Introduzione

Con questa guida il mio obiettivo è quello di presentare il tema delle reti neurali in modo semplice per coloro che hanno almeno delle basi universitarie dei concetti di Analisi 1 e 2 per il calcolo delle derivate parziali e di Algebra lineare per il calcolo matriciale. La guida è divisa in due fasi, la prima comprende la comprensione dell'argomento in termini matematico-logici e la seconda comprende la scrittura di un codice in Python che implementi quanto enunciato.

Reti neurali

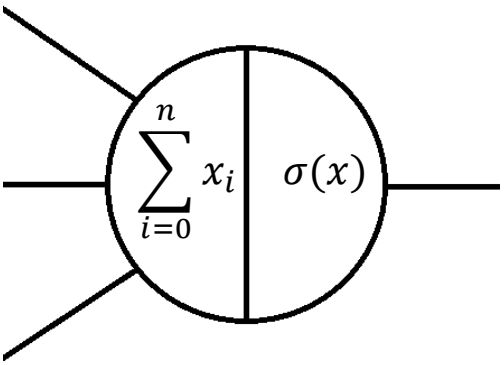
Incominciamo dalle basi, ovvero che cos'è una rete neurale artificiale? Partendo dal modello di rete neurale biologica, una rete neurale artificiale altro non è che uno schema logico secondo il quale passano informazioni. In termini matematici, è una funzione che trasforma un vettore in input in un vettore in output.

$$y = f(x) \quad con\ x \in \mathbb{R}^n\ e\ y \in \mathbb{R}^m$$

Consideriamo x il vettore di input e y il vettore di output in spazi multidimensionali, e la funzione f la nostra rete neurale, tale che $f: \mathbb{R}^n \to \mathbb{R}^m$. Questo è quello che avviene a livello macroscopico della nostra rete. Focalizziamoci invece ora sull'elemento fondamentale che la costituisce: il percettrone, ovvero il neurone artificiale, corrispondente di quello biologico.

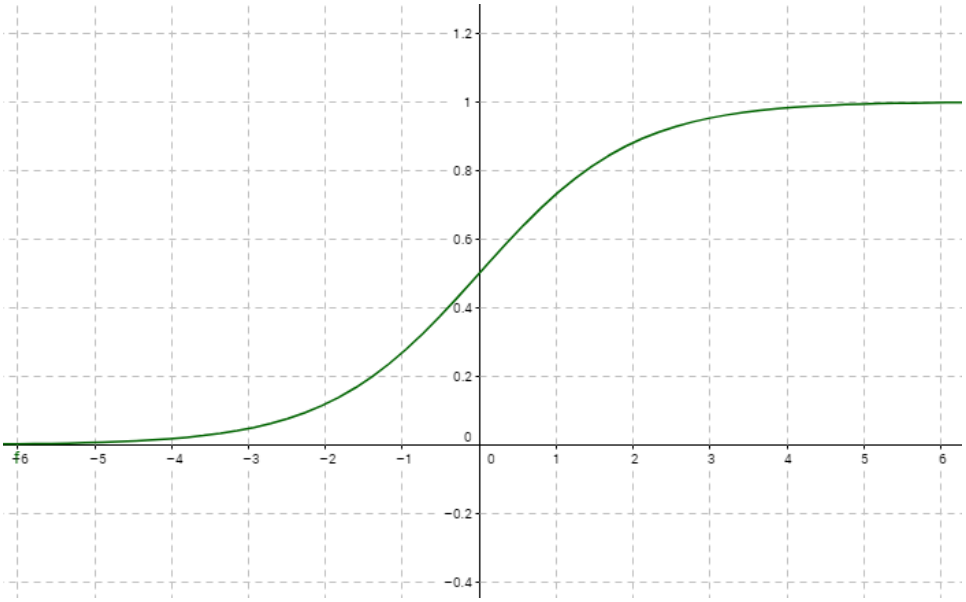
Una rete di percettroni

Per immaginare un percettrone possiamo prendere in esempio sempre il modello biologico del neurone. Ogni neurone è costituito essenzialmente da tre parti: dentriti, corpo cellulare e assone. I dentriti hanno il compito di ottenere le informazioni dai neuroni che lo precedono, per poi propagarla verso altri attraverso l'assone. Il percettrone si basa sostanzialmente sugli stessi aspetti di base.



