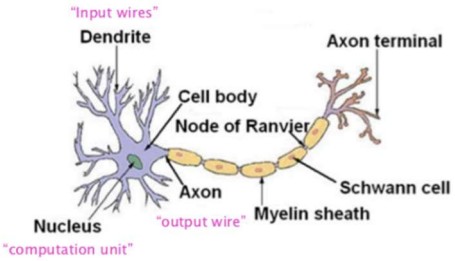
NEURAL NETWORKS

Algoritmi che cercano di simulare il comportamento del cervello.
Nel 1922 da alcuni neuroscienziati è stata sviluppata questa ipotesi del "one learning algorithm" che dice: "Prendendo dei cervelli di animali (gatti in particolare) e immaginando di avere una parte del cervello responsabile di processare i segnali uditivi, viene tagliato il collegamento tra questa parte e l'orecchio e sostituito con quello del sistema visivo. Hanno scoperto che la corteccia destinata agli stimoli uditivi impara a vedere. "
Questo dimostra che si può parlare di un unico algoritmo o analogamente di un unico insieme di cellule pensate per apprendere. La chiave sta non nella tipologia di cellule ma di come sono connesse.

Nel nostro cervello funziona in questo modo: si ha il neurone con un nucleo che svolge qualche funzione e ci sono delle diramazioni chiamate dendriti e l'assone; i **dendriti** fungono da canali di input mentre **l'assone** da output.

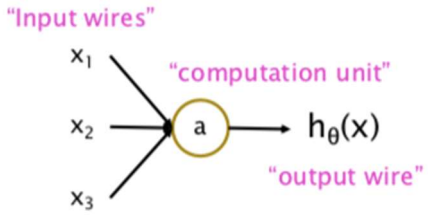


I neuroni sono collegati in reti neuronali, gli assoni sono collegati ai dendriti di altri neuroni e ai muscoli per stimolare le reazioni muscolari. Le comunicazioni avvengono attraverso gli spike elettrici.

- Esempi di esperimenti di reti neurali

Quello che è alla base delle reti neurali artificiali è il modello neurale

Neurone artificiale



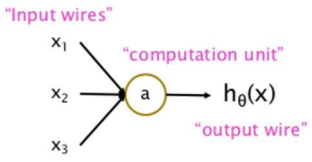
neurone → elemento circolare a (**computation unit**) con collegamenti di ingresso e collegamenti di uscita.

Il compito del **modello di computazione** è quello di calcolare una <u>funzione di attivazione</u>:

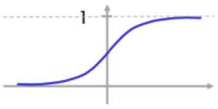
- Quali sono quegli input che attivano quel neurone.

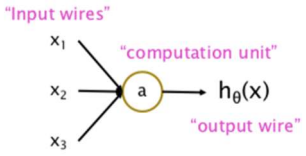
Neurone prende $x1*\theta_1 + x2*\theta_2 + x3*\theta_3$ e applica una sigmoide → $h_\theta(x) = \sigma\ (x1*\theta_1 + x2*\theta_2 + x3*\theta_3)$

Non si usa sempre una sigmoide come componente non lineare ma anche alcune più semplici come, ad esempio, una ReLU (RectifiedLinearUnit).

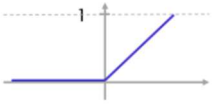


Sigmoid (logistic)
<u>activation function</u>



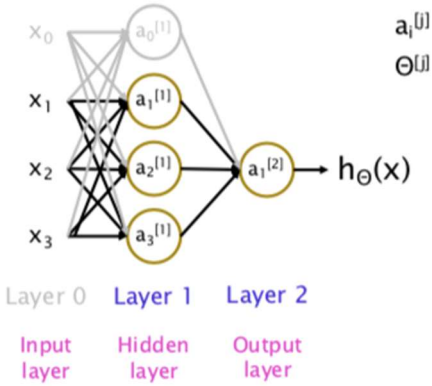


ReLU
<u>activation function</u>



Negli anni si è scoperto che le reti neurali, combinazioni di questi elementi, si comportavano più che bene. Quindi si è cominciato a voler aumentare i dati in pasto a tali reti per farle diventare sempre migliori. Il problema è che poi il calcolo computazione cresce esponenzialmente.
La **ReLU** è servita a migliorare le prestazioni scoprendo che le reti si comportavano altrettanto bene semplificando la funzione di attivazione.
Negli anni, quindi, sono cambiate due cose: **Più dati, reti migliori**

Alla fine delle reti neurali c'è un neurone che riceve come input l'output di altri neuroni.



