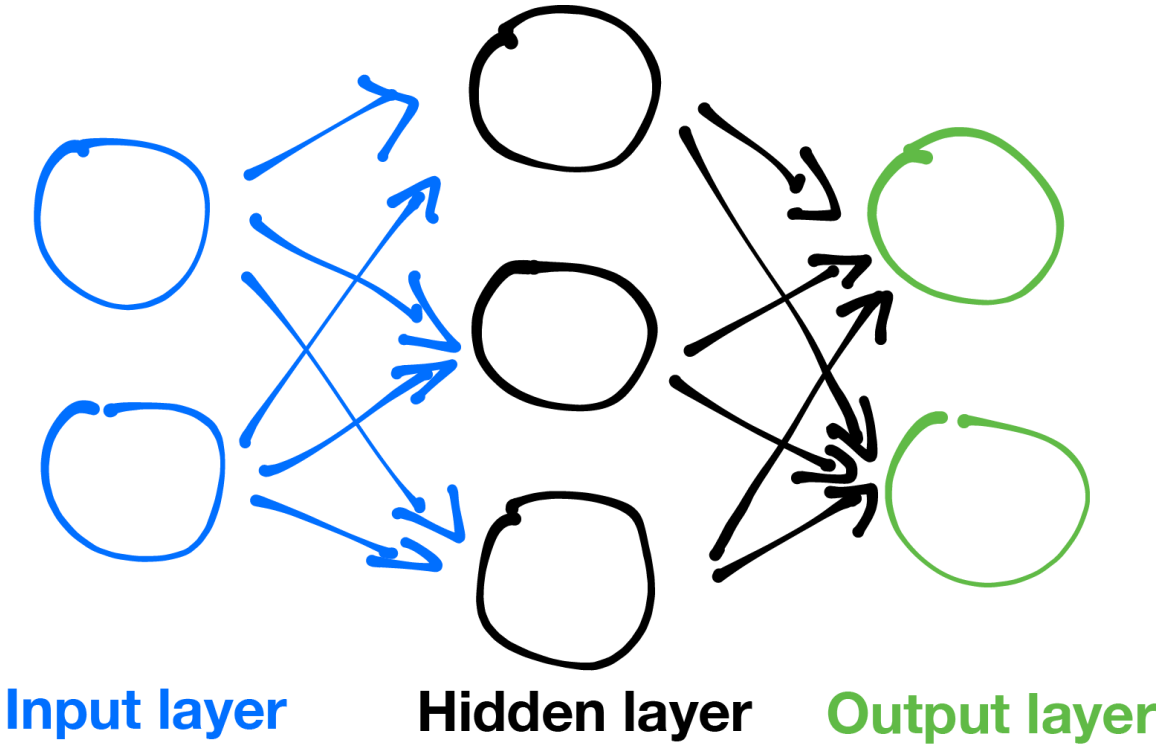


# Progetto di Machine Learning mod B

STUDIO DI UNA RETE NEURALE

Valerio Figliuolo | N97/214 | AA 2016/2017

# Implementazione della rete neurale

Per la implementazione di questo progetto, è stato utilizzato MATLAB/Octave come linguaggio di programmazione.

Si è cercato – per quanto possibile – di avere un approccio object-oriented alla stesura del codice. La rete neurale viene rappresentata da una struct chiamata `net`, che contiene tutti i parametri come pesi e funzioni che saranno poi passati alle varie funzioni, indipendenti fra loro, in modo tale da rendere il codice indipendente dalla rappresentazione dati scelta per la rete.

Unica restrizione del codice è quella di rappresentare i parametri "dipendenti" dai livelli (pesi, delta, etc.) come degli array di tipo 'cell', poiché il differente numero di neuroni per livello non ha reso possibile l'utilizzo di array multidimensionali.

## Creazione della rete

La rete viene creata tramite l'utilizzo della funzione

```
newNetwork(sizes, hnFunction, derHnFunction, outFunction, derOutFunction, eta)
```

che, presi in input parametri come le dimensioni (numero di livelli e di neuroni per livello, il che rende utilizzabile il pacchetto software anche per reti a più livelli), $\eta$ e le diverse funzioni di output, ritorna una rete neurale con pesi/bias casuali fra -1 e 1 (più precisamente, degli array cell di cui ogni elemento rappresenta un livello di pesi e bias) e, nel caso si voglia usare la tecnica rprop per l'aggiornamento dei pesi, i parametri $\Delta w_{ji}$ e $\Delta b_j$ già inizializzati a 0,1.

