UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Ingegneria Elettrica Elettronica Informatica

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Reti Neurali: studio del processo di apprendimento sui microcontrollori

Anno accademico [2020-2021]

Candidato: Andrea Giuffrida

Relatore: Paolo Pietro Arena

INDICE

INTRODUZIONE

ABSTRACT. L'intelligenza artificiale al giorno d'oggi è sfruttata in molti campi. In particolare vengono utilizzate le reti neurali per l'apprendimento di pattern complessi. Tali pattern possono provenire da misurazioni effettuate da sensori tramite microcontrollore. Può risultare quindi significativo apprendere tali pattern direttamente sui microcontrollori. In questo lavoro viene presentato una prima introduzione sulle reti neurali, seguita da uno studio sul processo di apprendimento della funzione non lineare XOR e successivamente di dati più complessi, prodotti da un modello che simula il comportamento del neurone biologico, tramite le funzioni di FitzHugh-Nagumo.

L'intelligenza artificiale è la branca di computer science che studia la progettazione, lo sviluppo e la realizzazione di sistemi hardware e software capaci di simulare le capacità caratteristiche dell'essere umano. Tali sistemi, quindi, sono in grado di perseguire autonomamente una finalità definita, prendendo delle decisioni che, in passato erano esclusivamente affidate agli esseri umani. A differenza dei software tradizionali, un sistema di IA non si basa sulla programmazione (cioè sul lavoro di sviluppatori che scrivono il codice di funzionamento del sistema) ma su tecniche di apprendimento: vengono cioè definiti degli algoritmi che elaborano un'enorme quantità di dati dai quali è il sistema stesso che deve derivare le proprie capacità di comprensione e ragionamento. Un sistema di IA tenta di simulare le capacità cognitive dell'uomo come la comprensione e l'elaborazione del linguaggio naturale e delle immagini, l'apprendimento, il ragionamento e la capacità di pianificazione e l'interazione con persone, macchine e ambiente.

Al giorno d'oggi, l'intelligenza artificiale trova applicazione in molti ambiti, come medicina, economia e macchine industriali, a volte anche molto distanti dall'informatica, grazie alla sua duttilità e flessibilità. Il suo utilizzo è stato adottato

per il riconoscimento di immagini (Krizhevsky Alex 2017), il riconoscimento vocale (Mikolov Tomas 2011), per l'analisi dei dati di un acceleratore di particelle (Ciodaro T. 2012) e la ricostruzione di circuiti nervosi (Helmstaedter Moritz 2013). Spesso si riccorre all'AI come unica opzione possibile per risolvere un problema complesso. Nello specifico, risulta infatti, particolarmente utile, nei problemi di approssimazione di funzioni (fitting di curve), la classificazione in base a delle "feature" e alla identificazione di "pattern".

Vi è poi una particolare branca dell'intelligenza artificiale, ovvero quella legata alle reti neurali. Le reti neurali possiedono un'elevata velocità di elaborazione e hanno la capacità di imparare la soluzione da un determinato insieme di esempi. Il modello su cui si basano le reti neurali prendono spunto dai meccanismi di elaborazione dell'informazione nel sistema nervoso del cervello umano.

Le reti neurali artificiali sono anche denominate "reti neurali", "sistemi di elaborazione distribuiti paralleli" e "sistemi connessionisti". Affinché un sistema informatico possa essere definito "rete neurale artificifiale", è necessario che abbia una struttura a "grafo diretto" in cui i nodi eseguono alcuni semplici calcoli (Dongare A. D. 2012, 189-194).

I metodi di deep learning sono metodi di apprendimento con più livelli di rappresentazione, ottenuti componendo moduli semplici ma non lineari che trasformano ciascuno la rappresentazione di un livello (a partire dall'input grezzo) in una rappresentazione di un livello più alto, leggermente più astratto. Con la composizione di abbastanza tali trasformazioni, è possibile apprendere funzioni molto complesse. (LeCun Yann 2015)

Ciò che rende il deep learning "profondo" è il fatto che vi sono più "livelli" di neuroni di vari pesi che aiutano una rete neurale a prendere le proprie decisioni. Il deep learning può essere suddiviso in due fasi: addestramento e inferenza.

Durante la fase di addestramento, viene definito il numero di neuroni e i livelli che comprenderà la rete neurale che verrà poi esposta ai dati di addestramento. Con questi dati, la rete neurale apprende da sola che cosa è "giusto" o "sbagliato".

Purtroppo però, il training di un modello di deep learning, è molto costoso sia dal punto di vista spaziale che da quello computazionale, a causa dei milioni di parametri che devono essere perfezionati iterativamente in più periodi di tempo. L'inferenza invece è il processo di valutazione di nuove immagini utilizzando una rete neurale per prendere delle decisioni. Inoltre, anche l'inferenza è computazionalmente costosa a causa della dimensionalità potenzialmente elevata dei dati di input (ad esempio, un'immagine ad alta risoluzione) e dei milioni di calcoli che devono essere eseguiti sui dati di input. L'elevata precisione e l'elevato consumo di risorse definiscono le caratteristiche del deep learning.

Per soddisfare i requisiti computazionali del deep learning, si utilizza comunemente un approccio che consiste nello sfruttare il cloud computing. Mentre, per utilizzare le risorse cloud, i dati devono essere spostati dalla posizione dell'origine dati sul perimetro della rete [ad esempio, da smartphone e sensori Internet of Things (IoT)] a una posizione centralizzata nel cloud. L'edge computing è una soluzione praticabile per soddisfare le sfide di latenza, scalabilità e privacy (Chen Jiasi 2019).

Tra le possibili applicazioni delle reti neurali, quella nell'ambito dei microcontrollori al giorno d'oggi è di grande interesse.

Il training su PC con la successiva installazione su microcontrollore è stata utilizzata per la linearizzarione di sensori (Medrano-Marques N. J. 2000), il keyword spotting (es. "ok google", "alexa") (Zhang Yundong 2018), nell'automotive (Crocioni Giulia 2021). Vi sono anche nuovi framework appositi per l'ottimizzazione delle reti neurali, affinché esse rientrino nei limiti di memoria e CPU dei microcontrollori. Tra quiesti i più importanti sono TinyML (Banbury Colby 2021), Tensor Flow Lite Micro(TFLM). In una ricerca ne è stato creato anche uno che effettua il learning online tramite TinyML, TinyOL (Haoyu Ren 2021).

Alcuni studi hanno effettuato il learning direttamente a bordo del microcontrollore: con un'architettura neuron-by-neuron (Cotton Nicholas J. 2011), un training di semplici funzioni (OR, AND, XOR e XNOR) su un Arduino Pro Mini (Parker Gary

2016) e un’approssimazione del segnale di un Elettrocardiogramma (Bogolovskii Ivan A. 2020).

Il vantaggio di avere un microcontrollore che effettui il learning direttamente a bordo è quello di essere in grado di gestire autonomamente i dati, ad esempio interpolando i dati mancanti dei sensori, e riadattandosi periodicamente ai cambiamenti. Tutto ciò potrebbe risultare fondamentale in un futuro in cui l’IOT (Internet of Things) avrà un ruolo sempre più preponderante nella società.

L’obbiettivo posto è quindi quello di implementare una rete neurale ed allenarla direttamente su un microcontrollore, in particolare su un ELEGOO UNO R3. Ciò è stato effettuato utilizzando una rete perceptrone multistrato (Multi Layer Perceptron, MLP), che nonostante sia una delle reti più semplici da eseguire, riesce comunque a risolvere problemi anche complessi. Si è quindi cercato di far approssimare alla rete la funzione XOR che è una delle funzioni non lineari più semplici, utilizzando successivamente dei dati prodotti da un modello che sfrutta le equazioni di FitzHugh-Nagumo.

1. Reti Neurali

Negli anni sono stati sviluppati numerosi modelli di reti neurali per risolvere problemi di diversa natura, tuttavia qui saranno presentati principalmente due modelli conosciuti come “multilayer perceptron” e “reti fully connected”, in quanto sono alla base della maggior parte delle applicazioni pratiche.

1.1. Reti Biologiche e Reti Neurali Artificiali

Una rete neurale, sia essa biologica o artificiale, è composta da un vasto numero di unità semplici, i neuroni, che ricevono e trasmettono segnali gli uni agli altri.

In biologia, i neuroni possono essere considerati celle cariche elettricamente. Ne esistono varie forme, anche se la maggior parte contiene i dentriti, costituiti da ramificazioni che trasmettono l’imput del segnale ai neuroni, e gli assoni che invece gestiscono l’uscita del segnale. In funzione dei segnali in entrata, il neurone si attiva ed emette un segnale che deve essere ricevuto dagli altri neuroni. Ciascun assone può essere collegato a uno o più dendriti in corrispondenza di intersezioni dette sinapsi (Figura 1). Ogni neurone è tipicamente connesso ad un migliaio di altri neuroni e, di conseguenza, il numero di sinapsi nel cervello supera 10^{14}.

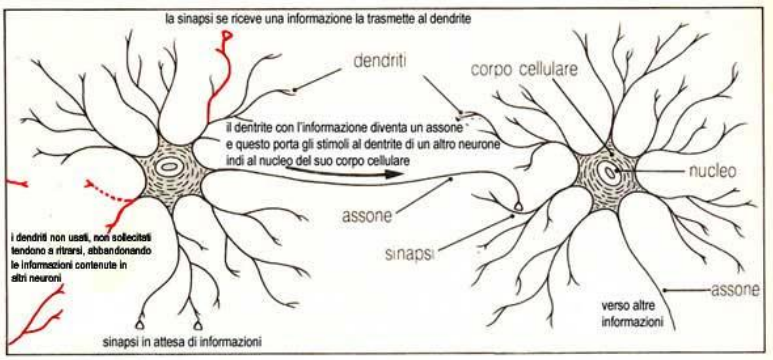

Figura 1.1 Schema di trasmissione del segnale tra 2 neuroni biologici

Il neurone normalmente si può trovare in uno stato di riposo oppure può essere attivato tramite un impulso elettrico, che viene trasportato lungo l'assone. Quando il segnale raggiunge la sinapsi avviene il ralascio di sostanze chimiche che entrano in altri neuroni. Il tipo di sinapsi regola la quantità di sostanze che aumentano o diminuiscono la probabilità che il successivo neurone si attivi. Ad ogni sinapsi è associato un peso che ne determina il tipo e l'ampiezza dell'effetto eccitatore o inibitore. Quindi, esemplificando, ogni neurone effettua una somma pesata degli ingressi provenienti dagli altri neuroni e, se questa somma supera una certa soglia, il neurone si attiva. Sulla base di questo meccanismo, sono state ideate le reti neurali artificiali.

Il primo modello di reti neronali artificiali che cercò di imitare il modello biologico si basa sul modello di McCulloch-Pitts.

1.2. Il perceptrone e il neurone

Il perceptrone fu utilizzato per la prima volta nei modelli di Warren Mc Culloch e Walter Pitts e Frank Rosenblatt, come semplice classificatore lineare binario, in grado di apprendere efficacemente la regola necessaria per riconoscere due classi di input diverse e linearmente separabili.