Il funzionamento è il seguente: il nucleo del percettrone riceve gli input dagli assoni dei percettroni che lo precedono, li somma tra loro e al risultato ottenuto applica una funzione $\sigma(x)$, chiamata "funzione attivatrice", che normalizza i risultati ottenuti in un range definito. Nel nostro caso prenderemo in considerazione come funzione attivatrice, la *funzione sigmoidea*, in quanto è una funzione derivabile in tutto l'intervallo $[-\infty, +\infty]$. Andamento della funzione sigmoidea in figura.

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

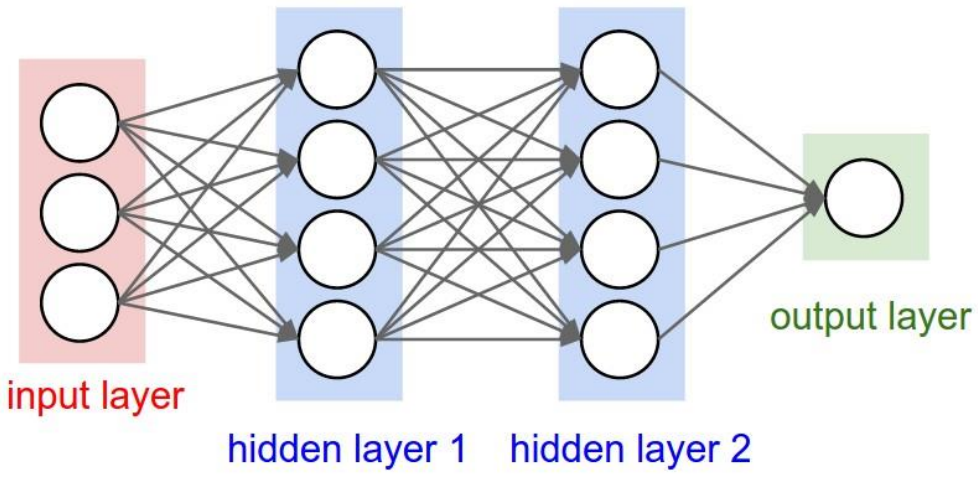


Weights e biases

Ogni percettrone riceve input da quelli che lo precedono. Gli input non arrivano con la massima intensità, ma sono ridotti di un certo fattore che va da 0 (valore minimo) a 1 (valore massimo). Questo fattore prende il nome di weight, ovvero “peso”. Per cui, ad ogni collegamento che vedremo all’interno di una rete neurale andremo a considerare un weight ad esso associato. Mentre i biases sono degli input supplementari che ogni percettrone possiede. Per definizione il bias ha sempre valore 1 e nel momento in cui viene aggiunto alla sommatoria degli input deve essere moltiplicato per il suo weight corrispondente (ad ogni collegamento corrisponde un weight). Nel nostro caso non andremo a considerare i bias, sia nei calcoli sia nella rete, ma si può applicare lo stesso metodo tenendo presente dei biases.

Una struttura a layer

Organizzando insieme più percettroni ottieniamo i diversi layer che compongono una rete neurale: input layer, che contiene il vettore di input; hidden layers, che contengono uno svariato numero di percettroni; output layer, che contiene il vettore di output.

