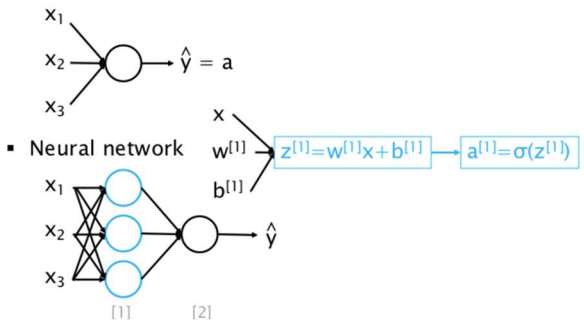
06 - TRAINING NEURAL NETWORK

Logistic regression vs NN

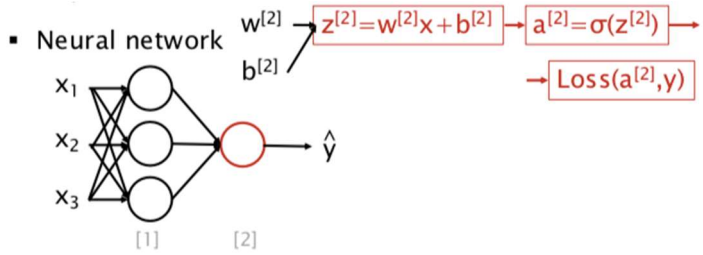
Facciamo un passo di forward propagation ottenendo delle attivazioni a

- Logistic regression



→ pesi del livello 1 + bias → funzione attivazione

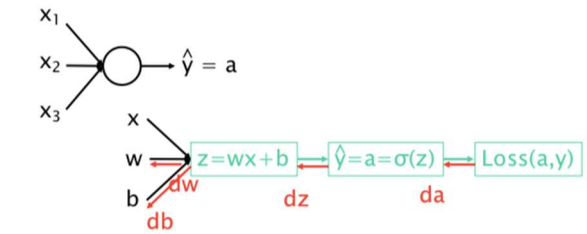
Che vengono passate al livello successivo ottenendo l'attivazione finale e quindi la Loss



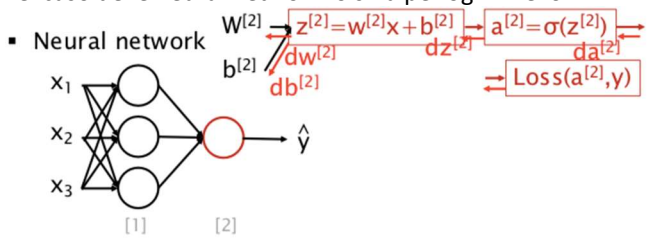
→ pesi del livello 2, vengono usati per moltiplicare le attivazioni che escono dal lvuello precedente + i bias del livello 2.
Calcolo della loss utilizzando l'uscita di ultimo livello e la y iniziale.

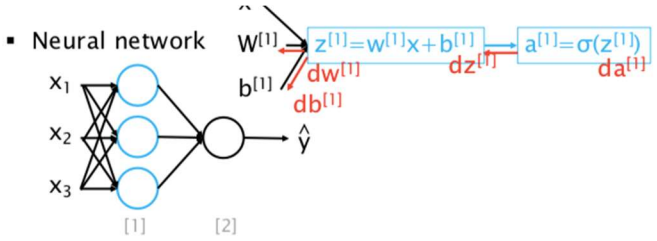
Avendo il computational graph, posso effettuare la back propagation calcolando le derivate parziali

- Logistic regression



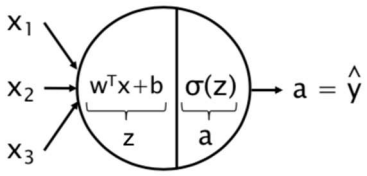
Nel caso delle neural network lo si fa per ogni livello





Neural network representation

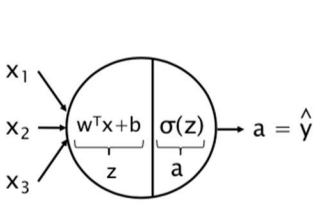
La singola unità è fatta così: è composta da due fasi.



