Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze informatiche

Analisi dell'efficienza di System on Chip su applicazioni parallele

Relazione finale in
High-Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Edoardo Barbieri

II Sessione di laurea
Anno Accademico 2017/2018

Indice

1 Calcolo parallelo **5**

1.1 Concetti base e motivazioni 5

1.2 OpenMP 7

1.3 Estensioni SIMD 9

1.4 Valutare le prestazioni 11

2 Machine Learning **13**

2.1 Introduzione al Machine Learning 13

2.2 Deep Learning 14

2.3 Dense neural network 15

2.3.1 Overfitting e tecniche di risoluzione 20

3 Componenti utilizzate **23**

3.1 Raspberry Pi 23

3.2 Processori x86 25

4 MNIST database **27**

4.1 Descrizione del dataset 27

4.2 Qualità del database 28

4.3 Ampliamento del dataset 29

4.4 Trasformazione affine 30

5 Sviluppo del progetto **34**

5.1 Rete neurale artificiale 34

5.1.1 Tecniche di parallelismo 43

5.2 Trasformazione affine 46

5.2.1 Algoritmo parallelo 49

6 Valutazione dei risultati **51**

6.1 Trasformazione affine 51

6.2 Rete neurale interamente connessa 55
6.3 Risultati machine learning 61

A Esecuzione della rete neurale **68**

Introduzione

In questa tesi analizzeremo le prestazioni di un *System on Chip* (SoC) in alcuni contesti di calcolo parallelo: in questo caso il SoC in questione è il Raspberry Pi. Questo calcolatore, montato su un'unica scheda, è nato con l'intento di promuovere l'insegnamento dell'informatica nelle scuole; per via del suo prezzo ridotto, infatti, è una risorsa accessibile a molti. Il Raspberry Pi è diventato popolare per via della grande comunità sul Web, che è cresciuta insieme al prodotto; la scheda inoltre viene utilizzata in settori come: Internet of Things, Sistemi Embedded e Robotica. Le ultime versioni della scheda Raspberry Pi sono molto più interessanti anche da un punto di vista computazionale rispetto ai primi modelli. L'ultima versione rilasciata (Raspberry Pi 3 B+) è dotata di 1 GB di RAM e di un processore quad-core con architettura ARM NEON, caratterizzata dai prezzi e consumi ridotti. Tali specifiche danno i presupposti per un inserimento di questo dispositivo in contesti di *high performace computing* (HPC) tramite l'utilizzo di una programmazione parallela specifica.
Per valutare le prestazioni di questa scheda si è voluto implementare un'applicazione che sia di uso comune in ambienti HPC, evitando semplici benchmark sulle singole componenti hardware. L'applicazione deve inoltre essere implementata tenendo conto dell'architettura di cui si dispone, in modo da ottenere risultati quanto più possibili legati alle caratteristiche hardware.
Al giorno d'oggi la materia del *machine learning* è molto popolare per via dei risultati che si stanno raggiungendo. Molti sistemi intelligenti necessitano però di grandi capacità di calcolo per essere addestrati, e per questo il settore dell'*high performace computing* lavora a stretto contatto con quello del *machine learning*.
Il progetto prevede di sviluppare un sistema di *machine learning*: una rete neurale artificiale che si addestra nel riconoscere cifre scritte a mano libera. Verranno quindi descritte e implementate alcune tecniche per parallelizzare la fase di addestramento della rete neurale. Questa applicazione verrà sfruttata per effettuare dei *benchmark* sia sul Raspberry Pi, che su un calcolatore

"standard": l'università di Bologna ha messo a disposizione un server di calcolo che dispone di due processori Intel Xeon. I dati raccolti su queste due architetture (Raspberry Pi e Xeon) saranno messi a confronto. L'obiettivo è quello di analizzare se effettivamente dispositivi come il Raspberry Pi possono avere un qualche tipo di vantaggio in contesti HPC: in particolare verrà svolta un'analisi sulla capacità di svolgere calcolo parallelo.

I capitoli sono suddivisi in questo modo:

- Capitolo 1: si introducono i concetti base del calcolo parallelo, e si descrivono alcuni strumenti per sfruttare architetture multi-core e architetture SIMD. Infine si definiscono dei parametri per analizzare gli incrementi di prestazioni ottenuti dalla parallelizzazione di un problema.
- Capitolo 2: si introduce brevemente la materia del *Machine Learning*, con particolare attenzione al *Deep Learning*. Viene descritta la struttura delle reti neurali artificiali, e alcuni algoritmi per addestrarle.
- Capitolo 3: vengono descritte le componenti hardware utilizzate per effettuare le prove sperimentali: Raspberry Pi e processore Xeon.
- Capitolo 4: viene descritto il dataset MNIST, esso viene utilizzato durante la fase di training della rete neurale artificiale. Viene poi trattato un metodo per estendere il dataset, ovvero la trasformazione affine sulle immagini.
- Capitolo 5: viene discussa l'implementazione dell'algoritmo di machine learning e le tecniche adottate per parallelizzare la fase di addestramento. Viene inoltre descritta l'implementazione della trasformazione affine e la sua parallelizzazione.
- Capitolo 6: sono descritte le prove sperimentali effettuate e vengono analizzati i risultati raccolti sia sulla scheda Raspberry Pi che sui processori Xeon.

Capitolo 1

Calcolo parallelo

In questo capitolo si introduce il calcolo parallelo, alcune tra le tecniche principali, e le motivazioni per cui è necessario usare questo tipo di approccio. Infine si parla di come valutare gli incrementi di prestazione ottenuti dalla parallelizzazione di un problema.

1.1 Concetti base e motivazioni

Al giorno d'oggi molte applicazioni necessitano di un'elevata potenza di calcolo, alcuni esempi possono essere: simulazioni di grandi dimensioni, animazione 3D, finanza, meteorologia, machine learning ecc... tuttavia prestazioni così elevate non possono essere ricavate da una singola unità di calcolo.

> Oggi la differenza tra calcolatori di uso comune e supercalcolatori non risiede più nella tecnologia, ma nell'architettura. Per raggiungere le massime prestazioni si ricorre infatti al parallelismo. L'idea di base è molto semplice: per moltiplicare le prestazioni dei già velocissimi microprocessori occorre moltiplicarne il numero, di modo che un calcolo particolarmente impegnativo non venga affidato a un singolo processore, ma a un insieme di processori che collaborino alla sua soluzione. Le unità di calcolo in un calcolatore parallelo lavorano pertanto simultaneamente e, nei casi più favorevoli, le prestazioni sono proporzionali proprio al numero delle unità. [1]

Non è possibile aumentare ulteriormente (di fattori significativi) le prestazioni ottenute da un singolo processore a causa di limiti strettamente fisici.

[1] Definizione tratta dall'enciclopedia Treccani [1]

Negli anni si sono costruiti transistor sempre più piccoli e veloci, con frequenze di lavoro più alte; questi fattori se portati a livelli estremi, possono provocare instabilità dovute alle alte temperature raggiunte, e dalla sempre maggiore difficoltà nel dissipare il calore prodotto. Per soddisfare il bisogno di potenza di calcolo, quindi, sono nati processori *multi-core*, in grado di eseguire flussi di istruzioni in maniera parallela.

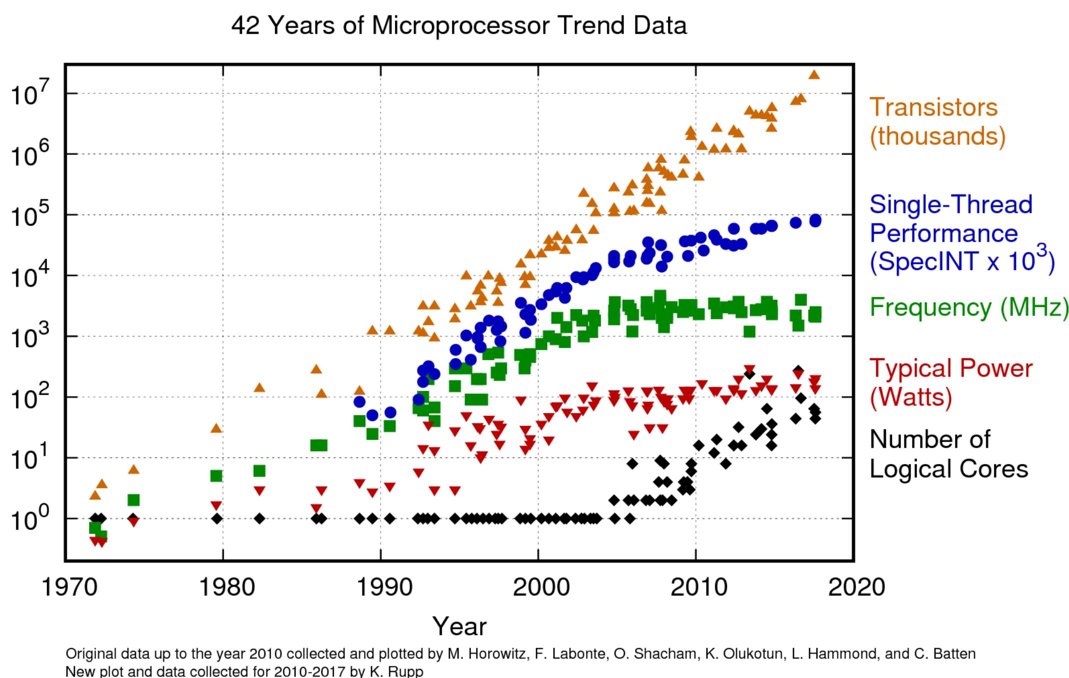


Figura 1.1: Evoluzione dei processori.

In figura 1.1 è mostrato l'andamento dell'evoluzione dei processori nella storia. Nei primi anni CPU *multi-core* non erano ancora giustificabili visti gli incrementi di prestazioni che si riuscivano ad ottenere sulla singola unità di calcolo. Imbattendosi poi in limiti fisici, si è iniziato a progettare CPU *multi-core*. Da qui nasce il bisogno di riuscire a sfruttare processori che permettono l'esecuzione parallela di molteplici flussi di istruzioni.
È necessario trovare delle tecniche di programmazione che permettano di suddividere un problema in sotto-parti quanto più possibili indipendenti, in modo da computarle in maniera parallela con l'utilizzo di più unità di calcolo. Per alcuni problemi questo può essere un processo molto facile, mentre per altri richiede particolare abilità. Per questo motivo il software parallelo è piuttosto raro, mentre architetture multi processore ad oggi sono la normalità. Nello scrivere software parallelo è importante tenere conto oltre che della natura

del problema, anche la struttura dell'hardware di cui si dispone. Esistono tre principali famiglie di architetture per il calcolo parallelo:

- Architetture a memoria condivisa: unità di calcolo multi-processore che condividono la stessa memoria.
- Architetture a memoria distribuita: unità di calcolo indipendenti con memoria individuale, comunicano tra loro mediante una rete inviandosi messaggi espliciti.
- Architetture ibride: nodi basati su architetture a memoria condivisa, interconnessi tramite una rete.

Processor / Memory wall

Un fenomeno di cui bisogna tenere conto è il divario di prestazioni tra processori e memorie. Infatti il miglioramento della velocità delle memorie non è stato in grado di seguire quello dei processori, generando un vero e proprio collo di bottiglia nelle attuali architetture. Per cercare di colmare questo divario, nei processori moderni una buona parte della superficie del processore è dedicata alla memoria *cache*. Questa memoria, molto più preformate e costosa, fa da tramite tra memoria centrare e processore, migliorando notevolmente le prestazioni nella maggior parte dei casi d'uso. In linea di massima se un certo dato viene letto o scritto con grande frequenza, esso viene mantenuto in memoria *cache* per ovviare ai lenti accessi in memoria centrale.

1.2 OpenMP

In questa sezione è descritto in breve il supporto che OpenMP fornisce alla programmazione multi-core.
OpenMP (Open Multi-Processing) è un modello di programmazione parallela per sistemi a memoria condivisa. Consiste in un insieme di direttive che indicano come certe parti del programma devono essere parallelizzate. Nato nel 2005 supporta i linguaggi C, C++, Fortran ed è in continua evoluzione.
OpenMP non è un sistema che parallelizza il codice in automatico, e non garantisce incrementi di prestazioni. Questi fattori dipendono dall'abilità del programmatore: valutare dove e in che modo sia possibile suddividere un determinato problema è un processo astratto, non banale, che per molte sue parti richiede ancora l'intervento umano.

Le direttive solitamente iniziano con *#pragma omp* e non sono altro che istruzioni per il preprocessore. Il codice così processato risulterà in una serie di fork e join di processi, dove il thread master (quello creato all'avvio dell'applicazione) si occupa di creare e distruggere i thread nei vari punti di parallelizzazione del programma. Come si mostra in figura 1.2 raggiunto un punto di divisone vengono creati più thread per l'esecuzione parallela, al termine si esegue un join tramite una barriera implicita. L'esecuzione continua sul master thread.

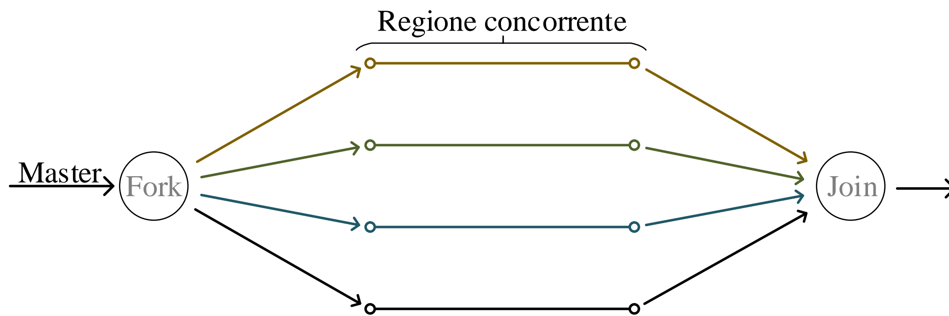


Figura 1.2: Schema di esecuzione OpenMP.

Nell'esempio sottostante quando il master raggiunge il #pragma crea un insieme di thread, ognuno eseguirà una copia della regione del codice (in questo caso il corpo del ciclo for). OpenMP si occupa di inizializzare i per ogni thread, e incrementarla del giusto valore.

```
1  int i, n = 100;
2  int* a, b;
3  #pragma omp parallel for schedule(static, 4)
4  //fork
5  for (i = 0; i < n; i++)
6  {
7      a[i] += a[i] * b[i];
8  }
9  //join
```

L'opzione *schedule(static, 4)* indica come si divide il lavoro tra i vari thread, in questo caso è suddiviso staticamente con una dimensione di blocco di lavoro pari a quattro. La figura 1.3 mostra più chiaramente il concetto di *scheduling* e blocco di lavoro. Il tipo di *scheduling* può essere statico o dinamico: se statico

le iterazioni da fare sono assegnate ai thread prima che il ciclo sia eseguito; se dinamico le iterazioni sono assegnate ai thread man mano che il ciclo esegue. La dimensione del blocco di lavoro indica quanto finemente si deve suddividere il lavoro.

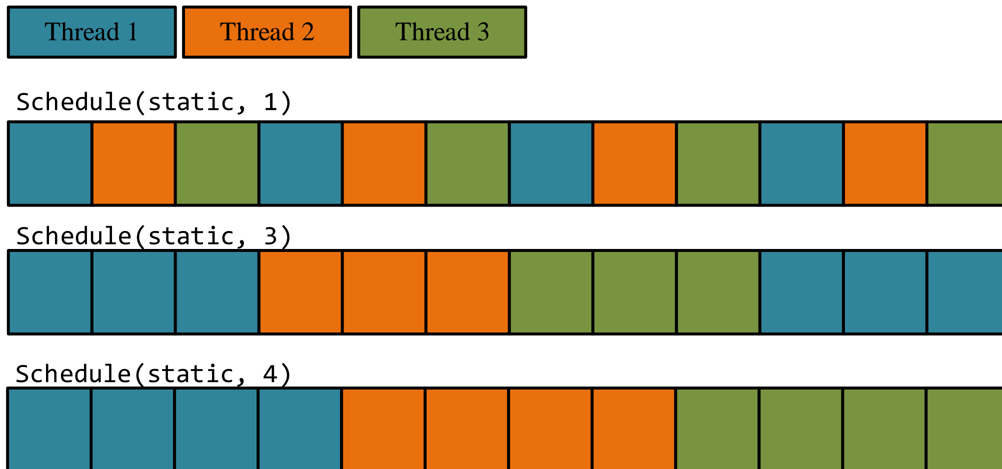


Figura 1.3: Esempio di scheduling statici in OpenMP.

Ci sono molti altri tipi di direttive e di opzioni da poter usare che coprono svariati casi d'uso.

1.3 Estensioni SIMD

Le estensioni SIMD sono tipi di istruzioni speciali, fornite a livello hardware dal processore, che permettono di eseguire un singola istruzione su molteplici dati (come dal nome: Single Istruction Multiple Data). Ad esempio si può applicare l'operazione somma tra due vettori (la dimensione di essi dipende dal processore), il risultato è un terzo vettore equivalante alla somma dei due. L'operazione in questo caso richiede circa lo stesso tempo di eseguire una singola somma. Nella figura 1.4 è visualizzato l'esempio appena descritto.

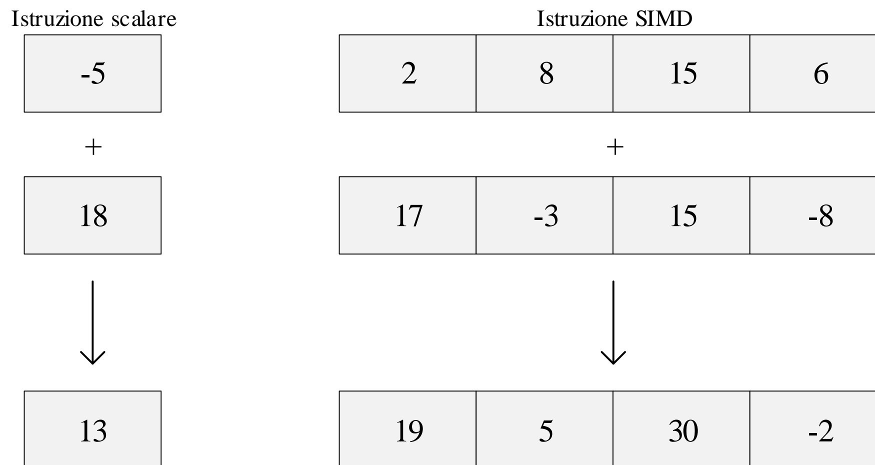


Figura 1.4: Istruzione scalare e istruzione SIMD.

Ci sono vari modi per sfruttare le SIMD fornite dal proprio hardware:

- Auto vettorizzazione: il compilatore esaminando il codice, cerca di trovare operazioni sostituibili con istruzioni SIMD, in modo da generare un'applicazione più veloce. Questa tecnica è totalmente trasparente al programmatore, che comunque può dare dei consigli al compilatore. Allo stesso tempo vi sono degli svantaggi tra cui: non sempre si ottiene la soluzione migliore, e in altri casi il compilatore potrebbe ignorare del tutto un'ottimizzazione anche se in pratica sarebbe stata possibile.
- Tipi di dato vettore: Essendo supportati dal compilatore sono dipendenti da esso. Si tratta di vettori che contengono variabili numeriche (float, int, unsigned int, ...), di dimensione limitata (generalmente da 2 a 8 elementi), a cui si possono applicare operazioni di confronto, logiche e aritmetiche. Il compilatore poi si occuperà di usare le giuste istruzioni SIMD per computare le operazioni sui vettori.
- SIMD instrinsics: In questo caso si sfrutta l'architettura SIMD utilizzando direttamente le funzioni fornite dal compilatore. In questo caso si crea una dipendenza alla piattaforma su cui si esegue il codice (ogni processore ha il suo set di istruzioni SIMD). È la tecnica che si userà in seguito.

```c
//somma gli array b, c salvandone il risultato in a
void sum(float* a, float* b, float* c, int n)
{
    for(int i = 0; i < n; i += 4)
    {
        //carico b, c
        float32x4_t b_vect = vld1q_f32(&b[i]);
        float32x4_t c_vect = vld1q_f32(&c[i]);
        //b + c
        float32x4_t a_vect = vaddq_f32(b_vector, c_vector);
        //salvo il risultato in a
        vst1q_f32(&a[i], a_vector);
    }
}
```

Esempio dell'utilizzo di SIMD instrinsics (ARM NEON)

Da tener presente che le istruzioni SIMD vengono eseguite su unità di calcolo singole, ovvero ogni core del processore ha la possibilità a livello hardware di eseguire le istruzioni SIMD. Il loro utilizzo infatti può essere combinato con una programmazione multi-core come visto nella sezione precedente.

