Istituto Istruzione Superiore Italo Calvino

ESAMI DI STATO 2017

Enrico Fiasché

Classe V B Elettronica Automazione

Anno Scolastico: 2016/2017

MACHINE LEARNING

Sommario

Intelligenza artificiale

Noi umani, per centinaia di anni, abbiamo cercato di capire come il nostro cervello e la nostra intelligenza lavorassero e abbiamo cercato, nel corso degli anni, di replicarli in una sorta di macchina.

Con il termine *intelligenza artificiale* s'intende generalmente l'abilità di un computer di svolgere funzioni e ragionamenti tipici della mente umana. Nel suo aspetto puramente informatico, essa comprende la teoria e le tecniche per lo sviluppo di algoritmi che consentano alle macchine di mostrare un'abilità intelligente, cioè la capacità di estrapolare da conoscenze precedenti delle linee guida da utilizzare per risolvere nuovi problemi che il calcolatore non ha mai affrontato sebbene possa averne affrontati di simili in passato. Non esiste ancora una definizione universale perché l'AI è un settore estremamente recente e in fortissima evoluzione.

L'intelligenza artificiale nasce negli anni '50 negli USA come disciplina scientifica. Le basi dell'intelligenza artificiale sono poste nel 1950 con la pubblicazione dell'articolo di Alan Turing, in cui il matematico inglese ipotizza la possibilità di realizzare una macchina intelligente e un apposito test (il Test di Turing) per individuarla. La possibilità di un elaboratore di simulare il processo di apprendimento e di ragionamento dell'uomo viene affrontato per la prima volta nel 1956 nel corso della conferenza al College di Hannover. Durante il convegno viene coniata per la prima volta l'espressione *Artificial Intelligence* (A.I.).

I primi studi dell'intelligenza artificiale si concentrano sui giochi e nella dimostrazione dei teoremi matematici. Una delle prime applicazioni sono il gioco di dama e degli scacchi. L'elaboratore apprende le mosse migliori dall'esperienza delle partite precedenti e migliora la propria performance di gioco. Deep Blue è stato il primo calcolatore a vincere una partita di scacchi contro il campione del mondo in carica nel 1997.
Oggi l'intelligenza artificiale viene impiegata in un'ampia varietà di campi ed applicazioni come la medicina, il mercato azionario, la robotica e la ricerca scientifica. L'intelligenza artificiale è largamente utilizzata per la realizzazione di assistenti automatici online principalmente nelle compagnie telefoniche. Anche nell'ambito dei trasporti il suo utilizzo sta aumentando rapidamente, le automobili a guida autonoma sviluppate da Google e Tesla fanno largamente uso di tecniche di intelligenza artificiale.

Alla base di queste intelligenze artificiali è presente una rete neurale artificiale che permette di risolvere questi problemi.

Nel programma esposto in questa tesina si vuole munire una macchina di intelligenza artificiale in modo da essere in grado di riconoscere numeri scritti a mano libera.

Rete Neurale

Il termine rete neurale viene utilizzato come riferimento a una rete composta da neuroni.

In molti organismi viventi sono presenti complesse organizzazioni di cellule nervose, con compiti di riconoscimento delle configurazioni assunte dall'ambiente esterno. Al fine di compiere tali operazioni, le reti biologiche si servono di un numero imponente di neuroni fittamente interconnessi in modo da variare la loro configurazione in risposta agli stimoli esterni.

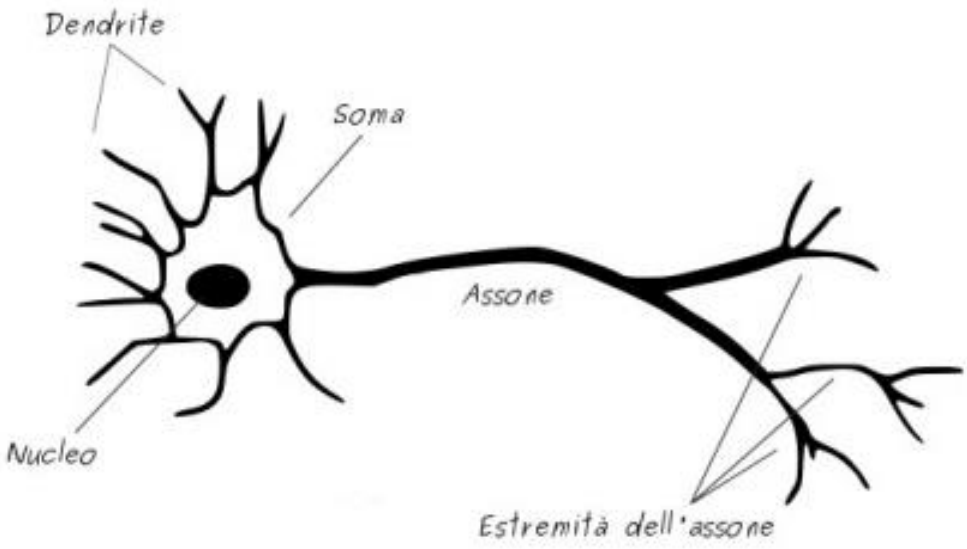


Figura 1: Struttura di un neurone biologico

Generalmente il neurone è costituito di 3 parti principali:

- Il *soma*: Corpo cellulare
- L'*assone*: linea di uscita del neurone unica ma che si dirama in migliaia di rami
- Il *dendrite*: linea di entrata del neurone che riceve segnali in ingresso da altri assoni tramite le sinapsi (struttura che consente la comunicazione dei neuroni tra loro)

I neuroni non reagiscono prontamente agli stimoli esterni, ma sopprimono l'input finché non raggiunge un valore relativamente alto da innescare l'uscita.
Il neurone, come si può vedere in figura 2, prima di fornire un valore alto di output deve ricevere in ingresso un valore che superi il livello di soglia (threshold).

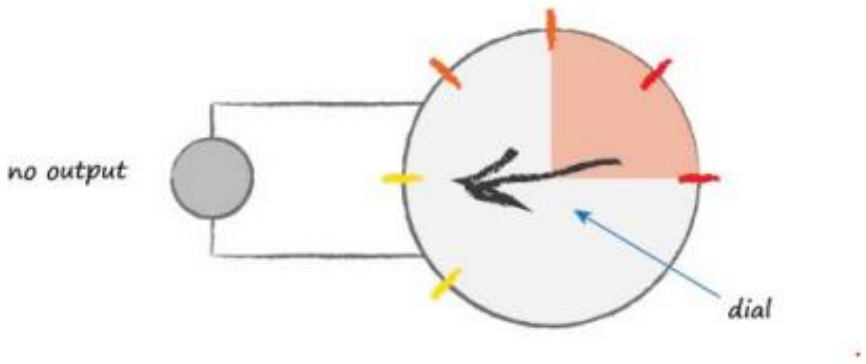
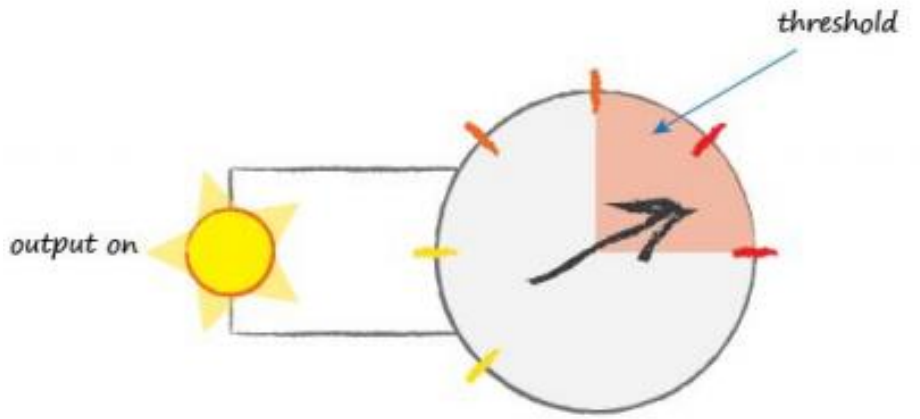


Figura 2: Superamento del livello di soglia e attivazione del neurone



La funzione che prende il segnale di input e genera un segnale di output, solo se ha superato un certo valore di soglia è chiamata funzione di attivazione. La funzione di attivazione può essere riprodotta con il grafico in figura 3, se in ingresso si ha un valore basso l'output vale zero. Quando viene raggiunto il valore di soglia l'output raggiunge il valore alto. Matematicamente, la funzione che permette questo è chiamata funzione di trasferimento sigmoidale, riprodotta in figura 4.

