View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AMS Tesi di Laurea

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

INGEGNERIA INFORMATICA M

TESI DI LAUREA MAGISTRALE

in

Logiche Riconfigurabili M

PROGRAMMAZIONE DI CONVOLUTIONAL NEURAL NETWORKS ORIENTATA ALL'ACCELERAZIONE SU FPGA

CANDIDATO
Alessandro Maragno

RELATORE
Prof. Stefano Mattoccia

CORRELATORE
Dott. Matteo Poggi

Anno Accademico 2015/16

Sessione II

INDICE:

CAPITOLO 1:

INTRODUZIONE

1.1 Il Contesto Applicativo

La *Computer Vision*, o visione artificiale, ha il principale obiettivo di estrapolare informazioni a partire da immagini acquisite digitalmente. Tali informazioni dette *features* (caratteristiche), possono essere utilizzate da un sistema al fine di prendere decisioni volte all'interazione con il mondo circostante, o essere elaborate per ottenere altre informazioni a un livello più raffinato, sempre con lo stesso fine. Oggigiorno, le conquiste ottenute in questo settore, hanno permesso a tale disciplina informatica di evolversi e raggiungere uno stato di maturità tale da consentirne l'applicazione nella pratica, rendendo possibile il compimento di diversi *task* in svariati ambiti applicativi. Tra questi, è sicuramente possibile citare l'ambito industriale, passando per quello dell'automazione e della robotica, fino ad arrivare anche a quello medico e dell'assistenza di persone affette da cecità [4], o anche a quello video-ludico, della videosorveglianza [1], fino a pervadere direttamente o indirettamente la vita di tutti i giorni.

Attualmente, l'interesse generale nel settore è orientato verso la *Object Detection*, ovvero l'individuazione di oggetti all'interno di un'immagine. Per oggetti s'intende qualunque entità fisica, che si tratti di persone, animali, oggetti propriamente detti, o anche dettagli di essi. Lo strumento più potente a disposizione per il riconoscimento di oggetti è la *Convolutional Neural Network* (rete neurale convolutiva), conosciuta anche con il suo acronimo (CNN) o con il nome di *ConvNet*. Si tratta di un sistema basato su un modello matematico, quindi implementabile sia in maniera software che hardware, il quale oltre a individuare features nelle immagini, è anche in grado di attribuir loro una classificazione, dando quindi una semantica all'immagine stessa. Tali sistemi hanno cominciato a diffondersi al termine degli anni '90, quando alcuni ricercatori hanno messo a punto le prime reti neurali convolutive [5] di successo in grado di leggere numeri e caratteri, e riconoscere codici postali. Tra le varie applicazioni delle CNN è possibile annoverare, oltre che il riconoscimento di oggetti,

anche l'analisi del contesto delle immagini, e successivamente di video per quanto riguarda il settore della visione artificiale, senza contare svariate applicazioni in altri settori, quali il riconoscimento del linguaggio naturale o la strutturazione automatica di molecole costituenti un farmaco. Inoltre, le CNN possono essere utilizzate efficacemente anche per problemi di *low-level vision* come mostrato recentemente in[2,3].

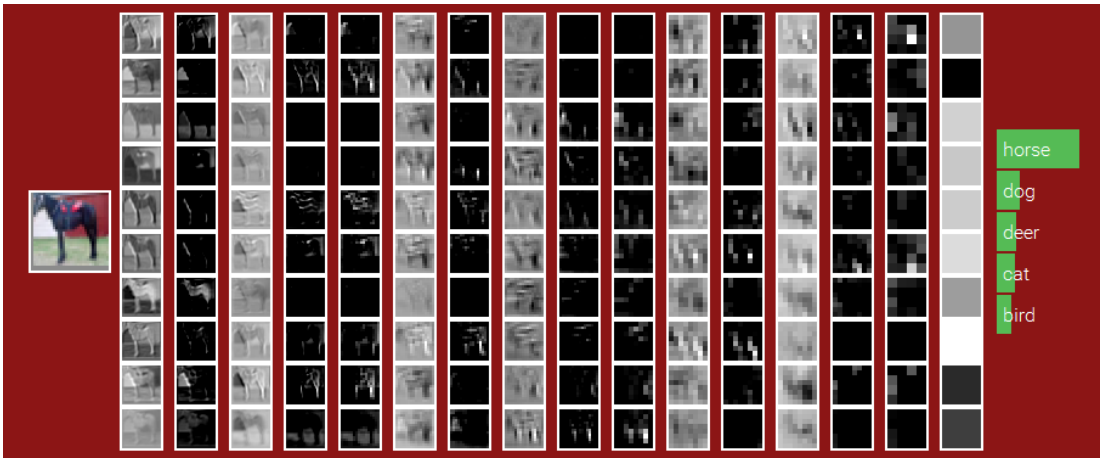


Figura 1.1: Esempio di rete neurale convolutiva in esecuzione sulla homepage del corso "Convolutional Neural Networks for Visual Recognition" dell'università di Stanford. L'immagine più a sinistra è l'input della rete, poi si notano i risultati delle elaborazioni intermedie, e infine sulla destra la classificazione attribuita all'input.

A scapito del loro grande potenziale e dell'estrema utilità, le reti neurali convolutive richiedono molta capacità computazionale e ampio utilizzo di memoria per poter processare i dati secondo il loro modello matematico e compiere, quindi, il loro *task*. Difatti, sono eseguite su unità di elaborazione molto potenti, tipicamente GPU, pertanto una delle sfide più significative al giorno d'oggi in questo settore, è l'esecuzione di tali sistemi **in tempo reale**, anche su unità di elaborazione più modeste e con una disponibilità di energia ridotta, come lo sono i **sistemi embedded**. Infatti, negli ultimi anni sono stati raggiunti buoni risultati anche per l'accelerazione di reti neurali convolutive su FPGA, come dimostrato dall'architettura nnX [6] sviluppata dalla compagnia Teradeep, uno spin-off di origini universitarie.

1.2 Obiettivo della tesi

In questo trattato sarà espostala realizzazione di un sistema software il quale implementa una Convolutional Neural Network riconfigurabile. Nella fase di sviluppo si è tenuto conto di caratteristiche strutturali quali la modularità e la semplicità di estensione futura del software. Inoltre,l'implementazione è stata svolta con l'idea che tale sistema sarà successivamente portato su architettura FPGA allo scopo di accelerarne le operazioni a seguito di ottimizzazioni ad-hoc, le quali non sono previste nell'ambito della presente tesi. Per la stessa motivazione, si è scelto di scrivere il software in linguaggio C, grazie alla sua capacità caratteristica di fornire un adeguato livello di astrazione, ma al contempo anche un buon grado di controllo della macchina sottostante. Dato che lo scopo principale di tale progetto è l'accelerazione della rete su FPGA, il lavoro non prende in considerazione la realizzazione di meccanismi di apprendimento automatico, tema che sarà comunque argomentato in seguito per ragioni di completezza. Pertanto, si è tenuto conto anche della necessità di configurare il sistema nel modo più generale possibile, eseguendo il caricamento dei parametri strutturali e operativi della rete già raffinati, da un file di configurazione. Il punto di partenza da cui configurare la rete consiste in modelli di reti neurali convolutive tipici di Torch [7], un framework implementato in linguaggio Lua per la computazione scientifica e di algoritmi di apprendimento automatico. Tali modelli di partenza sono stati opportunamente convertiti per la configurazione del sistema in discussione. L'utilizzo di questo strumento è avvenuto anche in fase di *testing* per la verifica delle funzionalità del sistema sviluppato.

CAPITOLO 2:

STRUMENTI UTILIZZATI

2.1 Fondamenti Teorici

2.1.1 Fondamenti di Computer Vision e Object Detection

La *computer vision* è spesso basata su tecniche di *image processing*[8], ovvero di elaborazione dell'immagine, grazie alla quale è possibile manipolare a livello di intensità dei pixel le immagini da cui estrarre le informazioni rendendo evidenti le *feature* più rilevanti. La manipolazione può avvenire mediante due diversi tipi di operatori [8]: gli *operatori puntuali* e gli *operatori locali*.

Gli operatori puntuali eseguono una trasformazione del pixel correntemente elaborato senza tenere conto dei pixel presenti nel suo intorno, e spesso basandosi sull'istogramma dei livelli tipico dell'immagine in fase di processing. Esempi tipici di applicazione di operatori puntuali sono il *thresholding*, o il *contrast stretching*, tra gli altri.

Gli *operatori locali* invece, processano comunque l'immagine pixel per pixel ma tenendo conto anche del valore dei pixel presenti in un loro intorno prestabilito. Nel caso specifico degli *operatori lineari spazio-invarianti* ciò avviene secondo il meccanismo della *sliding window*, il quale prevede che una piccola finestra chiamata *kernel*, opportunamente dimensionata e contenente i valori dell'operatore, venga fatta traslare lungo tutta l'area dell'immagine. In particolare, il kernel è centrato su di un pixel che, dopo l'applicazione dell'operatore, avrà in output un nuovo valore dipendente dagli altri pixel nel suo intorno coperti dal kernel. L'applicazione dell'operatore non comporta la sovrascrittura del pixel elaborato nell'immagine di input, ma genera dunque un'immagine di output con le stesse dimensioni della prima i cui pixel contengono i rispettivi valori trasformati. Questo procedimento è applicato a tutti i pixel presenti nell'immagine in elaborazione in modo da realizzare lo scorrimento della finestra. Si noti che nel caso degli operatori lineari spazio-

invarianti, i valori presenti nel kernel non variano a ogni spostamento ma sono costanti.

Per quanto riguarda l'applicazione dell'operatore, invece, questa consiste nella *convoluzione* [8] tra i valori contenuti nel kernel (che caratterizzano l'operatore stesso), e i valori dei pixel dell'immagine in elaborazione da esso coperti in quel momento. La convoluzione risulta essere dunque l'operazione cardine per l'applicazione degli operatori locali, e deriva dalla teoria dei segnali. Concettualmente, è possibile esprimerla come il confronto tra due segnali, il primo dei quali è fermo, mentre l'altro viene fatto traslare nel tempo e sovrapposto al primo. Matematicamente, per il caso continuo e monodimensionale, la convoluzione è esprimibile come:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau) - \ g(t - \ \tau) d\tau$$

Formula 2.1: Convoluzione di due segnali nel dominio del tempo

Il risultato di questa operazione è interpretabile come una misura della *risposta impulsiva* di *g(t)* al segnale *f(t)* nell'istante τ.

Nell'elaborazione delle immagini, queste possono essere interpretate come segnali discreti bidimensionali, le cui componenti (valori d'intensità dei pixel) hanno una frequenza spaziale piuttosto che temporale. Grazie a tale interpretazione, è possibile calcolare la convoluzione come risposta impulsiva di un determinato operatore locale all'immagine in ingresso ottenendone la trasformazione, secondo la formula:

$$I(i,j) * H(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(m,n) \ H(i-m, j-n)$$

Formula 2.2: Convoluzione tra due segnali discreti bidimensionali.

in cui *k* risulta essere il fattore di dimensione del kernel rappresentante l'operatore, il quale avrà larghezza ed altezza pari a*(2k+1)*.È possibile notare quanto la forma della convoluzione discreta sia simile a quella di un prodotto scalare tra le componenti di

due vettori. In tale contesto, gli operatori lineari spazio-invarianti sono detti anche filtri.

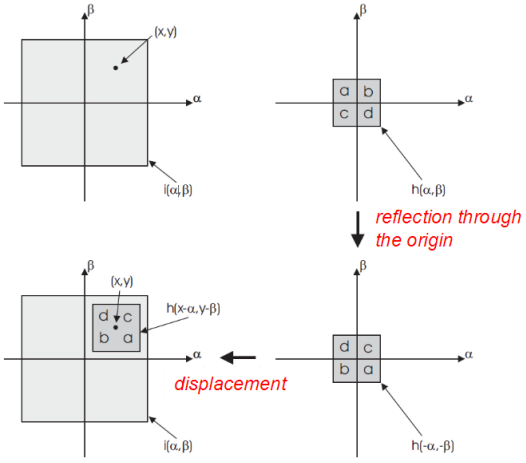
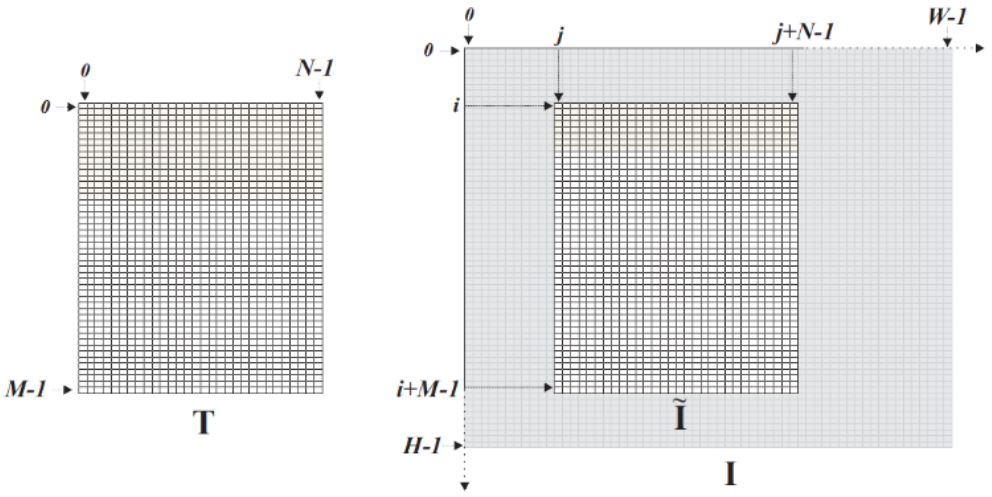


Figura 2.1: Rappresentazione grafica della convoluzione bidimensionale[8]

Tra i più classici esempi di utilizzo degli operatori lineari spazio-invarianti, vi sono la riduzione del rumore nelle immagini (*denoising*) e l'accentuazione dei bordi (*edge sharpening*).

