DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

FPGA: L'AVANGUARDIA DELL'ACCELERAZIONE HARDWARE PER L'INTELLIGENZA ARTIFICIALE

Relatore:
PROF. DANIELE VOGRIG

Laureanda:
SOFIA BELLIN

ANNO ACCADEMICO 2021-2022
DATA DI LAUREA: 21-09-2022

ABSTRACT

Con l’evoluzione della tecnologia il mondo sta diventando sempre più connesso e pensante, tra le principali tecnologie a cui si deve questo risultato troviamo l’intelligenza artificiale (AI) e l’*Internet of Things* (IoT).

L'intelligenza artificiale è probabilmente la più comunemente conosciuta tra i due fattori coinvolti perché molto spesso considerata sinonimo di robot intelligenti che un giorno prenderanno il sopravvento sull’umanità. In realtà, l’AI ad oggi ha ben poco a che fare con queste visioni catastrofiche frutto di pura fantascienza. Molti, infatti, ignorano che quest'ultima è invece già parte integrante della nostra quotidianità. A partire dai risultati che ci compaiono su un motore di ricerca, agli assistenti virtuali che troviamo nei nostri smartphone; ancora, nei suggerimenti che ci compaiono nelle varie piattaforme online, dal film che potrebbe piacerti ad un qualunque altro prodotto che potresti acquistare. Gli ambiti di applicazione sono quindi infiniti, come infinito è il potenziale che dal suo sviluppo ne può derivare.

Dall’altra parte, l'Internet delle cose o IoT, anticipa un mondo saturo di gadget intelligenti, spesso chiamati "oggetti intelligenti" interconnessi tramite Internet o altri mezzi di comunicazione come Bluetooth, infrarossi, ecc... Questi dispositivi sono dotati di sensori in grado di raccogliere e collezionare un’enorme quantità di dati che vengono poi trasmessi ad altri dispositivi o in un cloud. Il sistema ha lo scopo di monitorare, controllare e trasferire informazioni, per poi svolgere azioni conseguenti. Così facendo, si crea una mappa intelligente di tutte le cose, del loro funzionamento e delle informazioni rilevate da ognuna di esse, con lo scopo di creare nuove forme di conoscenza.

Una delle direzioni in cui si sta muovendo la ricerca del progresso tecnologico è quella di creare un rapporto proprio tra queste due forme di tecnologia, utilizzando l’AI nei dispositivi IoT al fine di sbloccare quello che è il loro vero potenziale. In questo modo, infatti, si consente a reti e dispositivi di imparare dalle decisioni passate, prevedere attività future e migliorare continuamente le prestazioni e le capacità decisionali. Con l’integrazione dell’AI i dispositivi sono in grado di "pensare da soli", interpretando i dati e prendendo decisioni in tempo reale senza i ritardi e congestioni che si verificano a causa dei trasferimenti di dati. Questa prospettiva però, vede un cambiamento nell’uso dei dispositivi IoT, i quali dal trasporto dei dati verso il cloud si muovono verso l’elaborazione dei dati all’Edge ovvero, in prossimità del dispositivo

stesso. La limitazione della larghezza di banda e il tempo di latenza che ne sarebbero derivati trasmettendo prima tutti i dati al cloud, vengono quindi rimossi.

In questa tesi viene tracciato il punto della situazione sul progresso ottenuto dall'uso degli FPGA nell'accelerazione hardware per l'intelligenza artificiale, attraverso l’analisi di alcuni esempi applicativi focalizzati sull'*Internet of Things* e quindi sull’accelerazione all’Edge.

Il Capitolo 1, dopo una breve introduzione all’intelligenza artificiale, si focalizza sulle architetture neurali che saranno oggetto di studio degli esempi analizzati in seguito, fornendo così gli strumenti necessari al lettore per comprenderli. Il Capitolo 2, dopo aver spiegato il perché in uno scenario edge sia necessaria un’accelerazione di tipo hardware, attraverso brevi confronti con altri dispositivi, spiega a livello teorico il motivo per cui l’FPGA risulta essere tra le migliori soluzioni presenti nel mercato. Infine, il Capitolo 3, attraverso la presentazione di quattro esempi di applicazione reali, ha lo scopo di fornire una dimostrazione a livello pratico di quanto esposto in precedenza.

INDICE

ACRONIMI E ABBREVIAZIONI 6

CAPITOLO 1 INTRODUZIONE ALL'INTELLIGENZA ARTIFICIALE 7

1.1 Definizione e Ambiti di Applicazione 7

1.2 Dall'Intelligenza Artificiale alle Reti Neaurali Artificiali 8

1.2.1 CNN: Reti Neurali Convoluzionali 11

1.2.2 MTCNN: Rete Neurale Convoluzionale Multi-Task 13

1.2.3 Tiny-YOLOv3 14

1.2.4 LSTM: Long Short-Term Memory 16

1.2.5 TCNN: Reti Neurali Convoluzionali Temporali 1D 18

CAPITOLO 2 ACCELERAZIONE HARDWARE 20

2.1 Dall'IoT all'Edge Computing 20

2.2 Strategie Implementative per l'Ottimizzazione 21

2.3 Confronti tra Dispositivi Hardware per L'inferenza Edge 22

2.4 Vantaggi sull'Utilizzo dell'FPGA 24

CAPITOLO 3 ESEMPI APPLICATIVI 26

3.1 Rilevamento e Allineamento Facciale Ottimizzato 26

3.1.1 Algoritmo 27

3.1.2 Architettura del Sistema 28

3.1.3 Implementazione Logica Programmabile 29

3.1.4 Struttura della PU 30

3.1.5 Macchina a Stati Finiti 31

3.1.6 Risultati e Conclusioni 32

3.2 Riconoscimento Oggetti 33

3.2.1 Architettura del Sistema 34

3.2.2 Strategie Implementative 37

3.2.3 Risultati e Conclusioni 40

3.3 Manutenzione Predittiva 44

3.3.1 Architettura del Nodo di Sistema 44

3.3.2 Implementazione Logica Programmabile 45

3.3.3 Unità di Calcolo 46

3.3.4 Risultati e Conclusioni 48
3.4 Classificazione Testo 51
3.4.1 Architettura del Sistema 51
3.4.2 Implementazione Logica Programmabile 53
3.4.3 Risultati e Conclusioni 54
CONCLUSIONI 57
BIBLIOGRAFIA 59

ACRONIMI E ABBREVIAZIONI

IoT	Internet of Things
AI	Intelligenza Artificiale
FPGA	Field-Programmable Gate Array
ML	Machine Learning
ANN	Rete Neurale Artificiale (Artificial Neural Network)
NN	Reti Neurali (Neural Network)
CNN	Rete Neurale Convoluzionale (Convolutional Neural Network)
RNN	Rete Neurale Ricorrenti (Recurrent Neural Network)
FC	Rete Completamente Connessa (Fully-Connected)
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ASIC	Application Specific Integrated Circuits
MAC	Multply-Accumulate
PU	Processing Unit
DSP	Digital Signal Processing
BRAM	Block Random Access Memory
FM	Mappa delle Caratteristiche (Feature Map)

Capitolo 1
INTRODUZIONE ALL’INTELLIGENZA ARTIFICIALE

Un dispositivo IoT è un punto di contatto e di interfaccia tra il mondo fisico e il mondo cibernetico. Nello specifico può essere visto funzionalmente come un endpoint in grado di acquisire dati dal mondo reale per renderli disponibili ai client autorizzati ovunque su Internet.[1] Qualsiasi dispositivo abilitato all'IoT può rilevare l'ambiente circostante, trasmettere, archiviare ed elaborare i dati raccolti per agire di conseguenza. L'ultima fase, ovvero quella di agire di conseguenza dipende interamente dalla qualità della fase di elaborazione ed è ciò che distingue un sistema IoT in grado di evolversi autonomamente da uno non. È in questo contesto che entra in gioco l’intelligenza artificiale, la quale aumenta il valore dei dispositivi IoT elaborando gli enormi volumi di dati e rilevando i modelli sottostanti. In questo modo, il dispositivo diventa in grado di prevedere le condizioni operative e le modifiche necessarie per ottenere risultati sempre migliori. [2]

1.1 **Definizione e Ambiti di Applicazione**

Per definizione, l’intelligenza artificiale è una riproduzione parziale dell'attività intellettuale propria dell'uomo (con particolare riguardo ai processi di apprendimento, di riconoscimento, di scelta) realizzata o attraverso l’elaborazione dei modelli ideali, o, concretamente, con la messa a punto di macchine che utilizzano perlopiù a tale fine elaboratori elettronici. [3]

Gli ambiti di applicazione sono tra i più disparati, dal riconoscimento facciale e classificazione di immagine, alla traduzione automatica quindi riconoscimento del linguaggio. Nell’ambito della sicurezza informatica può essere utilizzata per migliorare i sistemi di rilevamento delle intrusioni o per sventare attacchi apprendendo i comportamenti tipici del malware. In ambito finanziario per rilevare addebiti o reclami al di fuori della norma, contrassegnandoli per le indagini umane o in quello sanitario dove l’intervento dell’AI può aprire nuovi scenari della medicina ottimizzando e perfezionando il lavoro dei medici (es: interventi chirurgici robotici, diagnosi assistita, telemedicina, monitoraggio statistiche vitali).[4]

1.2 **Dall'Intelligenza Artificiale alle Reti Neurali Artificiali**

Uno degli strumenti principali per ottenere l'AI è il *machine learning* o apprendimento automatico. L'idea di base si fonda sul cercare di riprodurre il processo di apprendimento nelle macchine, permettendo loro di risolvere un problema senza che queste vengano esplicitamente programmate per farlo. Attraverso l'utilizzo di algoritmi, infatti, la macchina è in grado di definire modelli capaci di imparare e di prendere decisioni, estraendo informazioni e caratteristiche da set di dati che le vengono posti in ingresso. In questo modo questa risolve problemi e fa previsioni sulla base dell'esperienza acquisita in precedenza, riproducendo così la fase di apprendimento che avviene nell'essere umano.[5]

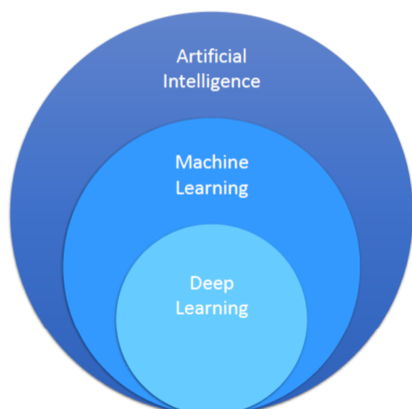


Figura 1.1 Gerarchia dell'AI

Una classe del *machine learning* è il *deep learning*, i cui algoritmi sono ispirati alla struttura e alla funzione del cervello umano, imitando il modo in cui gli umani acquisiscono alcune tipologie di conoscenza. Si tratta di un metodo ad hoc di machine learning che può ingerire dati non strutturati nella sua forma grezza e determinare automaticamente la gerarchia delle caratteristiche che li distinguono. Una delle tipologie di rete più comuni sono le reti neurali artificiali (ANN) o semplicemente *neural network* (NN). L'idea prende ispirazione dal sistema neurale biologico animale, il quale, è composto da neuroni interconnessi, che costituiscono le unità computazionali di base (chiamate nodi), che comunicano tra loro attraverso impulsi elettrici.

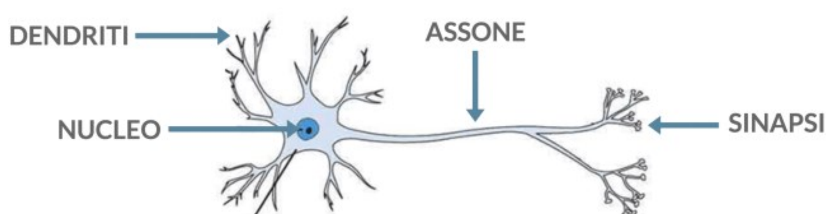


Figura 1.2 Struttura del Neurone Biologico

Ogni neurone biologico (Figura 1.2) è composto da: il corpo cellulare, al cui interno è contenuto il nucleo dove l'informazione viene processata; i dendriti, zone ricettive che ricevono il segnale di input; l'assone, linea di trasmissione attraverso la quale si produce e propaga il segnale di output; infine, le sinapsi, che permettono la trasmissione pesata di segnali tra assoni e dendriti per costruire reti neurali più grandi. Il comportamento di queste strutture è basato sul fatto che solo quando l'input eccede una certa soglia, l'impulso si propaga in uscita. Tutte le sinapsi possono modificare il valore del segnale che le attraversa e il modo in cui il cervello impara è proprio cambiando il valore associato a ciascuna di loro.[6] Le reti neurali artificiali utilizzano lo stesso approccio, in Figura 1.3 è presentato il modello matematico di un neurone artificiale anche detto *perceptron*. Il

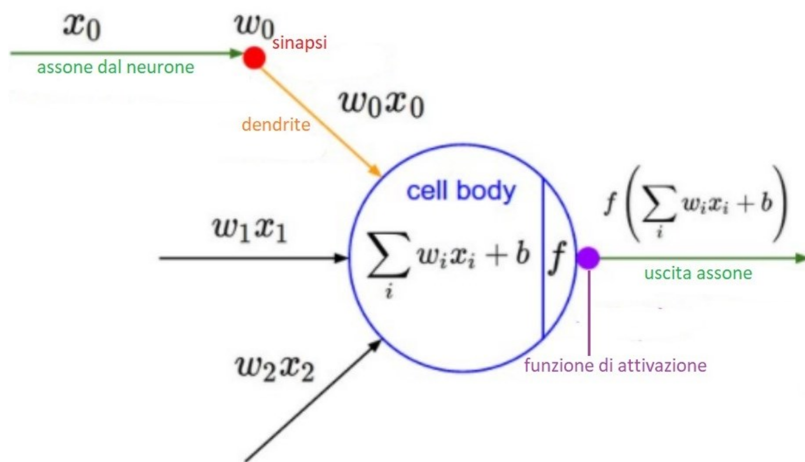


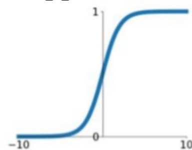
Figura 1.3 Struttura del Perceptron o Neurone Artificiale

neurone, o nodo, ha ingressi multipli provenienti da più nodi, a ciascuno di questi è associato un certo valore (chiamato peso) che moltiplica il corrispondente valore del segnale di input, permettendo di calcolare il segnale di output dell'unità in esame. Da un punto di vista matematico ogni nodo di un NN esegue una somma tra tutti gli input moltiplicati per i corrispondenti pesi e un parametro b di bias. Al valore risultante viene poi applicata una funzione non lineare o funzione di attivazione, la quale, genera l'output definitivo solo se il segnale di input supererà un certo valore di soglia.[6] $y = f\left(\sum(w_i x_i + b)\right)$

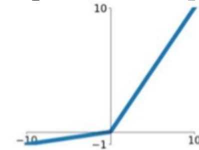
I pesi e le attivazioni sono comunemente organizzati per strati o layers rispettivamente in matrici e vettori riferendosi quindi alle somme pesate come moltiplicazioni matrice-vettore. [6]

In Figura 1.4 sono rappresentate alcune delle funzioni di attivazione più comuni.[7]

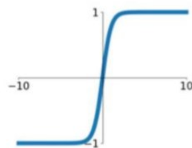
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

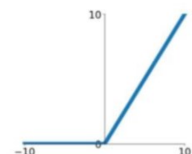


tanh
$\tanh(x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

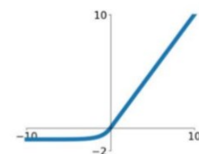


Figura 1.4 Principali funzioni di attivazione applicate ai valori di output dei neuroni

Alle quali si aggiunge la funzione softmax, o funzione esponenziale normalizzata, la quale converte un vettore di K numeri reali in una distribuzione di probabilità di K possibili risultati. Nelle reti neurali viene spesso usata come funzione di attivazione per normalizzare l'output in classi della rete in una distribuzione di probabilità.[8]

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

Una rete neurale generalmente può essere modellata come in Figura 1.5. Un insieme multiplo di nodi viene detto organizzato in layers, o livelli, nei quali ciascun nodo riceve input da nodi del livello precedente e invia output ad altri nodi appartenenti al livello successivo. Se ogni neurone di uno strato è connesso ad ogni neurone dello strato successivo, la rete si dice completamente connessa (*Fully-Connected* FC). Si possono identificare due forme principali di NN: *feedforward neural network* (FNN) e *recurrent neural network* (RNN).[6] La prima (con flusso in avanti) è detta così per la sua caratteristica di far muovere le informazioni in una sola direzione, avanti, dai nodi di ingresso verso quelli di uscita, passando per quelli nascosti se presenti.[9] Questa tipologia di rete non ha memoria degli input avvenuti a tempi precedenti, per cui l'output è determinato solamente dall'attuale input. D'altro canto, invece, nella RNN i neuroni possono ammettere loop e/o possono essere interconnessi anche a neuroni di un precedente livello. Pertanto, il segnale non si propaga solo in avanti, ma può muoversi anche all'indietro. Questa caratteristica, detta ricorrenza, introduce quindi il concetto di memoria di una rete. L'output di un neurone può influenzare sé stesso, in uno step temporale successivo o può influenzare neuroni della catena precedente, i quali a loro volta interferiranno con il comportamento del neurone su cui si chiude il loop.[10]

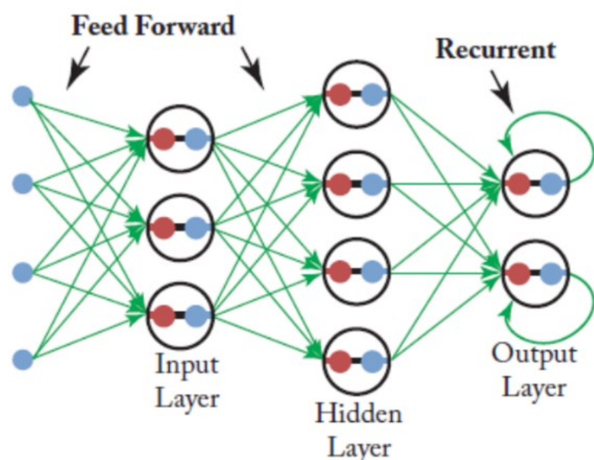


Figura 1.5 Generica struttura delle due principali NN: FNN e RNN

Il processo di apprendimento del *Deep learning* consiste nel modificare i valori di rete quali pesi e bias durante l'esecuzione dei dati di input, questa fase viene detta di *training* o allenamento. Una volta che la rete è allenata quindi i parametri sono stati calcolati, essa è in grado di risolvere i compiti assegnati attraverso il calcolo dell'output, questa fase viene detta di *inference* o inferenza. Esistono diversi algoritmi per allenare la rete, una delle scelte più comuni è quella di utilizzare l'algoritmo di *backpropagation* il quale, è composto da due fasi: *forward* e *backward*. La prima consiste in una propagazione ripetuta attraverso la rete degli input verso gli output. Mentre la seconda, propaga un errore calcolato mediante la *loss function* (fornisce una misura di quanto l'output predetto è vicino al risultato aspettato) dall'output verso l'input. Durante il passaggio attraverso i layers quindi i pesi vengono aggiustati in accordo con l'errore di ogni specifico layer. [7]

Nei sotto capitoli seguenti verranno analizzate le architetture di rete neurale artificiale più comuni e quelle che nello specifico saranno utilizzate in seguito nelle applicazioni prese in analisi.