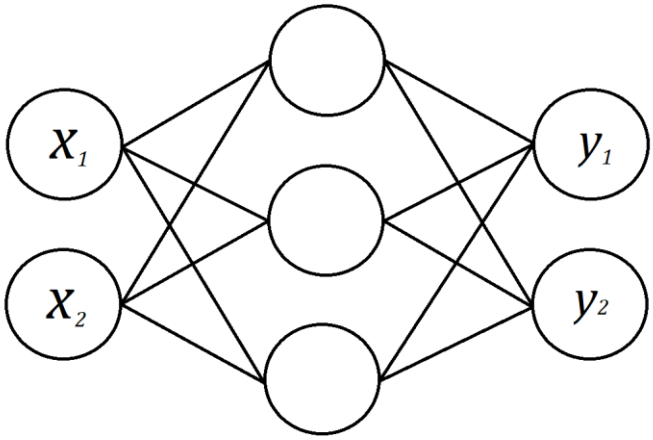


Forward-Propagation

Il Forward-Propagation è la tecnica che utilizza una rete neurale per far passare il vettore di input attraverso tutta la rete fino ad arrivare al vettore di output. Sull'input vengono eseguiti calcoli come prodotti, sommatorie e funzioni attivatrici in modo concatenato prima di arrivare al risultato ultimo.

Matrici e algebra lineare

Per il nostro esempio, prenderemo in considerazione una rete costituita da un input layer di due neuroni, un'hidden layer di tre neuroni e un output layer di altri due neuroni.



Scriviamo il vettore di input e quello di output come segue:

$$x = [x_1,\ x_2] \qquad\qquad y = [y_1,\ y_2]$$

Appartenenti entrambi ad $\mathbb{R}^2$

Mentre scriviamo il vettore dell'output dell'hidden layer come:

$$h = [h_1,\ h_2,\ h_3]$$

A questo punto, in base a tutti i collegamenti tra i percettroni che andremo a tracciare nel grafico avremo un certo numero di weights, in questo caso 12. Inoltre possiamo dividere, per comodità, i weights tra l'input layer e l'hidden layer e tra l'hidden layer e l'output layer. Possiamo riassumere questi due gruppi in due matrici:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \qquad W' = \begin{bmatrix} w'_{11} & w'_{12} \\ w'_{21} & w'_{22} \\ w'_{31} & w'_{32} \end{bmatrix}$$

Dove il generico elemento w_{ij} è il weight che collega l'i-esimo percettrone del layer precedente al j-esimo percettrone del layer successivo. A questo punto per far avanzare il segnale di input fino all'hidden layer, dobbiamo far passare il segnale attraverso la matrice di weights e poiché i segnali che arrivano ad un neurone dell'hidden layer devono essere sommati, l'operazione che andremo ad applicare ai nostri vettori e matrici

sarà un prodotto scalare. Come ultimo step andremo ad applicare la funzione sigmoidea a tutti i valori del vettore che otterremo.

$$x \cdot W = [x_1,\ x_2] \cdot \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$= [x_1 w_{11} + x_2 w_{21} \quad x_1 w_{12} + x_2 w_{22} \quad x_1 w_{13} + x_2 w_{23}]$$

Da un vettore 1x2 e una matrice 2x3 otteniamo infatti un vettore 1x3, ovvero il vettore che contiene le sommatorie degli input, moltiplicati per i rispettivi weights, di ogni singolo neurone dell'hidden layer. A questo punto applichiamo la funzione sigmoidea al vettore ottenuto.

$$\sigma([x_1 w_{11} + x_2 w_{21} \quad x_1 w_{12} + x_2 w_{22} \quad x_1 w_{13} + x_2 w_{23}]) =$$

$$[\sigma(x_1 w_{11} + x_2 w_{21}) \quad \sigma(x_1 w_{12} + x_2 w_{22}) \quad \sigma(x_1 w_{13} + x_2 w_{23})]$$

Abbiamo ottenuto l'output dei neuroni dell'hidden layer. Questo significa che l'output di un layer si ottiene sempre applicando una composizione di funzioni $\sigma(x \cdot W)$, dove x è il vettore di input, W la matrice dei weights, e σ la funzione attivatrice.

Ripetendo l'operazione per tutti gli hidden layer (nel caso ce ne fossero più di uno) e anche per l'output layer al termine del processo di Forward-Propagation otterremo il vettore di output generato da una funzione che è composizione di funzioni come prodotti scalari e sigmoidee.

L'errore e l'allenamento della rete

Nel momento di inizializzazione della rete i weights vengono generali in modo pseudo-randomico con valori decimali compresi tra 0 e 1. È abbastanza ovvio che la rete, per essere efficiente e generare vettori in output che rispecchiano un certo target, deve aggiustare sempre più i weights di cui è costituita, secondo il metodo della *discesa del gradiente*.