Figura 3: Funzione di attivazione

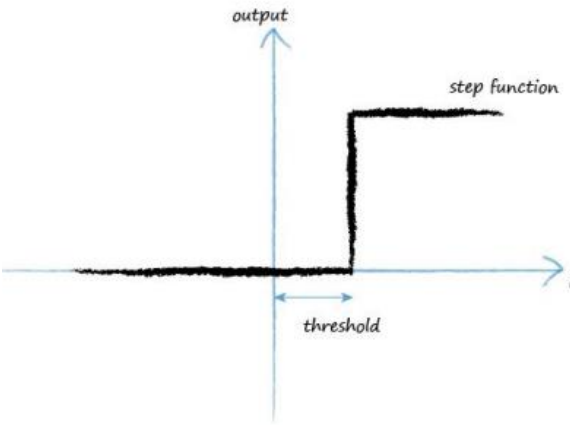
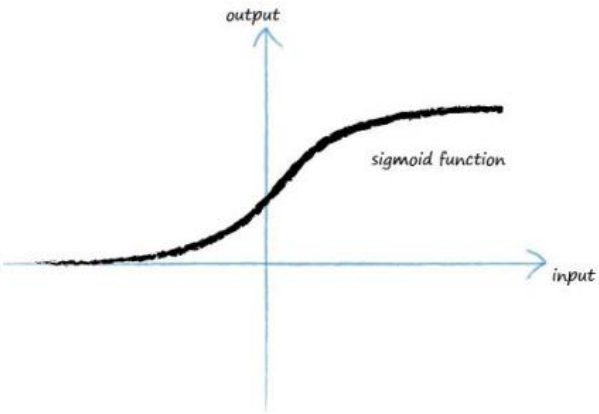


Figura 4: Funzione di trasferimento sigmoidale



La funzione sigmoidale viene anche chiamata funzione logistica e corrisponde a:

$$y = \frac{1}{1 + e^{-x}}$$

La funzione logistica è composta da un numero trascendente (e), che è una costante matematica pari circa a 2,71828. L'input x è negato e "e" viene elevato per questo fattore "–x". Al risultato viene aggiunto 1, per ottenere " $1 + e^{-x}$ ", e infine viene applicato l'inverso per ottenere la funzione logistica.

Esistono molte funzioni di attivazione per le reti neurali, ma questa permette di eseguire calcoli molto più semplici e successivamente nel programma si noterà l'utilità dell'utilizzo di questa funzione.

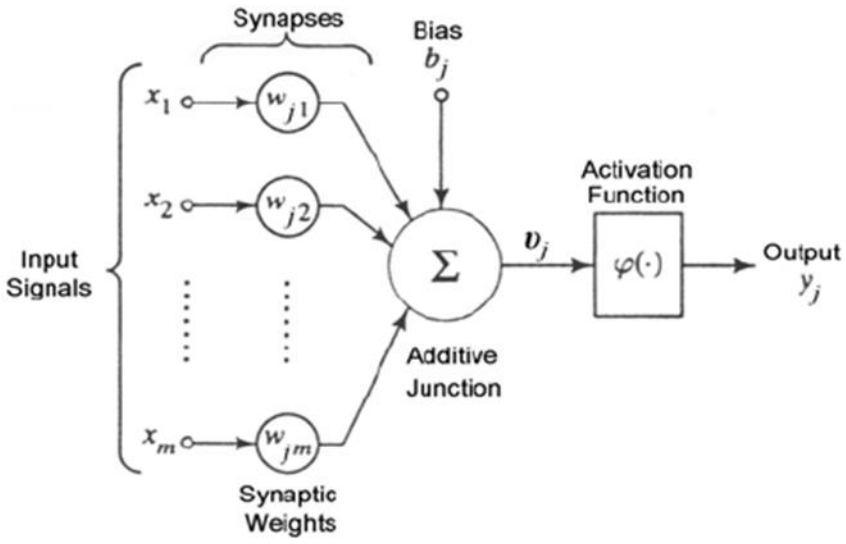
Rete neurale artificiale

L'uomo nel corso degli anni ha cercato di replicare la struttura precedentemente spiegata per poter rendere più efficienti le intelligenze artificiali, creando così una rete neurale artificiale.

Una rete neurale artificiale, normalmente chiamata solo "rete neurale", è un modello matematico/informatico di calcolo basato sulle reti neurali biologiche. Tale modello è costituito da un gruppo di interconnessioni di informazioni costituite da neuroni artificiali e può essere utilizzato per risolvere problemi ingegneristici di intelligenza artificiale come quelli che si pongono in diversi ambiti tecnologici.
La caratteristica più importante di questo sistemi è quella di poter apprendere modelli matematici attraverso "l'esperienza". La rete neurale non viene quindi programmata, bensì "addestrata" attraverso un processo di apprendimento basato su dati empirici.

L'unità elementare di calcolo di una rete neurale è il neurone. Può avere uno o più ingressi provenienti da altri neuroni e ha solo un'uscita eventualmente diretta verso uno più neuroni successivi. I segnali, sia in ingresso che in uscita possono assumere un qualsiasi valore continuo compreso tra 0 e 1.

Figura 5: Struttura di un neurone



Il neurone, in figura 5, ha 4 ingressi a cui arrivano i segnali X1 ÷ X*m* ciascuno associato ad un peso (*wj*1 ÷ *wjm*). I pesi sono un fattore moltiplicativo applicato al segnale di ingresso, possono essere maggiori o minori di 1, positivi o negativi. Il termine *bj* tiene conto dell'eventuale bias della funzione di attivazione a cui viene applicato l'ingresso fisso 1 in modo da avere la funzione di attivazione centrata sullo 0.

Si definisce attivazione interna del neurone (chiamata A) la somma pesata di tutti i segnali d'ingresso:

$$A = H1*w1 + H2 * w2 + H3 * w3 + H4 * w4$$

Il segnale in uscita dal neurone è chiamato attività ed è calcolato applicando la funzione di trasferimento sigmoidale.