$a_i^{[j]}$ = activation of unit i in layer j
$\Theta^{[j]}$ = matrix of weights controlling function mapping from layer j to layer j+1

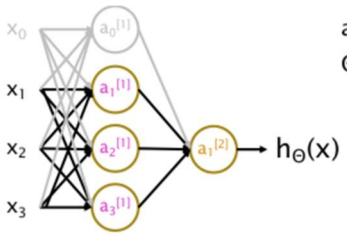
Perceptrone: multi-layer perceptron in questo caso usata per una classificazione binaria (in quanto ha solo un neurone in uscita).
Lo si capisce dall'unico neurone in uscita perché applica una sigmoide.

È una rete a **due livelli** perché non si conta il livello di input (in quanto non ci sono neuroni).

Il ruolo di ognuno dei neuroni nel **livello 1** è di calcolare una sigmoide di qualcosa (ricevono tutti gli ingressi).
Queste attivazioni vengono etichettate con $a_i^{[j]}$ dove "i" indica la riga in cui si trovano e [j] che indica il livello.
La notazione è importante le () erano usate per i training example mentre ora le [] sono usate per le attivazioni.
Le attivazioni saranno quindi delle funzioni applicate ad una combinazione lineare di parametri $\Theta^{[j]}$

L'ultimo neurone prende tutte le attivazioni, le combina con i propri parametri e **calcola l'attivazione** dell'ultimo livello che coincide col valore da predire.

Nota: Xo va a tutti i neuroni ed è messo ad 1, influisce poco aggiungendo una componente costante



$a_i^{[j]}$ = activation of unit i in layer j

$\Theta^{[j]}$ = matrix of weights controlling function mapping from layer j to layer j+1

$$a_1^{[1]}=g(\Theta_{10}^{[1]}x_0+\Theta_{11}^{[1]}x_1+\Theta_{12}^{[1]}x_2+\Theta_{13}^{[1]}x_3) = g(z_1^{[1]})$$
$$a_2^{[1]}=g(\Theta_{20}^{[1]}x_0+\Theta_{21}^{[1]}x_1+\Theta_{22}^{[1]}x_2+\Theta_{23}^{[1]}x_3) = g(z_2^{[1]})$$
$$a_3^{[1]}=g(\Theta_{30}^{[1]}x_0+\Theta_{31}^{[1]}x_1+\Theta_{32}^{[1]}x_2+\Theta_{33}^{[1]}x_3) = g(z_3^{[1]})$$

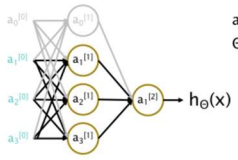
→ primo livello

$\Theta^{[1]} \in \mathbb{R}^{3\times4}$

$$a_1^{[2]}=h_\Theta(x)=g(\Theta_{10}^{[2]}a_0^{[1]}+\Theta_{11}^{[2]}a_1^{[1]}+\Theta_{12}^{[2]}a_2^{[1]}+\Theta_{13}^{[2]}a_3^{[1]})=g(z_1^{[2]})$$

→ secondo livello

$\Theta^{[2]} \in \mathbb{R}^{1\times4}$



$a_i^{[j]}$ = activation of unit i in layer j

$\Theta^{[j]}$ = matrix of weights controlling function mapping from layer j to layer j+1

$$a_1^{[1]}=g(\Theta_{10}^{[1]}a_0^{[0]}+\Theta_{11}^{[1]}a_1^{[0]}+\Theta_{12}^{[1]}a_2^{[0]}+\Theta_{13}^{[1]}a_3^{[0]}) = g(z_1^{[1]})$$
$$a_2^{[1]}=g(\Theta_{20}^{[1]}a_0^{[0]}+\Theta_{21}^{[1]}a_1^{[0]}+\Theta_{22}^{[1]}a_2^{[0]}+\Theta_{23}^{[1]}a_3^{[0]}) = g(z_2^{[1]})$$
$$a_3^{[1]}=g(\Theta_{30}^{[1]}a_0^{[0]}+\Theta_{31}^{[1]}a_1^{[0]}+\Theta_{32}^{[1]}a_2^{[0]}+\Theta_{33}^{[1]}a_3^{[0]}) = g(z_3^{[1]})$$

