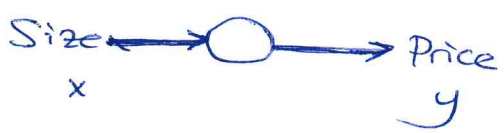
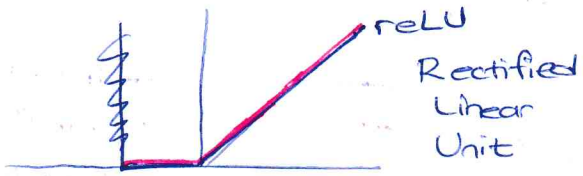
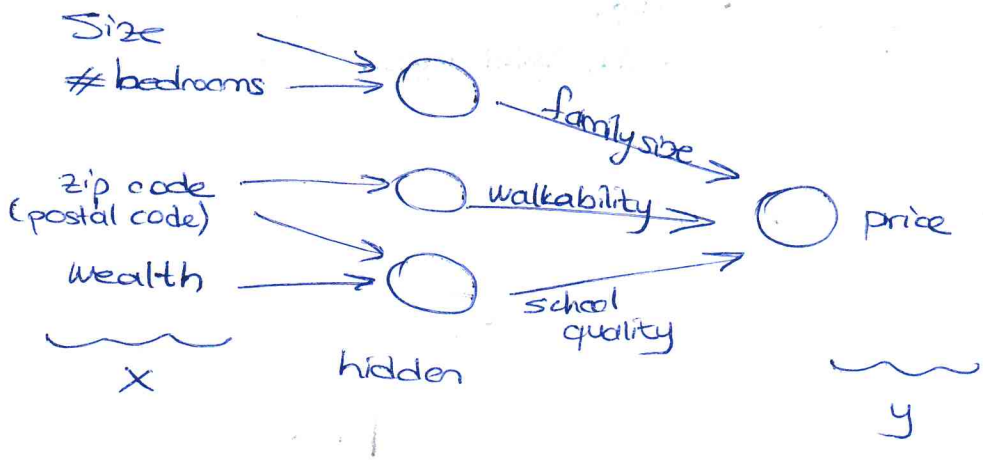
Neural Networks and Deep Learning

WEEK 1





Ex:



Supervised Learning:

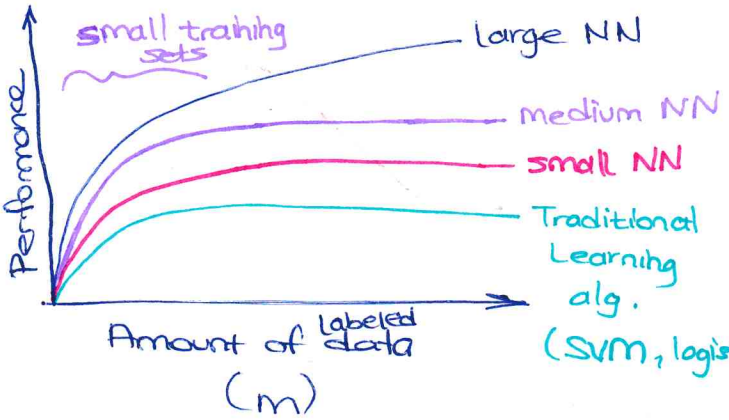
Input (x)	Output (y)	Application	
Home features	Price	Real Estate	} standart NN
Ad, user info	Click on ad? (0/1)	Online Advertising	
Image	Object (1,...,1000)	Photo tagging	} CNN
Audio	Text transcript	Speech recognition	} RNN
English	Chinese	Machine Translation	
Image, Radar Info	Position of other cars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

CNN are often used for image data

RNNs ~~are~~ ~~use~~ very good for this type of one-dimensional sequence data that has maybe a temporal component.

Structured Data	Unstructured Data
databases of data	audio, raw audio, image, text

Scale drives deep learning Progress



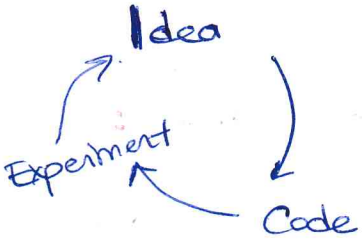
2 options to get better results

1. big neural network
2. large data

Scale drives deep learning progress

- Data
- Computation
- Algorithms

gradient is very slow so learning is very slow

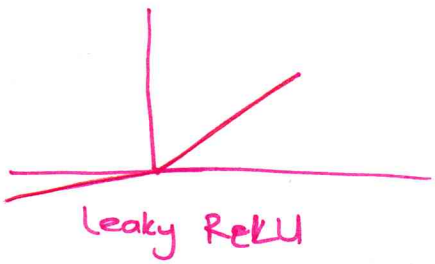


Outline of this course

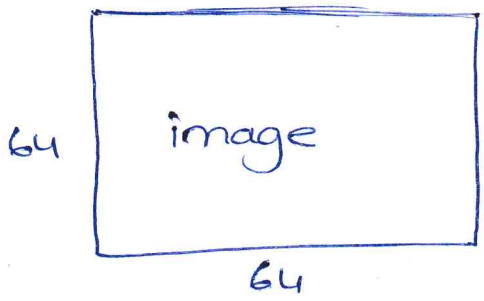
Week 1.
" 2.
" 3.
" 4.