Esso infatti riceve in input N valori, li moltiplica con dei pesi, attraverso una matrice ed infine ne somma i risultati. Se ciò che si ottiene è maggiore di una certa soglia (threshold) allora il perceptrone diventa attivo e viceversa. Questo si può esprimere secondo l'Eq. 1.1:

$$\text{output} = \begin{cases} 0 & \text{se } \sum_{j=0}^{N-1} x_j \cdot w_j \leq threshold \\ 1 & \text{se } \sum_{j=0}^{N-1} x_j \cdot w_j > threshold \end{cases}$$

Equazione 1.1

Dove x_j indica il j-esimo input e w_j il j-esimo peso. Il *Threshold* può anche essere spostato a sinistra della disequazione e viene tipicamente chiamato bias, indicato con b.

Da un modello come quello riportato in equazione 1.2 si ottiene un'output a gradino, che risulta quindi non derivabile e con il quale risulterebbe difficile lavorare. Diversamente, quindi si preferisce utilizzare un altro tipo di modello, in cui viene applicata una funzione derivabile, detta "funzione di attivazione".

$$\text{output} = \begin{cases} 0 & \text{se } b + \sum_{j=0}^{N-1} x_j \cdot w_j \leq 0 \\ 1 & \text{se } b + \sum_{j=0}^{N-1} x_j \cdot w_j > 0 \end{cases}$$

Equazione 1.2

Il risultato ottenuto porta ad un modello che simula il comportamento di un neurone (Eq. 1.3)

$$\begin{cases} z = b + \sum_{j=0}^{N-1} x_j \cdot w_j \\ \texttt{output} = \sigma(z) \end{cases}$$

Equazione 1.3

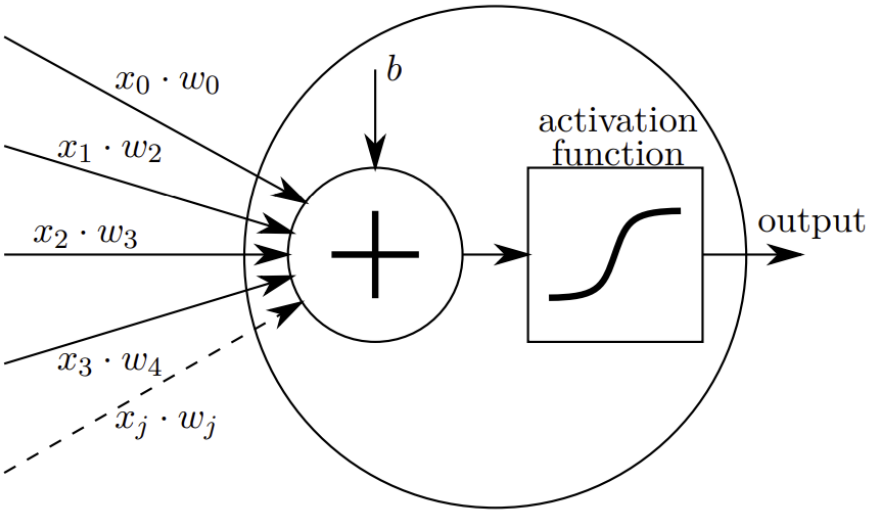

Figura 1.2 Schema di funzionamento del neurone

La funzione di attivazione più utilizzata è la funzione sigmoide (Eq.1.4, Figura 1.3). Questa funzione ha un valore prossimo a 1, per valori di input maggiori di 0, e un valore prossimo a 0 per valori di input minori di 0, approssimando quindi la funzione gradino.

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

Equazione 1.4

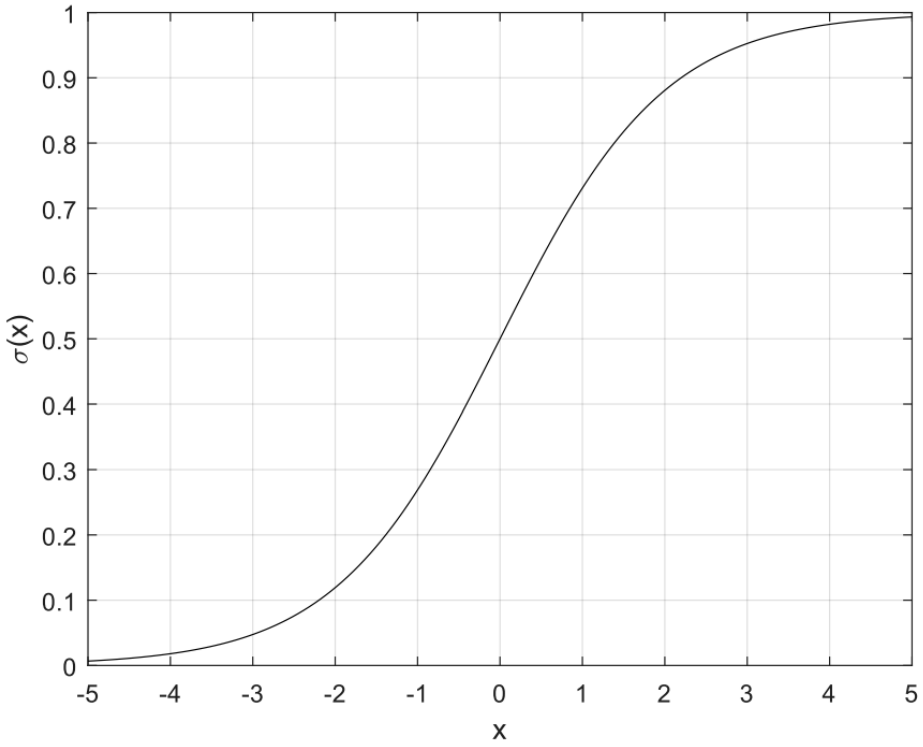

Figura 1.3 Grafico della funzione sigmoide

1.3. Reti fully connected

Le reti neurali sono organizzate in layer (livelli), inoltre per ogni rete troviamo sempre il layer di input, di output e layer nascosti (hidden). L'input layer si occuperà di trasmettere gli input al primo layer nascosto, senza processare i dati.

Le reti fully connected sono delle reti in cui tutti i layer sono di tipo fully connected. Questi layer sono i più comuni e semplici possibili, e connettono ogni neurone dello strato precedente con ogni neurone dello strato successivo. Ad ogni connessione verrà associato un peso, che è il valore per cui è moltiplicato l'input proveniente dal layer precedente, come indicato nello in Figura 1.4.

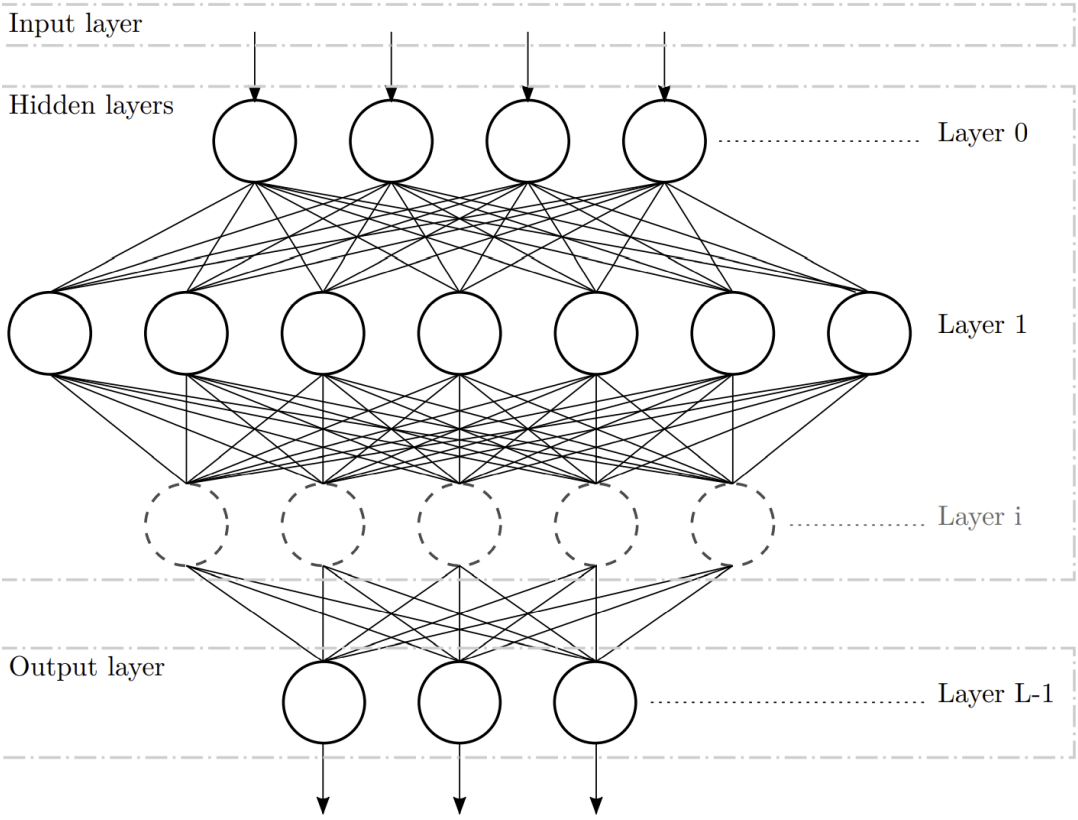

Figura 1.4 Schema di una rete fully connected

I vari pesi vengono aggiornati in ogni iterazione durante la cosiddetta fase di training. Durante questa fase gli input vengono propagati in avanti (feedforward) seguendo le regole illustrate prima per ogni neurone, fino a raggiungere l'ultimo layer, producento l'output della rete. A questo punto l'output viene confrontato con il valore atteso (target), fornito alla rete insieme ai dati di input.

A questo punto tramite una funzione detta di errore (error) o di costo (cost) viene calcolato l'errore che la rete produce nello stimare l'output.

Una delle funzioni di costo più utilizzate è quella riportata in Eq. 1.5

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\, log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\, log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

Equazione 1.5

Dove:

- m è il numero degli esempi di training
- K è il numero dei neuroni di output
- Y è la matrice
- $y_k^{(i)}$ è l'i-esimo output di training (target) per il k-esimo neurone di output
- $x^{(i)}$ è l'i-esimoinput di training (un vettore)
- $h_\Theta(x^{(i)})$ è il valore dell'ipotesi all'output k, con matrice dei pesi Θ, e input di training i

Lo scopo è quello di minimizzare l'errore per rendere l'output quanto più vicino al target.

A questo punto l'errore può essere visualizzato come una curva in uno spazio n dimensionale. Per esempio con n uguale a 2 la funzione errore apparirà graficamente come qualcosa simile a questo (Figura 1.5).