Ciò che andremo a fare analiticamente sarà:

- Calcolare di quanto si discosta l'output della rete dal valore di target, dato un certo input
- Calcolare l'errore totale rispetto ad ogni singolo weight, cioè quanto ogni weight partecipa all'errore totale della rete

Per fare ciò, ci affidiamo alle ***derivate parziali*** e al ***metodo di catena***.

Backward-Propagation

Come prima cosa, dato un certo input, calcoliamo di quanto si discosta il valore dell'output da quello desiderato (target).

Nel nostro caso, $y \in \mathbb{R}^2$ e quindi anche il target, che indichiamo con $y^* \in \mathbb{R}^2$

Considerando l'errore totale come la somma dei singoli errori del vettore di output, possiamo scrivere:

$$E_{TOT} = E_1 + E_2$$

$$con\ E_1 = \frac{1}{2}(y_1^* - y_1)^2 \quad e \quad E_2 = \frac{1}{2}(y_2^* - y_2)^2$$

Il fattore ½ verrà semplificato con il 2 nel momento in cui andremo a derivare l'errore, quindi è una componente di cui non dobbiamo preoccuparci.

Partendo a ritroso, deriviamo l'errore totale rispetto il weight che collega il primo percettrone dell'output layer con il primo percettrone dell'hidden layer. Facciamo questo per determinare di quanto w'_{11} abbia contribuito all'errore totale.

$$\frac{\partial E_{TOT}}{\partial w'_{11}} = \frac{\partial E_1}{\partial w'_{11}} + \frac{\partial E_2}{\partial w'_{11}} = \frac{\partial E_1}{\partial w'_{11}} + 0 = \frac{\partial E_1}{\partial w'_{11}}$$

Perché l'errore del secondo elemento di y^* non dipende da w'_{11}. Proseguiamo con i calcoli:

$$\frac{\partial E_1}{\partial w'_{11}} = \frac{\partial E_1}{\partial y_1} \cdot \frac{\partial y_1}{\partial \varphi} \cdot \frac{\partial \varphi}{\partial w'_{11}} \quad con\ \varphi = \sum_i h_i w'_{i1}$$

Applicando la ***regola di catena*** risulta evidente che la derivata parziale dell'errore rispetto il weight preso in considerazione è uguale alla derivata parziale dell'errore rispetto il suo output moltiplicata per la derivata parziale dell'output rispetto la somma di tutti i segnali in ingresso nell'ultimo percettrone moltiplicata per derivata parziale dell'input rispetto il weight considerato.

Calcoliamo le singole derivate parziali e successivamente mettiamole insieme.

$$\frac{\partial E_1}{\partial y_1} = -(y_1^* - y_1)$$

$$\frac{\partial y_1}{\partial \varphi} = \sigma'(y_1) = y_1(1 - y_1)$$

$$\frac{\partial \varphi}{\partial w'_{11}} = h_1$$

$$\frac{\partial E_{TOT}}{\partial w'_{11}} = -(y_1^* - y_1)\ y_1(1 - y_1)\ h_1$$

Quindi:

$$\frac{\partial E_{TOT}}{\partial w'_{11}} = -\delta(y_1)h_1 \quad con\ \delta(y_1) = (y_1^* - y_1)\ y_1(1 - y_1)$$

Abbiamo così ottenuto la **regola delta**.

Non ci resta che aggiornare il nostro weight utilizzando il valore appena ottenuto:

$$w'_{11} = w'_{11} + \Delta w'_{11} \quad con\ \Delta w'_{11} = l_r \cdot \delta(y_1)h_1$$

Dove l_r è il *learning rate* della rete. Più il learning rate è basso più la rete apprenderà in modo lento ma preciso. Contrariamente più il learning rate è alto, più la rete impara ad ogni ciclo ma con un'approssimazione maggiore.

Il processo va ripetuto per tutti i weights nella rete prima di processare nuovamente le informazioni in un nuovo ciclo.