Courses Resources

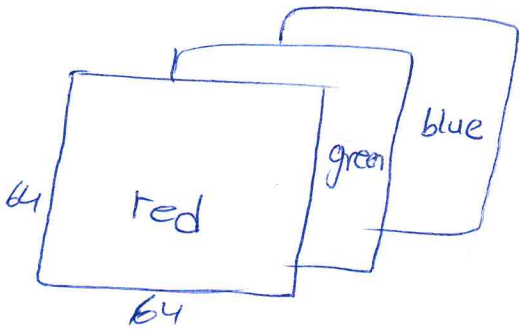
Discussion forum



Binary Classification



→ 1 (cat) vs 0 (non cat)



$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ \vdots \end{bmatrix}$ } red pixels; } green pixels; } blue pixels

$64 \times 64 \times 3 = 12288$

$n = n_x = 12288$ → representation of input dimension

$x \rightarrow y$

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

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

$m = m_{train}$ $\quad m_{test}$ = # test examples

$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(m)} \\ | & | & | & & | \end{bmatrix}$ n_x ; m

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

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

$X \in \mathbb{R}^{n_x \times m}$ $\quad X.shape = (n_x, m)$

$Y.shape = (1, m)$

Logistic Regression

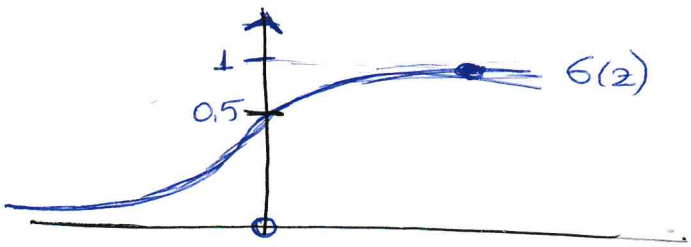
Given x $\quad \hat{y} = P(y=1|x)$, $0 \le \hat{y} \le 1$

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

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

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

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



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

If z large negative number $\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{big number}} \approx 0$

$m = m_{train}$

Logistic Regression Cost Function :

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

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

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

$x^{(i)}$ $y^{(i)}$ $z^{(i)}$: i^{th} example

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

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

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

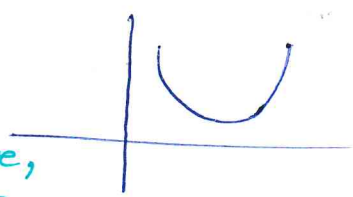
want $\log(1-\hat{y})$ large, want $\hat{y}$ small

want $\log \hat{y}$ large, want $\hat{y}$ large

→ many local opt.

gradient descent may not find the global optimum.

We need global optimum



→ we need this

★Finally, the loss function was defined to a single training example

Cost function which measures how well you're doing an entire training set.

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

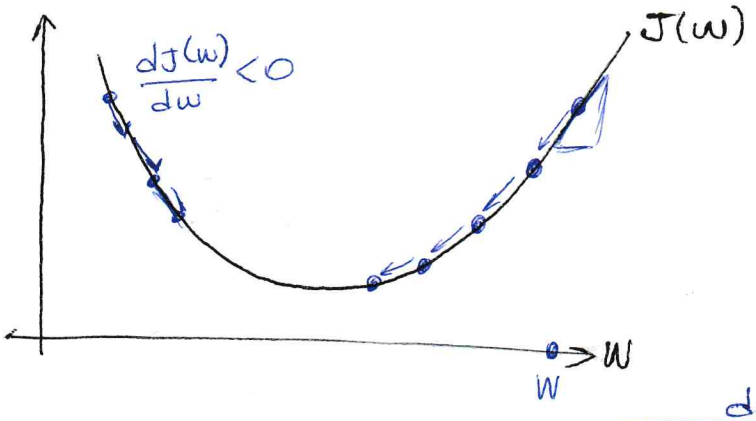
the loss function for training example and overall cost function for the parameters of your algorithm.

LR can be viewed as a very very small neural network.

