



# Implementazione embedded di un ANN in C basata sul MNIST

Riconoscimento di caratteri scritti a mano libera

2019 / 2020

---

Autore:  
Edoardo Cittadini

## Indice

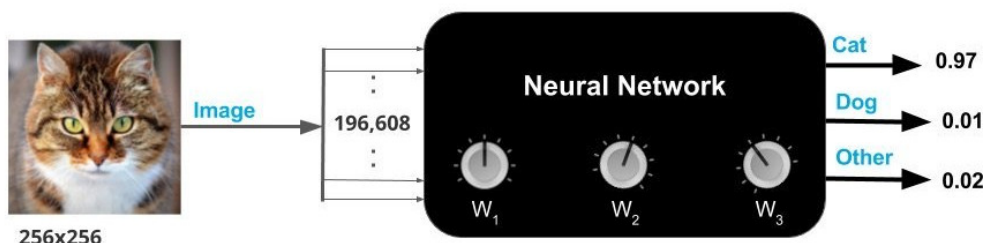
1	Introduzione	2
2	MLP - MultiLayer Perceptron	4
3	Perchè succede questo ?	5
4	La nostra topologia	6
5	MNIST dataset	7
6	Data pre-processing	9
7	Funzione di attivazione - La Softmax	10
8	One-Hot Encoding - Dalle etichette alla codifica binaria	12
9	Forward propagation - Dall' input all'output	13
10	Errore e funzione di loss - Calcolo del funzionale di costo	15
11	Backpropagation - Modifica di pesi e bias	17
12	Early Stopping	22
13	Train the model - Allenamento del modello	24
14	Test set - Prova del modello su nuove immagini	26
15	Net release - Rilascio della rete su piattaforme embedded RT	28
16	Image generator - Script per generare immagini vettorizzate	30
17	Conclusioni	31

# 1 Introduzione

Una rete neurale artificiale, nota anche come ANN (Artificial Neural Network), è un sistema di calcolo ispirato dal funzionamento dei neuroni biologici che compongono il nostro cervello.

Nell' implementazione dell' apprendimento supervisionato (Supervised learning), la rete impara a eseguire il suo compito basandosi su esempi. Tali esempi possono essere di qualsiasi tipo e ciò significa che potenzialmente la nostra rete può imparare a riconoscere qualsiasi input che le viene fornito a meno di un certo margine di errore accettabile (teorema globale dell'approssimazione) e ciò le rende molto flessibili e rappresenta uno dei principali vantaggi nel loro utilizzo pratico. Per esempio nel contesto del riconoscimento d'immagine la rete può imparare a riconoscere immagini contenenti gatti a partire da immagini che sono state manualmente classificate come "gatto" o "non gatto" e utilizzare i risultati dell' allenamento per riconoscere immagini di gatti mai utilizzate in precedenza.

Una ANN è basata su una collezione di unità connesse tra loro, dette anche nodi, ma comunemente chiamate "Neuroni" che approssimano in modo semplificato il modello del neurone artificiale. Esattamente come nel neurone vero, ciascun neurone artificiale è in grado di trasmettere un segnale agli altri neuroni a cui è collegato; tale segnale verrà poi processato e propagato allo stesso modo su tutta la rete fino a raggiungere lo strato di uscita (output layer).

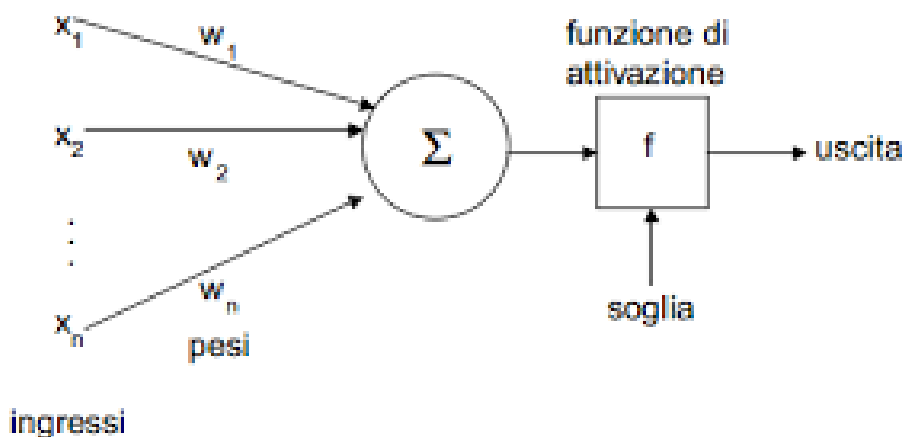


Nelle ANN il segnale trasmesso è un numero reale, e non può essere altro che un numero, e l'uscita di ciascun neurone è calcolata tramite una particolare funzione non-lineare chiamata funzione di attivazione che prende come input la somma pesata dei segnali in ingresso dai neuroni degli strati precedenti. Analizzeremo più approfonditamente le funzioni di attivazione più avanti.

Ogni neurone è caratterizzato da dei pesi e un bias (può anche non esserci); i pesi rappresentano appunto il peso che viene dato a un determinato input quando viene passato come ingresso della funzione di attivazione (i.e. se ho  $w_1 = 0.1$  e  $w_2 = 1$  allora significa che l' input  $x_2$  avrà maggior impatto dell'input  $x_1$  nella funzione di attivazione di quel neurone). I bias invece non hanno collegamenti con gli altri neuroni ma servono matematicamente per traslare

il grafico della funzione di attivazione per evitare che si verifichino attraversamenti dello zero (zero-crossing point). Il bias concettualmente può essere visto come un unità il cui input è l'uscita di un neurone il cui output (del neurone) è costante e sempre pari a 1. Pesi e bias caratterizzano il comportamento fortemente non-lineare del neurone.

I neuroni sono raggruppati in strati (layers); layer diversi possono applicare trasformazioni diverse sull'input (quando cambia la funzione di attivazione). Nel caso di reti Feedforward (propagate solo in avanti), i segnali viaggiano in modo unidirezionale dallo strato di input verso quello di output. Nell'ambito di questo progetto parleremo solo di reti neurali Feedforward.

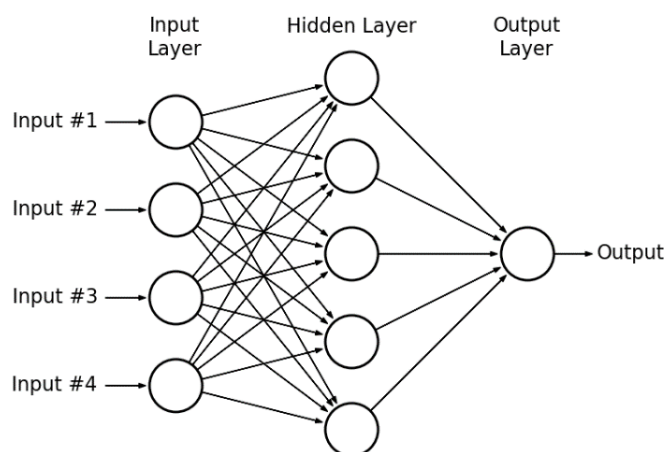


Dalla figura sopra si può vedere una comune rappresentazione di un neurone artificiale che riassume i punti precedentemente citati:

1. Ingressi: I valori  $x$  di input presentati al neurone
2. Pesi: Regolano l'incidenza di un input sulla funzione di attivazione del neurone
3. Sommatoria: La somma pesata degli input costituisce l'input alla funzione di attivazione (matematicamente  $W^T x$ )
4. Soglia: I bias spostano la funzione di attivazione fuori dal range di attraversamento dello zero
5. Funzione di attivazione: funzione non-lineare dell'input che restituisce l'output del neurone

Il bias contribuisce all'input della rete in quanto l'ingresso completo alla funzione di attivazione è dato da  $W^T x + b$ , dove  $W^T$  è la matrice dei pesi che esiste solo in caso di più neuroni in quanto ogni riga rappresenta tutti i pesi degli ingressi al singolo neurone (e quindi nel caso del neurone singolo riportato sopra si riduce ad un vettore riga di dimensione  $n$ ).