1.2.1 *CNN: Reti Neurali Convoluzionali*

Questa classe di reti neurali vengono adottate quando i dati in ingresso sono in relazione tra loro a livello spaziale o temporale. L'idea, infatti, è di sfruttare la correlazione esistente tra gli input vicini per ridurre il numero di parametri per layer. La differenza rispetto alle tradizionali ANN e la presenza di un nuovo layer detto di convoluzionale (CONV) nel quale, come già il nome suggerisce, l'operazione di base che viene eseguita è la convoluzione tra gli input e filtri o kernel in grado di estrarre informazioni da essi. Nei calcoli, sia gli input sia gli output relativi a questi layer sono organizzati in matrici denominate mappe delle caratteristiche o *feature maps* (FM), ognuna delle quali è definita come canale. I filtri vengono applicati alle feature map scorrendo sequenzialmente su di esse ed eseguendo l'operazione al fine di calcolare la mappa di output corrispondente quindi l'immagine caratterizzata (Figura 1.6 e 1.7). Una funzione di attivazione decide poi se una determinata caratteristica è presente in una determinata posizione dell'immagine o meno. La dimensione dell'output ottenuto in uscita dalla convoluzione di

$$n_{out} = \frac{n_{in}+2p-k}{s} + 1$$

un'immagine è uguale alla dimensione dell'input più due volte il riempimento p meno la dimensione del kernel k diviso lo *stride s* più uno. Dove *p* o *padding* rappresenta le dimensioni del bordo esterno applicato all'immagine di input ed s la velocità di traslazione del kernel sull'immagine, entrambe misurate in pixel orizzontali e verticali.[11]

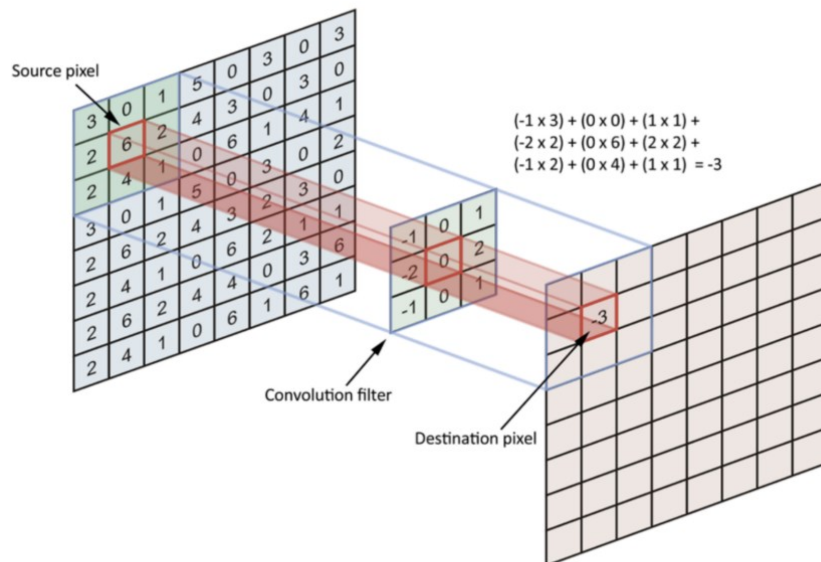


Figura 1.6 Processo di convoluzione in cui il filtro scorre sequenzialmente sulla mappa di input ottenendo un unico valore di output per ciascuna operazione matrice-matrice come somma delle moltiplicazioni valore per valore.

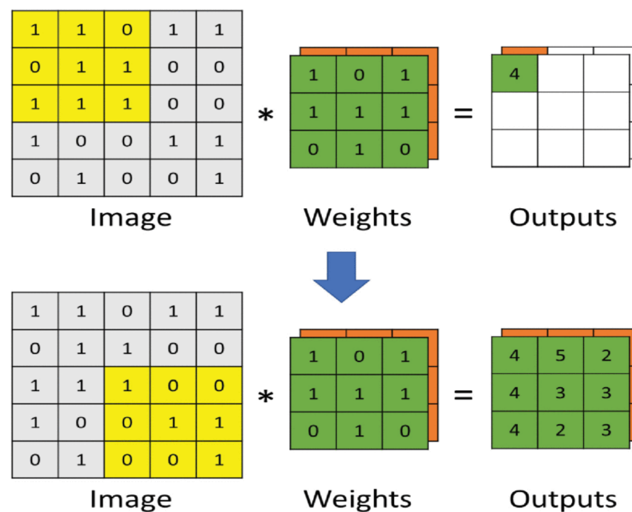


Figura 1.7 Generico esempio di un'operazione di convoluzione

Dopo i layer di convoluzione, le CNN possono contenere altre tipologie di layer tra cui *pooling*, attivazioni e normalizzazione. I primi, servono a diminuire le dimensioni dell'immagine in input scansionando l'immagine con una matrice delle dimensioni del pooling e dalla sottomatrice ottenuta, prende un unico valore che può essere il valore massimo (*max pooling*) o il valore medio (*average pooling*). In questo modo, il risultato finale non è un semplice ridimensionamento dell'immagine in quanto l'informazione massima o media di ogni sottomatrice viene mantenuta, quindi insieme le caratteristiche principali della stessa.[11] Le attivazioni invece corrispondono all'applicazione di una delle funzioni presentate sopra e servono ad aggiungere non linearità alla rete in modo tale che non vi sia alcuna relazione lineare tra ogni strato.[12] Così facendo, è possibile sia rimuovere informazioni inutili, sia aumentare il livello di scarsità della rete. Infine, la normalizzazione viene applicata per controllare la distribuzione dei valori nelle mappe delle caratteristiche e può aiutare a velocizzare l'apprendimento e aumentare l'accuratezza.[7]

Prima dell'output di classificazione di una CNN vengono infine posizionati i livelli completamente connessi, i quali, determinano a quale classe può appartenere effettivamente l'immagine. In generale quindi ogni livello della CNN apprende filtri di complessità crescente. I primi livelli apprendono i filtri di rilevamento delle caratteristiche di base (bordi, angoli, ecc..), quelli intermedi rilevano parti di oggetti (per i volti, potrebbero imparare a rispondere a occhi, nasi, ecc..), gli ultimi strati hanno rappresentazioni più alte: imparano a riconoscere oggetti interi, in diverse forme e posizioni.[13]

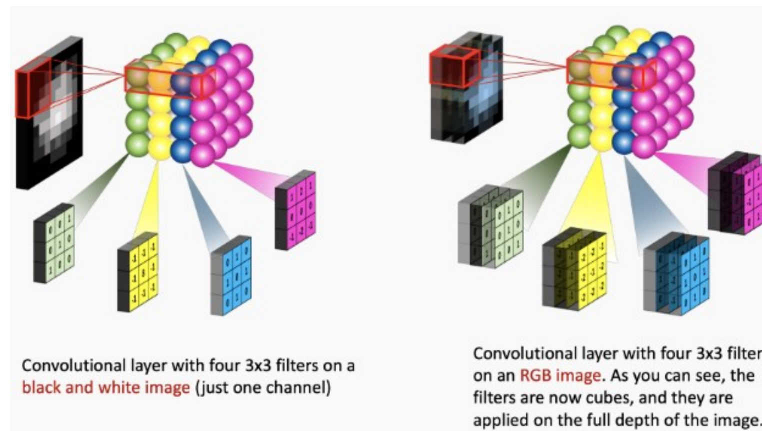


Figura 1.8 differenza tra la convoluzione di un'immagine monocromatica (sx) e una a colori (dx)

1.2.2 *MTCNN: Rete Neurale Convoluzionale a Cascata Multi-Task*

Con la rapida evoluzione delle tecnologie delle reti neurali profonde, sono stati proposti approcci di rilevamento dei volti basati sulla CNN per molte applicazioni di elaborazione delle immagini facciali; in particolare, esistono due tipi di algoritmi di rilevamento dei volti basati sulla CNN: non in cascata e in cascata. I primi richiedono una quantità di calcoli inferiore rispetto ai secondi. Tuttavia, gli algoritmi non in cascata hanno lo svantaggio di una minore accuratezza. Per ottenere prestazioni migliori, è stato recentemente introdotto un algoritmo a cascata noto come rete neurale convoluzionale a cascata multi-task (MTCNN). La sua maggiore precisione si ottiene utilizzando una correlazione tra il rilevamento del viso e l'allineamento ed è costituito da 3 reti CNN in cascata: *Propose Network* (P-Net), *Refine Network* (R-Net) e *Output Network* (O-Net), Figura 1.9. [14]

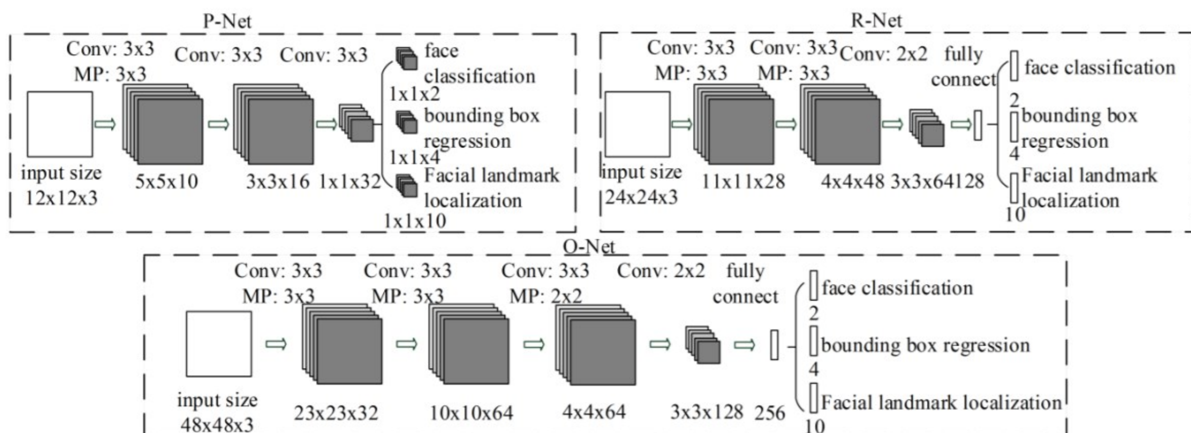


Figura 1.9 Struttura MTCNN

L'immagine viene scalata in più dimensioni in modo da rilevare volti di varia grandezza e per ogni copia in scala un kernel 12x12 passerà attraverso ogni parte dell'immagine, scansionandone i volti. Inizia nell'angolo in alto a sinistra, una sezione dell'immagine da (0,0) a (12,12). Questa parte dell'immagine viene passata a P-Net, che restituisce le coordinate di un riquadro di delimitazione se rileva una faccia.[14] Quindi, ripeterebbe quel processo con le

sezioni da (0+2a,0+2b) a (12+2a, 12+2b), spostando il kernel 12 x 12 di 2 pixel a destra o in basso alla volta. Lo spostamento di 2 pixel è noto come stride che, come visto, indica di quanti pixel si muove il kernel ad ogni iterazione. Il p-net rileva quindi più dimensioni del viso nell'immagine originale e la sua uscita consiste in un insieme di riquadri di delimitazione e punti di riferimento facciali.[15] Attraverso la soppressione non massima (NMS) vengono poi ridotti i riquadri di delimitazione sovrapposti che non sono sicuri, conservando solo i riquadri del viso più probabili. In particolare, infatti, il metodo dell'NMS consiste nel selezionare una singola entità tra molte entità sovrapposte, scartando quelle che sono al di sotto di un determinato limite di probabilità e tra le rimanenti sceglie ripetutamente l'entità con la probabilità più alta. Successivamente, tutti i candidati individuati da P-net vengono poi ridimensionati a 24x24 e inviati alla rete successiva R-net. Questo perfeziona i riquadri di delimitazione ricevuti, quindi esegue NMS per ridurre i sovrapposti. Al termine dell'esecuzione di R-net le uscite vengono ridimensionate in immagini 48 × 48 per essere inviate all'ultima rete O-Net le cui uscite sono leggermente diverse da quelle di P-Net e R-Net. O-Net fornisce 3 output: le coordinate del riquadro di delimitazione (out[0]), le coordinate dei 5 punti di riferimento facciali (out[1]) e il livello di confidenza di ciascun riquadro (out[2]). L'O-Net genera quindi i riquadri di delimitazione finali che hanno volti e punti di riferimento facciali.[15] I parametri e gli stati utilizzati nella rete sono contenuti in Tabella I.I.

Networks[1]	States[2]	Functions	Parameters
P-Net	00	Conv2D,Maxpooling	Ks[3]:3,ch[4]:10,st[5]:1,mp[6]:2
	01	Conv2D	ks:3,ch:16,st:1,mp:0
	10	Conv2D	ks:3,ch:32,st:1,mp:0
R-Net	00	Conv2D,Maxpooling	Ks:3,ch:28,st:1,mp:3
	01	Conv2D,Maxpooling	ks:3,ch:48,st:1,mp:3
	10	Conv2D	ks:3,ch:64,st:1,mp:0
	11	Fully Connected	neurons: 128
O-Net	000	Conv2D,Maxpooling	Ks:3,ch:32,st:1,mp:3
	001	Conv2D,Maxpooling	ks:3,ch:64,st:1,mp:3
	010	Conv2D,Maxpooling	ks:3,ch:64,st:1,mp:2
	011	Conv2D	ks:2,ch:128,st:1,mp:0
	100	Fully Connected	neurons: 256

TABELLA I.I Stati e parametri utilizzati nella rete

1.2.3 *Tiny-YOLOv3*

I rilevatori di oggetti individuano e classificano gli oggetti presenti in un'immagine in genere disegnando attorno ad esso un rettangolo di selezione; quindi, classificano gli oggetti come appartenenti a un insieme predefinito di classi.[16]

Tiny-YOLOv3 ha due diverse versioni per diverse dimensioni dell'immagine: 320 × 320, 416 × 416 o 608 × 608 e restituisce i riquadri di delimitazione candidati in due diverse scale: 26 × 26 e 13 × 13 per rilevare rispettivamente oggetti di piccole e grandi dimensioni. Di conseguenza, per lo stesso oggetto potrebbero essere trovati più riquadri di delimitazione, quindi per rimuovere tali rilevamenti multipli viene utilizzata la soppressione non massima

(NMS). L'insieme dei riquadri di selezione candidati viene filtrato in base a un punteggio che dipende da ciascuna classe.