1.4 Valutare le prestazioni

In questa sezione si descrivono alcuni concetti base per poter valutare i miglioramenti ottenuti nella parallelizzazione di un algoritmo.

Speedup

Definiamo con il nome *Speedup* l'incremento di prestazioni in termini di tempo e in funzione del numero di unità di calcolo utilizzate:

$$S(p) = \frac{T_{seriale}}{T_{parallelo}(p)} \tag{1.1}$$

In pochi casi particolare si ottiene $S(p) = p$ ma solitamente accade $S(p) \leq p$. Non sempre più unità di calcolo significano maggior prestazioni, infatti se l'algoritmo in questione ha una parte sequenziale anche minima allora esiste

un limite allo speedup che si può ottenere:

$$T_{parallelo}(p) = \alpha T_{seriale} + \frac{(1-\alpha)T_{seriale}}{p} \tag{1.2}$$

dove α è la frazione di tempo che il programma spende nell'esecuzione seriale. Da qui si nota che anche all'aumentare di p il termine $\alpha T_{seriale}$ rimane invariato, quindi:

$$\lim_{p \to +\infty} S(p) \to \frac{T_{seriale}}{\alpha T_{seriale}} = \frac{1}{\alpha} \tag{1.3}$$

Questa conclusione descritta dalla legge di Amdahl determina che il massimo speedup è determinato dalla parte seriale dell'algoritmo e tende in maniera asintotica a $1/\alpha$.

Efficienza

Questa misurazione indica quanto è efficiente un'applicazione all'aumentare del numero di unità di calcolo utilizzate. Esistono due tipi di misurazioni:

- Strong scaling efficiency: descrive l'efficienza nell'aumentare il numero di processori mantenendo un carico di lavoro costante:

  $$E(p) = \frac{S(p)}{p} \tag{1.4}$$

  Una buona pratica è quella di trovare un numero di processori che permettano di ottenere un tempo accettabile per l'esecuzione, senza avere uno spreco elevato di risorse causato dall'overhead della parallelizzazione. Bisogna anche tenere conto della legge di Amdahl che non permette speedup illimitati.

- Weak scaling efficiency: descrive l'efficienza nell'aumentare il numero di unità di calcolo mantenendo il carico di lavoro su ognuna di esse costante:

  $$W(p) = \frac{T_1}{T_p} \tag{1.5}$$

  T_1 è il tempo richiesto per completare un compito con un processore. T_p è il tempo richiesto per completare p compiti con p processori. In questo caso si ottiene una buona efficienza se si riesce a mantenere costante il tempo di esecuzione sebbene il carico di lavoro aumenti, utilizzando più processori.

Capitolo 2

Machine Learning

In questo capitolo si introduce brevemente il Machine Learning, con una particolare attenzione al Deep Learning e alle reti neurali artificiali. In seguito viene descritta la struttura di una rete neurale artificiale internamente connessa, e le operazioni che si svolgono durante la fase di addestramento. Lo sviluppo di questa rete neurale viene discussa nel capitolo 5, tenendo conto di come implementarla in modo adeguato a seconda degli obiettivi da raggiungere. Si è voluto implementare un algoritmo di questo genere perché si presta molto bene nell'effettuare sperimentazioni di calcolo parallelo, e inoltre appartiene al settore del machine learning che ha conquistato grande interesse al giorno d'oggi.

2.1 Introduzione al Machine Learning

Per risolvere un problema è necessario scrivere un algoritmo con una sequenza d'istruzioni che, dati un insieme di parametri in ingresso, calcolano il risultato. Per alcuni problemi tuttavia questo non è possibile, o meglio, sarebbe incredibilmente difficile da implementare. Si può pensare, per esempio, di dover scrivere un algoritmo che data un'immagine deve saper indicare se in essa è presente una persona o meno: l'algoritmo in questione deve essere capace di tener conto di moltissime variabili e circostanze, rendendo l'implementazione di un algoritmo "classico" ardua e poco flessibile. Qui entra in gioco la materia del machine learning che si fonda su un principio di base: quello che manca in conoscenza, lo si recupera nella disponibilità di dati posseduti. Questo principio calza molto bene al giorno d'oggi, dove spesso si dispone di grosse quantità di dati, che se sfruttate correttamente possono essere fonte di una

grande quantità d'informazione e conoscenza. L'obiettivo del machine learning è quello d'imparare la struttura e le caratteristiche comuni che vi sono in un ampio insieme di esempi, generando modelli che permettono di analizzare dati mai visti prima.
L'addestramento di un sistema di machine learning può avvenire in due modi:

- Supervised learning: l'addestramento è effettuato su un insieme di dati "etichettati", o meglio, dati di cui si conosce l'informazione che rappresentano. Per esempio se un sistema di machine learning viene addestrato nel classificare un certo insieme d'immagini, nel caso del *supervised learning* si conosce in anticipo la classe di appartenenza delle immagini di input. Questo permette di correggere l'algoritmo ogni qualvolta sbagliasse.

- Unsupervised learning: questo tipo di addestramento viene effettuato con dati di input senza etichette. È compito del sistema di machine learning capire le caratteristiche comuni dei dati forniti in input, ed elaborare un qualche tipo di conoscenza.

2.2 Deep Learning

Il deep learning ad oggi è una delle migliori e più promettenti tecnologie per quanto riguarda il settore del machine learning. Questo tipo di approccio viene utilizzato in moltissimi settori: un esempio può essere quello della computer vision. Il deep learning è uno specifico sottoinsieme della famiglia di algoritmi del machine learning, il quale è un sottoinsieme della più generica intelligenza artificiale (come mostrato in figura 2.1). Il deep learning si differenzia dal machine learning perché utilizza la struttura delle reti neurali all'interno dei propri algoritmi. Il punto di forza di questi sistemi è che una volta definita la struttura di una rete neurale artificiale, e costruito un insieme di dati su cui lavorare, il sistema riesce a estrarre automaticamente le caratteristiche essenziali dei dati, generando un modello in grado di generalizzare su nuovi esempi. L'addestramento di questi sistemi viene considerato da alcuni un'arte più che una disciplina esatta per via della moltitudine di scelte da effettuare in fase di progettazione.

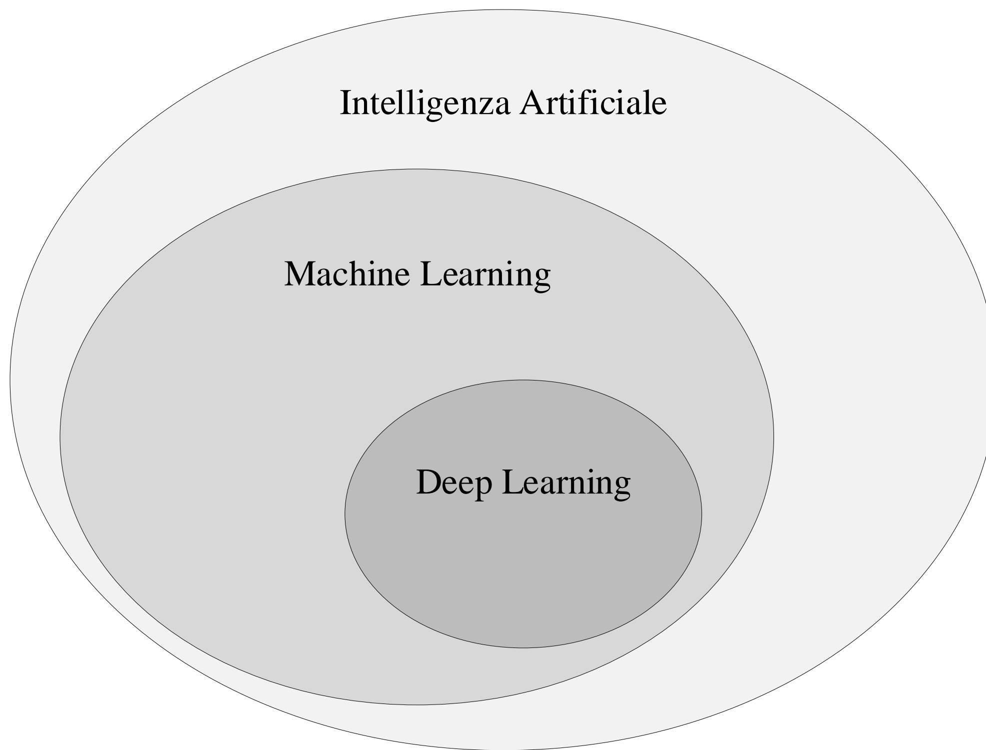


Figura 2.1: IA, Machine Learning e Deep Learning.

La struttura delle reti neurali è ispirata alla biologia del cervello umano, infatti è una struttura composta da molteplici livelli (da qui il nome deep), composti di nodi (neuroni) interconnessi fra loro.
Esistono diverse macro-strutture di reti neurali artificiali: la più semplice è la *Deep Feedforward Network* nella quale i dati di input transitano in una sola direzione: dal livello di input fino a quello di output. Esistono altre strutture per molti casi d'uso, come ad esempio le *Recurrent Neural Networks* (RNN) dove l'output di questa rete non dipende solo dall'input corrente, ma anche da quelli precedenti, perciò il flusso d'informazioni può muoversi in più direzioni. Di queste macro-strutture esistono molte specializzazioni: ad esempio una *Long Short-Term Memory* è un tipo specifico di RNN, e la *Convolutional Neural Network* (CNN) è un tipo specifico di *Deep Feedforward Network*.

2.3 Dense neural network

Dense neural network o *fully-connected neural network* o, in italiano, rete neurale interamente connessa, è una struttura specifica di rete neurale utilizzata

nel deep learning, e appartiene alla categoria delle *Deep Feedforward Network*. Si compone di uno strato (livello) di input e uno di output, tra essi vi sono uno o più livelli nascosti (interni) che eventualmente imparano le caratteristiche dei dati forniti in input. Il nome deriva dal fatto che questi strati fra loro sono connessi in maniera completa, ovvero, ogni nodo (neurone) di uno strato è connesso a tutti i nodi di quello precedente. In figura 2.2 è visualizzato uno schema di rete neurale interamente connessa.

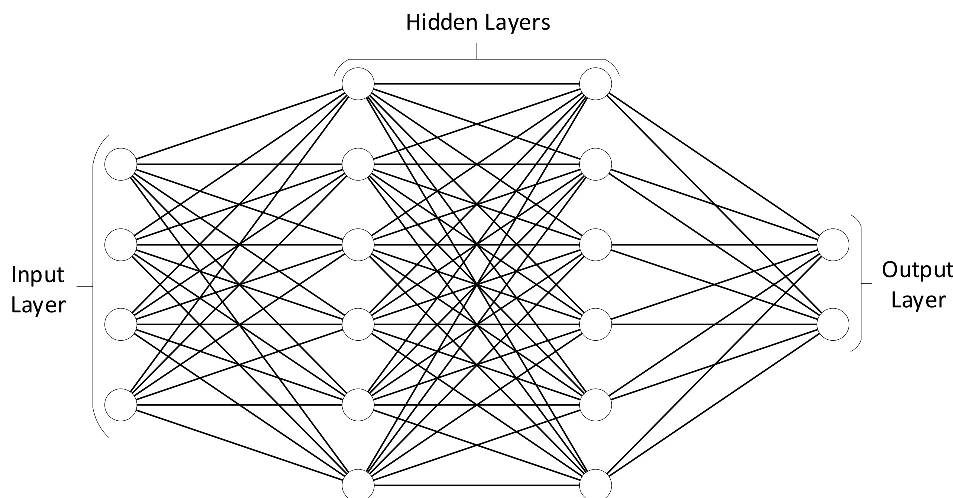


Figura 2.2: Dense neural network.

Questa particolare struttura di rete neurale è di tipo *general purpose* visto che non vi è nessun pattern d'interconnessione tra i neuroni, il che significa che non si fanno ipotesi sulle caratteristiche dei dati in input.

Struttura interna

Ogni nodo (neurone) è composto da diverse parti: le connessioni agli altri nodi, un bias e una funzione di attivazione. Il compito di un neurone è quello di prendere un insieme di valori numerici e trasformarli attraverso queste componenti in un singolo valore di output. La struttura di un neurone è mostrata in figura 2.3.

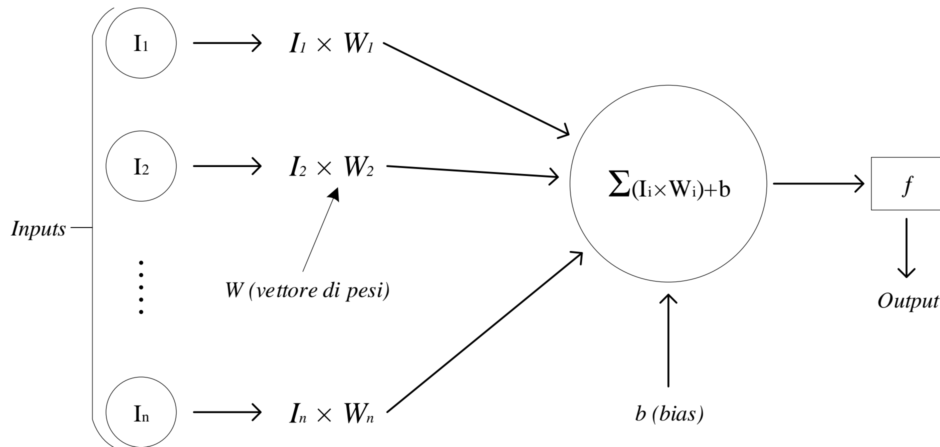


Figura 2.3: Struttura interna di un neurone.

Ogni connessione è rappresentata da un peso, che indica la "forza" di connessione con il nodo del livello precedente. Il bias è un offset che permette di traslare la funzione di attivazione, caratteristica utile nella fase di addestramento. Le funzioni di attivazione servono per introdurre un elemento di non-linearità nel sistema; quelle comunemente usate sono: Sigmoid $1/(1+e^{-x})$, Tanh $(1-e^{-2x})/(1+e^{-2x}))$, Re-Lu $max(0,x)$ e Softmax $e^{x_k}/(\sum_i e^{x_i})$. Questo modello di neurone è uno dei più comuni in letteratura, sebbene non sia l'unico.

La fase dove la rete neurale trasforma l'input in output è chiamata *feedforward*: in questa fase, dopo aver assegnato al livello di input i valori da analizzare, si calcola l'output di ogni neurone partendo dal primo livello nascosto, fino al livello di output. Ogni neurone di un livello può computare il proprio output indipendentemente dagli altri. L'espressione che si utilizza nel calcolo è la seguente:

$$O = f(b + \sum_{i=1}^{k} (W_i \times I_i)) \tag{2.1}$$

dove:

- O è l'output del neurone.
- f è la funzione di attivazione.
- W è il vettore di pesi.
- I è il vettore dei valori di input.

- W e I hanno la stessa lunghezza k.
- b è il bias.

Ogni neurone della rete neurale funziona nello stesso modo, escludendo eventualmente i neuroni di input, i quali non calcolano il valore di output ma gli viene assegnato all'inizio della fase di *feedforward*.
L'intera rete neurale può essere quindi definita come una funzione complessa con n variabili di input e m di output. Si definisce ora una funzione di costo (*loss function*), che misura quanto sia precisa la rete neurale artificiale. Questa funzione si definisce attraverso le m variabili di output e in base al comportamento che si vuole insegnare alla rete neurale. Un esempio di funzione di costo può essere la seguente:

$$\frac{1}{N}\sum(y-y')^2 \tag{2.2}$$

dove:

- y è la previsione attesa (nel caso del supervised learning è ricavata dal valore dell'etichetta)
- y' è la previsione della rete neurale.
- N è il numero di esempi presenti nell'insieme di input.

Questa particolare funzione prende il nome di *Mean absolute error* (MAE). In altri termini, è la media degli errori su ogni singolo esempio, calcolati tramite il quadrato della differenza tra valore atteso e valore previsto.
L'intera fase di training della rete neurale ha lo scopo di minimizzare la funzione di costo: bisogna trovare un modo per modificare le variabili della rete neurale (pesi e bias) in modo che la funzione di costo converga in un tempo accettabile al suo minimo. I pesi e i bias sono inizialmente scelti con valori pseudo-casuali, rendendo la funzione di costo molto "lontana" dal suo minimo. Si potrebbero permutare le variabili della rete neurale con valori casuali fino a che non si trova un valore di costo accettabile, questo richiederebbe un tempo spropositato per via del numero di variabili in gioco.
L'algoritmo chiamato *backpropagation* permette di calcolare efficientemente le "direzioni" che le variabili devono seguire per raggiungere il minimo della funzione di costo. L'algoritmo calcola attraverso le derivate parziali delle funzioni usate nella fase di *feedforward* le "direzioni" (gradienti). Il valore del gradiente può essere assegnato anche a un intero neurone, questo indica come si vorrebbe variare l'output del neurone affinché la funzione di costo diminuisca.

I gradienti iniziali (quelli assegnati al livello di output) non sono calcolati attraverso la tecnica di *backpropagation*, ma sono ricavati tramite la derivata della funzione di costo; infatti i gradienti del livello di output indicano come devono variare gli m valori di output affinché la funzione di costo assuma un valore più piccolo. Non appena si sono assegnati i gradienti ai neuroni del livello di output, essi si possono propagare in tutta la rete utilizzando l'algoritmo di *backpropagation*. Una volta terminata questa fase ogni variabile della rete neurale è accompagnata da un gradiente, che indica come essa deve variare al fine di ridurre la funzione di costo.
Per dare un'idea più chiara di cosa rappresenta il gradiente e perché viene utilizzato il calcolo delle derivate, in figura 2.4 è mostrato un ipotetico andamento della funzione di costo al variare di un solo peso W interno a una rete neurale.

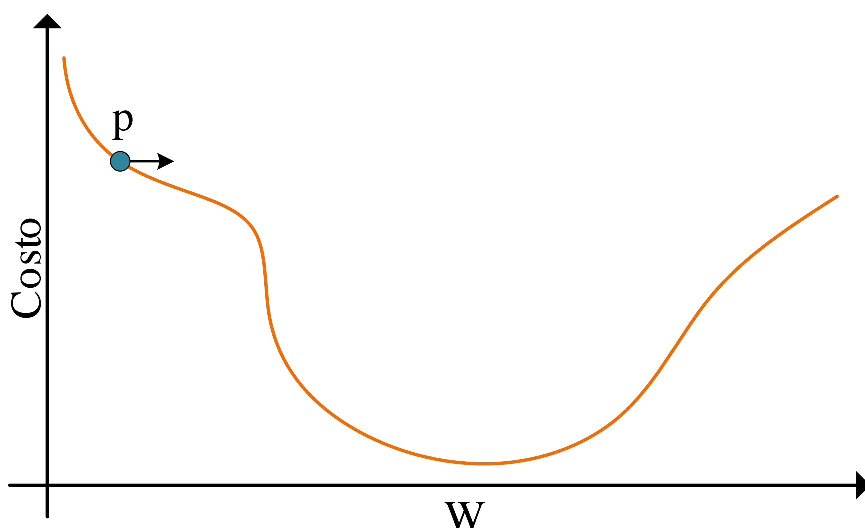


Figura 2.4: Costo in funzione di un peso

Nel momento iniziale il valore del peso si trova in corrispondenza del punto p. Calcolando la derivata in funzione del peso, nel punto p il valore di essa risulta negativo, ciò indica che la "discesa" è verso destra. Il gradiente, in questo caso, è il valore della derivata in funzione di W nel punto p. In generale se il gradiente è negativo bisogna incrementare il peso, se il gradiente è positivo bisogna decrementarlo ($W = W - G \times \alpha$). Questi incrementi vanno applicati con un coefficiente α chiamato *learning rate*. Si può intuire che con

un valore α piccolo il punto p impiegherebbe molte iterazioni per raggiungere il suo minimo, con un valore elevato, invece, p si potrebbe spostare sempre troppo a destra o troppo a sinistra "saltando" il minimo della funzione.
I gradienti calcolati si possono applicare tramite diverse tecniche: *Stochastic gradient descent* (SGD) prevede di applicare i gradienti non appena vengono calcolati, mentre la tecnica *gradient descent* fa una media dei gradienti calcolati nelle fasi di *backpropagation* per tutto il dataset di addestramento, per poi applicarli al termine. *Gradient descent* fa in modo che la funzione di costo converga al suo minimo in maniera più lineare, ma è molto più pesante da un punto di vista computazionale.
Le tecniche di *backpropagation* e *stochastic gradient descent* sono trattate nel dettaglio nel capitolo 5 dove se ne descrive il funzionamento parlando dell'implementazione della la rete neurale.