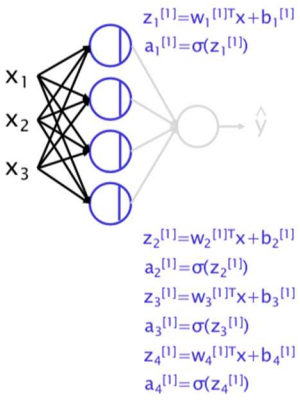
Sx: parte lineare
Dx: parte non lineare

Logistic regression unit

Una fase è la combinazione lineare e su questa l'altra fase calcola l'attivazione.
Per ogni unità si avrà $z_i^{[l]}$ e $a_i^{[l]}$



Logistic regression unit

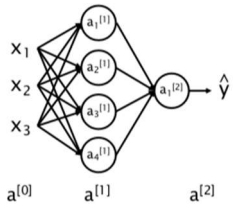


Ognuna delle equazioni appena scritte possono essere scritte utilizzando **la matrice dei pesi $W^{[1]}$** ottenendo un'unica equazione per livello

$$W^{[1]} = \begin{bmatrix} \text{---} & w_1^{[1]T} & \text{---} \\ \text{---} & w_2^{[1]T} & \text{---} \\ \text{---} & w_3^{[1]T} & \text{---} \\ \text{---} & w_4^{[1]T} & \text{---} \end{bmatrix}$$

Oltre alla matrice dei pesi, c'è anche il vettore di bias; Combinandoli, ottengo la funzione di attivazione (sigmoide).

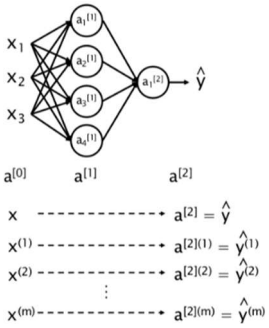
$$z^{[1]} = w^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$



$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$

Sostituiamo gli input x come attivazioni del livello 0, e passiamo al livello successivo ottenendo infine:

Per calcolarlo per tutti i training examples dovrei usare un ciclo for mettendo quindi **gli indici (i)**

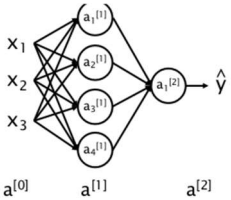


$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$

For one training example

for i=1 to m
$z^{[1](i)} = w^{[1]}a^{[0](i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = w^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$

Potrei sfruttare però la vettorizzazione:



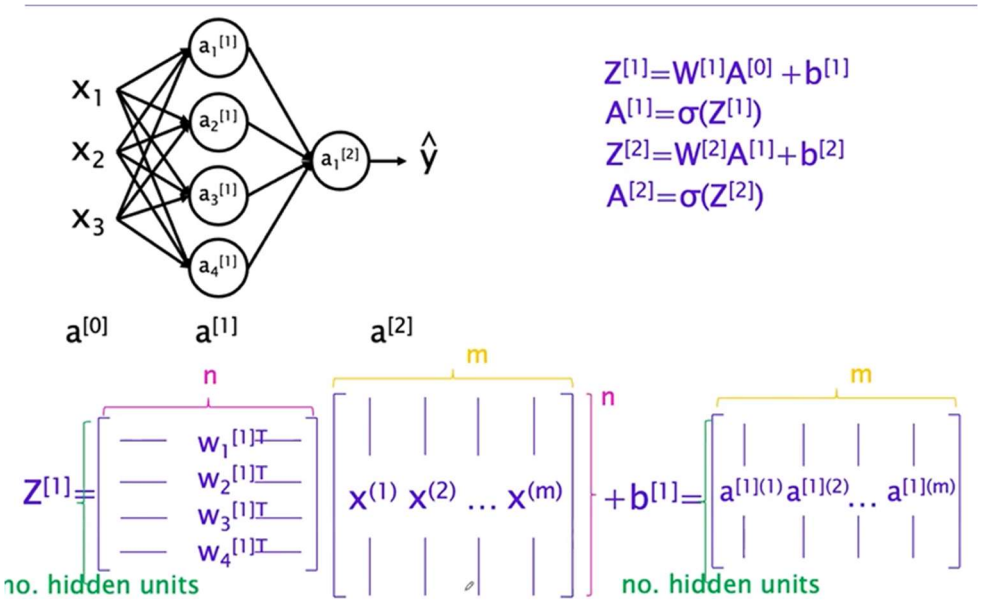
for i=1 to m
$z^{[1](i)} = w^{[1]}a^{[0](i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = w^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$

$$X = A^{[0]} = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & | & | \end{bmatrix}$$

$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

Ogni livello può essere espresso come calcolo matriciale avendo la matrice dei pesi W, delle attivazioni del livello precedente A e il bias di quel livello.