Il meccanismo dell'applicazione degli operatori lineari spazio-invarianti è stato d'ispirazione anche per uno dei più importanti obiettivi per la Computer Vision: la *object detection*. Infatti, una delle più intuitive tecniche di riconoscimento di oggetti all'interno delle immagini, chiamata *Template* (o *Pattern*) *Matching* [8], prevede la costruzione di un template, sarebbe a dire un'immagine contenente un oggetto da riconoscere, che è poi ricercato all'interno di un'immagine, detta *immagine target*, anch'esso grazie al meccanismo di sliding window. Pertanto, il template (di dimensioni inferiori rispetto all'immagine in cui è ricercato) è posizionato di volta in volta registrando la sua origine con un pixel dell'immagine target e successivamente viene calcolata una misura di similarità tra di esso e l'area dell'immagine su cui si trova sovrapposto. Generalmente, vi sono diversi modi per stimare la misura di similarità, ma i modi più efficaci hanno la forma di un prodotto scalare tra le intensità dei pixel (o i gradienti delle intensità per maggior robustezza della stima, come nel caso dello *Shape-based Matching* [8]) del template e quelle dei pixel nell'area dell'immagine target a esso sottesa.



$$NCC(i,j) = \frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m, j+n)\cdot T(m,n)}{\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} I(i+m, j+n)^2}\ \cdot\ \sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} T(m,n)^2}}$$

Figura 2.2: Esempio grafico di Template Matching e di misura di similarità (Normalized Cross-Correlation). L'algoritmo prevede che l'origine del template T venga traslata sulla target image I di pixel in pixel facendo sì che il template sovrasti l'area indicata come Ĩ per effettuarne il calcolo della misura di similarità.

2.1.2 Convolutional Neural Networks (CNN)

Prima di descrivere l'architettura delle Convolutional Neural Networks in modo più dettagliato, è bene introdurre le generiche reti neurali, poiché le CNN risultano essere un caso più specifico di queste ultime.

Una *rete neurale* è un sistema basato su un modello matematico, ispirato al funzionamento delle reti neurali biologiche. Tale modello è impiegato per attribuire dati non classificati a diverse categorie prestabilite. La potenza di tale strumento risiede nel fatto che si avvale di tecniche di *machine learning* [9] allo scopo di raffinare il proprio modello matematico e classificare i dati in modo più efficace. Per machine learning, ovvero apprendimento automatico, s'intende quell'insieme di tecniche e strategie volte a migliorare le prestazioni di un sistema nell'esecuzione del suo task in base all'esperienza del sistema stesso. Difatti, l'apprendimento

automatico avviene in diverse fasi, a partire da una certa base di dati opportunamente suddivisa:

1. *Fase di **Preprocessing** dei dati e **inizializzazione** del modello*: in questa fase preliminare, i parametri del modello matematico della rete neurale sono inizializzati. L'inizializzazione può avvenire in diversi modi, ma in generale avviene con piccoli numeri casuali vicini allo zero. Il set di dati, invece, è ripartito nell'insieme di training contenente dati già classificati, che sarà utilizzato nella fase di seguito illustrata, l'insieme di validazione ed infine quello di test.

2. *Fase di **Training***: l'insieme di training è suddiviso ulteriormente in sottoinsiemi più piccoli chiamati *batch*. Per ogni dato appartenente a un batch e fornito in ingresso alla rete neurale, questa calcola una *score function (funzione di punteggio),* la quale assegna al dato un punteggio per ogni categoria possibile, ed infine questo viene classificato nella categoria per cui ha ricevuto il punteggio più alto. Al termine della classificazione di tutti i dati presenti nel batch, è calcolata una *funzione di costo* a partire dalle differenze tra le categorie predette e le rispettive categorie reali, dato per dato. Lo scopo della rete neurale è quello di minimizzare tale funzione al fine di ottenere il set di parametri più efficace per la classificazione. Ciò può avvenire con diversi meccanismi di aggiornamento dei parametri come ad esempio il più importante e diffuso, ovvero la *backpropagation* [10]. Tale meccanismo sfrutta la *regola della catena* dei gradienti per far sì che la rete riceva un feedback riguardante la classificazione fatta, e possa aggiornare i propri parametri. L'aggiornamento avviene anche in funzione di un *iperparametro* chiamato *learning rate*, ovvero tasso di apprendimento. Tale procedimento è iterato per ogni batch finché non si ottiene l'insieme di parametri che ottimizza la funzione di costo.

3. *Fase di **Validazione***: in tale fase è sfruttato l'insieme di validazione introdotto in precedenza. I dati in esso contenuti sono processati dalla rete con il set di parametri ottimale derivante dalla fase precedente, e lo scopo di questa fase è quello di effettuare il *"tuning"* (raffinamento) di iperparametri

che definiscono strutturalmente il modello matematico adoperato. Infatti, una rete neurale ha una struttura a grafo aciclico diretto ordinato in vari livelli, che possono essere di due tipi: esposti (es: input e output) o nascosti (comprendono i nodi che eseguono il calcolo della score function e dei gradienti della funzione di costo). Ogni nodo di un livello è collegato in maniera *feed-forward* a tutti i nodi del livello successivo (livelli *fully-connected*). Alcuni esempi di iperparametri possono essere, quindi, il numero di *hidden layers* (livelli nascosti), o il numero di nodi presenti in ognuno di essi, o ancora il learning rate, precedentemente citato.

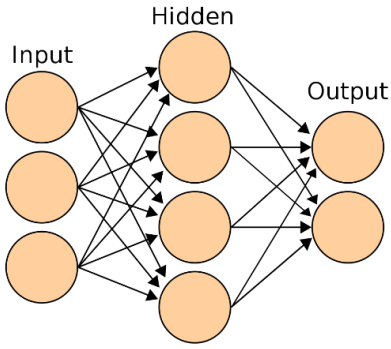


Figura 2.3: Esempio di struttura di rete neurale [9]

4. *Fase di* ***Test***: In questa terza e ultima fase, la rete raffinata nelle due fasi precedenti processa i dati presenti nell'insieme di test per misurarne le prestazioni e quindi stimarne la qualità, mediante il calcolo di parametri come, ad esempio, l'errore di classificazione globale.

Una volta terminato il procedimento di allenamento della rete appena descritto, questa è pronta per essere operativa.

È evidente che i costi di una rete neurale, in termini sia computazionali sia di occupazione della memoria, siano molto onerosi poiché, generalmente, la mole di dati da classificare è enorme, e talvolta anche il numero delle classi stesse può essere elevato. Inoltre, una buona classificazione richiede calcoli più precisi e raffinati che

si possono tradurre in un numero elevato di hidden layers, e quindi di parametri. Per questi motivi è richiesta una certa potenza di elaborazione per eseguire le reti neurali, quindi nella pratica sono utilizzate GPU, soprattutto per eseguire il training del modello.

Le reti neurali convolutive hanno lo stesso funzionamento appena descritto, ma con l'assunto che i dati in input siano i valori d'intensità dei pixel dell'immagine da elaborare. A seguito di tale assunzione, è possibile effettuare diverse ottimizzazioni che consentono di ridurre ulteriormente i costi di esecuzione della rete neurale. Tra queste, la più importante riguarda la riduzione del numero di parametri, che in questo contesto assumono il nome di *weight* (peso) ed eventualmente *bias* poiché la score function è una funzione di classificazione lineare così definita, per ogni possibile classe:

$$f(x, W, b) = \sum_{i=1}^{N} (x_i \cdot W_i) + b$$

Formula 2.3: Score function nelle vesti di una funzione di classificazione lineare

In tale formula, x_i rappresenta il valore d'intensità dell'i-esimo pixel dell'immagine da classificare composta da *N* pixel, W_i il valore del rispettivo peso, e b rappresenta un eventuale offset per il punteggio di una particolare classe. Se interpretiamo i pixel in input ed i pesi come due vettori con le rispettive componenti x_i e W_i, si nota come in questa funzione il punteggio sia calcolato come prodotto scalare tra questi due vettori, al cui risultato è sommato il bias, se previsto. Nelle normali reti neurali che utilizzano tale funzione di classificazione, vi è un peso W_i per ogni dato x_i. Ogni nodo in ogni livello ha quindi un numero di pesi pari al numero di pixel nell'immagine. Ciò rende i layer della rete di tipo fully-connected, e questo significa che il numero di parametri è molto elevato. Per quanto riguarda il set di bias, questo ha la stessa cardinalità dell'insieme di classi di uscita del layer di cui fanno parte. Notare che la presenza di bias all'interno di un layer è opzionale.

Come descritto in precedenza, l'elaborazione delle immagini, in questo caso ai fini di rilevazione e classificazione di oggetti, avviene soprattutto mediante operatori lineari

spazio-invarianti. In tale contesto, questi si traducono in template o filtri rappresentanti le feature da cercare, ovvero kernel di dimensioni limitate che godono quindi della proprietà di spazio-invarianza, poiché *se stiamo cercando una feature in un certo punto dell'immagine, allora ha senso cercarla in tutti i punti dell'immagine*[10]. Pertanto i valori del kernel non cambiano passando da un pixel all'altro dell'immagine elaborata. Tale caratteristica, dal punto di vista di una rete neurale fully-connected, si rifletterebbe in unità nascoste con set di pesi ridondanti. Da ciò ne deriva che è possibile ridurre drasticamente il numero di pesi in un livello sino alla cardinalità del template, che sarà poi traslato su tutto il volume dell'immagine in elaborazione calcolandone la funzione di classificazione lineare che assumerà il carattere di una convoluzione, ridimensionando di molto l'impatto della rete sulla memoria. Pertanto, i livelli di una rete neurale convolutiva si dicono *locally-connected* (localmente connessi). Tale ottimizzazione è detta *parameter sharing* [10].

2.1.2.1 Esempio di architettura di una CNN

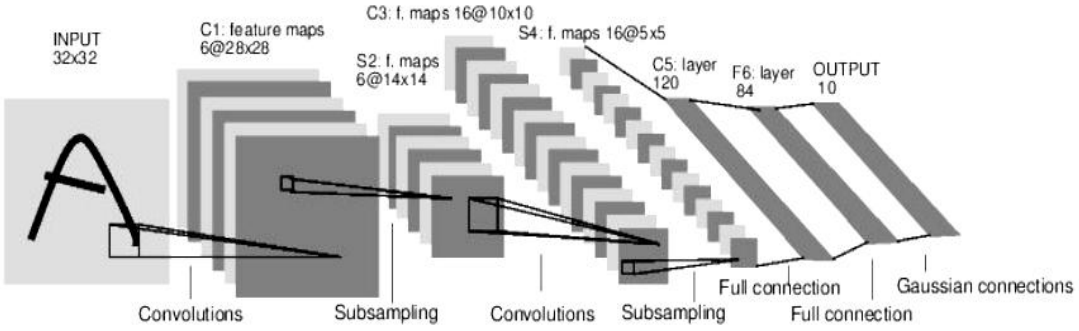


Figura 2.4: LeNet [5]. La prima applicazione di successo di una Convolutional Neural Network sviluppata da Yann LeCun negli anni '90, per la lettura di cifre e caratteri, e quindi per il riconoscimento di codici postali.