Di default sono incluse come funzione di attivazione (e rispettive derivate) sigmoide e funzione identità, ma l'approccio modulare del software permette l'utilizzo di qualsiasi funzione.

## Divisione del dataset

Per la creazione di {test,validation,data} set è stata usata la seguente politica: dato un intero `batchSize` (con $1 \leq$ batchSize $\leq$ max), divisibile per 4, la prima metà è usata come training set, mentre la seconda metà è divisa equamente fra validation set e test set.

La variabile `startTrainingSet` sarà il punto di partenza per la scelta delle immagini all'interno del dataset (default = 1)

# Fase di training

La fase di addestramento della nostra rete neurale si divide in quattro parti:

1. Propagazione in avanti (`feedForward.m`)
2. Calcolo dell'errore (`squaredError.m / crossEntropy.m`)
3. Calcolo delle derivate $\frac{\partial E}{\partial w_{ji}}$ e $\frac{\partial E}{\partial b_j}$ (`backPropagation.m`)
4. Aggiornamento dei pesi e dei bias (`gradientDescent.m / rprop.m`)

## PROPAGAZIONE IN AVANTI

Nella propagazione in avanti, ogni singolo neurone prende in input tutti gli output dei neuroni del livello precedente (moltiplicati per un peso $w_{ji}$), li somma, aggiung un valore di bias, utilizza il risultato come input per la funzione di attivazione e passa l'output di quest'ultima a tutti i neuroni del livello successivo, che applicheranno la stessa procedura, fino all'ultimo livello.

In termini matematici, per ogni neurone j della rete, il seguente calcolo va in scena:

$$a_j = f(z_j) \text{ , con } z_j = \sum_i (w_{ji} * a_i) + b_j$$

Dove $a_j$ è l'output del neurone, $f$ la funzione di attivazione del neurone, $w_{ji}$ i pesi delle connessioni dai neuroni i al neurone j, $b_j$ il bias del nodo e $a_i$ sono gli output dei neuroni del livello precedente.

### Implementazione

```
function [A, Z, output] = feedForward(x, W, B, activationF, outputF,
layers)
```

La funzione prende come parametro un vettore x di input (ovvero, i valori in scala di grigi dell'immagine che andiamo ad analizzare), un array cell di pesi, uno di bias, le funzioni di attivazioni di nodi interni e di output e le dimensioni della rete neurale. Come abbiamo già detto, la funzione è indipendente dalla rappresentazione scelta per la rete.

La funzione, come tutto il codice scritto in questo progetto, ha un approccio "matriciale": tratta ogni input come un vettore e ogni livello di pesi (e bias) come matrici. Un approccio quasi obbligato per garantire delle buone prestazioni.

Questo è il codice-fulcro della feed forward:

```
for i=2:layers-1
    Z{i} = W{i} * A{i-1} + repmat(B{i},1,size(A{i-1},2));
    A{i} = activationF(Z{i});
end
```

dove

- `W{}` è una matrice di pesi appartenente a $\mathbb{R}^{n*m}$, dove `n` è il numero di neuroni del livello attuale e `m` è il numero di neuroni del livello precedente
- `A{}` è il vettore degli output dei neuroni di un livello
- `activationF` è un function handler che rappresenta la funzione di attivazione data dall'utente al momento della creazione della rete
- `B{}` è il bias dell'attuale livello di neuroni.

Perché usare `repmat` per aggiungere il bias al calcolo? La risposta è semplice: il codice scritto per questa rete neurale può essere usato sia per un tipo di apprendimento online che per un tipo di apprendimento batch: nel primo caso sia l'input x per la funzione feedForward, sia il parametro A e il bias B saranno dei vettori monodimensionali; nel secondo caso saranno matrici e quindi sarà necessario duplicare i valori di bias per garantire la correttezza delle operazioni.