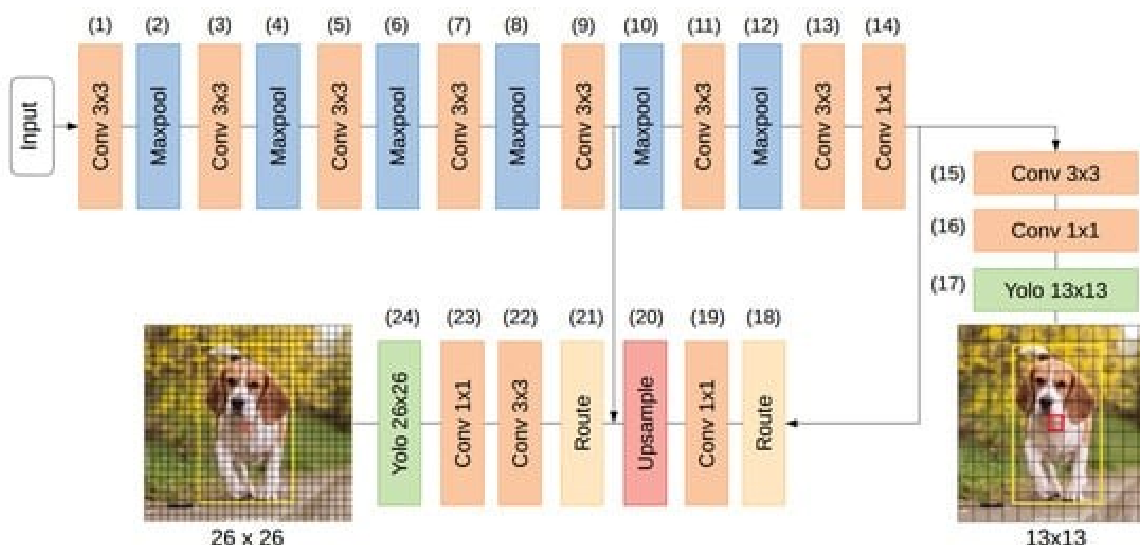


Figura 1.10 Layers che componogono l'architettura Tiny-YOLOv3

La prima parte della rete è responsabile dell'estrazione delle caratteristiche dall'immagine di input ed è composta da una serie di strati convoluzionali e maxpool. Gli strati Maxpool riducono gli FM di un fattore quattro lungo il percorso. Inoltre, dal momento che al livello 12 viene eseguito un pooling con stride 1, la risoluzione di input e output è la stessa. In questa implementazione di rete, le convoluzioni utilizzano il riempimento zero attorno agli FM di ingresso, quindi la dimensione viene mantenuta negli FM di uscita.

La seconda parte, esegue il rilevamento e la classificazione degli oggetti a (13 × 13) e (26 × 26). Il rilevamento ad una risoluzione inferiore si ottiene passando l'output di estrazione delle caratteristiche a 3 × 3 e 1 × 1 strati convoluzionali e uno strato YOLO alla fine. Il rilevamento alla risoluzione più alta segue la stessa procedura ma utilizza FM da due livelli della rete. Il secondo rilevamento utilizza risultati intermedi dai livelli di estrazione delle caratteristiche concatenati con FM ingranditi utilizzati per il rilevamento a risoluzione inferiore. La combinazione di FM da due diverse risoluzioni contribuisce a una maggiore significatività utilizzando le informazioni dal livello sovracampionato e le informazioni a grana più fine dalle mappe delle caratteristiche precedenti.[17] In Tabella I.II sono contenuti i dettagli in termini di tipologia di layers e dimensioni delle strutture che compongono la rete.

Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

TABELLA I.II Tipologia di Layers che compongono la rete con corrispondente dimensioni di filtri, input e output

1.2.4 *LSTM: Long Short-Term Memory*

Una rete LSTM è un tipo più avanzato di rete neurale ricorrente (RNN) in grado di apprendere le dipendenze a lungo termine tra i passaggi temporali dei dati di sequenza. Le RNN tradizionali, infatti, soffrono di memoria a breve termine a causa del problema del gradiente evanescente. In una rete neurale *feed-forward* convenzionale, l'aggiornamento del peso applicato su un particolare livello è un multiplo della velocità di apprendimento, del termine di errore dal livello precedente e dell'input di quel livello. Pertanto, il termine di errore per un particolare livello è da qualche parte un prodotto degli errori di tutti i livelli precedenti. Quando si tratta di funzioni di attivazione, come la funzione sigmoide, i piccoli valori delle sue derivate (che si verificano nella funzione di errore) vengono moltiplicati più volte mentre ci spostiamo verso i livelli iniziali. Di conseguenza, il gradiente quasi svanisce mentre ci spostiamo verso i livelli di partenza e diventa difficile allenare questi livelli. Un caso simile si osserva nelle reti neurali ricorrenti RNN ed è per ovviare a questo problema che sono state sviluppate le reti di memoria a lungo termine LSTM.[18]

Come gli RNN, anche gli LSTM utilizzano unità ricorrenti per apprendere dai dati della sequenza, ma ciò che le distingue è il contenuto di queste unità. Negli RNN standard vengono eseguite solo due operazioni principali: combinare lo stato nascosto precedente con il nuovo input e passarlo attraverso la funzione di attivazione (Figura 1.11).

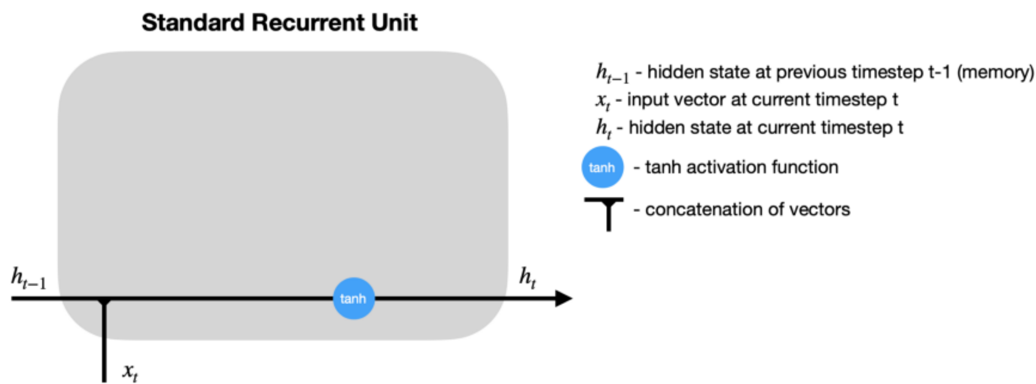


Figura 1.11 Composizione unità ricorrente standard

Dopo che lo stato nascosto è stato calcolato al timestep t, viene passato all'unità ricorrente e combinato con l'input al timestep t+1 per calcolare il nuovo stato nascosto al timestep t+1 e così via per un numero predefinito (n) di passaggi temporali.[19]

D'altro canto, invece, l'unità ricorrente dell'LSTM è molto più complessa e richiede di conseguenza più computazioni (Figura 1.12). Questa, infatti, è dotata di tre porte per ogni cella, pertanto, con l'aggiunta di più pesi allenabili per cella, gli LSTM possono apprendere relazioni tra eventi che si verificano a milioni di intervalli di tempo di distanza.[28] Innanzitutto, tutte le elaborazioni vengono eseguite sulla combinazione tra lo stato nascosto al timestep precedente h_{t-1} e l'input del timestep corrente x_t. In ordine, la forget gate controlla quali informazioni dovrebbero essere dimenticate; poiché la sigmoide è compresa tra 0 e 1, imposta quali valori della cella devono essere scartati (moltiplicati per 0), ricordati (moltiplicati per 1) o parzialmente ricordati (moltiplicati per un valore compreso tra 0 e 1). La seconda porta è quella di input che si occupa di identificare gli elementi importanti che vanno aggiunti allo stato della cella. Lo stato della cella aggiornato c_t è dato dalla moltiplicazione tra lo stato della cella precedente c_{t-1} e l'uscita della forget gate, a cui viene aggiunto il risultato della moltiplicazione tra la porta di input e lo stato della cella candidato. Infine, lo stato nascosto h_t è ottenuto dalla moltiplicazione tra lo stato della cella aggiornato, passato attraverso la funzione di attivazione tanh, e il risultato della porta di uscita. L'ultimo stato della cella e lo stato nascosto tornano nell'unità ricorrente, il

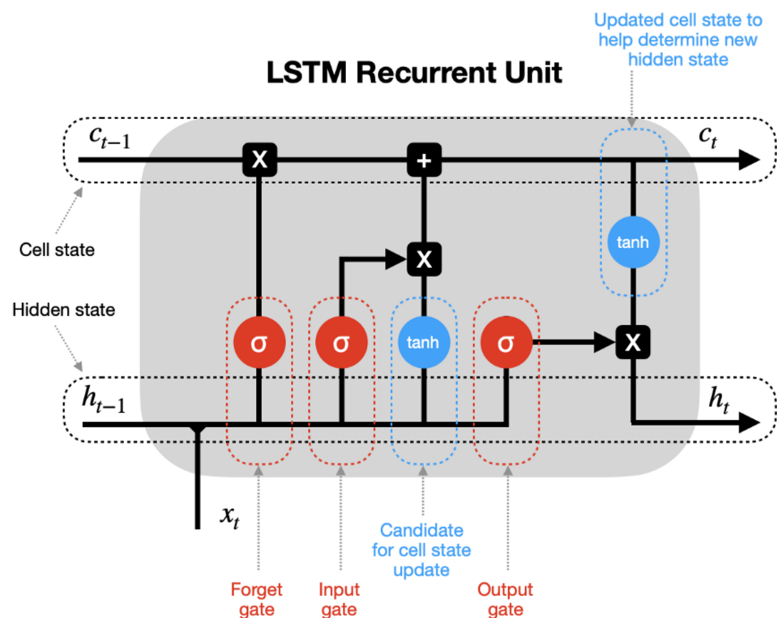


Figura 1.12 Struttura interna di una cella LSTM

processo si ripete al timestep t+1 e il ciclo continua fino a raggiungere la fine della sequenza.[19] In Figura 1.13 viene rappresentato il dettaglio di tutti i parametri che compongono una singola cella LSTM: 4 associati al vettore di input, 4 associati allo stato nascosto e infine 4 diversi parametri di bias.[31]

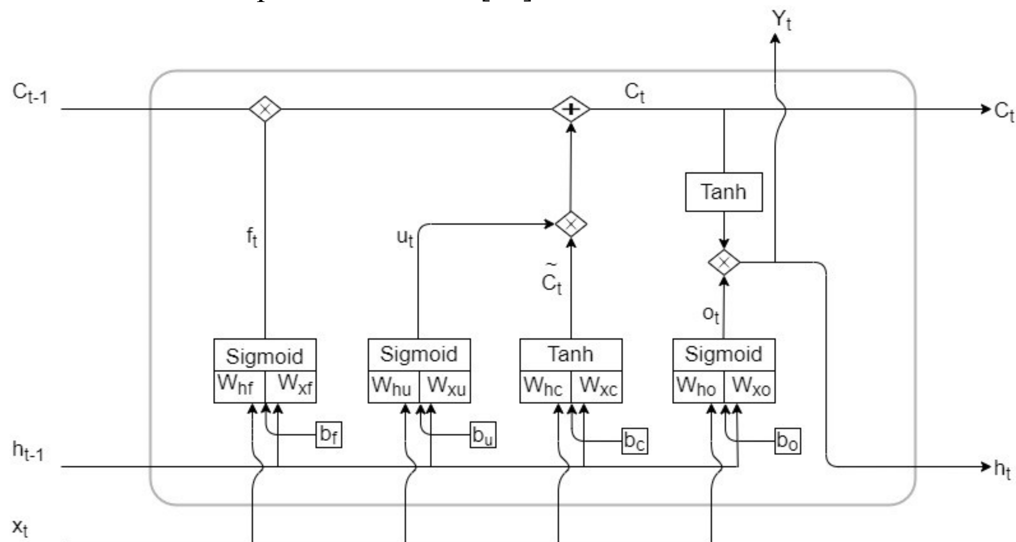


Figura 1.13 Struttura dettagliata dell'unità LSTM in termini di pesi e bias

Gli LSTM sono in genere progettati con più layers. Uno strato superiore denso costituito da uno strato di rete neurale completamente connesso, di solito con lo stesso numero di unità del numero di classi da rilevare, al quale in uscita viene applicata una funzione di attivazione softmax al fine di determinare la classe prevista della rete neurale.[28]

1.2.5 *TCNN: Rete Neurale Convoluzionale Temporale 1D*

Le Reti Neurali Convoluzionali Temporali (TCNN), rappresentano un'alternativa estremamente promettente alle architetture ricorrenti, comunemente utilizzate in un'ampia gamma di attività di modellazione di sequenze, tra cui modellazione linguistica e traduzione automatica. Sebbene le TCNN evidenzino alcuni potenziali svantaggi, come richiedere più memoria rispetto a RNN durante l'inferenza, esse presentano diversi vantaggi che possono essere sfruttati durante la progettazione e la formazione della rete, come, ad esempio, parallelismo intrinseco da sfruttare per accelerare il calcolo e adattabilità a diversi domini che richiedono diversi requisiti di memoria.[20]

L'architettura di rete è composta da 9 layers: 6 convoluzionali e 3 completamente connessi (FC). Nelle tabelle I.III e I.IV viene riportata una sintesi degli strati che compongono la rete. L'output consiste in 3 nodi a cui viene applicata la funzione softmax per convertire i valori ad una predizione probabilistica di quanto è ottimo il risultato.

Layer	*Features*	*Kernel*	*Pool*
1	256	7	3
2	256	7	3
3	256	3	NA
4	256	3	NA
5	256	3	NA
6	256	3	3

TABELLA I.III Parametri dei primi 6 livelli di rete convoluzionali

Layer	*Output Units*
7	1024
8	1024
9	3

TABELLA I.IV Parametri degli ultimi 3 livelli di rete FC

I livelli convoluzionali funzionano sulle matrici facendo passare un kernel scorrevole su di essa. Nello strato convoluzionale 1D, il filtro copre l'intera riga della matrice di input in una volta (Figura 1.14). Nelle applicazioni di elaborazione del linguaggio, *natural language processing* NLP, il filtro si muove lungo un flusso di caratteri che arrivano in un ordine sequenziale nel tempo, è uno strato convoluzionale temporale.[21]

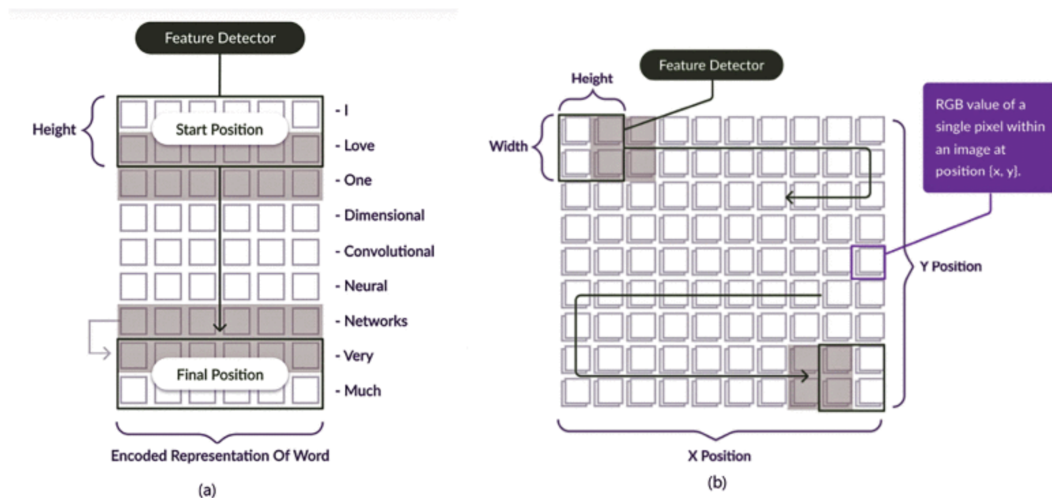


Figura 1.14 Differenza tra convoluzione 1D e standard

Capitolo 2
ACCELERAZIONE HARDWARE

Le reti neurali profonde, nonostante la semplicità delle operazioni da eseguire quali moltiplicazioni, addizioni, passaggio attraverso filtri e semplici funzioni non lineari, hanno una numerosità di computazioni e una complessità di rete molto elevata. Nello specifico, ad impattare nella topologia di rete sono numero e tipo dei livelli, dimensione e numero dei filtri, numero dei canali e numero dei pesi presenti nel modello. Ad esempio, decine di livelli possono richiedere l'elaborazione di un numero di pesi nell'ordine delle centinaia di milioni e un numero di decine di miliardi di operazioni di tipo MAC per ogni immagine elaborata. Al fine di offrire l'elevata potenza di calcolo richiesta dalle reti neurali, è necessario sfruttare unità di elaborazione hardware per accelerarne i calcoli.[22] L'accelerazione hardware si basa sull'idea di utilizzare hardware appositamente progettato per accelerare e aumentare l'efficienza durante l'elaborazione di attività specifiche. Pertanto, nel caso delle reti neurali si tratta di realizzarle a livello hardware al fine di ridurre i costi di sistema, ottimizzando le risorse necessarie, diminuendo i requisiti di alimentazione e migliorando al contempo le prestazioni.