Gradient Descent

~~Recap:~~

Want to find w, b that minimize $J(w,b)$



Repeat {

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

}

learning rate

$w := w - \alpha dw$

$J(w,b)$

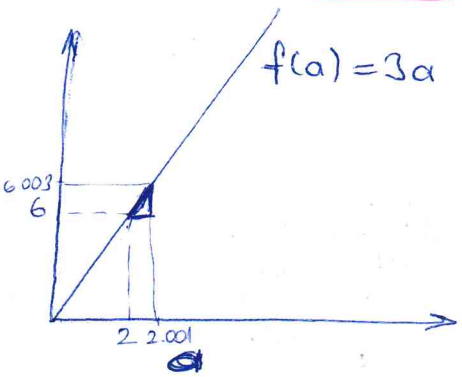
$w := w - \alpha \frac{\partial J(w,b)}{\partial (w)}$ dw

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

∂ "partial derivative"

$d \leftarrow J$ (one variable)

Derivatives

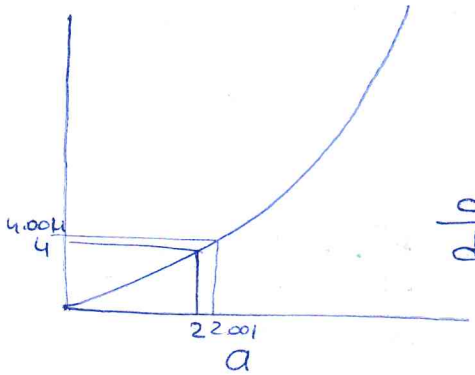


slope (derivative) of flow at $a=2$, $6=3$

$\frac{\text{height}}{\text{width}}$

slope at $a=5$ is also 3

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



$f(a) = a^2$

$\frac{d}{da} a^2 = 2a$

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

$a=2.001$ $f(a) \approx 4.004$ (4.004001)

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

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

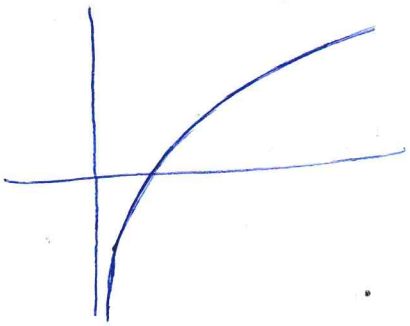
$a=5$ $f(a)=25$

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

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

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

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



$a = 2 \quad f(a) \approx 0.69315$

$a = 2.001 \quad f(a) \approx 0.69365$

$0.01 \qquad 0.0005$

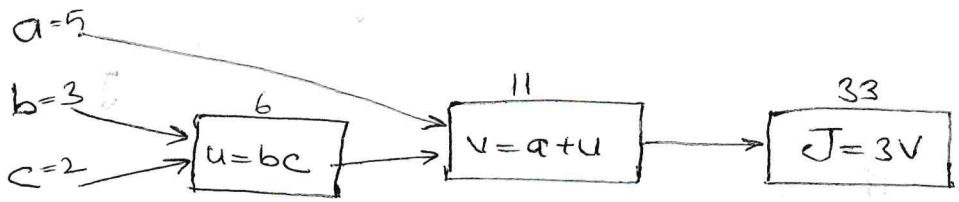
Computation Graph

$J(a,b,c) = 3(a+bc) = 3 \cdot (5 + 3 \times 2) = 33$

u : bc

v : $a+u$

J : $3v$



The computation graph organizes a computation with left-to-right computation.

$a \rightarrow v \rightarrow J$

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

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

$\frac{dv}{da} = 1$

$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6$ $\quad \frac{dJ}{du} = 3, \ \frac{du}{db} = 2$

$\left(\frac{dJ}{dv}\right) \ \left(\frac{dJ}{du}\right) \ \left(\frac{dJ}{da}\right) \ \left(\frac{dJ}{db}\right)$

$dv = 3 \quad du = 3 \quad da = 3 \quad db = 6$

$b = 3 \rightarrow 3.001$

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

$J = 33.006$

$\frac{dFinalOutputVar}{dvar}$

dJdvar

"dvar" in python

$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 9$ $\quad 3 \quad 3$

dvar represents the derivative of a final output variable with respect to various intermediate quantities.

Logistic Regression – Gradient Descent

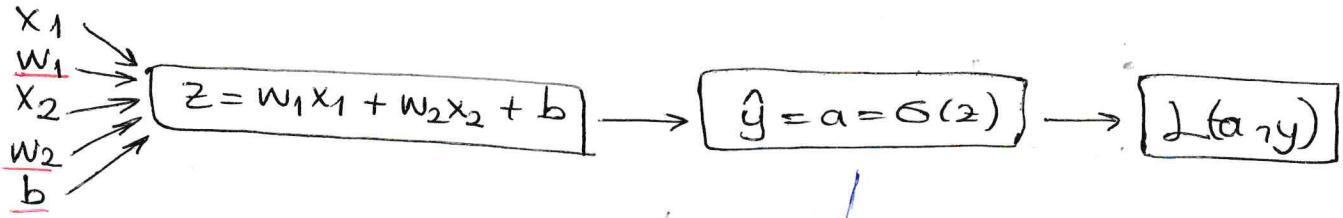
$z = w^T x + b$

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

$a \rightarrow$ is output of logistic regression

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

$y \rightarrow$ ground truth label



modify w_1, w_2, b in order to reduce Loss.