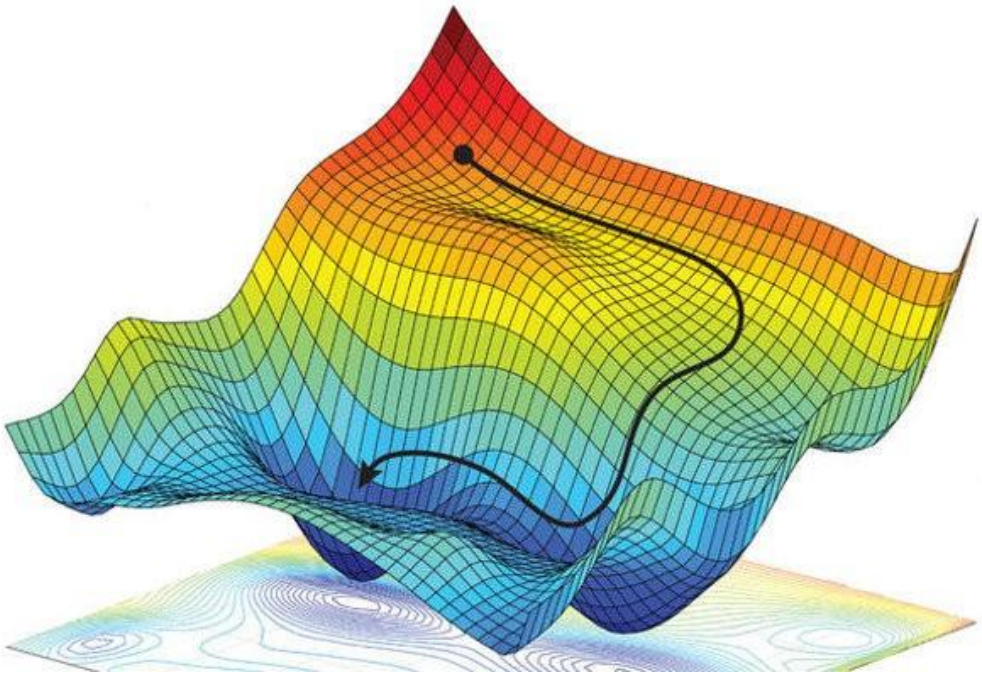

Figura 1.5 Grafico della funzione errore

Ciò a cui vogliamo arrivare è il minimo globale della funzione che minimizzerebbe il nostro errore. La direzione in cui la pendenza è massima è calcolata dal gradiente della funzione in quel punto (Eq. 1.6).

$$\nabla_w L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, ..., \frac{\partial L}{\partial w_m}]$$

Equazione 1.6

Il gradiente moltiplicato per il cosidetto learning rate η da come risultato la quantità che deve essere sommata al peso (Eq. 1.7).

$$\Delta w = \eta(-\nabla_w L\ )$$

Equazione 1.7

Questa quantità verrà poi propagata all'indietro nel layer precedente, attraverso una moltiplicazione con la matrice dei pesi. Qui verranno effettuati gli stessi passaggi andando a ricalcolare gradiente e Δw per i pesi del layer, e cosi via

Il processo verrà ripetuto, facendo quindi in modo che l'errore in output si propaghi all'indietro attraverso tutta la rete (Figura 1.6).

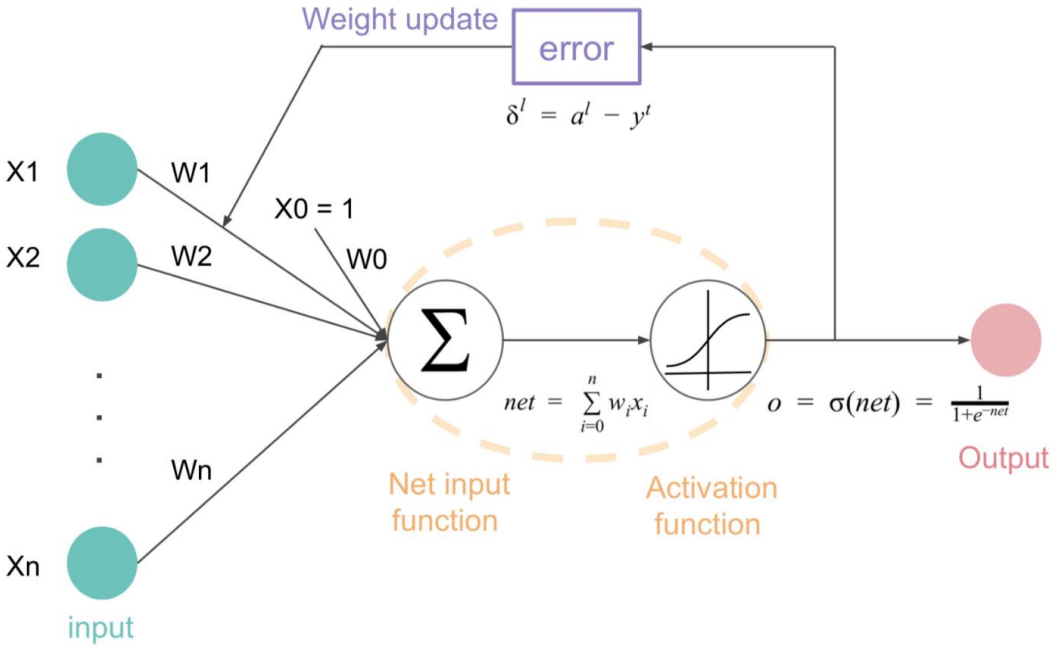

Figura 1.6 Schema del funzionamento di una rete neurale

Il procedimento si ripete finché l'errore non raggiunge una certa soglia, fissata come parametro. Durante l'allenamento o addestramento della rete si verificano molti problemi che devono essere affrontati, ma questi esulano dallo scopo del presente lavoro di tesi.

1.4. ELEGOO UNO R3

Il microprocessore ELEGOO UNO R3 prende ispirazione dall'Arduino UNO Rev3, uno dei processori più famosi e diffusi dell'azienda Arduino. Viene utilizzato in ambienti dominati dall'I/O e quindi risulta ampiamente sfruttato nell'ambito dell'IoT (Internet of Things).

Nella Tabella 1.1 vengono riportate le caratteristiche principali dell'ELEGOO UNO R3.

MICROCONTROLLORE	ATmega328P
TENSIONE DI LAVORO	5V
TENSIONE DI INPUT(RACCOMANDATA)	7-12V
TENSIONE DI INPUT(LIMITE)	6-20V
PIN DI I/O DIGITALI	14 (dei quali 6 con PWM)
PIN DI I/O CON PWM	6
PIN DI INPUT ANALOGIGI	6
CORRENTE DC PER PIN DI I/O	20 mA
CORRENTE DC PIN A 3.3V	50 mA
MEMORIA FLASH	32 KB (ATmega328P) dei quali 0.5 KB usati dal bootloader
SRAM	2 KB (ATmega328P)

EEPROM	1 KB (ATmega328P)
VELOCITÀ DI CLOCK	16 MHz
LED_INTEGRATO	13
LUNGHEZZA	68.6 mm
LARGHEZZA	53.4 mm
PESO	25 g

Tabella 1.1 Caratteristiche principali ELEGOO UNO R3

2. Implementazione rete e performance

Il codice sull'ELEGOO UNO R3 è stato implementato mediante il codice che sarà illustrato nel seguente capitolo e saranno riportatate le performance misurate.

2.1. Codice di base

Il codice di base utilizzato per costruire la rete è stato preso dal sito "the-diy-life.com" e l'autore è Ralph Heymsfeld, in un articolo pubblicato il 28/06/2018.

Il codice completo è riportato in Appendice A

L'obiettivo della rete è quello di riuscire a mappare i numeri che appaiono su un display LCD (Figura 2.1) tramite l'accensione e lo spegnimento dei vari segmenti, in numeri binari.

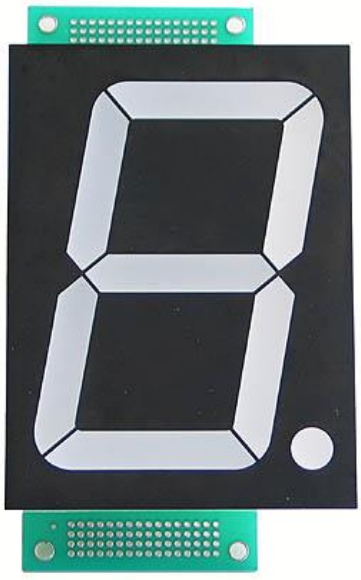

Figura 2.1 Display LCD a 7 segmenti

In breve il codice esegue le seguenti istruzioni:

- Crea gli array e assegna pesi casuali.

```
for( i = 0 ; i < HiddenNodes ; i++ ) {

    for( j = 0 ; j <= InputNodes ; j++ ) {

      ChangeHiddenWeights[j][i] = 0.0 ;

      Rando = float(random(100))/100;

      HiddenWeights[j][i] = 2.0 * ( Rando - 0.5 ) *
InitialWeightMax ;

    }

  }

for( i = 0 ; i < OutputNodes ; i ++ ) {

    for( j = 0 ; j <= HiddenNodes ; j++ ) {

      ChangeOutputWeights[j][i] = 0.0 ;

      Rando = float(random(100))/100;

      OutputWeights[j][i] = 2.0 * ( Rando - 0.5 ) *
InitialWeightMax ;

    }

  }
```

- Avvia un ciclo che attraversa ogni elemento dei dati di addestramento.

```
for( p = 0 ; p < PatternCount ; p++) {
```

- Randomizza l'ordine in cui i dati di training vengono eseguiti in ogni iterazione per garantire che non si verifichi la convergenza sui minimi locali.

```
for( q = 0 ; q < PatternCount ; q++ ) {

      p = RandomizedIndex[q];
```

- Forisce i dati alla rete calcolando l'attivazione dei nodi del layer nascosto, i nodi del layer di output e l'errore.

```
for( i = 0 ; i < OutputNodes ; i++ ) {

        Accum = OutputWeights[HiddenNodes][i] ;

        for( j = 0 ; j < HiddenNodes ; j++ ) {

          Accum += Hidden[j] * OutputWeights[j][i] ;

        }

        Output[i] = 1.0/(1.0 + exp(-Accum)) ;

        OutputDelta[i] = (Target[p][i] - Output[i]) * Output[i] *
(1.0 - Output[i]) ;

        Error += 0.5 * (Target[p][i] - Output[i]) * (Target[p][i]
- Output[i]) ;

      }
```

- Propaga all'indietro gli errori al livello nascosto.

```
for( i = 0 ; i < HiddenNodes ; i++ ) {

        Accum = 0.0 ;

        for( j = 0 ; j < OutputNodes ; j++ ) {
```

```
          Accum += OutputWeights[i][j] * OutputDelta[j] ;

        }

        HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]) ;

      }
```

- Aggiorna i pesi associati.

```
for( i = 0 ; i < HiddenNodes ; i++ ) {

        ChangeHiddenWeights[InputNodes][i] = LearningRate *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[InputNodes][i] ;

        HiddenWeights[InputNodes][i] +=
ChangeHiddenWeights[InputNodes][i] ;

        for( j = 0 ; j < InputNodes ; j++ ) {

          ChangeHiddenWeights[j][i] = LearningRate * Input[p][j] *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[j][i];

          HiddenWeights[j][i] += ChangeHiddenWeights[j][i] ;

        }

      }

for( i = 0 ; i < OutputNodes ; i ++ ) {

        ChangeOutputWeights[HiddenNodes][i] = LearningRate *
OutputDelta[i] + Momentum * ChangeOutputWeights[HiddenNodes][i] ;

        OutputWeights[HiddenNodes][i] +=
ChangeOutputWeights[HiddenNodes][i] ;

        for( j = 0 ; j < HiddenNodes ; j++ ) {

          ChangeOutputWeights[j][i] = LearningRate * Hidden[j] *
OutputDelta[i] + Momentum * ChangeOutputWeights[j][i] ;

          OutputWeights[j][i] += ChangeOutputWeights[j][i] ;
```

- Confronta l'errore con la soglia e decide se eseguire un altro ciclo o se il training è completo.

```
if( Error < Success ) break ;
```

- Mostra un campione dei dati di addestramento al monitor seriale ogni mille cicli.