$\Theta^{[1]} \in \mathbb{R}^{3\times4}$

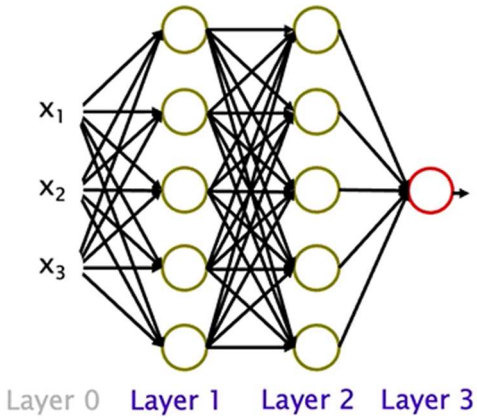
$$a_1^{[2]}=h_\Theta(x)=g(\Theta_{10}^{[2]}a_0^{[1]}+\Theta_{11}^{[2]}a_1^{[1]}+\Theta_{12}^{[2]}a_2^{[1]}+\Theta_{13}^{[2]}a_3^{[1]})=g(z_1^{[2]})$$

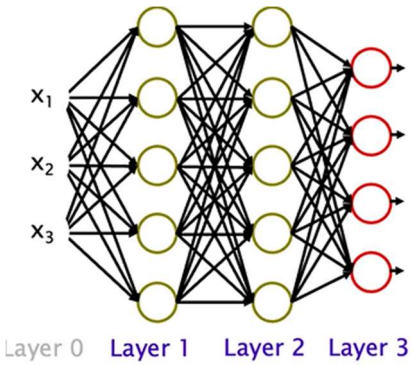
$\Theta^{[2]} \in \mathbb{R}^{1\times4}$

Un'ultima modifica è quella di chiamare gli input del livello 0 come a e non x in modo tale da esprimere le attivazioni del livello 1 in funzione di a.

Questo tipo di rete è una rete di "**forward propagation**", prende i valori di input, li combina linearmente in maniera completamente magliata, applica una compontente non lineare e produce un output che viene a sua volta combinato in maniera lineare e passato all'interno di una componente non lineare.

Esempio: Neural Network a 3 livelli con classificazione binaria



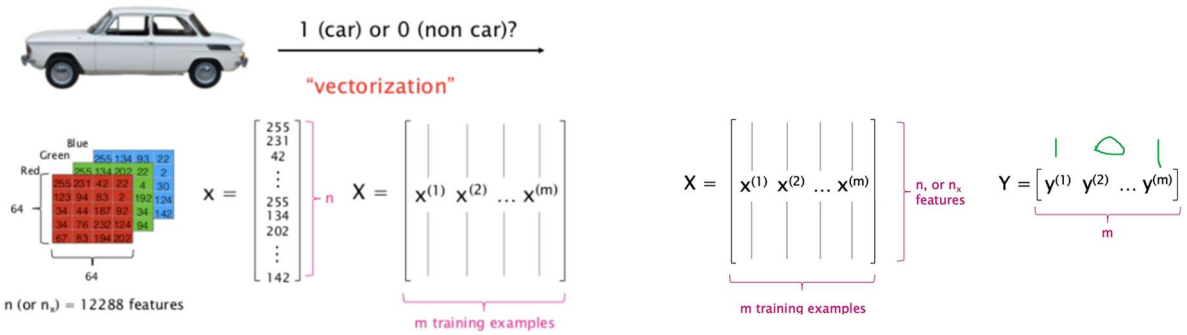


Multiclass class.
(e.g., with 4 classes)

Nota: si è dimostrato che le reti hanno bisogno di più neuroni nei primi livelli

Regressione Logistica come Neural Network

Addestriamo una rete affinché si comporti come una regressione logistica.



Siamo abituati a descrivere il problema come un insieme di features, però in questo caso ci serve un vettore.
Ora il numero di features è dato dal numero di elementi del vettore in ingresso = 64x64x3(canali RGB).

*(nota: 64*64 sarebbe bianco-nero; in caso di colori si hanno 3 canali quindi 64*64*3)*

Alla rete viene passato un certo numero di immagini ognuna rappresentata da un vettore, ognuno dei quali verrà dato in ingresso alla rete, tutti insieme sfruttando la capacità di parallelizzazione.