$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$

$= a - y$

$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

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

"da" $= \frac{d\mathcal{L}(a,y)}{da}$

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

$\frac{\partial \mathcal{L}}{\partial w_1} =$ "dw_1" $= x_1 \cdot dz$ (in python)

$dw_2 = x_2 \cdot dz \quad db = dz$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

Gradient Descent on m examples

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

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

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

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

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

gradient

$dw_1^{(i)}$ – $(x^{(i)}, y^{(i)})$

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

For $i=1$ to m

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

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

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

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

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

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

$n=2$

may need another dw_3 for loop dw_n

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

$J /= m$

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

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

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

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

Q: Why is there only one dw variable? (i.e. no i superscripts in the for loop)

A: The value of dw in the code is cumulative.

Vectorization

$z = w^T x + b$

Non-vectorized

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

- it is going to be slow

Vectorized

$w \leftarrow \mathbb{R}^{n_x}$

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

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

$w^T x$

CPU, GPU } have parallelization instructions called SIMD instruction. (single instruction multiple data)

More Vectorization Examples

Whenever possible, avoid explicit for-loops.

$u = Av$

$u_i = \sum_j A_{ij} v_j$

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

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

NOTE: `a*b` → is an element-wise computation

`np.dot` → matrix multiplication

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

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

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

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

np.log(v)
np.abs(v)
np.maximum(v,0)
v**2
```

instead of $dw_1 = 0 \quad dw_2 = 0$ we prefer `dw = np.zeros((n-x, 1))`

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

$dw /= m$

Vectorizing Logistic Regression

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

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

$Z = np.dot(w.T, X) + b$ → Broadcasting (b: $(1,1)$)

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

Vectorizing Logistic Regression's Gradient Computation

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

$dz = [dz^{(1)} \; dz^{(2)} \; \dots \; dz^{(m)}]$

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

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