Generalmente, le Convolutional Neural Networks hanno un'architettura simile a quella mostrata nella Figura 1.4, che richiama una *pipeline* per modellare i neuroni come nodi di un grafo aciclico come quello precedentemente descritto, e composta da moduli (livelli in cui i neuroni sono organizzati) che possono essere tra loro collegati in diverse modalità: la più comune consiste nel connettere i moduli in cascata. Ogni modulo prende in ingresso un input sottoforma di volume di pixel organizzati secondo larghezza, altezza e profondità, e produce in output un altro volume di pixel, o un vettore di punteggi distribuito secondo una delle tre dimensioni. Il volume in ingresso alla CNN è dimensionato secondo larghezza, altezza e profondità dell'immagine di input, mentre i volumi prodotti dai moduli hanno dimensioni dipendenti da quelle dell'input da elaborare e da altri iperparametri caratteristici che variano da un modulo all'altro. Per comprendere il funzionamento dei moduli più nel dettaglio, saranno di seguito introdotti quelli più significativi:

1. *Modulo di* ***Convoluzione***: come si evince dal nome, svolge la funzione più importante convolvendo i valori in input con i propri pesi per calcolare la score function. Questi ultimi sono organizzati in un certo numero di kernel, di cui larghezza (*kW*) e altezza (*kH*) sono iperparametri tipici del modulo, mentre la loro profondità è pari alla profondità del volume di input. A ogni kernel è associato un eventuale bias, che in tal caso sarà sommato al risultato della convoluzione per completare il calcolo della score function. Altri

iperparametri caratteristici di questo modulo sono il passo orizzontale (*strideW*) e quello verticale (*strideH*) con cui il kernel viene scorso lungo il volume di input, e l'ammontare di *zero-padding*, ovvero l'entità del riempimento con zeri da applicare orizzontalmente (*padW*) e verticalmente (*padH*) ai bordi dell'input ad ogni profondità per modellare larghezza ed altezza del volume di output. Infine, è considerato un iperparametro anche il numero stesso di kernel/bias. È proprio da tali iperparametri che dipendono le dimensioni del volume di output, che avrà profondità pari al numero di filtri (kernel) e larghezza ed altezza (*outW* e *outH*) rispettivamente dimensionate come segue:

$$outW = \frac{(inW - kW + 2padW)}{strideW} + 1;$$

$$outH = \frac{(inH - kH + 2padH)}{strideH} + 1;$$

Formula 2.4: Equazioni per il dimensionamento del volume di output per moduli che operano mediante traslazione del kernel sul volume di input.

2. *Modulo di* ***Attivazione***: tale modulo è necessario ai fini dell'apprendimento della rete, poiché le CNN sfruttano la backpropagation per questo scopo. A questo punto, i punteggi calcolati dai moduli di convoluzione sono degli scalari, pertanto in fase di backpropagation la loro derivata sarebbe pari a zero rendendo nullo l'apprendimento della rete. Quindi, questo modulo applica una funzione, detta *funzione di attivazione* o *di trasferimento* [10], a ogni elemento appartenente al volume in input, e si trova generalmente a valle di un modulo di convoluzione. Tra le più tipiche funzioni di attivazione utilizzate, troviamo la *sigmoide*, la *rectified linear unit* (*ReLU*) e loro varianti. Processando il proprio input elemento per elemento, questo modulo produrrà un volume di output con le stesse dimensioni di quello appena elaborato.
3. *Modulo di* ***Pooling***: questa unità svolge un *subsampling* (sottocampionamento) del volume in input per ridurne larghezza ed altezza. Anche questo modulo lavora con l'ausilio di un kernel traslato lungo tutto il volume in ingresso per svolgere il subsampling. Pertanto il volume di output

avrà la stessa profondità del volume in input, mentre larghezza e altezza saranno dimensionate alla stessa maniera del modulo di convoluzione. Tra i vari esempi di pooling si riporta il *maxpooling* [10], che per ogni spostamento del kernel traspone in uscita solo il valore massimo presente in esso. L'operazione di sottocampionamento è utile in una rete neurale per prevenire l'*overfitting* [9], ovvero quel fenomeno che si manifesta quando il modello matematico si specializza troppo nel riconoscimento di un oggetto a seguito del training, senza poi riconoscerlo in caso di artefici grafici quali occlusioni, variazioni di luminosità o altri fenomeni che possono alterarne l'aspetto, rendendo la classificazione inefficace.

4. *Modulo* ***Fully-Connected****:* in genere, questi moduli si trovano a valle della rete ed eseguono una trasformazione del volume che prendono in input elemento per elemento. Tale trasformazione coincide con la funzione di classificazione lineare tipica anche dei moduli di convoluzione, pertanto questo modulo sarà caratterizzato da un set di pesi connessi con tutti gli elementi del volume in input,e un set di bias. Nel caso questo modulo si trovi a valle della rete, produce in output il vettore di punteggi che rappresenta il verdetto della classificazione della rete per l'immagine in input appena processata.

Un'architettura di base per una rete neurale convolutiva contiene almeno i moduli appena descritti, in una certa quantità. Il numero di moduli rappresenta un iperparametro della rete, che può inizialmente essere arbitrario, e successivamente raffinato grazie ai feedback ricevuti in fase di validazione dell'apprendimento automatico. I moduli componenti la rete possono essere collegati in maniera sequenziale, ovvero il volume di output di un modulo a monte risulta essere l'input del modulo presente a valle. Alternativamente, possono essere connessi in maniera parallela, con o senza output concatenato: due o più moduli ricevono in input lo stesso volume di dati, che processano indipendentemente. In caso sia prevista la concatenazione degli output, questi risultano in un nuovo volume d'uscita in cui gli output dei moduli sono concatenati secondo una delle tre dimensioni. All'interno della rete, i vari moduli possono essere connessi tra loro anche in diverse modalità,

pertanto è possibile parlare di *container*: all'interno di questi, i moduli sono collegati secondo la tipologia del container (sequenziale o parallelo, con o senza concatenazione), ed in una rete possono essere presenti anche più container. È possibile vedere la pipeline di una ConvNet come un container sequenziale all'interno del quale possono essere istanziati altri moduli o container tra loro connessi sequenzialmente. Nel caso più semplice, i moduli sono collegati tutti sequenzialmente, e possono essere organizzati come mostrato di seguito [10]:

```
INPUT -> [[CONV -> ACTIV]*N -> POOL?]*M -> [FC ->ACTIV]*K -> FC
```

In questo caso, la rete contiene una o più ripetizioni di moduli convolutivi, ognuno collegato a valle con un modulo di attivazione. In genere, $N \leq 3$. L'output di tale serie è fornito in input a un modulo di pooling per il sottocampionamento. Tale sequenza di moduli può essere iterata nella rete per M volte, con $M \geq 0$. In seguito, quanto elaborato finora, è dato in input ad una serie di K moduli fully-connected con a valle un modulo di attivazione (con $K \geq 0$), ed infine un livello totalmente connesso per il calcolo dei punteggi finali per ogni classe. Il "?" di fianco al modulo di pooling indica che questo è opzionale. Questo perché è possibile ridurre larghezza ed altezza dei volumi intermedi utilizzando solo moduli convolutivi [11], inoltre si suppone che un utilizzo intensivo del sottocampionamento possa danneggiare l'apprendimento automatico della rete. I moduli fully-connected, invece, possono essere sostituiti anche da moduli convolutivi che eseguono una convoluzione 1x1 [10] poiché applicano al proprio input la stessa e identica operazione. Come sarà illustrato nel seguito della presente tesi, ciò rappresenta un vantaggio poiché intuitivamente è possibile ridurre, e quindi ottimizzare, la varietà di moduli con cui comporre una rete neurale convolutiva.

2.2 Torch: un framework per il calcolo scientifico

Torch [7] è uno strumento *Open-source* scritto principalmente in Lua, un linguaggio di alto livello molto dinamico e flessibile, che fa dell'introspezione uno dei suoi punti di forza, e pertanto conferisce a Torch le stesse caratteristiche. Per introspezione, s'intende la capacità di determinare il tipo, e più in generale di ottenere informazioni, riguardo gli oggetti, in fase di esecuzione. Nonostante ciò, il framework si appoggia su una solida base scritta in C. Nasce con supporto per sistemi operativi con kernel Linux (quindi anche per MacOS), ed è utilizzabile su Windows solo dopo averne compilato i sorgenti in tale sede con procedimenti di cross-compilation. Il framework si presenta all'utente come una *shell* interattiva con cui è possibile comunicare mediante comandi Lua grazie al supporto di LuaJIT, il compilatore *Just-In-Time* di Lua il quale consente di sfruttare tale linguaggio come linguaggio interpretato/di scripting. Tra i suoi punti di forza vi è sicuramente la modularità, poiché è possibile importare i vari *package* messi a disposizione (collezioni di script tematici) nel momento in cui se ne presenta l'utilità. È inoltre possibile creare e utilizzare anche i propri package aggiungendone il path all'opportuna variabile d'ambiente di LuaJIT.

Come accennato in precedenza, si tratta di un framework messo a punto per la computazione scientifica, con una particolare predisposizione per gli algoritmi di apprendimento automatico. Ciò lo rende ideale per lavorare con reti neurali convolutive e non, fornendo un'API semplice e intuitiva tramite il package "nn" (neural networks) nativo di Torch, sia per effettuarne il training che l'esecuzione vera e propria. A rafforzare ciò, vi è anche un buon supporto per l'elaborazione di tali algoritmi su GPU grazie all'integrazione dell'API CUDA. Tuttavia, tale supporto non è stato impiegato nell'ambito della presente tesi poiché non conforme con gli obiettivi descritti nel paragrafo 1.2.

Il package "nn" consente la costruzione, la modifica, il training e l'esecuzione vera e propria di reti neurali con la giusta semplicità. In tale contesto, la rete neurale può essere modellata come un contenitore di moduli selezionabili tra una vasta gamma, a partire dai moduli di base descritti nel capitolo precedente, passando per le loro varianti, sino ad arrivare a moduli più specifici. La semplicità di costruzione risiede

nel fatto che è possibile aggiungere moduli a un container con una sola istruzione, ed è possibile fare altrettanto per rimuoverli o sostituirli. Il container è interpretato come un modulo che può contenere altri moduli, pertanto può inglobare a sua volta altri container, rendendo possibile la strutturazione di reti complesse ma al contempo flessibili.

La semplicità di composizione di reti neurali messa a disposizione da Torch è stata, tra le altre, una fonte di ispirazione per il progetto che sarà trattato nei capitoli successivi.

2.3Libreria Mini-XML

Mini-XML [12] è una delle poche librerie *Open-source*, sviluppata in linguaggio C con lo scopo di creare e gestire alberi di nodi XML, fornendo un'essenziale interfaccia di programmazione, e consentendone la lettura e scrittura da e verso file. Nel contesto di questo progetto, tale libreria è stata utilizzata per lo sviluppo del *parser*, il quale legge la configurazione del sistema proprio da file XML, e per convertire la descrizione della rete neurale dal formato nativo di Torch in formato XML. Tale dettaglio sarà argomentato nel capitolo successivo della presente tesi.

Per questioni di praticità, i sorgenti dell'intera libreria sono stati compilati con il codice sorgente del progetto sviluppato, nel pieno rispetto della licenza GNU General Public License v2.

CAPITOLO 3:

IMPLEMENTAZIONE DEL SISTEMA

3.1 Descrizione del modello in XML

Al fine d'implementare una rete neurale convolutiva riconfigurabile, si è deciso di usare proprio questa caratteristica come punto di partenza, ovvero la capacità di poter essere riconfigurata.

Per rete neurale convolutiva riconfigurabile, s'intende un sistema come quello descritto finora nel capitolo precedente i cui moduli e loro operazioni possono essere ridefiniti attraverso la modellazione dei loro parametri (ad es: *weight* e *bias* per il modulo di convoluzione) e dei loro iperparametri. Questi configurano, ad esempio, le dimensioni dei volumi di output e le modalità con cui il modulo applica la propria operazione sul volume di input (ad es: lo *stride* con cui il kernel è traslato sul volume di input). Per molti sistemi di questo tipo già affermati, la configurazione avviene per mezzo di un file da cui è possibile leggere e acquisire i valori caratterizzanti parametri e iperparametri, che quindi ne formano una descrizione sufficientemente dettagliata. In particolare, per lo scopo di questa tesi, si suppone che la descrizione della rete contenuta nel file di configurazione sia costituita da valori frutto di un training già avvenuto. Attualmente, non esiste ancora uno standard definito per i file di configurazione di reti neurali, ed ogni framework già esistente adotta un proprio formato per quanto riguarda questi file. Pertanto, uno degli obiettivi di questo progetto è quello di configurare una CNN nel modo più generale e standard possibile.

A tale scopo, si è optato per una rappresentazione della rete in formato XML, il quale è risultato essere un linguaggio abbastanza incline alla descrizione di sistemi organizzati allo stesso modo delle reti neurali. Infatti, è possibile rappresentare un modulo come nodo di un albero XML, il quale ha come attributi i propri iperparametri caratteristici. In caso il modulo sia un container, allora il suo nodo

corrispondente avrà tanti nodi figli quanti sono i moduli in esso contenuti. Nel caso dei moduli che operano la convoluzione, si è deciso di trattare pesi e bias come nodi figli che hanno come attributo il numero di parametri contenuti. I loro valori veri e propri sono inclusi come testo di tali nodi. Per rendere al meglio l'idea, si presenta di seguito uno stralcio tratto da un file XML di configurazione, che è stato prodotto ed utilizzato in fase di test dell'intero sistema:

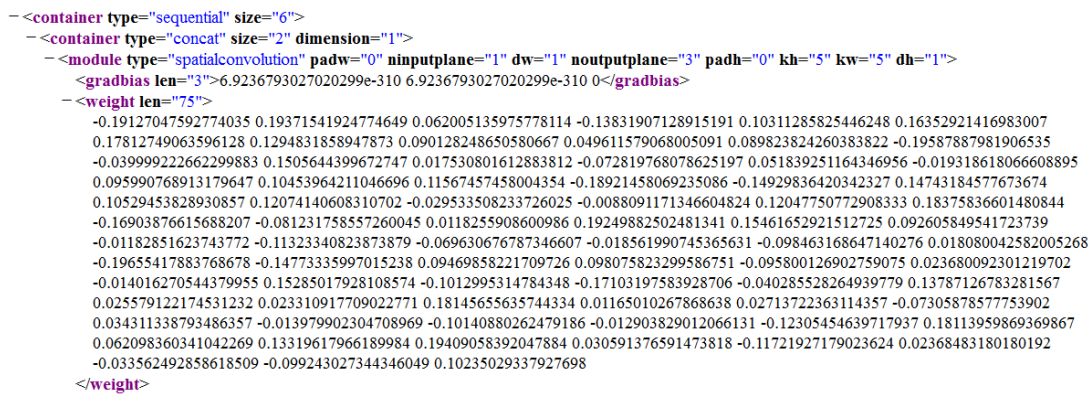


Figura 3.1: Stralcio di una descrizione di CNN da un file di configurazione in XML

Tutti i file di configurazione XML che sono stati utilizzati nell'ambito di questo lavoro, sono stati generati a partire da modelli di CNN tipici di Torch mediante del software scritto ad-hoc per ottenere tale risultato.