2.2. XOR

Successivamente sono stati modificati i dati, affinchè la rete imparasse l'operazione di OR esclusivo (XOR) (Tabella 2.1).

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 2.1 Tabella dello XOR

Lo XOR rimane un problema ostico per le reti neurali, poiché nonostante abbia un pattern molto semplice è un tipico esempio di indivisibilità lineare (linear un-division). Una rete neurale a perceptrone con un singolo layer infatti cercherebbe di unire i 4 punti dello XOR con una linea, problema ovviamente irrisolvibile.

Alcune soluzioni trovate sono: un perceptrone multi-layer, un perceptrone con link funzionale e un perceptrone single-layer migliorato con una funzione quadratica (Zhao Yanling 2002).

Sotto sono state riportate le righe di codice modificate:

```
const byte Input[PatternCount][InputNodes] = {

    {0, 0},

    {0, 1},

    {1, 0},

    {1, 1}

};



const byte Target[PatternCount][OutputNodes] = {

  {0},

  {1},

  {1},

  {0}

};
```

2.3. Performance

Le performance sono state determinate misurando il tempo ed il numero di cicli impiegati dalla rete per convergere, eseguendo 10 misurazioni e calcolando poi la media. Si fa presente che le misurazioni del tempo sono soggette ad errore, poiché misurate tramite un cronometro attivato a mano. Inoltre, si è deciso di studiare l'andamento dell'errore durante la fase di training, prendendo il valore dell'errore ogni 100 o 50 cicli a seconda dei casi.

I parametri che sono stati modificati per trovare la configurazione migliore della rete sono il numero di neuroni del layer nascosto e la soglia di interruzione dell'algoritmo.

Innazitutto sono state misurate le performance senza modificare i parametri, lasciando quindi 8 neuroni nel layer nascosto e fissando a 0.0004 la soglia (Tabella 2.2).

Tempo	Cicli
39.83	2158
38.34	2102
38.11	2080
45.80	2531
44.56	2481
30.25	1606
34.89	1886
39.06	2123
35.66	2019
33.17	1808
Media: 37.97	**Media: 2079**

Tabella 2.2 Prestazioni con 8 neuroni nel layer nascosto e soglia 0.0004.

Successivamente si è deciso di ridurre il numero di neuroni nel layer nascosto. I valori "NC" indicano che in quell'iterazione l'algoritmo non è riuscito a convergere (Tabella 2.3).

Tempo	Cicli
17,32	2585
16,24	2471
16,72	2570
17,04	2625
NC	
16,19	2470
16,25	2460
16,14	2481
16,78	2608
16,26	2491
Media: 16,5	**Media: 2529**

Tabella 2.3 Prestazioni con 2 neuroni nel layer nascosto e soglia 0.0004.

I dati sulla propagazione dell'errore sono riportati in tabella Tabella 2.4 e mostrati in Figura 2.2:

N ciclo	Errore
1	0.55391
100	0.51727
200	0.44027
300	0.09483

400	0.01082
500	0.00521
600	0.00337
700	0.00248
800	0.00195
900	0.00160
1000	0.00136
1100	0.00117
1200	0.00104
1300	0.00093
1400	0.00084
1500	0.00076
1600	0.00070
1700	0.00065
1800	0.00060
1900	0.00056
2000	0.00053
2100	0.00050
2200	0.00047
2300	0.00044
2400	0.00042

2500	0.00040
2509	0.00040

Tabella 2.4 Andamento dell'errore con 2 neuroni nel layer nascosto e soglia 0.0004.

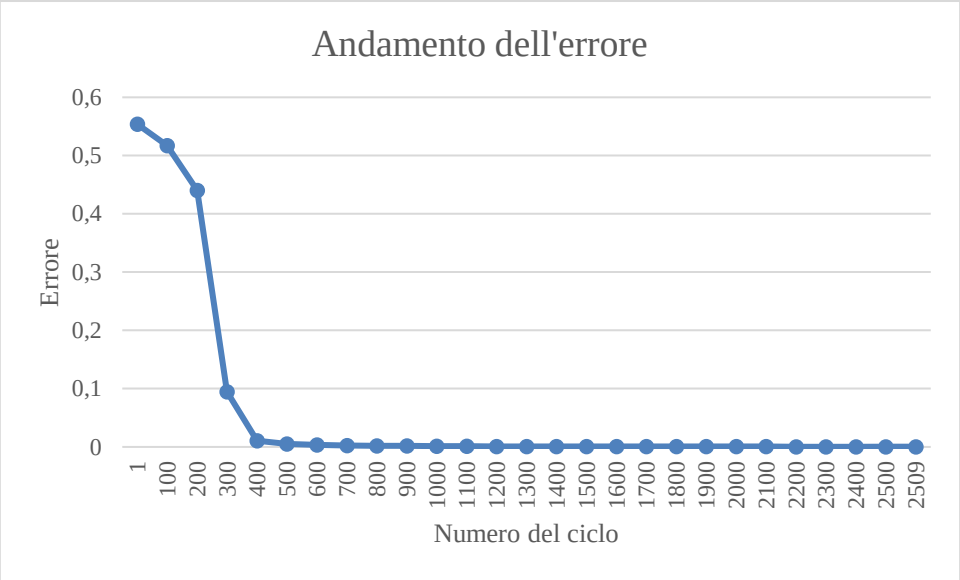

Figura 2.2 Grafico dell'andamento dell'errore con 2 neuroni nel layer nascosto e soglia 0.0004

Si nota in particolare che il tempo di convergenza dell'algoritmo è notevolmente diminuito ma che a volte non riesca a convergere. Inoltre l'algoritmo arriva a un errore accettabile già intorno al ciclo numero 400. Per questo motivo si è deciso di abbassare la soglia di successo a 0.002 (Tabella 2.5).

Tempo	Cicli
8,18	869
6,61	748
6,63	796
6,72	865

NC	
NC	
7,08	800
6,63	788
6,6	759
NC	
Media: 6,92	**Media: 803,6**

Tabella 2.5 Prestazioni con 2 neuroni nel layer nascosto e soglia 0.002.

Nella Tabella 2.6 e in Figura 2.3 è riportato l'andamento dell'errore:

N Ciclo	Errore
1	0,51900
50	0,50988
100	0,51186
150	0,53876
200	0,50973
250	0,50384
300	0,43474
350	0,36345
400	0,35248
450	0,34624

500	0,33085
550	0,10335
600	0,02081
650	0,01051
700	0,00686
750	0,00506
800	0,00397
850	0,00328
900	0,00278
950	0,00241
1000	0,00212
1026	0,00200

Tabella 2.6 Andamento dell'errore con 2 neuroni nel layer nascosto e soglia 0.002.

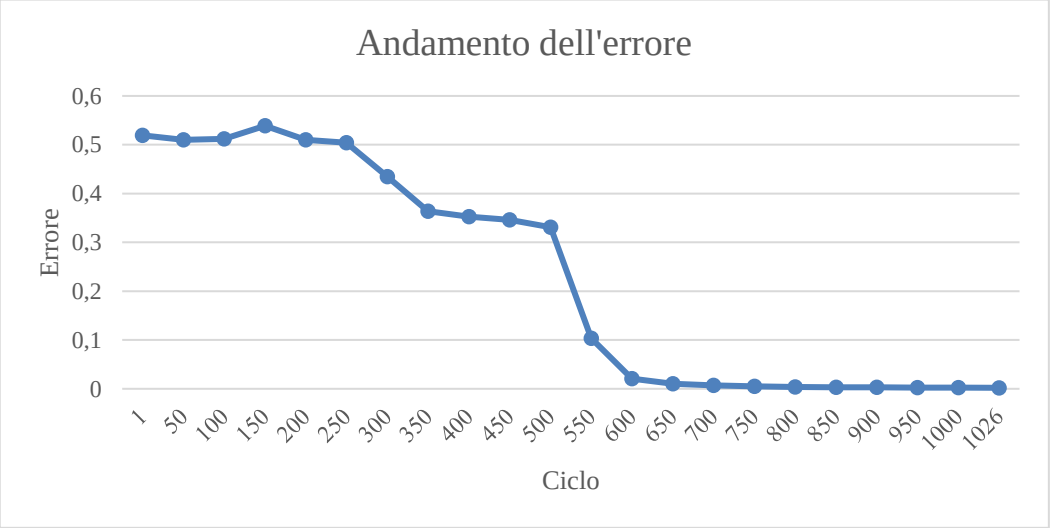

Figura 2.3 Grafico dell'andamento dell'errore con 2 neuroni nel layer nascosto e soglia 0.002.

Si può così osservare che le prestazioni sono migliorate ancora, ma continuano a persistere i problemi di convergenza. Si è quindi deciso di fare un tentativo con una rete con 4 neuroni, mantenendo la soglia 0.002 (Tabella 2.7).

Tempo	Ciclo
8,87	719
10,08	848
8,58	686
8,57	650
8,66	691
8,78	702
9,53	805
9,82	804
8,59	686
9,61	804
Media: 9,109	**Media: 739,5**

Tabella 2.7 Prestazioni con 4 neuroni nel layer nascosto e soglia 0.002.

Di seguito invece si riporta la propagazione dell'errore (Tabella 2.8, Figura 2.4)

N Ciclo	Errore
1	0,55386
50	0,51534
100	0,52408

150	0,23337
200	0,03327
250	0,0138
300	0,00834
350	0,00589
400	0,00452
450	0,00365
500	0,00305
550	0,00261
600	0,00229
650	0,00203
656	0,002

Tabella 2.8 Andamento dell'errore con 4 neuroni nel layer nascosto e soglia 0.002.

