Rete neurale di Hopfield

Alessia Barborini, Chiara Luppino

25.08.2025

1 Introduzione

Il progetto presentato si incentra sulla realizzazione di una **rete neurale di Hopfield** in grado di immagazzinare pattern binari e recuperarli nel caso in cui le vengano fornite versioni corrotte degli stessi. Le reti neurali trovano applicazione in numerosi ambiti, dalla robotica alla bioinformatica, dalla medicina fino al settore forense. Un esempio rilevante, reso possibile grazie a modelli più avanzati rispetto alla rete neurale di Hopfield, è il riconoscimento facciale, oggi impiegato per scopi di sicurezza e identificazione. Proprio per questo motivo, nella selezione del set di immagini destinato all'addestramento della rete, si è scelto di utilizzare dei volti e, per allineare la scelta con l'obiettivo ultimo del progetto (elaborato finale del corso di Programmazione per la **fisica**), sono stati presi in considerazione i volti di sei noti scienziati.

2 Scelte progettuali e implementative

Il progetto presenta come oggetto principale la classe ***Hopfield***, la quale contiene le funzioni che consentono di:

- ridimensionare un'immagine per adattarla alle altre della rete (*resizeImage*);
- convertire un'immagine in formato binario (*pattern*) e riottenerla in bianco e nero (*bawImage*);
- immagazzinare i pattern relativi alle immagini scelte per addestrare la rete (*loadPatterns*) e caricare la matrice dei pesi (*matrix* e *getMatrix*);
- ricostruire un pattern corrotto (*update*) e calcolare la funzione energia (*energy*) di un pattern in aggiornameto.

Sono inoltre presenti alcune funzioni libere di supporto, tra cui:

- *loadImage* e *loadSprite*, che permettono di caricare le immagini sfruttando gli oggetti della libreria grafica SFML;

- *interpolation*, che contiene la formula per l'interpolazione bilineare[1], tecnica sfruttata in *resizeImage* per ridimensionare in modo accurato le immagini;
- *corruption*, impiegata per corrompere i pattern memorizzati, al fine di testare la capacità della rete di ricostruirli correttamente.

Infine, si è scelto di introdurre il type alias *Pattern* (`std::vector<int>`) e la struct *Drawable* per snellire il codice e semplificare l'implementazione della rete. La struct *Drawable* è stata pensata per facilitare il caricamento delle immagini, racchiudendo in un unico oggetto l'immagine, la texture e lo sprite, rispettivamente di tipo `sf::Image`, `sf::Texture`, `sf::Sprite`. Questa scelta, insieme all'implementazione delle funzioni libere *loadImage* e *loadSprite* consente di rendere più leggibile la gestione delle immagini all'interno della funzione *main*, dove è sufficiente creare un singolo oggetto di tipo *Drawable* e, se necessario, impostarne solo la posizione e il fattore di scala in aggiunta.

2.1 Funzioni loadImage e loadSprite

La scelta di utilizzare due funzioni per il caricamento delle immagini non è casuale: essa nasce dalla volontà di ottimizzare l'uso della memoria durante il caricamento dei pattern. La funzione *loadPatterns*, infatti, sfrutta la funzione *pattern* che accetta come input un oggetto di tipo `sf::Image`. Di conseguenza, non è necessario caricare anche texture e sprite per ogni immagine immagazzinata in quanto esse sono necessarie solo per la rappresentazione grafica e non per l'addestramento della rete. La funzione *loadSprite*, che carica anche texture e sprite, viene utilizzata solo all'interno di *main.cpp*, quando è necessario visualizzare graficamente un'immagine.

2.2 Caricamento della matrice dei pesi: funzioni loadPatterns, matrix e getMatrix

La matrice dei pesi W_ costituisce, insieme a width_ e height_ (larghezza e altezza in pixel delle immagini immagazzinate), la parte privata della classe *Hopfield*. Essa viene calcolata tramite la funzione *matrix*, che accetta un `std::vector<Pattern>` ottenibile con *loadPatterns*. Una volta calcolata, *matrix* salva la matrice su un file di testo (*"weights.txt"*). In questo modo non è necessario ricalcolarla ad ogni esecuzione del programma, ma è sufficiente rileggerla dal file tramite la funzione *getMatrix*.