In particolare, è stato inizialmente sviluppato uno script in Lua chiamato `custom_persistence` da eseguire in Torch, il quale a partire da una rete già assemblata ne scrive la descrizione in un file temporaneo umanamente leggibile ed interpretabile. A questo scopo, è tornato molto utile il meccanismo di introspezione intrinseco di Lua. In seguito, è stato sviluppato un piccolo *tool* in C chiamato `cnn_xml_gen` che, grazie all'API messa a disposizione da Mini-XML e a partire dal file generato dallo script appena introdotto, crea un albero XML e lo scrive su un file dal contenuto analogo a quello in figura 3.1. Nelle successive versioni, questi due strumenti sono stati uniti in una sorta di *toolchain*, grazie al fatto che in Lua è presente il package "os" che fornisce il comando "execute" tramite cui è possibile effettuare una chiamata di sistema. In questo modo, a partire dallo script `custom_persistence`, una volta generato il file temporaneo è stato possibile richiamare l'eseguibile `cnn_xml_gen` in modo automatico dallo script stesso,

ottenendo così la generazione della descrizione XML tramite un solo comando eseguito da Torch.

Una rappresentazione del genere è risultata comoda anche per confronti e verifiche manuali grazie al fatto che è *human-readable* e inoltre, è stato riscontrato che il file di configurazione contenente una rete descritta in XML ha la stessa e identica occupazione in memoria di un file di configurazione per Torch in cui è descritta la stessa rete.

3.2 Rappresentazione interna dei dati

Le reti neurali convolutive, come già illustrato nei capitoli precedenti, lavorano su valori d'intensità di pixel organizzati secondo la struttura di un volume. Questo perché le immagini sono caratterizzate da un numero di canali, che possono essere considerati come piani sovrapposti che hanno la stessa larghezza e altezza (ad esempio, i 3 canali nel caso di immagini RGB). In tale ambito, i valori delle intensità dei singoli pixel sono rappresentati da un valore intero compreso tra 0 e 255 per ogni canale. Prima di essere forniti in input alla rete, questi valori vengono preprocessati passando per diverse fasi: inizialmente, se ne calcola la media totale che viene poi sottratta dal valore di ogni pixel, per ridurre l'entità di eventuali distorsioni già presenti nell'immagine che possano disturbare la classificazione, e successivamente si procede alla normalizzazione di tali valori. Dopo ciò, i valori delle intensità dei pixel saranno trasformati in numeri decimali in virgola mobile, pertanto l'efficacia del sistema nel classificare tali valori dipenderà sicuramente dalla precisione adottata per rappresentarli nel sistema.

A seguito di queste considerazioni, e sulla scia di Torch, si è deciso di organizzare i dati come tensori, ovvero strutture algebriche che estendono il concetto di vettore da una a più dimensioni, contenenti valori di tipo `float` o `double`. Uno dei problemi di questa rappresentazione è che generalmente questi tipi sono rappresentati rispettivamente con 32 e 64 bit a livello di hardware, e ciò significa che grandi tensori avranno bisogno di molta memoria per essere elaborati. Oltre a ciò, il fatto

che il tensore sia una struttura multidimensionale, rende meno banale la modalità d'accesso al proprio contenuto. Per rendere il sistema configurabile a partire già dal tipo di dati utilizzati, si è deciso di utilizzare un tipo definibile a *compile-time*:

```
typedef double tensor_data_t;
// oppure typedef float tensor_data_t;

typedef struct
{
      int d;
      int w;
      int h;
      tensor_data_t *data;
} tensor;
```

Come è possibile notare dall'estratto di codice, il contenuto del tensore è costituito da un puntatore ad un'area sequenziale di memoria che può essere di tipo `float` o `double`, la quale è gestita in modo tridimensionale con i tre interi (`d`, `w`, `h`) che ne indicano l'estensione in termini di volume. A un livello di astrazione leggermente più elevato, i dati presenti nel tensore saranno interrogati mediante tre indici, uno per la profondità, uno per la larghezza e l'altro per l'altezza, partendo dall'origine del volume in alto a sinistra del canale più superficiale, per poi essere scorsa dall'alto verso il basso e sempre più in profondità. A un livello più basso, questi tre indici sono trasformati in un unico indice con cui si andrà a scorrere effettivamente i valori nell'area puntata da `*data`.

Pertanto, alla mole inaudita di operazioni da eseguire ad ogni livello della rete si va ad aggiungere il calcolo per la trasformazione degli indici tridimensionali in un solo indice.

3.3 Sviluppo del motore di convoluzione

Una volta pensato a una descrizione formale della rete neurale che fosse il più generale possibile e ad una rappresentazione interna dei dati, si è passato al successivo *step*, ovvero lo sviluppo del cuore del sistema: il codice che implementa l'operazione di convoluzione, ampiamente argomentata nei capitoli precedenti.

Gli operandi per la convoluzione saranno sicuramente il tensore di input, la collezione di filtri/kernel con cui convolverlo, il vettore contenente i valori di bias, ed

infine gli iperparametri che andranno a configurare l'applicazione dell'operazione, quali l'ammontare di padding orizzontale e verticale, ed il passo orizzontale e verticale di applicazione dei filtri. Si noti che anche i kernel sono modellati come tensori tridimensionali.

Dato il carattere dell'operazione finora descritto, e la rappresentazione interna dei dati appena introdotta, la versione più banale del motore di convoluzione che può venire in mente ha la forma di una serie di cicli `for` innestati. Nell'ultimo livello, quello più interno, si effettua la trasformazione degli indici tridimensionali in monodimensionali in modo da indicizzare adeguatamente sia i dati del tensore di input che i dati del kernel correntemente applicato, ed infine viene elaborata la convoluzione elemento per elemento. Tutto ciò risulta nel seguente estratto di codice:

```
1.  tensor convolve_tensors(tensor input, tensor *filter_banks, tensor *bias, int n_filters,
    int pad_h, int pad_w, int stride_h, int stride_w)
2.  {
3.      tensor out_vol;
4.
5.      out_vol.w = (input.w - filter_banks[0].w + 2*pad_w)/stride_w + 1;
6.      out_vol.h = (input.h - filter_banks[0].h + 2*pad_h)/stride_h + 1;
7.      out_vol.d = n_filters;
8.      out_vol.data = calloc(out_vol.w*out_vol.h*out_vol.d, sizeof(tensor_data_t));
9.
10.     if(pad_h && pad_w) input = pad_tensor(input, pad_h, pad_w);
11.
12.     int d, h_ker, w_ker;
13.     int fil, h_out, w_out;
14.
15.     for(fil = 0; fil < n_filters; fil++)
16.     {
17.         tensor curr_filter = filter_banks[fil];
18.
19.         for(h_out = 0; h_out < out_vol.h; h_out++)
20.         {
21.             for(w_out = 0; w_out < out_vol.w; w_out++)
22.             {
23.                 int out_i = fil*out_vol.h*out_vol.w + h_out*out_vol.w + w_out;
24.
25.                 for(d = 0; d < curr_filter.d; d++)
26.                 {
27.                     for(h_ker = 0; h_ker < curr_filter.h; h_ker++)
28.                     {
29.                         for(w_ker = 0; w_ker < curr_filter.w; w_ker++)
30.                         {
31.                             int in_i = d*input.h*input.w + (h_out*stride_h + h_ker)*input
    .w + w_out*stride_w + w_ker;
32.                             int ker_i = d*curr_filter.w*curr_filter.h + h_ker*curr_filter
    .w + w_ker;
33.
34.                             out_vol.data[out_i] += input.data[in_i] * curr_filter.data[ke
    r_i];
35.                         }
36.                     }
37.                 }
38.
39.                 out_vol.data[out_i] += bias != NULL ? (*bias).data[fil] : 0;
40.             }
41.         }
42.     }
43.
44.     return out_vol;
45. }
```

Com'è possibile costatare, la trasformazione degli indici da quelli tridimensionali a quelli monodimensionali (righe 31 e 32) è frutto di una funzione paradossalmente più complessa di quella per il calcolo della score function (riga 34).

Se si considera che tali operazioni sono svolte all'interno di 6 cicli `for` innestati, è possibile immaginare quale possa essere il carico computazionale di queste per input di dimensioni reali. Pertanto si potrebbe pensare di ottimizzare questa operazione, soprattutto in vista del *porting* su FPGA, mediante tecniche di vettorizzazione dei cicli per velocizzare i calcoli. Tali dettagli saranno affrontati più da vicino nel successivo capitolo. Comunque, a seguito di quanto assunto finora e dopo i primi test del sistema completo, si è pensato di partire dall'ottimizzazione a livello di codice. Ciò ha portato a una nuova versione del motore di convoluzione possibilmente ottimizzata, qui di seguito introdotta.

3.3.1 Possibile ottimizzazione

L'ottimizzazione del codice del motore di convoluzione è avvenuta sulla base di alcune considerazioni fatte a partire dal codice mostrato in precedenza.

Il calcolo degli indici monodimensionali per input e kernel dipende dagli indici delle tre dimensioni del volume di output, che fungono da *offset* combinati alla misura del passo. Proprio per questo è possibile calcolare gli offset, e quindi gli indici, in maniera incrementale non appena gli indici aggiornati per la formula sono disponibili, distribuendo così i calcoli nei vari strati dei `for` innestati. Ciò significa che a ogni iterazione possono esser valutate espressioni molto meno complesse di quelle alle righe 31 e 32 del codice mostrato.

Inoltre, per il fatto che è possibile calcolare gli offset in maniera incrementale non c'è più bisogno che questi dipendano dagli indici delle dimensioni del volume di output,di cui sarebbe possibile indicizzare il contenuto indipendentemente e in maniera monodimensionale.

Il codice risultante da tali considerazioni è riportato di seguito:

```
1.  static void convolve_patches(tensor_data_t *out, tensor filter, tensor input, int depth,
    int h_offset, int w_offset)
2.  {
3.      int fil_depth_offset = depth*filter.w*filter.h;
4.      int in_depth_offset = depth*input.w*input.h;
5.
6.      int input_offset = in_depth_offset;
7.      int filter_offset = fil_depth_offset;
8.
9.      tensor_data_t *in_data_ptr = input.data;
10.     tensor_data_t *fil_data_ptr = filter.data;
11.
12.     int h_i, w_i;
13.     for(h_i = 0; h_i < filter.h; h_i++)
14.     {
15.         input_offset += (h_offset + h_i)*input.w;
16.         filter_offset += h_i*filter.w;
17.
18.         for(w_i = 0; w_i < filter.w; w_i++)
19.             *out += in_data_ptr[input_offset + w_offset + w_i]*fil_data_ptr[filter_offset
     + w_i];
20.
21.         input_offset = in_depth_offset;
22.         filter_offset = fil_depth_offset;
23.     }
24. }
25.
26. tensor convolve_tensors(tensor input, tensor *filters, tensor *bias, int n_filters, int p
    ad_h, int pad_w, int stride_h, int stride_w)
27. {
28.     tensor out_vol;
29.
30.     out_vol.w = (input.w - filters[0].w + 2*pad_w)/stride_w + 1;
31.     out_vol.h = (input.h - filters[0].h + 2*pad_h)/stride_h + 1;
32.     out_vol.d = n_filters;
33.     out_vol.data = calloc(out_vol.w*out_vol.h*out_vol.d, sizeof(tensor_data_t));
34.
35.     if(pad_h && pad_w) input = pad_tensor(input, pad_h, pad_w);
36.
37.     tensor_data_t *out_data_ptr = out_vol.data;
38.
39.     int out_len = out_vol.d*out_vol.w*out_vol.h;
40.     int out_slice_len = out_len/out_vol.d;
41.
42.     int h_offset = 0;
43.     int w_offset = 0;
44.     int depth;
45.
46.     int filter_i = 0;
47.
48.     int out_i;
49.     for(out_i = 0; out_i < out_len; out_i++)
50.     {
51.         for(depth = 0; depth < input.d; depth++)
52.             convolve_patches(&out_data_ptr[out_i], filters[filter_i], input, depth, h_off
    set, w_offset);
53.
54.         if(bias)
55.             out_data_ptr[out_i] += (*bias).data[filter_i];
56.
57.         if(w_offset + stride_w + filters[filter_i].w <= input.w)
58.             w_offset += stride_w;
59.         else
60.         {
61.             w_offset = 0;
62.
63.             if(h_offset + stride_h + filters[filter_i].h <= input.h)
64.                 h_offset += stride_h;
65.             else
66.             {
67.                 h_offset = 0;
68.                 filter_i++;
69.             }
70.         }
71.     }
72.
73.     return out_vol;
74. }
```

Com'è possibile notare, il numero complessivo di cicli `for` innestati è stato ridotto da 6 a 4 grazie al fatto che gli offset sono calcolati in maniera incrementale da un sistema di `if/else` innestati a due livelli. I risultati dei test e i confronti a livello di prestazioni con la precedente versione saranno trattati nel capitolo successivo.