Con questa formula è possibile aggiornare solo i weights che si trovano tra l'output layer e l'hidden layer, ma non quelli intermedi tra gli altri layer. Per fa ciò abbiamo bisogno di fare altre considerazioni, in quanto gli altri weights partecipano anche all'errore in y_2 che finora abbiamo ovviamente considerato.

Allo stesso modo, cerchiamo di calcolare la variazione del weight che collega il primo valore in input x_2 con il primo percettrone dell'hidden layer.

$$\frac{\partial E_{TOT}}{\partial w_{11}} = \frac{\partial E_1}{\partial w_{11}} + \frac{\partial E_2}{\partial w_{11}}$$

$$\frac{\partial E_1}{\partial w_{11}} = \frac{\partial E_1}{\partial y_1} \cdot \frac{\partial y_1}{\partial \varphi_1} \cdot \frac{\partial \varphi_1}{\partial h_1} \cdot \frac{\partial h_1}{\partial \varphi_2} \cdot \frac{\partial \varphi_2}{\partial w_{11}}$$

$$con\ \varphi_1 = \sum_i h_i w'_{i1} \quad e\ \varphi_2 = \sum_j x_j w_{j1}$$

Calcoliamo le derivate, ad esclusione delle prime due in quanto già calcolate in precedenza:

$$\frac{\partial \varphi_1}{\partial h_1} = w'_{11}$$

$$\frac{\partial h_1}{\partial \varphi_2} = h_1(1 - h_1)$$

$$\frac{\partial \varphi_2}{\partial w_{11}} = x_1$$

Facciamo lo stesso con E_2:

$$\frac{\partial E_2}{\partial w_{11}} = \frac{\partial E_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial \varphi_1} \cdot \frac{\partial \varphi_1}{\partial h_1} \cdot \frac{\partial h_1}{\partial \varphi_2} \cdot \frac{\partial \varphi_2}{\partial w_{11}}$$

$$con\ \varphi_1 = \sum_i h_i w'_{i2} \quad e\ \varphi_2 = \sum_j x_j w_{j2}$$

$$\frac{\partial E_2}{\partial y_2} = -(y_2^* - y_2)$$

$$\frac{\partial y_2}{\partial \varphi_1} = y_2(1 - y_2)$$

$$\frac{\partial \varphi_1}{\partial h_1} = w'_{12}$$

$$\frac{\partial h_1}{\partial \varphi_2} = h_1(1 - h_1)$$

$$\frac{\partial \varphi_2}{\partial w_{11}} = x_1$$

Mettendo insieme le derivate parziali calcolate sia ora e sia in precedenza:

$$\frac{\partial E_{TOT}}{\partial w_{11}} = -(y_1^* - y_1)\, y_1(1 - y_1) w'_{11} h_1(1 - h_1) x_1 - (y_2^* - y_2)\, y_2(1 - y_2)\, w'_{12} h_1(1 - h_1)\, x_1$$

$$\frac{\partial E_{TOT}}{\partial w_{11}} = -(\delta(y_1)\, w'_{11} + \delta(y_2)\, w'_{12})\, h_1(1 - h_1)\, x_1$$

$$\frac{\partial E_{TOT}}{\partial w_{11}} = -\underbrace{\sum_i (\delta(y_i)\, w'_{1i})\, h_1(1 - h_1)}_{\delta'(h_1)}\, x_1$$

$$\frac{\partial E_{TOT}}{\partial w_{11}} = -\delta'(h_1)\, x_1$$

Con il valore ottenuto possiamo aggiustare il weight w_{11}:

$$w_{11} = w_{11} + \Delta w_{11} \quad con\ \ \Delta w_{11} = l_r \cdot \delta'(h_1) x_1$$

Questo ci fa capire come la variazione dei i-esimo peso dipende unicamente dall'errore del percettrone nel layer successivo e dal valore di input del percettrone nel layer precedente che esso collega. A livello pratico l'errore di ogni percettrone può essere calcolato propagando all'indietro l'errore sull'output layer per ogni rispettivo weight, sommandoli tra di loro. Questo lo si può evincere dalla formula ricavata.