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

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

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

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

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

$n \times 1$

$Z = w^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

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

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

→ gradient descent for logistic regression

→ for one iteration

Broadcasting in Python

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

→ vertically (0), horizontally (axis=1)

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

(3,4) ↑ ↳ it can be used to be sure.

General Principle :

(m,n) matrix	+ − * /	(1,n)	→ (m,n)
		(m,1)	→ (m,n)

(m,1)

$\begin{bmatrix}1\\2\\3\end{bmatrix}$ + R

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

[1 2 3] + 100 = [101 102 103]

Matlab or Octave (bsxfun function)

A note on python/numpy vectors

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

a.shape=(5,)

"rank 1 array"

} don't use

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

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

↓ checks the shape. if is not (correct) consistent, it stops.

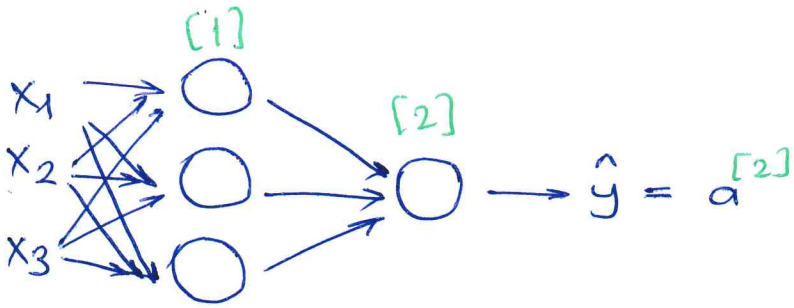
→ assertion statement is really inexpensive

! it also help to serve as documentation for your code.

! don't be shy about calling the reshape operation

Neural Networks Overview

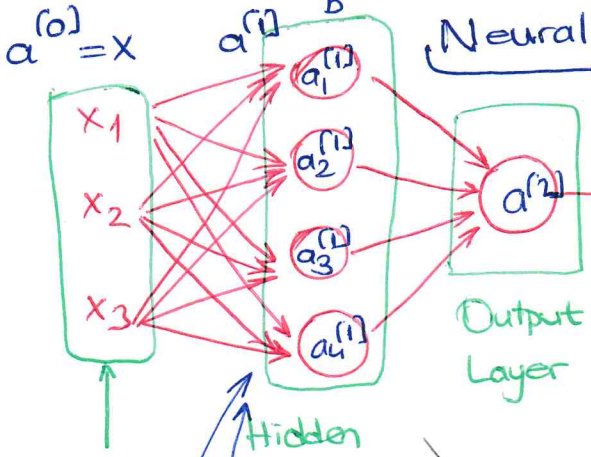
Elif Cansu YILDIZ



$w^{[1]}$, $b^{[1]}$ → $z^{[1]} = W^{[1]}x + b^{[1]}$ → $a^{[1]} = \sigma(z^{[1]})$ → ...

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

Neural Network Representation



Input Layer

Hidden Layer

Output Layer

$\hat{y} = a^{[2]}$ → 2 Layer NN

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

Hidden layers are not seen in the training set.

Training set only includes the input layer and output layer.

In conventional usage, hidden layers and output layers are counted.

for this example

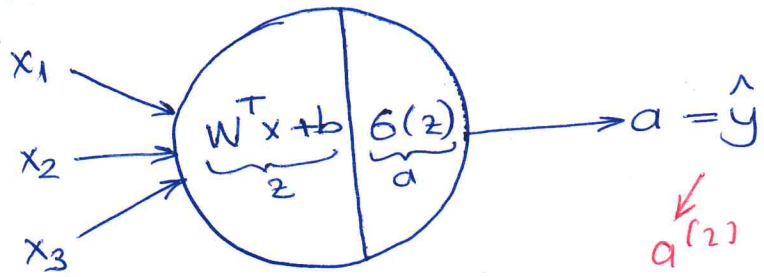
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix}$$

$W^{[1]}$ (4,3) ↓ hidden layer dimension, ↓ inputs

$b^{[1]}$ (4,1) ↓ hidden layer dimension

Computing a Neural Network's Output

Neural Network Representation:



$x = a^{[0]}$

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

$$a = \sigma(z)$$

$a_i^{[l]}$ → layer

↳ node in layer

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

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

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

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

Given input x:

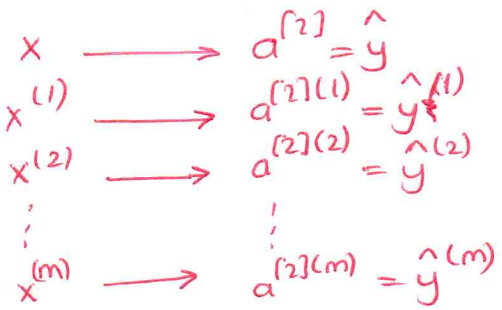
$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$

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

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

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

Vectorizing Across Multiple Examples



$a^{[2](i)}$ → training example, layer 2

for $i = 1$ to m:

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

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

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

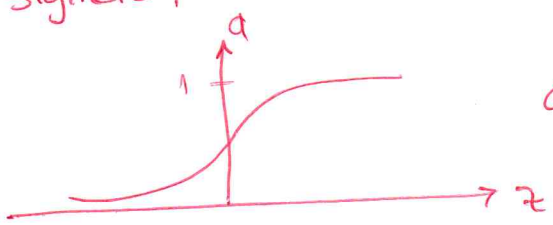
(n_x, m)