3.4 Struttura del codice

Lo sviluppo del motore di convoluzione e la decisione della rappresentazione interna dei dati sono stati i primi passi per la costruzione dell'intero sistema, del quale è possibile trovare la rappresentazione grafica UML e l'intera documentazione in Appendice. In seguito, si è proceduto alla strutturazione di un software che fosse il più possibile modulare e di facile estensibilità. Tale strutturazione è stata realizzata come riportato di seguito:

- `tensor.c`/`tensor.h`: a partire dal motore di convoluzione, è stata sviluppata questa *core library* che mette a disposizione diverse funzioni per gestire i tensori ed effettuare operazioni tra di essi. In particolare, vi sono funzioni per l'inizializzazione, per l'eliminazione e per la messa a video dei tensori, o ancora funzioni per compiere l'operazione di *max pooling* e la concatenazione di tensori.

- `activation_functions.c`/`activation_functions.h`: anche in questa libreria sono state implementate delle *core functions*, in particolare le funzioni di attivazione o di trasferimento di cui si è parlato nel Paragrafo 2.1.2.1. Dato che queste sono applicate elemento per elemento al tensore di input, questa libreria include `tensor.h`.

  Tra le funzioni implementate è possibile trovare le più importanti, quali la tangente iperbolica, la sigmoide, la *rectified linear unit* ed un'altra serie di funzioni, sempre di attivazione o di utilità.

- `module.c/module.h`: In questa libreria è contenuta l'astrazione di modulo, ovvero l'elemento base di una rete neurale, il quale è caratterizzato dalla seguente struttura dati tipizzata:

```
1.  typedef struct module
2.  {   // BASICS
3.      module_t type;
4.
5.      tensor *input;
6.      tensor output;
7.
8.      // CONVOLUTIONAL AND POOL
9.      int n_filters;
10.     tensor *filters;
11.     tensor bias;
12.
13.     int pad_h, pad_w;
14.     int stride_h, stride_w;
15.
16.     int ker_h, ker_w;
17.
18.     // ACTIVATION
19.     activation_t act_type;
20.
21.     // CONTAINER
22.     container_t cont_type;
23.     int concat_dim;
24.     int n_modules;
25.     struct module *modules;
26. } module;
```

  Com'è possibile notare dal codice riportato, l'astrazione di modulo risulta essere una struttura dati che contiene i campi rappresentanti le diverse caratteristiche che i vari tipi di modulo possono avere. Inoltre, è evidente anche in questo caso la dipendenza da `tensor.h`. Un modulo possiede sempre due campi rappresentanti il volume di input (`tensor *input`) e quello di output (`tensor output`) ed è inoltre caratterizzato da un tipo, `module_t type` implementato come enumeratore, che stabilisce quali degli altri campi nella struttura abbia senso utilizzare o meno. Ad esempio, un modulo di tipo convolutivo prende in considerazione solo i campi `n_filters`, `ker_h`, `ker_w`, `pad_h`, `pad_w`, `stride_h`, `stride_w`, la collezione di filtri `*filters` in cui saranno contenuti e organizzati tutti i pesi gestita come *array* di tensori, e infine un tensore gestito sempre come vettore monodimensionale di `bias`. Alcuni moduli, come ad esempio quello di attivazione o il container in questo sistema, possono avere dei sottotipi che indicano precisamente quale sia la sottocategoria di modulo che si

intende utilizzare, sempre allo scopo di dare un senso ad alcuni campi piuttosto che ad altri. Anche tali sottotipi sono stati implementati come enumerativi. Da notare anche il campo ricorsivo `struct module *modules` che rappresenta il puntatore ai moduli istanziati in un container. Pertanto con la stessa struttura dati è possibile descrivere l'intero sistema nel suo stato attuale. Questa libreria fornisce, ovviamente, una serie di funzioni per l'inizializzazione, la deallocazione della memoria e la messa a video del modulo. Da notare la funzione `forward`, utile per dar via al processing dell'input una volta che lo si è impostato con l'apposita funzione *setter*, e la funzione `add_module`, la quale ha effetto solo se il modulo passato come primo argomento è un container ed in caso ne aggiunge il modulo il cui puntatore è passato come secondo argomento.

In caso di estensione dovuta all'aggiunta di altri tipi di modulo, bisognerebbe semplicemente aggiungere i campi opportuni che lo caratterizzano alla struttura dati finora discussa, ed anche eventuali funzioni specifiche in caso il tipo aggiunto ne richieda. All'inizializzazione, qualora vi siano campi della struttura non presi in considerazione, questi sono pari a `0` o a `NULL`.

- `*_module.c`/`*_module.h`: l'asterisco implica un nome di modulo qualunque, pertanto questi file contengono l'implementazione per ogni modulo specifico. Attualmente, il sistema può avvalersi di componenti quali il modulo di convoluzione, il modulo di pooling, il modulo di attivazione, e due tipi di container, entrambi gestiti nello stesso modulo software. In particolare, in questi file risiede il codice delle funzioni utili per l'inizializzazione, per l'elaborazione dell'input e per la messa a video, le quali per ogni diverso tipo di modulo, hanno un'implementazione diversa. Nel caso delle funzioni di inizializzazione queste differiscono molto spesso anche per la *signature*, poiché è facile intuire che per ogni modulo vi sia una lista di argomenti diversi, e di diversa lunghezza. Si noti che si tratta delle stesse funzioni realizzate in

`module.c` il quale importa questi componenti software per smistare correttamente le chiamate a queste funzioni verso i giusti moduli. Per approfondimenti a riguardo, si rimanda all'Appendice.

Nel caso si voglia estendere un modulo aggiungendo dei sottotipi, è necessario realizzare un enumerativo che lo rappresenti, aggiungere gli opportuni campi al tipo `module` e le relative funzioni per gestirlo nell'apposita libreria che va a specializzare. Finora, questo procedimento è stato messo in atto solo per il `container_module`, il quale costituisce un buon esempio di estensione.

- `net_parser.c`/`net_parser.h`: grazie a questo componente software la rete è in grado di configurarsi automaticamente a partire da un file che ne contiene la descrizione XML, come quello illustrato nel paragrafo 3.1. Per fare ciò, il `net_parser` si appoggia alla libreria esterna Mini-XML introdotta nel paragrafo 2.3. Il *parsing* del file avviene in modo conforme alla struttura dell'albero XML poiché questo presenterà prima un container e poi i moduli in esso inclusi, e viene effettuato mediante la chiamata ad una sola funzione, `parse_network` che a partire dal nome del file che contiene la configurazione restituisce una variabile di tipo `module` che contiene l'intera rete. Inoltre, questa libreria mette a disposizione anche una funzione per configurare tensori a partire da file, molto utile quindi anche per l'inizializzazione del tensore che sarà fornito in input alla rete.

  In caso di aggiunta di nuovi tipi di modulo, è necessario aggiornare anche il *parser* per far sì che di questo ne sia riconosciuta la descrizione in fase di configurazione.

- `20c-net.c`: si tratta del file contenente la funzione *main*, da cui parte tutto il procedimento di inizializzazione ed esecuzione della rete.

Come ultimo passo per la realizzazione del sistema appena illustrato, tutti i tipi realizzati e le costanti utili sono stati raccolti in un solo *header* chiamato `net_types.h` per semplificare ulteriormente la struttura del software al fine di

facilitare eventuali e future estensioni. Infatti, in caso di aggiunta di un altro tipo di modulo, o sottocategoria di un tipo di modulo, è possibile aggiornare o inserire l'opportuno enumerativo o struttura tipizzata in questo header. Ciò è utile, inoltre, ad evitare che vengano a crearsi dipendenze inutili (*"spaghetti programming"*) tra i vari componenti del software, le quali potrebbero renderne la manutenzione ingestibile.

CAPITOLO 4:

TEST E RISULTATI SPERIMENTALI

4.1 Premesse

I test per valutare il funzionamento e le prestazioni del sistema oggetto di questa tesi sono svolti su un notebook con processore Intel T2370 dual-core, a 64 bit, da 1.73 GHz, equipaggiato con 2GB di memoria RAM. L'esecuzione avviene in maniera totalmente sequenziale, impiegando un solo core.

I test sono svolti facendo processare dei dati di input a una rete in Torch, il cui output è salvato su un file. Per far sì che i test avvengano con gli stessi dati in input, anche tali tensori sono scritti su un file. In seguito, a partire da Torch, la stessa rete è convertita in formato XML, mediante la toolchain illustrata nel paragrafo 3.1, per configurare il sistema in esame. In seguito,è fornito in ingresso al sistema appena configurato lo stesso input in precedenza processato tramite Torch, e l'output derivante è a sua volta scritto su un altro file. Infine, è eseguito il confronto tra i contenuti dei due file di output elemento per elemento. Tale confronto è effettuato con un piccolo tool creato a parte (`test_net.c`).

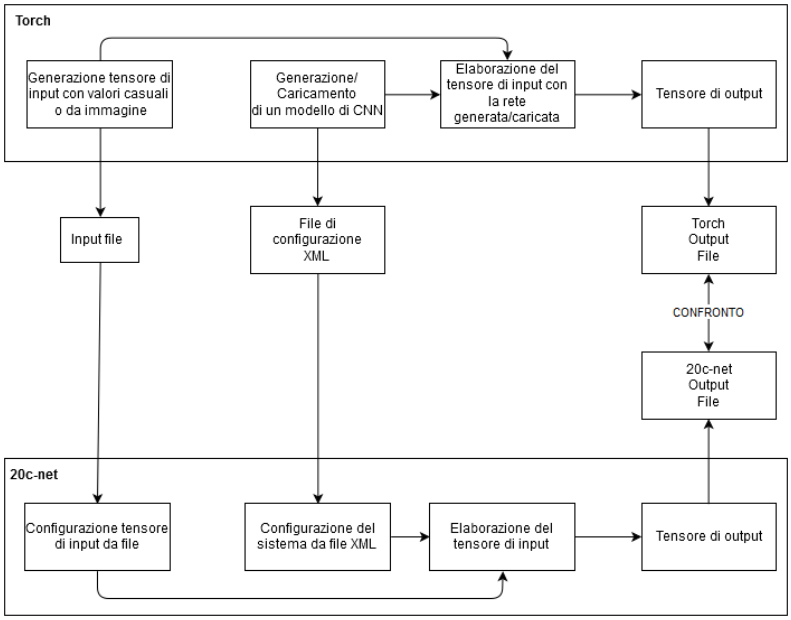


Figura 4.1: Rappresentazione grafica dello svolgimento dei test

4.2 Test del sistema con dati di input sintetici

I primi test, soprattutto quelli svolti in fase di sviluppo del sistema per verificare il funzionamento delle core functions, sono stati operati utilizzando tensori di input sintetici, inizializzati in maniera pseudo-casuale. Infatti, la funzione `init_tensor`, per la cui descrizione dettagliata si rimanda alla documentazione in Appendice, è configurabile mediante un argomento che se uguale ad 1 fa sì che l'inizializzazione avvenga con numeri generati mediante la funzione `rand()` nativa di C. In particolare, i numeri generati da questa funzione sono normalizzati e quindi convertiti in numeri decimali di tipo `tensor_data_t` compresi tra 1 e -1 per simulare un volume di dati reale.

Successivamente, una volta verificato il funzionamento delle operazioni base del sistema, ne è stata sviluppata una prima architettura a moduli, avendo così una prima versione interamente testabile. Per valutarne il funzionamento, è stata composta una rete di esempio in Torch con la seguente struttura, pensata ad-hoc per testare anche il caso in cui la rete contenga un altro container:

- Container sequenziale
    - Container parallelo con concatenazione dell'output
        - Modulo convolutivo
        - Modulo convolutivo
    - Modulo di attivazione ReLU
    - Modulo di max pooling
    - Modulo convolutivo
    - Modulo di attivazione Tanh
    - Modulo convolutivo

Tale rete prende in ingresso tensori di dimensioni 1x32x32 e restituisce in uscita un vettore di 10 punteggi arrangiato secondo la profondità. Quindi, sempre da Torch è stato creato, con valori casuali di tipo `double`, un tensore di input con le dimensioni sopra specificate, che è stato poi processato dalla rete. Al che, il sistema in oggetto è stato configurato secondo il modello di cui sopra e in maniera tale che operasse con dati di tipo `double`, a cui è stato in seguito fornito lo stesso tensore di input per

effettuarne l'elaborazione. Da ciò, si è manifestato in output lo stesso vettore di punteggi prodotto dalla medesima rete in Torch, indicando così un funzionamento corretto del sistema nella sua interezza.