Avremo la matrice W che ha come righe il numero di hidden units ognuna lunga n features, la matrice di input X (o meglio A) che ha m righe per i training examples ognuno con n features. Il risultato è la matrice Z che avra m vettori lunghi quanto il numero di hidden units.

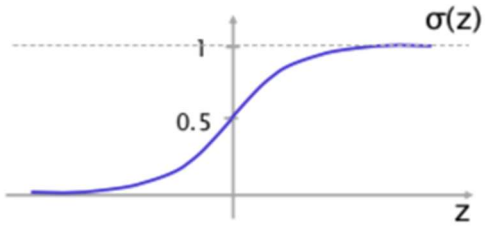


Funzioni di attivazione

Sigmoide

La sigmoide va bene se nell'ultimo livello mi serve fare una classificazione binaria.

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

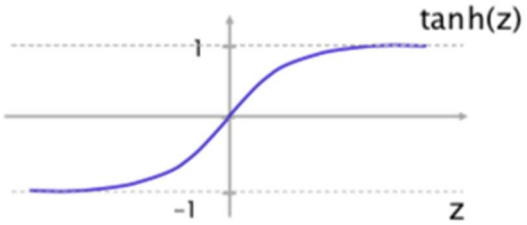


+ comprime output tra [0;1]
+ passa da 0.5 alla meta
- noi calcoliamo delle derivate del valore di uscita, e la derivata per valori di z molto alti, tende ad essere molto piccola (dunque convergenza molto lenta).

Tangene iberbolica

shiftata della sigmoide. Funziona meglio perché porta ad avere l'attivazione ad una media 0 → più bilanciamento.

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



- Continua ad avere il problema degli asintoti

Per valori molto piccoli e molto grandi di z si hanno comportamenti asintotici che rallentano l'algoritmo in quanto il gradient descent andrà a fare dei passi molto piccoli.

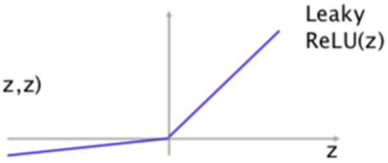
Si usa per questo molto comunemente negli hidden layer la ReLU e la sua nuova versione LeakyReLU

ReLU

$$\text{ReLU}(z) = \max(0,z)$$

→ più facile da calcolare ma per valori di Z negativi, ha derivata nulla, dunque è stata inventata **la Leaky ReLU**

$$\text{Leaky ReLU}(z) = \max(0.01z,z)$$



La Leaky funziona meglio perchè la derivate della parte a sinistra non è più 0 ma un valore costante.

Gradient descent for NN

Assunzioni semplificative: nel livello intermedio abbiamo la funzione di **attivazione generica g** e nell'ultimo una **sigmoide σ**.
Calcoliamo la backward propagation:

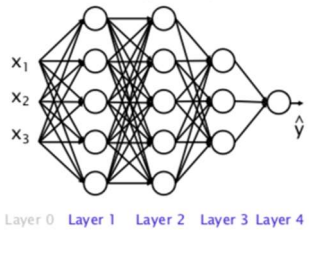
Forward propagation	Backward propagation
$Z^{[1]}=W^{[1]}A^{[0]}+b^{[1]}$	$dZ^{[2]}=A^{[2]}-Y$
$A^{[1]}=g^{[1]}(Z^{[1]})$	$dW^{[2]}=1/m\ (dZ^{[2]}\ A^{[1]T})$
$Z^{[2]}=W^{[2]}A^{[1]}+b^{[2]}$	$db^{[2]}=1/m\ sum(dZ^{[2]})$
$A^{[2]}=g^{[2]}(Z^{[2]})=\sigma(Z^{[2]})$	$dZ^{[1]}=W^{[2]T}\ dZ^{[2]}\ .*\ g^{[1]'}\ Z^{[1]}$
	$dW^{[1]}=1/m\ (dZ^{[1]}\ A^{[0]T})$
	$db^{[1]}=1/m\ sum(dZ^{[1]})$

Nel caso più generale io devo saper calcolare la derivata della funzione di attivazione.

