

# Rete neurale di Hopfield

Laura Poli  
Andrea Masini  
Lorenzo Vannucci  
Ettore Tasini

Ultima modifica al codice: 06/02/2026

## 1 Scelte progettuali e implementative

### 1.1 Organizzazione del codice

Abbiamo deciso di suddividere il progetto in tre *translation unit* principali: **pattern**, **matrix**, **acquisition**. Abbiamo poi inserito una *translation unit* minore per l'interfaccia utente al fine di avere un main più ordinato e leggibile.

- **acquisition:** utilizza la libreria SFML per implementare le funzioni che trasformano le immagini in pattern binari (vettori da dare in input alla rete).
- **pattern:** contiene le funzioni necessarie per lavorare con i pattern/vettori (creazione, conversioni e operazioni varie).
- **matrix:** implementa la logica principale della rete di Hopfield, in particolare le fasi di apprendimento tramite matrice dei pesi e richiamo. In questa *translation unit* abbiamo anche definito la funzione energia.
- **userInterface:** contiene le interfacce di input e output per le due fasi principali di training e recall, permettendo di fare più tentativi con lo stesso set di immagini.

L'organizzazione dei test è simmetrica a quella del codice sorgente: per ogni *translation unit* principale c'è il corrispettivo `file_test.cpp` con i test. Per esempio nel file relativo a **pattern** (`pattern_test.cpp`) sono contenuti i test che controllano il corretto funzionamento delle funzioni definite in `pattern.cpp`.

### 1.2 Implementazione del Simulated Annealing

Per cercare di migliorare le prestazioni della rete, abbiamo implementato un sistema chiamato *Simulated Annealing*. Grazie a questa implementazione, il nostro programma non accetta più soltanto modifiche al pattern che riducono l'energia del sistema (come in una rete di Hopfield standard),

ma permette di accettare, con una certa probabilità, anche modifiche che la aumentano. Questa probabilità è proporzionale alla temperatura.

All’inizio della fase di richiamo abbiamo impostato una temperatura della rete alta: di conseguenza, sono più frequenti le modifiche che aumentano l’energia del sistema. Dopo ogni ciclo, la temperatura viene abbassata per consentire alla rete di stabilizzarsi in corrispondenza di un minimo della funzione energia.

Questo meccanismo aiuta a evitare che la rete smetta di aggiornare il pattern in corrispondenza di un minimo locale della funzione energia, restituendo un’immagine indesiderata. Un effetto collaterale di questa implementazione è l’aumento del “tempo di ragionamento” della rete, dovuto alla mole aggiuntiva di calcoli che deve eseguire.

## 2 Dipendenze esterne

Abbiamo usato la libreria **SFML** per trasformare le immagini in vettori (input della rete) e per mostrare a schermo le immagini riformattate e i risultati dell’elaborazione.

**Installazione su sistemi Debian/Ubuntu:**

```
sudo apt update
sudo apt install libsFML-dev
```

## 3 Come eseguire il progetto

### 3.1 Prerequisiti

- compilatore C++ 20
- Libreria grafica SFML installata
- Cmake versione 3.28 o superiore

### 3.2 Esecuzione

Per eseguire la Rete Neurale, scrivere nel terminale (nella cartella scelta per l’eseguibile):

```
./ReteNeurale
```

Per eseguire i test invece scrivere:

```
./ReteNeuraleTest
```

## 4 Input e Output

### 4.1 Parametri di input

- **imgName** (stringa): le immagini scelte sono già caricate all'interno della cartella images. Al fine di dare in input un'immagine è necessario scrivere il nome della figura scelta (senza menzionare l'estensione .png) quando richiesto; il codice provvederà a riconoscere la stringa e selezionare l'immagine di riferimento.
- **lato** (unsigned): permette di scegliere il formato delle immagini da memorizzare e richiamare (ad esempio, dando in input 64 le immagini verranno convertite in formato 64x64 pixel).
- **testImgName** (stringa): funzionamento analogo a imgName, indica l'immagine da sporcare con la funzione `addNoise`.
- **noiseLevel** (float): indica la percentuale di noise che si vuole inserire (scala da 0 a 100). Per un corretto funzionamento non deve essere inserito il simbolo %.
- **input** (stringa): file o cartella di input (per inferenza).

### 4.2 Formato di output

- **display**: prendendo in input la misura del lato e un pattern apre una finestra e disegna quest'ultimo considerando il valore dei neuroni (+1 o -1).
- **energy** (vector<float>): vettore in cui vengono memorizzati i valori dell'energia del pattern ad ogni step di recall.
- **isIdentical** (bool): riconosce se un pattern è identico o invertito rispetto ad un altro, chiamandola è quindi possibile controllare la convergenza dopo il recall.