NOTA: I rispettivi weight dei biases si calcolano esattamente allo stesso modo, tenendo presente come input il valore costante 1, e come errore l'errore del percettrone al quale sono collegati.

Implementazione in Python

Il linguaggio che andremo ad utilizzare per implementare i concetti appena enunciati sarà il Python, utilizzando una comodissima libreria per svolgere velocemente calcoli tra matrici: "Numpy". Trovate la vostra versione ed installatela utilizzando il comando di "pip". Nel mio caso, poiché utilizzo Python3.6, la libreria ufficiale non ancora è stata rilasciata per questa versione, pertanto mi sono affidato a release in beta che svolgono egregiamente il loro compito. Per chi avesse difficoltà a trovare la libreria per Python3.6 può scaricarla direttamente da qui, e poi installarla con "pip install numpy-1.13.0+mkl-cp36-cp36m-win32.whl".

Spiegazione del codice

Come prima cosa importiamo la libreria Numpy e dichiariamo le variabili epochs e learning_rate, dove epochs sono le iterazioni durante le quali la rete perfezionerà i suoi weights e il secondo è il tasso di apprendimento.

```
import numpy as np

# Training VARs
epochs = 10000
learning_rate = .05
```

Subito dopo dichiariamo la struttura della rete come segue. La struttura mostra la configurazione di ogni layer compreso i neuroni che la compongono.

```
# Network Structure - INPUT / HIDDEN / OUTPUT
input = 2
hidden = [3]
output = 2

structure = np.array([[input], hidden, [output]])
```

Fatto ciò impostiamo l'array che conterrà tutte le matrici dei weights inizializzandole con valori casuali compresi tra 0 e 1. Ci saranno tante matrici quanti saranno gli spazi tra ogni layer.

```
# Set weights matrices
weights = []
layers = np.concatenate(structure);
for i in range(0, len(layers) - 1):
	weights.append(np.random.rand(layers[i], layers[i + 1]))
```

Creiamo 2 array, uno che conterrà gli errori dei percettroni mentre l'altro il loro output.

```
# Set errors array
errors = []
# Set outputs array
outputs = []
```

Ora implementiamo tutti i metodi necessari.

L'**execute_forward** propaga il segnale in avanti nella nostra rete utilizzando chiamate ricorsive per ogni layer.

```
def execute_forward(input, weights_matrix_index = 0):
	global outputs
	if weights_matrix_index > 0:
		input = sigmoid(input)
	outputs.append(input)
	if weights_matrix_index == len(weights):
		return input
	else:
		sum_of_previous_layer = np.dot(input, weights[weights_matrix_index])
		return execute_forward(sum_of_previous_layer, weights_matrix_index + 1)
```

La **funzione sigmoidea** è una funzione matematica derivabile che funge da attivatore del percettrone.

```
def sigmoid(x):
	return 1 / (1 + np.exp(-x))
```

Il **train_network** si occupa di allenare la rete facendo passare ad ogni ciclo (epoca) i vettori appartenenti al dataset

```
def train_network(input_dataset, target_dataset, epochs, learning_rate):
	global errors
	for epoc in range(0, epochs):
		for data_row in range(0, input_dataset.shape[0]):
			train_network_main(input_dataset[data_row,:],
target_dataset[data_row,:], learning_rate)
```

Il **train_network_main** è il metodo che allena la rete in una singola epoca e confrontando gli output dà il via al processo di propagazione all'indietro dell'errore e al successivo allenamento dei weights.

```
def train_network_main(input, target, learning_rate):
	global errors, outputs
	output = execute_forward(input)
	backpropagate_error(np.multiply(np.multiply(target - output, output), 1 - output).T,
len(weights) - 1)
	update_weights(learning_rate)
	errors = []
	outputs = []
```

Il **backpropagate_error** è il metodo che propaga indietro l'errore partendo quindi dall'ultimo layer (output) fino al primo hidden layer. Anch'esso è un metodo ricorsivo.

```
def backpropagate_error(error, weights_matrix_index):
	global errors, outputs
	errors.insert(0, error)
	if weights_matrix_index > 0:
		transported_error =
np.multiply(np.multiply(np.dot(weights[weights_matrix_index], error),
outputs[weights_matrix_index].T), 1 - outputs[weights_matrix_index].T)
		backpropagate_error(transported_error, weights_matrix_index - 1)
```