Gradient descent for deeper NN

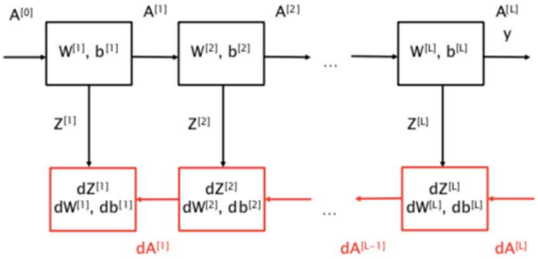
Si può iniziare a parlare di reti non più "shallow" perché si ha sia un livello avanti che uno dietro al centro e posso scrivere le cose più genericamente.

- Notation (e.g., 4 layer NN)



L=no. of layers (4)
$n^{[l]}$=no. units in layer l
$a^{[l]}=activ.\ in\ layer\ l$
$a^{[l]}=g^{[l]}(z^{[l]})$
$w^{[l]}, b^{[l]}$=weights for $z^{[l]}$

Traduco la rete in blocchetti per ogni livello →



Mi serve memorizzare i risultati di ogni livello perché serviranno per la back propagation.

Riscrivendo le propagations in forma generica otteniamo

Forward propagation	Backward propagation
$Z^{[l]}=W^{[l]}A^{[l-1]}+b^{[l]}$	$dZ^{[l]}=dA^{[l]}\ .*\ g^{[l]'}\ Z^{[l]}$
$A^{[l]}=g^{[l]}(Z^{[l]})$	$dW^{[l]}=1/m\ (dZ^{[l]}\ A^{[l-1]T})$
	$db^{[1]}=1/m\ sum(dZ^{[l]})$
	$dA^{[l-1]}=W^{[l]T}\ dZ^{[l]}$

Inizializzazione

Ci sono problemi con **l'inizializzazione a zero** con le reti neurali anche se funziona normalmente con la logistic regression.
Non funziona con le reti neurali perché il primo livello calcolerebbe le stesse funzioni di attivazione dato che tutti gli input sono uguali e sono 0. Si dimostra inoltre che anche le derivate parziali sarebbero uguali anche dopo la prima iterazione e quindi si calcolerebbe sempre la stessa funzione.

Questo problema è noto come il **symmetric weights problem**: dopo ogni update, i parametri corrisponenti all'input del livello successivo sono identici. → la rete non impara nulla, tutte le computation unit andrebbero a calcolare la stessa funzione.

La soluzione sta nell'inizializzazione di ogni input con un valore molto piccolo [-e, e]. (funziona anche se con reti più sofisticate ci sono inizializzazioni piùcomplesse che funzionano ancora meglio)

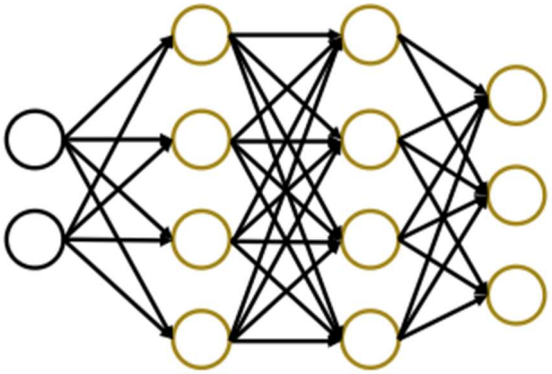
Per addestrare:

- Scegliere architettura di rete
- Scegliere il numero di input unit → legata al numero di cfeature
- Scehleire il numero di output unit → lòegata al numero di classi che voglio predire
- Scegli il numero di hidden layer → più middle layer metti → più unit avrai → più paramertri avrai → decison boundary potra variare tanto in quanto i parametri danno flessibilità → più vanno fatti calcoli ad ogni passo delg radient descend
- Inizializzo i pesi
- Forward propagation → calcolo funzioni di costo
- Backword propagation → derivate parziali
- Con tutte le informazioni calcolate, faccio un passo di gradient descent con l'obbiettivo di minimizzare la funzione di costo

Training a neural network

Prendiamo come esempio la seguente rete "fully connected"

