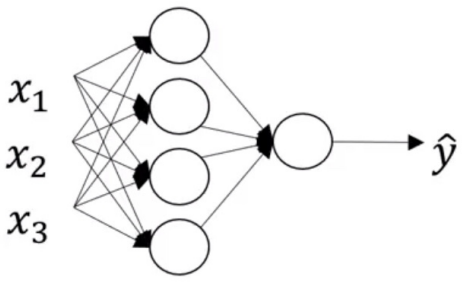
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Given input x:

$$z^{[1]} = W^{[1]} x + b^{[1]} \qquad (4,1) \quad (4,3)(3,1) \quad (4,1)$$
$$a^{[1]} = \sigma(z^{[1]}) \qquad (4,1) \quad (4,1)$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \qquad (1,1) \quad (1,4)(4,1) \quad (1,1)$$
$$a^{[2]} = \sigma(z^{[2]}) \qquad (1,1) \quad (1,1)$$

This is for $\frac{1}{=}$ training example.

Vectorising across m training examples



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

$x \longrightarrow a^{[2]} = \hat{y}$ using $\int$ for single training sample.

If we have m training samples, we need to have

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

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

$\vdots$

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

notation: $a^{[2](i)}$ — $[2]$: layer 2; (i): training example i

unvectorised version:

for i = 1 to m:

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

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

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

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

rewrite

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

Recall we define the matrix X:

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

(n_x, m)

To vectorise this for loop for m training samples:

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

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

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

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

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

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

↕ nodes

→ training sample

Explanation for vectorised implementation

Let's go through part of the propagation calculation for the few examples.

For the first train sample: $z^{[1](1)} = w^{[1]} x^{(1)} + b^{[1]}$; for second train sample: $z^{[1](2)} = w^{[1]} x^{(2)} + b^{[1]}$; for the third training sample $z^{[1](3)} = w^{[1]} x^{(3)} + b^{[1]}$

$b^{[1]} \to 0$

for simplification purpose

$w^{[1]}$ = [ ] (same matrix) $\Rightarrow w^{[1]} x^{(1)} = [\ ]$ $w^{[1]} x^{(2)} = [\ ]$ $w^{[1]} x^{(3)} = [\ ]$

(column vector)

Now consider training set with 3 samples:

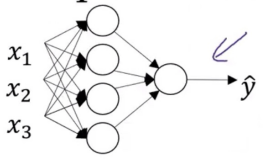
$$w^{[1]} \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \cdot & \cdot & \cdot \\ | & | & | \end{bmatrix}$$

X

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

↳ If we stack up X in cols, then Z will also stack up, for my training examples.

Recap of vectorizing across multiple examples



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

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

```
for i = 1 to m
```

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

} non-vectorized

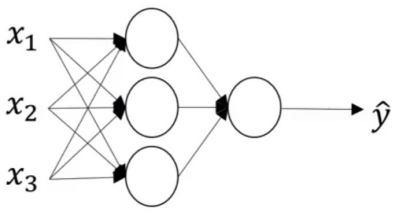
$Z^{[1]} = W^{[1]} X + b^{[1]}$ ($b^{[1]}$ → $B^{[1]}$)
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

} vectorized

vectorize w.r.t. samples w/o for loop

Andrew Ng

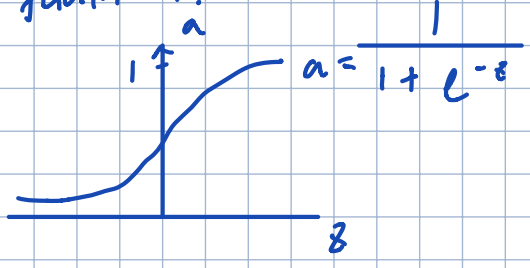
Activation Functions



Given x:

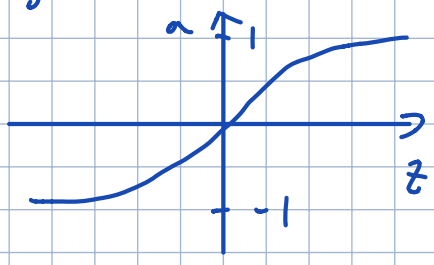
$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

In forward propagation, there are steps we used sigmoid function as the activation function.



→ more generally, we can have a function $g(z^{[1]})$, which is a non-linear function.

e.g. tanh



$a = \tanh(z)$

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

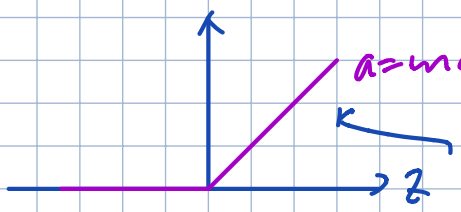
For hidden layer, tanh is almost always better than the sigmoid function, because the values between +1 and -1, the mean of the activations are closer to having a 0 mean. When we pre-process data, we might center the data.

↳ for output layer, if y is either 0 or 1, then sigmoid make sense as $0 \leq \hat{y} \leq 1$.