2.3.1 Overfitting e tecniche di risoluzione

Esiste un fenomeno nei sistemi di machine learning chiamato overfitting: si manifesta quando la rete neurale si adatta completamente all'insieme dei dati forniti in fase di addestramento, non riuscendo a generalizzare su esempi futuri. Una tecnica che permette di scoprire se si verifica l'overfitting, è quella di dividere il dataset di apprendimento in due categorie: training-set e test-set. Nel training-set sono presenti i dati che si forniscono al sistema nella fase di addestramento, mentre il test-set serve per testare il sistema su esempi mai visti prima. Il test-set e il training-set devono contenere dati della stessa natura, ma devono essere composti da esempi diversi. Nella figura 2.5 si mostra la curva, in termini di costo, che si ha in una fase di training colpita dall'overfitting.

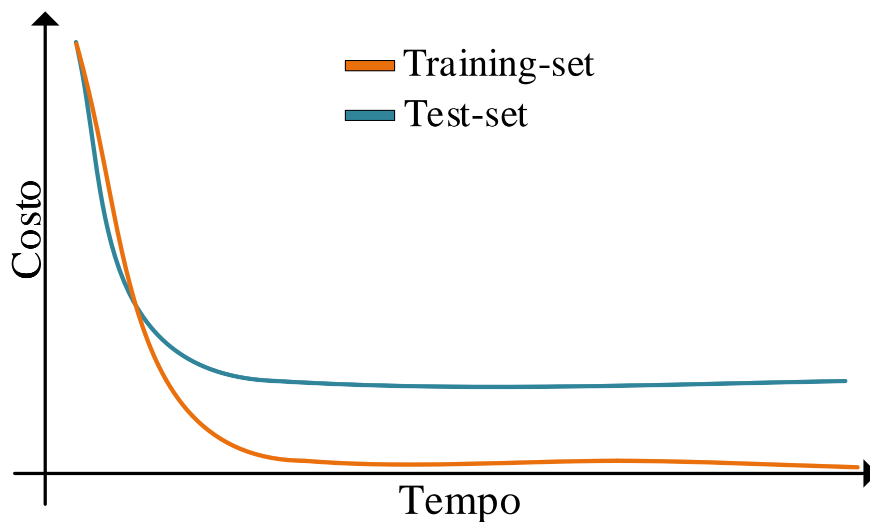


Figura 2.5: Effetto overfitting

L'overfitting si manifesta soprattutto in reti neurali di grandi dimensioni, dove la complessità della rete permette un adattamento completo al training-set. Ciò si può evitare avendo un ampio dataset di training, in questo modo la rete neurale fatica ad adattarsi per via della moltitudine di esempi. Se ciò non bastasse si possono applicare tecniche di regolarizzazione, tra cui: regolarizzazione L1, L2, e dropout. Le regolarizzazioni L1 e L2 prevedono di minimizzare insieme alla funzione di costo vista in precedenza, anche la complessità della rete neurale. Nel caso della regolarizzazione L1 la complessità è definita come il modulo della somma di ogni singolo peso, nella regolarizzazione L2, invece, è definita come la somma dei quadrati di ogni peso. Queste regolarizzazioni hanno l'obbiettivo di ridurre la complessità della rete neurale, in modo da non farla adattare perfettamente al training-set, preservando la sua capacità di generalizzare. La tecnica del dropout invece prevede di disattivare casualmente alcuni neuroni durante la fase di addestramento. Questo induce un rumore interno alla rete neurale, evitando, in alcuni casi, l'effetto overfitting.

In questo progetto verrà implementata una *Deep Feedforward Networs* del tipo *fully-connected neural network* con lo scopo di classificare le cifre numeriche scritte a mano libera. Il training che si esegue è di tipo supervisionato e si

usa la tecnica della *backpropagation* associata a quella di *stochastic gradient descent* nella fase di training. Si utilizzano tecniche per prevenire l'overfitting tra cui: regolarizzazione L1 e ampliamento del dataset. I dati grezzi di input sono immagini rappresentanti cifre numeriche, con associate le corrispondenti etichette. L'output della rete neurale è la classificazione di una cifra in una delle 10 classi, dove ognuna rappresenta un valore numerico (0-9).

Capitolo 3

Componenti utilizzate

In questo capitolo vengono descritte le componenti usate per effettuare le prove sperimentali discusse nei successivi capitoli. Si è utilizzato il system on a chip (SoC) Raspberry Pi e un processore Intel Xeon. A seguire si descrivono le principali caratteristiche dell'hardware, alcuni impieghi possibili, e le capacità di effettuare calcolo parallelo su queste componenti.

3.1 Raspberry Pi

Il Raspberry Pi è un computer costruito interamente su una singola scheda dalle dimensioni di una carta di credito, basato su un processore di architettura ARM (Advanced RISC Machine).

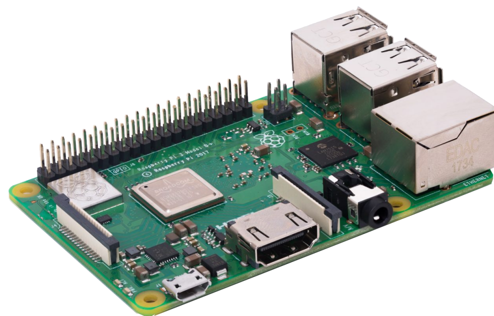


Figura 3.1: Raspberry Pi 3 Modello B+

Nato con l'intento di promuovere l'insegnamento dell'informatica nelle scuole [4], ha poi ottenuto molta popolarità grazie alla comunità fortemente attiva

sul Web, permettendo al prodotto di diffondersi ed evolversi in maniera molto rapida. Gli aspetti positivi nell'usare questi piccoli calcolatori sono principalmente il basso costo e il basso consumo energetico, per questo sono adottati anche in progetti di Internet of Things e di Sistemi Embedded. La scheda nasce nel 2012 e viene supportata da diversi sistemi operativi GNU/Linux per architettura ARM: attualmente il più utilizzato è Raspbian, derivato dalla distribuzione Debian. In figura 3.1 è mostrato l'ultimo modello di Raspberry Pi (Terza generazione, Modello B+).

Caratteristiche

Tutte le versioni di questa scheda utilizzano una microSD come memoria di massa per contenere il sistema operativo. Ad oggi ogni versione monta il processore grafico VideoCore IV in grado di fornire 24 GFlops; per molti altri aspetti invece la scheda Raspberry Pi ha subito diverse evoluzioni migliorative nel corso degli anni. Le principali caratteristiche [11] sono riportate nella seguente tabella:

Modello	CPU	ARM NEON	RAM	GPU	Ethernet	Consumo (max)	Prezzo
A	single-core 700 MHz	No	256 MB	VideoCore IV	Via USB	1.5 W	25$
A+	single-core 700 MHz	No	512 MB	VideoCore IV	Via USB	1 W	20$
B	single-core 700 MHz	No	512 MB	VideoCore IV	10/100 Mbit/s	3.5 W	35$
B+	single-core 700 MHz	No	512 MB	VideoCore IV	10/100 Mbit/s	1.8 W	35$
2 B	quad-core 900 MHz	Si	1 GB	VideoCore IV	10/100 Mbit/s	4.3 W	35$
3 B	quad-core 1.2 GHz	Si	1 GB	VideoCore IV	10/100 Mbit/s	6.5 W	35$
3 B+	quad-core 1.4 GHz	Si	1 GB	VideoCore IV	10/100/1000 Mbit/s (~300 Mbits/s)	6 W	35$
Zero	single-core 700 MHz	No	512 MB	VideoCore IV	No	1.8 W	5$

I valori qui riportati sono solo quelli essenziali da un punto di vista computazionale, trascurando molti altri aspetti di cui questa scheda è composta. Dalle informazioni riportate in tabella si nota che la scheda ha subito vari miglioramenti hardware, mantenendo comunque un prezzo modesto. L'ultima versione ha fatto un salto di qualità notevole, introducendo un'interfaccia ethernet gigabit e un processore quad-core a 1.4 Ghz (Cortex-A53) dotato inoltre dell'architettura ARM NEON. Con un consumo di solo 6 W (nominali) questa scheda potrebbe risultare molto interessante per alcune applicazioni. Se si vuole trovare un punto debole del SoC, potrebbe essere la scarsa quantità di memoria dedicata, che arriva solo a 1 GB; inoltre viene condivisa con il processore grafico, riducendone ulteriormente la grandezza.

L'ultima versione (Raspberry Pi 3 Model B+) sarà quella utilizzata nelle sperimentazioni dei capitoli a seguire. Questa scheda ha capacità di calcolo parallelo non indifferenti, infatti vi è la possibilità di sfruttare l'architettura multi-core e l'architettura SIMD. Si può tenere conto anche dell'uso del processore grafico VideoCore IV, capace di fornire un throughput non indifferente. Si possono eventualmente costruire, con cifre modeste, cluster di Raspberry Pi, interconnessi da una rete ethernet. Un progetto simile, di nome Raspèin [13] infatti è già stato sviluppato presso l'università di Bologna. In questa trattazione ci si limita a scoprire le potenzialità di una singola scheda Raspberry Pi, tramite una programmazione parallela che tiene conto della struttura dell'hardware.

3.2 Processori x86

I processori della famiglia x86 sono i più comunemente usati in ambiti desktop e server. Il nome deriva dal set d'istruzioni che supportano. Essendo, a differenza delle CPU ARM, nella categoria di processori CISC (Complex instruction set computer) sono processori strutturalmente più complessi. Solitamente queste CPU sono in grado di fornire prestazioni più elevate di quelle che possono essere ricavate da un processore ARM, essendo caratterizzati però da un prezzo e un consumo energetico maggiore.
Nei test che si andranno a effettuare si utilizza un processore Intel Xeon E5-2603 v4. Questo processore è dotato di 6 core fisici, 15 MB di cache e una frequenza di lavoro pari a 1.7 Ghz; il consumo nominale è di 85 W.

Figura 3.2: Intel Xeon E5-2603 v4

CISC e RISC

CISC (complex instruction set computer) e RISC (reduced instruction set computer) sono due tipi di architetture di microprocessori. I processori basati su architettura RISC prediligono un ridotto e semplice set d'istruzioni per garantire un'esecuzione rapida di tali, inoltre garantisce una ridotta struttura interna del microprocessore. In contrapposizione troviamo il modello CISC, che indica la costruzione di un microprocessore capace di eseguire operazioni anche complesse tramite singole istruzioni.
L'approccio che l'architettura CISC adotta è quello di completare un certo task nel minor numero d'istruzioni possibili, lasciando all'hardware il compito di decodificare ed eseguire le varie operazioni da svolgere. Questo permette di avvicinare il divario che c'è tra linguaggio macchina e linguaggio ad alto livello, ottenendo quindi, programmi di dimensioni ridotte.
L'architettura RISC invece punta a usare solo istruzioni base che possono venire eseguite in un ciclo di clock, permettendo la realizzazione di microprocessori strutturalmente più semplici, lasciando maggior logica di controllo a livello software.
Le principali differenze tra queste due architetture possono essere riassunte in questa tabella [7]:

CISC

- Enfasi sull'hardware
- Comprende istruzioni complesse
- Istruzioni memory-to-memory
- Uso di poche istruzioni macchina per un dato task
- Uso di molti transistor per realizzare le istruzioni

RISC

- Enfasi sul software
- Solamente istruzioni base
- Istruzioni register-to-register
- Uso di molte istruzioni macchina per un dato task
- Uso dei transistor sopratutto per registri di memoria

Capitolo 4

MNIST database

In questo capitolo viene descritto l'insieme d'immagini scelto per addestrare il sistema di machine learning, il database MNIST, le sue caratteristiche e i motivi per cui è così largamente utilizzato. Infine è descritto un modo per estendere il dataset con lo scopo di avere più esempi nella fase di training di una rete neurale artificiale, vedendo in pratica la tecnica della trasformazione affine sulle immagini.

4.1 Descrizione del dataset

All'interno del MNIST database (Modified National Institute of Standards and Technology database) [10] troviamo un insieme d'immagini rappresentanti cifre numeriche (0-9) scritte a mano libera. Questo dataset è stato costruito apportando alcune modifiche al più datato database NIST [6]. Le cifre sono state scritte da alcuni impiegati di *Census Bureau* e alcuni studenti di scuola superiore, poi distribuite in maniera accurata in due insiemi diversi: l'insieme di training composto da 60000 immagini, e l'insieme di test composto da 10000 immagini.

Figura 4.1: Esempio di alcune cifre all'interno del MNIST database

Le cifre presenti nel database sono a scala di grigi a 256 livelli, di dimensioni $28x28$, centrate rispetto alla massa dell'immagine, e ognuna di esse è associata a una etichetta che indica il valore reale della cifra.
In rete è possibile trovare varie sperimentazioni nell'uso di questo dataset con lo scopo di realizzare sistemi in grado di riconoscere il maggior numero d'immagini del test-set, utilizzando il solo training-set come base per sviluppare l'algoritmo. Il risultato migliore è ottenuto mediante una rete neurale convoluzionale (CNN), abbinata a una tecnica di distorsione elastica per ampliare il data-set, ottenendo un errore del solo 0.23% [14].
Dal 2017 è presente un altro database di cifre scritte a mano, EMNIST database (Extended MNIST database) [3] che contiene fino a 350000 immagini. Per questa sperimentazione tuttavia si è utilizzato il MNIST database.

4.2 Qualità del database

MNIST database ha una qualità molto elevata per via della grande quantità d'immagini presenti, e della sorgente di tali cifre, ovvero, circa 500 persone. Inoltre il training-set e il test-set non condividono scrittori comuni, il che garantisce una più alta qualità del database. Se per esempio il training-set fosse composto da cifre scritte da una sola persona, avendo una propria calligrafia, sarebbe molto più probabile che nel momento in cui il sistema dovesse riconoscere cifre di altre persone, effettui previsioni errate; mentre sarebbe un ottimo sistema per riconoscere cifre dello stesso scrittore. Per questo è una buona ca-

ratteristica che il training-set e test-set siano disgiunti per quanto riguarda la sorgente dei dati; il test-set serve proprio per scoprire se il sistema è in grado di generalizzare su esempi diversi.
Questo database d'immagini è un grande aiuto per chi vuole sperimentare tecniche di machine learning senza dover costruire il proprio da zero, tuttavia esso contiene delle cifre molto difficili da riconoscere persino per l'uomo. Nel set di training infatti possiamo trovare cifre come quelle in figura 4.2:

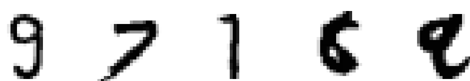


Figura 4.2: Esempio di cifre all'interno del training set

a prima vista possono sembrare normalissime cifre, ma nel momento in cui si guarda l'etichetta abbinata a queste (in ordine sono: 3, 4, 7, 8, 9) è chiaro che nella fase di training della rete neurale queste immagini non apportano un grande beneficio. Essendo comunque in numero veramente ristretto non è affatto un problema. Per quanto riguarda il test-set invece vi è un esempio in figura 4.3:

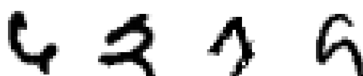


Figura 4.3: Esempio di cifre all'interno del test set

in ordine 6, 2, 7, 9, si intuisce che è pressoché impossibile ottenere una precisione dell'algoritmo di machine learning del 100%, in quanto anche una mente umana non riesce a dare una risposta con assoluta certezza ad alcune cifre. Questi fattori sono chiaramente trascurabili al fine di sperimentazioni pratiche.

4.3 Ampliamento del dataset

Un ampliamento del database può presentare diversi benefici nella fase di training di un sistema di machine learning, tra cui: ci sono molti più esempi su cui imparare, si evita un possibile effetto overfitting. Esistono diverse possibili tecniche di ampliamento di questo dataset, quelle comunemente utilizzate

sono: distorsione elastica e trasformazione affine. In questo caso si utilizza la trasformazione affine per ruotare e ridimensionare le immagini. Certamente non si applicano trasformazioni pesanti, per esempio una rotazioni di 90 gradi, ma lievi cambiamenti in modo che l'immagine mantenga il suo senso e allo stesso tempo sia diversa da quella originale.

Figura 4.4: Esempio di trasformazione affine applicata ad una cifra

Nella figura 4.4 è mostrata una serie di trasformazioni applicate a un'immagine rappresentante il numero tre. La prima è la cifra originale, le due seguenti sono ruotate di ± 15 gradi, le ultime sono scalate di ± 0.15 in altezza e in larghezza. Durante la fase di training della rete neurale artificiale si andranno ad applicare trasformazioni affini casuali alle cifre; nel momento in cui invece viene provata la precisione del sistema attraverso il test-set queste trasformazioni non vengono utilizzate.

4.4 Trasformazione affine

Per trasformazione affine di un'immagine s'intende la sua modifica in un'immagine equivalente che può essere ruotata, scalata e traslata. Possiamo immaginare che l'immagine da trasformare sia situata in uno spazio di due dimensioni, e che ogni pixel sia situato nella corrispondente coordinata $(X, Y) \in \mathbb{N}$ in figura 4.5 è mostrato un esempio. Definizioni:

- (X_n, Y_n): coppia di indici riferiti alla posizione di un pixel nell'immagine finale.
- (X_o, Y_o): coppia di indici riferiti alla posizione di un pixel nell'immagine originale.
- I è una funzione che data una coppia di indici (x, y) restituisce l'intensità del pixel corrispondente.
- Un'immagine di dimensione NxM ha un insieme di NxM pixel, indicizzati con la coppia di valori $(X, Y) \in \mathbb{N}, 0 \leq X < N, 0 \leq Y < M$

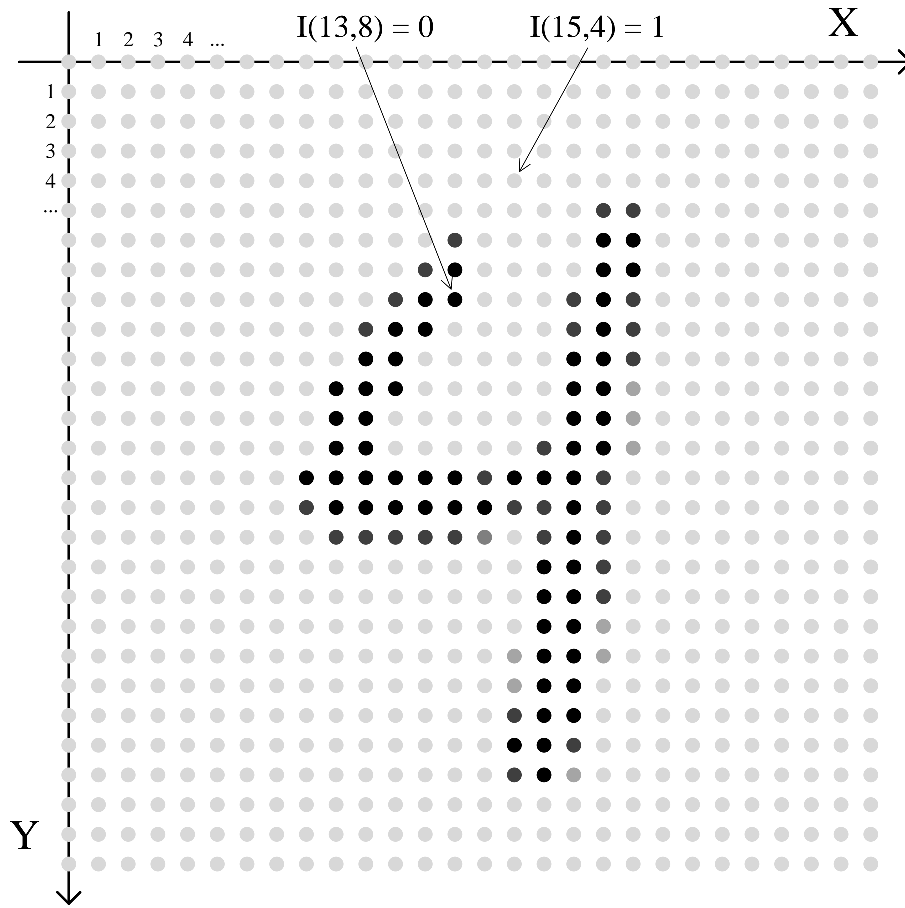


Figura 4.5: Immagine visualizzata su un piano cartesiano

A questo punto bisogna definire una funzione composta dai tre parametri: rotazione, ridimensionamento e spostamento, che trasforma un punto (X, Y) in una nuova coordinata secondo i parametri scelti. Si procede con la trasformazione di tutti i punti nell'insieme NxM per ottenere un nuovo insieme che descrive l'immagine finale. Una possibile funzione per questo procedimento è la seguente [2]:

$$\begin{bmatrix} X_n \\ Y_n \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \forall (X_o, Y_o) \in NxM \quad (4.1)$$

θ: valore della rotazione in radianti

s_x, s_y: valori di ridimensionamento

t_x, t_y: valori di traslazione

ogni coordinata (X_o, Y_o) viene mappata su una nuova coordinata (X_n, Y_n), andando ad inserire il valore $I[X_o, Y_o]$ nella nuova posizione relativa all'immagine finale. Siccome anche l'immagine finale è distribuita in uno spazio bidimensionale partendo dal punto $(0, 0)$ e con le coordinate dei pixel appartenenti a numeri interi, vi sono tre tipi di problemi nell'adottare questa tecnica:

- I valori della nuova coordinata potrebbero non essere numeri interi, per cui non si sa a quale pixel dell'immagine finale dare il valore $I[X_o, Y_o]$.
- Alcuni pixel dell'immagine iniziale vengono mappati fuori dai bordi $[0, N[; [0, M[$ della nuova immagine.
- Alcuni pixel dell'immagine finale potrebbero non venire assegnati, lasciando delle regioni vuote.