2.1 **Dall'IoT all'Edge Computing**

Come visto l'apprendimento automatico è costituito da due fasi, addestramento o training e inferenza o inference. Mentre la fase di addestramento, che richiede una grande quantità di potenza di calcolo, viene solitamente eseguita nel cloud, per la fase di inferenza, nella maggior parte dei casi, non vale lo stesso principio. L'inferenza, infatti, è la distribuzione dei modelli addestrati per ottenere informazioni dettagliate sui flussi di dati del mondo reale. Nel nuovo scenario IoT, la capacità richiesta ai dispositivi è che siano in grado di elaborare i dati raccolti in modo tale da prendere decisioni intelligenti in tempo reale. Pertanto, il cloud computing mostra molte carenze per questo tipo di utilizzo. A causa della distanza tra la posizione fisica del server cloud e il dispositivo terminale, infatti, il modello di cloud computing di elaborazione e archiviazione centralizzata dei dati deve affrontare problemi in termini di latenza, larghezza di banda e consumo energetico.[23] Con molti settori che adottano rapidamente l'intelligenza artificiale per ottenere informazioni dettagliate sui dati in aumento da miliardi di dispositivi connessi, combinata con una richiesta di bassa latenza, c'è una spinta crescente per spostare

l'inferenza più vicino al luogo in cui vengono creati i dati, parlando quindi di edge computing o inference at the edge. Così facendo, le dipendenze di rete, rispetto a un'implementazione di elaborazione centralizzata, si riducono, portando ad un aumento delle prestazioni e ad una riduzione della latenza del carico di lavoro o dell'applicazione. Il concetto di edge computing è quello di fornire sufficienti capacità di calcolo e archiviazione nelle aree locali, in modo tale da mitigare l'effetto del throughput e del ritardo della rete.[24]

2.2 **Strategie Implementative per l'Ottimizzazione**

L'esecuzione degli algoritmi di AI all'edge pone però dei limiti derivanti dai dispositivi IoT stessi che presentano una quantità di risorse limitate in termini di memoria, risorse computazionali ed energetiche. A questo proposito, è importante l'utilizzo di strategie implementative al fine di ottimizzare il più possibile la rete a livello sia software sia hardware. Dal punto di vista hardware, uno dei metodi di ottimizzazione fondamentali si concentra sul parallelismo. Questo, sfrutta la caratteristica delle NN di essere altamente parallelizzabili, utilizzando architetture hardware in grado di eseguire più operazioni contemporaneamente.[24] Ad esempio, in una rete convoluzionale generica con un singolo input la quantità delle operazioni può arrivare fino a decine di milioni pertanto, eseguirle contemporaneamente riduce la latenza in modo notevole.[24] La piattaforma hardware potrebbe però non essere in grado di eseguirle tutte in un unico ciclo, pertanto, è importante focalizzare l'attenzione sulla lettura/scrittura dei dati in memoria che deve avvenire in queste fasi intermedie. A differenza della memoria on-chip, l'accesso alla memoria off-chip significa maggiore latenza e maggiore consumo energetico, ma essendo dispositivi con risorse limitate molto spesso le prime non sono sufficienti per contenere tutte le operazioni di calcolo quindi l'utilizzo di memoria esterna diventa inevitabile. In questo contesto, entra in gioco un altro aspetto importante per l'ottimizzazione ovvero, l'attenzione verso il riutilizzo dei dati in modo tale da ridurre il più possibile gli accessi in memoria e quindi gli svantaggi che ne derivano.[25]

Altri metodi si concentrano invece sulla compressione del modello di rete neurale al fine di ridurre la complessità di calcolo e archiviazione. Tra questi troviamo la quantizzazione dei pesi e delle attivazioni che dal formato in virgola mobile, vengono sostituiti con quello a virgola fissa e con un numero di bit inferiore. Questo, da un lato aiuta a ridurre la larghezza di banda e i requisiti di archiviazione del sistema di elaborazione della rete neurale, dall'altro l'utilizzo di una rappresentazione semplificata riduce il costo dell'hardware per ciascuna operazione.[25]

Oltre a ridurre la larghezza di bit di attivazione e pesi, un altro metodo consiste nel ridurre proprio il numero di pesi. Ad esempio, approssimando la matrice dei pesi con una

rappresentazione di rango più basso o utilizzando la tecnica della potatura che rimuove tra i pesi gli zeri o i valori assoluti molto piccoli.[25]

2.3 **Confronti tra Dispositivi Hardware per l'Inferenza Edge**

L'inferenza sull'edge può essere eseguita su una gamma di dispositivi hardware al silicio, i principali sono Central Processing Unit (CPU), Graphic Processing Unit (GPU), Application Specific Integrated Circuits (ASIC) e Field Programmable Gate Arrays (FPGA). In questa sezione viene effettuato un confronto tra queste piattaforme.

- CPU: è un processore per uso generico che offre un'elevata flessibilità ed è in grado di eseguire diversi carichi di lavoro. Tuttavia, la flessibilità ha un costo. Le CPU, basate sull'architettura Von Neumann, eseguono le istruzioni in modo sequenziale, pertanto, eseguiranno spesso specifici compiti complessi molto più lentamente di un hardware dedicato come ASIC e FPGA. Inoltre, il grande sovraccarico dovuto allo spostamento di dati e istruzioni in un'architettura generica, rende le CPU relativamente inefficienti e affamate di energia. Sebbene la maggior parte dei lavori di inferenza all'edge oggi vengano eseguiti utilizzando CPU a causa della loro flessibilità e facilità d'uso, le loro scarse prestazioni e i profili di potenza elevata le rendono non ideali per applicazioni di inferenza edge di *deep learning mission-critical*.[26]
- GPU: sono costituite da molti core piccoli e specializzati che funzionano in parallelo offrendo un throughput elevato rispetto a una CPU. In origine, avevano lo scopo di scaricare attività ad alta intensità di calcolo altamente ripetitive come il rendering grafico dalle CPU. Nel tempo, queste hanno trovato impiego nelle applicazioni di deep learning, in particolar modo nella fase di addestramento grazie alle loro architetture ad elevato parallelismo e all'elaborazione dei dati in batch. Sebbene eccellano in questa fase, non si può dire lo stesso per la fase di inferenza nelle condizioni imposte dalle reti perimetrali. I dati infatti vengono spesso trasmessi in streaming dai dispositivi finali (fotocamere, sensori e dispositivi IoT) quindi i requisiti imposti in termini di latenza non consentono il *batching* dei dati di input, ma richiedono invece un'elaborazione in tempo reale. Una GPU, quindi, svolge un lavoro scadente nella gestione del flusso di dati e offre prestazioni inferiori a quelle di un FPGA, in cui i dati in entrata e in uscita dai motori di elaborazione possono essere facilmente convogliati utilizzando la logica personalizzata. Inoltre, come le CPU, anche le GPU sono dispositivi affamati di energia e generano molto calore. Mentre per l'addestramento e l'inferenza basati su cloud non risulta un problema importante in quanto è disponibile l'accesso a un'alimentazione costante e a meccanismi di raffreddamento adeguati, lo è invece per l'ambiente periferico. Qui, infatti, il montaggio di sistemi di raffreddamento (quali

ventole) è costoso sia in termini di costi che di proprietà. Inoltre, un data center opera in un ambiente controllato in cui la temperatura e altri parametri sono rigorosamente regolati. Il dispositivo edge al contrario è immerso nel mondo reale, in cui l'hardware potrebbe essere soggetto a condizioni meteorologiche estreme come calore, polvere e umidità che potrebbero influire negativamente sulle prestazioni e sulla funzionalità delle GPU. La loro durata inoltre è abbastanza breve, da 3 a 5 anni, pertanto, l'eventuale sostituzione con questa frequenza nel dispositivo finale non è conveniente.[26]

- ASIC: con CPU e GPU che offrono elevata flessibilità e basse prestazioni, gli ASIC si collocano all'estremo opposto. Questi dispositivi, infatti, sono personalizzabili specificamente per un'applicazione finale che offre prestazioni elevate con un consumo energetico minimo, tali vantaggi però hanno un costo non indifferente. Primo fra tutti è il fatto che una volta progettato, non può più essere modificato senza doverlo rifare completamente, richiedendo quindi uno sforzo ingegneristico non indifferente. Inoltre, i costi NRE iniziali e il tempo di sviluppo necessari per sviluppare un ASIC sono molto più elevati rispetto a quelli di altri tipi di processore. Il volume del chip, in termini di numeri da produrre, deve poi essere sufficientemente elevato da giustificare i costi iniziali di ricerca e sviluppo. Con i modelli ML ancora agli inizi e in rapida evoluzione, non è sempre possibile impegnarsi in un'unica architettura che può essere rilevante fino al prossimo ciclo di progettazione ASIC. Con un'ampia gamma di applicazioni di inferenza che richiedono una gamma di soluzioni, i costi associati sostenuti per lo sviluppo di più ASIC aumenteranno esponenzialmente. Il silicio programmabile al contrario offre la possibilità di aggiornare lo stesso hardware al volo senza dover cambiare l'intera scheda ogni volta che c'è un aggiornamento del modello. A differenza degli ASIC, possono essere facilmente adattati alle applicazioni in evoluzione semplicemente eseguendo un aggiornamento del software senza dover passare attraverso un processo costoso e dispendioso in termini di tempo.[26]

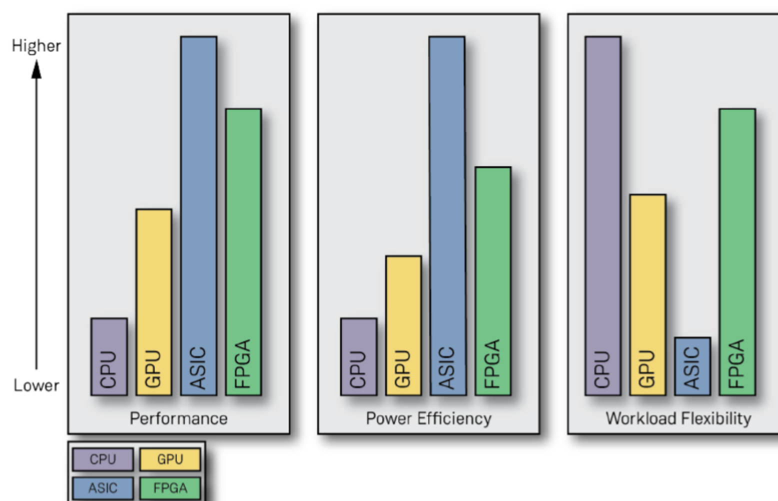


Figura 2.1 Confronto prestazioni, performance e flessibilità tra CPU, GPU e FPGA

È in questo contesto che il *Field Programmable Gate Array* (FPGA) risulta essere la miglior soluzione che bilancia performance, potenza e flessibilità per l'inferenza all'edge.

2.4 **Vantaggi sull'utilizzo dell'FPGA**

Un FPGA è costituito da milioni di celle logiche configurabili, elementi di memoria e unità aritmetiche collegate tramite interruttori programmabili. Il dispositivo può essere configurato per implementare qualsiasi funzionalità, applicando schemi creati dall'utente a ciascuna delle celle logiche e combinarli poi tra loro in modo selettivo attraverso gli interruttori.[26]

Gli FPGA in genere funzionano a frequenze di clock più basse rispetto a GPU o CPU, ma non significa che abbiano prestazioni inferiori. A differenza di queste, infatti, eseguono le istruzioni direttamente nell'hardware. In questo modo, viene eliminato gran parte del sovraccarico di elaborazione delle istruzioni associato ai processori per uso generico (GPP), offrendo latenze molto più basse. Inoltre, l'enorme array di celle logiche può essere efficacemente sfruttato per eseguire calcoli paralleli (a differenza dell'approccio seriale di un GPP). Ogni funzione logica, infatti, occupa una regione separata, permettendo ad ognuna di loro di poter essere eseguite contemporaneamente senza influenzarsi a vicenda. Ciò, offre all'FPGA un notevole aumento del throughput complessivo rispetto ai processori tradizionali.

Uno dei colli di bottiglia, sulla velocità di elaborazione e consumo di energia nel processo di gestione dei dati, è la larghezza di banda della memoria.[27] Nell'FPGA, invece, il progettista del programma può predisporre il calcolo per consentire ai dati di fluire attraverso di esso. In particolare, al termine di una fase di calcolo, può passare direttamente alla fase successiva senza inviare i dati alla memoria.[27] Pertanto, la personalizzazione del flusso di dati e di controllo nella logica specifica dell'applicazione si traduce in un notevole risparmio di tempo e consumo di energia.[26]

Gli FPGA sono dotati di uno storage su chip maggiore rispetto alle GPU, a tal fine si riducono tutti gli accessi non necessari alla memoria esterna migliorando sia la latenza che l'energia consumata.[26]

Un altro grande vantaggio dell'utilizzo degli FPGA per le applicazioni di inferenza ML è la loro capacità di supportare una grande varietà di tipi di dati. L'addestramento ML opera su aritmetica in virgola mobile (FP32) a precisione standard o doppia poiché richiede un'elevata precisione. L'inferenza ML, invece, a differenza dell'addestramento, può tollerare una significativa perdita di precisione e può operare con tipi di dati più compatti come INT8, INT16, FP15, FP24 o BF16, binari, ternari e persino tipi di dati personalizzati. Lavorare con aritmetica di precisione inferiore, riduce drasticamente i requisiti di alimentazione, spazio di archiviazione e larghezza di banda. Le GPU funzionano eccezionalmente bene con alcuni tipi di dati nativi

come FP32 e FP16, ma risentono delle prestazioni con tipi di dati personalizzati a bassa precisione.

Complessivamente, quindi, con il loro hardware riconfigurabile e l'architettura altamente parallela, gli FPGA sono altamente adatti all'inferenza edge.[26]

Capitolo 3
ESEMPI APPLICATIVI

In questo capitolo verranno descritti alcuni esempi di applicazioni IoT in cui l'FPGA è stato utilizzato come acceleratore hardware di reti neurali artificiali. Lo scopo di tali descrizioni è quello di fornire una dimostrazione a livello pratico su come l'FPGA rappresenti realmente una soluzione di successo per questa tipologia di applicazioni. Ciascun sottocapitolo contiene un esempio che riassume un singolo articolo scientifico, nell'ordine, rispettivamente [14], [17], [28] e [21] ([14], [28], [21] presi dai IEEE e [21] da mdpi). Il primo esempio (3.1 e [14]), utilizza l'FPGA come acceleratore hardware per implementare la rete neurale MTCNN la quale si occupa di effettuare il rilevamento facciale su immagini acquisite da una videocamera IoT. Il secondo esempio (3.2 e [17]), implementa la rete neurale Tiny-YOLOv3 su un FPGA di basso costo e bassa potenza al fine di effettuare il riconoscimento di oggetti per applicazioni IoT. Il terzo esempio (3.3 e [28]) implementa su FPGA una rete neurale di tipo LSTM che esegue il processo di inferenza tramite i dati raccolti dai sensori IoT perimetrali posti su un macchinario al fine di individuare, prima che si verifichino, eventuali guasti presenti sullo stesso. Infine, il quarto esempio (3.4 e [21]) implementa su un FPGA di fascia bassa la rete neurale TCNN 1D che esegue il riconoscimento del testo per dispositivi IoT.

3.1 **Rilevamento e Allineamento Facciale Ottimizzato**

Nella maggior parte delle applicazioni che richiedono l'elaborazione delle immagini facciali (riconoscimento facciale, analisi dell'espressione, ricostruzione facciale 3D ecc..), il rilevamento e l'allineamento dei volti è un compito essenziale e fondamentale da eseguire. Nell'architettura per IoT qui descritta, quest'ultima operazione viene calcolata direttamente all'interno del dispositivo edge; in questo modo è possibile evitare di trasmettere enormi quantità di dati video grezzi attraverso la rete IoT che ha larghezza di banda limitata, aumentando quindi l'efficienza complessiva del sistema.

Le immagini acquisite dal mondo reale, a causa delle grandi variazioni degli angoli dell'immagine, delle condizioni di illuminazione dello sfondo e degli oggetti di blocco intermedi, rendono il rilevamento e l'allineamento dei volti compiti complessi e impegnativi.

L'utilizzo della GPU come acceleratore sappiamo poter richiedere un budget di alimentazione elevato e troppo costoso per essere incorporato all'interno dei dispositivi IoT. D'altro canto, i dispositivi basati su FPGA presentano un vantaggio nell'elaborazione dei dati, fornendo un'accelerazione hardware ad alte prestazioni, basso consumo energetico ed efficienza nei costi.

Nell'esempio riportato viene presentata la progettazione ed implementazione di un MTCNN (sottocapitolo 1.2.2) su una piattaforma di calcolo a basso costo e a bassa potenza. Viene utilizzato un chip Xilinx ZYNQ contenente un processore ARM, DSP dedicati, RAM e FPGA ZYNQ della famiglia Artix 7. Il modello di FPGA specificato appartiene ad una famiglia di dispositivi a fascia bassa in linea con i requisiti richiesti ad un dispositivo edge a basso costo e bassa potenza. Nello specifico l'FPGA contiene 13.300 sezioni logiche, una BRAM da 630 kB e 220 sezioni DSP; viene poi condivisa con il processore una SDRAM DDR da 512 MB attraverso l'interfaccia AXI. Mentre tutte le operazioni vengono per l'appunto eseguite tramite il processore ARM, le porzioni ad alta intensità di calcolo vengono implementate nell'FPGA realizzando così l'accelerazione del MTCNN. Il core ARM, infatti, controlla il flusso computazionale complessivo, dalla ricezione dell'immagine in ingresso della telecamera all'invio di segnali di avvio e istruzioni alla logica programmabile. Nelle porte logiche FPGA, infatti, vengono eseguiti e quindi accelerati i calcoli per l'MTCNN. Quando le funzioni MTCNN vengono chiamate nella funzione principale, un segnale di avvio e bit di istruzione vengono inviati alla parte logica programmabile.