## 2 MLP - MultiLayer Perceptron



Il MultiLayer Perceptron, o Percettrone Multistrato, è una delle topologie di rete più utilizzate ed efficienti nell'ambito delle ANN.

Tali reti sono spesso etichettate come "Vanilla networks" alle quali ci si riferisce ad una topologia con un solo strato nascosto (hidden layer).

Un MLP consiste in 3 strati: Input, Hidden layer, Output.

A parte lo strato di input tutti gli altri neuroni degli altri strati utilizzano funzioni di attivazione fortemente non lineari.

L'addestramento di tali reti consiste nel trovare i valori ottimi di pesi e bias, dove ottimo è inteso in senso matematico, e quindi i valori per i quali viene minimizzato l'errore. Tale processo è il più complicato nel processo di creazione, allenamento ed utilizzo della rete e viene effettuato tramite un algoritmo chiamato Backpropagation (Retropropagazione) che usa in modo intensivo le derivate delle funzioni di attivazione per capire quanto la variazione del valore di un peso/bias influisce sul valore finale dell'errore.

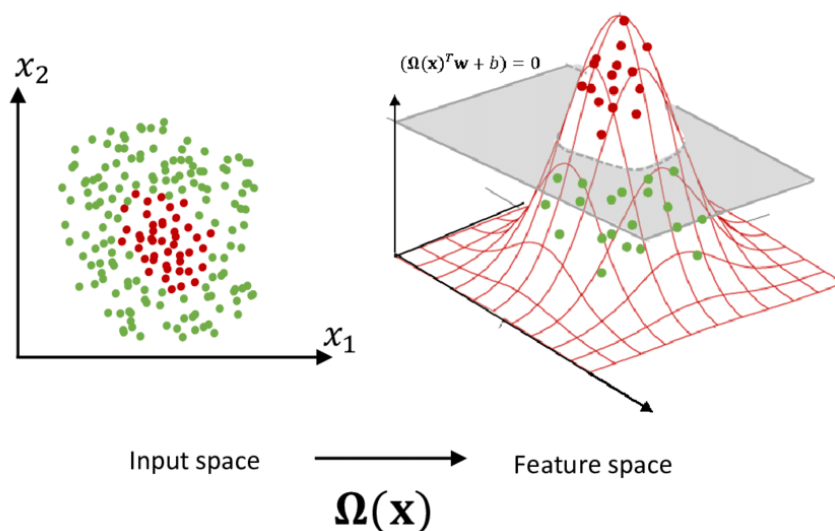
Non preoccupatevi se non avete capito o non avete idea di quanto scritto nelle ultime due frasi, a breve sarò più chiaro ma prima abbiamo bisogno di altri due concetti fondamentali.

**IMPORTANTISSIMO:** Un MLP con un numero a piacere di strati ma con neuroni con solo funzioni di attivazione lineari non ha nessun beneficio rispetto ad una rete sempre con neuroni con attivazione lineare ma con meno strati; perciò gli hidden layers e gli output devono essere sempre funzioni non-lineari degli input.

### 3 Perché succede questo ?

Le funzioni lineari non permettono margine di separazione in uno spazio dei campioni non-lineare; fin qui tutto ragionevole... proviamo ora a chiederci il contrario, ovvero "Cosa succede quando utilizzo una funzione non-lineare ?"

Utilizziamo una categoria di funzioni non-lineari importantissime in matematica, chiamate **Funzioni Kernel**; queste funzioni hanno la notevolissima caratteristica di poter mappare spazi vettoriali non-linearmente separabili in uno spazio a più dimensioni separabile poi con degli Iperpiani, e quindi di fatto lo rendono linearmente separabile.



Dalla figura sopra si vede chiaramente che lo spazio degli input a sinistra non è per nulla separabile con una qualsiasi linea (non è possibile tracciare una linea che separi la classe verde dalla classe rossa).

La funzione Kernel prende in input lo spazio vettoriale linearmente non separabile e lo proietta in un numero di dimensioni più elevato (potenzialmente infinito) in cui le classi sono linearmente separabili, o meglio possono essere separate attraverso un'applicazione affine, che genera un Iperpiano separatore.

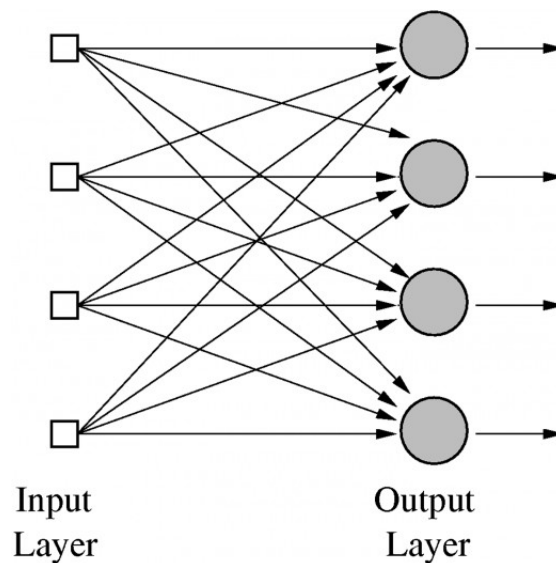
Le funzioni Kernel sono sempre invertibili, ciò significa che è sempre possibile tornare dallo spazio delle Features allo spazio degli Input senza indurre alcun cambiamento sulla disposizione dei dati. La matematica alla base delle funzioni Kernel è molto complicata e fuori dalla trattazione di questa introduzione alle ANN ma tuttavia l'utilizzo di tali funzioni è di vitale importanza per il nostro scopo.

## 4 La nostra topologia

Lo scopo di questo progetto è introdurre alle reti neurali, partendo da zero e avendo piena consapevolezza matematica di quello che si sta implementando, ergo librerie esterne zero (noi non facciamo queste cose), chi vorrebbe ma non sa da dove partire oppure chi come il sottoscritto partendo da keras e python ha voluto poi portare le ANN su microcontrollori.

Sì, la nostra rete alla fine di tutto potrà essere eseguita anche su Arduino.

A tale scopo ho deciso di mantenere tutto estremamente semplice e la topologia che utilizzeremo sarà quella di base riportata sotto senza strati nascosti.



Tale architettura nonostante sia molto semplice ci permetterà di raggiungere un livello di precisione di 93.50% sul training set e del 90.80% sul test set.

Si tratta di una **Rete Densa**; sono definite tali, le reti in cui ogni input è collegato a tutti gli output o equivalentemente gli output dello strato precedente sono tutti collegati a tutti gli input dello strato successivo (ciò caratterizza anche gli MLP ma in generale tutte le reti con strati "fully-connected").

## 5 MNIST dataset

Il dataset MNIST (Modified National Institute of Standards and Technology) dataset ci fornisce uno spazio dei campioni di 60.000 immagini di numeri scritti a mano per il training set col corrispondente vettore di 60.000 etichette utilizzate per l'apprendimento supervisionato e un test set formato nello stesso modo ma con 10.000 campioni.

Tutte le immagini sono in formato 28x28 pixels con valore di ogni pixel codificato in byte mentre le etichette sono di tipo int e corrispondente al numero contenuto nell'immagine.

Le immagini e le etichette sono fornite su 4 file binari:

1. Train images
2. Test images
3. Train labels
4. Test labels

Le immagini sono codificate come vettori di bytes di 784 elementi ( $28 \times 28 = 784$ ), ciò significa che per il training set ci occorrerà una matrice  $[60.000][784]$ .