Per risolvere questi problemi si applica una trasformazione inversa, ovvero dalla coordinata del pixel della nuova immagine si vuole ottenere la coordinata riferita all'immagine di partenza:

$$\begin{bmatrix} X_o \\ Y_o \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 \\ 0 & \frac{1}{s_y} \end{bmatrix} \begin{bmatrix} cos(-\theta) & sin(-\theta) \\ -sin(-\theta) & cos(-\theta) \end{bmatrix} \left(\begin{bmatrix} X_n \\ Y_n \end{bmatrix} - \begin{bmatrix} t_x \\ t_y \end{bmatrix} \right) \forall (X_n, Y_n) \in NxM \tag{4.2}$$

In questo modo iterando la funzione per tutte le coppie (X_n, Y_n), si ottengono altrettante coppie (X_o, Y_o), si procede con la copia del valore in coordinate (X_o, Y_o) nel pixel (X_n, Y_n) dell'immagine finale. Utilizzando questa tecnica ci si può imbattere in tre casistiche:

- La coordinata (X_o, Y_o) cade fuori dall'immagine iniziale: si utilizza una valore di background prestabilito. $I[X_n, Y_n] = Background$
- La coordinata (X_o, Y_o) ha valori interi e cade dentro l'immagine: si copia il valore del pixel corrispondente. $I[X_n, Y_n] = I[X_o, Y_o]$
- La coordinata (X_o, Y_o) non ha valori interi e cade dentro l'immagine: si effettua un'interpolazione di Lagrange tra i quattro pixel adiacenti per determinare il valore del nuovo pixel. $I[X_n, Y_n] = Lagrange(X_o, Y_o)$

In questo modo l'immagine sarà ruotata sempre con centro in posizione $(0, 0)$, se si desidera spostare il punto di rotazione è sufficiente traslare l'immagine, ruotarla e poi traslarla nuovamente nella posizione iniziale. Nel capitolo 5 si descrive un'implementazione che prevede anche la scelta del centro di rotazione e della risoluzione dell'immagine finale.

Interpolazione di Lagrange

Il valore del pixel $I[X_n, Y_n]$ viene determinato dai quattro pixel più vicini alla coordinata (X_o, Y_o) calcolata precedentemente.

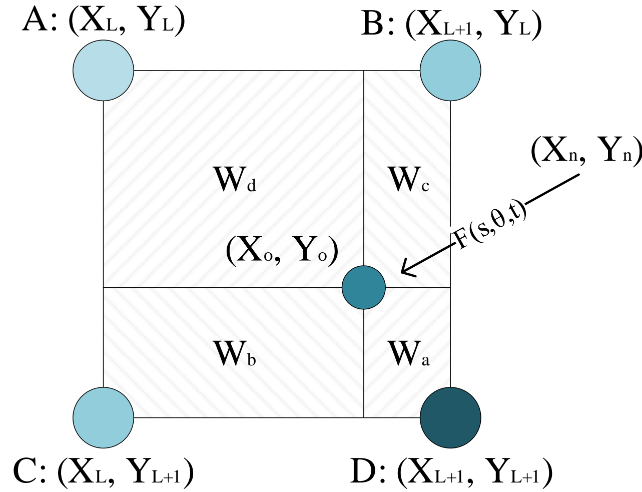


Figura 4.6: Interpolazione di Lagrange su quattro pixel

Si calcola quanto influisce ogni pixel a seconda dell'area del rettangolo costruito sul vertice (X_o, Y_o) e il vertice opposto. In figura 4.6 è raffigurato il concetto che si applica.
$W_a = (X_L + 1 - X_o) \times (Y_L + 1 - Y_o)$
$W_b = (X_o - X_L) \times (Y_L + 1 - Y_o)$
$W_c = (X_L + 1 - X_o) \times (Y_o - Y_L)$
$W_d = (X_o - X_L) \times (Y_o - Y_L)$
$(X_L,\ Y_L)$ sono gli interi più grandi minori o uguali a (X_o, Y_o).
Per calcolare il valore del nuovo pixel è sufficiente moltiplicare l'intensità di ciascuno dei quattro pixel per l'influenza rispettiva:

$$I[X_n, Y_n] = \frac{I[A] \times W_a + I[B] \times W_b + I[C] \times W_c + I[D] \times W_d}{W_a + W_b + W_c + W_d} \tag{4.3}$$

La somma delle aree dei quadrati è equivalente a 1: $W_a + W_b + W_c + W_d = 1$

$$I[X_n, Y_n] = I[A] \times W_a + I[B] \times W_b + I[C] \times W_c + I[D] \times W_d \tag{4.4}$$

Capitolo 5

Sviluppo del progetto

In questo capitolo si descrive lo sviluppo pratico del progetto con cui vengono effettuate le prove sperimentali. Si discute l'implementazione di una rete neurale artificiale interamente connessa (fully-connected neural network o dense neural network) e della trasformazione affine sulle immagini. Si descrivono poi le implementazioni parallele, pensate per sfruttare le diverse unità di calcolo in una macchina a memoria condivisa e l'architettura SIMD.

5.1 Rete neurale artificiale

La rete neurale artificiale implementata è di tipo interamente connessa (fully-connected neural network): questo significa che ogni neurone viene connesso a tutti i neuroni del corrispettivo livello precedente. Le caratteristiche di questa rete sono: un livello di input di dimensione $784 = 28 \times 28$, un livello di output composto da dieci neuroni, mentre il numero di livelli nascosti e le loro dimensioni sono variabili. I neuroni sono caratterizzati dalla funzione di attivazione di tipo Sigmoid ($1/(1 + e^{-x})$), che restringe il valore di uscita di un neurone sempre tra zero e uno.
L'uso della rete neurale è il seguente: a livello input si assegnano i valori dei pixel di un'immagine del MNIST dataset, mentre a livello di output si fa corrispondere ognuno dei dieci neuroni a una singola cifra. Si vuole fare in modo che il primo neurone di output si attivi in corrispondenza delle immagini raffiguranti il numero zero, il secondo quelle raffiguranti il numero uno, ecc... creando così una rete neurale in grado di classificare le cifre numeriche.
La definizione della funzione di costo è importante perché descrive come deve evolvere l'intera rete neurale. L'addestramento (training) infatti viene svolto

cercando di minimizzare la funzione di costo, che indica quanto è precisa la rete neurale. La funzione in questo caso è così definita:

$$C(d) = \frac{1}{len(d)} \sum_{n=1}^{len(d)} \sum_{i=1}^{10} [l(d_n)_i - p(d_n)_i]^2 \tag{5.1}$$

dove:

- d: dataset, $len(d)$: dimensione del dataset (numero di elementi).
- d_n: n-esimo elemento del dataset d. In questo caso d_n è un'immagine del MNIST dataset con associata l'etichetta che indica quale sia il valore numerico raffigurato.
- $C(d)$: costo del dataset d.
- $l(d_n)_i$ valore richiesto dall'i-esimo neurone sulla cifra d_n. per esempio il valore richiesto dal terzo neurone sulla cifra "5" è zero, mentre sulla cifra "2" è uno. Ogni neurone è specializzato nel riconoscere una cifra (come visto in precedenza). Questo valore quindi dipende dall'indice del neurone e dall'etichetta associata a d_n.
- $p(d_n)_i$: previsione effettiva dell'i-esimo neurone sulla cifra d_n. La previsione effettiva è il valore di output dell'i-esimo neurone del livello di output (valore sempre compreso tra zero e uno).

Il valore finale di questa funzione è sempre compreso tra dieci (precisione nulla) e zero (precisione massima). La funzione di costo può essere applicata sia al training-set che al test-set durante la fase di training per monitorare l'avanzamento del sistema. Le strutture implementate sono le seguenti:

```c
typedef struct layer
{
    float* outputs;
    float* biases;
    float* gradients;
    float** weights;
    struct layer* input;
    int size;
} Layer;
```

```c
typedef struct digit {
    float* data;
    int label;
} Digit;
typedef struct dnn {
    Layer* input_layer;
    int hidden_layers_count;
    Layer** hidden_layers;
    Layer* output_layer;
} DenseNeuralNetwork;
```

- Digit: rappresenta una singola immagine 28×28 con associata un'etichetta indicante il valore della cifra raffigurata. Viene utilizzata per caricare in memoria tutte le 70000 immagini del MNIST dataset.
- Layer: rappresenta un singolo livello nella rete neurale, contiene quindi un insieme di neuroni composti da: valore di output, bias, gradiente, e a sua volta un insieme di pesi. Un Layer può avere un altro livello di input, in questo caso essendo una rete interamente connessa, il numero di pesi di un neurone coincide con la dimensione del livello di input. Nella figura 5.1 è visualizzata la struttura.
- DenseNeuralNetwork: rappresenta un'intera rete neurale artificiale interamente connessa, è sempre composta da almeno un livello di input e uno di output, mentre i livelli nascosti sono di numero variabile.

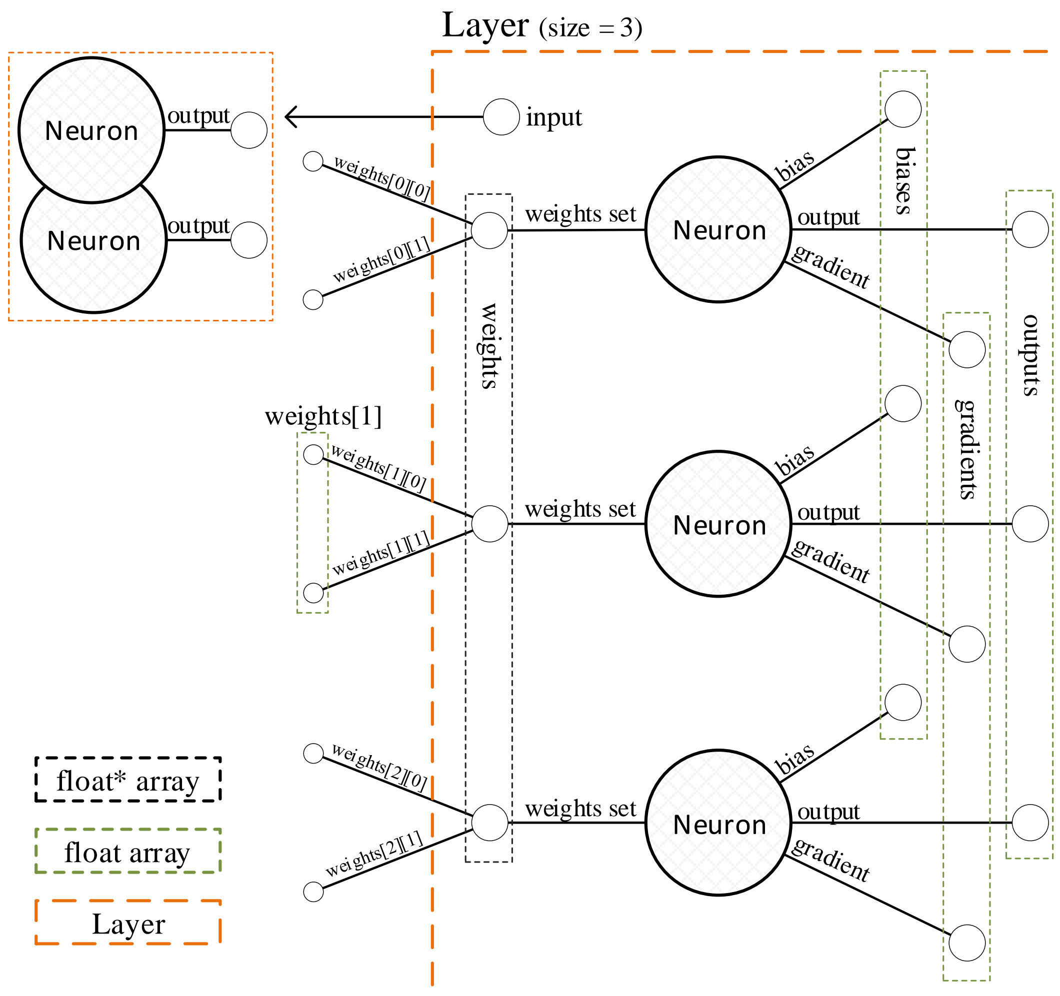


Figura 5.1: Rappresentazione grafica della struttura Layer.

La struttura Layer può essere divisa in sotto-strutture di tipo neurone, tuttavia, nel modo appena descritto si hanno i valori numerici predisposti in maniera contigua in memoria, caratteristica fondamentale nell'uso delle istruzioni SIMD.
Le fasi principali che deve svolge questa rete neurale artificiale sono tre: feedforward, backpropagation e gradient descent (in questo caso stochastic gradient descent (SGD)). Nella fase di feedforward la rete neurale effettua la classificazione di un dato input: una volta assegnato al primo livello i valori dell'immagine da analizzare, si procede per tutti i successivi Layer calcolando il valore di output di ogni neurone. Questa operazione è così composta: somma dei valori ricavati dalla moltiplicazione dei pesi del neurone per i valori di output del livello precedente; somma del bias al risultato; attivazione Sigmoid sul risultato ottenuto che riduce l'output del neurone tra zero e uno. La fase di feedforward di un neurone si può ridurre in questa espressione (come visto nel capitolo 2 *deep feedforward neural network*):

$$O = s(b + \sum_{i=1}^{k} (W_i \times I_i)) \tag{5.2}$$

dove:

- O è l'output del neurone.
- s è la funzione di attivazione Sigmoid.
- W è il vettore di pesi del neurone.
- I è il vettore di output del livello precedente, quindi l'input per questo neurone.
- W e I hanno la stessa lunghezza k, che in questo caso è la dimensione del livello a cui è collegato il neurone (livello precedente).
- b è il bias.

Nel codice mostrato di seguito è implementata la fase di feedforward: dopo aver assegnato al livello di input i valori dei pixel dell'immagine si procede calcolando O per tutti i neuroni, partendo dal primo livello nascosto fino ad arrivare a quello di output.

```
1 //feedforward di un layer.
2 void layer_forward(Layer* layer)
3 {
```

```
  for (int i = 0; i < layer->size; i++)
  {
    //input layer * pesi
    layer->outputs[i] = 0;
    for(int j = 0; j < layer->input->size; j++)
      layer->outputs[i] += layer->weights[i][j] *
                           layer->input->outputs[j];

    //sooma del bias
    layer->outputs[i] += layer->biases[i];

    //attivazione sigmoid
    layer->outputs[i] = 1.0f / (1.0f + exp(-layer->outputs[i]));
  }
}
//feedforward dell'intera rete neurale.
void feedforward(float* data)
{
  //assegnamento dell'immagine all'input layer
  memcpy(network->input_layer->outputs, data,
  sizeof(float) * network->input_layer->size);

  //computazione dei livelli
  for (int i = 0; i < network->hidden_layers_count; i++)
    layer_forward(network->hidden_layers[i]);
  layer_forward(network->output_layer);
}
```

Il primo passo è quello di scegliere una cifra tra le 60000 del training-set, eventualmente applicargli una trasformazione, per poi chiamare la funzione *feedforward*. Si ottengono così, nel vettore *outputs* dell'ultimo livello, dieci valori compresi tra zero e uno. Questi valori indicano le confidenze con cui la rete neurale prevede che l'immagine in input sia una certa cifra. Nel momento in cui la previsione di un neurone è errata, si può interagire con la rete neurale attraverso i valori dei gradienti del livello di output: se il neurone non ha riconosciuto la propria cifra (ci si aspettava il valore uno ma ha dato un valore più basso) si assegna al gradiente un valore positivo, se invece il neurone si attiva

(valore alto in uscita) quando in input non vi era la cifra corrispondente, si assegna un valore negativo.
Questi valori (gradiente positivo e gradiente negativo) sono ricavati dal modo in cui è stata definita la funzione di costo: il costo per un singolo esempio e per un singolo neurone viene calcolato con: $c = (l(d_n)_i - p(d_n)_i)^2$ dove n e i sono fissati. Derivando c in funzione di p si ottiene $c' = \frac{dc}{dp} = 2p(d_n)_i - 2l(d_n)_i$. Sapendo come pende la funzione c attraverso c' si può capire come variare p al fine di diminuire la funzione c, diminuendo di conseguenza anche la funzione C (funzione di costo). In generale se la funzione c' fornisce un valore (gradiente) negativo si deve aumentare la risposta del neurone, se fornisce un valore positivo, invece, si vuole diminuire l'output del neurone.
Per esempio se un neurone calcola 1 come output ($p(d_n)_i = 1$) quando in realtà il valore desiderato è 0 ($l(d_n)_i = 0$) si calcola $2p(d_n)_i - 2l(d_n)_i = 2\times1-2\times0 = 2$. Siccome questo gradiente va seguito in maniera inversa per ridurre la funzione di costo, si deve seguire la direzione -2 (quindi si vuole decrementare la risposta del neurone). Nell'applicazione sono stati usati dei valori prestabiliti (1, -1) tenendo conto solo della direzione da seguire. In questo esempio infatti si vuole abbassare la risposta del neurone ($p(d_n)_i = 1$ ma si voleva $p(d_n)_i = 0$) e si assegna il valore -1 al suo gradiente.
Terminata la fase di *feedforward* si esegue un controllo come il seguente: iterando tra i dieci neuroni di output si confronta il valore richiesto con quello ottenuto, aggiustando il gradiente del neurone nella maniera descritta. Il valore richiesto (come visto nella definizione della funzione di costo) può essere zero o uno in funzione dall'indice del neurone e dall'etichetta dell'immagine analizzata. Se l'etichetta è ad esempio "9", allora il valore richiesto è 1 solo per il decimo neurone di output, mentre per tutti gli altri è 0. Nel fare il confronto tra valore richiesto e quello ottenuto si utilizzano dei margini di correttezza: se ci si aspetta il valore 1 da un certo neurone, è sufficiente un valore vicino, come può essere 0.9; lo stesso principio vale per il valore 0. Questi margini permettono di saltare le due fasi successive (backpropagation e SGD) se non viene assegnato nessun gradiente, velocizzando l'intera fase di training.
La codifica di questo controllo è la seguente:

```
1 //controllo sulla risposta della rete neurale.
2 for (int i = 0; i < 10; i++)
3 {
4   int must_be = digit.label == i ? 1 : 0;
```

```
    //si vuole alzare la risposta del neurone
    if (must_be == 1 && network->output_layer->outputs[i] <= 0.9)
        network->output_layer->gradients[i] = 1;
    //si vuole abbassare la risposta del neurone
    if (must_be == 0 && network->output_layer->outputs[i] >= 0.1)
        network->output_layer->gradients[i] = -1;
}
```

Successivamente si esegue l'operazione di *backpropagation*. Questo procedimento, in base hai gradienti appena assegnati, calcola (propaga) i restanti gradienti in tutta la rete. Questi valori indicano le variazioni che ogni peso e bias devono seguire per ridurre la funzione di costo. Si adotta la tecnica *stochastic gradient descent* (SGD) per applicare questi gradienti alle variabili: durante ogni fase di *backpropagation* si aggiornano i pesi e i bias in funzione del gradiente calcolato e dal un parametro *learning rate*. Queste due operazioni fanno in modo che una previsione futura (sull'immagine di input appena analizzata) sia più precisa. In generale la *backpropagation* è la fase in cui il sistema capisce dove ha sbagliato e la fase di SGD è il momento in cui il sistema impara.
I gradienti propagati durante la fase di *backpropagation* vengono calcolati tramite una tecnica analitica (analytic gradient [8]). Il principio di base è quello della *chain-rule*: una volta assegnato un gradiente a un dato neurone si può calcolare come ogni componente del neurone (pesi, bias, e input) deve variare. Per esempio se a un neurone viene assegnato un gradiente positivo, si può calcolare come il bias, i pesi e gli input dovrebbero variare al fine di far elaborare al neurone un risultato più positivo. Le componenti del neurone comprendono anche i neuroni di input associati ad esso, quindi a ognuno di essi verrà assegnato a sua volta un gradiente che sarà propagato con la stessa tecnica.
Nella fase di *feedforward* le operazioni che effettua un neurone sono quattro: moltiplicazione dei pesi per i valori di input, sommatoria dei risultati, somma del bias, attivazione Sigmoid. Per calcolare il gradiente che si propaga a ogni componente di un neurone (bias, pesi, input) si procede in questo modo:

- Derivazione della funzione di attivazione Sigmoid $s(x) = 1/(1 + e^{-x})$: $s'(x) = s(x)(1 - s(x))$. Il gradiente da propagare quindi si trasforma in $G_s = s'(x) \times G$ dove G è il gradiente del neurone assegnato dal ciclo di controllo o, se in un livello intermedio, dalla propagazione di un Layer;

$s(x)$ è l'output del neurone calcolato nella fase di *feedforward*, e $x = b + \sum_{i=1}^{k}(W_i \times I_i)$.