## CALCOLO DELL'ERRORE

Una volta calcolati l'output della propagazione in avanti, ci serve un metodo per misurare quanto l'output della rete diverge rispetto all'output desiderato, ovvero le etichette già fornite dal dataset. Una funzione di errore deve rispettare necessariamente alcune caratteristiche:

- Il suo valore deve essere piccolo quando output e valore desiderato sono molto vicini, e viceversa
- Deve essere positiva
- Deve essere derivabile

### Implementazione

```
[error, gradient] = errorFunction(output, trainingSetLabels);
```

Di default sono incluse due funzioni di errore: somma dei quadrati (`squaredError.m`) e cross-entropy (`crossEntropy.m`); entrambe prendono in input il risultato della feed forward e l'output desiderato e ritornano sia la valutazione della funzione di costo che il valore di $\frac{\partial E}{\partial y}$ (o *gradiente*), necessaria per la fase di backpropagation. Dato che il calcolo di $\frac{\partial E}{\partial y}$ dipende strettamente dalla funzione di errore scelta, per motivi di semplicità si è preferito includere entrambi i calcoli nella stessa funzione.

## PROPAGAZIONE ALL'INDIETRO

La teoria ci ricorda che l'output di una rete neurale può essere visto come una funzione nei parametri w e b (pesi e bias di tutta la rete). Possiamo quindi (come abbiamo fatto) calcolare l'errore E della rete, da cui poi ricavare le derivate parziali dell'errore in funzione di pesi e bias ($\frac{\partial E}{\partial w_{ji}}$ e $\frac{\partial E}{\partial b_j}$).

Calcolare tutte le derivate nella maniera classica, specialmente nel caso di reti con un elevato numero di parametri, è decisamente troppo dispendioso. Ed è qui che entra in gioco la *backpropagation*. Introduciamo un nuovo parametro, δ, che definiamo come la derivata parziale della funzione di errore rispetto agli input pesati dei neuroni:

$$\delta_j = \frac{\partial \mathrm{E}}{\partial z_j}$$

Purtroppo anche calcolare questo valore uno ad uno è troppo dispendioso. Ricordando che, con un po' di analisi matematica, possiamo scrivere

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_{ji}} \frac{\partial z_{ji}}{\partial w_{ji}}$$

e, inoltre, dato che

$$\mathrm{a}_j = \frac{\partial z_{ji}}{\partial w_{ji}}$$

possiamo infine dire che

$$\frac{\partial E}{\partial w_{ji}} = \delta_j \ \mathrm{a}_i$$

Il che significa che basterà calcolare i $\delta_j$ per ogni neurone per poter avere le rispettive derivate parziali.

I $\delta_j$ vengono calcolati in modo diverso in base al livello della rete neurale in cui ci troviamo. Per quanto riguarda il livello di output, troviamo che

$$\delta_j = f'(z_j) \frac{\partial E}{\partial y_j}$$

Mentre per i livelli interni

$$\delta_j = f'(z_j) \sum_k w_{kj} \, \delta_k$$

In parole povere, stiamo prendendo l'errore del livello di output (rispetto al valore di $z_j$) e lo stiamo portando "all'indietro" per tutta la rete (da qui l'uso del termine *backpropagation*)

Si dimostra inoltre che $\frac{\partial E}{\partial w_{ji}} = \delta_j \ \mathrm{a}_i$ e $\delta_j = \frac{\partial E}{\partial b_j}$.

## Implementazione

```
function [deltas, dW] = backPropagation(gradient, A, Z, W, layers,
derivativeO, derivativeH)
```

La funzione prende in input il gradiente, le varie matrici (A, Z, W), il numero di livelli della rete neurale e le derivate delle funzioni di output – implementando i calcoli descritti nella sezione precedente.

I $\delta_j$ saranno sommati nel caso di un approccio batch. Saranno ritornati tutti i $\frac{\partial E}{\partial w_{ji}}$ e i $\frac{\partial E}{\partial b_j}$, necessari per tutte e due le strategie di aggiornamento dei pesi che abbiamo implementato.

## DISCESA DEL GRADIENTE

Il tipo di aggiornamento più semplice è la cosiddetta *discesa del gradiente* – così chiamata perché utilizza appunto l'informazione data dalle derivate parziali per ridurre l'errore di valutazione sul training set.
Il valore di queste derivate determinerà di quanto saranno modificati pesi e bias per avvicinare i valori di output della rete al valore desiderato.