* activation function can differ from layer to layer: $g^{[1]}(z^{[1]})$ ← layer 1

If z is either very large or very small, then the gradient of the function is small.

ReLU:



this slope makes ReLU learning faster than tanh

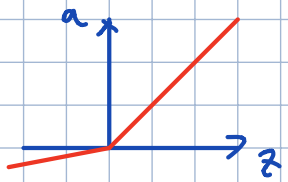
Some guideline:

* If output is 0/1, binary classification, sigmoid function for output layer
* All other units, use ReLU

Disadvantage as ReLU:

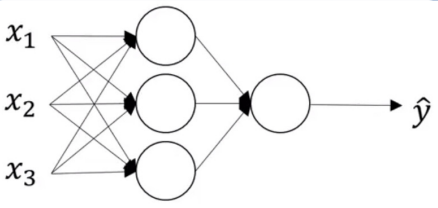
→ Gradient = 0 if z is negative

↳ leaky ReLU:



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

why activation function?



In forward propagation, we use activation function.

If we use linear activation function:

$g(z) = z$ (practically do nothing)

If we do this, then this model just computing $\hat{y}$ as a linear function of input feature.

Given x:

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

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

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

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

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

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

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

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

$$= W' X + b'$$

i.e. neural network just outputting a linear function of the input.

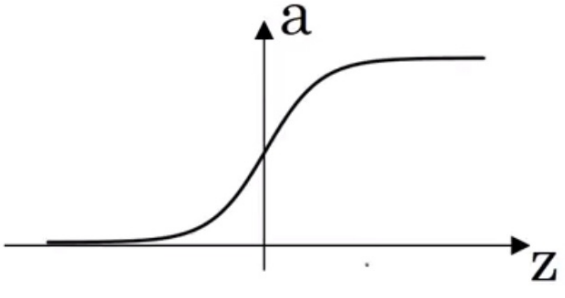
↳ no matter how many layers ⇒ output is just linear function of the input.

For Regression problem, the output layer <u>may be</u> can use linear activation function, to output a real value.

Derivatives of activation functions

when implementing back propagation, need slope of the activation function.

Sigmoid function



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

slope of $g(z)$ at z: $\frac{d}{dz} g(z) = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$

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

Sanity check: if z is large e.g. $z = 10$, $g(z) \approx 1$

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

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

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

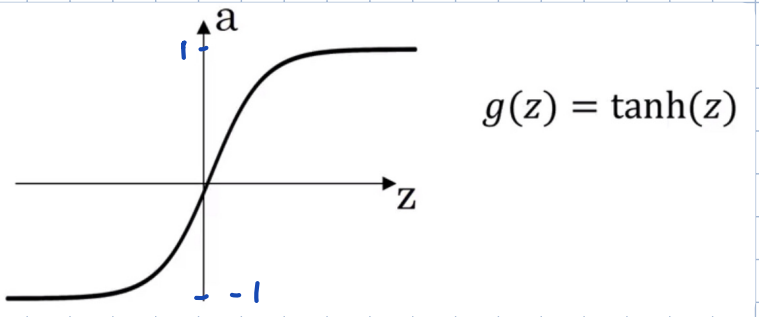
if $z = 0$, $g(z) = 1/2$ $\Rightarrow \frac{d}{dz} g(z) = \frac{1}{2}(1-1/2) = 1/4$

Notation: $g'(z) = \frac{d}{dz} g(z)$

In neural network : $a = g(z) = \frac{1}{1+e^{-z}}$

$\hookrightarrow \frac{da}{dz} = a(1-a) = g'(z)$

Tanh function



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

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

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

If $a = g(z)$

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

sanity check : $z = 10$ $\tanh(z) \approx 1$

$\hookrightarrow g'(z) \approx 0$

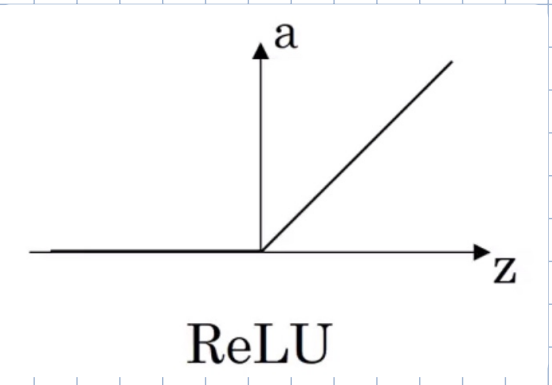
$z = -10$ $\tanh(z) \approx -1$

$\hookrightarrow g'(z) \approx 0$

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

$\hookrightarrow g'(z) = 1$

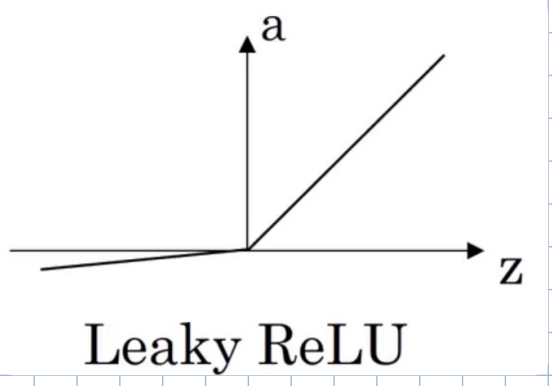
ReLU and Leaky ReLU



ReLU

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

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



Leaky ReLU

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

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

Gradient Descent for Neural Networks

A neural network with a single hidden layer, will have parameters

$$W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$$

Recall that we have:

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

input features — hidden units — output units

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

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

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

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

Cost function: (assume binary classification)

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

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

To train these parameters, need to perform gradient descent:

* need to initialize parameters randomly (rather than to all 0s)

Repeat {

compute predict ($\hat{y}^{(i)}$, for $i = 1 \dots m$)

$$dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}}, \quad dw^{[2]}, \quad db^{[2]}$$

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

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

$$w^{[2]} := w^{[2]} - \alpha\, dw^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha\, db^{[2]}$$

}

true value

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

To get all the derivatives:

forward propagation:

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

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

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

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

for binary classification

Back propagation:

assume Sigmoid

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

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

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

horizontally

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

have this 1x1 number

∟ so far similar to logistic regression

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

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

element-wise

derivative of whatever activation function on layer 1

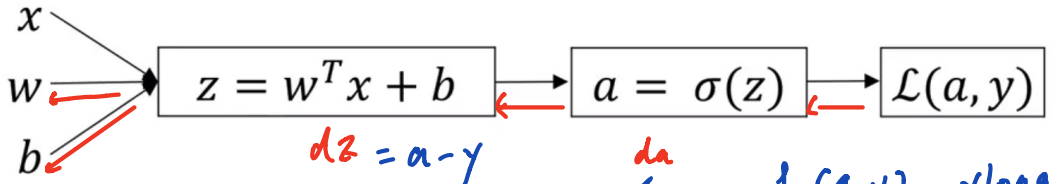
$(n^{[1]}, 1)$

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

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

Back propagation

Logistic regression



$dz = a - y$

da

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

$dw = dz \cdot x$

$db = dz$

$dz = da \cdot g'(z)$

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

$\hat{y}$

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

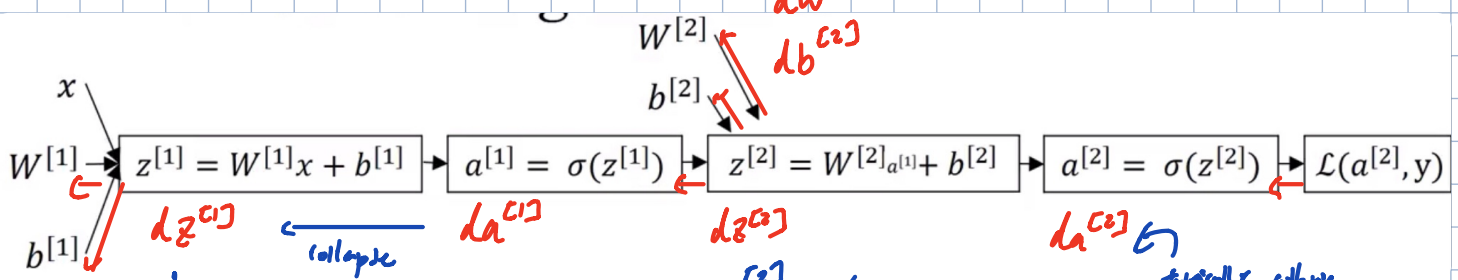
for 1 train example

for w just stack horizontally

$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{da}{dz}$

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

Neural Network Gradients



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

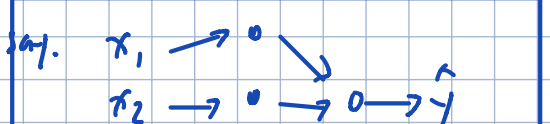
typically collapse with $dz^{[2]}$

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

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

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

$\frac{dx}{x}$ same dim



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

for 1 training example

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

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

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

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

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

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

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

Summary of gradient descent

on training, stack vertically

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

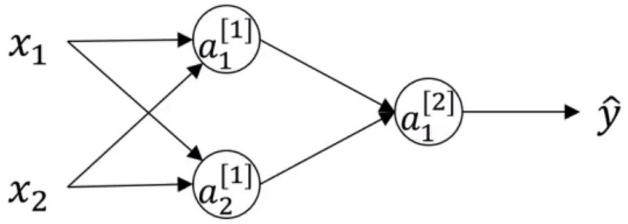
1 training	m training
$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$ $(n^{[1]}, 1)$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$ $(n^{[1]}, m)$ $(n^{[1]}, m)$ $(n^{[1]}, m)$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

element-wise product

Random Initialisation

when training NN, it is important to initialise the weights randomly.

What happens if you initialize weights to zero?



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

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

$W^{[1]}$: problem; $b^{[1]}$: OK

$\hookrightarrow a_1^{[1]} = a_2^{[1]} \Rightarrow dz_1^{[1]} = dz_2^{[1]} \Rightarrow$

$$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$$

each row is same

updates same for both nodes.