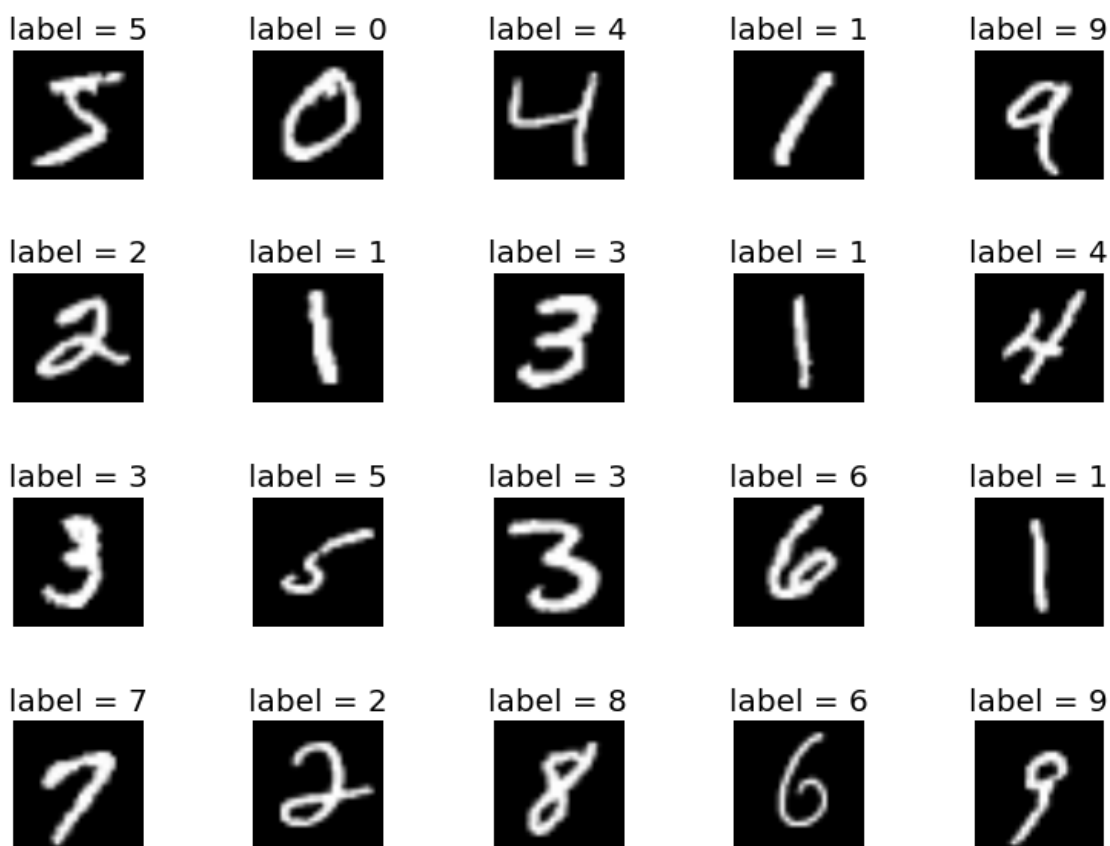
I files delle immagini iniziano con un'intestazione formata da 4 interi a 32-bit (che assurdamamente sono codificati in BIG ENDIAN e quindi vanno trasformati in LITTLE ENDIAN per i processori Intel) che rappresentano:

1. Magic number: 2051 in decimale
2. Size: Numero di immagini nel file
3. Width: Larghezza dell'immagine in pixels
4. Height: Altezza dell'immagine in pixels



I files delle etichette hanno invece un'intestazione di 2 interi a 32-bit che rappresentano:

1. Magic number: 2049 in decimale
2. Size: Numero di etichette nel file



Sopra sono riportate le prime immagini con le relative etichette del training set.

## 6 Data pre-processing

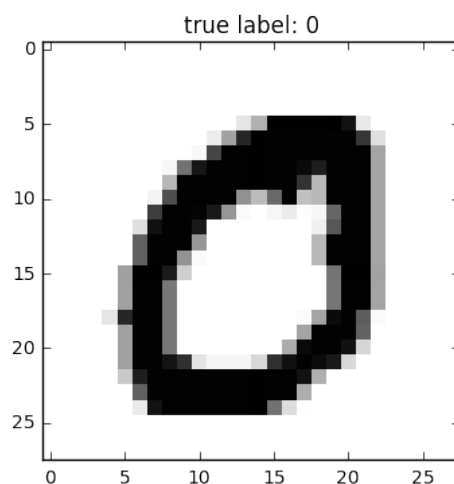
È una fase fondamentale e serve per ridurre le differenze ed i margini di errore nei dati.

Dalla documentazione del dataset su kaggle si vede (altra assurdità) che i colori sono invertiti, perciò 0 rappresenta il bianco e 1 rappresenta il nero. È importante sapere questo per quando andremo noi a codificare immagini scritte a mano come input della rete per provarla.

Nel caso del MNIST ogni byte letto deve essere convertito in float per un motivo che sarà chiaro dopo la prossima frase.

I dati in input devono sempre essere normalizzati; nel caso di questo dataset essendo ogni pixel codificato come byte sappiamo che il massimo valore che un pixel può assumere è  $2^8 - 1 = 255$ , perciò per ogni pixel di ogni immagine faremo la divisione per 255.0 e questo rappresenterà l'input normalizzato alla rete.

Il perchè di tale operazione lo si può osservare immediatamente considerando che la "distanza" tra 0 e 255 è 255 mentre quella tra 0 e 1 è 1 (molto minore).



Stampando da python con il modulo matplotlib un'immagine dopo la normalizzazione il risultato che si ottiene è come quello in figura.

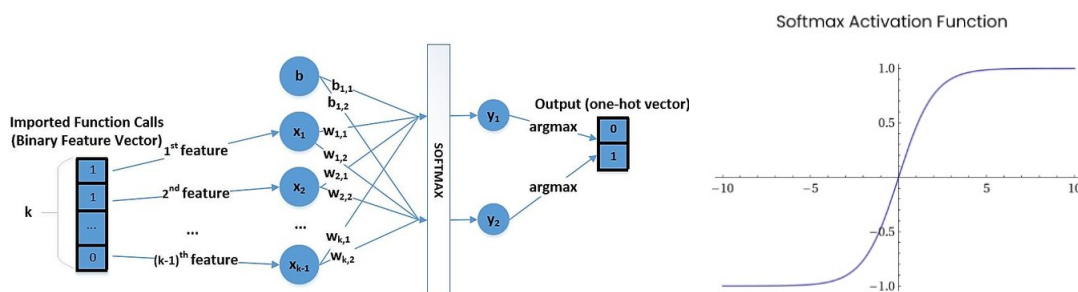
## 7 Funzione di attivazione - La Softmax

In matematica, la Softmax, nota anche come Softargmax o Funzione Esponenziale Normalizzata è una funzione che prende come input un vettore di  $n$  numeri reali e li mappa in una distribuzione di  $n$  probabilità ciascuna proporzionale all'esponenziale del valore di ingresso.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Questo è quanto, prima di applicare la Softmax, alcune delle componenti del vettore potrebbero essere negative o maggiori di 1, ma dopo l'applicazione della Funzione Esponenziale Normalizzata ciascuna componente sarà nell'intervallo  $[0; 1]$  e la somma di tutte le componenti del vettore sarà sempre 1.

Ha perfettamente senso infatti, che l'input più grande sia mappato con la probabilità più alta e molto spesso la Softmax è la funzione dello strato di uscita della rete perchè è in grado di normalizzare l'uscita globale della rete rispetto al numero di classi definite dal problema di classificazione (nel nostro caso numeri da 0 a 9, quindi 10 classi).