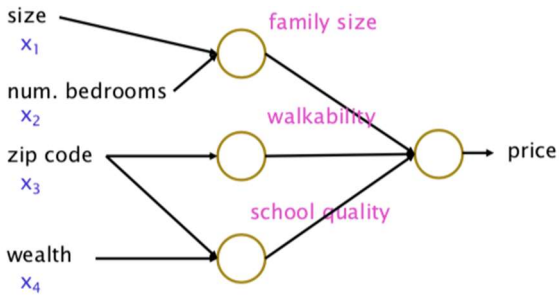
- Nell'architettura il numero di input è dato dal numero di features $x^{(i)}$, il numero di output dal numero di classi e gli hidden layers uno o più dove più ne metto e meglio è solo che è proporzionale al costo computazionale.
- Inizializzare randomicamente i pesi
- Implemento la forward propagation per ottenere $\hat{y}^{(i)}$ per ogni $x^{(i)}$
- Implemento il codice per calcolare la funzione di costo J(w,b)
- Implemento la backward propagation per calcolare le derivate parziali
- Uso il gradient descent o altre tecniche per provare a minimizzare J(w,b)



Viene fuori che la funzione di costo avrà un po' di minimi locali ma non è un problema critico.

NNs and non-linear problems

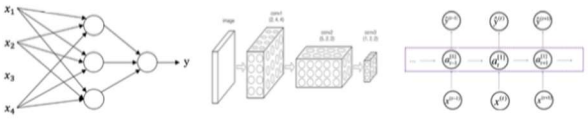
Perché le reti neurali sono buone per questi problemi?



La rete in pratica impara le proprie features a partire dalle feature iniziali (impara family size, walkability, school quality a partire dai 4 che avevo dato all'inizio).
Più la rete è grande e più può prendere features complesse.

Applicazioni delle reti neurali

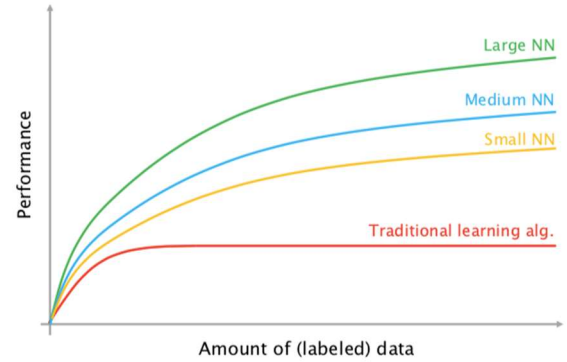
Input(x)	Output (y)	Application	
Home features	Price	Real Estate	"standard" NNs
Ad, user info	Click on ad? (0/1)	Online advertising	
Image	Object (1,...,1000)	Photo tagging	CNNs
Audio	Text transcript	Speech recognition	RNNs
English	Italian	Machine translation	
Image, radar info	Position of other cars	Autonomous driving	Hybrid



Inoltre le reti neurali si comportano bene con i dati non strutturati

Perchè le reti neuroli sono esplose ora con il deep learning.

Scale driving machine/deep learning



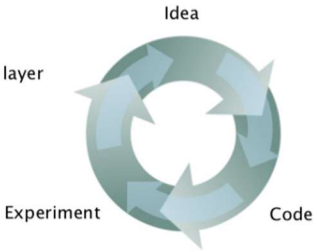
Si vede che a prescindere dal numero dei dati ad un certo punto si raggiunge una stabilità nelle performance quindi quello che permette di avere performance migliori è un'architettura diversa.

- Data
- Computation
- Algorithms

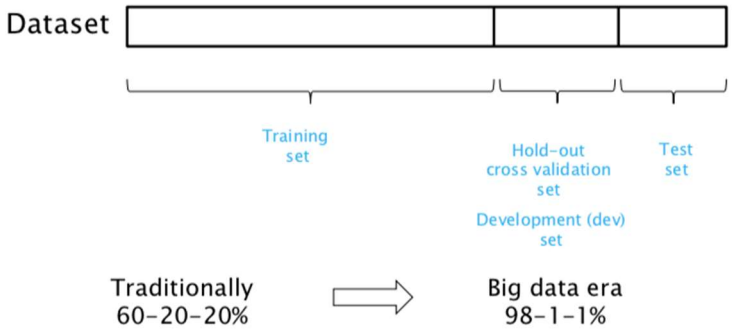
Iterative process

Con Algoritmo si intendono gli iperparametri:

- Parameters: W[1], b[1], W[2], b[2], ...
- Hyperparameters
  - Learning rate α
  - No. iterations
  - No. hidden layers
  - No. hidden units per layer
  - Activation functions
  - ...



Training/Dev/Test sets



Spezzo il dataser intre parti:

- Prima parte: addestramento: addestro tante varianti e le provo su sotto-insiemi di dati mai visti diversi. Provo sul Test set mai visto e vedo come performa la rete.

È importante che i dati vengano dalla stessa distribuzione!