- Propagazione al bias: in questo caso la derivata da calcolare è $\frac{d(b+\sum_{i=1}^{k}(W_i \times I_i))}{db} = 1$, il risultato è 1 perciò il gradiente assegnato al bias resta invariato ed è G_s.
- Propagazione ai pesi: in questo caso la derivata della sommatoria $\sum_{i=1}^{k}(W_i \times I_i)$ in funzione di W è equivalente a I. I gradienti assegnati ai pesi sono quindi $I \times G_s$. W è il vettore di pesi del neurone, I è il vettore dei valori di input (in questo caso l'output del livello precedente); i due vettori hanno lo stesso numero di elementi.
- Propagazione del gradiente al corrispettivo Layer di input: la derivata della sommatoria $\sum_{i=1}^{k}(W_i \times I_i)$ in funzione di I equivale a W; i gradienti che si propagano al livello precedente quindi sono $W \times G_s$.

Successivamente si applicano i valori dei gradienti tramite SGD:

- Il bias viene aggiornato secondo il gradiente G_s e α quindi $b = b + G_s \times \alpha$ dove b è il bias e α è il parametro di *learning rate*.
- I pesi vengono aggiornati secondo il gradiente calcolato e α, quindi $W = W + G_s \times I \times \alpha$.

La fase di *backpropagation* inizia dai neuroni del livello di output, per poi proseguire per tutti i precedenti livelli. Ovviamente il livello di input è un livello "speciale" e i suoi neuroni sono composti solo dall'output assegnato nella fase di *feedforward*. Non si propagano i gradienti al livello di input anche perché non si vuole modificare l'immagine analizzata, ma bensì i pesi e i bias della rete neurale. Il codice di seguito mostra il metodo che esegue la fase di backpropagation e l'applicazione dei gradienti tramite SGD per un dato livello.

```
1 void backpropagation_sgd_layer(Layer* layer)
2 {
3   //backpropagation: calcolo dei gradienti
4   for (int i = 0; i < 10; i++)
5   {
6     //derivata della funzione di attivazione sigmoid
7     for (int i = 0; i < layer->size; i++)
```

```
      layer->gradients[i] = layer->gradients[i] *
        layer->outputs[i] * (1 - layer->outputs);

    //propagazione dei gradienti
    if(layer->input->gradients != NULL)
      for (int i = 0; i < layer->size; i++)
        for (int j = 0; j < layer->input->size; j++)
          layer->input->gradients[j] += layer->weights[i][j] *
          layer->gradients[i];


    //SGD: aggiornamento dei pesi e bias
    for (int i = 0; i < layer->size; i++)
    {
      for (int j = 0; j < layer->input->size; j++)
        layer->weights[i][j] =
        layer->weights[i][j] * regularization_force +
        learning_rate * layer->input->outputs[j] *
        layer->gradients[i];

      layer->biases[i] += layer->gradients[i] * learning_rate;
    }
}
```

Per training della rete neurale s'intende l'esecuzione di questi tre procedimenti (*feedforward*, *backpropagation*, *SGD*) per tutto l'insieme d'immagini del training-set, per più volte. Il parametro *learning_rate* indica quanto velocemente l'algoritmo impara: con un valore troppo elevato la rete neurale non riuscirebbe a essere stabile, con un valore troppo piccolo la fase di training richiederebbe molto tempo. Il parametro *regularization_force*, invece, indica con quanta forza i pesi vengono portati a zero. Infatti questa tecnica (L1 regularization) è utile per prevenire l'overfitting della rete, e consiste nel cercare di minimizzare la somma dei pesi [5]. Al termine della fase di training (ed eventualmente anche a intervalli regolari), utilizzando il test-set, si analizza la precisione del sistema in termini di funzione di costo e numero d'immagini riconosciute con successo. La fase di training è suddivisa in sotto-fasi; una sotto-fase prende il nome di "epoca", solitamente dopo ognuna di esse si

controlla l'avanzamento del sistema tramite il test-set. Nella definizione del training, quindi, vi è anche la dimensione di un'epoca, e il numero totale di esse. Solitamente un'epoca è di dimensioni pari al training-set, in modo che per ognuna di esse la rete neurale abbia analizzato l'intero training-set.

5.1.1 Tecniche di parallelismo

Per utilizzare tutte le risorse di calcolo di una macchina a memoria condivisa, si cerca di parallelizzare il codice al fine di eseguire la fase di training della rete neurale più velocemente. Esistono varie tecniche di parallelizzazione, in questo caso ne vengono usate due. La prima è implementata con l'uso delle SIMD intrinsics. Infatti come visto in precedenza, nelle varie fasi di training della rete neurale, vi sono molti cicli che lavorano su vettori di float contigui in memoria, questo permette un uso piuttosto semplice delle SIMD. Prendendo l'esempio di feedforward di un Layer: l'operazione base da svolgere è quella di moltiplicare tutti i pesi di un neurone per l'input del livello precedente, sommando i risultati; questa operazione si traduce in una mera moltiplicazione di vettori, seguita da una riduzione somma. Con l'uso delle SIMD intrinsics può essere cosi implementata:

```
//return SUM[b*c]
float multiply_and_sum(float* b, float* c, int n)
{
    float sum = 0;
    vecf sum_vect = simd_bcast_f32(0);
    vecf b_vect;
    vecf c_vect;
    int i, l = n + 1 - VLEN, j;
    for(i = 0; i < l; i += VLEN)
    {
        b_vect = simd_load_f32(&b[i]);
        c_vect = simd_load_f32(&c[i]);
        // a = a + b * c;
        sum_vect = simd_mla_f32(sum_vect, b_vect, c_vect);
    }
    for(; i < n; i ++)
        sum += b[i] * c[i];
    for(j = 0; j < VLEN; j++)
```

```
        sum += a_vect[j];
    return sum;
}
```

Questo parallelismo si può considerare di livello fine, in quanto si suddivide il lavoro in dimensioni pari alla lunghezza dei vettori SIMD utilizzati. La seconda tecnica adottata [9] si può descrivere con i seguenti passaggi: all'avvio del programma si genera una rete neurale artificiale con valori casuali, successivamente si crea un clone di questa per ogni thread disponibile. Ogni processo esegue una sotto-fase di training con un proprio clone e un proprio sottoinsieme disgiunto del training-set. Quando tutti i thread hanno terminato la propria sotto-fase di training si uniscono i cloni per ricomporre una sola rete neurale, ciò avviene attraverso una media dei pesi e dei bias, ottenendo così un modello che può essere clonato nuovamente per ripetere gli stessi passaggi. Nella figura 5.2 è visualizzato lo schema di questa tecnica.

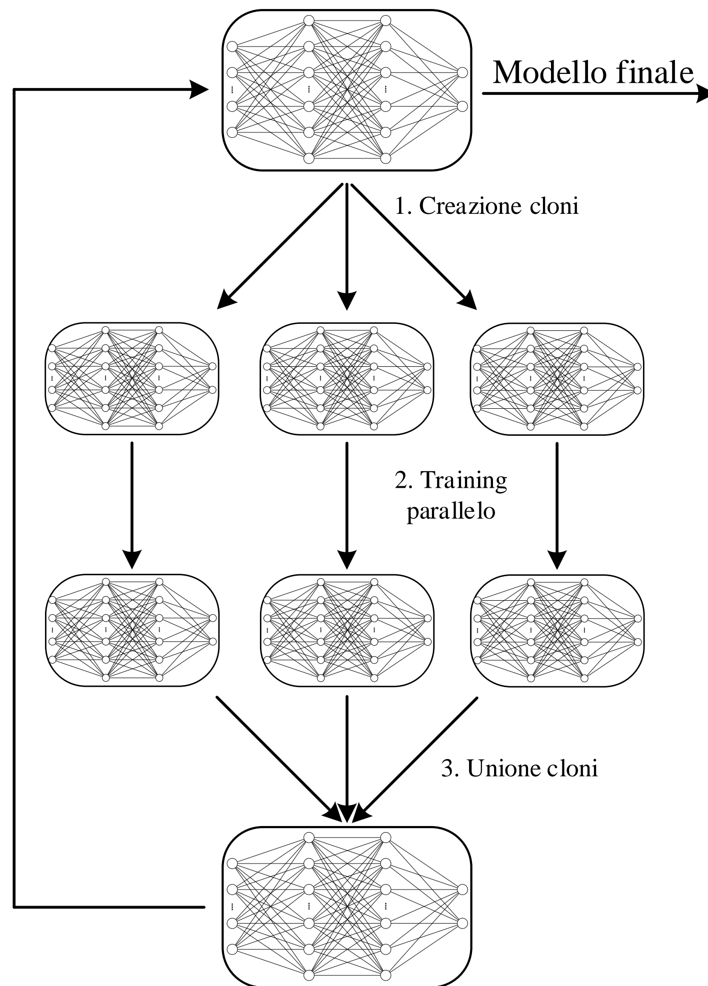


Figura 5.2: Rappresentazione grafica del metodo di parallelizzazione adottato.

È buona pratica eseguire tutto il processo più volte e di non fare durare una singola sessione di training parallelo troppo a lungo, in caso contrario nasce il rischio di far evolvere i cloni molto diversamente, generando nella fase di unione un modello peggiore in termini di precisione rispetto ai singoli. Questa tecnica permette di utilizzare molte risorse di calcolo anche in reti neurali di dimensioni non troppo elevate, sebbene introduca un certo overhead dovuto alla creazione di cloni e alla loro unione.
Questa tecnica di parallelismo presenta un forte problema di sbilanciamento del carico, in quanto uno o più cloni potrebbero finire la sessione di training prima degli altri, creando dei momenti vuoti prima dell'unione. Per evitare questo problema, si possono sincronizzare le diverse fasi di training in questo

modo: non appena termina il training di un qualunque clone, anziché aspettare la fine degli altri, si lancia un segnale che fa terminare tutte le fasi di training. Nella sessione successiva si aumenta la dimensione dell'epoca in modo da controbilanciare la prematura fine di questa. Così facendo alcuni cloni potrebbero aver analizzano meno cifre di altri, ciò tuttavia non influisce sull'avanzamento del sistema.

5.2 Trasformazione affine

L'implementazione della trasformazione affine sulle immagini viene impiegata durante la fase di training della rete neurale artificiale. Ogni immagine del training-set, prima di essere data in input alla rete neurale, viene trasformata attraverso questa tecnica con parametri casuali (entro certi intervalli). Questo permette di ampliare il dataset come visto nel capitolo 4.3. Il vantaggio è quello di evitare l'effetto *overfitting* della rete neurale, permettendo di raggiungere precisioni più elevate dal sistema di machine learning. L'implementazione della trasformazione affine viene effettuata come trattato nel quarto capitolo, introducendo altri due paramenti per completezza dell'algoritmo, tra cui: scelta del centro di rotazione e della risoluzione finale. Viene quindi implementata una funzione di questo tipo:

```
void affine_transform(
float* image, //input image
int width, //width of input image
int height, //height of input image
float background, //default background value
float degree, // rotation degree
float Sx, float Sy, //scale factor x and y
float Tx, float Ty, //traslation value x and y
float RCx, float RCy, //rotation center [0,1]
float* out_image, //output image
int out_width, //width of output image
int out_height) //height of outputimage
{
    //...
}
```

Per tenere conto di questi ulteriori due fattori si utilizza sempre la tecnica descritta nel capitolo 4.4 applicandola più volte:

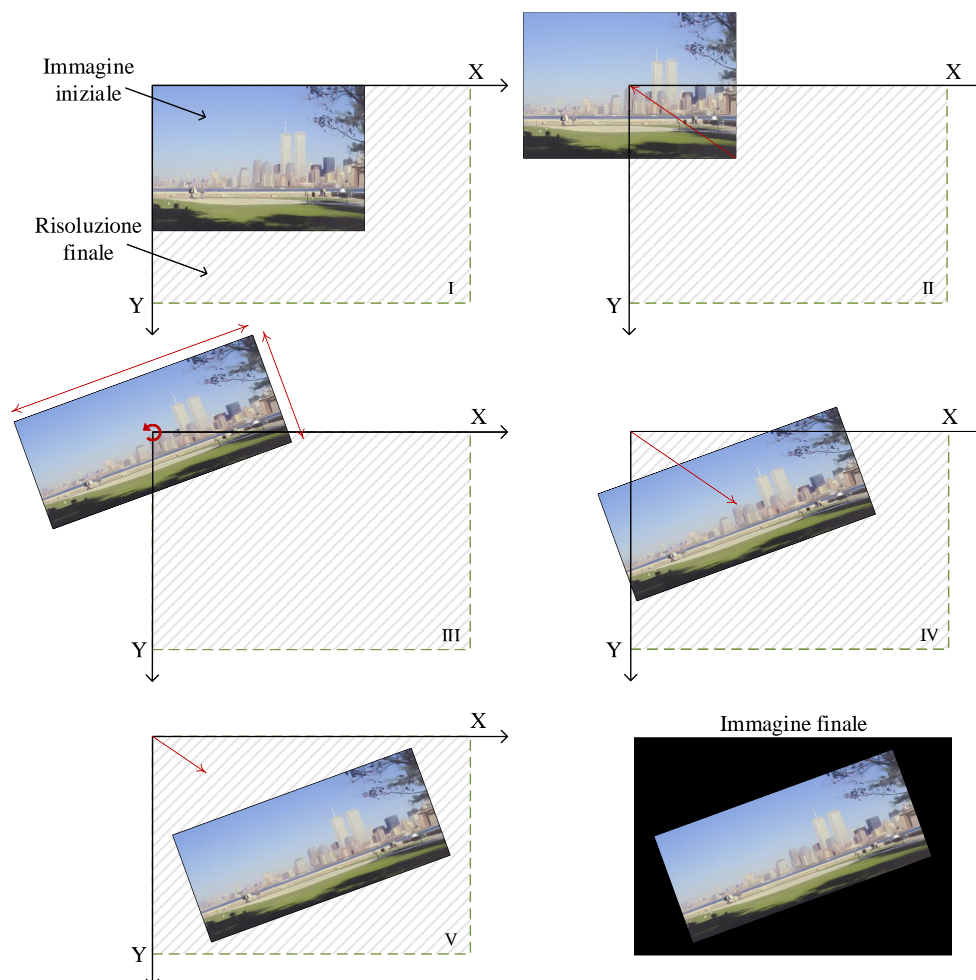


Figura 5.3: Esempio di trasformazione affine applicata a un'immagine.

Per esempio, si vuole trasformare un'immagine ridimensionandola e ruotandola rispetto al centro, per poi collocarla in una nuova immagine di risoluzione maggiore. Nella figura 5.3 sono mostrate le varie fasi per ottenere l'immagine trasformata:

- I: è mostrata l'immagine iniziale e la risoluzione finale.
- II: si trasla la figura per fare coincidere l'origine con il centro di rotazione desiderato.
- III: si applicano le trasformazioni di ridimensionamento e rotazione in quest'ordine.

- IV: si esegue l'operazione complementare al II passaggio, ovvero, si trasla l'immagine per far tornare il centro di rotazione nella posizione iniziale.
- V: si centra l'immagine trasformata nella risoluzione dell'immagine finale eseguendo uno spostamento.

In questo caso non si è traslata l'immagine ulteriormente ($t_x = t_y = 0$), altrimenti sarebbe stato necessario un altro passaggio per eseguire un'operazione di spostamento.
La formula della trasformazione diretta quindi diventa:

$$\begin{bmatrix} X_n \\ Y_n \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \left(\begin{bmatrix} X_o \\ Y_o \end{bmatrix} - \begin{bmatrix} C_x \\ C_y \end{bmatrix} \right) + \begin{bmatrix} C_x \\ C_y \end{bmatrix} + \begin{bmatrix} R_x \\ R_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{5.3}$$

dove $(C_x, C_y) = (RC_x \times I_w, RC_y \times I_h)$ indica il vettore di spostamento nel centro di rotazione, $(R_x, R_y) = ((F_w - I_w)/2, (F_h - I_h)/2)$ rappresenta il vettore per centrare l'immagine nel centro della risoluzione finale. $(F_w, F_h), (I_w, I_h)$ sono rispettivamente la risoluzione finale e la risoluzione dell'immagine iniziale.
Infine la formula per la trasformazione inversa diventa:

$$\begin{bmatrix} X_o \\ Y_o \end{bmatrix} = \begin{bmatrix} \frac{1}{s_x} & 0 \\ 0 & \frac{1}{s_y} \end{bmatrix} \begin{bmatrix} cos(-\theta) & sin(-\theta) \\ -sin(-\theta) & cos(-\theta) \end{bmatrix} \left(\begin{bmatrix} X_n \\ Y_n \end{bmatrix} - \begin{bmatrix} C_x \\ C_y \end{bmatrix} - \begin{bmatrix} R_x \\ R_y \end{bmatrix} - \begin{bmatrix} t_x \\ t_y \end{bmatrix} \right) + \begin{bmatrix} C_x \\ C_y \end{bmatrix} \tag{5.4}$$

Il corpo della funzione è:

```
double theta = -M_PI * degree / 180.0f;
float Xa = (1.0f / Sx) * cos(-alpha);
float Xb = (1.0f / Sx) * sin(-alpha);
float Ya = -(1.0f / Sy) * sin(-alpha);
float Yb = (1.0f / Sy) * cos(-alpha);
float Cx = RCx * width;
float Cy = RCy * height;
float Rx = (out_width - width) / 2;
float Ry = (out_height - height) / 2;
float FTx = Cx + Rx + Tx;
float FTy = Cy + Ry + Ty;
for(int y = 0; y < out_height; y++)
{
```

```
14      for(int x = 0; x < out_width; x++)
15      {
16          float a = x - FTx;
17          float b = y - FTy;
18          float Xo = Xa * a + Xb * b + Cx;
19          float Yo = Ya * a + Yb * b + Cy;
20
21          out_image[y * out_width + x] = linear_interpolation(Xo,
22          Yo, image, width, height, background);
23      }
24  }
```

5.2.1 Algoritmo parallelo

Siccome ogni pixel può venire calcolato indipendentemente, questo tipo di algoritmo è facilmente parallelizzabile in un sistema a memoria condivisa. La prima tecnica usata è quella di parallelizzare il ciclo for più esterno, il lavoro quindi viene suddiviso per righe. Il secondo metodo prevede d'implementare l'algoritmo tramite l'utilizzo delle SIMD intrinsics; in pratica una riga viene suddivisa in blocchi di 4 pixel, che verranno computati contemporaneamente tramite operazioni SIMD.
Esempio di parallelizzazione con SIMD intrinsics e OpenMP:

```
1   #pragma omp parallel for schedule(static, 4)
2   for(int y = 0; y < out_height; y++)
3   {
4     for(int x = 0; x < out_width; x++)
5     {
6       vecf y_v = simd_bcast_f32((float)y);
7       vecf x_v = simd_bcast_f32((float)x);
8       x_v = simd_add_f32(x_v, index); //index = [0, 1, 2, ...]
9
10      vecf a = simd_sub_f32(x_v, FTx_v);
11      vecf b = simd_sub_f32(y_v, FTy_v);
12      vecf Xo = simd_add_f32(
13          simd_mla_f32(Cx_v, Xb_v, b),
14          simd_mul_f32(Xa_v, a));
```

```
15      vecf Yo = simd_add_f32(
16          simd_mla_f32(Cy_v, Yb_v, b),
17          simd_mul_f32(Ya_v, a));
18
19      vecf value = linear_interpolation(Xo,
20          Yo, image, width, height, background);
21
22      simd_store_f32(&(out_image[y * out_width + x]), value);
23    }
24  }
```

I nomi delle funzioni SIMD appartengono a una libreria per il supporto multi-piattaforma ARM NEON, AVX2, SSE2. Ad esempio se si compila in un ambiente ARM NEON la funzione *simd_add_f32* viene trasformata in *vaddq_f32* e le strutture *vecf* in *float32x4_t*. La libreria in questione è stata fornita dal professore di High Performance Computing dell'Università di Bologna: Moreno Marzolla. In seguito sono state introdotte alcune modifiche come il supporto per ARM NEON. Un esempio di funzione all'interno di questa libreria è la seguente:

```
1  static INLINE vecf simd_mul_f32(vecf a, vecf b) {
2    #if defined __AVX2__
3      return _mm256_mul_ps(a, b);
4    #elif defined __SSE2__
5      return _mm_mul_ps(a, b);
6    #elif defined __ARM_NEON__
7      return (vecf)vmulq_f32((float32x4_t)a, (float32x4_t)b);
8    #else //nessun supporto SIMD intrinsics
9      vecf res;
10     for(int i = 0; i < VLEN; i++)
11       res[i] = a[i] * b[i];
12     return res;
13   #endif
14 }
```

Capitolo 6

Valutazione dei risultati

In questo capitolo si descrivono le prove sperimentali effettuate, analizzando nel dettaglio i risultati ottenuti. Si discute di alcuni test effettuati sull'algoritmo della trasformazione affine; si procede con l'analisi delle prestazioni ottenute sull'algoritmo di machine learning. Si ricavando *speed-up* ed efficienze di varie configurazioni, sia sulla scheda Raspberry Pi che sui processori Intel Xeon. Vengono evidenziate le differenze tra versioni SIMD e versioni sequenziali, e si paragonano i risultati della scheda Raspberry Pi a quelli dei processori Xeon.