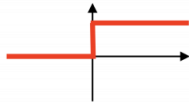
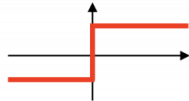
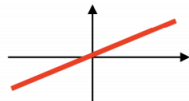
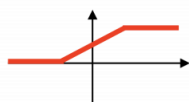
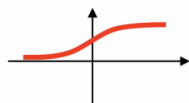
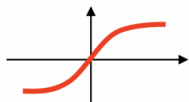
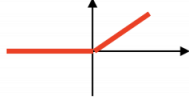
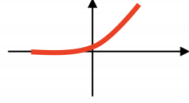


**IMPORTANTISSIMO:** La somma delle probabilità in uscita dalla Softmax deve tassativamente essere 1.

Dalla figura sopra si vede che la forma propria del grafico della funzione è una

"S", infatti la Softmax appartiene alla famiglia delle "S-shape functions" che si comportano come funzioni esponenziali all'inizio per poi frenare e mantenere valore costante negli stati chiamati di "maturità". Tale nomenclatura è tipica delle funzioni logistiche tra cui anche la Sigmoide e la Tangente Iperbolica. L'ampiezza della "S" è determinata da un parametro che determina quanto gli elementi di una popolazione (in senso statistico) influenzano la crescita della popolazione stessa.

Nella tabella sotto sono riportate alcune tra le altre principali funzioni di attivazione

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Copyright © Sebastian Raschka 2016  
(<http://sebastianraschka.com>)

## 8 One-Hot Encoding - Dalle etichette alla codifica binaria

Le ANN lavorano sempre con numeri reali; tutto quello che non è un numero deve essere convertito in un valore reale per poter essere processato dalla rete.

Nel caso del nostro problema tale situazione non è evidente perchè trattiamo numeri, ma tuttavia se considerassimo un problema di classificazione tra "cane", "gatto" e "topo", queste tre etichette devono essere espresse come numeri.

Una delle tecniche più utilizzate è il One-Hot Encoding che consiste nel creare una matrice quadrata in cui ogni riga e ogni colonna rappresenta la codifica binaria della classe astratta, ad esempio cane [0 0 1], gatto [0 1 0], topo [1 0 0]. Per fornire un esempio, in tabella è riportato un problema di classificazione a 5 classi:

	A	B	C	D	E	F	G	H	I
1	Original data:		One-hot encoding format:						
2	id	Color	id	White	Red	Black	Purple	Gold	
3	1	White	1	1	0	0	0	0	
4	2	Red	2	0	1	0	0	0	
5	3	Black	3	0	0	1	0	0	
6	4	Purple	4	0	0	0	1	0	
7	5	Gold	5	0	0	0	0	1	
8									
9									

Le etichette codificate con tecnica One-Hot vengono chiamate variabili categoriche e rappresentano l'uscita della Softmax in binario, basta infatti impostare ad 1 la probabilità maggiore e a 0 tutte le altre per ottenere la codifica One-Hot della previsione della rete.

Nel nostro problema di classificazione a 10 classi i numeri sono rappresentati come un vettore di 10 elementi la cui componente ad 1 corrisponde al numero identificato e tutte le altre a 0.

Ad esempio i valori 1 e 2 saranno

1. one = [0 1 0 0 0 0 0 0 0 0]
2. two = [0 0 1 0 0 0 0 0 0 0]
3. ....

## 9 Forward propagation - Dall' input all'output

La propagazione in avanti altro non è che il normale ciclo di predizione della rete.

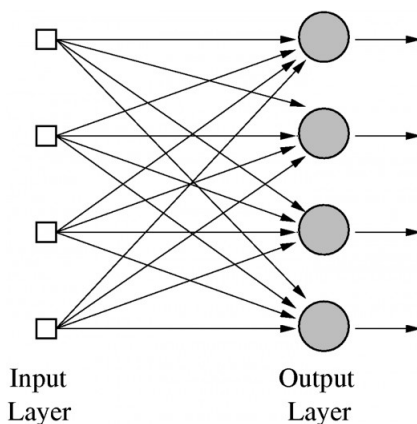
La transizione da uno stato all'altro viene effettuata applicando una trasformazione affine data da  $W^T x + b$  con  $W^T$  la matrice dei pesi,  $x$  l'input della rete e  $b$  i bias.

In matematica una trasformazione affine è la stessa cosa di una trasformazione lineare con l'aggiunta di una costante non nulla (nel nostro caso  $b$ ).

La nostra rete ha 10 neuroni di uscita, perciò la trasformazione deve restituirci un vettore di 10 elementi da poter poi passare alla Softmax. Eseguendo un paio di conti di algebra lineare sui prodotti riga per colonna otteniamo che

$$W^T = (10, 784) \quad x = (784, 1) \rightarrow W^T x = (10, 1)$$

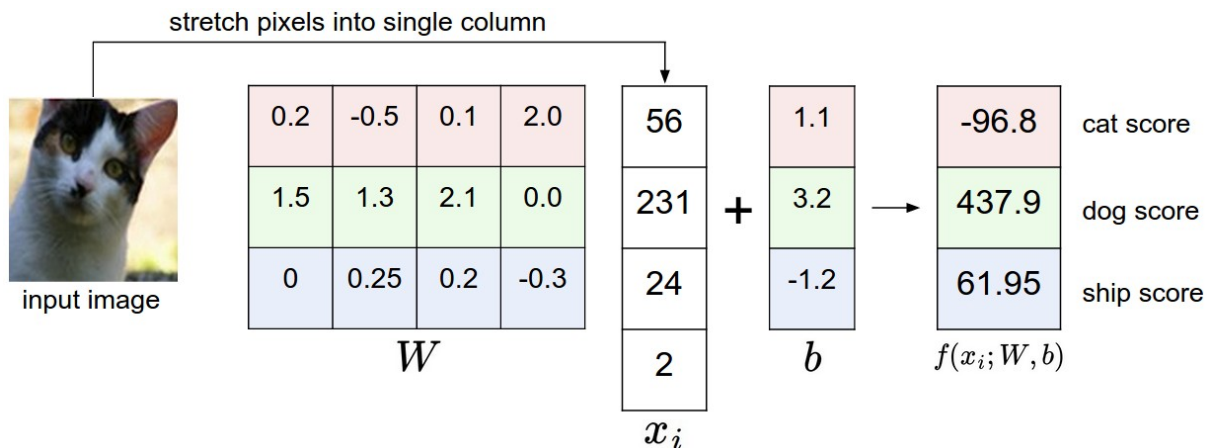
$$W^T x = (10, 1) \quad b = (10, 1) \rightarrow W^T x + b = (10, 1)$$



Proponendo di nuovo il nostro modello di rete possiamo capire che le due righe sopra rappresentano la trasformazione dai nodi bianchi quadrati di input verso i nodi rotondi grigi (Softmax). Il risultato finale, ovvero il vettore colonna di 10 elementi, può essere interpretato sia come l'output dello strato zero (i quadrati bianchi) sia come l'input dello strato uno (i cerchi grigi). Questo vettore viene

quindi passato alla funzione di attivazione Softmax dello strato di uscita che mapperà le 10 componenti nelle relative 10 probabilità (la cui somma dovrà sempre sempre essere 1). A questo punto basterà convertire il vettore della probabilità nella variabile categorica corrispondente per ottenere la predizione della rete.

Riporto sotto una figura che mostra graficamente quando scritto sopra in termini matematici (la  $W$  considerata è già trasposta perciò non si applica  $W^T$ ).



Il processo di feedforwarding, o propagazione in avanti, è il normale procedimento di predizione della rete, **anche del modello perfettamente allenato**; nessuna magia, pura matematica e statistica.