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

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

Activation Functions

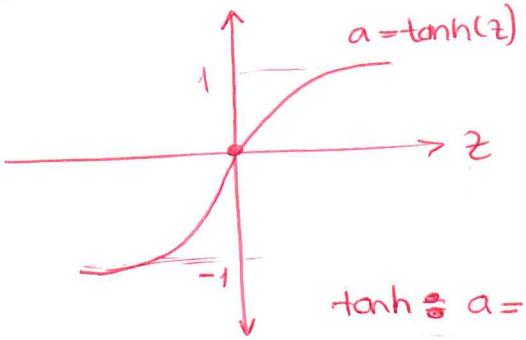
sigmoid:



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

never use this, except for the output layer, if you are doing binary classification. or maybe almost never use this.

tanh function:

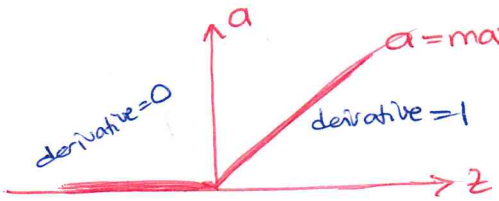


$$\tanh: a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

Superior than sigmoid.

When z is very large or very small, the slope of the function would be very close to 0. This can slow down gradient descent

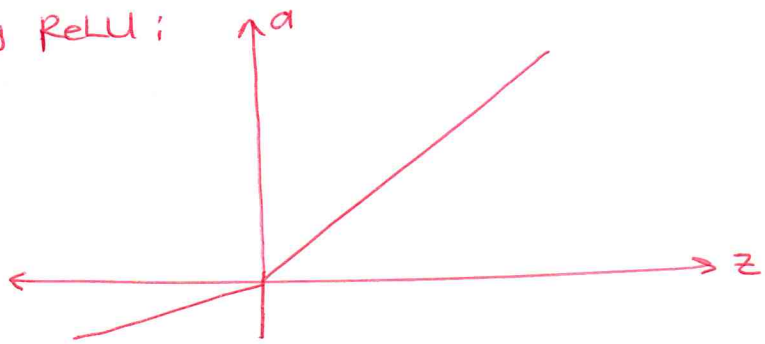
ReLU function: (Rectified Linear Unit)



ReLU $\Rightarrow a = \max(0, z)$

more popular

Leaky ReLU:



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

Why do we need non-linear activation functions?

linear activation function = identity activation function

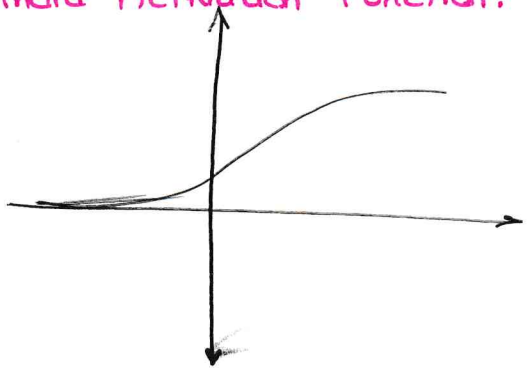
There is just one place where you might use a linear activation function: if you are doing machine learning on the regression problem.

So the one place you might use a linear activation function is usually in the output layer.

Composition of two linear functions is southern linear functions.

Derivatives of Activation Functions

Sigmoid Activation Function:



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

$g'(z)$

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

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

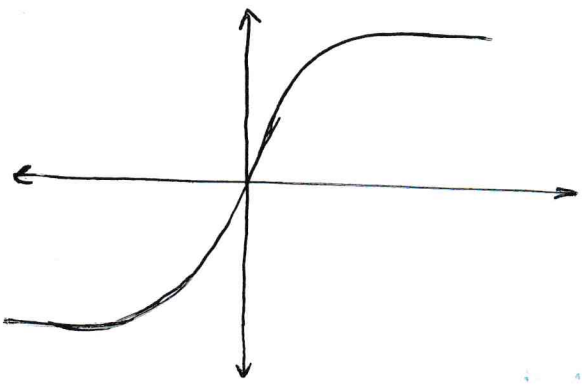
$$= g(z)\,(1-g(z))$$

$z = 10 \quad g(z) \approx 1 \quad \frac{d}{dz} g(z) \approx 1 \cdot (1-1) \approx 0$

$z = -10 \quad g(z) \approx 0 \quad \frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$

$z = 0 \quad g(z) = \frac{1}{2} \quad \frac{d}{dz} g(z) = \frac{1}{2}\left(1-\frac{1}{2}\right) = \frac{1}{4}$

Tanh Activation Function:



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

$$= \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = a \qquad g'(z) = 1 - a^2$$

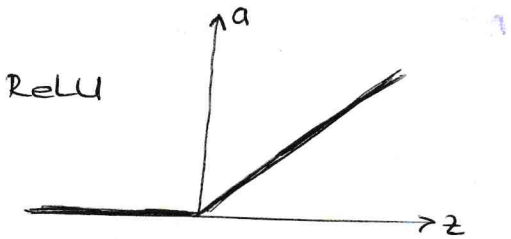
$$g'(z) = \frac{d}{dz} g(z) = 1 - (\tanh(z))^2$$

$z = 10 \quad \tanh(z) \approx 1 \quad g'(z) = 0$

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

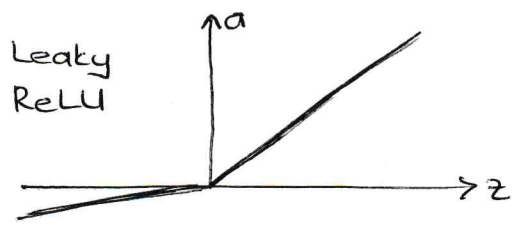
$z = 0 \quad \tanh(z) = 0 \quad g'(z) = 1$

ReLU and Leaky ReLU:



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

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



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

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

Gradient Descent For Neural Networks

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

$n_x = n^{[0]}$ (input feature), $n^{[1]}$ (hidden units), $n^{[2]} = 1$ (output units)

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

When training a neural network, it is important to initialize the parameters randomly rather than to all zeros.

Gradient Descent:

Repeat {

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

$dW^{[1]}$, $db^{[1]}$, $dW^{[2]}$, $db^{[2]}$

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

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

⋮

}

Forward Propagation:

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

if this is binary classification