Più precisamente: la funzione di errore è, per l'appunto, una funzione. Una funzione che vogliamo minimizzare al fine di avere una classificazione più accurata dei nostri elementi. Considerando la funzione di errore nelle variabili $w_{ji}$ e $b_j$, le rispettive derivate parziali (*gradienti*) ci indicano la "direzione" da prendere per minimizzare il costo. Dobbiamo quindi modificare pesi e bias tenendo conto dei valori di $w_{ji}$ e b nel seguente modo:

| $w_{ji} = w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}$ | $b_j = b_j - \eta \frac{\partial E}{\partial \mathrm{b}_i}$ |
|---|---|

Perché $\eta$? I gradienti ci dicono appunto come dovrebbero cambiare i pesi e i bias per ridurre l'errore, però usarli così come sono ci porterebbe a grandi cambiamenti dei pesi/bias ad ogni iterazione (data l'eterogeneità del dataset). Dobbiamo quindi ridurre l'impatto dei gradienti delle singole epoche, per poter permettere alla nostra rete di generalizzare bene. Per questo usiamo $\eta$, detto *learning rate*, ovvero una costante moltiplicativa che ci permette di ammortizzare il contributo delle derivate.

### Implementazione

Implementiamo semplicemente i calcoli della tabella precedente.

## RPROP

Rprop è un algoritmo di aggiornamento dei pesi di una rete neurale adattivo. *Adattivo* nel senso che tiene conto dell'*andamento storico* della rete neurale per scegliere come aggiornare i pesi.

Scendendo più nel dettaglio, l'algoritmo non utilizza totalmente l'informazione sulla attuale derivata di pesi/bias; bensì memorizza i precedenti valori delle derivate, li moltiplica per gli attuali e ne considera il segno. L'informazione sul segno serve a sapere se la precedente modifica ha portato la rete a *"saltare"* un minimo locale (nel caso i segni delle due derivate fossero discordi) o se la rete sta procedendo nella giusta direzione (nel caso i segni siano concordi). Nel primo caso il valore di aggiornamento, $\Delta_{ij}^{(t)}$, deve essere decrementato moltiplicandolo per un parametro $\eta^-$, nel secondo caso deve essere aumentato moltiplicandolo per un parametro $\eta^+$, con $0 < \eta^- < 1 < \eta^+$; nel caso di un prodotto nullo, $\Delta_{ij}^{(t)}$ rimane uguale a quello del tempo t-1.

Dopodiché, se la derivata attuale è positiva (l'errore sta incrementando), $\Delta w_{ij}^{(t)} -= \Delta_{ij}^{(t)}$; se è negativa, $\Delta w_{ij}^{(t)} += \Delta_{ij}^{(t)}$. Se è nulla, $\Delta w_{ij}^{(t)} = 0$. Riassumiamo in pseudo codice cosa succede ad ogni epoca di training della rete neurale:

**For all weights and biases**{
  **if** $(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0)$ **then** {
    $\triangle_{ij}(t) =$ **minimum** $(\triangle_{ij}(t-1) * \eta^+, \triangle_{max})$
    $\triangle w_{ij}(t) = -$ **sign** $(\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$
    $w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}(t)$
  }
  **else if** $(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0)$ **then** {
    $\triangle_{ij}(t) =$ **maximum** $(\triangle_{ij}(t-1) * \eta^-, \triangle_{min})$
    $w_{ij}(t+1) = w_{ij}(t) - \triangle w_{ij}(t-1)$
    $\frac{\partial E}{\partial w_{ij}}(t) = 0$
  }
  **else if** $(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) = 0)$ **then** {
    $\triangle w_{ij}(t) = -$ **sign** $(\frac{\partial E}{\partial w_{ij}}(t)) * \triangle_{ij}(t)$
    $w_{ij}(t+1) = w_{ij}(t) + \triangle w_{ij}(t)$
  }
}

## Alcuni dettagli tecnici

Il paper originario consiglia di impostare $\eta^+ = 1{,}2$ e $\eta^- = 0{,}5$. Inoltre ci sono anche due parametri, $\Delta_{min} = 0{,}0001$ e $\Delta_{max} = 50$, usati per evitare cambiamenti eccessivi dei valori.

## IMPLEMENTAZIONE

```
function [delta_c, weights_c, derEW_c, diffW_c] = rprop(oldWeights_c,
oldDeltas_c, old_derWdE_c, derEW_c, oldDiffW_c, layers)
```

Anche qui tutti gli attributi relativi ai singoli pesi vengono rappresentati come array cell di matrici.