Il progetto è quindi ora finito... non proprio, e se la predizione della rete fosse sbagliata ?

## 10 Errore e funzione di loss - Calcolo del funzionale di costo

L'uscita di una rete viene espressa in termini di errore (loss) calcolato utilizzando una determinata funzione di costo chiamata Loss function.

Esistono varie funzioni di costo, tuttavia per mantenere l'obiettivo 1 del progetto, ovvero la semplicità, utilizzeremo la più semplice ma comunque molto utilizzata  $MSE$  (Mean Squared Error), il famoso Errore Quadratico Medio definito come segue

$$MSE = \frac{1}{2} \sum_{i=0}^n (target_i - out_i)^2$$

Da qua seguono 2 osservazioni:

1. Il valore  $\frac{1}{2}$  è portato fuori dalla sommatoria perchè costante e comune a tutti gli addendi
2. Il valore  $\frac{1}{2}$  ci fa comodo quando calcoleremo la derivata perchè si annullerà con l'esponente  $2 * \frac{1}{2} = 1$

Le variabili della funzione *target* e *out* sono rispettivamente una rappresentante l'uscita esatta della rete, ovvero target, mentre out è l'uscita **in termini di probabilità** della rete.

Facendo riferimento al solito esempio di "cane", "gatto", "topo" e supponendo che il valore esatto da ottenere sia "topo" = [1 0 0] e che le probabilità ottenute in output siano "out" = [0.3 0.5 0.2] allora possiamo calcolare il valore di loss come

$$loss = 0.5 * [(1 - 0.3)^2 + (0 - 0.5)^2 + (0 - 0.2)^2] = 0.5 * (0.49 + 0.25 + 0.04) = 0.39$$

Un errore comune tipico di chi si avvicina per le prime volte al mondo delle ANN è pensare che una rete "impari" solo dagli esempi classificati in modo errato e soprattutto pensare che l'apprendimento della rete sia pilotato dal punteggio di precisione (accuracy) ottenuta sul trainig set.



Entrambe queste affermazioni sono sbagliate, ma procediamo con ordine:

1. **Affermazione 1:** Abbiamo mostrato come calcolare il valore di loss, l'errore proprio della rete per la predizione, e abbiamo sottolineato e lo ripetiamo che la variabile out del MSE è rappresentata dal vettore delle probabilità. Se perciò, mantenendo sempre il problema "cane", "gatto", "topo" dell'esempio precedente, cambiassi le probabilità ad esempio con "out" =  $[0.9 \ 0.1 \ 0.0]$  otterrei una classificazione corretta di out ma posso anche calcolare il valore di loss pari a  $loss = 0.01$ . Osserviamo che il valore di loss non è nullo e quindi non lo sarà manco la sua derivata prima e perciò la rete ha la possibilità di migliorarsi anche utilizzando gli esempi classificati correttamente
2. **Affermazione 2:** Il livello di apprendimento della rete è dettato dal valore della funzione di loss. Vediamo il problema da un punto di vista diverso, non pensando in chiave "ne ho beccato 50 su 75" ma bensì in termini di "quanto è basso il valore di loss che ho ottenuto". Apparentemente le due cose appaiono scollegate ma in realtà sono fortemente legate dalla funzione di backpropagation utilizzata per aggiornare pesi e bias. Il valore di loss ci indica quanto stiamo sbagliando nel fare la predizione. L'allenamento della rete è impostato come un problema matematico di ottimizzazione in cui si cerca di minimizzare l'errore, ovvero un problema dove la funzione obiettivo è la funzione di loss e la regione fattibile  $\Omega$  è definita dai vincoli affini  $W^T x + b$ . Trovare un minimo di tale funzione significa trovare un punto appartenente ad  $\Omega$  in cui il valore della sua derivata prima sia 0, e come vedremo in seguito, ciò è possibile solo se tutte le componenti di target corrispondono con le componenti di out. In pratica non si avrà mai un valore di loss pari a 0 ma si cercherà di farcelo avvicinare il più possibile; tanto più il loss è basso tanto più le predizioni saranno robuste. Per riassumere in modo molto intuitivo quanto appena detto, si può affermare in modo del tutto informale che "la rete non migliora aumentando la percentuale di precisione ma migliora riducendo l'errore"; da qui si capisce perfettamente che sbagliare meno equivale a dire di essere più precisi ma matematicamente i due concetti portano a funzioni completamente diverse.

## 11 Backpropagation - Modifica di pesi e bias

La correzione delle due precedenti affermazioni spiana ora la strada all'essenza delle ANN, la retropropagazione dell'errore per poter aggiornare i pesi e rendere la nostra rete sempre migliore dopo ogni ciclo di allenamento.

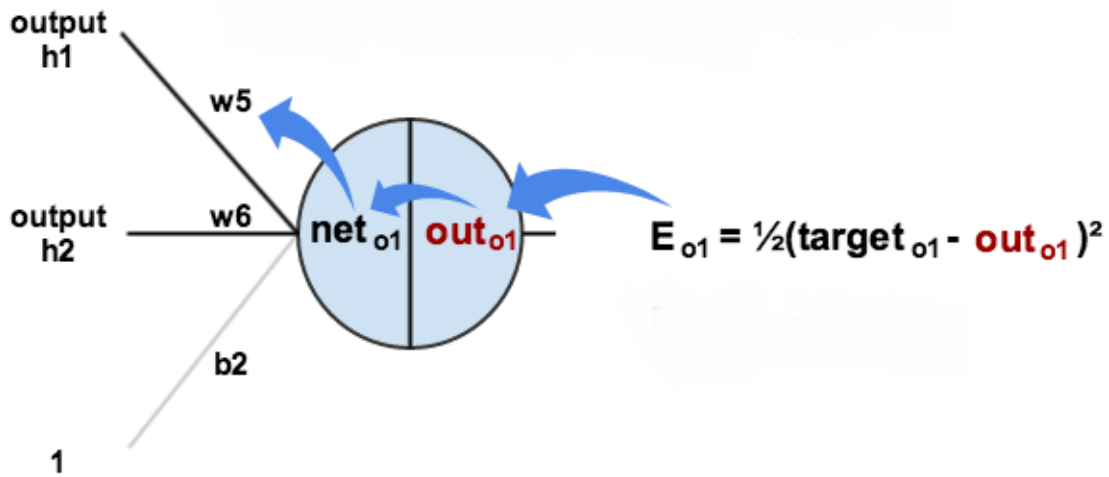
La nostra topologia non ha hidden layers, ciò è dovuto sia al fatto che vogliamo sempre attenerci all'obiettivo 1, ovvero la semplicità di realizzazione, sia al fatto che vogliamo che il modello non abbia troppi parametri in quando desideriamo rilasciare il modello allenato su piattaforme embedded a medio-basse prestazioni come Arduino.

Nella backpropagation il primo passo da eseguire è la derivata della funzione di loss rispetto all'uscita del neurone e quindi avremo

$$\frac{\partial E_{tot}}{\partial out_i} = \frac{1}{2} 2(target_i - out_i) * -1 + 0 = out_i - target_i$$

Questo scritto sopra altro non è che il gradiente  $\nabla$ , il vettore delle derivate parziali, della funzione di loss (MSE).