2.3 Fase di richiamo: funzioni corruption e update

Una volta addestrata la rete, è possibile verificarne la capacità di ricostruire pattern corrotti. La corruzione dei pattern avviene selezionando casualmente un certo numero di pixel e invertendone lo stato (un pixel acceso diventa spento e viceversa). Le variabili passate alla funzione *corruption* sono il pattern da

corrompere e un parametro `unsigned int rate` che rappresenta il tasso di corruzione: la probabilità che un pixel venga corrotto è $\frac{1}{rate}$. In realtà, il tasso effettivo risulta leggermente inferiore, poiché l'algoritmo di generazione casuale potrebbe selezionare più volte lo stesso pixel, annullando l'effetto di corruzioni precedenti.
La funzione *update*, infine, aggiorna lo stato di ciascun neurone seguendo la regola di Hopfield e restituisce il pattern aggiornato. L'aggiornamento deve essere ripetuto finché il pattern non converge. Quest'ultima operazione viene effettuata direttamente nel *main* per poter visualizzare graficamente l'evoluzione.

3 Strumenti esterni da installare

Il progetto presenta un file *"CMakeLists.txt"* per facilitare la compilazione. Installando il *ninja build system* sarà possibile generare configurazioni per la compilazione in modalità Debug e Release. Per installarlo basta eseguire il seguente comando (Ubuntu 24.04):
`sudo apt install ninja-build`.

È inoltre necessario installare la libreria esterna SFML con il seguente comando:
`sudo apt install libsfml-dev`

4 Compilare ed eseguire il programma

I comandi da utilizzare per configurare e buildare il progetto sono i seguenti:

- `cmake -S . -B build -G"Ninja Multi-Config"`
- `cmake --build build --config Debug`

Quest'ultimo serve per compilare in modalità Debug. Per eseguire il programma in questa modalità basta utilizzare:

`build/Debug/neuralnet`.

Se invece si vuole compilare in modalità Release è necessario usare:

- `cmake --build build --config Release`

per poi eseguire con:

`build/Release/neuralnet`

5 Input e output

Una volta eseguito il programma, sarà richiesto di scegliere lo scienziato del quale si vuole visualizzare l'immagine. Digitarne il nome e premere invio. A

questo punto si aprirà una finestra grafica in cui saranno presenti, da sinistra a destra:

- l'immagine originale a colori
- l'immagine ridimensionata e in bianco e nero
- l'immagine corrotta con la funzione *corruption*
- l'evoluzione dell'immagine fino a raggiungere lo stato stazionario

Se si desidera testare il funzionamento della rete su un'altra immagine è sufficiente chiudere la finestra grafica ed eseguire nuovamente il programma (a questo punto la matrice dei pesi non verrà più ricalcolata, poiché è già stata salvata alla prima esecuzione).
Una volta chiusa la finestra grafica, sarà possibile visualizzare i valori della funzione energia relativa ai pattern in aggiornamento.

6 Interpretazione dei risultati ottenuti

Questo tipo di implementazione della rete neurale di Hopfield presenta alcuni limiti; il problema principale risiede nella convergenza dei pattern verso stati spuri (come avviene, ad esempio, con l'immagine di Schrodinger). Dall'esecuzione del programma si nota che la maggior parte delle immagini converge verso uno stato molto simile a quello iniziale, dimostrando che la rete è in grado di ricostruire correttamente i pattern memorizzati. Un'ulteriore conferma del corretto funzionamento di questa versione "primitiva" della rete di Hopfield è fornita dai valori decrescenti della funzione energia durante l'aggiornamento.

7 Strategia di test

Per verificare il corretto funzionamento della rete è stato impiegato il framework Doctest. Il file *hopfield.test.cpp* è stato organizzato in diversi TEST_CASE, ciascuno suddiviso in più SUBCASE. Ogni TEST_CASE è dedicato al testing di al più due funzioni, concentrandosi sugli aspetti critici del loro comportamento. Particolare attenzione è stata riservata alle funzioni *resizeImage*, *matrix*, *getMatrix* e *update*, che costituiscono il nucleo portante del programma, e su cui sono stati effettuati la maggior parte dei test.

8 Uso di sistemi di Intelligenza Artificiale generativa

I sistemi di Intelligenza Artificiale generativa sono stati utilizzati esclusivamente come supporto nello sviluppo della finestra grafica, integrando le informazioni ottenute dai tutorial ufficiali di SFML. Inoltre, sono stati impiegati per facilitare

i calcoli dei test più complessi, in particolare nelle funzioni *resizeImage*, *matrix* e *update*.

References

[1] Marco Gribaudo, *Sistemi Multimediali: Trasformazione di immagini*, dispense universitarie, PDF, 2024.