NOTA:

- N: n dimensione della singola immagine
- M: numero di immagini che sto elaborando

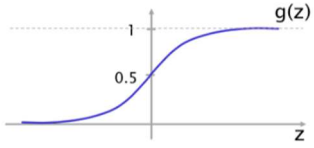
Parlando di regressione logistica, c'è un cambio di terminologia, si abbandonano i θ e si parla di w e b.

- Binary classification: want $0 \le h_\theta(x) \le 1$
- Function applied to $\theta^T x$

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

- Sigmoid or logistic function

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



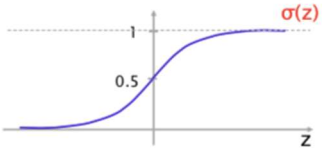
→

- Binary classification: want $0 \le \hat{y} \le 1$
- Function applied to $w^T x + b$

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

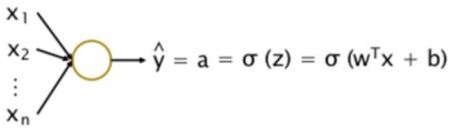
- Sigmoid or logistic function

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



Ora ci serve definire la funzione di costo

- Representation of logistic regression as a neural network



- Regression loss

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \overbrace{\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2}^{\text{Loss}(\hat{y},y)}$$

Questa funzione nel caso della regressione logistica non è convessa e quindi si hanno problemi con il gradient descent, per non avere problemi usiamo la seguente, ottenendo così una **classification loss** che è convessa.

- Classification loss

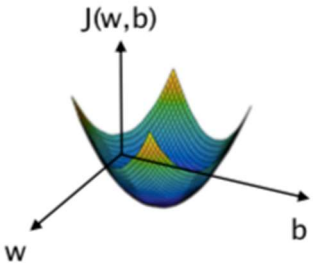
$$\text{Loss}(\hat{y},y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

(for a single training example)

J(w,b) convex

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

Noi vogliamo trovare i vettori **w e b che minimizzano la funzione di costo J(w,b)** usando il gradient descent.



repeat until convergence {

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

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

}

repeat until convergence {

$w := w - \alpha\ dw$

$b := b - \alpha\ db$

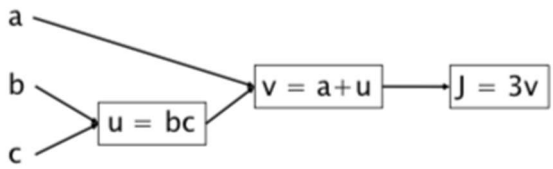
}

Computation graph

(come calcolare le derivate)

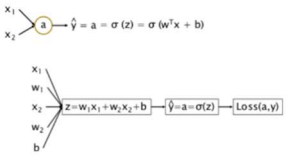
Supponiamo di avere una funzione J(a,b,c) = 3 (a+bc) che può essere scomposta come variabili intermedie:

- u = bc
- v = a + u
- J = 3v

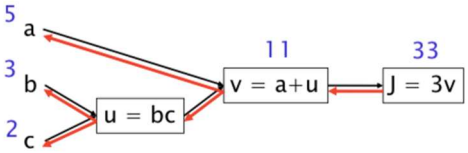


Questo viene chiamato il computation graph.
Dando dei valori a questi ingressi riusciamo a calcolare il valore della funzione attraverso le variabili intermedie.

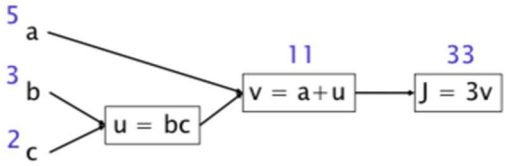
La stessa cosa la si può scrivere per la regressione logistica:



Tutto questo può essere usato per calcolare le derivate, ovvero per fare il backward propagation o back prop.



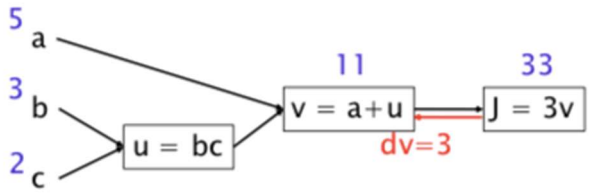
Quindi sappiamo come calcolare J ma ora ci serve appunto come calcolare le sue derivate parziali.
Iniziamo col perturbare una variabile intermedia "v" per scoprire che la derivata di J rispetto a v è pari a 3