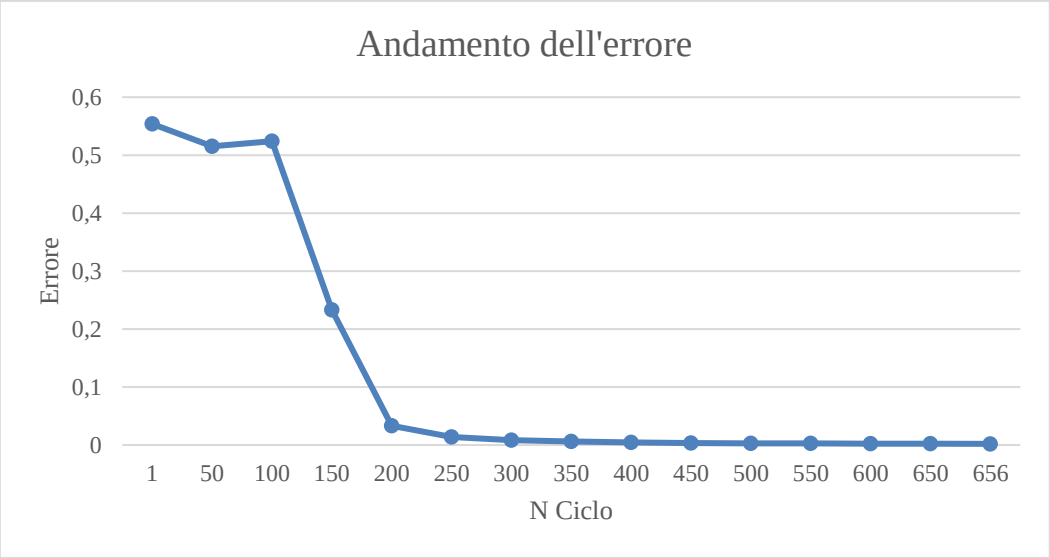

Figura 2.4 Grafico dell'andamento dell'errore con 4 neuroni nel layer nascosto e soglia 0.002.

Con 4 neuroni l'algoritmo converge per ogni iterazione, e le performance non si discostano eccessivamente da quelle della rete con 2 neuroni. Pertanto questa scelta di parametri per la rete semberebbe la soluzione ottimale.

3. Dati dal sistema FitzHugh-Nagumo

Successivamente, si è provato ad utilizzare un modello ridotto bidimensionale. In particolare per i dati di input si è utilizzato il “FitzHugh-Nagumo system” Questo sistema simula il comportamento lento-veloce di un neurone biologico.

3.1. Equazioni di FitzHugh-Nagumo

Nonostante siano stati riportati differenti modelli descritti da equazioni accurate per la propagazione dell'impulso, tuttavia la descrizione di tali equazioni non rappresenta tutti i tipi di neurone. Per questo motivo Fitzhugh, nel 1961, ha riportato un sistema di equazioni semplificato rispetto alle equazioni di Hodgkin-Huxley (Nelson Mark 1998) precedentemente utilizzate per la conduzione dei nervi. L’idea era quella di utilizzare due scale temporali, supponendo che una variabile cambiasse su una scala veloce, mentre l'altra variabile su una dinamica lenta.

Le equazioni utilizzate nel modello per la produzione dei dati sono (Eq. 3.1):

$$\begin{cases} \epsilon \dot{v} = f(v) - \omega - \omega_0 \\ \dot{w} = v - \gamma w - v_0 \end{cases}$$

Equazione 3.1

Il modello è quindi un sistema del secondo ordine dove la funzione non lineare ha la forma (Eq. 3.2):

$$f = Av(1-\alpha)(1-v)$$

Equazione 3.2

3.2. Generazione dei dati

Nel presente lavoro di tesi, i dati utilizzati sono stati generati tramite l'integrazione di Eulero, con uno step di tempo dt = 0.1. L'intero segnale dura 3000 campioni, che corrispondono a circa quattro periodi e mezzo di oscillazione. Inoltre, 101 diverse condizioni iniziali sono state utilizzate per ottenere una vasta gamma di casi. Le due variabili **υ** e **ω** rappresentano gli input della rete neurale, mentre le stesse variabili ritardate di un campione, corrispondono alle uscite del segnale. Le diverse serie temporali sono state organizzate in un unico array di celle distinguendo le 2 variabili. Le 2 uscite del sistema sono mostrate nella figura seguente (Figura 3.1):

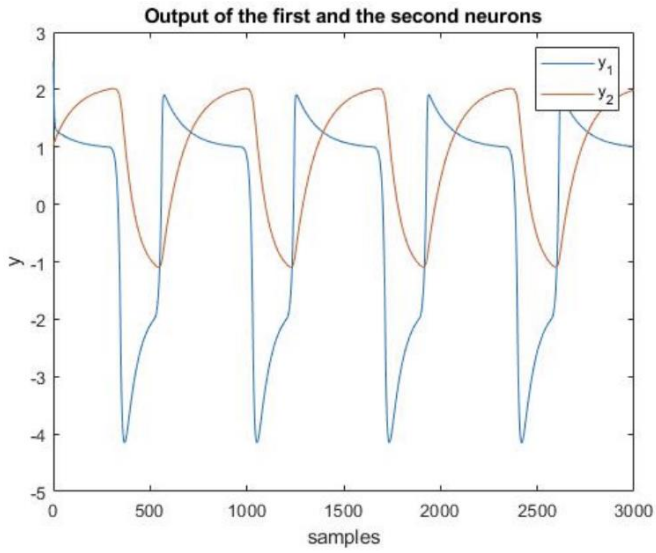

Figura 3.1 Uscite della prima e della seconda variabile di stato

3.3. Prestazioni Rete Neurale

Poiché la memoria del microcontrollore è limitata e sicuramente di molto inferiore alle dimensioni tipiche delle memorie dei PC a cui si è abituati, si è deciso di utilizzare esclusivamente i primi 700 campioni. I dati sono stati salvati nella memoria flash dell'ELEGOO (attraverso la parola chiave "PROGMEM") invece che all'interno della SRAM, la quale non sarebbe stata in grado di contenerli.

Le matrici di dati, attraverso l'ausilio del tool "MATLAB", sono state trasposte e ridotte, (in modo tale da avere le dimesioni di 700 x 2 ciascuna) e normalizzate, per avere i valori compresi tra 0 e 1 e poter essere trattati dalla rete. Infine sono state esportate nel formato csv, formato più facilmente sfruttabile dall'ambiente di Arduino. I dati per poter essere riconosciuti dal microcontrollore dovevano essere racchiusi all'interno di parentesi graffe. Per tale motivo è stato utilizzato un "Notepad++" e la sua funzione di "trova e sostituisci", grazie alla quale si è riusciti ad avere i dati nel formato desiderato.

Infine il file per il microcontrollore è stato adattato per leggere i dati:

- PatternCount è stato impostato a 700.
- InputNodes è stato impostato a 2.
- OutputNodes è stato impostato a 2.
- Sono state rimosse le operazioni che randomizzavano l'ordine dei dati, poiché essi erano già in ordine randomico.
- Per accedere alle matrici dei dati si utilizza la funzione "pgm_read_float()" che permette di leggere dalla memoria flash.
- È stato aggiunto del codice per calcolare l'errore sui dati di validazione.
- Sono state introdotte variabili per il calcolo del tempo di esecuzione, che sfruttano la funzione "millis()".
- È stata rimossa la funzione "toTerminal()", per evitare di stampare tutti i valori di input, output e target.

I parametri sono poi stati settati come segue:

- HiddenNodes pari a 20.
- Success pari a 3. In questo caso il controllo della soglia viene effettuato sull'errore dei dati di validazione.
- LearningRate pari a 0,3.

A questo punto è stato eseguito il codice. Dopo 10 esecuzioni sono stati estrapolati i seguenti dati (Tabella 3.1):

Tempo	**Cicli**	**Errore Learning**	**Errore Test**
74,166	6	0,01907	2,93595
86,244	7	0,02058	2,80934
73,771	6	0,01793	2,89103
73,893	6	0,02199	2,98414
86,76	7	0,01861	2,81521
NC	NC	NC	NC
98,858	8	0,01329	2,95133
73,575	6	0,01777	2,74417
84,965	7	0,01432	2,88323
73,424	6	0,02325	2,92169
Media: 80,628	**Media: 6,55**	**Media: 0,0185**	**Media: 2,8817**

Tabella 3.1 Tempi, cicli ed errori della rete con 20 neuroni nascosti.

Dai dati si nota come l'algoritmo sia riuscito a convergere arrivando ad un errore inferiore alla soglia pari a 3, in 9 casi su 10.

Nella Tabella 3.2, vengono riportati i dati relativi allo studio dell'andamento dell'errore (Figura 3.2).

Ciclo	Errore Learning	Errore Test
1	8,80294	28,74831
2	1,54649	17,10994
3	0,63251	8,62254
4	0,18361	4,63615
5	0,0428	3,34242
6	0,01733	2,93889

Tabella 3.2 Andamento dell'errore nella rete con 20 neuroni nascosti.

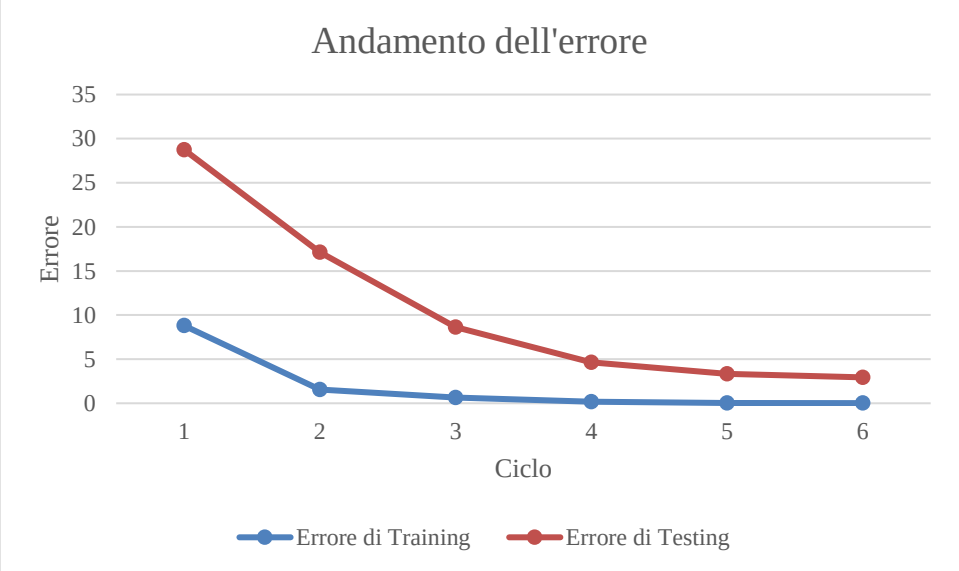

Figura 3.2 Grafico dell'andamento dell'errore nella rete con 20 neuroni nascosti.

3.4. Studio dell'overfitting e bontà del sistema

Successivamente si è provato ad abbassare la soglia a 2,5 per andare a studiare il fenomeno di "overfitting" dei dati.

L'overfitting si verifica quando la rete è molto allenata su un particolare set di dati, ma non risulta abbastanza buona nello stimare i valori a partire da un set di dati con cui non è stata allenata.

In questo caso, utilizzando pochi dati e allenando la rete iterativamente solo su questi dati è molto probabile che si verifichi il problema dell'overfitting. Si è quindi deciso di registrare i dati di 30 cicli e studiare nuovamente l'andamento dell'errore (Tabella 3.3)(Figura 3.3, 3.4, 3.5).