6.1 Trasformazione affine

La prima prova effettuata prevede di applicare una serie di trasformazioni affini casuali a una certa immagine, ripetendo lo stesso test con un numero di processori sempre crescente, ricavando così *speed-up* e *strong-scaling-efficiency*. La seconda prova invece prevede di aumentare il carico di lavoro in proporzione al numero di processori usati, calcolando la *weak-scaling-efficiency*. Come visto nel primo capitolo, le formule utilizzate per l'analisi delle prestazioni sono:

$$S(p) = \frac{T_{seriale}}{T_{parallelo}(p)} \tag{6.1}$$

$$E(p) = \frac{S(p)}{p} \tag{6.2}$$

$$W(p) = \frac{T_1}{T_p} \tag{6.3}$$

dove $S(p)$ indica lo *speed-up*, $E(p)$ è la *strong-scaling-efficiency* e $W(p)$ la *weak-scaling-efficiency*. p è il numero di processori, $T_{seriale}$ è il tempo di esecuzione

seriale, $T_{parallelo}(p)$ è il tempo di esecuzione con p processori, T_1 è il tempo impiegato a svolgere un'unità di lavoro con un processore, T_p è il tempo impiegato a svolgere p unità di lavoro con p processori.
Il test della *strong-scaling-efficiency* prevede di trasformare un'immagine 2000×2000 con parametri dell'algoritmo casuali. Il test della *weak-scaling-efficiency*, invece, prevede di elaborare un carico di lavoro costante per processore: 2000×2000 pixel per singola unità di calcolo. In figura 6.1 si mostrano le due efficienze e relativo *speed-up* che si sono ottenuti sul Raspberry Pi, mentre in figura 6.2 quelle relative ai processori Xeon. I test vengono eseguiti più volte, facendo una media dei risultati per ottenere una maggiore affidabilità. Nel Raspberry Pi si arriva a un grado di parallelismo di quattro, mentre sugli Xeon di dodici, in quanto si dispone di due processori Xeon dotati di sei core ciascuno.

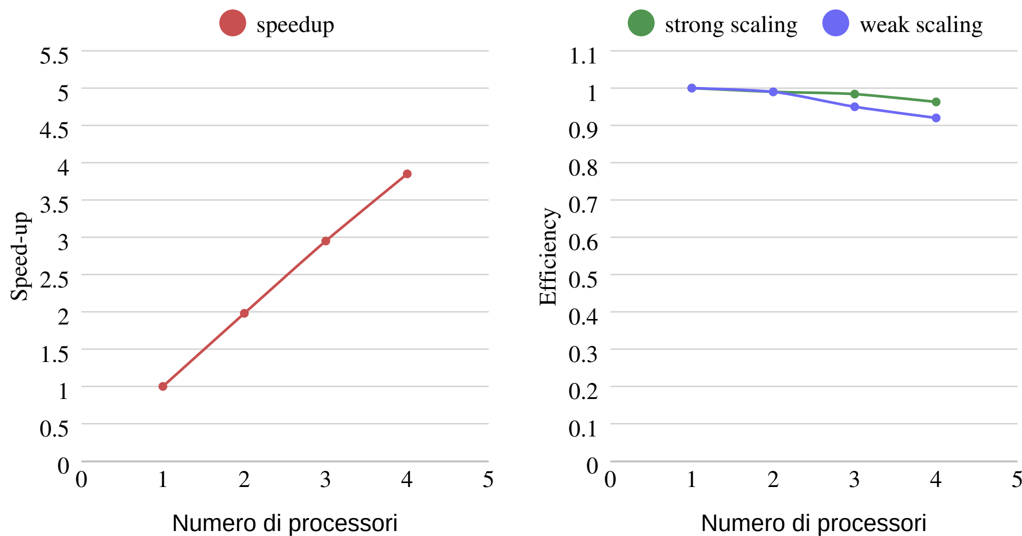


Figura 6.1: Scalabilità ed efficienza della trasformazione affine su Raspberry Pi.

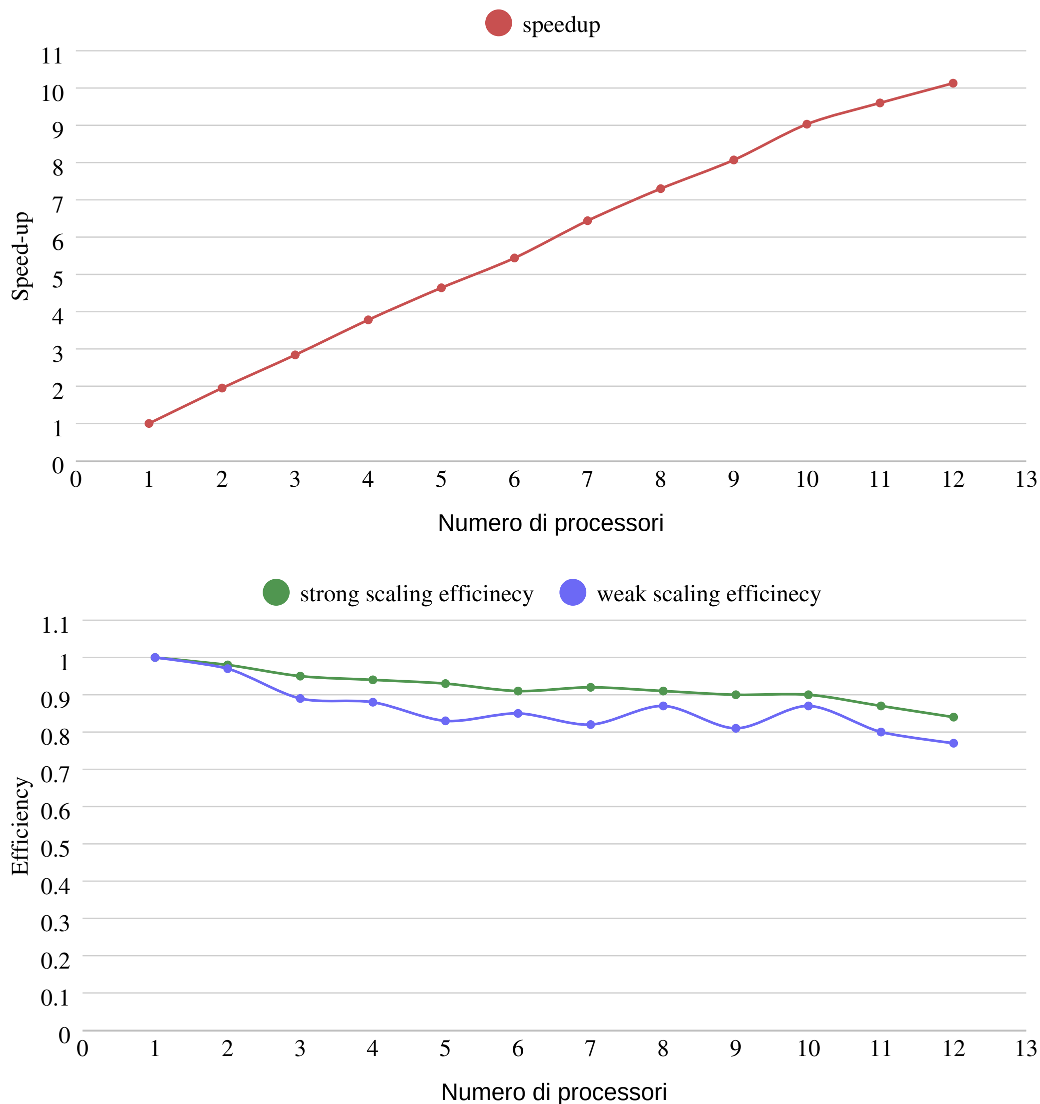


Figura 6.2: Scalabilità ed efficienza della trasformazione affine sui processori Xeon.

Per sottolineare la differenza di prestazioni tra i due dispositivi (Raspberry Pi e Xeon) in figura 6.3 sono visualizzati i tempi di esecuzione del test relativo alla *strong-scaling-efficiency*, con *speed-up* calcolato tra Xeon e Raspberry Pi. L'asse y dei millisecondi rappresenta quanto tempo è stato impiegato per elaborare una singola immagine. Dal grafico si nota che a parità di core utilizzati il processore Xeon è circa quattro volte più veloce rispetto al Raspberry Pi (a parità di core utilizzati).

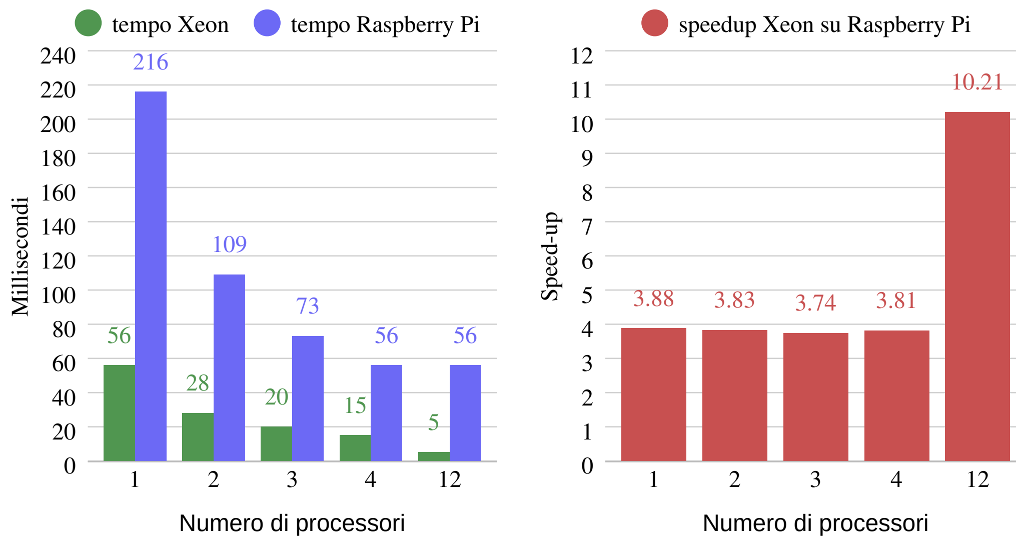


Figura 6.3: Differenza di prestazioni tra Raspberry Pi e processori Xeon nella trasformazione affine.

Nella figura 6.4 si mostra la differenza di prestazioni tra versione SIMD e versione sequenziale. L'uso delle SIMD non fornisce uno *speed-up* apprezzabile perché i pixel necessari nella fase d'interpolazione non sono contigui in memoria. Questo fattore rende necessario il caricamento dei pixel in modo sequenziale, riducendo le prestazioni guadagnate dall'uso delle SIMD. Questo test è stato svolto su molteplici trasformazioni affini casuali come nei test precedenti, con immagini 4000×4000, e con un solo processore. L'asse dei millisecondi rappresenta quanto tempo è stato impiegato per elaborare una singola immagine. Dal grafico si nota che si riescono a ricavare lievi *speed-up* dall'uso delle architetture SIMD, pur avendo regioni sequenziali non indifferenti. Il processore Xeon, in entrambi i casi, è circa quattro volte più performante del Raspberry Pi.

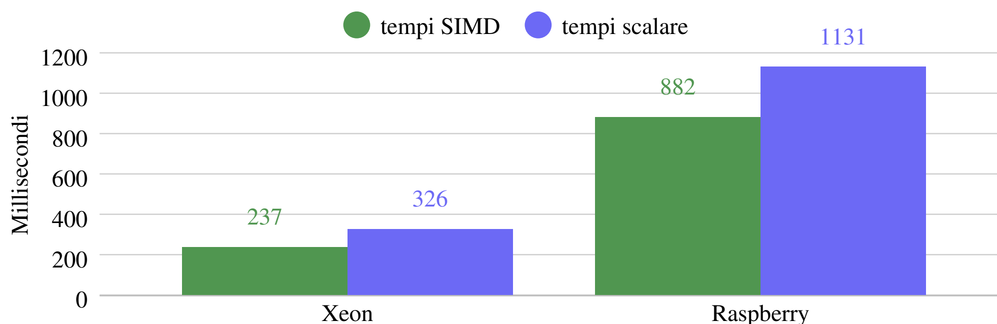


Figura 6.4: Differenza di prestazioni tra versione sequenziale e versione SIMD nella trasformazione affine.

6.2 Rete neurale interamente connessa

In questa sezione si mostrano i risultati raccolti nel training di diverse reti neurali. La prima prova è stata svolta utilizzando una rete neurale 784 : 240 : 120 : 10. Questa notazione (784 : 240 : 120 : 10) indica che la rete neurale è composta da 4 livelli: il livello di input di dimensione 784, il primo livello nascosto di dimensione 240, il secondo livello nascosto di dimensione 120, infine il livello di output di dimensione 10. In figura 6.5 sono mostrati i risultati della sessione di training sul Raspberry Pi. Lo *speed-up* e la *strong-scaling-efficiency* sono calcolati mantenendo costate il carico di lavoro e aumentando il grado di parallelismo; la *weak-scaling-efficiency*, invece, è calcolata incrementando la dimensione dell'epoca in funzione del numero di processori usati. Risultati così scarsi sono giustificabili dalla dimensione della rete neurale: 218160 pesi e in generale da circa 220000 variabili float, che occupano una dimensione di 860 KB. Il processore Cortex-A53, montato sul Raspberry Pi, è dotato di 512 KB di memoria cache di secondo livello. All'aumentare del grado di parallelismo, aumenta anche il numero di reti neurali (come visto nel capitolo sull'implementazione), provocando una percentuale elevata di *cache-miss*, con conseguente perdita di prestazioni.

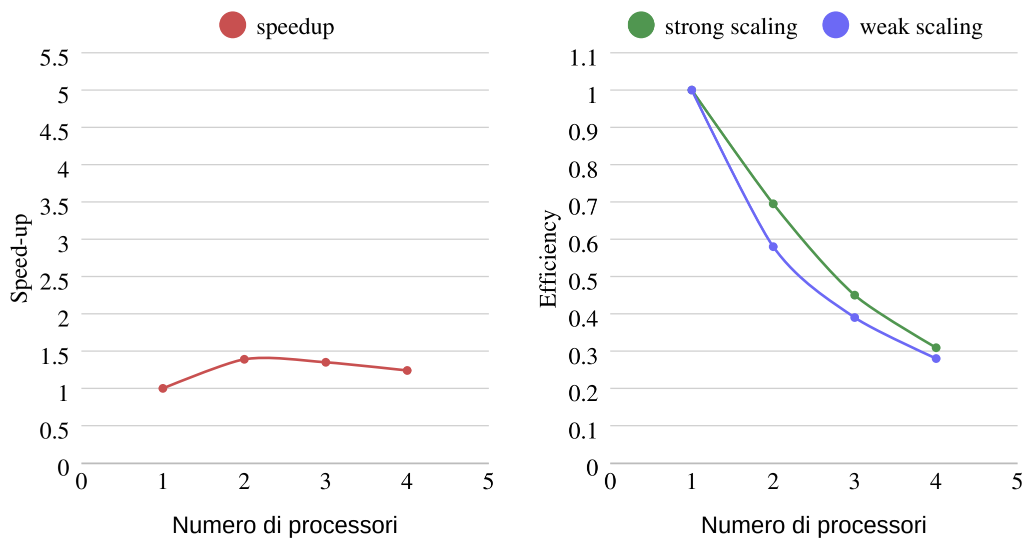


Figura 6.5: Scalabilità ed efficienza del training della rete neurale 784 : 240 : 120 : 10 su Raspberry Pi.

Al fine di verificare che il problema sia la saturazione della memoria cache, si è eseguito un test analogo con una rete neurale 728 : 30 : 10 che occupa circa 22 KB di memoria. In figura 6.6 sono mostrati i risultati ottenuti: la dimensione della rete neurale sembra influire pesantemente sulle prestazioni in questo dispositivo.

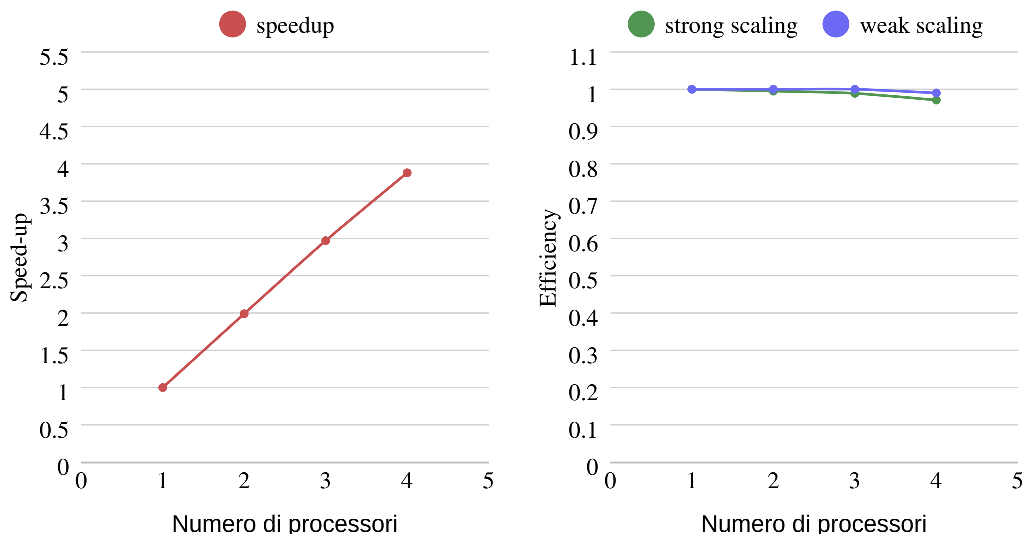


Figura 6.6: Scalabilità ed efficienza del training della rete neurale 784 : 30 : 10 su Raspberry Pi.

La stessa prova con la rete 784 : 240 : 120 : 10 è stata svolta sui due processori Xeon, arrivando a un grado di parallelismo di dodici. In questo

caso nella figura 6.7 le prestazioni non sembrano gravare della dimensione della rete neurale, e i risultati ottenuti sono soddisfacenti. Infatti ognuno di questi due processori è dotato di 15 MB di memoria cache, coprendo ampiamente le dimensioni delle reti neurali.