4.3 Test del sistema con modello e dati reali

Una volta verificato il funzionamento strutturale e operativo, si è proceduto con ulteriori test per verificare l'impatto di un'elaborazione di dati reali effettuata configurando il sistema con un modello di CNN reale. A tal proposito, si è deciso di utilizzare un modello di CNN [3], preventivamente convertito per poter eseguire su CPU, sviluppato nell'ambito di un altro progetto di ricerca. In particolare, la suddetta rete neurale convolutiva apprende una misura di confidenza per un sistema di visione stereo a partire da mappe di disparità [3]. Queste misure sono utili allo scopo di validare l'accuratezza e la veridicità di tali mappe.

Il modello impiegato è composto da 14 moduli, strutturati come di seguito:

- Container Sequenziale
  - Modulo Convolutivo
  - Modulo di attivazione ReLU } x6
  - Modulo Convolutivo
  - Modulo di attivazione Sigmoid

Il primo test è stato eseguito elaborando una sola immagine proveniente dal dataset KITTI [13], con dimensioni 1x1226x370 che è stata convertita in un volume di `double` alla quale è stato applicato uno zero-padding pari a 4 sia orizzontalmente che verticalmente in modo da ottenere un volume di output delle stesse dimensioni. L'elaborazione è avvenuta inizialmente configurando il sistema per operare con dati dello stesso tipo di quelli di input, ma questa non è giunta al termine poiché 2GB di memoria RAM non sono stati sufficienti dato che, se l'input ha già di per sé dimensioni importanti, allora anche i risultati intermedi sono molto voluminosi. Pertanto, si è convertita l'immagine di input in un tensore di `float`, e configurato il sistema allo stesso modo. L'esecuzione è durata 25 minuti e 15 secondi ed ha riservato un massimo di 1.58 Gigabyte di memoria. Inoltre, il confronto degli output

prodotti ha messo in luce un errore medio complessivo di 0.0369, calcolato come somma mediata di differenze assolute:

$$avg_{error} = \frac{\sum_{i=1}^{N} | out_i - truth_i |}{N}$$

Formula 4.1: errore medio complessivo. N è il numero di elementi presenti nei vettori "out" e "truth" che contengono, rispettivamente, l'output prodotto dal sistema sviluppato e l'output calcolato dalla rete su Torch.

Questo errore è dovuto alla conversione in tensori di tipo `float` della mappa di disparità in input e dei parametri del modello. Ciò è stato verificato confrontando il tensore di input, che dal sistema è stato appositamente riscritto su file, con il file contenente il tensore di input originato da Torch. Si è scoperto che il framework scrive i `float` su file tenendo traccia delle cifre decimali fino all'undicesima. In C, non è stato possibile recuperarle tutte per ogni elemento dell'input in quanto non si possono utilizzare le stringhe di formato per specificare la quantità di cifre decimali da leggere, a differenza di come è possibile fare per la stampa a video, o su file. Mediante la stringa di formato `"%f"` è possibile avere numeri a precisione singola con al massimo sei cifre decimali. Provando a stampare con stringa di formato `"%.11f"` numeri di tipo `float` letti in precedenza da file con la stringa di formato di cui sopra, e provando a confrontarli con il contenuto di tale file, si è riscontrato che le ultime cinque cifre decimali sono diverse, confermando che l'errore fosse effettivamente dovuto all'approssimazione dei dati.

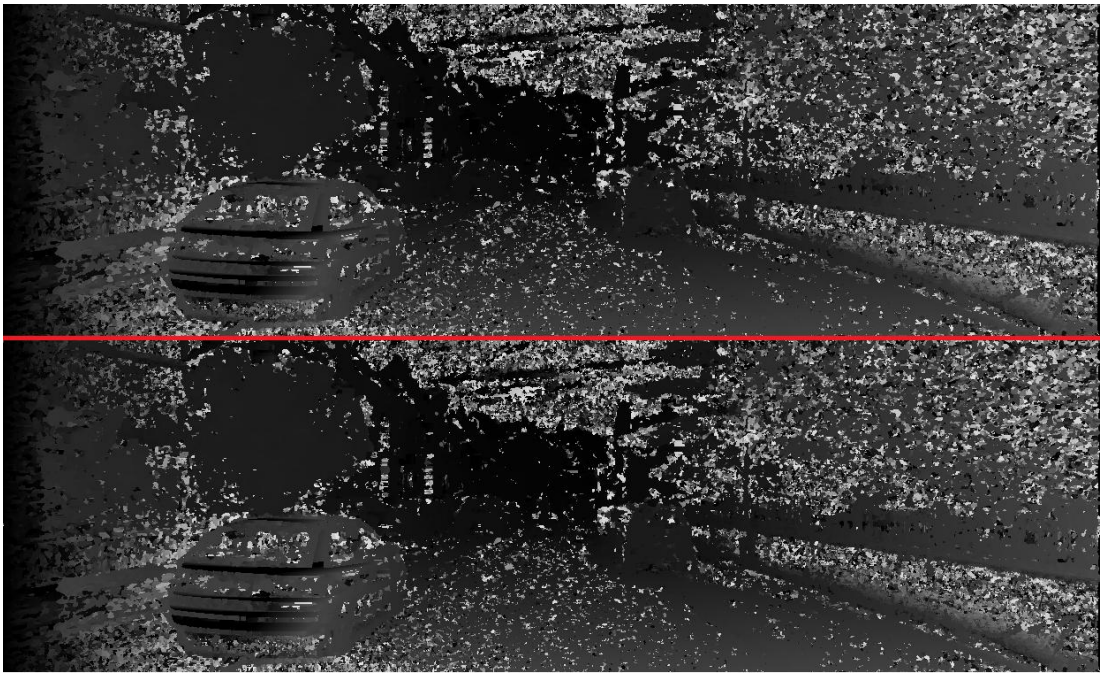


Figura 4.2: Confronto tra l'input originale (in alto, immagine 6 da KITTI) fornito per il test, e l'input del sistema (in basso) riscritto su file dopo il caricamento nel sistema e riconvertito in immagine tramite Torch. È possibile notare delle lievissime differenze, soprattutto guardandone i bordi. Aguzzando ulteriormente la vista si può notare anche una quasi impercettibile differenza nelle tonalità di grigio, nelle prossimità dell'automobile. Ciò è dovuto alle motivazioni appena illustrate.

Un altro dato rilevante è stato il tempo di esecuzione, giudicato molto elevato, da cui è scaturito il tentativo di ottimizzazione del motore di convoluzione illustrato nel paragrafo 3.3.1. La nuova versione del motore ha permesso un guadagno di soli 2 minuti sul tempo di esecuzione registrato in precedenza, dando luogo ad un'esecuzione durata 23 minuti. Tuttavia, un vero guadagno è stato registrato specificando il *flag* `-O3` [14] nella compilazione del software mediante `gcc`, grazie al quale è stato possibile ridurre il tempo di esecuzione fino a 5 minuti e 43 secondi. In particolare, il *flag* `-O3` di `gcc` indica al compilatore di sfruttare un meccanismo di *inlining* aggressivo del codice, ed algoritmi euristici per la vettorizzazione dei cicli, mantenendo la consistenza del software. È possibile dire che questo flag di ottimizzazione abbia appiattito la differenza tra i due motori di convoluzione sviluppati, o meglio, che abbia sensibilmente ottimizzato di più le prestazioni del primo (guadagnando 43 secondi) rispetto al secondo, poiché, in termini di codice, il primo ha una struttura più regolare, ad innesto di `for`.

A seguito di questo enorme guadagno sui tempi di esecuzione, si è deciso di effettuare ulteriori test sul sistema impiegando un *batch* di 130 immagini, estratte anch'esse dal dataset KITTI, le quali possono avere quattro diversi tagli in fatto di dimensioni. In particolare, vi sono immagini da (canali x larghezza x altezza):

- 1x1226x370
- 1x1238x374
- 1x1241x376
- 1x1242 x 375

La finalità di quest'ultimo test è stata quella di misurare le prestazioni del sistema secondo i parametri di seguito illustrati, con i rispettivi valori risultanti:

- Occupazione media in memoria dei volumi intermedi ……. **1.58 GB**
- Occupazione media in memoria del singolo volume intermedio ……. **115.80 MB**
- Occupazione in memoria del modello utilizzato ……... **512.95 KB**

- Tempo totale di esecuzione su 130 immagini …… **12 h 54 m**
- Tempo medio di elaborazione di una sola immagine ……... **5 m 58 s**

- Errore medio di approssimazione su 130 immagini …… **0.009956**

Si è inoltre scoperto che l'errore medio di approssimazione tende a scomparire quasi del tutto man mano che aumentano le immagini elaborate, come mostrato di seguito:

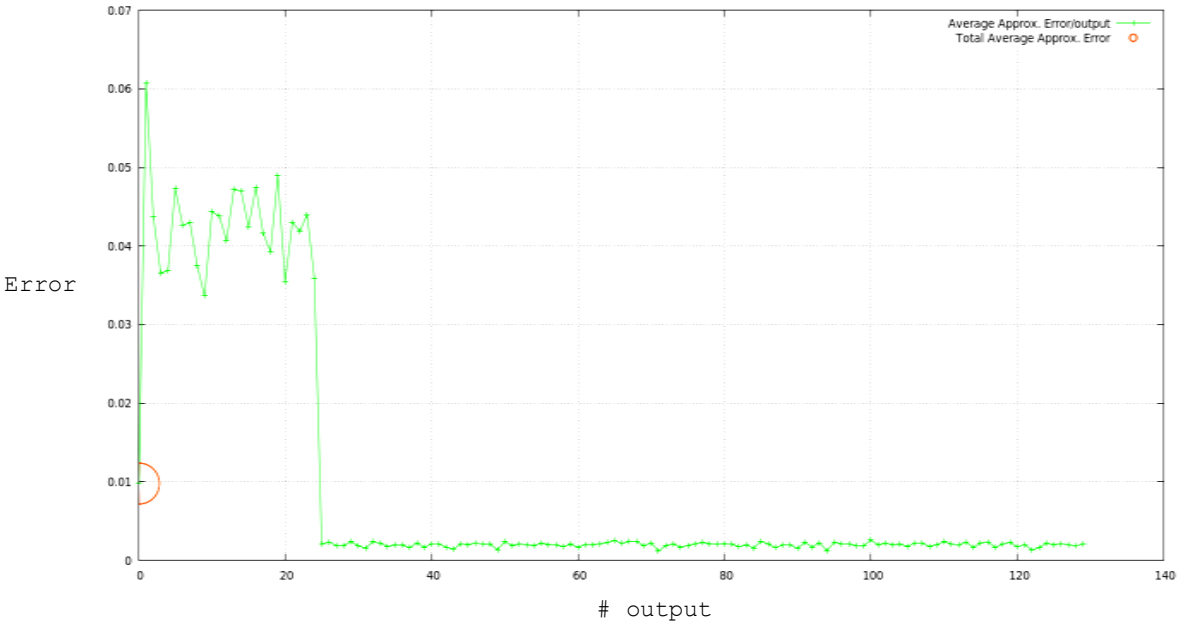


Figura 4.3: grafico rappresentante l'andamento dell'errore medio di approssimazione.

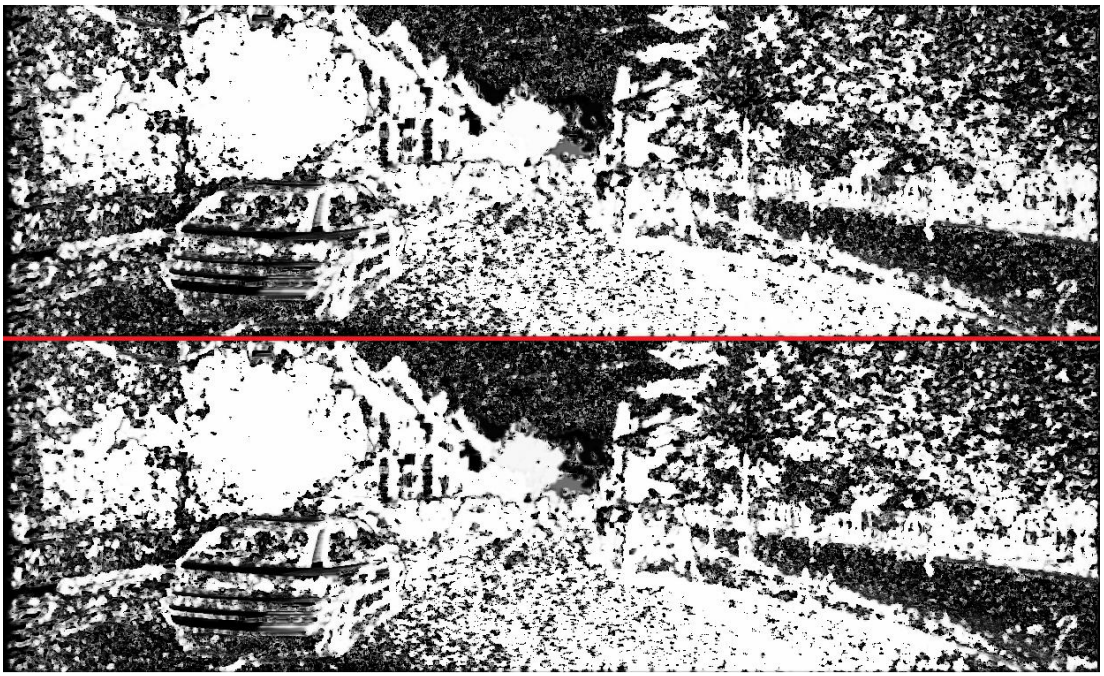


Figura 4.4: Confronto tra gli output prodotti dall'elaborazione dell'immagine 6 del dataset KITTI. In alto, l'output originale, in basso l'output generato dal sistema e convertito in immagine tramite Torch. Come è possibile notare, le differenze sono praticamente impercettibili. Inoltre, l'immagine 6 è stata una tra quelle che hanno generato l'errore medio più alto dovuto all'approssimazione dei dati in input durante il caricamento.