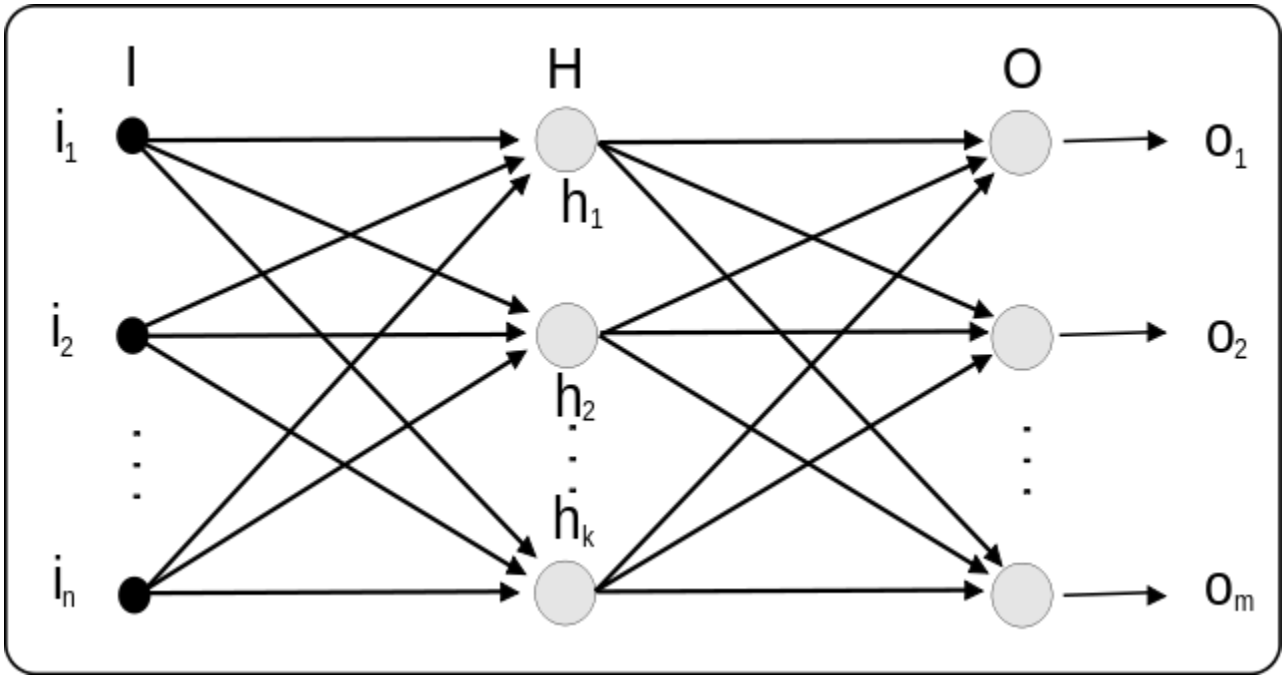


Figura 6: Struttura rete neurale artificiale

Nella struttura della rete neurale artificiale i singoli neuroni vengono collegati alla schiera di neuroni successivi, in modo da formare una rete di neuroni come in figura 6, attraverso delle linee chiamate link. Ad ogni link è associato un peso, cioè una costante che verrà moltiplicata per il valore di ingresso presente nel nodo. Normalmente una rete è formata da tre strati: nel primo abbiamo gli ingressi (*I, input*), questo strato si preoccupa di trattare gli ingressi in modo da adeguarli alle richieste dei neuroni. Se i segnali in ingresso sono già trattati può anche non esserci. Il secondo strato è quello nascosto (*H*, *hidden*), si preoccupa dell'elaborazione vera e propria e può essere composto anche da più colonne di neuroni. Il terzo strato è quello d'uscita (*O, output*) e si preoccupa di raccogliere i risultati ed adattarli alle richieste del blocco successivo della rete neurale. Queste reti possono essere anche molto complesse e coinvolgere migliaia di neuroni e decine di migliaia di connessioni.

Esistono vari algoritmi per addestrare una rete neurale. L'algoritmo utilizzato nel programma di questa tesina è definito come *Apprendimento supervisionato:* qualora si disponga di un insieme di dati per l'addestramento (*training set*) comprendente esempi tipici d'ingresso con le relative uscite loro corrispondenti, la rete può imparare a dedurre la relazione che li lega. Successivamente, la rete è addestrata mediante un opportuno algoritmo (backpropagation), il quale usa tali dati allo scopo di modificare i pesi ed altri parametri della rete stessa in modo da

minimizzare l'errore di previsione relativo all'insieme d'addestramento. Se l'addestramento ha successo, la rete è in grado di riconoscere la relazione incognita che lega le variabili d'ingresso a quelle d'uscita, ed è quindi in grado di fare previsioni anche laddove l'uscita non sia nota a priori. L'obiettivo finale dell'apprendimento supervisionato è la previsione del valore dell'uscita per ogni valore valido dell'ingresso, basandosi soltanto su un numero limitato di esempi di corrispondenza.

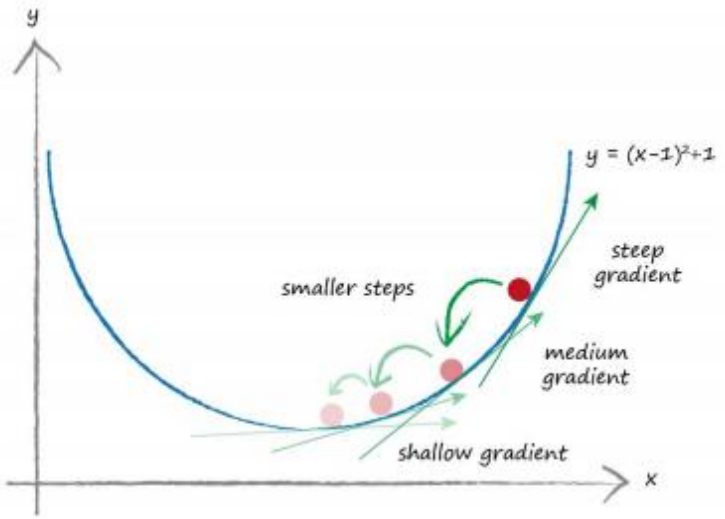
La rete neurale è in grado di:

- Apprendere per esempi, come si è detto parlando del paradigma di apprendimento supervisionato;
- Generalizzare le soluzioni apprese, ovvero rispondere correttamente a stimoli simili a quelli usati durante l'apprendimento;
- Astrarre nuove soluzioni, ossia rispondere correttamente a stimoli simili a quelli usati durante l'apprendimento;
- Trattare dati "rumorosi", ossia rispondere correttamente anche in presenza di dati alterati o parziali.

Uno dei metodi più noti ed efficaci per l'addestramento di tale classe di reti neurali è il cosiddetto algoritmo di retro-propagazione dell'errore (error backpropagation), il quale modifica sistematicamente i pesi delle connessioni tra i nodi, così che la risposta della rete si avvicini sempre di più a quella desiderata. Il modo per ottenere un errore relativamente piccolo è quello di calcolare il minimo della funzione. Quando si ha una funzione la quale è difficile calcolare il minimo si utilizza il metodo della "Pendenza Gradiente", questo metodo non da la risposta precisa e corretta ma si avvicina molto. In pratica si variano a "step", "passi", i pesi della rete in direzione opposta alla pendenza della derivata (gradiente).

Nel semplice esempio in figura 7 si può vedere l'utilizzo degli step che procedono verso il minimo della funzione.

Figura 7: Trovare il minimo della funzione utilizzando gli step



Riconoscimento della scrittura umana

L'obiettivo del progetto esposto nella tesina è quello di far riconoscere ad una macchina la scrittura di numeri a mano libera di qualsiasi persona.

Per la realizzazione di questo progetto ho creato una rete neurale con il linguaggio di programmazione Python.

Python è un linguaggio di programmazione che ha tra i principali obiettivi dinamicità, semplicità e flessibilità. Grazie a queste ottime qualità è uno dei linguaggi più utilizzati nelle università e nelle ricerche scientifiche. Le caratteristiche immediatamente riconoscibili di Python sono le variabili non tipizzate e l'uso dell'indentazione per la definizione delle specifiche. Una variabile è un contenitore al quale viene associata un'etichetta (il nome) che può essere associata a diversi contenitori anche di tipo diverso durante il suo tempo di vita.

Grazie al linguaggio di programmazione precedentemente descritto ho creato una classe della rete neurale generica che permette, se ben impostata, di riconoscere un numero scritto a mano libera. Un buon programmatore, scienziato e matematico, prova sempre a creare un codice generale. E' un ottimo metodo, perché ci obbliga a pensare a risolvere problemi in modo più profondo e più ampio, in modo da poter applicare questo programma in scenari e problemi differenti.

Lo studio della creazione della rete neurale è stato suddiviso in 3 parti:

- Inizializzazione – per definire la dimensione della rete;
- Addestramento – per calcolare i pesi dopo aver fornito la lista di esempi;
- Test – per ricevere una risposta in uscita quando forniamo un esempio mai affrontato precedentemente.

La struttura del programma è stata facilitata dall'uso di librerie pubbliche in Python che permettono, per esempio nel caso di *numpy* e *scipy*, di fare calcoli numerici vettoriali con semplicità, infatti la rete neurale viene strutturata mediante matrici e vettori.