3.1.1 *Algoritmo*

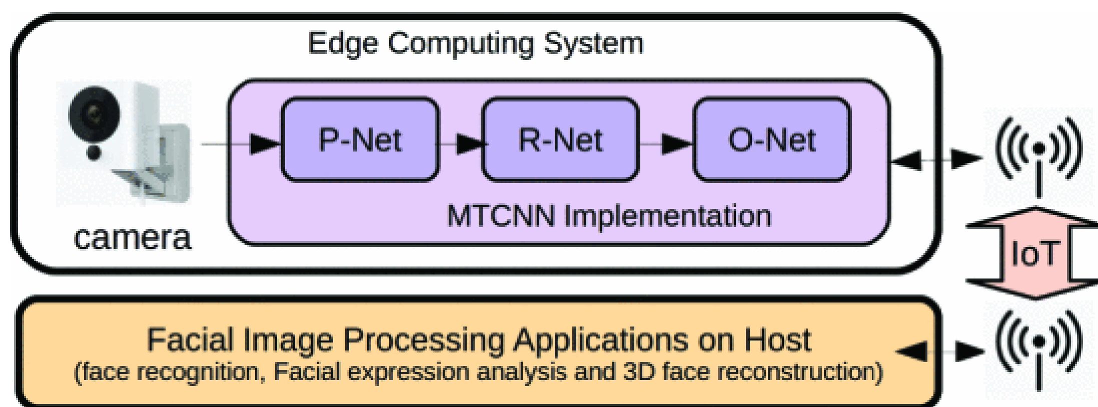


Figura 3.1 Sistema di elaborazione edge AI per applicazioni di elaborazione di immagini facciali acquisite da telecamera

Dato che i livelli lavorano in cascata, l'output di ciascun livello fornisce l'input al livello successivo. Essendo richiesta un'elaborazione in tempo reale, ogni livello ha pertanto un numero massimo di riquadri di delimitazione da poter fornire in uscita e quindi all'ingresso del livello successivo, in modo tale da riuscire a mantenersi al di sotto di un tempo di elaborazione

ed esecuzione accettabile (Tabella III.I). Nello specifico, da P-Net ed R-net possono uscire rispettivamente un massimo di 40 e 20 riquadri di delimitazione. Pertanto, la rete è in grado di rilevare un massimo di 20 volti.

Tasks	P-Net	R-Net	O-Net	Etc
Func[1]	Conv2D	Conv2D and FC	Conv2D and FC	Pre and Post
Time[2] (ms)	*33.9*	*133.6*	*21.2*	*23.3*

TABELLA III.I Risultati del tempo di esecuzione impiegato dalle diverse fasi che compongono l'algoritmo

3.1.2 *Architettura del sistema*

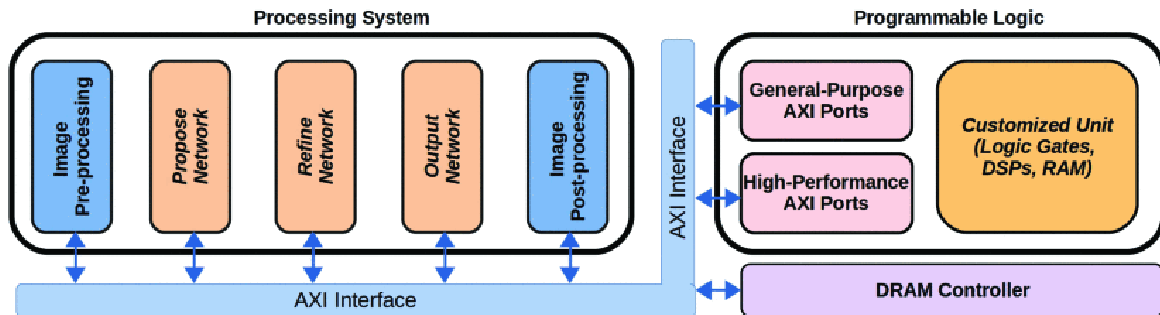


Figura 3.2 Architettura di sistema complessiva mappata sul sistema di elaborazione ARM e sulla logica programmabile. Entrambe condividono una memoria DDR tramite interfacce AXI e il controller di memoria DDR

La funzione principale viene eseguita sul processore ARM mentre il calcolo della rete neurale viene accelerato sulla logica programmabile. In particolare, l'FPGA viene utilizzato per elaborare operazioni di moltiplicazione e accumulazione (MAC) in parallelo. Il processore ARM invece calcola l'elaborazione pre-immagine per estrarre un'immagine in scala che viene alimentata nella rete al fine di produrre l'immagine di output con i volti e i relativi punti di riferimento facciali. Le tre parti indipendenti e sequenziali dell'MTCNN richiedono il funzionamento a logica programmabile tramite l'interfaccia AXI. Ciascuna rete invia un segnale di avvio, insieme al comando e agli indirizzi di memoria del peso e ai dati dell'immagine in ingresso, alla logica programmabile. All'inizio del calcolo, la logica programmabile invia richieste di lettura in memoria tramite l'interconnessione AXI al fine di ottenere i pesi e i dati dell'immagine in ingresso dalla DRAM. Dopo che i calcoli sono stati completati, viene inviata una richiesta di scrittura al processore ARM per memorizzare i risultati.

La memoria ad accesso casuale a blocchi (BRAM) su chip viene utilizzata per memorizzare tutti i risultati intermedi. Ogni rete ha un minimo di 3 livelli di convoluzione2D e le ultime due hanno anche uno strato *Fully Connected* (FC) da eseguire, la stima della quantità di memoria occupata da ciascuna di esse, con i rispettivi calcoli intermedi, è contenuta in Tabella III.II Si evince pertanto che è possibile salvare i risultati di un livello, necessari poi al successivo della stessa rete, nella memoria ad accesso casuale a blocchi (BRAM) su chip. Così facendo, è

possibile evitare di scrivere e leggere i risultati intermedi da e verso la DRAM esterna migliorando le prestazioni e l'efficienza energetica del sistema.

Network[1]	P-Net	R-Net		O-Net	
Function[2]	Conv2D	Conv2D	FC	Conv2D	FC
Memory[3] (kB)	*1.57*	*16.62*	*2.81*	*93.31*	*5.63*

TABELLA III.II Requisiti BRAM su chip in ogni strato

3.1.3 *Implementazione logica Programmabile*

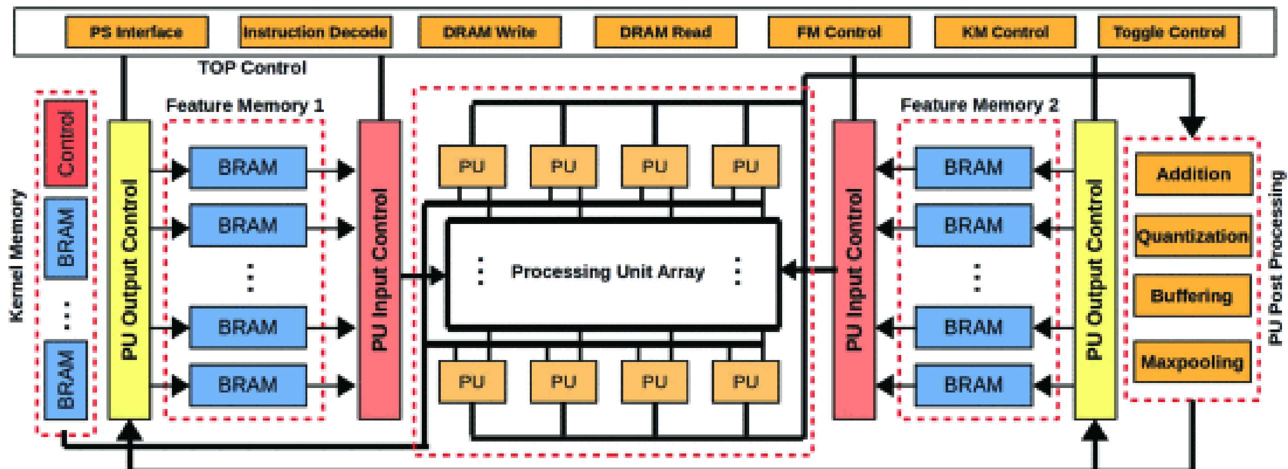


Figura 3.3 Architettura generale della logica programmabile. Dopo aver caricato tutti i dati dalla DRAM, vengono avviate le operazioni MTCNN. I risultati finali e un segnale fatto vengono inviati al processore attraverso l'interfaccia AXI

È costituita da 5 componenti:

- Blocco di controllo principale;
- Proccessing unit array (PU);
- Due blocchi di memoria on chip BRAM per le FM;
- Memoria di kernel con blocco di controllo.

A seguito del segnale di start, viene ricevuta l'istruzione di input del blocco di controllo principale la quale, è costituita da 89 bit e strutturata come in Figura 3.4.

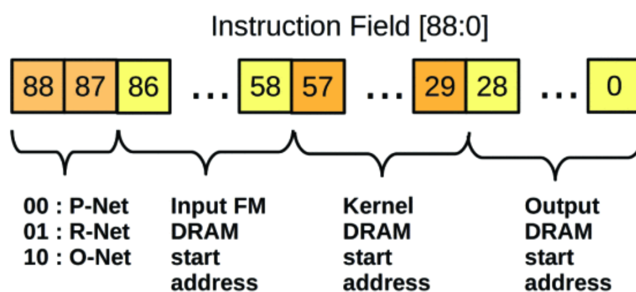


Figura 3.4 Campo di istruzione dal processore

Due bit sono assegnati ad indicare quale delle 3 reti deve essere attivata (P-net, R-net, O-net); i restanti bit divisi a gruppi di 29 indicano rispettivamente gli indirizzi di memoria DRAM nella quale sono contenuti: la mappa di input, i dati del kernel e dove destinare l'output. Il blocco di controllo decodifica l'istruzione, controlla gli input PU e i blocchi di controllo output,

alimentando la memoria on chip kernel e quelle delle funzionalità. In particolare, la PU esegue operazioni MAC in parallelo e invia i risultati al blocco di post-elaborazione della PU. Quel blocco calcola i risultati intermedi per il livello completamente connesso (FC), esegue la quantizzazione e il buffering dei dati e implementa il max-pooling. Quindi, i risultati finali vengono memorizzati nella BRAM tramite il blocco di controllo di output PU. Al termine dell'elaborazione della rete, il segnale di fine viene inviato al processore con gli indirizzi di memoria in cui si trovano i risultati. Quindi, la pre e post elaborazione vengono eseguite e inizia l'elaborazione della rete successiva.

La Figura 3.5 mostra l'operazione di controllo delle due memorie di funzionalità BRAM. Viene utilizzato uno schema di commutazione che alterna l'archiviazione dei dati intermedi tra due layer successivi all'interno della stessa rete. Se ad inviare i dati all'interno della Processing Unit è la memoria 1 allora i dati in uscita dalla PU verranno archiviati nella memoria 2 e viceversa. Il blocco di controllo di input e output PU genera indirizzi BRAM per archiviare e inviare dati, da e verso BRAM, in base alle diverse dimensioni dei kernel e al numero di canali in ogni fase della rete.

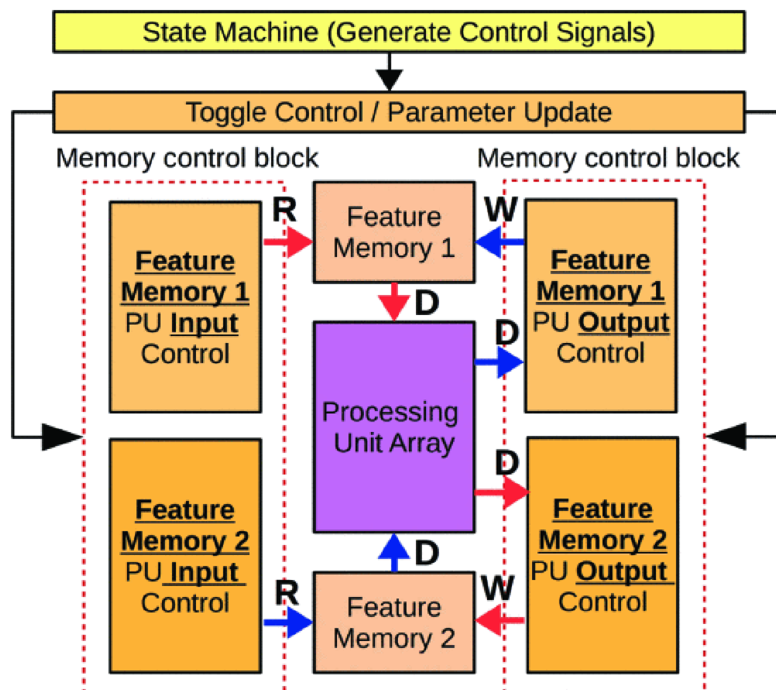


Figura 3.5 Operazioni di controllo della memoria

3.1.4 *Struttura della PU*

Ogni PU contiene all'interno quattro 2-MAC e una 1-MAC. Nelle conv2D dell'MTCNN vengono usate due tipologie di matrici kernel 2x2 e 3x3. Dal momento che le matrici kernel 2x2 vengono usate una sola volta prima dei livelli *Fully Connected*, nell'R-net e nell'O-net, si è cercato di focalizzare l'ottimizzazione parallelizzando le operazioni svolte con il kernel 3x3

che, contrariamente alle prime, vengono usate con una maggiore frequenza. Nove funzioni e dati del kernel vengono moltiplicati e i risultati della moltiplicazione vengono sommati in una PU. Il blocco di controllo dell'ingresso della PU invia la mappa delle caratteristiche 3 × 3 e i dati del kernel alla PU utilizzando i parametri predefiniti, in base alle informazioni di rete nelle istruzioni. Le unità MUX mostrate in Figura 3.6 sono utilizzate per elaborare gli strati completamente connessi (FC) in R-Net e O-Net. Ad esempio, il risultato dell'addizione precedente dal flip-flop viene immesso nel sommatore e l'operazione di accumulazione viene eseguita continuamente per sommare i risultati della moltiplicazione fino al raggiungimento del numero predefinito di operazioni MAC.

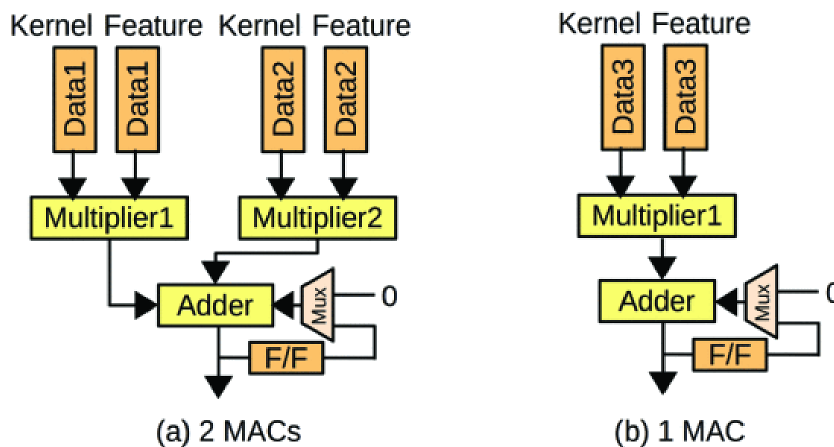


Figura 3.6 Architettura degli elementi di elaborazione- una PU contiene quattro 2-MAC (a) e una 1-MAC (b). Nel caso di un conv2D con un kernel 3 × 3, tutti i calcoli del kernel 3 × 3 vengono elaborati contemporaneamente. Nel caso di un conv2D con kernel 2 × 2, due kernel 2 × 2 vengono elaborati utilizzando 8 MAC in parallelo in una PU.

3.1.5 *Macchina a Stati Finiti*

La macchina a stati gestisce tutti i blocchi di sottocontrollo. Ci sono quattro macchine a stati: una per il blocco superiore e tre per le reti MTCNN. La macchina a stati principale controlla l'avvio delle tre macchine a stati per i tre livelli della rete e gestisce l'interfaccia con il processore. Dopo che il segnale di avvio per P-Net e l'istruzione arrivano alla macchina di stato superiore dal processore, lo stato della macchina di stato superiore viene spostato dallo stato di idle P-Net, allo stato di esecuzione. Una volta completata la procedura P-Net, la macchina a stati superiore va allo stato P-Net fatto e attende il segnale di avvio e l'istruzione di R-Net. Quando è stato ricevuto, l'operazione R-Net viene avviata e lo stato della macchina a stati superiori passa da R-Net idle allo stato di esecuzione. Quando tutte le operazioni R-Net sono terminate, la macchina a stati superiore passa dall'esecuzione R-Net allo stato R-Net terminato e attende l'arrivo del segnale di avvio e dell'istruzione O-Net. Dopo che tutta l'elaborazione O-Net è stata completata, i risultati finali vanno al processore e lo stato della macchina a stati superiore passa dall'esecuzione O-Net a idle e attende un nuovo segnale di avvio e istruzione per la P-Net.