Ciclo	**Errore Learning**	**Errore Test**
1	8,63998	31,7526
2	1,77101	21,12455
3	0,87577	11,00423
4	0,28534	5,3964
5	0,06559	3,54801
6	0,02292	3,00075
7	0,01711	2,83267
8	0,01632	2,78036
9	0,01609	2,76512
10	0,01592	2,76512
11	0,01575	2,7631

12	0,01558	2,76551
13	0,01542	2,76843
14	0,01526	2,77151
15	0,01511	2,77468
16	0,01496	2,77787
17	0,01418	2,78109
18	0,01467	2,78432
19	0,01453	2,78756
20	0,01439	2,78756
21	0,01426	2,79405
22	0,01413	2,7973
23	0,014	2,80055
24	0,01388	2,8038
25	0,01376	2,80705
26	0,01364	2,8103
27	0,01352	2,81355
28	0,01341	2,8168
29	0,0133	2,82003
30	0,01319	2,82326

Tabella 3.3 Andamento dell'errore con soglia a 2,5 per lo studio dell'overfitting

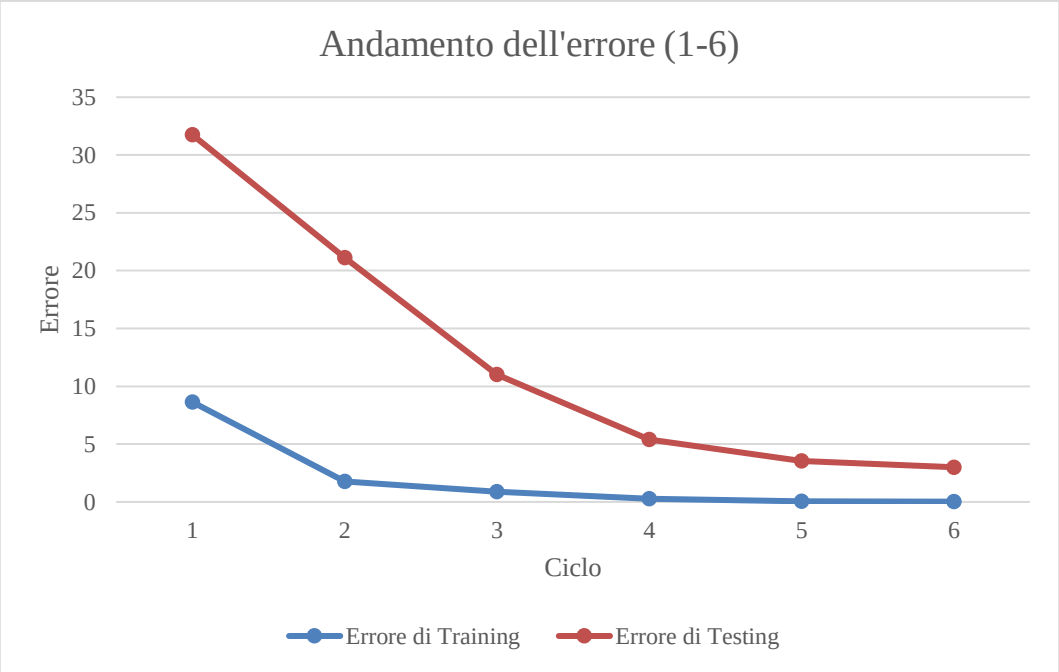

Figura 3.3 Grafico dell'andamento dell'errore con soglia a 2,5 per lo studio dell'overfitting. Dal ciclo 1 al ciclo 6.

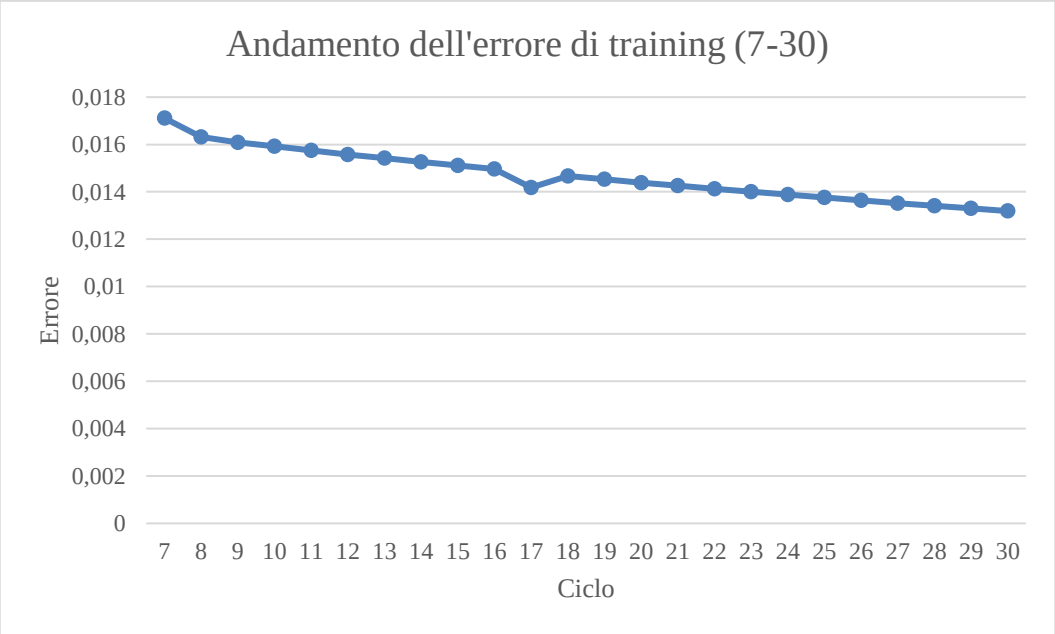

Figura 3.4 Grafico dell'andamento dell'errore di training con soglia a 2,5 per lo studio dell'overfitting. Dal ciclo 6 al ciclo 30.

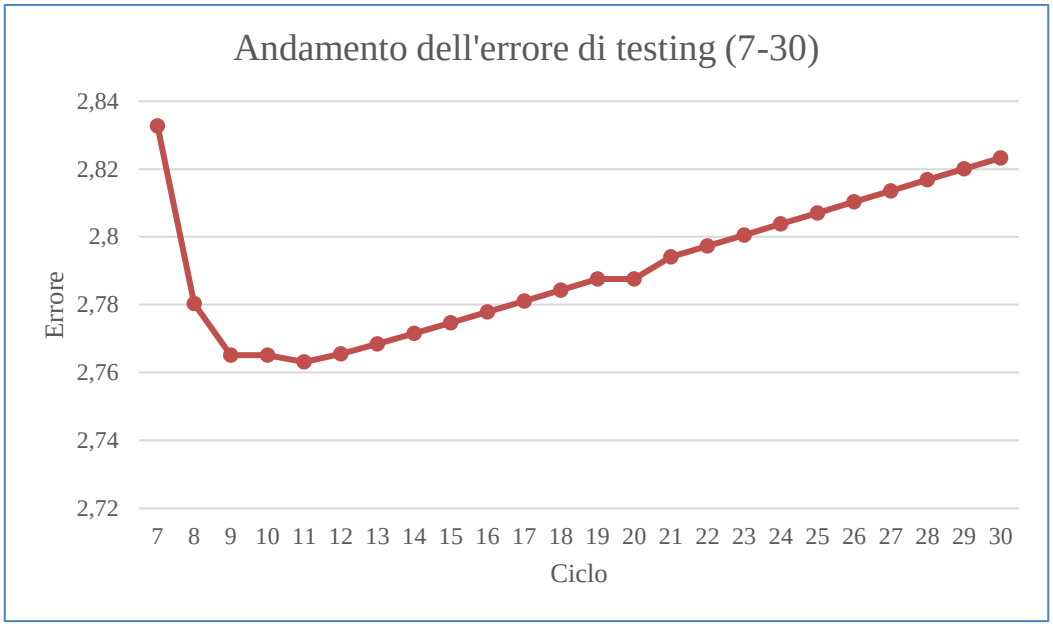

Figura 3.5 Grafico dell'andamento dell'errore di testing con soglia a 2,5 per lo studio dell'overfitting. Dal ciclo 6 al ciclo 30.

In Figura 3.4 si può notare come l'errore di training risulti sempre discendente, mentre l'errore di testing (Figura 3.5) dopo una iniziale discesa, circa al ciclo numero 10 inizi a risalire. I risultati ottenuti mostrano come la rete migliori nel predire i dati di training, anche se contestualmente peggiori nello stimare i dati su cui è stata allenata.

Alla luce delle simulazioni effettuate si può concludere che la soluzione migliore risulta quella di far completare alla rete non più di 12 cicli di training, chiamati "epoche", piuttosto che utilizzare una soglia di errore.

Infine, con lo scopo di confrontare i risultati, la rete è stata allenata con un "learning rate" pari a 0,05 e un "momentum" pari a 0,1. Il training in questo caso ha richiesto 12025460 millisecondi. I pesi della rete così ottenuti sono stati salvati. I dati sono stati quindi riportati ai loro valori iniziali (è stata annullata la normalizzazione) e sono stati riportati in grafico mediante MATLAB (Figura 3.6, 3.7, 3.8, 3.9).

Per verificare l'efficacia, in Figura 3.6 è riportato il confronto relativo ai dati di test della prima variabile del sistema di FitzHugh-Nagumo (prima colonna delle matrici), mentre in Figura 3.7, il confronto relativo alla seconda variabile. In

particolare, la linea blu rappresenta l'output fornito dai dati e la linea arancione l'output calcolato dalla rete.

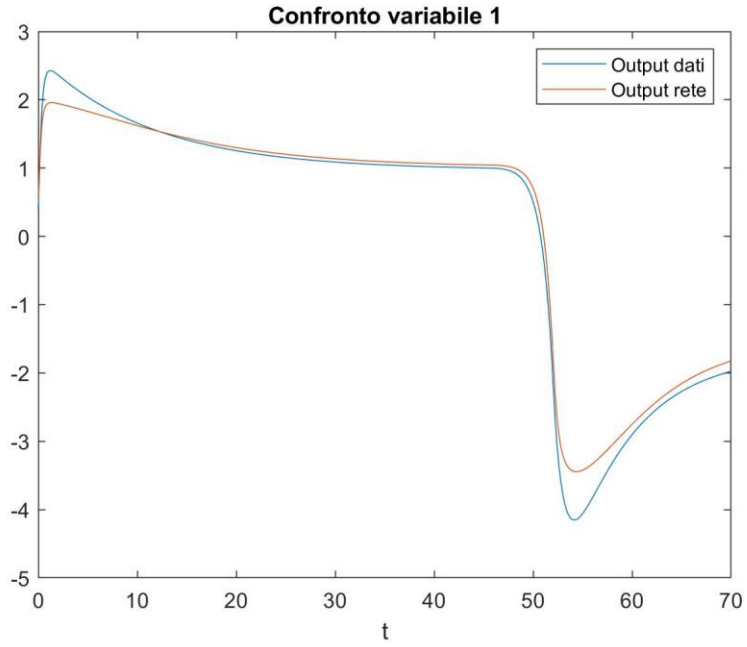

Figura 3.6 Grafico di contronto della variabile 1.