1 – Inizializzazione

La funzione di inizializzazione della classe *neuralNetwork* serve per definire la dimensione della rete neurale. Inizialmente, come si può notare nella figura 8, la funzione imposta il numero di nodi nel layer input, centrale e output (iNodes, hNodes, oNodes). Successivamente inizializza i pesi dei vari link che collegano i nodi, per rendere più semplici i calcoli i valori dei pesi vengono inseriti dentro delle matrici, grazie all'uso della libreria interna *numpy* che permette di creare ed eseguire calcoli matriciali attraverso l'uso di semplici funzioni.

Il valore del peso del link inizialmente viene inizializzato in modo casuale, modificato poi nella fase di addestramento. La distribuzione di probabilità utilizzata per i pesi è di tipo gaussiano centrata

intorno allo zero con una deviazione standard proporzionale al numero di collegamenti in ingresso in un nodo, 1/√(numero di collegamenti in entrata).
Nell'esempio inserito nel codice della Figura 8 si può vedere che il programma esegue l'elevamento a potenza di -0.5 dei nodi centrali [*pow(hNodes, -0.5)*].

Viene quindi impostata la funzione di attivazione sigmoidale presente nella libreria *scipy* con il nome di *expit*.

Figura 8: Segmento di codice della funzione che inizializza la rete neurale

```
def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):

        self.iNodes = inputnodes
        self.hNodes = hiddennodes
        self.oNodes = outputnodes

        # impostando i pesi dei link da input al centrale, dal centrale
all'output con numeri casuali
        # wih = weight input-hidden     who = weight hidden-output
        # w11 w21
        # w12 w22 ecc
        self.wih     =     numpy.random.normal(0.0,     pow(self.hNodes,     -0.5),
(self.hNodes, self.iNodes))
        self.who     =     numpy.random.normal(0.0,     pow(self.oNodes,     -0.5),
(self.oNodes, self.hNodes))

        # learning rate
        self.lr = learningrate

        # creazione della funzione sigmoid tramite la libreria scipy.special
        self.sigmoidFunction = lambda x: scipy.special.expit(x)

        pass
```

3 – Apprendimento

La funzione di apprendimento prende l'input e restituisce in uscita la riposta della rete. Viene analizzata per prima la fase di apprendimento perché più facile ed è più simile a quella di Addestramento. Il segnale d'ingresso deve passare tra lo strato di input dei nodi, attraverso lo strato centrale e infine nello strato d'output finale. Il segnale d'ingresso viene moderato attraverso i pesi e viene utilizzata la funzione di attivazione sigmoid per "schiacciare" il valore proveniente da quei nodi.

Il segnale di input inizialmente viene moltiplicato per i pesi che collegano i nodi input con quelli centrali. W_{input_hidden} = Pesi input-centrale.

$$X_{hidden} = W_{input\_hidden} \cdot Input$$

Questa moltiplicazione tra matrici avviene attraverso l'utilizzo della funzione "*numpy.dot*", che accetta come parametri le due matrici da moltiplicare e restituisce in uscita il loro prodotto.

Il risultato del prodotto viene quindi passato alla funzione di attivazione.

La stessa cosa viene effettuata, come si può notare nel segmento di codice in figura 9, per ricavare i segnali che emergono dai nodi finali. Infine la funzione ritorna la risposta finale della rete.

Figura 9: Segmento di codice della funzione che ritorna la risposta della rete

```
def query(self, input_list):

        # converte la lista input in un vettore 2d e fa la trasposta
        inputs = numpy.array(input_list, ndmin=2).T

        # calcola la matrice che entra nei nodi centrali
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calcola la matrice che esce dai nodi centrali, applicando la funzione
sigmoid
        hidden_outputs = self.sigmoidFunction(hidden_inputs)

        # calcola la matrice che entra nei nodi finali
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calcola la matrice che esce dai nodi finali, applicando la funzione
sigmoid
        final_outputs = self.sigmoidFunction(final_inputs)

        return final_outputs
```

2 – Addestramento

La funzione di addestramento ridefinisce i pesi della rete per ottenere un errore finale sempre più piccolo, accettando come parametri d'ingresso l'input, che può essere sia un'immagine che una lista di target che contiene la risposta corretta. Inizialmente questa funzione sarà molto simile alla funzione di apprendimento perché entrambe devono calcolare il valore finale che in questo caso serve per confrontarlo con il valore ideale.

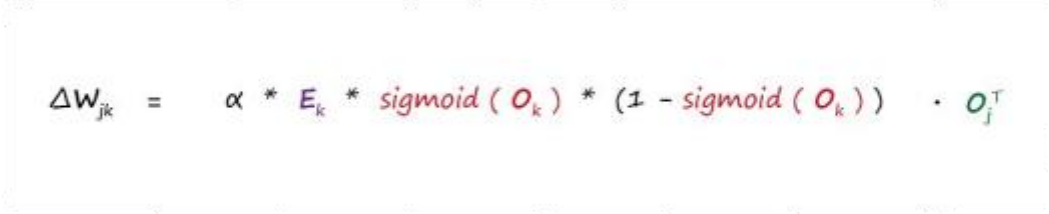
Migliorare i pesi dei vari link attraverso l'errore tra il valore d'uscita calcolato e quello ideale è il cuore della rete neurale.

Per prima cosa bisogna calcolare l'errore, la differenza tra il valore ideale e quello ideale.

$$output\_errors = targets - final\_output$$

Una volta eseguita questa semplice differenza si può calcolare per ridefinire i pesi ad ogni strato. Per i pesi tra i nodi centrali e finali si utilizza l'errore di output (output_errors), per i pesi tra i nodi iniziali e centrali si utilizza l'errore centrale (hidden_errors).

Figura 10: Formula ridefinizione dei pesi



La formula nella figura 10 permette di ricavare il peso dei link da aggiungere ai vari nodi. α è la soglia di apprendimento, e sigmoid è la funzione di attivazione. Il simbolo " * " indica la moltiplicazione normale elemento per elemento e il punto indica la moltiplicazione matriciale. Grazie al linguaggio di programmazione Python, è possibile rendere la formula molto più semplice con l'utilizzo delle varie librerie.

Figura 11: Traduzione della formula in codice Python

```
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)),
numpy.transpose(hidden_outputs))
```

La lunga riga di codice analizzata nella Figura 11 rappresenta l'aggiornamento dei pesi tra i nodi centrali e finali (who). E' stata inserita in diversi colori per poter aiutare a notare la relazione con la Figura 10.

Il codice finale dell'addestramento della rete neurale è inserito nella Figura 12 con l'aggiunta dell'aggiornamento dei pesi dei nodi iniziali-centrali.

Figura 12: Segmento di codice della funzione di addestramento della rete neurale

```
def train(self, input_list, target_list):

        # converte la lista input in un vettore 2d e fa la trasposta (.T)
        inputs = numpy.array(input_list, ndmin=2).T
        # converte la lista target in un vettore 2d e fa la trasposta (.T)
        targets = numpy.array(target_list, ndmin=2).T

        # calcola la matrice che entra nei nodi centrali
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calcola la matrice che esce dai nodi centrali, applicando la funzione
sigmoid
        hidden_outputs = self.sigmoidFunction(hidden_inputs)

        # calcola la matrice che entra nei nodi finali
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calcola la matrice che esce dai nodi finali, applicando la funzione
sigmoid
        final_outputs = self.sigmoidFunction(final_inputs)

        # l'errore in uscita è (target - actual)
        output_errors = targets - final_outputs
```

```
        # l'errore del nodo centrale è l'errore d'uscita diviso nei pesi del
nodo centrale
        # errore nodo centrale = Trasposta dei pesi tra centro e fine * errore
in uscita
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # aggiornamento dei pesi nei link tra la parte centrale e l'output
        # il learning rate * la moltiplicazione matriciale tra: (l'errore
d'uscita, l'uscita attuale finale, 1 - l'uscita attuale finale) e la trasposta
dell'uscita del nodo centrale
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1 -
final_outputs)), numpy.transpose(hidden_outputs))
        # aggiornamento dei pesi nei link tra l'input e la parte centrale
        self.wih += self.lr * numpy.dot(( hidden_errors * hidden_outputs * (1 -
hidden_outputs)), numpy.transpose(inputs))

        pass
```

In tal modo viene definita la classe della rete neurale.