3.1.6 *Risultati e Conclusioni*

Slice LUTs	Slice Registers	BRAM	DSPs
41644 (78.28%)	*66489 (62.49%)*	*62(44.28%)*	*216(98.18%)*

TABELLA III.III Utilizzo delle risorse FPGA delle unità di elaborazione

Nella Tabella III.III sono mostrate le percentuali di risorse utilizzate dell'FPGA. Avendo implementato 24 PU (unità di elaborazione) contenenti ognuna 9 moltiplicatori e 4 accumulatori, vengono usati quasi tutti i DSP (24 × 9 = 216) 98,18% per elaborare i calcoli della CNN in parallelo. Inoltre, l'utilizzo delle LUT slice è elevato (78,28%) poiché nei blocchi di controllo di input e output della PU la logica di generazione degli indirizzi è significativamente complessa. Infine, l'utilizzo della BRAM è del 44,28% perché i risultati intermedi vengono archiviati nella BRAM e le FIFO basate su BRAM vengono utilizzate per caricare e archiviare i dati dalla DRAM.

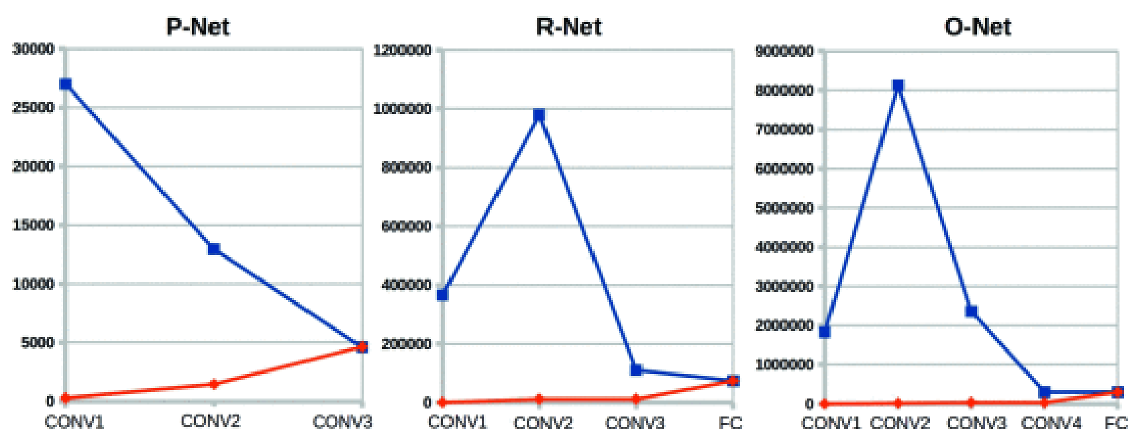


Figura 3.7 Numero di parametri e addizioni multiple (Madds): l'asse X indica i livelli CNN e l'asse y indica il numero di parametri e Madds. La linea blu indica il numero di Madds e la linea rossa il numero di parametri.

La Figura 3.7 mostra il numero di parametri (linea rossa) e le addizioni multiple (linea blu) di ciascuna rete. La frequenza di clock di sistema è 125 MHz per un periodo di 8 nsec. Il P-Net richiede 2,16 ms per elaborare i tre livelli CNN. In ciascun livello vengono calcolate rispettivamente 27.000, 12.960 e 4.608 moltiplicazioni per un totale di 44.568 MAC. Considerando che nella p-net un volto viene rilevato attraverso le tre fasi precedenti utilizzando una patch immagine 12 × 12 × 3 e che una finestra dell'immagine 12 × 12 viene spostata di 2 pixel dall'angolo superiore sinistro all'angolo inferiore destro nell'immagine complessiva di dimensioni 224 × 224, il numero totale in cui vengono ripetute queste operazioni è 107 × 107 = 11.449. R-Net impiega 0,068 ms per elaborare quattro fasi nella rete. Nei tre livelli conv2D vengono eseguite rispettivamente 365.904, 979.776 e 110.592 moltiplicazioni mentre nel livello finale completamente connesso FC, vengono calcolate 73.728 moltiplicazioni. In R-Net vengono quindi calcolati un totale di 1,53 MMAC. Simile a R-Net, O-Net esegue cinque fasi e

impiega 0,573 ms per calcolare 12,9 MMAC. Il tempo di esecuzione di R-Net e O-Net è variabile in funzione del numero dei candidati della rete precedente, ma come detto in precedenza per ottenere un'elaborazione in tempo reale, il numero massimo di candidati da R-Net e O-Net è limitato rispettivamente a 40 e 20.

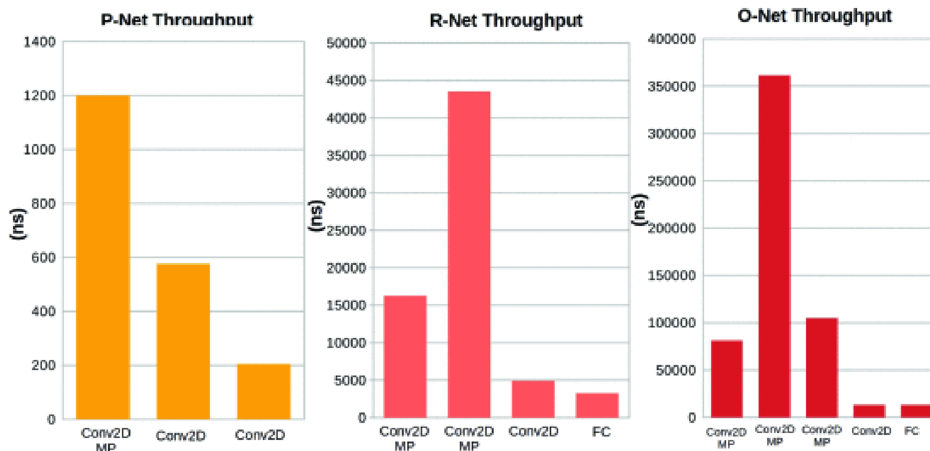


Figura 3.8 Throughput di P-Net, R-Net e O-Net. L'asse orizzontale elenca i livelli in ciascuna rete e l'asse verticale è la latenza.

Operating Frequency	FPS	Peak Performance	Power /Frame(J)
ARM:650MHz FPGA:125MHz	15.2	54 GOPS	0.28

TABELLA III.IV Risultati di performance, energia consumata e FPS

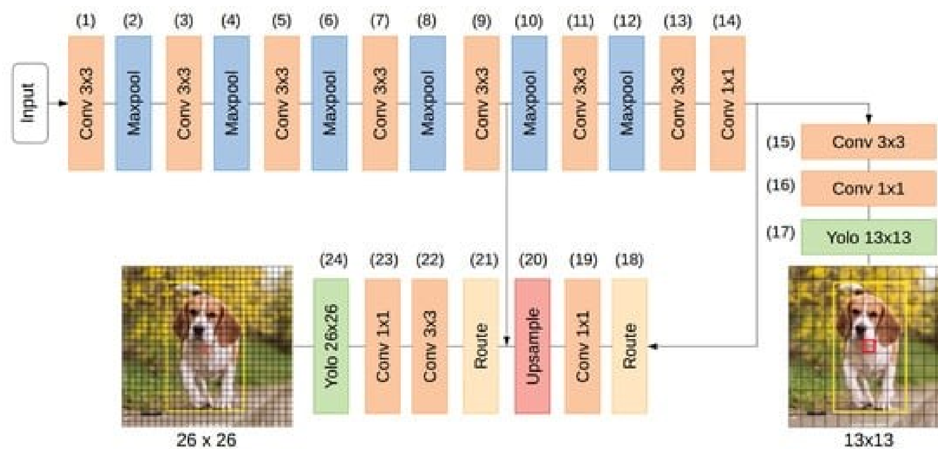
Il sistema di rilevamento e allineamento facciale appena descritto, ottimizzato e implementato su un chip a basso costo e a bassa potenza, dati i risultati ottenuti, (Tabella III.IV) dimostra che può essere utilizzato per varie applicazioni di elaborazione delle immagini facciali nei sistemi IoT. L'implementazione, infatti, raggiunge una velocità di trasmissione di 15,2 frame al secondo e consuma 2,76 volte meno energia rispetto a un'implementazione precedente dello stesso. Pertanto, il sistema ha un grande potenziale per essere adottato nei dispositivi IoT edge e contribuisce a migliorare le prestazioni complessive dei futuri sistemi IoT. [14]

3.2 **Riconoscimento Oggetti**

Il rilevamento degli oggetti è un'abilità essenziale per tantissime applicazioni, tra cui trasporto, sicurezza e ambito medico. Questa capacità sempre più spesso è necessaria direttamente sui dispositivi IoT perimetrali, con lo scopo di renderli in grado di prendere decisioni localmente e in un tempo sempre più limitato, aumentandone quindi l'efficienza. È possibile progettare i rilevatori di oggetti utilizzando le reti convoluzionali tramite due approcci: a due stadi (segue l'algoritmo tradizionale trovando prima le regioni di interesse identificandole come riquadri di delimitazione e in un secondo stadio le classifica) e a fase singola. I primi,

sono migliori in termini di precisione, ma allo stesso tempo computazionalmente più impegnativi. Pertanto, in uno scenario di edge computing dove la complessità è limitata dalle risorse disponibili altrettanto limitate, si utilizzano prevalentemente reti convoluzionali a fase singola (YOLO e versioni successive).

In questo studio viene proposto un rilevatore di oggetti che implementa su FPGA una rete a fase singola Tiny-YOLOv3 (sottocapitolo 1.2.3), in quanto offre flessibilità hardware, elevata potenza di calcolo, con altrettanta elevata efficienza energetica.



3.2.1 *Architettura del Sistema*

L'architettura utilizza l'accesso diretto alla memoria (DMA) per leggere e scrivere dati da e verso la memoria esterna principale, al fine di lasciare libera la CPU di eseguire altri compiti durante il trasferimento dei dati. I dati dalla memoria principale sono archiviati in memorie distribuite su chip. Ci sono tre serie di memorie nell'architettura: una per le mappe delle caratteristiche di input, una per le mappe delle caratteristiche di output e un'altra per i pesi dei kernel e i bias. Le memorie includono un insieme di generatori di indirizzi che inviano i dati in un ordine particolare ai core di calcolo da elaborare. Questi ultimi, sono organizzati in una matrice bidimensionale per consentire l'intra- e inter-parallelismo. Mentre i core nella stessa linea ricevono la stessa mappa delle caratteristiche di input ma kernel diversi, quelli nella stessa colonna ricevono gli stessi kernel ma mappe delle caratteristiche di input diverse. I risultati di ciascuna riga principale vengono scritti nei buffer di memoria di output, quindi trasferiti nella memoria principale tramite il DMA (Figura 3.9).

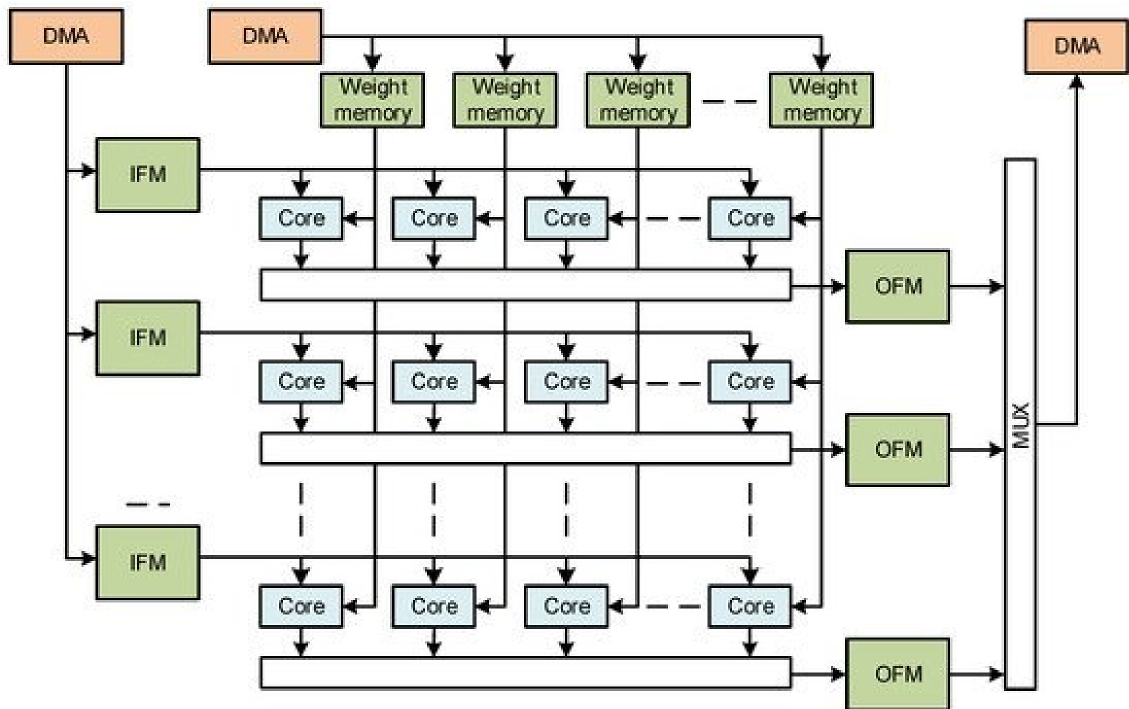


Figura 3.9 Schema a blocchi dell'acceleratore proposto della rete CNN di Tiny-YOLO

Il flusso di dati dell'acceleratore è suddiviso in tre fasi: lettura della memoria, calcolo e scrittura della memoria, come illustrato nella Figura 3.10.

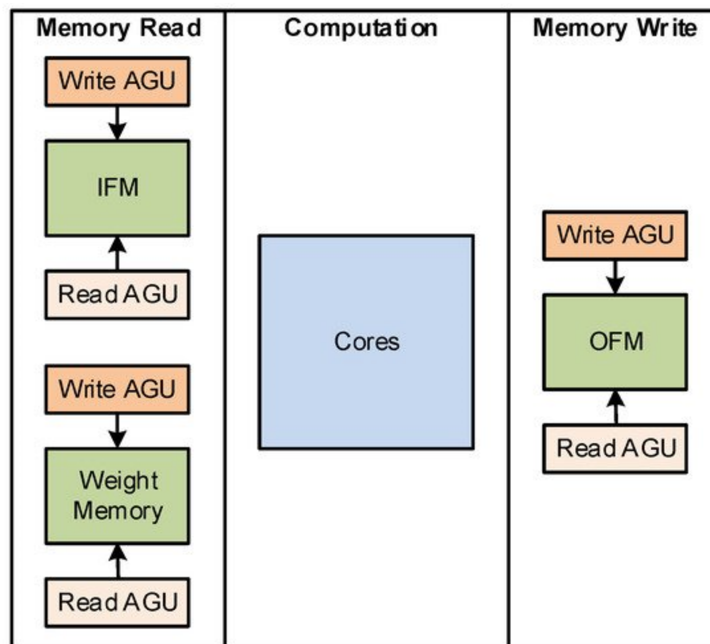


Figura 3.10 Flusso dei dati nell'acceleratore

Nella fase di lettura della memoria, i dati vengono trasferiti dalla memoria principale alle memorie IFM. In questa fase, le operazioni di lettura dalla memoria principale e le scritture nelle memorie, sono controllate da unità esterne di generazione di indirizzi (AGU).

Nella fase di calcolo l'IFM e i pesi vengono inviati sulla memoria on-chip. Qui vengono eseguiti i calcoli dei vari layers nei core personalizzati e al termine viene scritto il risultato nelle

memorie OFM. In questa fase, tutte le AGU interne controllano le letture dalle memorie IFM, le operazioni ai core e le scritture sulle memorie OFM.

Nella parte di scrittura della memoria, i dati dalle memorie OFM vengono ritrasferiti alla memoria principale. Un'AGU esterna controlla gli accessi in lettura alle memorie OFM e i trasferimenti in scrittura alla memoria principale.

Questa architettura consente la pipeline di flussi di dati consecutivi. L'esecuzione di un secondo flusso di dati inizia al termine della fase di lettura della memoria del primo flusso di dati. A questo punto, l'acceleratore esegue contemporaneamente la fase di lettura della memoria del secondo flusso di dati e la fase di calcolo del primo flusso di dati.