Infine **update_weights** aggiorna semplicemente i weights basandosi sugli errori e gli output dei neuroni.

```
def update_weights(learning_rate):
	global errors, outputs
	for i in range(0, len(weights)):
		weights[i] += learning_rate * (errors[i] * outputs[i]).T
```

A questo punto i dataset attraverso i quali andremo ad allenare la rete saranno molto semplici. Nel mio caso allenerò la rete per fornire l'inverso del vettore di input che andrò a fornirgli.

```
# Datasets
input_dataset = np.matrix([

		[1, 0],
		[0, 1],

	])
target_dataset = np.matrix([

		[0, 1],
		[1, 0],
	])
```

Sperimentiamo il codice

Per allenare la rete basta chiamare il metodo:

```
# Train network
train_network(input_dataset, target_dataset, epochs, learning_rate)
```

E per visualizzare l'output desiderato:

```
# Test network
print(execute_forward(np.array([0, 0])))
# Expected output is [ .5 .5 ] - because it is a mid of [ 0 1 ] and [ 1 0 ]
```

Codice completo

```
import numpy as np

# Training VARs
epochs = 10000
learning_rate = .05

# Network Structure - INPUT / HIDDEN / OUTPUT
input = 2
hidden = [3]
output = 2

structure = np.array([[input], hidden, [output]])

# Set weights matrices
weights = []
layers = np.concatenate(structure);
for i in range(0, len(layers) - 1):
	weights.append(np.random.rand(layers[i], layers[i + 1]))

# Set errors array
errors = []
# Set outputs array
outputs = []

# Methods
def execute_forward(input, weights_matrix_index = 0):
	global outputs
	if weights_matrix_index > 0:
		input = sigmoid(input)
	outputs.append(input)
	if weights_matrix_index == len(weights):
		return input
	else:
		sum_of_previous_layer = np.dot(input, weights[weights_matrix_index])
		return execute_forward(sum_of_previous_layer, weights_matrix_index + 1)

def sigmoid(x):
	return 1 / (1 + np.exp(-x))

def train_network(input_dataset, target_dataset, epochs, learning_rate):
	global errors
	for epoc in range(0, epochs):
		for data_row in range(0, input_dataset.shape[0]):
			train_network_main(input_dataset[data_row,:],
target_dataset[data_row,:], learning_rate)
```

```
def train_network_main(input, target, learning_rate):
	global errors, outputs
	output = execute_forward(input)
	backpropagate_error(np.multiply(np.multiply(target - output, output), 1 -
output).T, len(weights) - 1)
	update_weights(learning_rate)
	errors = []
	outputs = []

def backpropagate_error(error, weights_matrix_index):
	global errors, outputs
	errors.insert(0, error)
	if weights_matrix_index > 0:
		transported_error =
np.multiply(np.multiply(np.dot(weights[weights_matrix_index], error),
outputs[weights_matrix_index].T), 1 - outputs[weights_matrix_index].T)
		backpropagate_error(transported_error, weights_matrix_index - 1)

def update_weights(learning_rate):
	global errors, outputs
	for i in range(0, len(weights)):
		weights[i] += learning_rate * (errors[i] * outputs[i]).T

# Datasets
input_dataset = np.matrix([

		[1, 0],
		[0, 1],

	])

target_dataset = np.matrix([

		[0, 1],
		[1, 0],

	])

# Train network
train_network(input_dataset, target_dataset, epochs, learning_rate)

# Test network
print(execute_forward(np.array([0, 0])))
# Expected output is [ .5 .5 ] - because it is a mid of [ 0 1 ] and [ 1 0 ]
```

Conclusioni

Spero che questa guida sia stata di vostro aiuto nella comprensione di come una rete neurale funzioni. Per qualsiasi dubbio o chiarimento su qualsiasi concetto spiegato mandate una mail all'amministratore di Project Information e saremo lieti di aiutarvi.