Test della rete neurale

L'obiettivo del progetto esposto nella tesina è quello di far riconoscere ad una macchina i primi dieci numeri scritti a mano libera.

Avere un computer in grado di classificare correttamente cosa contenga un'immagine, chiamato anche problema di riconoscimento delle immagini, è un problema ancora molto diffuso nel mondo. Solo recentemente è stata trovata una soluzione in cui la rete neurale abbia un ruolo cruciale nel riconoscimento delle immagini.
Per dare un senso di quanto complesso sia il problema del riconoscimento delle immagini, noi umani spesso siamo in disaccordo su cosa un'immagine contenga. Saremmo in disaccordo su cosa sia effettivamente un carattere scritto a mano, soprattutto nel caso di un carattere scritto di fretta e senza cura.

L'insieme di dati utilizzati in questo programma per testare la rete neurale, sono due database composti da cifre scritte a mano libera da studenti di un istituto superiore.
Il database di addestramento, chiamato *training set*, è composto da 60.000 cifre etichettate. Etichettato vuol dire che ogni ingresso viene fornito con la risposta, l'output desiderato, cioè con quello che dovrebbe essere l'output corretto.
Il database di apprendimento, chiamato *test set*, è composto da 10.000 cifre e viene utilizzato per vedere come l'algoritmo del programma funziona. Anche in questo database i dati contengono la risposta corretta così da poter verificare il corretto funzionamento della rete neurale.
Vengono utilizzati due database diversi per l'addestramento e l'apprendimento per essere sicuri che la rete analizzi immagini mai viste precedentemente.

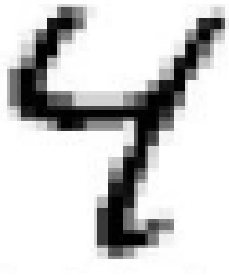


Figura 13: Cifra del Database

Un esempio di una cifra all'interno del database è quella posta nella figura 13. L'immagine rappresenta un numero 4 scritto a mano libera. Il contenuto delle righe di testo presente nel dataset di apprendimento è strutturato come segue:

- Il primo valore è il numero rappresentato nell'immagine. Questa è la risposta che la rete neurale deve imparare per rispondere correttamente.
- Le sequenze di numeri successivi, separati da una virgola, corrispondono ad un valore pixel da 0 a 255 che compongono la cifra, per esempio se il pixel è completamente nero il suo valore sarà 255 viceversa se è bianco sarà 0. Questi numeri sono suddivisi in una matrice 28x28, e in totale sono 784 valori.

La rete neurale, perciò, in ingresso avrà 784 nodi, uno per ogni pixel.

La scelta del numero di nodi centrali non è così scientifico. Non si può scegliere un numero di nodi più grande di 784 perché possono essere espressi in una forma più breve dell'input stesso. Così scegliendo un valore più piccolo del numero di input, si forza la rete a cercare di riassumere le caratteristiche chiave dell'input.

La rete neurale, perciò, nel layer centrale avrà 200 nodi.

Preparazione dati database

Innanzitutto bisogna aprire nel programma di Python il file del database. Ciò è possibile grazie a 3 semplici istruzioni:

- open("*indirizzo dove si trova il file*", 'r'), attraverso questa istruzione si apre il file desiderato in sola lettura in modo tale da non modificare accidentalmente il documento;
- training_data.readlines(), per leggere tutti i dati del file;
- file.close(), si chiude il file precedentemente aperto.

Successivamente si crea un ciclo che permette di analizzare tutte le cifre del database. Ogni elemento da 0 a 255 viene diviso da una virgola e ridimensionato, perché la rete neurale lavora correttamente se in ingresso ha valori bassi da 0 a 1.

Adesso bisogna pensare all'uscita della rete neurale. La funzione sigmoid che viene utilizzata non può mandare in output un valore come -2.0 o 255. Il range di valori in uscita possono essere da 0 a 1. Ma che tipo di output si vuole avere?

La rete neurale deve classificare correttamente l'immagine dando la risposta corretta, che è uno dei 10 numeri, da 0 a 9. Perciò basta avere 10 uscite, una per ogni possibile uscita. Se per esempio la risposta corretta è il numero "0" la rete in uscita al primo nodo deve inviare un valore alto (1) e "silenzierà" il resto dei nodi (0).

La risposta della rete neurale viene confrontata con una lista contenente la risposta corretta. La lista è chiamata *targets* ed è composta da 10 valori uguali a 0.01. La risposta viene inserita nella posizione corretta, se per esempio la risposta è "0" il primo valore sarà uguale a 0.99, viceversa se il numero è "5" il sesto valore (sesto, perché si parte da 0 e si arriva a 9) sarà uguale a 0.99.

Il primo database che viene analizzato è quello di addestramento, composta da 60.000 cifre. Una volta concluso il ciclo in figura 14 la rete è stata correttamente addestrata.

Figura 14: Segmento di codice che addestra la rete neurale

```
# carica la lista dei file di train
training_data_file = open("C:/Users/enric/Desktop/Tesina di maturita/Rete
Neurale/mnist_dataset/mnist_train.csv",'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# fase di apprendimento della rete neurale
for record in training_data_list:
    # divide ogni numero appena incontra la ","
    all_values = record.split(',')

    # riscala gli input da 0-255 a 0.01 a 0.99
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01

    # crea una lista composta da 10 valori che corrispondo ai target, numeri da
riconoscere da 0 a 9
    targets = numpy.zeros(output_nodes) + 0.01

    # all_values[0] contiene il numero da identificare
    # se all_values[0] è 5 il programma indica quale numero bisogna riconoscere
    # targets sarà [0.01, 0.01, 0.01, 0.01, 0.01, 0.99, ....]
    # il 0.99 si trova al sesto posto perciò corrisponde al numero 5
    targets[int(all_values[0])] = 0.99
    n.train(inputs, targets)
    pass
```

Per sapere se la rete è stata correttamente addestrata viene utilizzato il secondo database di apprendimento, composto da 10.000 elementi.

Innanzitutto, come fatto precedentemente con il primo database, bisogna aprire il file nel programma e salvare tutti i dati in una variabile.

Successivamente si crea un ciclo che permette di analizzare tutte le immagini del database. Ogni elemento da 0 a 255 viene diviso da una virgola e ridimensionato, come fatto precedentemente.
La risposta corretta dell'immagine viene inserita in una variabile, per poi sapere se la rete ha risposto correttamente al nuovo stimolo ricevuto in ingresso. Attraverso la funzione *n.query()* si ottiene la risposta del sistema. In uscita la rete fornisce una lista di 10 valori, la posizione a cui corrisponde il valore più alto è la risposta del sistema. La risposta corretta e quella della rete vengono confrontate, se la risposta è corretta il programma incrementa un contatore per vedere quanti elementi riconosce correttamente. Infine, come si può vedere in figura 15, il programma stampa in uscita la percentuale di successo della rete neurale.

L'accuratezza della rete neurale, su un database di 10.000 immagini, è circa del 98%!