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

Need to initialize the weight randomly

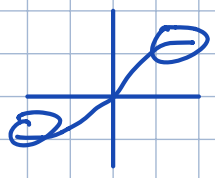
initialize to small random values

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

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

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

$b^{[2]} = 0$



keep value off the flat part

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

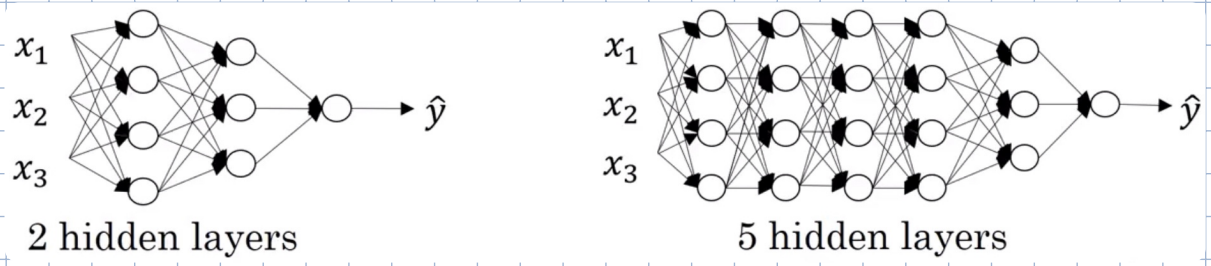
$a^{[1]} = g^{[1]}(z^{[1]})$

if not, gradient descent will be slow

For deep network, we want choose different parameter 0.01.

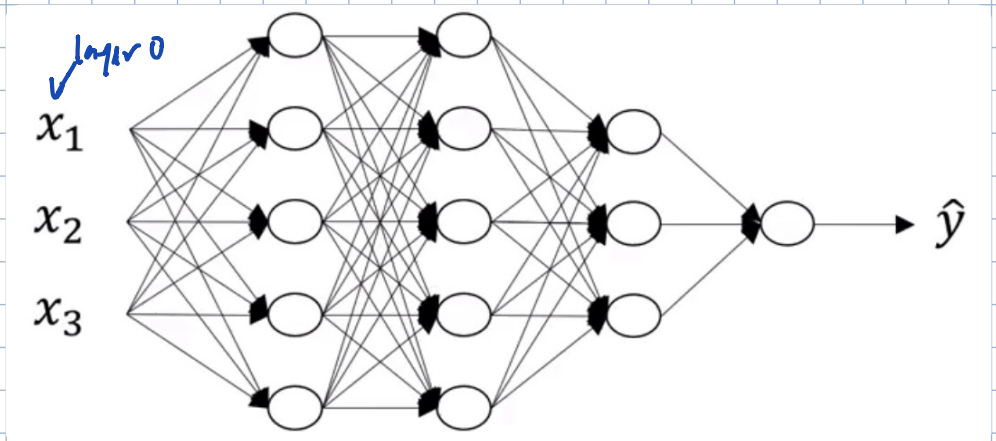
week 4 (Deep Neural Networks)

Deep L-layer neural network



Deep neural network notation

4 layer NN (3 hidden)