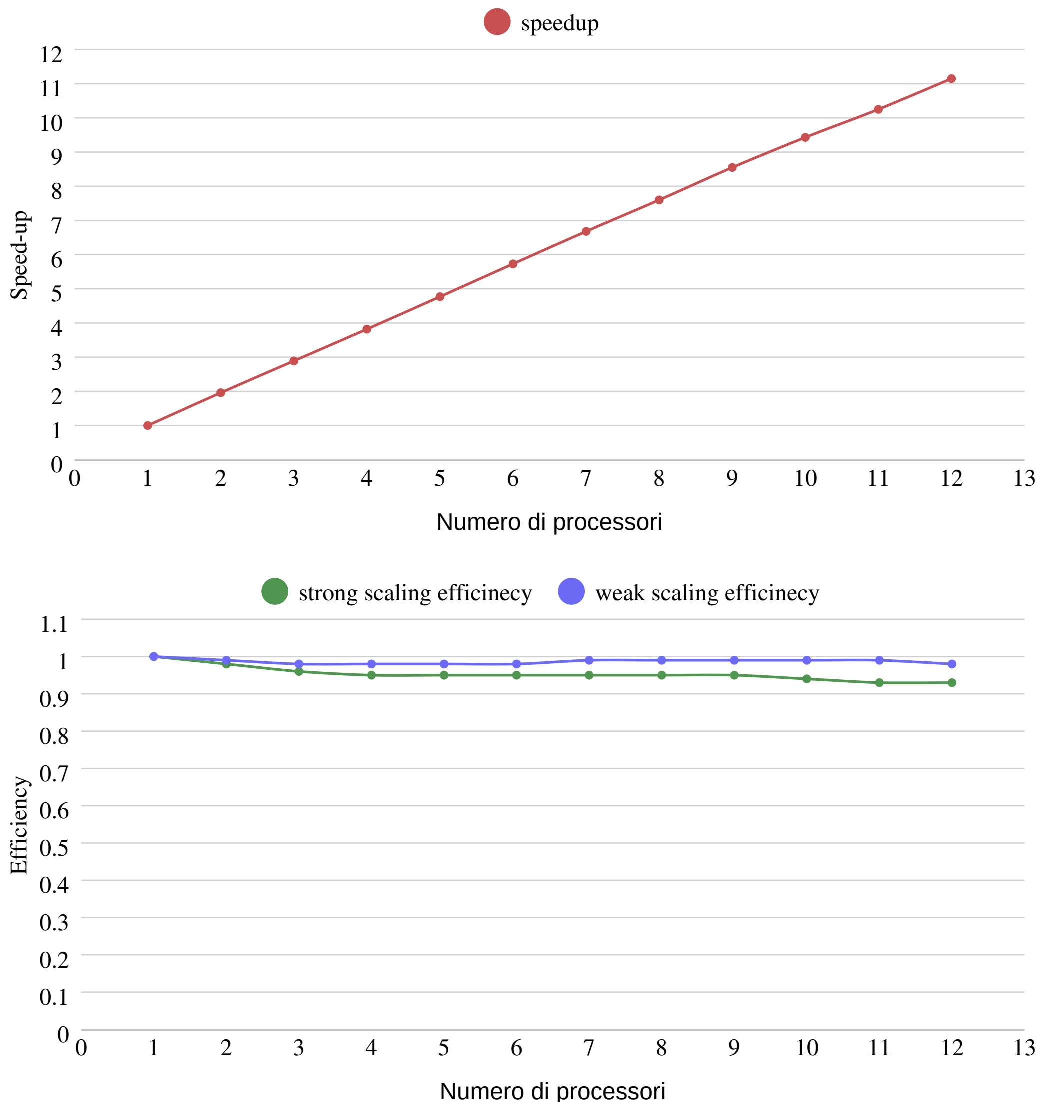


Figura 6.7: Scalabilità ed efficienza del training della rete neurale 784 : 240 : 120 : 10 su processori Xeon.

Si è voluta stressare la memoria cache dei processori Xeon con una rete neurale 784 : 2048 : 1024 : 512 : 10. Lo scopo è di verificare il comportamento della macchina nel caso in cui il working-set sia più grande della memoria cache a disposizione. Una singola rete neurale così composta occupa circa 16 MB di sole variabili. Nella figura 6.8 si verifica che le prestazioni degradano

molto più in fretta rispetto al test precedente; si riesce comunque a ottenere un incremento di prestazioni dall'uso di più unità di calcolo.

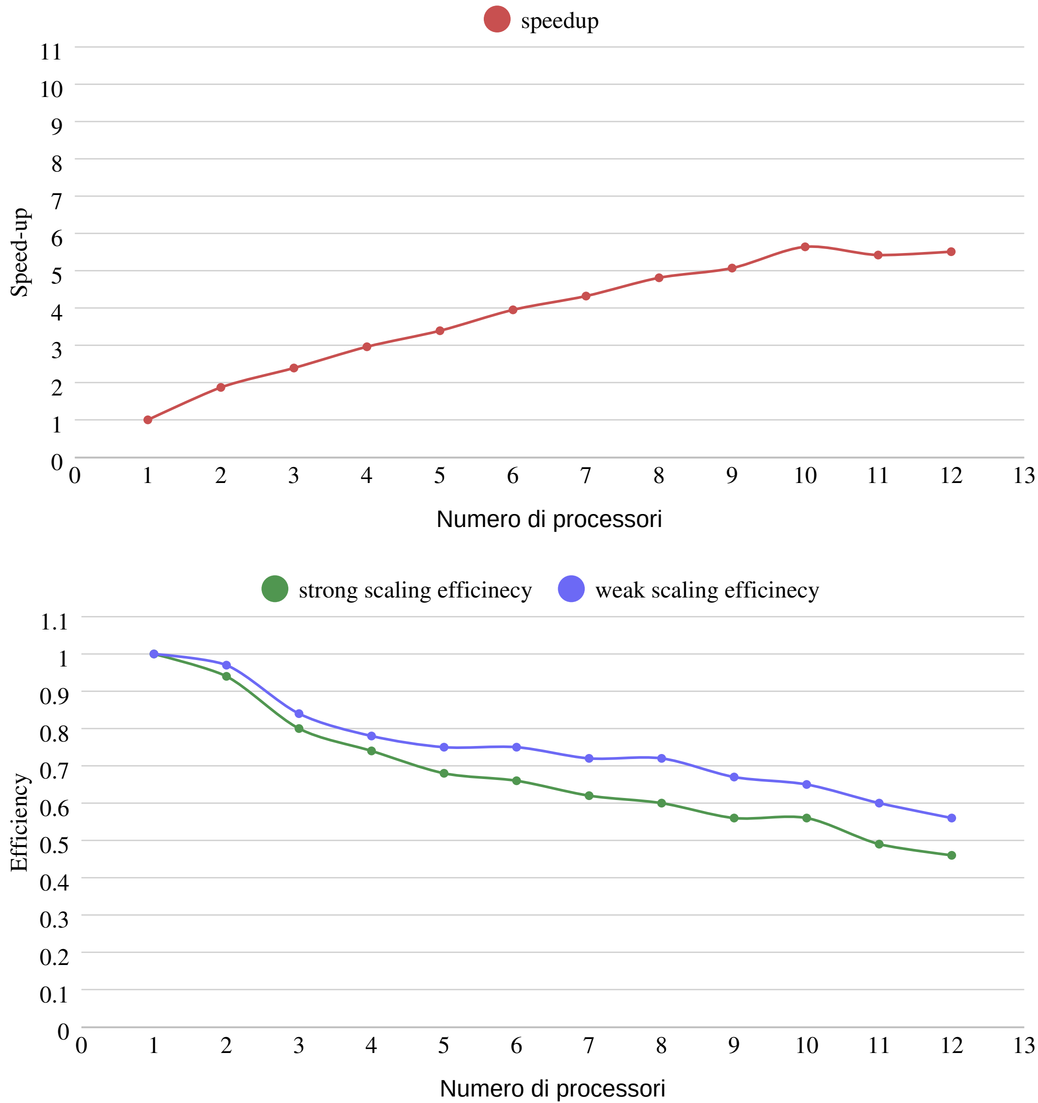


Figura 6.8: Scalabilità ed efficienza del training della rete 784 : 2048 : 1024 : 512 : 10 su processori Xeon.

Il confronto di prestazioni Raspberry-Xeon sulla rete neurale 784 : 240 : 120 : 10 è chiaramente a favore dei processori Xeon, infatti, il Raspberry Pi in quel test raggiunge la saturazione della memoria cache. Si sono confrontate quindi le prestazioni sulla rete neurale di dimensioni ridotte (784 : 30 : 10), e in figura 6.9 è mostrato il confronto Xeon-Raspberry Pi. L'asse y indica il tempo necessario a completare la fase di training, composta da 300 epoche di dimensione 40000. Come nel caso della trasformazione affine il processore

Xeon (a parità di core utilizzati) è circa quattro volte più veloce del Raspberry Pi.

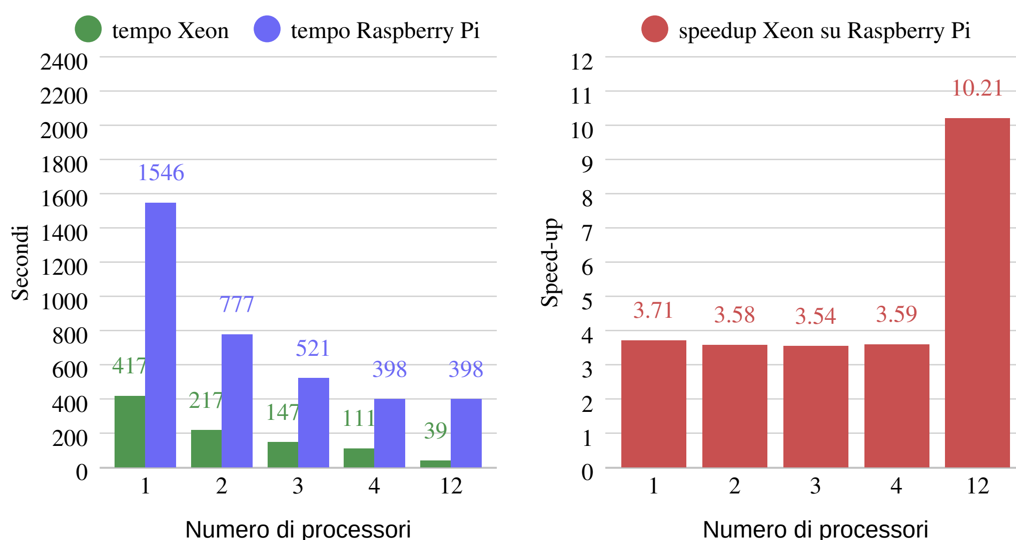


Figura 6.9: Differenza di prestazioni tra Raspberry Pi e processori Xeon sul training della rete neurale 784 : 30 : 10.

Le differenze di prestazioni tra versione SIMD e sequenziale sono sottolineate nella figura 6.10. Il test è svolto con un solo processore e con una rete neurale di dimensione 784 : 240 : 120 : 10. Si ottengono incrementi di prestazioni importanti attraverso l'uso delle SIMD: nel processore Xeon il training è svolto oltre tre volte più veloce, nel Raspberry Pi l'addestramento ha impiegato circa la metà del tempo.

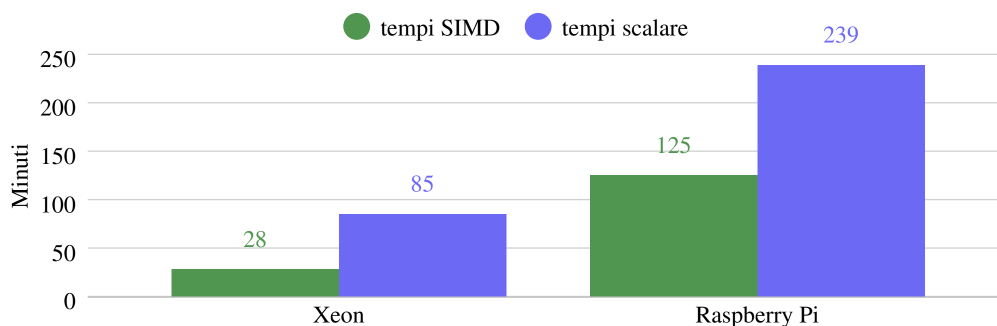


Figura 6.10: Differenza di prestazioni tra versione SIMD e sequenziale sul training della rete neurale 784 : 240 : 120 : 10.

Un ultimo test è stato svolto con l'ausilio dello strumento di analisi delle prestazioni del kernel Linux: perf. In questo test si vuole sottolineare la correlazione tra dimensione del working-set, cache-miss e *speed-up*. Lo strumento non risulta disponibile per il kernel installato sul Raspberry Pi, perciò il test è stato svolto sulla macchina con i processori Xeon. Le prove prevedono di addestrare prima una rete neurale 784 : 1536 : 10, poi una di dimensioni 784 : 100 : 10. All'aumentare del grado di parallelismo aumenta anche in numero di reti neurali (cloni), facendo crescere la dimensione del working-set.

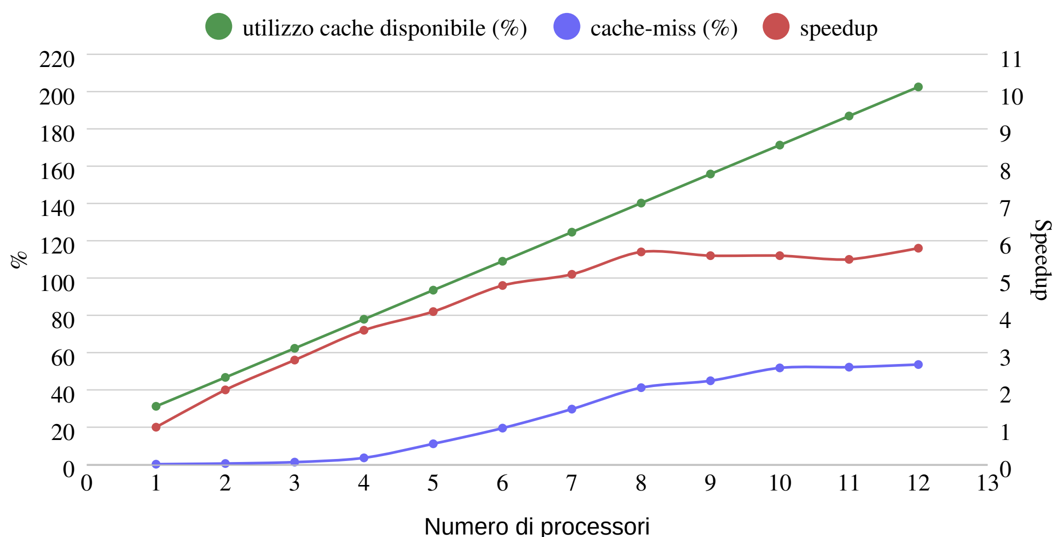


Figura 6.11: Working-set che cresce oltre la dimensione della memoria cache con l'aumentare del grado di parallelismo.

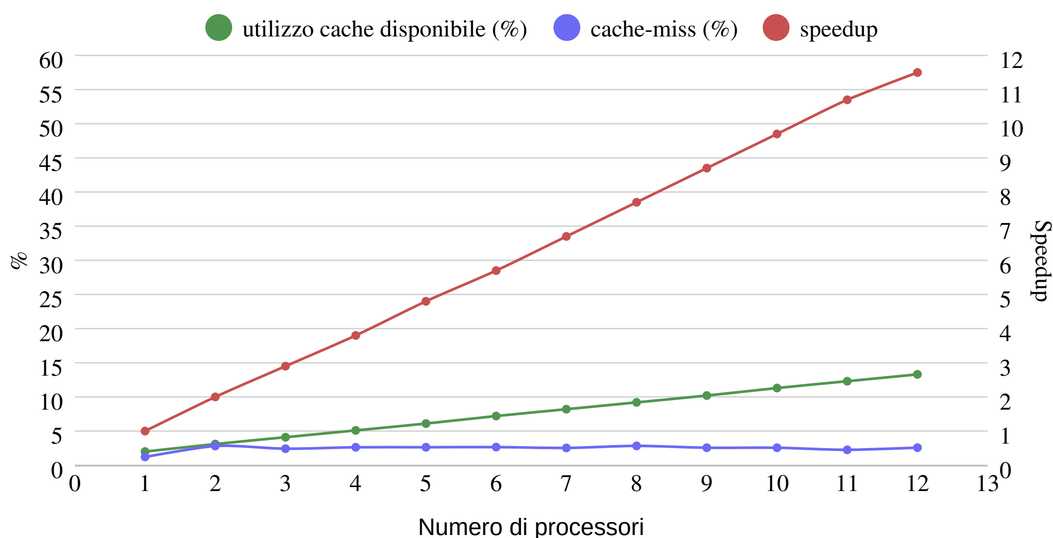
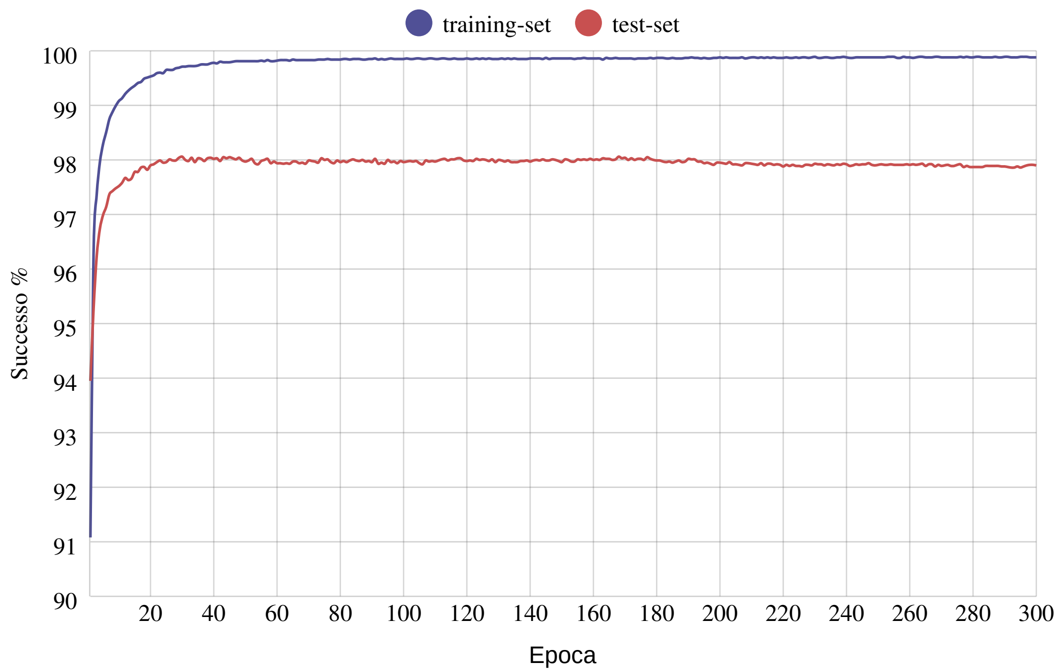


Figura 6.12: Working-set di dimensione inferiore alla memoria cache.

Il valore "utilizzo cache disponibile (%)" indica in percentuale la dimensione totale delle reti neurali (quindi del working-set) divisa per la quantità di memoria cache.
Nalla figura 6.11 si può notare che le prestazioni iniziano a peggiorare, in termini di *speed-up*, nel momento in cui si raggiunge una dimensione del working-set pari a quella della memoria cache. Queste condizioni obbligano l'hardware a fare accessi in memoria centrale, con relativo rallentamento delle operazioni. Il Raspberry Pi sembra soffrire molto più pesantemente di questo fenomeno rispetto ai processori Xeon.

6.3 Risultati machine learning

Per completezza della trattazione si descrivono i risultati della la rete neurale artificiale implementata. I test di seguito riportati sono stati effettuati con una rete 784 : 240 : 120 : 10 e il training è stato svolto su 300 epoche di dimensione 40000. La figura 6.13 mostra l'andamento del training senza trasformazioni affini applicate al training-set: già dalle prime epoche il tasso di successo sul training-set è molto più alto rispetto a quello sul test-set. L'effetto *overfitting* si manifesta proprio nel momento in cui il sistema rimane in stallo, senza evolversi ulteriormente, con un tasso di successo sul training-set prossimo alla perfezione (come in questo caso). Si raggiunge così una precisione effettiva di circa 98%, senza poter più migliorare a causa del completo adattamento della rete neurale al training-set.