Figura 15: Segmento di codice che esegue il test della rete neurale

```
test_data_file = open("tesina/test.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
scorecard = []

for record in test_data_list:
    # divide ogni numero della lista testa appena incontra una virgola
    all_values = record.split(',')

    # inserisce la risposta corretta in una variabile
    correct_label = int(all_values[0])
    print correct_label, "numero corretto"

    #riscala gli input da 0-255 a 0.01 a 0.99
    inputs = (numpy.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01

    # uscita della rete
    outputs = n.query(inputs)
    # prende il valore più alto tra le 10 uscite e lo assegna a label
    label = numpy.argmax(outputs)
    print label, "risposta della rete"

    #confronta il valore giusto con quello della rete neurale
    if label == correct_label:
        # aggiunge 1 alla lista scorecard
        scorecard.append(1)

else:
        # aggiunge 0 alla lista scorecard
        scorecard.append(0)

    pass
print scorecard
scorecard_array = numpy.asarray(scorecard)
print "Accuratezza = ", (float(scorecard_array.sum()) /
scorecard_array.size)*100, "%"
```

Interfacciamento del programma con il mondo esterno

Il programma precedentemente sviluppato riconosce immagini scaricate da un database. Se si volesse utilizzare la rete neurale per riconosce un nuovo numero non presente nel database, bisogna interfacciare il programma con una semplice telecamera.

La telecamera viene utilizzata per scattare la foto del numero disegnato e per poterlo poi successivamente analizzare. La telecamera da interfacciare al programma può essere una qualsiasi, si può utilizzare la telecamera interna del pc portatile oppure una telecamera collegata opportunatamente al computer.

In questa tesina, per scattare la foto al numero disegnato, vengono utilizzate le telecamere del robot Nao.

Figura 16: Nao, il robot umanoide

Nao è un robot umanoide nato nel 2004 di taglia media, autonomo e programmabile, sviluppato dalla Aldebaran Robotics, società francese di tecnologia, acquistata recentemente nel 2013 da un'altra società giapponese chiamata SoftBank Robotics.

Nao dispone di 25 gradi di libertà ed è equipaggiato da cinque sensori di prossimità ad ultrasuoni rivolti in direzioni diverse e da sette sensori tattili posti sui piedi, sulle mani e sulla testa. Dispone anche di un sistema multimediale evoluto della sintesi vocale (riproduzione artificiale della voce umana), la localizzazione nello spazio, e per il riconoscimento facciale e di oggetti. Possiede inoltre 4 microfoni, due altoparlanti e due telecamere. Il robot ha un sistema operativo integrato Linux ed è possibile programmarlo usufruendo della connessione Wi-Fi e dell'utilizzo del suo ambiente di sviluppo chiamato Choregraphe oppure attraverso diversi linguaggi di programmazione come il C++ e il Python.

La gamma delle possibili utilizzazioni è molto vasta, dipendendo dalla sua programmabilità: tra gli usi immaginabili vi è quello di un robot di compagnia, compagno di giochi, assistenti ai malati, oggetto interattivo, ecc.

Nel progetto sviluppato in questa tesina viene utilizzato Nao come "macchina fotografica", e non una macchina qualunque, perché ha un aspetto molto simile a noi e provoca l'interesse di molte persone. Inoltre viene utilizzato perché permette al programmatore di usufruire di molteplici sensori. Per esempio ogni volta che si vuole scattare una foto può eseguire l'azione attraverso un comando vocale oppure attraverso la pressione di un opportuno sensore tattile.

Nel progetto di questa tesina, quando viene premuto un sensore tattile di Nao scatta una foto salvandola in una apposita cartella e la analizza inserendola all'interno della rete neurale.
Il programma in figura 17 illustra il codice utile allo scatto della foto.

La funzione "showNaoImage" come parametri (come valori importanti) accetta l'indirizzo IP del robot umanoide Nao, la porta di comunicazione, e il nome con cui viene salvata l'immagine. Attraverso l'uso di Python è possibile utilizzare dei moduli già presenti in Nao che permettono di usufruire dei sensori presenti in esso. Fintantoché non viene toccato un sensore tattile a Nao, il programma non procede con le istruzioni successive. Attraverso il modulo "*AlVideoDevice*" si può usufruire delle telecamere di Nao. Successivamente il programma imposta la risoluzione e i colori e scatta la foto, ottenendo le dimensioni delle immagini e i pixel. Unendo tutti i pixel ottenuti forma l'immagine e la salva con il nome precedentemente impostato (ofile).

Figura 17: Segmento di codice che scatta una foto

```
def showNaoImage(IP, PORT, ofile):
    camProxy = None

     try:
        # utilizza la telecamera di nao connettendosi all'indirizzo e la porta
giusta
        camProxy = ALProxy("ALVideoDevice", IP, PORT)
        #imposta risoluzioni e colori della foto
        resolution = 2
        colorSpace = 11   # RGB

        videoClient = camProxy.subscribe("python_client", resolution,
colorSpace, 5)

        # ottiene la foto dalle telecamere di Nao
        # image[6] contiene i dati dell'immagine trasformati in array di
carattere ASCII
        naoImage = camProxy.getImageRemote(videoClient)

        # ottiene le dimensioni dell'immagine e i pixel
        # i primi due parametri della lista naoImage sono la dimensione
dell'immagine
        # dal 7 elemento (naoImage[6]) contiene i pixel dell'immagine
        imageWidth = naoImage[0]
        imageHeight = naoImage[1]
        array = naoImage[6]

        # Crea un'immagine di unendo i vari pixel
```

```
        im = Image.frombytes("RGB",(imageWidth,imageHeight) ,array)
        # salva l'immagine
        im.save(ofile, "PNG")

    except Exception as e:
            print e.message

    finally:
        if camProxy!= None:
            camProxy.unsubscribe(videoClient)
```

"*ReactToTouch*" è una classe che permette di rilevare l'evento della pressione di un sensore tattile di Nao, sempre grazie ai moduli presenti sul robot. Inoltre questa funzione permette anche di far parlare il robot umanoide attraverso la funzione "*say*" in figura 18.

Figura 18: Segmento di codice che rileva la pressione di un sensore tattile

```
class ReactToTouch(ALModule):

    def __init__(self,name):
        ALModule.__init__(self,name)

        global memory
        memory = ALProxy("ALMemory")
        memory.subscribeToEvent("TouchChanged", "ReactToTouch", "onTouched")
        self.tts.saay("Sono pronto")

    # viene attivata questa funzione ogni volta che viene premuto un sensore
tattile
    def onTouched(self, strVarName, value):
        # Disattiva l'evento quando parla per evitare ripetizioni
        memory.unsubscribeToEvent("TouchChanged","ReactToTouch")

        # scatta una foto
        showNaoImage(IP, PORT, ofile)
        # analizza attraverso la rete neurale la nuova foto
        label = RispostaRete()

        # Nao dice ciò che la rete neurale ha riconosciuto
        self.say(label)

        # Attiva l'evento di riconoscimento tattile
        memory.subscribeToEvent("TouchChanged","ReactToTouch","onTouched")


# funzione che permette di far parlare Nao
    def say(self,number):
        sentence = "Il numero è: " + number
        self.tts.say(sentence)
```

Ridimensionare e filtrare le immagini

Le immagini presenti nei database sono numeri digitalizzati in scala di grigi di dimensioni 28x28.

Le foto scattate con il robot umanoide, invece, sono foto a colori con una risoluzione maggiore di dimensione 640x480. La foto, perciò, deve essere impostata in bianco e in nero e ridimensionata correttamente per essere analizzata dalla rete neurale.

Queste operazioni sono facilmente eseguibili attraverso il linguaggio di programmazione Python, che utilizza librerie interne per il trattamento di immagini.

Nel segmento di codice riportato in figura 19 sono presenti le istruzioni che permettono di ridimensionare l'immagine nel formato desiderato (28x28) e di salvarla con un nome differente (FotoNao_28x28) per poterla poi successivamente analizzare. La nuova foto viene aperta e filtrata con una scala di grigi in modo ottenere i dati dei pixel con l'intensità di colore rilevato. Dopo aver effettuato queste operazioni, il programma analizza la foto attraverso la rete neurale e permette a Nao di dire la soluzione finale.