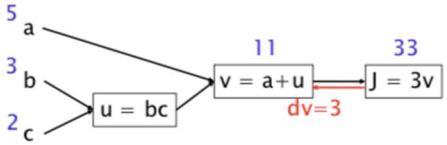
$$\frac{dJ}{dv} = 3$$

v = 11 → 11.001
J = 33.003

Abbiamo quindi fatto un passo indietro ottenendo dv = 3

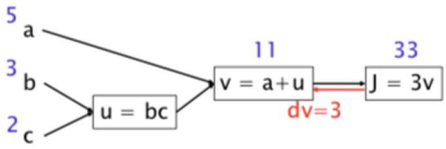


Poi faccio variare "a" e scopro che la minima perturbazione è pari a 3



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

a = 5 → 5.001
v = 5.001+6=11.001
J = 33.003



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

"chain rule"

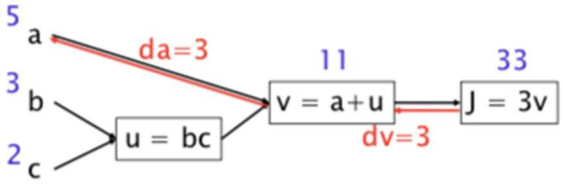
La **"chain rule"** è una proprietà dell'analisi e mi dice sostanzialmente che per capire <u>quanto l'uscita sia influenzata dall'ingresso</u> posso scriverlo come i passi parziali siano influenzati dai cambiamenti intermedi.

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

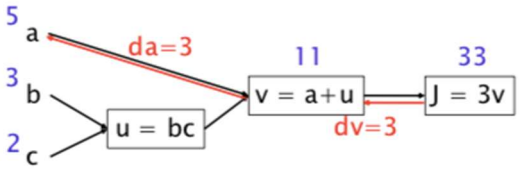
$3 \quad 1$

$a = 5 \rightarrow 5.001$
$v = 5.001+6=11.001$

Si può quindi usare il computation graph per calcolare le derivate parziali.



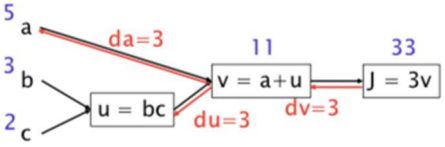
Continuiamo a calcolare le derivate per gli altri componenti