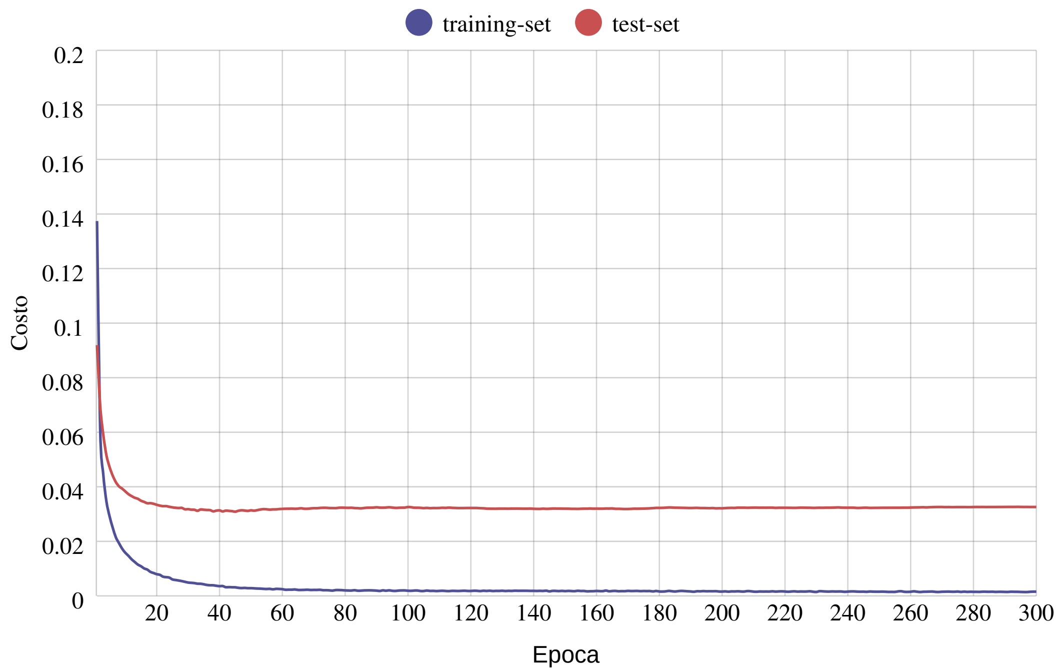


Figura 6.13: Training senza trasformazione affine sul training-set.

In figura 6.14 è mostrato l'andamento del training applicando la tecnica della trasformazione affine sul training-set: il tasso di successo sul test-set è sempre maggiore a quello del training-set. Difficilmente si presenteranno problemi di *overfitting* per via della moltitudine d'immagini differenti possibili. Ogni cifra del training-set viene casualmente trasformata in larghezza, altez-

za, rotazione, e spostamento, generando un elevato numero di combinazioni che permettono al sistema d'imparare su molti più esempi. In questo caso si riescono a raggiungere precisioni di circa 99.3%.

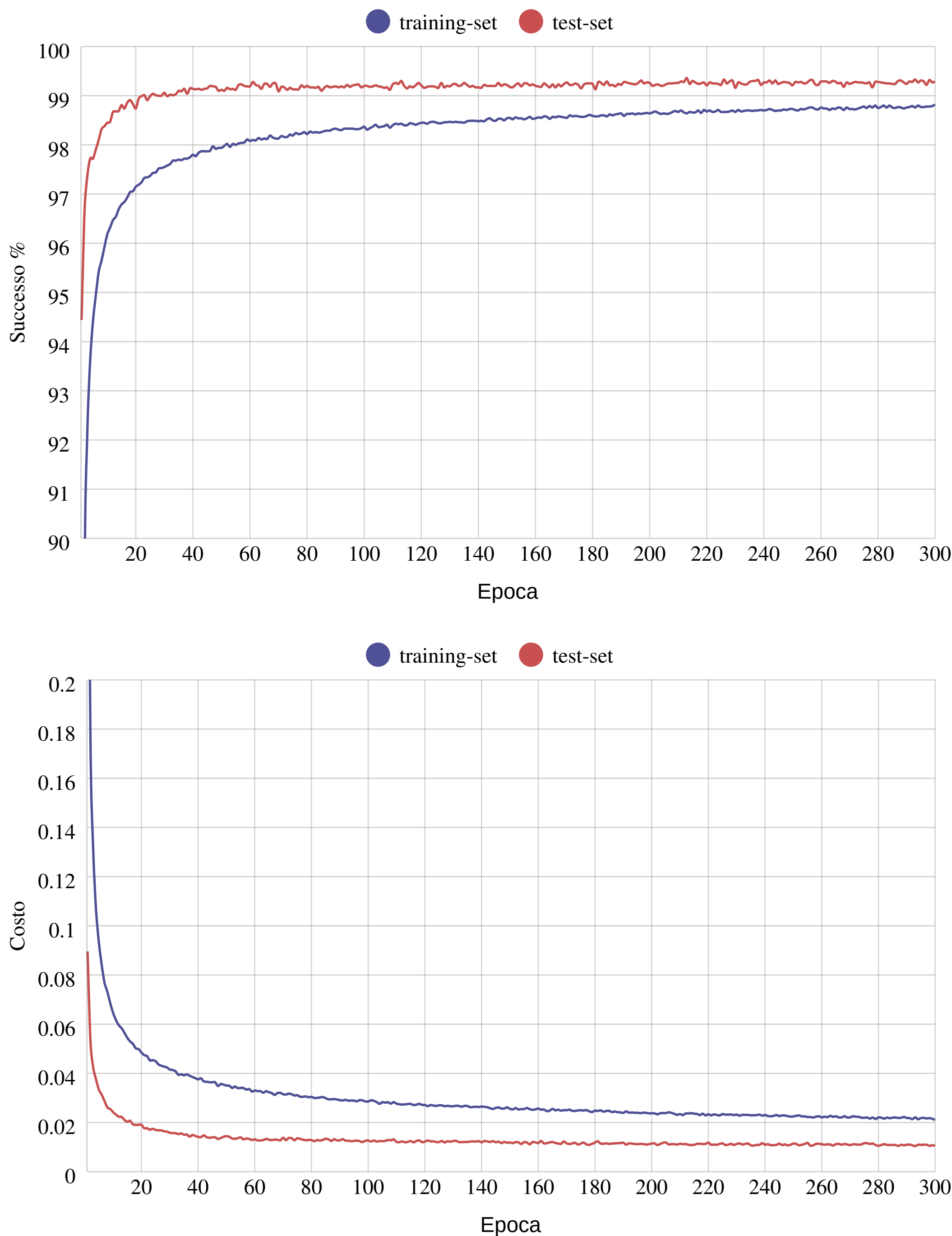
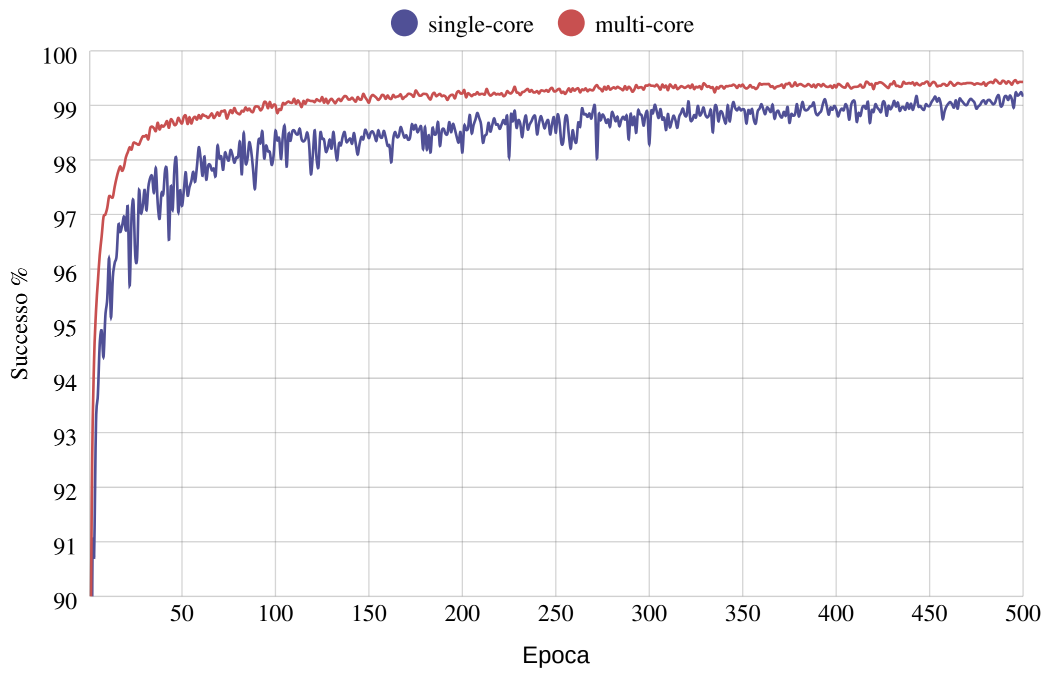


Figura 6.14: Training con regolarizzazione dei pesi e con trasformazione affine sul training-set.

La tecnica di parallelizzazione del training (quella che prevede di clonare la

rete neurale per ogni thread disponibile) non garantisce *speed-up* in termini di velocità di addestramento. Per testare la tecnica adottata si sono svolte due sessione di training con una rete neurale 784 : 360 : 240 : 120 : 10: la prima sessione in modo sequenziale, la seconda con l'utilizzo dei cloni. Nel training sequenziale si hanno 500 epoche di dimensione 10000, mentre in quello parallelo 500 epoche di dimensione 120000 (12 core), tutti gli altri parametri (es. *learning rate*) restano invariati. In questo modo entrambe le fasi di training durano circa 20 minuti. I risultati ottenuti in termini di funzione di costo e precisione sul test-set sono mostrati in figura 6.15, confermando che la sessione di training parallelo ottiene i medesimi risultati in tempi più brevi.



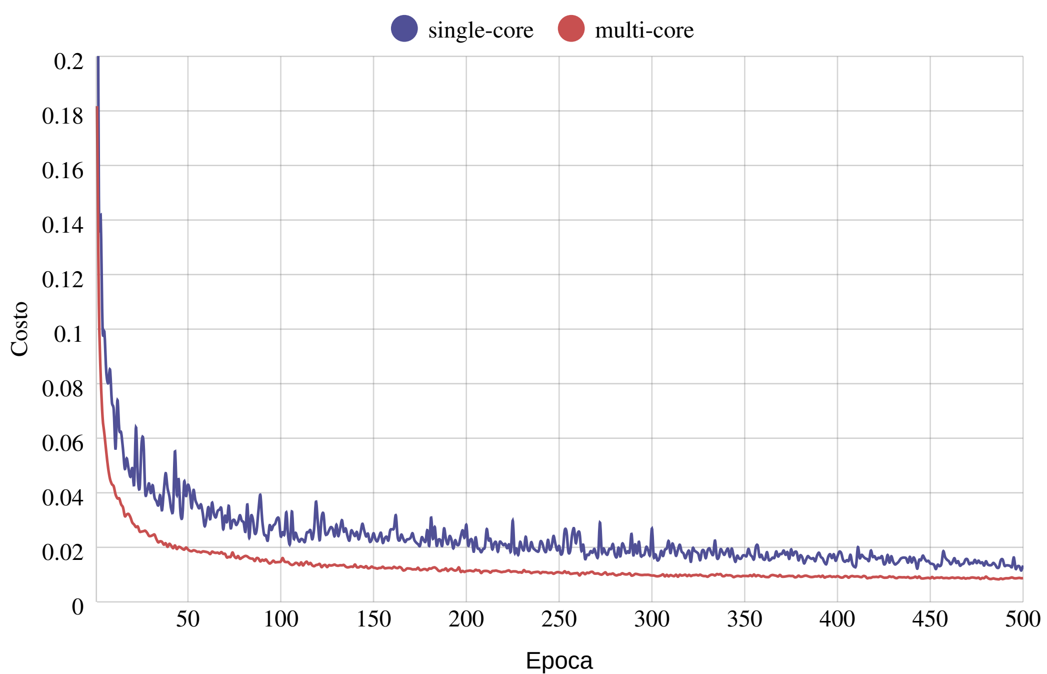


Figura 6.15: Differenza tra training sequenziale e parallelo nello stesso arco di tempo.

Conclusioni

Questo progetto ha come obiettivo quello di scoprire se, in un contesto di *parallel computing*, le prestazioni ricavate da un *System on Chip* (in questo caso il Raspberry Pi) siano accettabili. Per paragonare i risultati ottenuti con qualcosa di più collaudato, si sono poi svolti gli stessi test su una macchina server dotata di due processori Intel Xeon. Lo studio è stato svolto al fine di individuare hardware alternativo nel settore dell'*high performance computing*.
Come duplice obiettivo vi è quello di sviluppare un'applicazione *"real life"* che permetta di eseguire dei *benchmark* in contesti reali. A tale scopo si è voluto studiare la materia del *deep learning*, e implementare una rete neurale artificiale che cerca di riconoscere le cifre numeriche scritte a mano libera.
I risultati ottenuti sottolineano le difficoltà che questi dispositivi (Raspberry Pi) hanno nell'eseguire calcolo parallelo in certi contesti. Tenendo conto dell'architettura hardware però se ne derivano le cause: i test svolti hanno sottolineato che il Raspberry Pi ha difficoltà nei casi in cui il processore deve utilizzare intensamente la memoria centrale. In particolare il caso analizzato dimostra che quando il *working-set* del programma è di dimensioni maggiori della memoria cache, le prestazioni diminuiscono pesantemente per via dei molti accessi in memoria centrale. In tutti gli altri casi, invece, ha dimostrato ottime prestazioni, tenendo conto che la scheda è caratterizzata da costi e consumi ridotti rispetto a un processore x86. A parità di numero di *core* utilizzati il Raspberry Pi è soltanto circa quattro volte più lento di un processore Xeon (entrambi nelle loro condizioni ottimali di lavoro). Non si dispone di dati a sufficienza per paragonare queste due architetture in termini di prestazioni per watt in modo accurato, tuttavia se si considera il consumo nominale di questi dispositivi, si ottiene che il Raspberry Pi 3 B+ consuma circa 13 volte in meno rispetto al processore Xeon utilizzato (6W contro 80W).
La conclusione, derivata dai risultati ottenuti, è che questo tipo di dispositivo non sia in grado di inserirsi in ambiti di *high performance computing*, o meglio, ne sarebbe in grado solo in applicazioni specifiche, che andrebbero valutate

caso per caso. Il processore ARM Cortex-A53 montato sul Raspberry Pi fornisce ottime prestazioni, che però sono destinate a essere perdute per via della lentezza dell'intercomunicazione con la memoria centrale. Per alcuni casi specifici, l'utilizzo di questo dispositivo potrebbe essere una soluzione alternativa, caratterizzata dai bassi costi e bassi consumi. Nei contesti dove il tempo di esecuzione è critico, invece, questo dispositivo non è sicuramente la soluzione migliore da adottare.

Questa tesi si concentra sull'analisi della capacità di effettuare calcolo parallelo su questi dispositivi, lasciando altri aspetti dell'*high performance computing* inesplorati. Tra gli sviluppi futuri vi è sicuramente quello di provare l'applicazione di machine learning, o comunque un'applicazione dai requisiti simili, in un sistema distribuito di Raspberry Pi. Questo permetterebbe di analizzare il comportamento del dispositivo in un contesto di calcolo distribuito, dove quindi vi è anche intercomunicazione a livello di rete. Infatti l'ultima versione di questa scheda (uscita nel marzo 2018) ha fatto un notevole passo avanti sulla velocità della porta ethernet.
Un altro possibile sviluppo è quello di testare la scheda su diverse applicazioni parallele, analizzando se fosse possibile trarre vantaggio, in termini di costi, dall'utilizzo di questa famiglia di dispositivi. Sarebbe anche interessante svolgere dei test approfonditi sull'efficienza in termini di prestazione per watt ricavata dal Raspberry Pi o da dispositivi simili.

Questa tesi mi ha permesso di apprendere ancora più in dettaglio molti aspetti del calcolo parallelo, già affrontato nel corso di High Performance Computing. Inoltre grazie allo sviluppo dell'applicazione di machine learning ho appreso e messo in pratica alcuni concetti base di questa materia, rendendomi conto di quali siano le sue potenzialità. Si è sperimentato con mano che una cura dei dettagli implementativi, destinati ad un uso più completo dell'hardware, porta benefici sostanziali.

Appendice A

Esecuzione della rete neurale

Il repository del progetto è situato all'indirizzo https://bitbucket.org/edobrb/dnn-hpc e comprende tutto il codice scritto per la realizzazione della tesi.
Per compilare il progetto è sufficiente eseguire il comando:

```
make
```

o nei casi in cui ci si trovi in un ambiente ARM:

```
make ARM=1
```

Viene prodotto l'eseguibile *mnist-dnn* ed è possibile avviare il training della rete neurale artificiale tramite il comando:

```
./mnist-dnn < settings.txt
```

dove *settings.txt* è il file di configurazione del programma, un esempio può essere il seguente:

```
hidden_layers_count=2           //numero di livelli nascosti
layer_size=240                  //dimensione livello 1
layer_size=120                  //dimensione livello 2 ...
h=0.15                          //learning rate iniziale
final_h=0.05                    //learning rate finale
regularization_force=5          //forza di regolarizzazione L1
backprop_threshold=0.9          //margine di backpropagation
affine_traformation=1           //trasformazione affine {0,1}
r=15                            //rotazione in gradi +-r
s=0.15                          //ridimensionamento 1+-s
t=2                             //spostamento +-t
epochs=300                      //numero di epoche
epoch_size=60000                //dimensione dell'epoca
```

Per cambiare il grado di parallelismo dell'applicazione si può utilizzare il comando:

```
OMP_NUM_THREADS=x ./mnist-dnn < settings.txt
```

dove x è il numero di core che si vogliono usare. Se si desidera eliminare il supporto SIMD è sufficiente commentare la riga:

```
#define SIMD_SUPPORT
```

all'interno del file sorgente *mnist-dnn.c*, e ricompilare il codice. Per ulteriori informazioni e/o aggiornamenti si consiglia di consultare il README.md all'interno del repository.

Parole chiave

System on Chip
Raspberry Pi
Calcolo parallelo
High performance computing
Machine learning
Deep learning
Trasformazione affine
MNIST database

Ringraziamenti

Il ringraziamento più sentito è per la mia famiglia, Stefano, Sara e Davide che mi hanno sempre incoraggiato e sostenuto anche nei momenti più difficili. Un grazie speciale ai miei genitori che hanno fatto grandi sacrifici per permettermi di seguire questo percorso.

Un grazie ai miei amici di sempre, con cui ho passato i momenti migliori, e che mi hanno fatto pensare ad altro oltre che allo studio durante questi tre anni.

Grazie ai miei compagni di studi, Emanuele e Lorenzo, con i quali ho affrontato questo percorso con molta più gioia. Non solo compagni ma amici.

Un sentito ringraziamento va al professore Moreno Marzolla che mi ha dato la possibilità di svolgere questo progetto, dedicandomi parte del suo tempo sia come insegnante che come relatore di questa tesi.

Bibliografia

[1] Nicola Cabibbo. Computer. calcolo parallelo. *Enciclopedia della Scienza e della Tecnica, Treccani*, 2008. http://www.treccani.it/enciclopedia/computer-calcolo-parallelo_(Enciclopedia-della-Scienza-e-della-Tecnica)/.

[2] Raffaele Cappelli. Fondamenti di elaborazione di immagini - operazioni sulle immagini. Ingegneria e scienze informatiche – Università di Bologna.

[3] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926, May 2017.

[4] Raspberry Pi Foundation. https://www.raspberrypi.org.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[6] Patrick J Grother. Handprinted forms and characters database. Technical report, National Institute of Standards and Technology, 1995. https://www.nist.gov/sites/default/files/documents/srd/nistsd19.pdf.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[8] Andrej Karpathy. Hacker's guide to neural networks. http://karpathy.github.io/neuralnets/.

[9] V. Khomenko, O. Shyshkov, O. Radyvonenko, and K. Bokhan. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. In *2016 IEEE First International Conference on Data Stream Mining Processing (DSMP)*, pages 100–103, Aug 2016.

[10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database. http://yann.lecun.com/exdb/mnist/.

[11] ModMyPi. Tabella di comparazione tra raspberry pi. https://www.modmypi.com/blog/raspberry-pi-comparison-table.

[12] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[13] Parallel and Distributed Simulation Research Group. Raspèin project. Technical report, University of Bologna, 2015. http://pads.cs.unibo.it/doku.php?id=raspein:index.

[14] Jurgen Schmidhuber. Multi-column deep neural networks for image classification. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pages 3642–3649, Washington, DC, USA, 2012. IEEE Computer Society.