Figura19: Segmento di codice che ridimensiona e filtra l'immagine e ottiene la risposta della rete

```
def RispostaRete():

# Apertura immagine
img = cv2.imread(imgFileName + sExt,0)
img = cv2.medianBlur(img,5)
# conversione in scala di grigi dell'immagine
th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY,11,2)

###Codice per rendere più 'ciccione' il numero
kernel = np.ones((5,5), np.uint8)
th3 = cv2.erode(th3, kernel, iterations=1)

# l'immagine viene ridimensionata e salvata con un nuovo nome
th3 = cv2.resize(th3,(28, 28), interpolation = cv2.INTER_CUBIC)
cv2.imwrite(imgFileName + imgRszName + sExt,th3)

# l'immagine viene convertita in array e messa in bianco e nero
image_file_name = imgFileName + imgRszName + sExt
img_array = scipy.misc.imread(image_file_name, flatten = True)

# converte i dati dell'array in intensita' di colore rivelato
img_data = 255.0 - img_array.reshape(784)
img_data = (img_data / 255.0 * 0.99) + 0.01

# ottiene la risposta dalla rete neurale
inputs = numpy.asfarray(img_data)
outputs = n.query(inputs)
label = numpy.argmax(outputs)

return label
```

In tal modo il programma è pronto a rispondere correttamente a stimoli esterni.

Serializzazione avanzata della Rete Neurale

Compilare il codice richiede grandi quantità di tempo perché ogni volta viene creata una nuova rete neurale priva di esperienza e viene addestrata con migliaia di elementi.

Per evitare che ogni volta venga creata una nuova rete è stato inserito nel programma un salvataggio di essa in un file esterno.

Questa operazione è facilmente eseguibile attraverso l'utilizzo di poche righe di codice, poste in figura 20.

Figura 20: Segmento di codice che serializza la rete neurale

```
# Serializza l'oggetto rete neurale
    # librerie che permettono di serializzare la rete
    import cPickle as pickle
    import cloud.serialization
    # viene copiata la rete e inserita dentro result
    result = cloud.serialization.cloudpickle.dumps(n)
    # apre un nuovo file e inserisce la rete
    with open(ser_file, 'wb') as dest:
        dest.write(result)
```

Nel caso la rete fosse già stata salvata si può aprire e riutilizzare attraverso le righe di codice poste in figura 21.

Figura 21: Segmento di codice che carica la rete neurale dal file in cui era stata salvata

```
   # carica la rete neurale dal file in cui era stata
salvata(serilizzata) precedentemente
    filer = open(ser_file, 'rb')
    n = pickle.load(filer)
    filer.close()
```

In tal modo è possibile avere la rete neurale già addestrata in pochissimo tempo.

Conclusioni

Il programma esposto in questa tesina ha come obiettivo il riconoscimento di numeri scritti a mano libera. In futuro spero di ampliare il programma in modo tale da riconoscere anche le lettere dell'alfabeto e intere frasi scritte a mano libera.

Molte imprese nel mondo stanno già lavorando a questo progetto. Una Startup genovese di nome Horus Technology ha creato un dispositivo indossabile con le funzioni di un assistente personale.

Il dispositivo, come si può vedere in figura 22, viene posto su qualunque tipo di occhiali e dalla sua sede laterale osserva la realtà circostante, effettuando un'analisi che permette di trasformare immagini in parole. Horus può individuare il testo e guidare la persona ad inquadrarlo nella maniera più conveniente per effettuare la lettura. E' in grado di apprendere gli oggetti e i volti delle persone più care all'utente, per poterli riconoscere in svariate situazioni e può fornire supporto nella mobilità tramite l'individuazione di ostacoli e segnaletica stradale, come ad esempio strisce pedonali.

Figura 22: Occhiali Horus

Linkografia

Database di test con 10 elementi:
https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyourownneuralnetw ork/master/mnist_dataset/mnist_test_10.csv

Database di addestramento con 100 elementi:
https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyourownneuralnetw ork/master/mnist_dataset/mnist_train_100.csv

Database di test con 10.000 elementi:
http://www.pjreddie.com/media/files/mnist_test.csv

Database di addestramento con 60.000 elementi:
http://www.pjreddie.com/media/files/mnist_train.csv