CAPITOLO 5:

CONCLUSIONI E SVILUPPI FUTURI

In conclusione, è possibile affermare che il sistema sviluppato nell'ambito della presente tesi può essere un buon candidato ai fini di un porting su architettura FPGA con lo scopo di accelerarne le operazioni, grazie al fatto che è stato sviluppato seguendo i principi della semplicità e della modularità. Inoltre, gli ultimi test hanno messo in luce che anche modelli di ConvNets impiegati per applicazioni reali e composti da un numero relativamente corposo di moduli convolutivi, quando vanno a configurare il sistema in oggetto necessitano di una quantità di memoria accettabile per quelli che sono i limiti di un sistema embedded (in questo caso, 512.95 KB).

Gli altri parametri inerenti alle prestazioni del sistema, quali il tempo di elaborazione, sia per la singola immagine sia per un intero batch di queste, e la memoria necessaria alla persistenza dei risultati intermedi, ribadiscono che a scapito delle possibili ottimizzazioni operabili a livello di codice, sistemi di questa natura non sono adatti per essere eseguiti su architetture che elaborano in maniera sequenziale, e che sono equipaggiati con una quantità di memoria RAM pari o inferiore a 2 GB. Pertanto, si propongono ottimizzazioni a livello di paradigma di programmazione, come ad esempio la parallelizzazione dell'esecuzione del codice includendovi OpenMP [15], un'API compatibile con C, C++ e appositamente sviluppata per questo scopo. Un'altra possibile ottimizzazione può riguardare l'utilizzo di tecniche per ridurre l'esigenza di memoria,come ad esempio il *tiling*, che potrebbe essere utile per risolvere il problema dovuto all'eccessivo volume dei dati intermedi. In particolare, questa tecnica, già impiegata per i sistemi embedded, consente di frammentare i dati da elaborare in diversi *tiles* (mattonelle) che vengono gradualmente caricati in RAM dal processore per far sì che l'elaborazione da parte del coprocessore richieda meno memoria.

Pertanto, le fasi successive alla realizzazione del presente progetto di tesi riguarderanno il porting di questo su architettura FPGA mediante modifica del codice con il tool Vivado HLS (High Level Synthesis) di Xilinx, il quale consente di trasformare istruzioni scritte in linguaggio C dapprima in RTL (Register Transfer

Language) e successivamente in *bitstream*, grazie al quale è possibile configurare direttamente l'FPGA. Saranno introdotte, inoltre, ottimizzazioni ad-hoc, come ad esempio l'utilizzo di un altro tipo di dati che possa rendere l'elaborazione meno onerosa in termini di carico computazionale. In particolare, è stato provato che i numeri decimali in virgola fissa, più precisamente in formato Q8.8 come riportato in [6], garantiscono risultati approssimati ma soddisfacenti in questo contesto. Infine, si provvederà allo sviluppo di un sistema di *scheduling* grazie al quale sarà possibile, ad esempio, configurare l'architettura facendo sì che questa esegua un modulo della rete alla volta. I volumi di dati per compiere l'elaborazione saranno reperiti di volta in volta dalla memoria DDR dove, una volta terminate le operazioni, sarà scritto il risultato che a sua volta sarà reperito in qualità input dal successivo layer di elaborazione.

BIBLIOGRAFIA

[1] M. Boschini, M. Poggi, S. Mattoccia, "Improving the reliability of a 3D people tracking leveraging on deep-learning", International Conference on 3D Imaging (IC3D), Liège, Belgium, 13-14 December 2016 [PDF]

[2] M. Poggi, S. Mattoccia, "Deep Stereo Fusion: combining multiple disparity hypotheses with deep-learning", 2016 International Conference on 3D Vision (**3DV 2016**), October 25-28, 2016, Stanford University, California, USA [PDF]

[3] M. Poggi, S. Mattoccia, "Learning from scratch a confidence measure", 27th British Machine Vision Conference (**BMVC 2016**), September 19-22, 2016, York, UK [PDF] [Code]

[4] M. Poggi, S. Mattoccia, "A wearable mobility aid for the visually Impaired based on embedded 3D vision and deep learning", First IEEE Workshop on ICT Solutions for eHealth (IEEE ICTS4eHealth 2016) in conjunction with the Twenty-First IEEE Symposium on Computers and Communications, June 27-30, 2016, Messina, Italy [PDF]

[5] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-Based Learning Applied to Document Recognition", November 1998 [PDF]

[6] V. Gokhale, J. Jin, A. Dundar, B. Martini, E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks", Electrical and Computer Engineering, and Weldon School of Biomedical Engineering, Purdue University [PDF]

[7] Torch website [URL]

[8] L. Di Stefano, slide del corso "Computer Vision and Image Processing M", 1. Introduction [PDF], 3.Intensity Transformations [PDF], 4.Spatial Filtering [PDF], 11. Object Detection [PDF] [URL della pagina del corso]

[9] M. Lippi, slide del corso "Machine Learning PhD", Lecture 1 (Basic concepts and Introduction to Artificial Neural Networks) [PDF], Lecture 2 (Deep Learning I: Energy Based Models, Auto-Encoders, Tricks of the Trade) [PDF]

[10] A. Karpathy, corso“CS231n Convolution Neural Networks for Visual Recognition”, Backpropagation notes [URL], Neural Nets notes 1 [URL], ConvNet notes [URL]

[11] J. T. Springenberg, A. Dosovitskiy, T. Brox, M. Riedmiller, “Striving for Simplicity: The All Convolutional Net”, 21 December 2014 [URL]

[12] Mini-XML website [URL]

[13] KITTI dataset website[URL]

[14] GNU GCC optimization flags comparison on documentation [URL]

[15] OpenMP website [URL]

APPENDICE:

DOCUMENTAZIONE DEL SOFTWARE

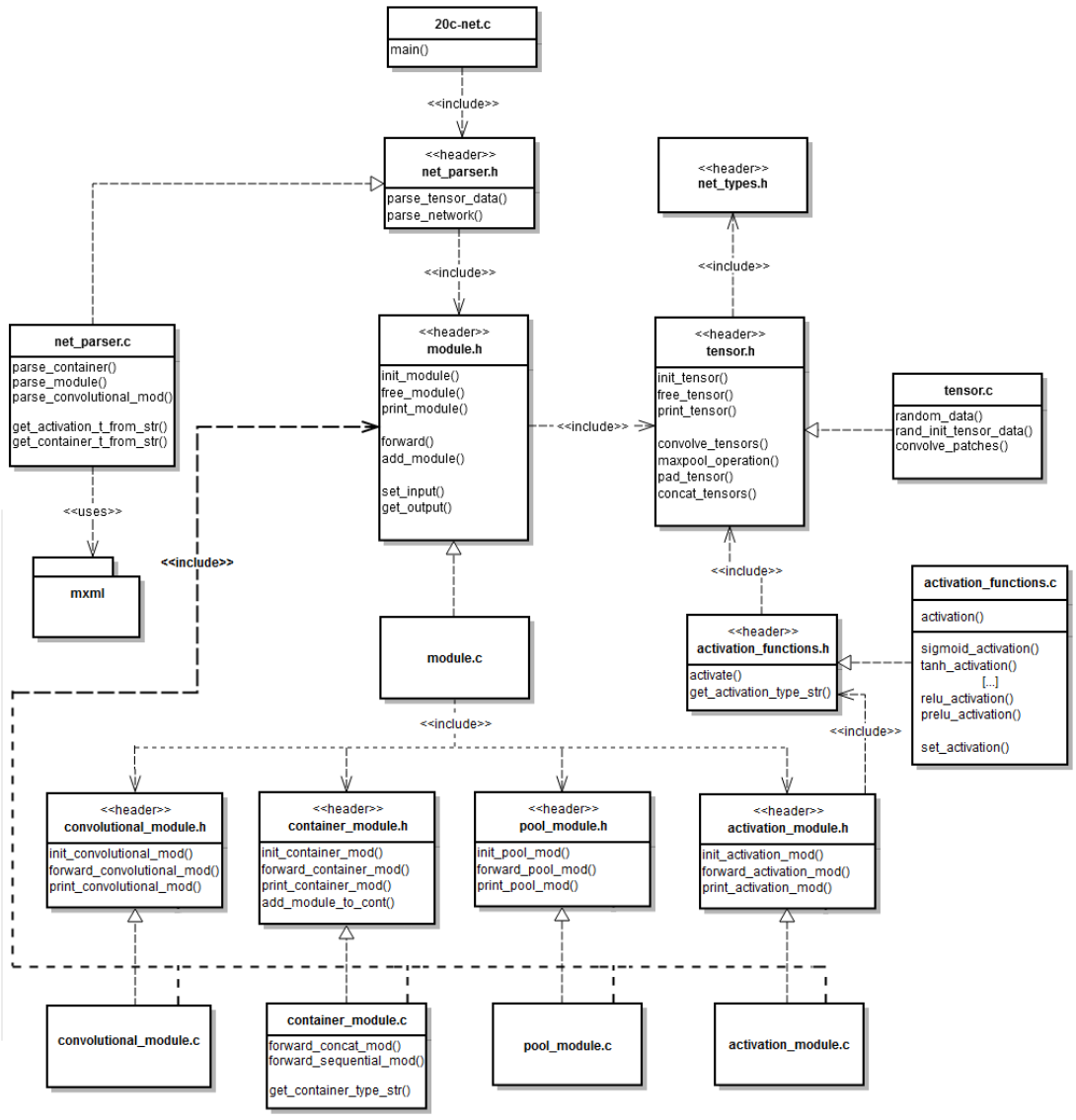


Figura A.1: Diagramma UML del software sviluppato

In questa documentazione, le parole chiave appartenenti al linguaggio C saranno denotate in corsivo, le costanti in rosso, e i tipi in verde scuro. Inoltre, queste saranno rappresentate con un font apposito. La documentazione di funzioni con tipo di ritorno `void` non hanno la sezione “Output”.

- **Libreria `net_types.h`:**
  - *`typedef`* **`... tensor_data_t:`** tipo riconfigurabile della rete. I tipi finora utilizzati sono stati `float` e `double`.
  - *`struct`* **`tensor:`** implementazione di tensore
    - `int d, w, h:` dimensioni del tensore. Sono, rispettivamente profondità, larghezza ed altezza.
    - `tensor_data_t *data:` puntatore all'area di memoria in cui risiede il contenuto del tensore.
  - *`enum`* **`activation_t:`** enumeratore dei tipi di funzione di attivazione attualmente disponibili:
    - `IDENTITY`
    - `BIN_STEP`
    - `SIGMOID`
    - `TANH`
    - `ARCTAN`
    - `SOFTSIGN`
    - `RELU`
    - `PRELU`
    - `ELU`
    - `SIN`
  - *`enum`* **`module_t:`** enumeratore dei tipi di modulo supportati dal sistema. L'enumerazione parte da 1 in quanto l'enumerativo 0 indica il modulo vuoto.
    - `CONVOLUTIONAL`
    - `CONTAINER`
    - `POOL`
    - `ACTIVATION`
  - *`enum`* **`container_t:`** enumeratore dei tipi di container attualmente disponibili.
    - `SEQUENTIAL`
    - `CONCAT`

- *struct* **module**: implementazione universale di modulo costituita dai seguenti campi
  - `module_t type`: tipo di modulo.
  - `tensor *input`: puntatore al tensore in ingresso. Punta all'output del modulo a monte o all'immagine di input per la rete.
  - `tensor output`: tensore in cui è immagazzinato il risultato dell'operazione svolta dal modulo.
  - `int n_filters`: numero di filtri. Diverso da 0 solo se il modulo è di tipo convolutivo.
  - `tensor *filters`: collezione di filtri/kernel. Diverso da `NULL` solo se il modulo è di tipo convolutivo.
  - `tensor bias`: set di bias. Inizializzato solo se il modulo è di tipo convolutivo, e se previsto da questo.
  - `int pad_h, pad_w`: iperparametri indicanti l'entità di zero-padding da applicare all'input del modulo. Valori significativi solo se il modulo è di tipo convolutivo o di pooling.
  - `int stride_h, stride_w`: iperparametri indicanti l'entità del passo di applicazione del kernel. Valori significativi solo se il modulo è di tipo convolutivo o di pooling.
  - `int ker_h, ker_w`: rispettivamente, altezza e larghezza di un filtro. Valori significativi solo se il modulo è di tipo convolutivo o di pooling.
  - `activation_t act_type`: tipo di funzione impiegata da un modulo di attivazione.
  - `container_t cont_type`: tipo di container.
  - `int concat_dim`: questo parametro è significativo solo se il modulo è un container di tipo `CONCAT` ed indica la dimensione secondo cui è concatenato il suo output. In

tal caso può valere come una delle seguenti costanti offerte dal sistema: `DEPTH`, `HEIGHT`, `WIDTH`.