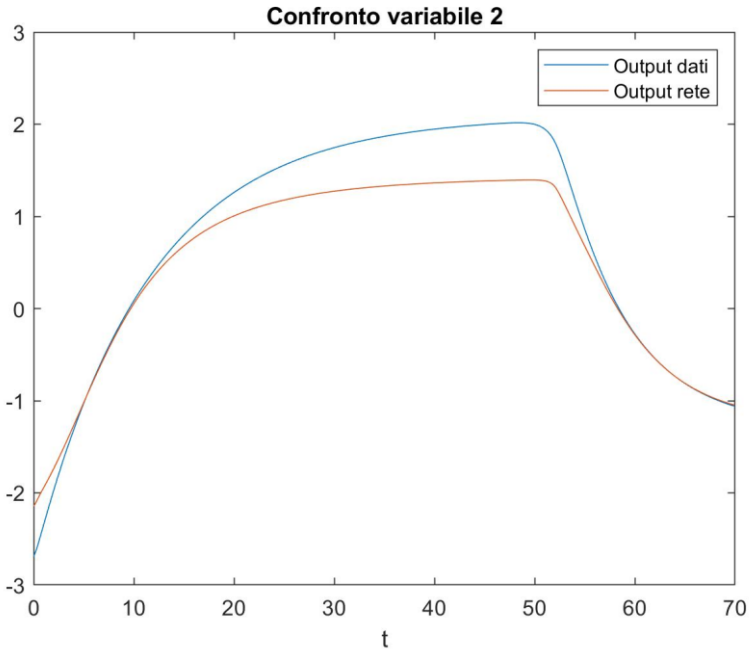

Figura 3.7 Grafico di confront della variabile 2.

Nelle figure 3.8 e 3.9 invece, è riportata la differenza tra l'output dei dati e l'output prodotto dalla rete.

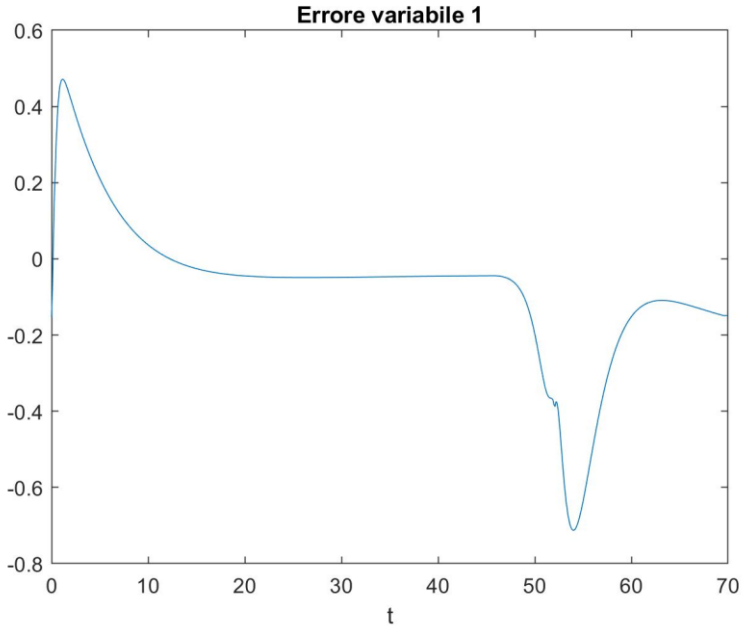

Figura 3.8 Grafico dell'errore della variabile 1.

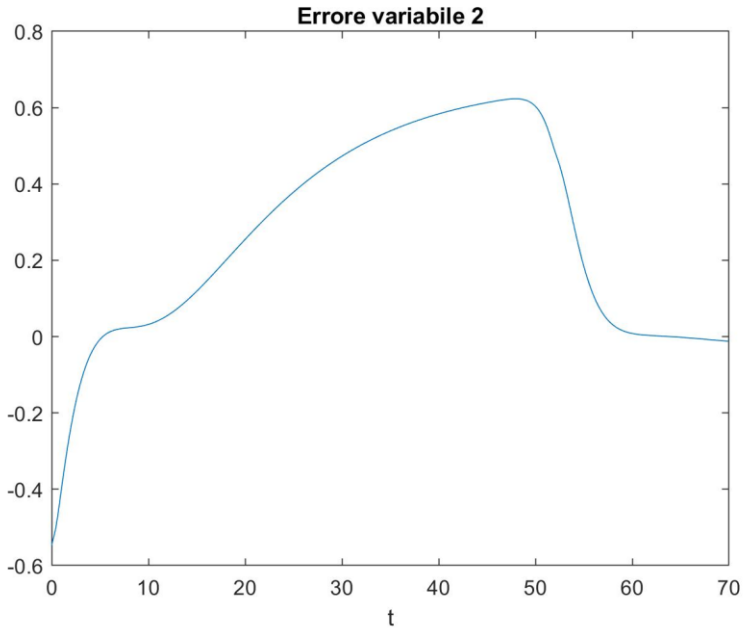

Figura 3.9 Grafico dell'errore della variabile 2.

È possibile notare come per quando riguarda la prima variabile il discostamento massimo registrato è di circa 0,7 intorno al 55esimo secondo, mentre per la seconda variabile il discostamento massimo registrato è di circa 0,6 attorno al secondo 50.

CONCLUSIONI

L'obiettivo del presente lavoro di tesi è stato articolato in due fasi: nella prima sono stati descritti modelli di reti neurali ampiamente riconosciuti dalla comunità scientifica, mentre nella seconda si è cercato di studiare il comportamento di una di tali reti neurali a bordo di un microcontrollore.

Nella seconda parte inizialmente si è partiti da una funzione semplice come lo XOR, che per quanto semplice presenta un problema non risolvibile linearmente. Per questo motivo è stato necessario utilizzare una rete perceptrone con uno strato nascosto, in grado di risolvere problemi anche non lineari. Si è cercato di migliorare la perfomance della rete variando i suoi stessi parametri e paragonando i risultati al fine di arrivare alla configurazione migliore (4 neuroni nascosti, soglia di successo pari a 0,002). In questa configurazione la rete è riuscita a imparare il pattern con un tempo medio di 9,109 secondi e un numero medio di cicli pari a 739,5.

Dopo aver valutato la rete con uno XOR in input, si è utilizzato il set di dati prodotti dal neurone FitzHugh-Nagumo. Si è optato per avere una rete con 20 neuroni nascosti, che risultano sufficienti per imparare il pattern, ma non eccessivi, tali da rallentare troppo la rete. In questo caso per la valutazione delle prestazioni sono state utilizzate altre 2 matrici di dati, utilizzati come dati di validazione. Dopo aver effettuato le misurazioni, si può affermare che la rete è in grado di convergere con un tempo medio di 80,628 secondi, un numero di cicli medio di 6,55, un'errore di learning medio di 0,0185 e un'errore di testing medio di 2,8817. Inoltre è stato valutato il fenomeno di overfitting che si verifica circa al ciclo numero 10. Pertanto è stato ritenuto opportuno allenare la rete per al massimo 12 cicli per evitare appunto il verificarsi di overfitting. Inoltre dai grafici sottoposti, è possibile visualizzare come la rete approssimi abbastanza bene i dati di test.

Alla luce dei risultati ottenuti possiamo concludere che è possibile utilizzare una rete neurale per il learning di pattern complessi anche direttamente a bordo di un microcontrollore, benché persistano alcune limitazioni dovute principalmente alla ristretta dimensione della memoria.

Il sistema potrebbe sicuramente migliorare, aggiungendo un componente esterno per la lettura da SD card, in grado di memorizzare un maggior numero di dati, oppure utilizzando la comunicazione seriale tra microcontrollore e PC per salvare i dati poco alla volta. Queste due soluzioni sono solo un esempio di come poter utilizzare un maggior numero di dati per rendere il processo di learning ancora più accurato.

RINGRAZIAMENTI

Mi è doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione dello stesso.

In primis, un ringraziamento speciale al mio relatore Paolo Arena, per la sua immensa pazienza, per i suoi indispensabili consigli, per le conoscenze trasmesse durante tutto il percorso di stesura dell'elaborato.

Ringrazio infinitamente i miei genitori che mi hanno sempre sostenuto, appoggiando ogni mia decisione, fin dalla scelta del mio percorso di studi.

Un grazie di cuore ai miei colleghi Luigi, Francesco ed Emanuele e mia cugina Cristina con cui ho condiviso l'intero percorso universitario. È grazie a loro che ho superato i momenti più difficili. Senza i loro consigli, non ce l'avrei mai fatta.

INDICE DELLE FIGURE

INDICE DEI GRAFICI

INDICE DELLE TABELLE

INDICE DELLE EQUAZIONI

Bibliografia

Banbury Colby, Zhou Chuteng, Fedorov Igor, Navarro Ramon M., Thakker Urmish, Gobe Dibakar, Reddi Vijay J., Mattina Matthew, Whatmough Paul N. "MicroNets: Neural Network Architectures for Deploying TinyML Applicaions on Commodity Microcontrollers." *Proceedings of the 4th MLSys Conference.* San Jose, CA, USA, 2021.

Bogolovskii Ivan A., Stepanov Andrey B., Ermolenko Daniil V., Pomogalova Albina V. "Implementation of an Approximator Based on a Multilayer Perceptron and Wavelet-Neural Network on the STM32 Microcontroller." *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus).*St. Petersburb and Moskow, Russia: IEEE, 2020. 1372-1377.

Chen Jiasi, Ran Xukan. "Deep Learning With Edge Computing: A Review." *Proceeding of the IEEE* 107, no. 8 (2019): 1655-1674.

Ciodaro T., Deva D., de Seixas J., Damazio D. "Online particle detection with neural networks based on topological calorimetry information." *Journal of Physics: Conference Series* 368 (2012).

Cotton Nicholas J., Wilamowski Bogdan M. "Compensation of Nonlinearities Using Neural Networks Implemented on Inexpensive Microcontrollers." *IEEE Transaction of Industrial Electronics* 58, no. 3 (2011): 733-740.

Crocioni Giulia, Gruosso Gianbattista, Pau Danilo, Denaro Davide, Zambrano Luigi, Di Giore Giuseppe. "Characterization of Neural Networks Automatically Mapped on Automotive-grade Microcontrollers." *TinyML Research Symposium'21.* San Jose, CA: arXiv:2103.00201, 2021.

Dongare A. D., Kharde R. R., Kachare Amit D. "Introduction to Artificial Neural Network." *International Journal of Engineering and Innovative Technology (IJEIT)* 2, no. 1 (2012): 189-194.

Haoyu Ren, Anicic Darko, Runkler Thomas A. "TinyOL: TinyML with Online-Learning on Microcontrollers." *The International Joint Conference on Neural Network (IJCNN).* IEEE, 2021.

Helmstaedter Moritz, Briggman Kevin L., Srinivas Turaga C., Jain Virein, Seung Sebastian H., Denk Winfried. "Connectomic reconstruction of the inner plexiform layer in the mouse retina." *Nature* 500 (2013): 168-174.

Krizhevsky Alex, Sutskever Ilya, Hinton Geoffrey E. "ImageNet classification with deep convolutional neural networks." *Communication of the ACM* 25, no. 6 (2017): 84-90.

LeCun Yann, Bengio Yoshua, Hinton Geoffrey. "Deep learning." *Nature* 521 (2015): 436–444.

Medrano-Marques N. J., Martin-del-Brio B. "A general method for sensor linearization based on neural network." *2000 IEEE International Symposium on Circuits and Systems (ISCAS).* Geneva, Switzerland: IEEE, 2000. 497-500.