### 4.3 Esempi immediatamente usabili per la valutazione

La scelta dei parametri di input è guidata: una volta eseguito il programma viene chiesto all'utente di rispondere a una serie di domande, attraverso le quali il programma decide l'input da dare alla rete. Inserendo un elevato numero di pixel, i tempi di esecuzione aumentano notevolmente; perciò si suggerisce di rimanere fra 60 e 100. Inoltre, per una resa migliore, si consiglia di utilizzare quattro immagini.

```
Hopfield Network Creation.
Insert the number of pixels you want to use (32~128): 90
[1] Insert the name of a png file ('list' to see the names) or write 'stop': Trooper
[2] Insert the name of a png file ('list' to see the names) or write 'stop': stop
Training phase completed.
Insert the name of image you want to corrupt or write 'stop' : Trooper
Insert the percentage you want to corrupt the image with: 25
```

## 5 Risultati e loro interpretazione

### 5.1 Funzionamento generale

Il programma è in grado di memorizzare, richiamare e correggere pattern binari; implementa quindi con successo una memoria associativa basata sul modello di Hopfield.

Come previsto, la rete non è priva di limitazioni. In particolare abbiamo notato che il numero di immagini che la rete riesce a memorizzare con successo è molto inferiore al suo limite massimo teorico ( $0.138 N$ , con  $N$  numero di neuroni): già dopo la memorizzazione di circa 5 immagini iniziano a comparire dei problemi e non si osserva una proporzionalità diretta fra il numero di neuroni della rete e il numero di immagini che questa riesce a immagazzinare.

Questo fenomeno è probabilmente da attribuire al fatto che le immagini insegnate alla rete sono immagini reali e, quindi, non rispettano le assunzioni di base di pseudo-casualità e ortogonalità dei pattern usate per calcolare il limite teorico.

### 5.2 Considerazioni sul Simulated Annealing

L'introduzione del Simulated Annealing non ha portato miglioramenti sensibili alle prestazioni della rete, probabilmente perchè la limitazione principale non risiede nella presenza di minimi locali superficiali, ma nella struttura stessa del paesaggio energetico. Crediamo che, a causa della forte correlazione tra le immagini, i bacini di attrazione dei pattern memorizzati tendano a sovrapporsi, “confondendo” la rete.

La logica del *Simulated Annealing* ci è sembrata interessante e, quindi, abbiamo deciso di lasciare questa implementazione nel programma. Tuttavia, abbiamo mantenuto anche la versione originale della funzione *recall*, commentandola all'interno del codice, in modo da poter tornare in qualsiasi momento alla versione standard della rete di Hopfield.

## 6 Strategia di test

I test sono stati implementati usando **doctest**, con l'obiettivo di verificare il corretto funzionamento della rete e il rispetto del modello matematico di Hopfield. Per chiarezza dividiamo i test in tre categorie:

- **Test di compatibilità col modello:** controllano che la matrice dei pesi sia simmetrica e con diagonale nulla; verificano che l'energia non aumenti dopo la fase di richiamo e che aumenti dopo la corruzione di un pattern; verificano che la rete non distingua fra un pattern e il suo inverso (stesso pattern invertito di segno).
- **Test su casistiche base:** sono numerosi e verificano il comportamento delle funzioni definite nel codice sorgente nei casi più semplici. Ad esempio, la funzione `resize` definita in `acquisition.cpp` uniforma le immagini a una dimensione esatta: in `acquisition_test.cpp` è presente un test che controlla la dimensione dell'immagine restituita in output da `resize` su un'immagine di prova.

- **Test su casistiche limite (edge cases):** verificano che il programma gestisca le eccezioni in maniera controllata, senza entrare in loop infiniti o arrestarsi bruscamente. Ad esempio controllano cosa succede se si inseriscono percentuali di rumore non valide ( $<0\%$  o  $>100\%$ ), oppure se si effettuano operazioni su pattern di dimensioni incompatibili con la matrice.

## 7 Dichiarazione uso IA generativa

**Uso di sistemi di IA generativa:** È stato usato il software di intelligenza artificiale Gemini per suggerimenti e confronti e per l'implementazione della parte grafica.

## 8 Altre informazioni utili

Per lavorare in gruppo abbiamo usato GitHub, di seguito condividiamo la repository pubblica sulla quale abbiamo caricato il codice.

- Repository GitHub: [laurapoli118/progetto](https://github.com/laurapoli118/progetto)