Dato che questo algoritmo chiede di fare dei calcoli per ogni singolo peso (a differenza degli algoritmi precedenti che potevano essere naturalmente scritti con un approccio matriciale), una implementazione naive avrebbe usato dei cicli for per ogni derivata parziale $\frac{\partial E}{\partial w_{ji}}$ e per ogni $\Delta_{ij}^{(t)}$. Purtroppo un simile approccio si è rivelato troppo lento per i nostri scopi.

Abbiamo così utilizzato la funzione `find()` di MATLAB, che ci permette di individuare gli indici degli elementi di una matrice che rispettano una determinata proprietà. Usando quindi queste tre istruzioni:

```
ind_pos = find(matrix > 0);
ind_neg = find(matrix < 0);
ind_zer = find(matrix == 0);
```

Possiamo ricavare gli indici degli elementi positivi/negativi/neutri della matrice che contiene il prodotto delle derivate e implementare l'algoritmo come se fosse un algoritmo per matrici.

## Fase di validazione & test

Una volta aggiornati i pesi (sia in modalità batch che online) viene calcolato l'errore sia sul training set che sul validation set. Abbiamo scelto di far finire la fase di training quando l'errore sul validation set (che, ricordiamo, è grande la metà del training) supera l'errore sul training set, ovvero quando la rete sta andando in overfitting.

Abbiamo infine calcolato precision, recall e numero di classificazioni corrette/sbagliate sul dataset.

## Dettagli del progetto

### TRACCIA

(Gr.2) Si consideri come input le immagine raw del dataset mnist dimezzandone opportunamente le dimensioni (ad esempio utilizzando in matlab la funzione imresize). Si
consideri un problema a 2 classi, scegliendo un unico digit come classe da riconosce, i restanti come non appartenenti alla classe. Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi, si studi l'apprendimento della rete neurale al variare delle dimensioni di training, validation e test set. Considerare un rete neurale con un solo strato di nodi interni. Scegliere e mantenere invariati tutti gli altri "parametri" come funzioni di output e parametri della Rprop.

Il dataset utilizzato è il MNIST (*Modified National Institute of Standards and Technology*), contenente 60000 immagini di caratteri scritti a mano e già classificati.

Per questo progetto si è scelto un problema di classificazione binaria, scegliendo un unico digit come classe da riconosce, i restanti come non appartenenti alla classe.

Abbiamo quindi deciso di usare come funzione di errore la cross-entropy (adatta a problemi di classificazione binaria e dalla veloce convergenza) e la rprop come politica di aggiornamento dei pesi.
Infine abbiamo analizzato l'andamento della rete in base alla grandezza del dataset preso in considerazione.

```
Stiamo usando un set di 200 elementi
Elapsed time is 0.016930 seconds.
Precision: 1.0000000000 // Recall: 0.2000000000
Riconosciuti: 46 // Non riconosciuti: 4
Epoche necessarie per il training = 5
------
Stiamo usando un set di 800 elementi
Elapsed time is 0.026402 seconds.
Precision: 1.0000000000 // Recall: 0.8750000000
Riconosciuti: 197 // Non riconosciuti: 3
Epoche necessarie per il training = 12
------
Stiamo usando un set di 2000 elementi
Elapsed time is 0.040864 seconds.
Precision: 0.9333333333 // Recall: 0.8936170213
Riconosciuti: 492 // Non riconosciuti: 8
Epoche necessarie per il training = 11
------
Stiamo usando un set di 4000 elementi
Elapsed time is 0.123318 seconds.
Precision: 0.9393939394 // Recall: 0.8942307692
Riconosciuti: 983 // Non riconosciuti: 17
Epoche necessarie per il training = 20
------
Stiamo usando un set di 8000 elementi
Elapsed time is 0.248902 seconds.
Precision: 0.9657142857 // Recall: 0.8366336634
Riconosciuti: 1961 // Non riconosciuti: 39
Epoche necessarie per il training = 24
------
Stiamo usando un set di 20000 elementi
Elapsed time is 0.950745 seconds.
Precision: 0.9515151515 // Recall: 0.9420000000
Riconosciuti: 4947 // Non riconosciuti: 53
Epoche necessarie per il training = 36
------
Stiamo usando un set di 40000 elementi
Elapsed time is 2.268170 seconds.
Precision: 0.9549098196 // Recall: 0.9361493124
```

```
Riconosciuti: 9890 // Non riconosciuti: 110
Epoche necessarie per il training = 43
```

Questi sono i valori ricavati da un run della rete neurale, rappresentativi del comportamento medio della rete. Si denota come come l'accoppiata rprop + cross-entropy permette una classificazione con alti valori di precision e recall (circa ≥ 90%) già con running time vicini al decimo di secondo.