- `int n_modules:` campo significativo solo in caso il modulo sia un container. In tal caso ne indica la quantità di moduli contenuti.
- *`struct`* `module *modules:` vettore di moduli presenti in un container. In caso il modulo non sia un container, questo campo vale `NULL`.

- **Libreria `tensor.h`:**
  - `tensor` **`init_tensor`**`(int d, int w, int h, int rand_init)`
    - **Input**: `d` = profondità; `w` = larghezza; `h` = altezza; `rand_init` = 1 per indicare inizializzazione dei dati del tensore con numeri random, 0 altrimenti.
    - **Output**: tensore inizializzato.
  - `void` **`free_tensor`**`(tensor *t)`
    - **Input**: `*t` = puntatore al tensore da deallocare.

    Questa funzione libera la memoria occupata dal campo `*data` del tensore puntato da `*t`.
  - `void` **`print_tensor`**`(tensor t)`
    - **Input:** `t` = tensore da stampare.

    Questa funzione mette a video il contenuto di `t` a partire dall'origine in alto a sinistra del tensore, dall'alto verso il basso, un livello di profondità alla volta.
  - `tensor` **`convolve_tensors`**`(tensor input, tensor *filters, tensor *bias, int n_filters, int pad_h, int pad_w, int stride_h, int stride_w)`

- **Input:** `input` = tensore in ingresso; `*filters` = collezione di filtri da applicare; `bias` = set di bias; `n_filters` = numero di filtri da applicare; `pad_h/w` = quantità di zero-padding da applicare orizzontalmente e verticalmente ad `input`; `stride_h/w` = entità dei passi orizzontali e verticali con cui i filtri sono applicati ad `input`.
- **Output:** tensore di output risultante.

Questa funzione è chiamata dal modulo convolutivo per effettuare la convoluzione di `input` con tutti i filtri contenuti in `*filters`.

- `tensor` **`maxpool_operation`**`(tensor input, int ker_height, int ker_width, int pad_h, int pad_w, int stride_h, int stride_w)`
  - **Input:** `input` = tensore in ingresso; `ker_height/width` = altezza e larghezza del kernel; `pad_h/w` = zero-padding orizzontale e verticale; `stride_h/w` = passi verticale ed orizzontale di traslazione del kernel;
  - **Output:** tensore di output risultante.

  Funzione invocata dal modulo di pooling per effettuare il sottocampionamento di `input` mediante max pooling.

- `tensor` **`pad_tensor`**`(tensor src, int pad_h, int pad_w)`
  - **Input:** `src` = tensore in ingresso; `pad_h/w` = entità di zero-padding da applicare orizzontalmente e verticalmente.
  - **Output:** tensore di output risultante.

  Funzione invocata per aggiungere `pad_h` colonne e `pad_w` righe di zeri agli estremi del volume `src`.

- `void` **`concat_tensors`**`(int concat_dim, int n_tensors, tensor *to_concat, tensor *out)`
  - **Input:** `concat_dim` = `DEPTH`/`WIDTH`/`HEIGHT`; `n_tensors` = numero di tensori da concatenare; `*to_concat` = puntatore ai tensori da concatenare; `*out` = puntatore al tensore di uscita;

    Questa funzione è invocata da container di tipo `CONCAT` per concatenare gli `n_tensors` tensori puntati da `*to_concat` secondo la dimensione indicata da `concat_dim`. Le costanti secondo cui tale parametro può essere espresso sono definite nella libreria `net_types.h`. Il risultato è immagazzinato in `*out`.

- **Libreria `activation_functions.h`:**
  - `tensor` **`activate`**`(tensor t, activation_t act)`
    - **Input:** `t` = tensore in ingresso; `act` = tipo di funzione di attivazione da applicare;
    - **Output:** tensore di output risultante.

    Questa funzione trasforma `t` elemento per elemento secondo la activation function indicata da `act`. È invocata da moduli di attivazione.
  - `char` **`*get_activation_type_str`**`(activation_t act)`
    - **Input:** `act` = tipo di funzione di attivazione;
    - **Output:** stringa di caratteri con il nome del tipo indicato da `act`.

    Funzione chiamata ai fini di stampa.

- **Libreria `module.h`:**
  - `void` **`set_input`**`(tensor *in, module *m)`
    - **Input:** `*in` = puntatore al tensore sorgente; `*m` = puntatore al modulo a cui assegnare l'input;

    Funzione utile per assegnare un tensore in input al modulo specificato.

  - `tensor` **`get_output`**`(module *m)`
    - **Input:** `*m` = puntatore al modulo da cui prelevare il tensore di output;
    - **Output:** tensore di output del modulo passato in input.

  - `module` **`init_module`**`(module_t type, int n_args, ...)`
    - **Input:** `type` = tipo del modulo da istanziare; `n_args` = numero di argomenti richiesti per inizializzare il modulo; `...` = lista di argomenti;
    - **Output:** modulo inizializzato.

    Questa funzione inizializza un modulo di tipo `type` smistando la chiamata verso una delle librerie che li implementano, la quale sarà illustrata successivamente. Nonostante questi si inizializzino con un diverso numero di parametri, questo task è comunque portato a termine grazie all'ausilio della libreria nativa di C, `stdarg.h`, per la quale si rimanda alla documentazione ufficiale.

  - `void` **`free_module`**`(module *m)`
    - **Input:** `*m` = puntatore al modulo da deallocare;

  - `void` **`print_module`**`(module m, int print_tensors)`
    - **Input:** `m` = modulo da stampare a video; `print_tensors` = 1 per mettere a video anche i tensori che fanno parte del modulo, 0 altrimenti;

    Questa funzione stampa a video le caratteristiche del modulo `m` (iperparametri ed eventuali parametri, in caso questi siano previsti e `print_tensors = 1`), smistando la chiamata verso la guista

libreria che lo implementa mediante ispezione del tipo del modulo in input.

- `void` **`forward`**`(module *m)`
  - **Input:** `*m =` puntatore al modulo a cui far partire l'elaborazione;

  Anche questa funzione, tramite ispezione del tipo di modulo, smista la chiamata verso il giusto modulo software. In particolare, fa sì che questo cominci l'elaborazione del tensore esplicitamente assegnatogli in precedenza come input tramite la funzione `set_input`.

- `void` **`add_module`**`(module *container, module to_add)`
  - **Input:** `*container =` puntatore al container; `to_add =` modulo da aggiungere.

  Questa funzione ha effetto solo se il tipo di `*container` è effettivamente `CONTAINER`. In tal caso, aggiunge il modulo `to_add` alla prima posizione libera nel campo `*modules` del container.

- **Librerie `convolutional_module.h`, `activation_module.h`, `pool_module.h` e `container_module.h`:**

  La documentazione per questi componenti software è stata accorpata dato che implementano le stesse funzioni, ma in maniera diversa per realizzare il comportamento dei vari tipi di modulo presenti in `net_types.h`. Di seguito, tali funzioni saranno quindi denotate con il loro nome contenente però un asterisco al posto del nome specifico del modulo. Inoltre, le funzioni realizzate in questi componenti software vengono chiamate solo ed esclusivamente dalle corrispondenti funzioni presenti in `module.h`, libreria che funge da interfaccia per qualunque tipo di modulo appartenente al sistema, e che internamente ne smista le chiamate in modo corretto.

- `module` **`init_*_mod`**`(...)`
  - **Input:** `...` = lista di parametri per inizializzare il modulo.
  - **Output:** modulo inizializzato.

  Questa funzione inizializza un certo tipo di modulo in base alla lista di parametri con cui è invocata la funzione `init_module` presente in `module.h`. In particolare, è possibile istanziare i moduli disponibili mediante le seguenti chiamate alla funzione appena citata:

  - **`init_module`**`(CONTAINER, N_INIT_ARG_CONT, container_t cont_type, int n_modules, int concat_dim)`
    Inizializzazione di container. **Parametri** utili, in ordine di inserimento: tipo di container, numero di moduli che conterrà, dimensione secondo cui concatenare gli output in caso `cont_type = CONCAT`.

  - **`init_module`**`(CONVOLUTIONAL, N_INIT_ARG_CONV, int n_fil, int ker_h, int ker_w, int pad_h, int pad_w, int stride_h, int stride_w, int input_depth, tensor_data_t *weights, tensor_data_t *bias)`
    Inizializzazione del modulo convolutivo. **Parametri** utili, in ordine di inserimento: numero di filtri, altezza e larghezza dei filtri, quantità di zero-padding verticale ed orizzontale, entità del passo di applicazione verticale ed orizzontale dei filtri, vettore di pesi con cui inizializzare i filtri, vettore di bias.

  - **`init_module`**`(POOL, N_INIT_ARG_POOL, int ker_h, int ker_w, int pad_h, int pad_w, int stride_h, int stride_w)`
    Inizializzazione del modulo di pooling. **Parametri** utili, in ordine di inserimento: altezza e larghezza del kernel con cui effettuare il sottocampionamento, quantità di zero-padding verticale ed orizzontale, entità del passo di applicazione verticale ed orizzontale del kernel.

  - **`init_module`**`(ACTIVATION, N_INIT_ARG_ACTIV, activation_t act)`

Inizializzazione del modulo di attivazione. **Parametri** utili, in ordine di inserimento: tipo di funzione di attivazione che sarà utilizzata dal modulo.

A partire dal terzo parametro in ingresso ad `init_module`, è possibile considerare tali parametri come la lista di parametri d'ingresso del tipo di modulo che si sta istanziando. Le costanti `N_INIT_ARG_*` rappresentano la lunghezza di questa lista caso per caso, e sono definite nella libreria `net_types.h`.

- `void` **`forward_*_mod`**`(module *m)`
  - **Input:** `*m` = modulo per cui far partire l'elaborazione del tensore assegnatogli in input.

  Funzione invocata dalla `forward` in `module.h` a seconda del tipo di modulo, che invoca a sua volta una core function. Nel caso del modulo di attivazione, è chiamata la funzione `activate`; nel caso del modulo convolutivo è chiamata la funzione `convolve_tensors`; nel caso del modulo di pooling è chiamata la funzione `maxpool_operation`; nel caso di un container, in base alla sua natura, questo si occupa di impostare il tensore in input per i moduli in esso contenuti, ed in seguito ne invoca la funzione `forward`, che sarà sempre smistata dal componente `module.h`. Il risultato dell'operazione è memorizzato sempre nel campo `output` del rispettivo modulo.

- `void` **`print_*_mod`**`(module m, int print_tensors)`
  - **Input:** `m` = modulo di cui stampare a video la descrizione; `print_tensors` = 1 per mettere a video i tensori interni al modulo, se previsti, 0 altrimenti.

  Funzione invocata dalla `print_module` in `module.h` in base al tipo di `m`, la quale ne propaga anche il valore di `print_tensors`. Stampa a video la descrizione del modulo esponendone gli

iperparametri che lo caratterizzano, ed i parametri in caso questi siano previsti e `print_tensors = 1`.

- **Libreria `net_parser.h`:**
  - `tensor_data_t` **`*parse_tensor_data`**`(`*`const`* `char *str, int n_data)`
    - **Input:** `*str =` stringa di caratteri; `n_data =` lunghezza della stringa di caratteri `*str`.
    - **Output:** puntatore all'area di memoria contenente i dati del tensore ricavati a partire dalla stringa in input.

    Questa funzione converte gli `n_data` numeri presenti sottoforma di stringhe di caratteri in `*str`, la quale è stata precedentemente inizializzata a partire dal contenuto di un file, e ne restituisce il risultato in output.

  - `module` **`parse_network`**`(char *to_parse_fn)`
    - **Input:** `*to_parse_fn =` nome del file XML di configurazione della rete, inclusivo di percorso relativo o assoluto.
    - **Output:** modulo contenente l'intera rete inizializzata.

    Questa funzione legge la configurazione della rete dal file XML indicato da `*to_parse_fn` e la inizializza come modulo, dato che in genere una rete è composta da almeno un container, il quale è anch'esso un modulo, in cui sono organizzati i vari moduli. Infine, il modulo correttamente istanziato è restituito in output.

- **Istruzioni per la compilazione del software su sistemi operativi Linux:**

  Collocarsi con un terminale all'interno della cartella radice in cui sono presenti i file con il codice sorgente del programma, ed inserire il seguente comando:

```
gcc -o 20c-net -I. -I./mxml ./*.c ./mxml/*.c -lm -lpthread -O3
```

  Tale istruzione genererà l'eseguibile `20c-net` nella stessa cartella, pronto per l'esecuzione.