In realtà per la backpropagation ci serve qualcosa di leggermente più complesso; la catena di derivate (derivative chain) che ci fornirà un indice di quanto il valore di loss varia al variare del singolo peso preso in esame. La catena per la nostra topologia non viene retropropagata all'indietro negli altri strati ma viene applicata solo allo strato di uscita. La derivative chain, aiutandoci con la figura del neurone sotto per semplificare la nomenclatura, è espressa come



$$\frac{\partial E_{tot}}{\partial w_5} = \frac{\partial E_{tot}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

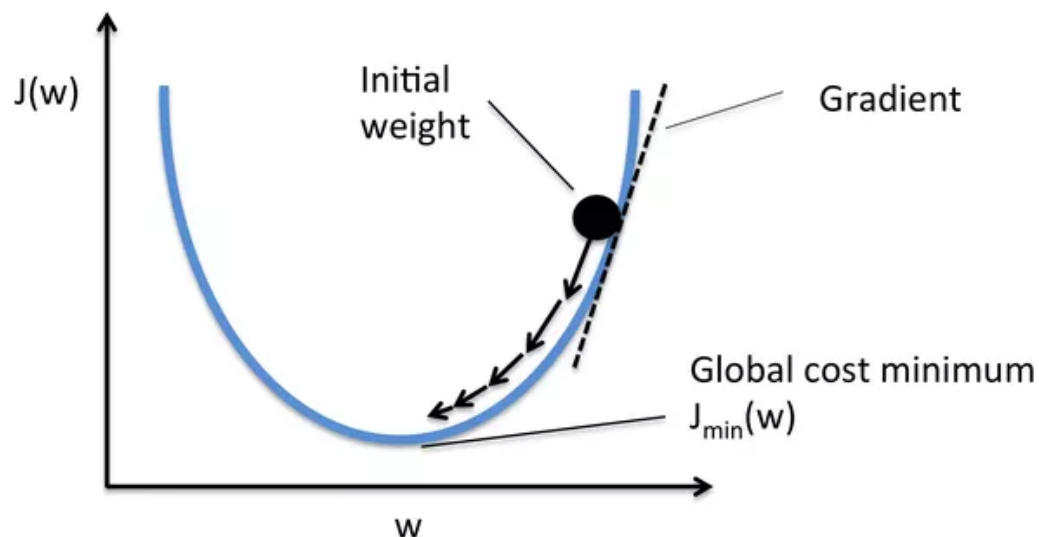
La formula matematica appena descritta si legge come "Il cambiamento dell'errore (loss) al variare del peso  $w_5$  è ottenuto dalla variazione dell'errore rispetto all'uscita (in probabilità) del neurone preso in analisi che a sua volta è influenzato dalla variazione della funzione di attivazione che a sua volta è influenzata dal cambiamento del peso  $w_5$  che vogliamo modificare".

È ora chiaro che ci servono 3 gradienti  $\nabla$  per poter modificare con successo in modo ottimo i pesi della nostra rete. Sostanzialmente stiamo applicando l'algoritmo di discesa del gradiente (Gradient Descent) alla funzione di loss.

Come ? Tre gradienti per calcolare **una** discesa del gradiente ??? Esatto. La Rete Neurale altro non è che una funzione matematica estremamente complessa, per cui non tracciabile e non esprimibile in modo esplicito (a meno di reti banali), che mappa uno spazio vettoriale altrettanto complesso e viene utilizzata in modalità **Black Box**. Sviluppatori ed esperti del settore hanno a disposizione una serie di strumenti con cui è possibile fornire degli input, osservare le uscite

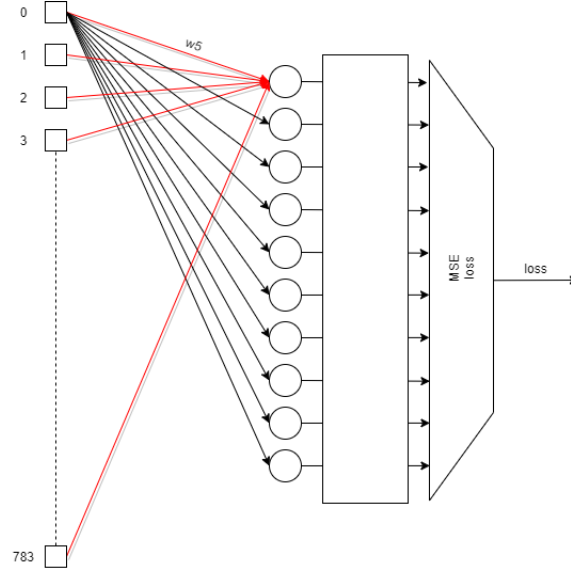
ed in base a queste (le uscite) effettuare delle azioni correttive basate solo sulla osservazione dei valori (è lo stesso concetto che si applica in teoria del controllo quando si usano gli osservatori/stimatori di stato del sistema come ad esempio l'Osservatore ottimo di Luenberger, meglio noto come Filtro di Kalman).

Prima di proseguire coi calcoli, capiamo cosa succede esattamente quando alleniamo la rete



**IMPORTANTISSIMO:** La funzione di costo (loss) non può essere una funzione qualsiasi; tutte le funzioni di questa categoria **devono necessariamente** essere **convesse** perchè la condizione di convessità ci garantisce l'esistenza di un minimo.

Bene, ciascuna delle frecce nere verso il minimo nella figura sopra rappresenta il miglioramento della rete, o matematicamente la convergenza del gradiente della funzione di loss verso il minimo, ed è dovuto all'aggiornamento progressivo dei pesi che è esattamente quello che facciamo con la catena di derivate descritta sopra che sviluppiamo ora meglio nel dettaglio.



La prima delle 3 derivate parziali l'abbiamo già espressa ed è la derivata della Loss Function

$$\frac{\partial E_{tot}}{\partial out_i} = out_i - target_i$$

La seconda è la derivata della funzione di attivazione dello strato di uscita, nel nostro caso la Softmax che dopo alcuni passaggi appare nella sua forma finale come segue

$$\frac{\partial Softmax}{\partial net_i} = \frac{e^{net_i} * \sum_{k=0, k \neq i} e^{net_k}}{(\sum_{k=0} e^{net_k})^2}$$

L'ultima derivata che ci interessa è data dalla variazione dell' input alla funzione di attivazione (Softmax) rispetto alla variazione del peso che vogliamo modificare. Ma l'input alla funzione di attivazione è dato dalla sommatoria degli input moltiplicati per i relativi pesi più il bias, il che significa che nella derivata parziale rispetto al peso  $w_5$  che stiamo considerando tutte le componenti che non lo

coinvolgono direttamente sono costanti e perciò quando deriviamo diventano 0. Il tutto quindi si riduce semplicemente a

$$\frac{\partial net_i}{\partial w_5} = (w_5 * input_0) + (w_6 * input_1) + \dots + bias_0 = input_0 + 0 + \dots + 0 = input_0$$

Perfetto, abbiamo finito la catena di derivate e siamo ora in grado di ripetere per tutti i pesi da aggiornare lo stesso ragionamento. In caso di topologie più complesse, con strati nascosti il ragionamento è lo stesso e si continua a propagare all'indietro la catena di derivate fino all'input della rete. Sempre per l'obiettivo 1, la semplicità e la possibilità di rilascio su piattaforme embedded come Arduino, ho scelto di spiegare l'implementazione di una rete senza strati nascosti ma comunque dalle buone prestazioni.

Ricapitolando formuliamo di seguito tutto il passo di retropropagazione; si tenga presente che per quanto possano sembrare tante le variabili, in realtà la quasi totalità di queste sono note dalla forward propagation che precede immediatamente la backpropagation.

$$\frac{\partial E_{tot}}{\partial w_5} = (out_0 - target_0) * \frac{e^{net_0} * \sum_{k=0, k \neq 0} e^{net_0}}{(\sum_{k=0} e^{net_0})^2} * input_0$$

Benissimo, quanto appena scritto, ovvero tutta la catena di derivate in letteratura viene anche chiamata **Delta rule** ed indicata con  $\delta$ .