Backpropagation:

$$dz^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} \cdot dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \, np.sum(dz^{[2]}, axis=1, keepdims=True)$$

it prevents Python from outputting one of those funny rank 1 arrays

$$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(z^{[1]})}_{(n^{[1]}, m)}$$

* : element-wise product

$$dW^{[1]} = \frac{1}{m} dz^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \cdot np.sum(dz^{[1]}, axis=1, keepDims=True)$$

$(n^{[2]}, 1)$ → $(n^{[1]}, 1)$

Backpropagation Intuition

$$dz^{[2]} = a^{[2]} - y$$
$$dw^{[2]} = dz^{[2]} a^{[1]T}$$
$$db^{[2]} = dz^{[2]}$$
$$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$
$$dw^{[1]} = dz^{[1]} x^T$$
$$db^{[1]} = dz^{[1]}$$

Vectorized Implementation

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

was mention in previous page.

dimensions

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

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

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

element wise

$$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$
$$(n^{[1]}, 1) = (n^{[1]}, n^{[2]}) \quad (n^{[2]}, 1) * (n^{[1]}, 1)$$

$$dz^{[2]} = a^{[2]} - y$$
$$dw^{[2]} = dz^{[2]} \cdot a^{[1]T}$$
$$db^{[2]} = dz^{[2]}$$

why?

"$dw = dz \cdot x$"

$w_2^{[1]} = [\ \_\_\_\ ]$ for one output

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

$$dz^{[2]} = A^{[2]} - Y$$
$$dw^{[2]} = \frac{1}{m} dz^{[2]} \cdot A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True)$$
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(Z^{[1]})$$

elementwise

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

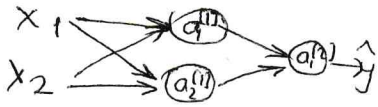
$$dW^{[1]} = \frac{1}{m} dz^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

Random Initialization

if W is initialized with 0, all units in hidden layer will produce the same value and will be symmetric.

As a solution: you initialize these parameters (W) with very small random values and you prevent the symmetry problem (breaking).

$$W^{[1]} = np.random.randn((2,2)) * 0.01$$
$$b^{[1]} = np.zero((2,1))$$
$$W^{[2]} = np.random.rand((1,2)) * 0.01$$
$$b^{[2]} = 0$$



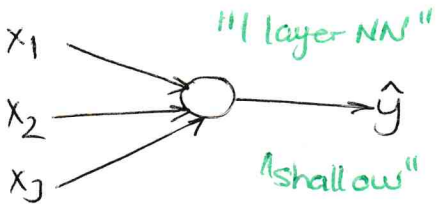
if we give very big numbers to W (initially), the derivatives would be very small (close to 0) and learning would be very slow.

Notes:

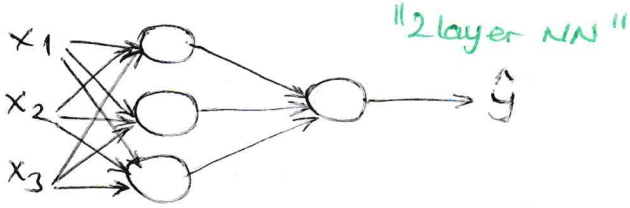
1. The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, so it centers the data better for the next layer.

The output of the tanh is between -1 and 1, it thus centers the data which makes the learning simpler for the next layer.

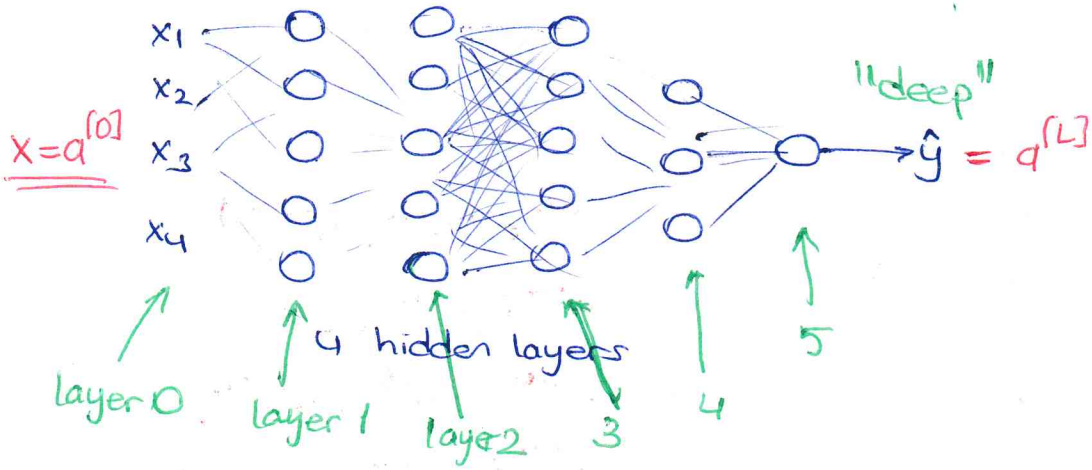
Deep L-layer Neural Network



logistic Regression



1 hidden layer



$L = 4$ (# layers)

$n^{[l]}$ = # units in layer l

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

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

$b^{[l]}$

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

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

$n^{[L]} = 1$

Forward Propagation in a Deep Network

$z^{[1]} = w^{[1]} A^{[0]} + b^{[1]}$ $\quad X = A^{[0]}$

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

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

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