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

$u = 6 \rightarrow 6.001$
$v = 5+6=11.001$
$J = 33.003$

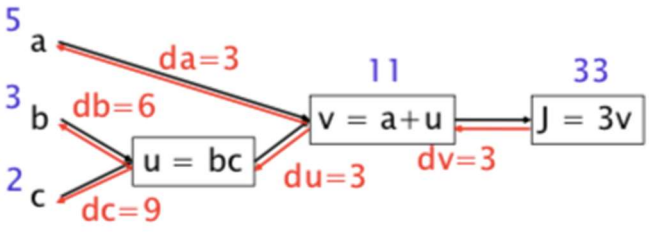
Che è pari a 3

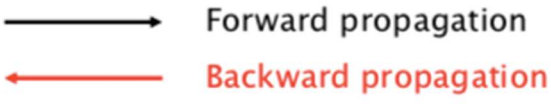


$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6$$

$3 \quad 2$

Quindi alla fine ottengo

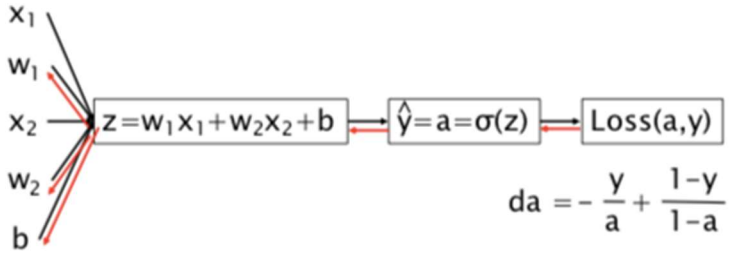




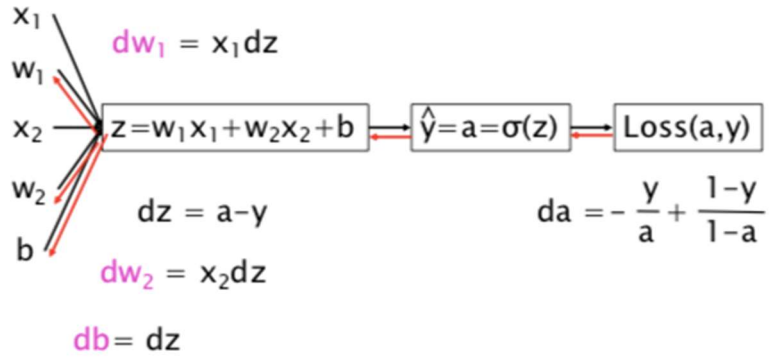
Derivate per la regressione logistica

Questa derivata risulterà dal fatto che essendo regressione logistica abbiamo a che la funzione di attivazione è una sigmoide. La prima cosa che dobbiamo fare è calcolare la derivata del Loss rispetto ad "a" (equivalente a "$\hat{y}$" perché è vista come l'attivazione dell'ultimo livello)

Se abbiamo una corr-entity loss, se abbiamo una sigmoide nell'ultimo livellom si può scrivere la derivata come qui sotto:



Si calcolano poi le derivate per i pesi e il bias:



Ritornando alla scrittura iniziale avremo questo risultato per ogni derivata parziale

Nota: noi abbiamo bisogno di calcolare da, percheè con da, noi otteniamo db, dz e tutti gli altri pesi.

Logistic regression as NN P

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

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

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

...

Sostanzialmente derivando la funzione di costo è come mettere una derivata parziale del Loss rispetto alle variabili.

Per il calcolo del gradient descent avremo invece che per ogni example del training set dovremo calcolare una serie di elementi:

- La combinazione lineare z
- La attivazione “a” applicando la sigmoide su z
- Da “a” ottenere la funzione di costo
- Calcolare le derivate parziali

Infine bisogna dividere tutto per m visto che i valori sono accumulati nelle variabili

PSEUDOCODICE DI 1 PASSO DI GRADIENT-DESCENT

Quando è possibile bisogna evitare i cicli e sfruttare la vettorizzazione

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

for i = 1 to m ➔ Ciclo esterno per passare tutti i training example

$z^{(i)}=w^Tx^{(i)}+b$ → componente lineare z

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

$J+=-[y^{(i)}\log a^{(i)}+(1-y^{(i)})\log(1-a^{(i)})]$ → funzione di costo usando il cross-entroy loss (per un training example)

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

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

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

for j = 1 to n
$dw_j+=\ldots$

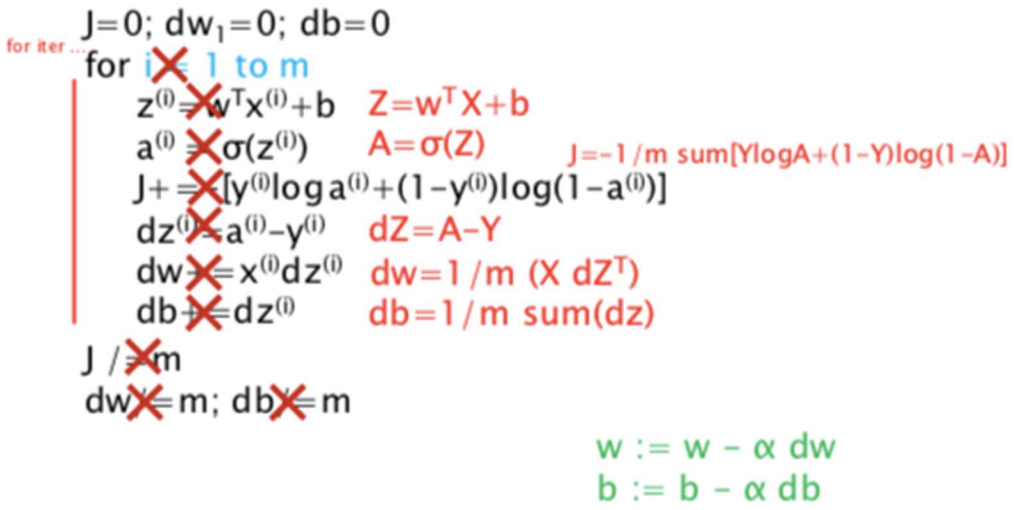
$db+=dz^{(i)}$

→ in questo caso 2, in altri casi di più, ciclo interno per passare tutti i parametri/input

$J\ /=m$

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

Nota: cercare di evitare i cicli, in quanto le GPU sono pensate per parallelizzare. Meglio vettorizzazione come qui sotto.



L'unica cosa che non si può rimuovere è che il ciclo esterno va ripetuto un certo numero di volte fino ad arrivare a convergenza.