Un iperparametro importantissimo, da impostare con cura, è il **Learning rate** (tasso di apprendimento), che altro non è che un fattore di scala da applicare alla Delta rule prima di modificare il valore del peso e viene indicato con la lettera greca "eta", il cui simbolo è  $\eta$  (solitamente si ha  $\eta_{bias} \leq \eta_{pesi}$ ).

L'espressione finale si riduce quindi a

$$w_5 = w_5 - \eta \delta$$

e lo stesso ragionamento deve essere ripetuto per tutti i 7840 pesi più i 10 bias.

## 12 Early Stopping

Nel Machine Learning l'Early Stopping è una forma di regolarizzatore usato per evitare l' overfitting quando si allena in modo supervisionato con un metodo iterativo come la Discesa del gradiente, che è esattamente quello che facciamo.

Nella programmazione di alto livello (python + keras), tale funzione permette di impostare alcuni parametri per definirne in modo dettagliato il comportamento

1. Monitor: È la variabile che viene controllata
2. Mode: Se il trigger deve avvenire per un delta positivo o negativo
3. Patience: Numero di epoche da attendere per di interrompere l'allenamento
4. Restore\_best\_weights: Riporta la rete allo stato migliore raggiunto

Un obiettivo mio personale è cercare di riproporre in maniera più simile possibile le funzioni dei famosi framework di machine learning. N.B. io non conosco l'implementazione interna di tali funzioni in python-Keras ma ho sviluppato le mie funzioni C in modo che implementino la stessa funzionalità. Non avendo livelli di complessità elevati, ma soprattutto non avendo un Validation set, i parametri precedenti sono stati leggermente semplificati:

1. Il monitor non è impostabile ma è di base "loss"
2. Non si ha la possibilità del restore best weight, questo perchè il training della nostra rete viene eseguito su CPU (contro la GPU di Keras) e pertanto salvare tutti i pesi ed i bias del modello per ciascun miglioramento rallenta parecchio tale processo. Tuttavia impostando correttamente i Learning Rates e il parametro "patience" si può arrivare facilmente, per questa rete, a stoppare il training al momento ottimo accelerando così l'esecuzione del training e rendendo il runtime molto più snello
3. La modalità per le stesse ragioni elencate al punto precedente è solo in direzione di discesa, in Keras sarebbe "min", che è la direzione verso il minimo della loss, ovvero quello che interessa a noi.

4. È invece certamente impostabile il valore "patience"
5. Un altro parametro settabile è il "min-delta" ovvero il valore di differenza tra due valori di loss consecutivi sotto il quale viene attivata la conta di "patience", se durante il processo di conta si dovesse riscontrare un valore della differenza maggiore di "min-delta" il contatore di patience verrà azzerato.

Ritengo che mantenere l'interfaccia di tali funzioni simili tra i framework e i progetti privati abbia il duplice vantaggio di permettere a chi già ha lavorato con Keras/Tensorflow di poter utilizzare le funzioni nello stesso modo al quale è già abituato e viceversa aiutare chi sta imparando per la prima volta da questo progetto a poter poi fare il passo successivo con Colab + Python + Keras.



## 13 Train the model - Allenamento del modello

Abbiamo ora tutti gli elementi per poter passare alla fase di allenamento vera e propria. Sappiamo che la fase di allenamento si comporta inizialmente come una previsione normale della rete, a cui segue la backpropagation (retropropagazione dell'errore) ed il controllo della Early stopping.

Alla fine dell'allenamento pesi e bias andranno salvati, sennò tutto il modello sarà perso. Nell'ambito dei sistemi embedded, tipo Arduino, è necessario caricare in modo hardcoded pesi e bias nei relativi vettori per via dell'assenza di un file system strutturato (è più semplice ed efficace inizializzarli direttamente che caricarli). Quanto appena detto non è il motivo principale per cui i vettori devono essere istanziati e inizializzati a priori, infatti date le ridotte capacità delle memorie, l'unica speranza di poter salvare e caricare vettori grandi, tipo quello dei pesi, (7840 floats) è salvarli nella memoria programmi come costanti utilizzando la macro PROGMEM.

Potrete direttamente sperimentare che eliminando PROGMEM dalla dichiarazione sfonderete immediatamente la massima capacità della memoria, inoltre purtroppo il codice necessita di almeno Arduino MEGA ma può essere caricato pure sulle schede STM32F103C (le meglio note "Blue/Black Pill"). Su Arduino UNO e Nano si arriva ad occupare il 129% della memoria.

```
int main(){
    int i,k;

    if(mode == TRAIN){
        load_training_set_images(train_images_p, training_images);
        load_test_set_images(test_images_p, test_images);
        load_labels(train_labels_p, training_labels, TRAIN_LABELS);
        load_labels(test_labels_p, test_labels, TEST_LABELS);
        initialize_weights_and_bias(weights_matrix, bias);

        for(i = 0; i < EPOCHS; i++){
            for(k = 0; k < TRAIN_IMAGES; k++){
                get_image_and_label_from_dataset(input_image, training_images, k, training_labels, &expected_label);
                affine_transformation(weights_matrix, input_image, bias, res);
                cumulative_output_softmax(res);
                predict(res, &actual_label);
                update_statistics(actual_label, expected_label, &correct_classifications);
                backpropagation(weights_matrix, input_image, actual_label, expected_label, bias, res, &tot_loss);
            }
            squared_error_loss_function(&tot_loss, TRAIN_IMAGES);
            early_stopping(&tot_loss, &prev_tot_loss, MIN_DELTA, PATIENCE, &patience_cnt, &i, EPOCHS, ENABLED);
            print_epoch_results(i, &correct_classifications, TRAIN_IMAGES, &tot_loss, &prev_tot_loss, patience_cnt);
        }

        evaluate_test_set(test_images, test_labels);
        save_trained_model(weights_matrix_p, weights_matrix, bias_p, bias);
    }
```

Il ciclo di training deve necessariamente essere eseguito su un PC e si presenta come riportato in figura alla pagina precedente.

Il numero di epoche determina quante volte la rete processerà tutto il dataset di 60.000 immagini; se usate l' Early stopping, attivabile impostando l'ultimo parametro della funzione come { ENABLED, DISABLED }, è consigliabile sovrastimare il numero di epoche perchè tanto in ogni caso tale numero verrà automaticamente regolato dalla funzione `early_stopping()` stessa.

Per tutto il dataset, che costituisce un'epoca, il ciclo consiste nel prelevare un'immagine e la relativa etichetta dal training set, segue poi tutto il passo di propagazione in avanti fino al risultato finale. Quando si ottiene la predizione della rete è possibile calcolare l'errore ed effettuare la retropropagazione per via del fatto che il risultato è noto dall'etichetta prelevata insieme all'immagine. Nella backpropagation i pesi ed i bias vengono aggiornati.

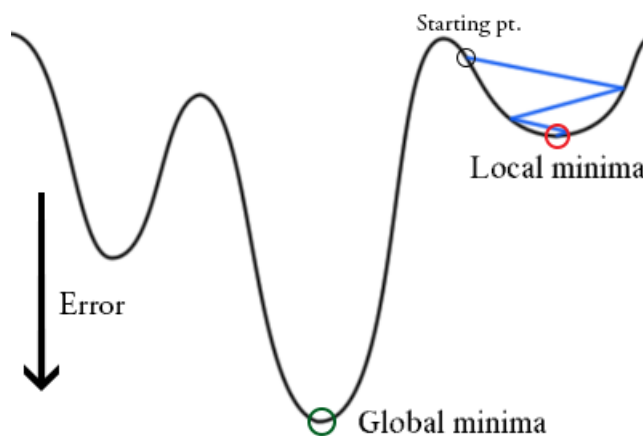
Dopo aver valutato e migliorato la rete per tutti i campioni di un'epoca si considera il valore di loss come la media dei valori delle 60.000 loss ottenute e tale valore viene utilizzato per valutare l'early stopping.