Vediamo ora con una strategia online, discesa del gradiente, cross-entropy ed un $\eta = 0{,}2$:

```
Stiamo usando un set di 200 elementi
Elapsed time is 0.024206 seconds.
Precision: 0.7500000000 // Recall: 0.6000000000
Riconosciuti: 47 // Non riconosciuti: 3
Epoche necessarie per il training = 3
------
Stiamo usando un set di 800 elementi
Elapsed time is 0.263477 seconds.
Precision: 0.7857142857 // Recall: 0.9166666667
Riconosciuti: 192 // Non riconosciuti: 8
Epoche necessarie per il training = 8
------
Stiamo usando un set di 2000 elementi
Elapsed time is 0.478812 seconds.
Precision: 0.5657894737 // Recall: 0.9148936170
Riconosciuti: 463 // Non riconosciuti: 37
Epoche necessarie per il training = 6
------
Stiamo usando un set di 4000 elementi
Elapsed time is 0.782540 seconds.
Precision: 0.9142857143 // Recall: 0.9230769231
Riconosciuti: 983 // Non riconosciuti: 17
Epoche necessarie per il training = 5
------
Stiamo usando un set di 8000 elementi
Elapsed time is 1.571150 seconds.
Precision: 0.9877300613 // Recall: 0.7970297030
Riconosciuti: 1957 // Non riconosciuti: 43
Epoche necessarie per il training = 5
------
Stiamo usando un set di 20000 elementi
Elapsed time is 8.504947 seconds.
Precision: 0.9849785408 // Recall: 0.9180000000
Riconosciuti: 4952 // Non riconosciuti: 48
Epoche necessarie per il training = 10
------
Stiamo usando un set di 40000 elementi
Elapsed time is 18.565135 seconds.
Precision: 0.9373219373 // Recall: 0.9695481336
Riconosciuti: 9903 // Non riconosciuti: 97
Epoche necessarie per il training = 5
```

Anche qui riusciamo ad avere risultati soddisfacenti ma il running time è molto superiore.

Comportamento simile anche per l'approccio batch (eta = 0.001), che però non mantiene buone performance su dataset molto grandi (ma tiene bene, anche se con tempi di esecuzione molto superiori rispetto ad un approccio rprop, su dataset più piccoli):

```
Stiamo usando un set di 200 elementi
Elapsed time is 0.391920 seconds.
Precision: 1.0000000000 // Recall: 0.6000000000
Riconosciuti: 48 // Non riconosciuti: 2
Epoche necessarie per il training = 659
------
Stiamo usando un set di 800 elementi
Elapsed time is 1.040562 seconds.
Precision: 0.9090909091 // Recall: 0.8333333333
Riconosciuti: 194 // Non riconosciuti: 6
Epoche necessarie per il training = 782
------
Stiamo usando un set di 2000 elementi
Elapsed time is 0.976474 seconds.
Precision: 0.9545454545 // Recall: 0.8936170213
Riconosciuti: 493 // Non riconosciuti: 7
Epoche necessarie per il training = 344
------
Stiamo usando un set di 4000 elementi
Elapsed time is 2.788180 seconds.
Precision: 0.9029126214 // Recall: 0.8942307692
Riconosciuti: 979 // Non riconosciuti: 21
Epoche necessarie per il training = 537
------
Stiamo usando un set di 8000 elementi
Elapsed time is 7.566468 seconds.
Precision: 0.9941520468 // Recall: 0.8415841584
Riconosciuti: 1967 // Non riconosciuti: 33
Epoche necessarie per il training = 776
------
Stiamo usando un set di 20000 elementi
Elapsed time is 19.901950 seconds.
Precision: 0.9592668024 // Recall: 0.9420000000
Riconosciuti: 4951 // Non riconosciuti: 49
Epoche necessarie per il training = 703
------
Stiamo usando un set di 40000 elementi
Elapsed time is 0.478254 seconds.
Precision: 0.1018000000 // Recall: 1.0000000000
Riconosciuti: 1018 // Non riconosciuti: 8982
Epoche necessarie per il training = 6
```

Ne deriviamo che – come affermato dal paper che introduce la tecnica rprop – quest'ultima è effettivamente molto più veloce della classica backpropagation mantenendo ottime performance e risultati.