Programma finale: Machine Learning

```
# Programmatore: Enrico Fiasché

# scipy.special contiene alcune funzioni speciali come la funzione sigmoid
# la funzione sigmoid si chiama expit()
import sys, time, numpy, numpy as np, matplotlib.pyplot as plt, random,
scipy.special, scipy.misc, cv2, PIL, argparse

from PIL import Image

# serializzazione avanzata
import cloud.serialization
import cPickle as pickle

import os.path

# moduli Nao
from naoqi import ALProxy
from naoqi import ALBroker
from naoqi import ALModule

IP = "172.31.1.34"
PORT = 9559

file_mnist_name = "mnist_train.csv"

imgFileName= "FotoNao"   # nome file contenente l'immagine originale
imgRszName= '_28x28' # nome file contenente l'immagine originale ridimensionata
a 28x28 pixels
sExt = ".png"   # estensione file immagine

ofile = imgFileName + sExt # nome della foto scattata con Nao

######################### INIZIO CLASSE RETE NEURALE ##########################

class neuralNetwork:
    # inizializzazione della rete neurale
    # iNodes = input Nodes, hNodes = hidden Nodes, oNodes = output Nodes
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):

        self.iNodes = inputnodes
        self.hNodes = hiddennodes
        self.oNodes = outputnodes

        # impostando i pesi dei link da input al centrale, dal centrale
all'output con numeri casuali
        # wih = weight input-hidden     who = weight hidden-output
        # w11 w21
        # w12 w22 ecc
        self.wih = numpy.random.normal(0.0, pow(self.hNodes, -0.5),
(self.hNodes, self.iNodes))
        self.who = numpy.random.normal(0.0, pow(self.oNodes, -0.5),
(self.oNodes, self.hNodes))

        # learning rate
        self.lr = learningrate

        # creazione della funzione sigmoid tramite la libreria scipy.special
        self.sigmoidFunction = lambda x: scipy.special.expit(x)
```

```
    # apprendimento della rete neurale
    def train(self, input_list, target_list):

        # converte la lista input in un vettore 2d e fa la trasposta (.T)
        inputs = numpy.array(input_list, ndmin=2).T
        # converte la lista target in un vettore 2d e fa la trasposta (.T)
        targets = numpy.array(target_list, ndmin=2).T

        # calcola la matrice che entra nei nodi centrali
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calcola la matrice che esce dai nodi centrali, applicando la funzione
sigmoid
        hidden_outputs = self.sigmoidFunction(hidden_inputs)

        # calcola la matrice che entra nei nodi finali
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calcola la matrice che esce dai nodi finali, applicando la funzione
sigmoid
        final_outputs = self.sigmoidFunction(final_inputs)

        # l'errore in uscita è (target - actual)
        output_errors = targets - final_outputs
        # l'errore del nodo centrale è l'errore d'uscita diviso nei pesi del
nodo centrale
        # errore nodo centrale = Trasposta dei pesi tra centro e fine * errore
in uscita
        hidden_errors = numpy.dot(self.who.T, output_errors)

        # aggiornamento dei pesi nei link tra la parte centrale e l'output
        # il learning rate * la moltiplicazione matriciale tra: (l'errore
d'uscita, l'uscita attuale finale, 1 - l'uscita attuale finale) e la trasposta
dell'uscita del nodo centrale
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 -
final_outputs)), numpy.transpose(hidden_outputs))
        # aggiornamento dei pesi nei link tra l'input e la parte centrale
        self.wih += self.lr * numpy.dot(( hidden_errors * hidden_outputs * (1.0
- hidden_outputs)), numpy.transpose(inputs))

        pass

    # risposta della rete neurale
    # input_list è la lista degli input che entrano nei nodi iniziali
    # esempio di input_list: input_list = [1.0, 2.3, 1000.0]
    def query(self, input_list):

        # converte la lista input in un vettore 2d e fa la trasposta
        inputs = numpy.array(input_list, ndmin=2).T

        # calcola la matrice che entra nei nodi centrali
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calcola la matrice che esce dai nodi centrali, applicando la funzione
sigmoid
        hidden_outputs = self.sigmoidFunction(hidden_inputs)

        # calcola la matrice che entra nei nodi finali
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calcola la matrice che esce dai nodi finali, applicando la funzione
sigmoid
        final_outputs = self.sigmoidFunction(final_inputs)

        return final_outputs
```

```
########################## FINE CLASSE RETE NEURALE ##########################
########### INIZIO CLASSE RICONOSCIMENTO PRESSIONE SENSORE TATTILE ############

class ReactToTouch(ALModule):
    # inizializzazione della classe
    def __init__(self, name):
        ALModule.__init__(self, name)

        # Create un proxy per poter poi successivamente parlare
        self.tts = ALProxy("ALTextToSpeech")

        # Attiva l'evento di riconoscimento tattile
        global memory
        memory = ALProxy("ALMemory")
        memory.subscribeToEvent("TouchChanged",
            "ReactToTouch",
            "onTouched")
      self.tts.say("Sono pronto")

    # viene attivata questa funzione ogni volta che viene premuto un sensore
tattile
    def onTouched(self, strVarName, value):
        # Disattiva l'evento quando parla per evitare ripetizioni
        memory.unsubscribeToEvent("TouchChanged",
            "ReactToTouch")

        # scatta una foto
        showNaoImage(IP, PORT, ofile)
        # analizza attraverso la rete neurale la nuova foto
        label = RispostaRete()

        # Nao dice ciò che la rete neurale ha riconosciuto
        self.say(label)

        # Attiva l'evento di riconoscimento tattile
        memory.subscribeToEvent("TouchChanged",
            "ReactToTouch",
            "onTouched")

    # funzione che permette a Nao di parlare
    def say(self, number):
      sentence = "Il numero e'" + str(number)
        self.tts.say(sentence)

############# FINE CLASSE RICONOSCIMENTO PRESSIONE SENSORE TATTILE ############
######################### INIZIO FUNZIONE SCATTA FOTO NAO ######################

def showNaoImage(IP, PORT, ofile):
    camProxy = None

    try:
        # utilizza la telecamera di nao connettendosi all'indirizzo e la porta
giusta
        camProxy = ALProxy("ALVideoDevice", IP, PORT)
        # imposta risoluzione dei colori e della foto
        resolution = 2
        colorSpace = 11    # RGB

        videoClient = camProxy.subscribe("python_client", resolution,
colorSpace, 5)

        # ottiene la foto dalle telecamere di Nao
```

```
        # image[6] contiene i dati dell'immagine trasformati in array di
carattere ASCII
        naoImage = camProxy.getImageRemote(videoClient)

        # ottiene le dimensioni dell'immagine e  i pixel
        # i primi due parametri della lista naoImage sono la dimensione
dell'immagine
        # dal 7 elemento (naoImage[6]) contiene ii pixel dell'immagine
        imageWidth = naoImage[0]
        imageHeight = naoImage[1]
        array = naoImage[6]

        # Crea un'immagine unendo i vari pixel
        im = Image.frombytes("RGB",(imageWidth,imageHeight) ,array)

        # Salva l'immagine
        im.save(ofile, "PNG")

    except Exception as e:
            print e.message

    finally:
        if camProxy!= None:
            camProxy.unsubscribe(videoClient)

########################## FINE FUNZIONE SCATTA FOTO NAO ##########################
######################### INIZIO FUNZIONE FILTRA IMMAGINE #########################

def binarize_image(img_path, target_path, threshold,bShow=False):

    image = cv2.imread(img_path)

    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    arImg = numpy.array(gray_image)
    img_BW = binarize_array(arImg, threshold)
    if target_path != None:
        cv2.imwrite(target_path,img_BW)

    return img_BW

def binarize_array(numpy_array, threshold=200):

    for i in range(len(numpy_array)):
        for j in range(len(numpy_array[0])):
            if numpy_array[i][j] > threshold:
                numpy_array[i][j] = 255
            else:
                numpy_array[i][j] = 0

    return numpy_array

########################## FINE FUNZIONE FILTRA IMMAGINE ##########################
######################### INIZIO FUNZIONE ANALIZZA IMMAGINE #########################

def RispostaRete():
    img = cv2.imread(imgFileName + sExt,0)

    img = cv2.medianBlur(img,5)

    # conversione in scala di grigi dell'immagine
    ret,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
```

```
    th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
                    cv2.THRESH_BINARY,11,2)
    th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
                    cv2.THRESH_BINARY,11,2)

    ###Codice per rendere più 'ciccione' il numero
    kernel = np.ones((5,5), np.uint8)
    th3 = cv2.erode(th3, kernel, iterations=1)

    # l'immagine viene ridimensionata e salvata con un nuovo nome
    th3 = cv2.resize(th3,(28, 28), interpolation = cv2.INTER_CUBIC)
    cv2.imwrite(imgFileName + imgRszName + sExt,th3)

    image_file_name = imgFileName + imgRszName + sExt
    print image_file_name
    # l'immagine viene convertita in array e messa in bianco e nero
    img_array = scipy.misc.imread(image_file_name, flatten = True)

    # converte i dati dell'array in intensita' di colore rivelato
    img_data = 255.0 - img_array.reshape(784)
    img_data = (img_data / 255.0 * 0.99) + 0.01

    inputs = numpy.asfarray(img_data)

    outputs = n.query(inputs)
    print outputs
    label = numpy.argmax(outputs)

    return label
######################## FINE FUNZIONE ANALIZZA IMMAGINE ########################

############################## INIZIO PROGRAMMA ##################################

start = time.time()

# nome del file contenente la rete neurale serializzata
ser_file = "nnLR0_1E5h200.bin"
n = None

if ser_file == None or not os.path.exists(ser_file):
    # numero di nodi d'ingresso, centrali e di output
    input_nodes = 784
    hidden_nodes = 200
    output_nodes = 10

    # soglia di apprendimento è di 0.1
    learning_rate = 0.1

    # creo istanza della rete neurale
    n = neuralNetwork(input_nodes, hidden_nodes, output_nodes, learning_rate)

    # carica la lista dei file di train
    training_data_file = open(file_mnist_name,'r')
    training_data_list = training_data_file.readlines()
    training_data_file.close()

    # epochs è il numero di volte che i dati di addestramneto sono usati per
l'addestramento
    epochs = 5

    for e in range(epochs):
        # fase di apprendimento della rete neurale
```

```
        for record in training_data_list:
            # divide ogni numero appena incontra la ","
            all_values = record.split(',')

            # riscala gli input da 0-255 a 0.01 a 0.99
            inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01

            # crea una lista composta da 10 valori che corrispondo ai target,
numeri da riconoscere da 0 a 9
            targets = numpy.zeros(output_nodes) + 0.01

            # all_values[0] contiene il numero da identificare
            # se all_values[0] è 5 il programma indica quale numero bisogna
riconoscere
            # targets sarà [0.01, 0.01, 0.01, 0.01, 0.01, 0.99, ....]
            # il 0.99 si trova al sesto posto perciò corrisponde al numero 5
            targets[int(all_values[0])] = 0.99
            n.train(inputs, targets)
            pass

else:
    # carica la rete neurale dal file in cui era stata salvata(serilizzata)
precedentemente
    filer = open(ser_file, 'rb')
    n = pickle.load(filer)
    filer.close()

def main(ip, port):
    # Main del programma
    # MyBroker permette di abilitare I moduli NAOqi
    myBroker = ALBroker("myBroker",
       "0.0.0.0",   # listen to anyone
       0,           # find a free port and use it
       ip,          # parent broker IP
       port)        # parent broker port

    global ReactToTouch
    ReactToTouch = ReactToTouch("ReactToTouch")

    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print "Interrupted by user, shutting down"
        myBroker.shutdown()
        sys.exit(0)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", type=str, default="172.31.1.34",
                        help="Robot ip address")
    parser.add_argument("--port", type=int, default=9559,
                        help="Robot port number")
    args = parser.parse_args()
    main(args.ip, args.port)
```