Tutto questo procedimento si ripete per il numero di epoche stabilito a priori, dopo il quale il programma esegue una valutazione del test set, salva pesi e bias in files binari e termina l'esecuzione.

## 14 Test set - Prova del modello su nuove immagini

Ora che abbiamo un modello allenato e conosciamo i valori di precisione (accuracy) del train e del test set dobbiamo capire se i risultati ottenuti sono soddisfacenti o se sarà necessario un nuovo allenamento.

Ricordiamo che il punto di partenza del training non è sempre lo stesso, in quanto i pesi sono inizializzati in modo casuale, e che tale punto influisce in modo non trascurabile sui risultati dell'allenamento ne segue perciò che i risultati degli allenamenti possono essere al più simili ma mai uguali, o nel peggiore dei casi molto diversi.



La figura sopra evidenzia il motivo per cui il punto di partenza incide sui risultati finali dell'allenamento; esistono tecniche come il "momentum" per uscire dai minimi locali ma il successo non è garantito.

Generalmente i risultati ottenuti sul test set riportano punteggi di accuracy leggermente inferiori rispetto a quelli ottenuti sul training set, ma in realtà è solo l'esperienza e la pratica che permette ad un esperto di capire se effettivamente il compromesso raggiunto è buono. È invece facile capire se si è verificato overfitting.

Per la nostra rete i migliori risultati sono stati raggiunti coi parametri che vi fornisco di base che portano l'accuracy del training set al 93.40% e quella del test set a 90.79%; risultati di tutto rispetto per una topologia così semplice.

Leggere i punteggi di accuracy e constatare di quanto differiscono è sufficiente per poter trarre conclusioni ? **ASSOLUTAMENTE NO!** ricordiamoci che la bontà di una rete si basa sulla ricerca di un minimo della loss, per cui è opportuno che tale valore sia il più possibile vicino allo 0. Valori di accuracy più alti ma con loss più alto sicuramente sono peggiori di soluzioni con accuracy inferiore e loss inferiore, questo ha perfettamente senso in quanto un valore di loss più alto indica una maggiore tendenza della rete all'errore e le accuracy più alte probabilmente sono dovute a casualità statistiche.

Per rendere meglio il concetto, dato un test set e nota la probabilità di classificare correttamente un elemento, supponiamo 30%, consideriamo la combinazione per la quale per un vasto numero di campioni ricado proprio sul 30% invece del più probabile 70%. Ecco questo è esattamente un esempio in cui si raggiungono punteggi alti di accuracy nonostante il valore di loss rimanga alto.

In conclusione accuracy e loss devono **sempre** essere valutati dipendentemente l'uno dall'altro, solo questo è il metro che ci permette di valutare una rete in prima istanza.

Esistono tuttavia tecniche più complesse che vengono utilizzate in parallelo a quanto descritto finora per rafforzare la certezza della bontà del modello. Alcune tra queste tecniche sono Curve ROC, Confusion Matrix e K-Fold Cross-Validation.

## 15 Net release - Rilascio della rete su piattaforme embedded RT

Ora che il nostro modello è allenato possiamo utilizzarlo per fare predizioni.

Iniziamo la fase di rilascio per piattaforme embedded creando lo sketch Arduino che dividiamo nel programma principale ed un file .h che conterrà la matrice dei pesi, il vettore dei bias e le varie macro.

Per salvare i valori nella memoria programmi utilizziamo

```
const PROGMEM float weights_matrix[10][784] = {...};  
const PROGMEM float bias[10] = {...};
```

Per aumentare la leggibilità del codice e per assicurarci che non ci siano altre variabili definite da un altro programmatore riportiamo nelle variabili globali i due vettori precedenti dichiarati come "extern"; tale parola chiave rende noto al compilatore che quella specifica variabile esiste ma è istanziata ed inizializzata in un altro file, nel nostro caso il file .h (serve anche a risolvere problemi di dipendenze per il linker anche se non è il caso di questo progetto).

Il codice nel main, o meglio nel file .ino, le riporterà così

```
extern const PROGMEM float weights_matrix[10][784];  
extern const PROGMEM float bias[10];
```

Successivamente eliminiamo tutte le funzioni di allenamento che non servono a nulla e riducono solo la quantità di memoria disponibile. Le variabili nel contesto di Arduino, ma nell'embedded in generale, sono quasi sempre utilizzate globali, per cui possiamo cancellare tutti i parametri passati alle funzioni perchè tali variabili sono sempre visibili e modificabili in tutto lo scope del programma. Questo sia perchè i programmi sono specifici e la possibilità di riutilizzo del codice è generalmente bassa, sia perchè semplifica la gestione delle variabili (ci permette di evitare tutti i puntatori utilizzati nel codice C) ma soprattutto perchè si evita di dover pushare elementi sullo stack (parametri formali delle funzioni) prima di invocare la CALL alla subroutine.

Ci serve poi un buffer di 784 float che conterrà l'immagine vettorizzata da fornire in input alla rete, un buffer per le uscite della rete sotto forma di probabilità ed un intero che conterrà la classe predetta. Vi fornisco già tutti e 3 e sono

```
float input_image[FLATTEN_IMAGE]; //immagine di input
float res[10]; //probabilità (dopo la Softmax)
int actual_label; //classe predetta
```

Dopo aver riempito il buffer "input\_image" con l'immagine desiderata basterà invocare

```
net_prediction();
```

che eseguirà il calcolo e scriverà la predizione della rete come intero nella variabile actual\_label.

Potete quindi ora usare la rete per riconoscere numeri in ogni campo applicativo, volutamente non ho forzato nessun meccanismo di input per lasciare a voi un programma di base completamente general purpose e personalizzabile.

## 16 Image generator - Script per generare immagini vettorizzate

Vi fornisco un programma, un semplice script in python, che accetta in input un immagine qualsiasi e vi restituisce l' opportuna codifica per il MNIST di tale immagine dopo averla ridimensionata al formato della rete, ovvero 28x28.

Il programma utilizza l'array argv e quindi accetta da linea di comando dei parametri dopo l'invocazione del programma per cui dovrete usarlo come segue

*#su windows da cmd*

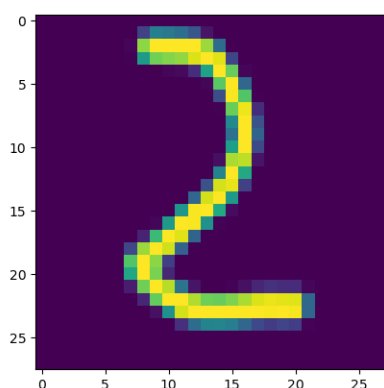
```
python generate_image.py images/image_0.png
```

*#su Linux / Mac / Windows su WSL*

```
python3 generate_image.py images/image_0.png
```

dove "images/image\_0.png" è il path dell' immagine a partire dalla cartella in cui si trova generate\_image.py.

Il programma oltre a fornirvi la codifica per la rete come vettore "copia-incollabile" sulla console vi mostra anche come graficamente l'immagine entra nella rete, per esempio un 2 scritto in nero su sfondo bianco diventa come segue



Mi raccomando, utilizzate python3 e non python 2.7 che oltre a essere deprecato e obsoleto ha alcuni seri problemi a livello interprete; chi ha Linux lo ha già di base nel sistema; chi usa Windows o Mac deve scaricarlo e installarlo.

## **17 Conclusioni**

Spero che questo progetto vi sia piaciuto e che possa servire per proseguire lo studio e lo sviluppo di sistemi basati su ANN per piattaforme embedded RT.

Ad ogni modo per qualsiasi dubbio potete contattarmi e sarò felice di potervi aiutare.