Mikolov Tomas, Deoras Anoop, Povey Daniel, Burget Lukas, Cernocky Jan. "Strategies for training large scale neural network language models." *2011 IEEE Workshop on Automatic Speech Recognition & Understanding.* Waikoloa, HI, USA: IEEE, 2011. 196–201.

Nelson Mark, Ritzel John. "The Hodgkin—Huxley Model." In *The Book of GENESIS*, by Bower J. M., 29-30. New York, NY: Springer, 1998.

Parker Gary, Khan Mohammad. "Distributed Neural Network: Dynamic Learning via Backpropagation with Hardware Neurons using Arduino Chips." *2016 International Joint Conference on Neural Networks (IJCNN).* Vancouver, BC, Canada: IEEE, 2016. 206-212.

Zhang Yundong, Suda Naveen, Liangzhen Lai, Chandra Vikas. "Hello Edge: Keyword Spotting on Microcontrollers." Arm, San Jose, CA, Stanford University, Stanford, CA: arXiv:1711.07128, 2018.

Zhao Yanling, Deng Bimin and Wang Zhanrong. "Analysis and study of perceptron to solve XOR problem." *The 2nd International Workshop on Autonomous Decentralized System*. Beijing, China: IEEE, 2002. 168-173.

APPENDICE A

```
//Author: Ralph Heymsfeld

//28/06/2018



#include <math.h>



/******************************************************************
*

 * Network Configuration - customized per network


******************************************************************
/



const int PatternCount = 10;

const int InputNodes = 7;

const int HiddenNodes = 8;

const int OutputNodes = 4;

const float LearningRate = 0.3;

const float Momentum = 0.9;

const float InitialWeightMax = 0.5;

const float Success = 0.0004;



const byte Input[PatternCount][InputNodes] = {

  { 1, 1, 1, 1, 1, 1, 0 },  // 0
```

```
  { 0, 1, 1, 0, 0, 0, 0 },  // 1
  { 1, 1, 0, 1, 1, 0, 1 },  // 2
  { 1, 1, 1, 1, 0, 0, 1 },  // 3
  { 0, 1, 1, 0, 0, 1, 1 },  // 4
  { 1, 0, 1, 1, 0, 1, 1 },  // 5
  { 0, 0, 1, 1, 1, 1, 1 },  // 6
  { 1, 1, 1, 0, 0, 0, 0 },  // 7
  { 1, 1, 1, 1, 1, 1, 1 },  // 8
  { 1, 1, 1, 0, 0, 1, 1 }   // 9
};


const byte Target[PatternCount][OutputNodes] = {
  { 0, 0, 0, 0 },
  { 0, 0, 0, 1 },
  { 0, 0, 1, 0 },
  { 0, 0, 1, 1 },
  { 0, 1, 0, 0 },
  { 0, 1, 0, 1 },
  { 0, 1, 1, 0 },
  { 0, 1, 1, 1 },
  { 1, 0, 0, 0 },
  { 1, 0, 0, 1 }
};
```

```
/*****************************************************************
*

 * End Network Configuration

*****************************************************************
/




int i, j, p, q, r;

int ReportEvery1000;

int RandomizedIndex[PatternCount];

long  TrainingCycle;

float Rando;

float Error;

float Accum;




float Hidden[HiddenNodes];

float Output[OutputNodes];

float HiddenWeights[InputNodes+1][HiddenNodes];

float OutputWeights[HiddenNodes+1][OutputNodes];

float HiddenDelta[HiddenNodes];

float OutputDelta[OutputNodes];

float ChangeHiddenWeights[InputNodes+1][HiddenNodes];
```

```
float ChangeOutputWeights[HiddenNodes+1][OutputNodes];



void setup(){

  Serial.begin(9600);

  randomSeed(analogRead(3));

  ReportEvery1000 = 1;

  for( p = 0 ; p < PatternCount ; p++ ) {

    RandomizedIndex[p] = p ;

  }

}



void loop (){




/*************************************************************
*

* Initialize HiddenWeights and ChangeHiddenWeights

*************************************************************
/



  for( i = 0 ; i < HiddenNodes ; i++ ) {

    for( j = 0 ; j <= InputNodes ; j++ ) {

      ChangeHiddenWeights[j][i] = 0.0 ;

      Rando = float(random(100))/100;
```

```
      HiddenWeights[j][i] = 2.0 * ( Rando - 0.5 ) *
InitialWeightMax ;

    }

  }

/******************************************************************
*

* Initialize OutputWeights and ChangeOutputWeights

******************************************************************
/



  for( i = 0 ; i < OutputNodes ; i ++ ) {

    for( j = 0 ; j <= HiddenNodes ; j++ ) {

      ChangeOutputWeights[j][i] = 0.0 ;

      Rando = float(random(100))/100;

      OutputWeights[j][i] = 2.0 * ( Rando - 0.5 ) *
InitialWeightMax ;

    }

  }

  Serial.println("Initial/Untrained Outputs: ");

  toTerminal();

/******************************************************************
*

* Begin training

******************************************************************
/
```

```
  for( TrainingCycle = 1 ; TrainingCycle < 2147483647 ;
TrainingCycle++) {



/****************************************************************
*

* Randomize order of training patterns

****************************************************************
/



    for( p = 0 ; p < PatternCount ; p++) {

      q = random(PatternCount);

      r = RandomizedIndex[p] ;

      RandomizedIndex[p] = RandomizedIndex[q] ;

      RandomizedIndex[q] = r ;

    }

    Error = 0.0 ;

/****************************************************************
*

* Cycle through each training pattern in the randomized order

****************************************************************
/

    for( q = 0 ; q < PatternCount ; q++ ) {

      p = RandomizedIndex[q];



/****************************************************************
*

* Compute hidden layer activations
```

```
******************************************************************
/



      for( i = 0 ; i < HiddenNodes ; i++ ) {

        Accum = HiddenWeights[InputNodes][i] ;

        for( j = 0 ; j < InputNodes ; j++ ) {

          Accum += Input[p][j] * HiddenWeights[j][i] ;

        }

        Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;

      }



/******************************************************************
*

* Compute output layer activations and calculate errors

******************************************************************
/



      for( i = 0 ; i < OutputNodes ; i++ ) {

        Accum = OutputWeights[HiddenNodes][i] ;

        for( j = 0 ; j < HiddenNodes ; j++ ) {

          Accum += Hidden[j] * OutputWeights[j][i] ;

        }

        Output[i] = 1.0/(1.0 + exp(-Accum)) ;

        OutputDelta[i] = (Target[p][i] - Output[i]) * Output[i] *
(1.0 - Output[i]) ;
```

```
        Error += 0.5 * (Target[p][i] - Output[i]) * (Target[p][i]
- Output[i]) ;

      }



/******************************************************************
*

* Backpropagate errors to hidden layer

******************************************************************
/



      for( i = 0 ; i < HiddenNodes ; i++ ) {

        Accum = 0.0 ;

        for( j = 0 ; j < OutputNodes ; j++ ) {

          Accum += OutputWeights[i][j] * OutputDelta[j] ;

        }

        HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]) ;

      }







/******************************************************************
*

* Update Inner-->Hidden Weights

******************************************************************
/
```

```
      for( i = 0 ; i < HiddenNodes ; i++ ) {

        ChangeHiddenWeights[InputNodes][i] = LearningRate *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[InputNodes][i] ;

        HiddenWeights[InputNodes][i] +=
ChangeHiddenWeights[InputNodes][i] ;

        for( j = 0 ; j < InputNodes ; j++ ) {

          ChangeHiddenWeights[j][i] = LearningRate * Input[p][j] *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[j][i];

          HiddenWeights[j][i] += ChangeHiddenWeights[j][i] ;

        }

      }



/******************************************************************
*

* Update Hidden-->Output Weights

******************************************************************
/



      for( i = 0 ; i < OutputNodes ; i ++ ) {

        ChangeOutputWeights[HiddenNodes][i] = LearningRate *
OutputDelta[i] + Momentum * ChangeOutputWeights[HiddenNodes][i] ;

        OutputWeights[HiddenNodes][i] +=
ChangeOutputWeights[HiddenNodes][i] ;

        for( j = 0 ; j < HiddenNodes ; j++ ) {

          ChangeOutputWeights[j][i] = LearningRate * Hidden[j] *
OutputDelta[i] + Momentum * ChangeOutputWeights[j][i] ;

          OutputWeights[j][i] += ChangeOutputWeights[j][i] ;

        }
```

```
      }

    }



/*****************************************************************
*

* Every 1000 cycles send data to terminal for display

*****************************************************************
/

    ReportEvery1000 = ReportEvery1000 - 1;

    if (ReportEvery1000 == 0)

    {

      Serial.println();

      Serial.println();

      Serial.print ("TrainingCycle: ");

      Serial.print (TrainingCycle);

      Serial.print ("  Error = ");

      Serial.println (Error, 5);



      toTerminal();



      if (TrainingCycle==1)

      {

        ReportEvery1000 = 999;

      }
```

```
      else

      {

        ReportEvery1000 = 1000;

      }

    }




/****************************************************************
*

* If error rate is less than pre-determined threshold then end

****************************************************************
/



    if( Error < Success ) break ;

  }

  Serial.println ();

  Serial.println();

  Serial.print ("TrainingCycle: ");

  Serial.print (TrainingCycle);

  Serial.print ("  Error = ");

  Serial.println (Error, 5);



  toTerminal();
```

```
  Serial.println ();
  Serial.println ();
  Serial.println ("Training Set Solved! ");
  Serial.println ("--------");
  Serial.println ();
  Serial.println ();
  ReportEvery1000 = 1;
}


void toTerminal()
{


  for( p = 0 ; p < PatternCount ; p++ ) {
    Serial.println();
    Serial.print ("  Training Pattern: ");
    Serial.println (p);
    Serial.print ("  Input ");
    for( i = 0 ; i < InputNodes ; i++ ) {
      Serial.print (Input[p][i], DEC);
      Serial.print (" ");
    }
    Serial.print ("  Target ");
    for( i = 0 ; i < OutputNodes ; i++ ) {
```

```
      Serial.print (Target[p][i], DEC);
      Serial.print (" ");
    }
/******************************************************************
*
* Compute hidden layer activations
******************************************************************
/

    for( i = 0 ; i < HiddenNodes ; i++ ) {
      Accum = HiddenWeights[InputNodes][i] ;
      for( j = 0 ; j < InputNodes ; j++ ) {
        Accum += Input[p][j] * HiddenWeights[j][i] ;
      }
      Hidden[i] = 1.0/(1.0 + exp(-Accum)) ;
    }


/******************************************************************
*
* Compute output layer activations and calculate errors
******************************************************************
/

    for( i = 0 ; i < OutputNodes ; i++ ) {
      Accum = OutputWeights[HiddenNodes][i] ;
```

```
      for( j = 0 ; j < HiddenNodes ; j++ ) {
        Accum += Hidden[j] * OutputWeights[j][i] ;
      }
      Output[i] = 1.0/(1.0 + exp(-Accum)) ;
    }
    Serial.print ("  Output ");
    for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Output[i], 5);
      Serial.print (" ");
    }
  }


}
```