Ciascun core riceve in input i valori della mappa delle caratteristiche, i bias e i pesi delle memorie su chip inviando poi il risultato del calcolo alla memoria OFM. In aggiunta, richiedendo più parole di dati IFM e pesi del kernel, il core può essere configurato per calcolare più canali IFM in parallelo con un maggior numero di MAC (parametro nMAC). Questo, inoltre, è in grado di supportare più funzioni che possono essere selezionate durante il runtime, permettendo di accelerare quindi più tipologie di layers. Al risultato della funzione di convoluzione, che esegue la semplice somma delle uscite dei vari blocchi MAC che lo compongono, è possibile applicare delle funzioni di attivazione lineari e non. In particolare, le attivazioni non lineari sono implementate per il formato a virgola fissa e il core poi si occupa di shiftare il risultato finale in base al formato di quantizzazione utilizzato.

Nei layers da 1 a 8 viene disabilitato il bypass della convoluzione e attivato il maxpooling eseguendo quindi i calcoli in coppie convoluzionali + maxpool. I layers 10 e 12 essendo solo maxpool, bypassano la convoluzione, mentre le coppie 16–17 e 23–24 sono coppie convoluzionali + YOLO. Questi layers possono essere calcolati insieme poiché la convoluzione ha un'attivazione lineare e lo strato YOLO consiste nell'eseguire l'attivazione sigmoidea alle mappe delle caratteristiche di input. Pertanto, questi due tipi di strati possono essere combinati in una convoluzione con attivazione sigmoidea. Tutti i layers convoluzionali, inoltre, usano bias. Il core personalizzato è controllato da una AGU il cui periodo controlla il numero di operazioni richieste dalla funzione per ottenere il risultato. Per esempio, un maxpool richiede un periodo di 2x2=4, mentre la convoluzione con 3x3x16 kernels richiede 3x3x16/nMAC periodi in quanto ci sono nMAC canali di input che vengono eseguiti in parallelo. Il numero di uscite finali per l'esecuzione di una fase è quindi definito dalle iterazioni del generatore di indirizzi.

I parametri di configurazione dell'acceleratore sono i seguenti:

- nCols: numero di memorie di peso per i kernel;
- nRows: Il numero di memorie IFM utilizzate per le mappe delle caratteristiche di input;
- nCols x nRows: numero di core dell'array di calcolo;
- nMACs: numero di MAC in ciascun core;
- DDR_ADDR_W : imposta la larghezza dell'indirizzo di memoria principale;
- DATAPATH_W: determina la larghezza dei dati utilizzata per l'input e l'output di calcolo;
- MEM_BIAS_ADDR_W, MEM_WEIGHT_ADDR_W, MEM_TILE_ADDR_W e MEM_ADDR_W: dimensioni per polarizzazione, peso e memorie IFM e OFM

L'acceleratore proposto ha un'API (Application Program Interface, un intermediario software che permette a due applicazioni di comunicare) C++ che consente la configurazione di runtime da parte della CPU. La classe principale CODCore ha un attributo per l'indirizzo di base della memoria del core nel sistema. Oltre al costruttore, la classe ha i metodi run(), done() e clear(). Il primo segnala l'esecuzione di una fase del flusso di dati nell'acceleratore, il secondo verifica il completamento di run(), infine il terzo ripristina tutte le configurazioni.

La classe principale contiene tre oggetti, ciascuno di una classe diversa. Ogni classe contiene metodi per configurare core specifici nell'architettura. La classe CDma ha metodi per le configurazioni DMA. La classe CRead ha metodi per gli AGU associati alle memorie di pesi e bias. La classe CComp contiene oggetti di classi che a loro volta controllano le AGU associate alle memorie IFM, ai core di elaborazione personalizzati e alle memorie OFM. Infine, anche i core personalizzati sono configurati con i metodi di questa classe.

3.2.2 *Strategie implementative*

Al fine di ridurre il numero di cicli necessari per accedere alla memoria e prelevare i dati, questi ultimi vengono archiviati in un formato speciale: ZXY anziché nel formato originale XYZ. Il formato originale, infatti, memorizza i valori per colonna e riga di ciascuna mappa delle caratteristiche, rendendo più irregolare la lettura dei dati. I valori di input vengono dispersi in 9 gruppi di tre valori contigui (Figura 3.11) richiedendo quindi tre cicli per scorrere tutti i valori (un ciclo per scorrere i valori in ciascuna riga FM, un ciclo per iterare attraverso le righe di un FM e uno per scorrere differenti FM). Nel formato ZXY invece, i dati vengono memorizzati prima per canale. Dalla Figura 3.11 è possibile osservare che i valori richiesti per

il calcolo del primo pixel sono memorizzati in tre gruppi di 9 valori contigui. Questo formato quindi, a differenza del primo, richiede due cicli per accedere a tutti i valori: un ciclo per eseguire l'iterazione su tutte le righe e un ciclo per eseguire l'iterazione, in ogni riga, su tutti i valori e canali.

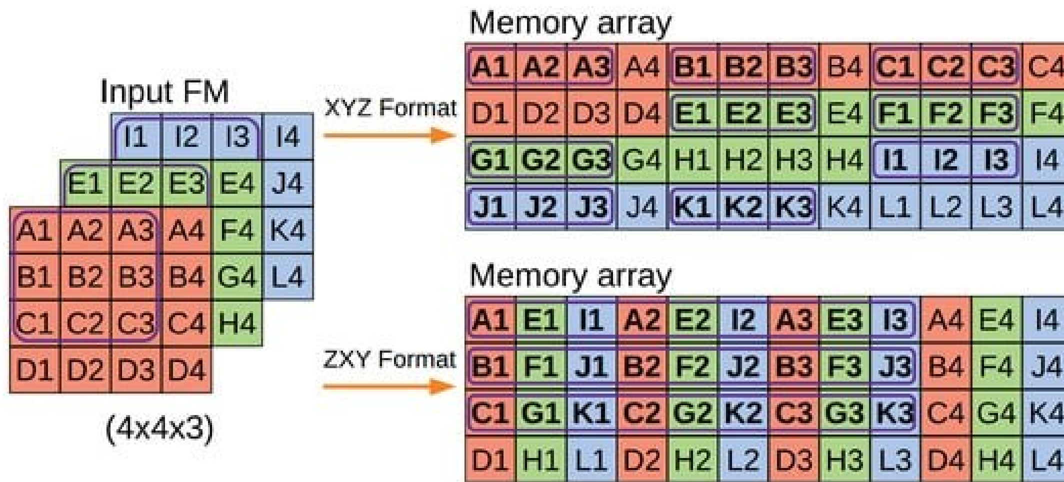


Figura 3.11 Formato di archiviazione in memoria dei dati

I primi 8 livelli della rete possono essere eseguiti in coppie di livelli convoluzionali + maxpool. Per ridurre al minimo i trasferimenti dei dati, ogni riquadro IFM viene letto dalla memoria principale una volta e il corrispondente riquadro di uscita viene calcolato per tutti i kernel. Con questa strategia, tutti i kernels vengono caricati dalla memoria per lo stesso set di riquadri di ingresso IFM.

Per ovviare al problema delle limitazioni dei buffer di memoria su chip, viene utilizzata la strategia della piastrellatura. L'idea principale della piastrellatura è dividere i dati in sottoinsiemi di iterazioni ben strutturati per riorganizzare l'ordine di esecuzione dei nidi di loop, al fine di aumentare la quantità di riutilizzo dei dati, nonché la loro località e migliorare la pianificazione dei calcoli paralleli.

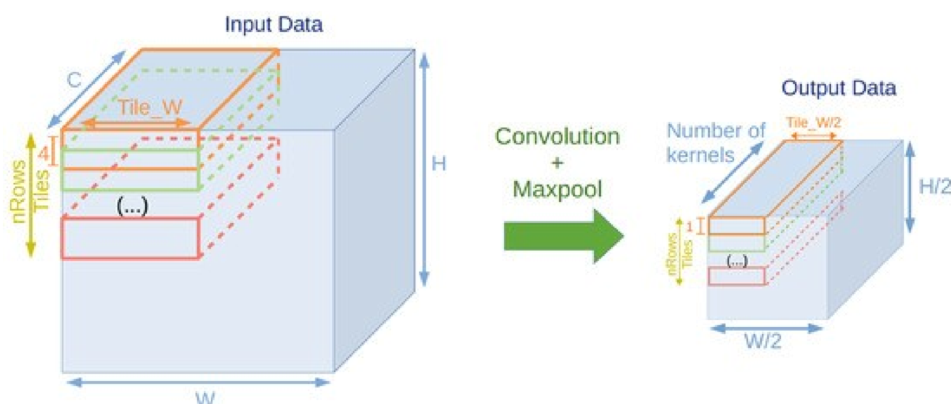


Figura 3.12 Diagramma di piastrellatura dati dei livelli 1-8

Dal nono livello in poi, dal momento che cambia il formato degli IFM, i quali sono entrambi 26x26 e 13x13, cambia anche la tipologia di piastrellatura.

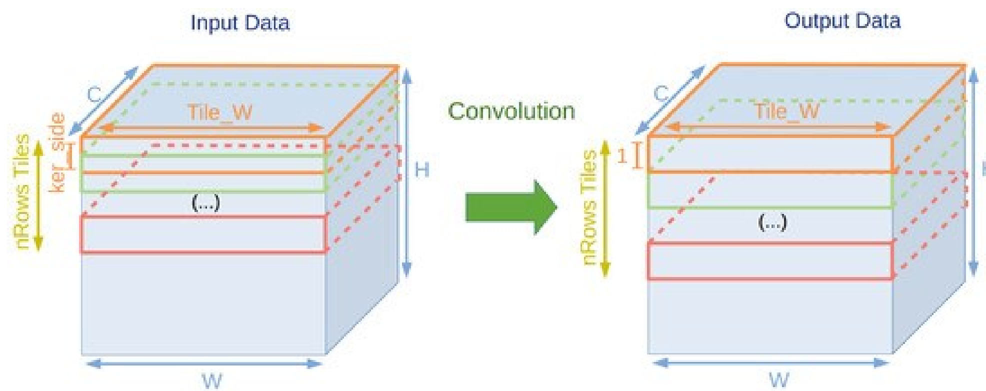


Figura 3.13 Diagramma di piastrellatura dati dei livelli dal 9 in poi

Gli strati convoluzionali 16 e 23 sono accoppiati ed eseguiti con gli strati YOLO 17 e 24. Dopo l'esecuzione della convoluzione dei primi, i secondi equivalgono all'esecuzione dell'attivazione sigmoidea sui canali di ingresso selezionati, pertanto la loro combinazione consiste nell'eseguire uno strato convoluzionale regolare con attivazione sigmoidea.

Lo strato di 19 è accoppiato invece con un layer di sovracampionamento vedi Figura 3.14.

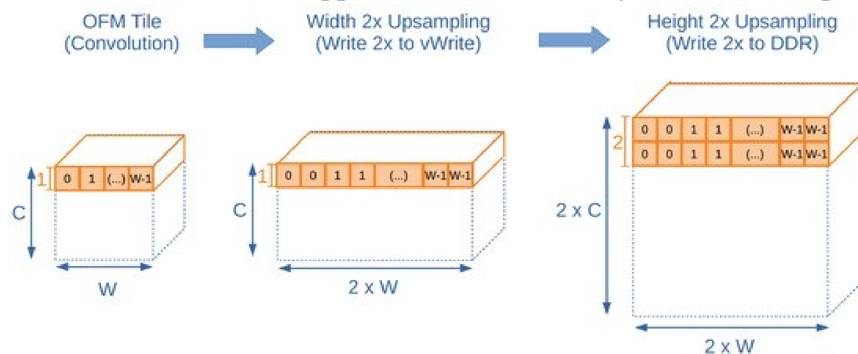


Figura 3.14 Diagramma di sovracampionamento dopo la convoluzione

Infine, i livelli 10 e 12 vengono invece accelerati in isolamento.

Il maxpool si esegue invece secondo la Figura 3.15. Il primo flusso di dati legge l'input del livello completo nelle memorie piastrellate IFM. Ogni memoria IFM ha una piastrella 2 × L × C. Ogni flusso di dati esegue il *maxpooling* su un blocco di ingresso a 2 × 2 × C e uscite a blocco 1 × 1 × C. Ogni blocco di uscita viene scritto nella memoria principale al termine della configurazione del flusso di dati.

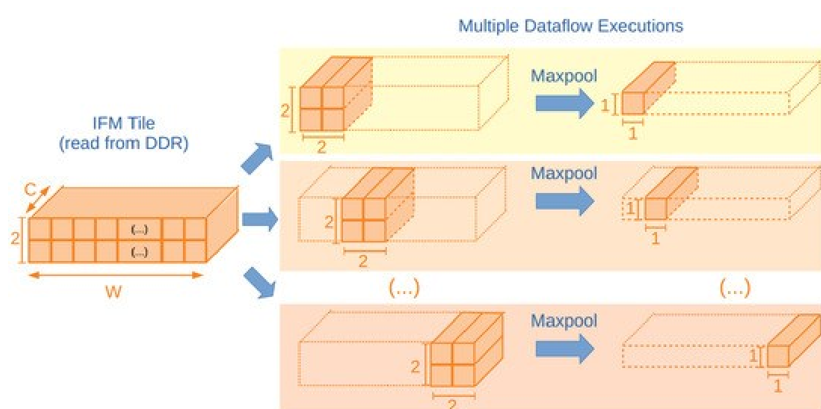


Figura 3.15 Input e Output per i layer di Maxpool

3.2.3 *Risultati e Conclusioni*

Per la valutazione dei risultati sono state implementate due diverse configurazioni dell'architettura le quali differiscono per le dimensioni del percorso dati 8 e 16. L'acceleratore è stato implementato nella logica programmabile FPGA ZYNQ, un dispositivo di costo limitato adibito a dispositivi IoT ad alte prestazioni, e integrato con il sistema di elaborazione contente un processore ARM. Per illustrare la configurabilità dell'acceleratore, sono stati implementati diversi progetti con un diverso numero di core e determinato l'occupazione delle risorse FPGA.

Cores	LUTs	BRAMs	DSPs
13×8	33,346	120	208
13×4	20,356	112	104
13×2	13,656	108	52
8×8	20,746	80	128
8×4	12,698	72	64
4×8	10,588	48	64
4×4	6528	40	32

TABELLA III.V Utilizzo risorse acceleratore a 8 bit, 4 MACs per core e diversi numeri di core

Dalla Tabella III.V si evince che maggiore è il numero di linee di core, più si paga in termini di costo in quanto ogni linea aggiunge un buffer di output. Come si può notare, infatti, 8x4 e 4x8, nonostante abbiano lo stesso numero di core, hanno un costo diverso. In particolare, le prime, che hanno un maggior numero di linee, richiedono più BRAM e LUT. Le memorie necessaria ai pesi e alle IFM sono le stesse per ogni configurazione e vengono supposte rispettivamente a 8 e 32 kByte per ogni buffer. In seguito, vengono confrontate le risorse utilizzate dalle due quantizzazioni a 8 e 16 bit.

Risorsa	ZYNQ7020	
Percorso dati	16	8
LUT	27.454	33.346
BRAM da 36kB	120	120
DSP	208	208

TABELLA III.VI Utilizzo delle risorse in un FPGA ZYNQ7020

Nell'FPGA ZYNQ7020 a basso costo, il design è principalmente limitato dal numero di DSP e BRAM. L'elevata percentuale di utilizzo di questi moduli hardware influenza la frequenza operativa a causa del routing. Poiché un singolo DSP può implementarne due 8×8 moltiplicazioni, la soluzione a 8 bit raddoppia il numero di MAC. È possibile ridurre il numero di BRAM della soluzione a 8 bit, ma un numero maggiore di BRAM aumenta il numero di strati

che possono beneficiare della tecnica del ping-pong delle memorie. Pertanto, entrambe le soluzioni utilizzano lo stesso numero di memorie.

L’architettura Tiny-YOLOv3 è stata eseguita nell’acceleratore con tutte le configurazioni proposte in Tabella III.V e i parametri di configurazione impostati sono contenuti in Tabella III.VII.

<table>
<tr><th>Parametro</th><th colspan="8">Acceleratore</th></tr>
<tr><td>Architettura</td><td>A1</td><td>A2</td><td>A3</td><td>A4</td><td>A5</td><td>A6</td><td>A7</td><td>A8</td></tr>
<tr><td>nCols</td><td>8</td><td>4</td><td>2</td><td>8</td><td>4</td><td>8</td><td>4</td><td>4</td></tr>
<tr><td>nRighe</td><td>13</td><td>13</td><td>13</td><td>8</td><td>8</td><td>4</td><td>4</td><td>13</td></tr>
<tr><td>nMAC</td><td colspan="8">4</td></tr>
<tr><td>DDR_ADDR_W</td><td colspan="8">32</td></tr>
<tr><td>DATAPATH_W</td><td colspan="7">8</td><td>16</td></tr>
<tr><td>MEM_BIAS_ADDR_W</td><td colspan="8">3</td></tr>
<tr><td>MEM_WEIGHT_ADDR_W</td><td colspan="8">14</td></tr>
<tr><td>MEM_TILE_ADDR_W</td><td>15</td><td>15</td><td>15</td><td>15</td><td>15</td><td>16</td><td>16</td><td>15</td></tr>
<tr><td>MEM_TILE_EXT_ADDR_W</td><td colspan="8">15</td></tr>
</table>

TABELLA III.VII Parametri di configurazione per l'acceleratore

Tutte le architetture sono state sintetizzate con una frequenza di clock di 100 MHz e testate con Tiny-YOLOv3 (si vedano i risultati delle prestazioni in Tabella III.VIII e Figura 3.16).