$X = a^{[0]}$

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

L = no. of layers = 4

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

↳ $n^{[1]} = 5$, $n^{[2]} = n_x = 3$

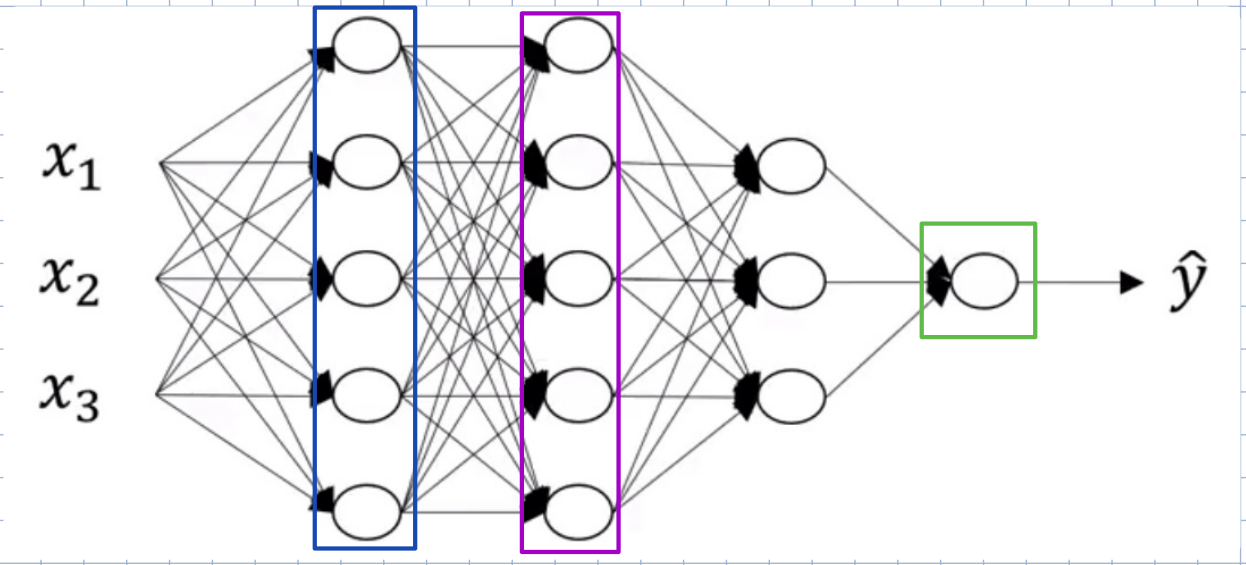
$n^{[3]} = 3$, $n^{[4]} = n^{[L]} = 1$

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

↳ $a^{[l]} = g^{[l]}(z^{[l]})$

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

Forward Propagation in a Deep Network



for a single training example x

for the first layer

$$z^{[1]} = W^{[1]} x + b^{[1]} \quad (a^{[0]} = x)$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$
$$\vdots$$
$$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$$
$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$$

in general:

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

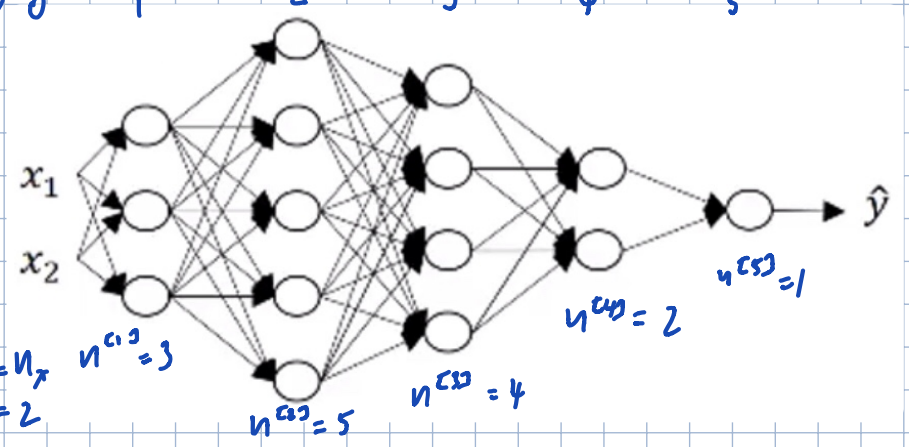
vectorized (on training examples):

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

for $l = 1$ to L

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

Getting the matrix dimension right



$L = 5$

↳ 4 hidden layers + 1 output layer

Forward propagation:

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

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

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

independent of m

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

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

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

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

↳ $w^{[3]} : (4, 5)$

↳ $w^{[4]} : (2, 4)$

↳ $w^{[5]} : (1, 2)$

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

$$\Downarrow$$

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

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

$$\Downarrow$$

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

Note that $a^{[l]} = g^{[l]}(z^{[l]}) \Rightarrow z^{[l]}$ and $a^{[l]}$ same dimension in this type of network.

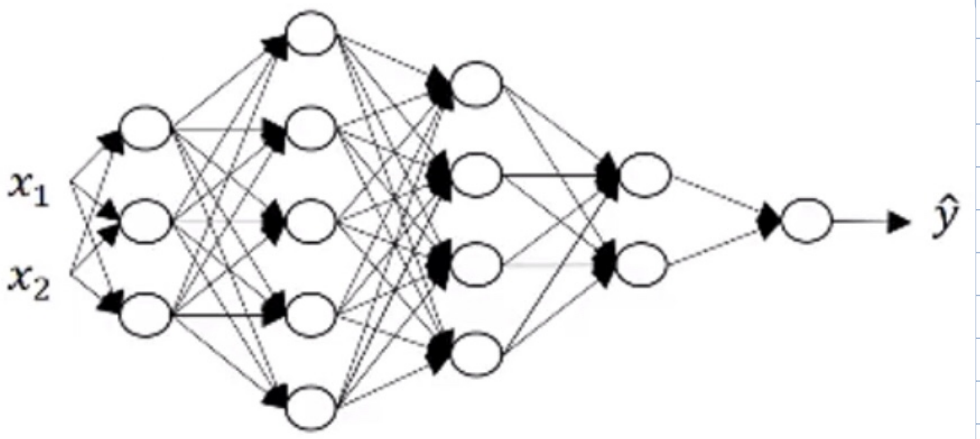
$$a^{[l]} : (n^{[l]}, 1)$$
$$z^{[l]} : (n^{[l]}, 1)$$

1 training example!

vectorised version (m training example)

dimension of W, dW and db remains the same for vectorised version,

but dimension of Z, a, X will change



single example:

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

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

m examples:

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

$[n^{[0]}, m]$

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

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

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

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

Python broadcast

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

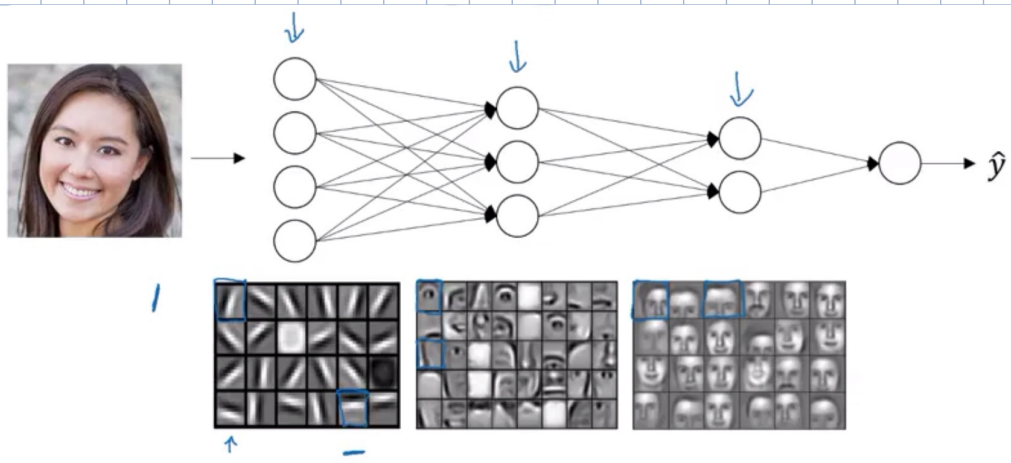
$Z^{[1]}, A^{[1]} : (n^{[1]}, m)$

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

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

why deep representation?

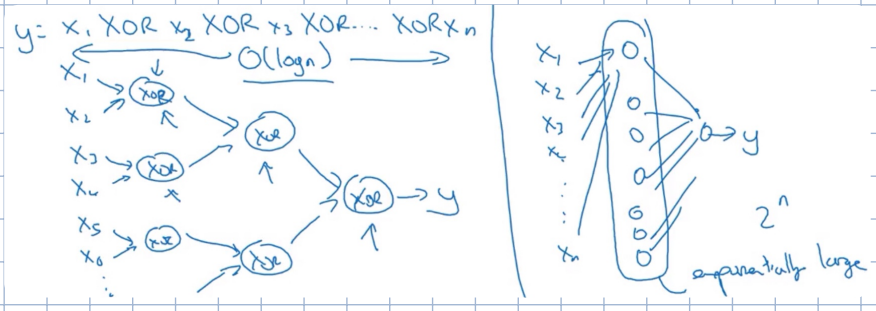
Useful networks have many hidden layers, why?



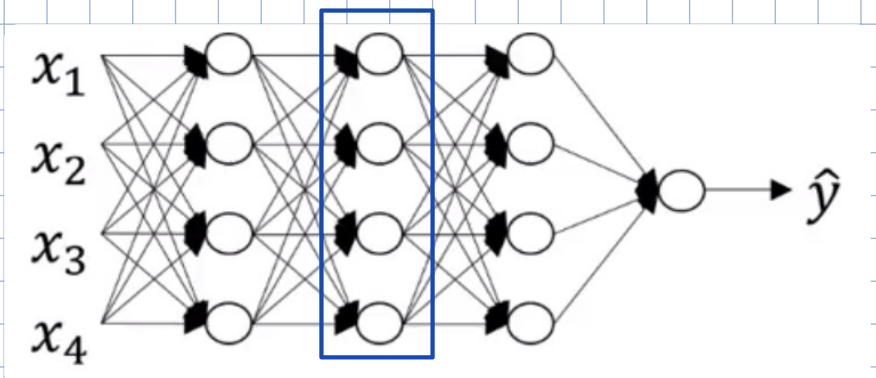
Circuit theory and deep learning

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.

less hidden units



Building blocks of deep neural networks



Say we focus on this particular layer

$\Rightarrow$ for layer l: we have some parameters $W^{[l]}, b^{[l]}$

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

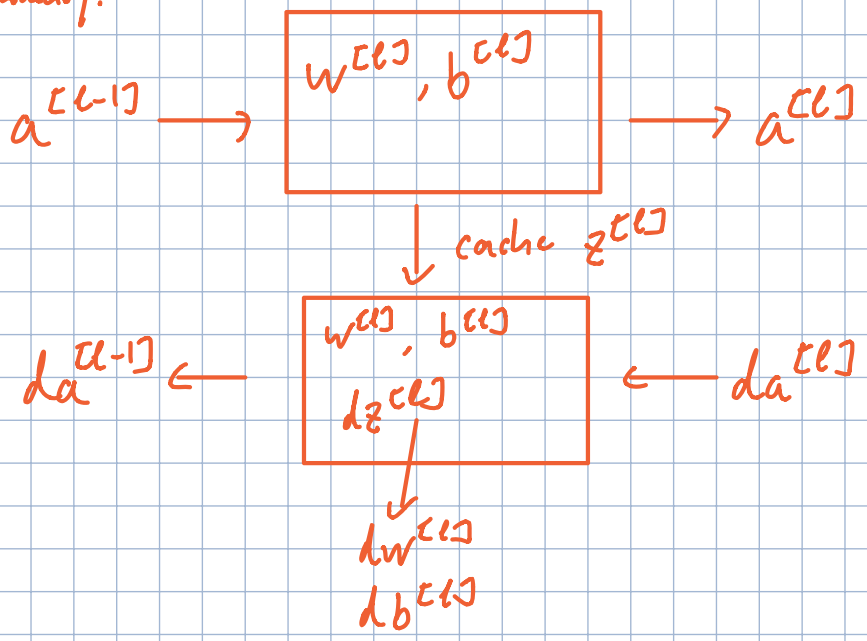
(using: $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ cache $z^{[l]}$ for later use