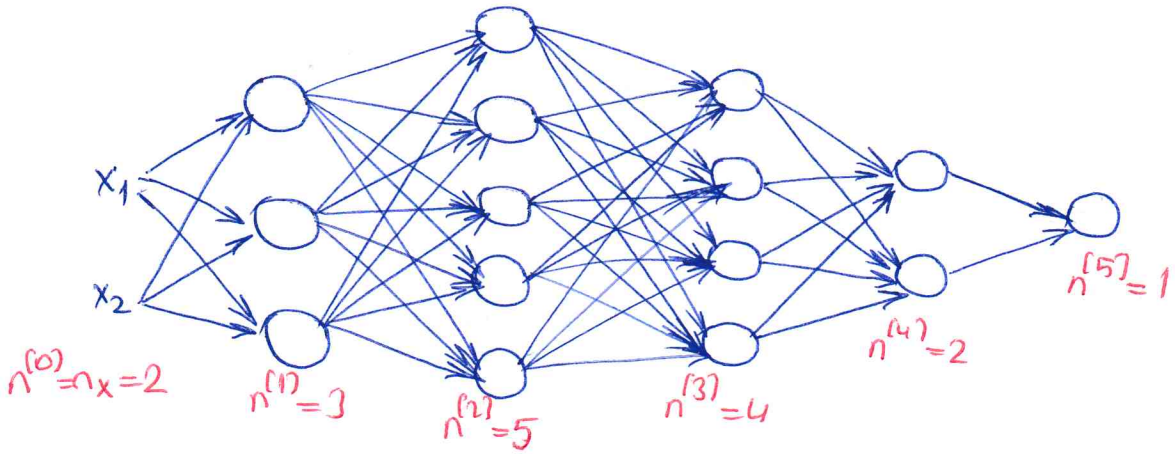
$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

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

for loop $\quad$ for $l = 1 \ldots L$

There is no way to implement this without an explicit For loop. In this place, it's perfectly okay to have an explicit For loop.

Getting Your Matrix Dimensions Right



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

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

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

- $w^{[1]} : (n^{[1]}, n^{[0]})$
- $w^{[2]} : (5,3)\ (n^{[2]}, n^{[1]})$
- $z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$
- $(5,1)\quad (5,3)\quad (3,1)\quad (5,1)\ (n^{[2]},1)$

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

$w^{[4]} : (3,4)\ ,\ w^{[5]} : (1,2)$

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

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

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

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

Vectorized Implementation

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

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

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

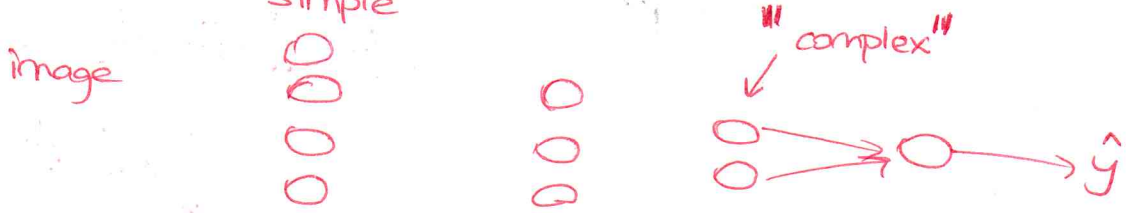
$$Z^{[1]} = W^{[1]} \cdot \underbrace{X}_{A^{[1-1]}} + b^{[1]}$$

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

With broadcasting it acts like $(n^{[1]},m)$

Why deep representation?

Intuition about deep representation

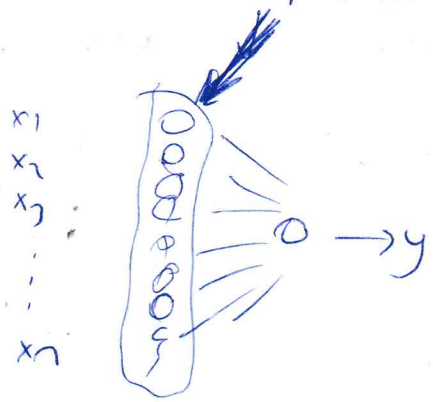


Audio → low level audio → phonene CAT → words → sentence phrases

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.

it necessicate exponentially large units



deep learning is a branding. It is actually neural network with many hidden layers.

Forward and Backward Functions

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

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

cache $z^{[l]}$

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

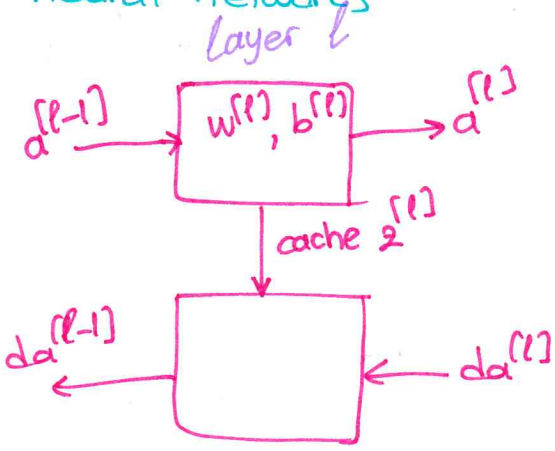
$a^{[l]} = g^{[l]}(z^{[l]})$

Backward: Input $da^{[l]}$

cache $z^{[l]}$

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

Building blocks of deep neural networks



Forward Propagation for layer l

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

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

Backward Propagation for layer l

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

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

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

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

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

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

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

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

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

Parameters vs Hyperparameters

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \ldots$

Hyperparameters:
- Learning rate
- # iterations
- # hidden layers L
- # hidden units $n^{[1]}, n^{[2]}, \ldots$
- choice of activation function

} determine the final parameters (W, b)

Later: momentum, mini-batch size, regularization parameters

Applied deep learning is a very empirical process.