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

Backward: Input: $da^{[l]}$, cache($z^{[l]}$) output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

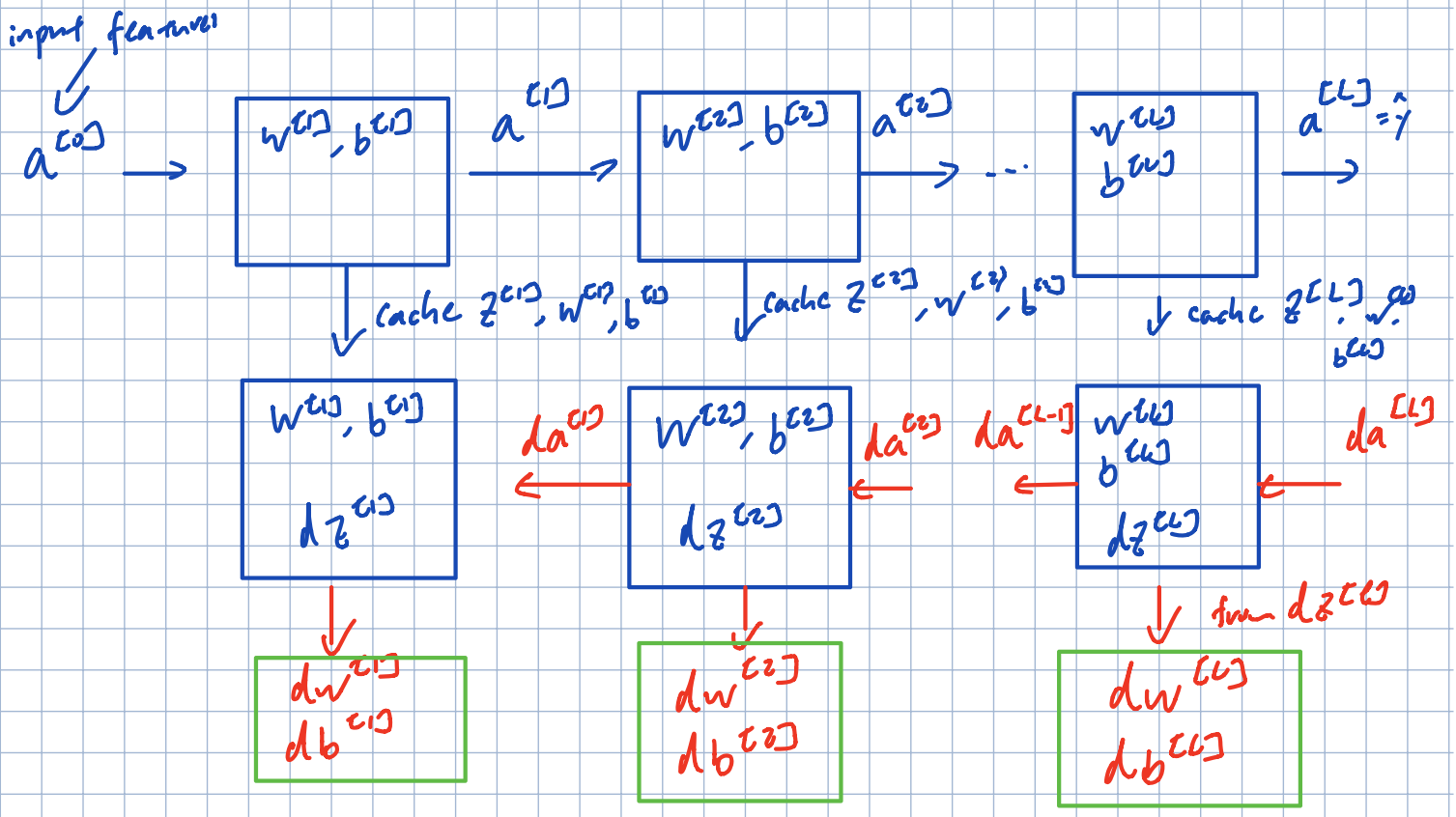
Summary:

layer l



given the derivatives w.r.t. these activations (i.e. da)

Basic computation of NN



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

1 forward, 1 backward, update w, b ← 1 iteration of gradient descent

Forward and Backward propagation

Forward for layer l:

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

output $a^{[l]}$, cache ($z^{[l]}, w^{[l]}, b^{[l]}$) — implementation wise

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

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

vectorize:

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

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

backward for layer l:

Input $da^{[l]}$

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

$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$ (* = element-wise)

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

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

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

vectorized:

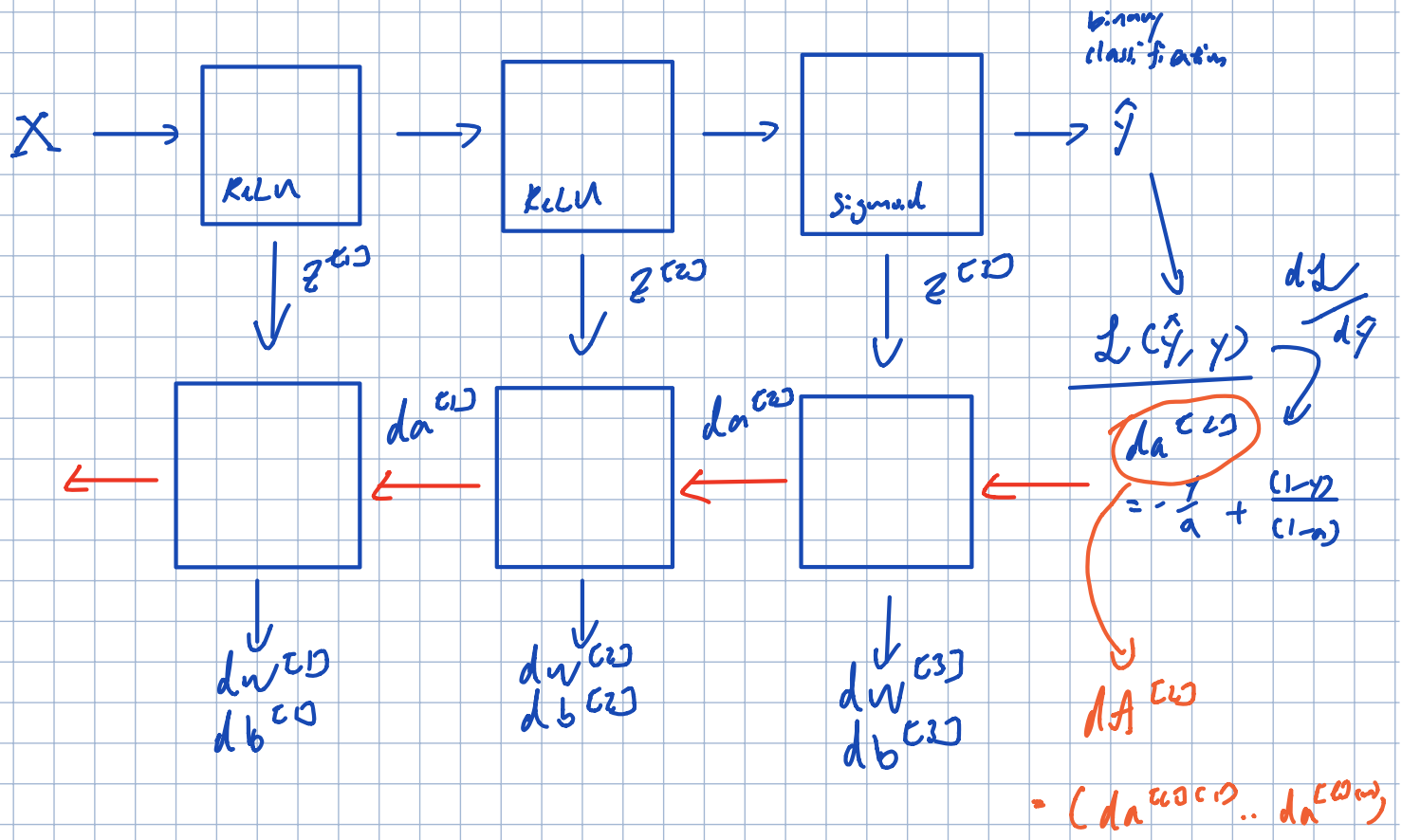
$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$

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

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

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

Summary



Parameters vs Hyperparameters

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

Hyperparameters: learning rate α

iterations for gradient descent

hidden layer L

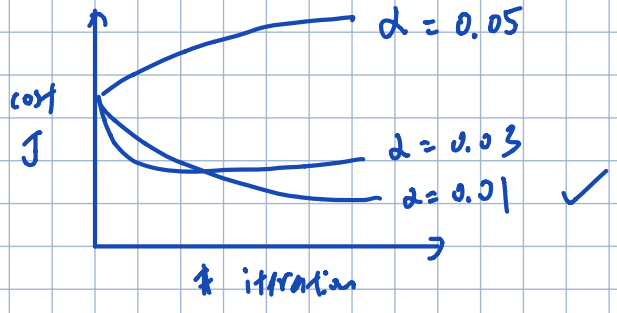
hidden units: $n^{[1]}, n^{[2]}, \ldots$

choice of activation function

These hyperparameters can ultimately decide parameters, and they cannot be learned explicitly.

More advanced hyperparameters.

momentum, mini batch size, regularisation parameters...



Summary

Forward and backward

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

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

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

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

$$dZ^{[L-1]} = W^{[L]^T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

Note that * denotes element-wise multiplication)

$$\vdots$$

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

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]^T}$$

Note that $A^{[0]^T}$ is another way to denote the input features, which is also written as X^T

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