Arq	A1	A2	A3	A4	A5	A6	A7	A8
Esec. (SM)	68	135	268	129	259	246	492	140
FPS	14.7	7.4	3.7	7.8	3.9	4.1	2.0	7.1
FPS/core	0.14	0.14	0.14	0.12	0.12	0.13	0.13	0.14

TABELLA III.VIII Tempi di esecuzione di Tiny-YOLOv3 sull’architettura proposta con diverse configurazioni della matrice core

Le soluzioni più efficienti utilizzano 13 core per colonna, poiché le dimensioni delle mappe delle caratteristiche sono un multiplo di 13. Scegliere le altre configurazioni avrebbe significato una perdita nell’efficienza delle prestazioni poiché, in alcune iterazioni dell'algoritmo, alcuni core non vengono utilizzati. Ad esempio, l'esecuzione di una mappa delle funzionalità di dimensione 26 nell'architettura configurata con otto righe di core richiederebbe quattro iterazioni e nell'ultima iterazione sarebbero in esecuzione solo due righe di core.

Le configurazioni A6 e A5 utilizzano lo stesso numero di core, ma A6 è più veloce poiché il numero inferiore di core per colonna migliora l'efficienza. Entrambe le architetture A8 e A2 hanno lo stesso numero di core, ma l'architettura A8 è per la quantizzazione a 16 bit.

L'architettura a 8 bit è leggermente più veloce e consuma meno risorse al costo di 0,7 pp di precisione.

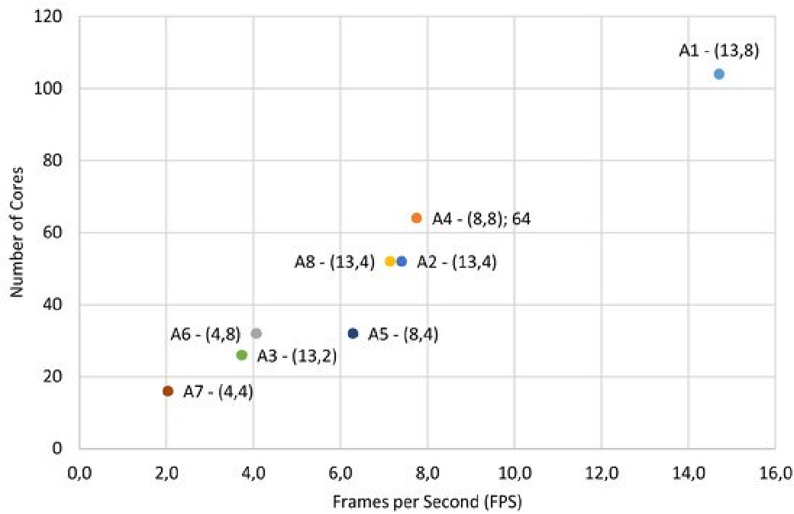


Figura 3.16 Il numero di core rispetto ai frame al secondo di ciascuna configurazione dell'architettura. I grafici indicano la configurazione come numero di righe di nuclei e numero di colonne di nuclei

La Tabella III.IX presenta i tempi di esecuzione della rete Tiny-YOLOv3 su più piattaforme: Intel i7-8700 a 3,2 GHz, GPU RTX 2080ti e GPU integrate Jetson TX2 e Jetson Nano. I risultati di CPU e GPU sono stati ottenuti utilizzando la rete Tiny-YOLOv3 originale con rappresentazione in virgola mobile. Il risultato della CPU corrisponde all'esecuzione di Tiny-YOLOv3 implementato in C. Il risultato della GPU è stato ottenuto dall'esecuzione di Tiny-YOLOv3 in ambiente Pytorch utilizzando le librerie CUDA.

Tiny-YOLOv3 sulle CPU desktop è troppo lento. Il tempo di inferenza su una GPU RTX 2080ti ha mostrato un aumento della velocità di 109 rispetto alla CPU desktop. Utilizzando l'acceleratore proposto, i tempi di inferenza erano 140 e 68 ms, nello ZYNQ7020. L'FPGA a basso costo era 6X (16 bit) e 12X (8 bit) più veloce della CPU, con un piccolo calo di precisione rispettivamente di 1,4 e 2,1 punti. Rispetto alla GPU incorporata, l'architettura proposta era del 15% più lenta ma il vantaggio dell'utilizzo dell'FPGA è nel consumo di energia di molto inferiore. Jetson TX2 ha una potenza prossima ai 15 W, mentre l'acceleratore proposto ha una potenza di circa 0,5 W. La Nvidia Jetson Nano consuma al massimo 10 W ma si aggira intorno 12x più lento dell'architettura proposta.

Versione software	piattaforma	CNN (ms)	FPS
Virgola mobile	CPU (Intel i7-8700 @ 3,2 GHz)	819.2	1.2
Virgola mobile	GPU (RTX 2080ti)	7.5	65.0
Virgola mobile	eGPU (Jetson TX2) [43]	-	17
Virgola mobile	eGPU (Jetson Nano) [43]	-	1.2
Punto fisso-16	**ZYNQ7020**	**140**	**7.1**
Punto fisso-8	**ZYNQ7020**	**68**	**14.7**

TABELLA III.IX Tempi di esecuzione di Tiny-YOLOv3 su più piattaforme

L'implementazione proposta è stata inoltre confrontata con i precedenti acceleratori di TINY-YOLOv3. In Tabella III.X viene riportata la quantizzazione, la frequenza operativa, l'occupazione delle risorse FPGA (DSP, LUT e BRAM), tempo di esecuzione e frame al secondo. Inoltre, per quantificare correttamente l'efficienza delle risorse hardware in soluzioni diverse, le quali fanno uso di un numero diverso di risorse, vengono presi in considerazione tre parametri normalizzati FPS/kLUT, FPS/DSP e FPS/BRAM. In questo modo, si determina il numero di ciascuna risorsa utilizzata per produrre un frame al secondo, più sono alti i valori e maggiore è l'efficienza di utilizzo di queste risorse.

	[38]	[39]	[41]	[40]	**Nostro**	
Dispositivo	ZYNQZU9EG	ZYNQ7020	VirtexVX485T	US XCKU040	ZYNQ7020	
Set di dati	Segnaletica pedonale	set di dati COCO				
quant.	8	16	18	16	16	8
Freq. (MHz)	-	100	200	143	100	100
DSP	-	120	2304	832	208	208
LUT	-	26K	49 K	139 K	27,5 mila	33,4 K
BRAM	-	93	70	384	120	120
Esec. (SM)	9.6	532.0	-	24.4	140	68
FPS	104	1.9	-	32	7.1	14.7
FPS/kLUT	-	0.07	-	0,23	0,26	0,44
FPS/DSP	-	0,016	-	0,038	0,034	0,068
FPS/BRAM	-	0,020	-	0,083	0,058	0,116

TABELLA III.X Confronto delle prestazioni con altre implementazioni FPGA

L'acceleratore implementato ha raggiunto 7 e 14 FPS eseguendo la rete neurale Tiny-YOLOv3 con punteggi di precisione di 31,5 e 30,8 mAP_{50} nel set di dati di test COCO 2017 (vicino a 32,9 del modello originale con virgola mobile), rispettivamente per quantizzazioni a 16 e 8 bit. Data la drastica semplificazione ottenuta con la rappresentazione a virgola fissa, questa perdita di accuratezza è accettabile.

La configurabilità dell'acceleratore consente di personalizzare il design dell'architettura per diverse dimensioni di FPGA e versioni Tiny-YOLO, purché la CNN utilizzi solo i livelli supportati dall'acceleratore. Inoltre, è possibile modificare le dimensioni di quantizzazione utilizzate per rappresentare i dati. Nel progetto testato è stata implementata una rappresentazione in virgola fissa a 16 e 8 bit di pesi, bias e attivazioni.

In conclusione, i risultati ottenuti sono molto promettenti per un uso IoT dato che la soluzione è almeno sei volte più veloce di una CPU all'avanguardia e l'FPGA utilizzato è di basso costo. Al fine di migliorare ulteriormente il throughput del rilevamento di oggetti nei dispositivi IoT, è possibile seguire due direzioni di ricerca, ridurre la dimensione del modello e la quantizzazione (che determina i requisiti computazionali). [17]

3.3 **Manutenzione Predittiva**

La manutenzione predittiva è un'applicazione innovativa molto importante per gli attuali sistemi cyber-fisici (CPS) in quanto, permette di rilevare eventuali segni di guasto imminente nel dispositivo meccanico in cui viene applicata. In questo modo, è possibile eseguire la manutenzione del dispositivo prima del suo eventuale guasto, riducendone significativamente i tempi di fermo e i costi che ne sarebbero derivati. Gli approcci più moderni per effettuare tali previsioni vedono la combinazione tra scienza statistica e apprendimento automatico applicato ai flussi di dati che vengono raccolti tramite sensori (IoT). A seconda delle proprietà meccaniche del dispositivo, tra le tipologie di dati che vengono utilizzate troviamo vibrazioni, accelerazione, temperatura, umidità, suono, corrente elettrica, induttanze varie ecc… ognuna raccolta in posizioni diverse del dispositivo meccanico. Poiché tali rilevamenti non devono essere intrusivi ed intaccare l'accesso all'alimentazione dell'infrastruttura, è essenziale che i nodi funzionino con un'alimentazione a batteria. Ciò limita la quantità di calcolo disponibile su un nodo, nonché la larghezza di banda di comunicazione su connessioni wireless assetate di energia. In questo contesto, entra in campo l'edge computing e l'utilizzo di acceleratori hardware che aiutano a porre rimedio alla grande quantità di calcoli che da essa ne derivano.

In questa applicazione, viene presentato Eciton, un'acceleratore hardware basato su FPGA che usa reti neurali ricorrenti a memoria a lungo termine (LSTM) (sottocapitolo 1.2.4).

3.3.1 *Architettura del nodo di sistema*

Eciton amplia un nodo finale CPS/IoT con un acceleratore LSTM flessibile implementato su un FPGA a basso costo e a bassa potenza. I dati raccolti dal sensore vengono prima valutati dall'acceleratore LSTM e inviati in modalità wireless ad un server centrale solo se il guasto locale viene rilevato. Sul server centrale, un sistema software più complesso può quindi prendere decisioni di livello superiore in base ai dati provenienti da più nodi.

La Figura 3.17 mostra l'architettura complessiva di un nodo. L'FPGA filtra i dati provenienti dal sensore, riducendo il traffico e il carico di lavoro riservato al microcontrollore (MCU) il quale, a causa della complessità della gestione della rete, si occupa dell'interfaccia di rete.

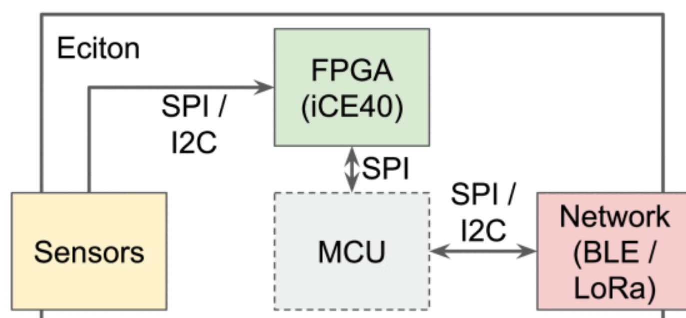


Figura 3.17 Architettura del nodo finale Eciton

L’acceleratore LSTM (Figura 3.18) è costituito da due nuclei separati LSTM e strati densi. Il MCU è responsabile del caricamento dei comandi e dei pesi in entrambi i core, i quali poi sono pronti per eseguire l’inferenza con i dati provenienti dai sensori.

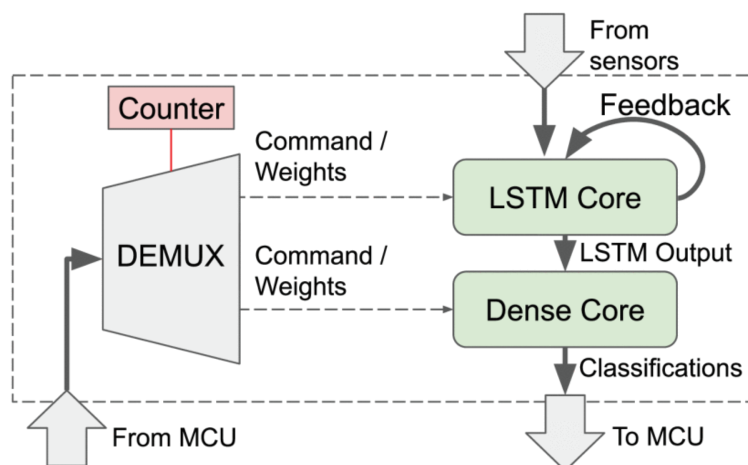


Figura 3.18 L'MCU fornisce pesi ai core LSTM e Dense, dopodiché l'acceleratore può elaborare l'input dai sensori.

3.3.2 *Implementazione logica programmabile*

La piattaforma FPGA utilizzata è iCE40 UP5K, un dispositivo a basso costo e bassa potenza con rigorose restrizioni sulle risorse. Oltre alla normale RAM, questo dispositivo è dotato di una memoria SRAM su chip di 1024kbit suddivisa in quattro blocchi single-port RAM (SPRAM). Inoltre, è dotato di 8 blocchi DSP a 16 bit per aritmetica veloce. Al fine di rendere la rete neurale eseguibile all’interno del dispositivo e di ottimizzarne le prestazioni, vengono applicati metodi di compressione della rete e di quantizzazione dei dati. Eciton a seguito del training della rete esegue la quantizzazione dei pesi ottenuti da 4 byte in virgola mobile a 8 bit in virgola fissa, riducendo in questo modo l’ingombro della memoria di ¼. Al fine di ridurre le risorse richieste al chip, l'acceleratore utilizza il sigmoide rigido lineare sia per il kernel che per le funzioni di attivazione ricorrente di LSTM (al posto di quelle non lineari come sigmoide e tanh). Questo perché il sigmoide rigido può essere implementato in modo efficiente fornendo un'approssimazione più accurata delle funzioni non lineari, rispetto a funzioni più semplici come l'unità lineare rettificata (ReLU).

L’architettura di base dell’LSTM è mostrata in Figura 3.19. Sia l’ingresso che l’uscita vengono eseguiti in unità di parole a 8 bit. Nei quattro blocchi SPRAM su chip vengono archiviati tutti i valori dei pesi combinati in un unico spazio di indirizzi, esponendo un’unica interfaccia a 16 bit. La pipeline di calcolo è costituita da due sotto elementi: la pipeline MAC e la pipeline degli stati per calcolare lo stato della cella (o carry) e gli stati nascosti. Entrambe condividono un unico modulo ALU quantizzato.

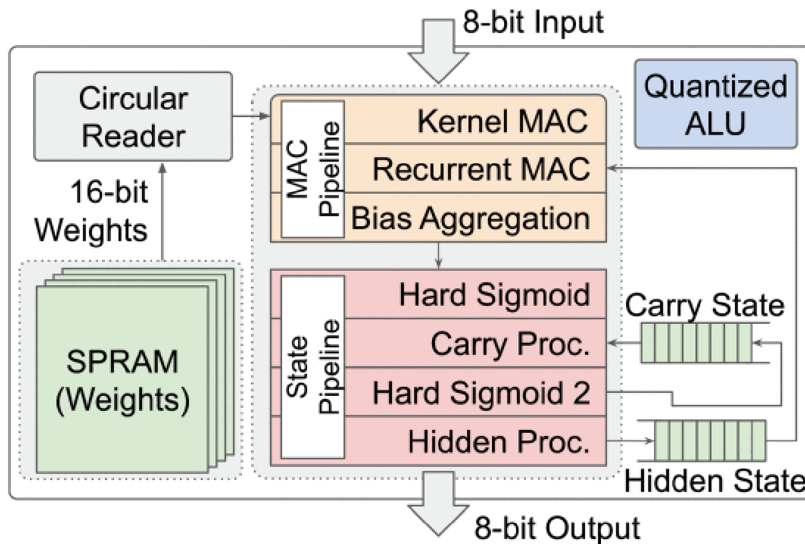


Figura 3.19 Microarchitettura del Core LSTM

I pesi sono memorizzati in modo tale che per ogni tupla di input, l'intera memoria dei pesi venga scansionata linearmente una volta dal modulo lettore circolare. Il layout di memoria, infatti, per ciascun livello LSTM è costituito nell'ordine prima dai pesi del kernel, poi da quelli ricorrenti e infine dai pesi di bias. Ogni peso è composto da 1 byte, pertanto, i pesi di una stessa cella dell'acceleratore sono raggruppati a blocchi di 4 byte. Il calcolo viene quindi ordinato dalla microarchitettura nello stesso ordine, in modo che la scansione della memoria avvenga in modo lineare.

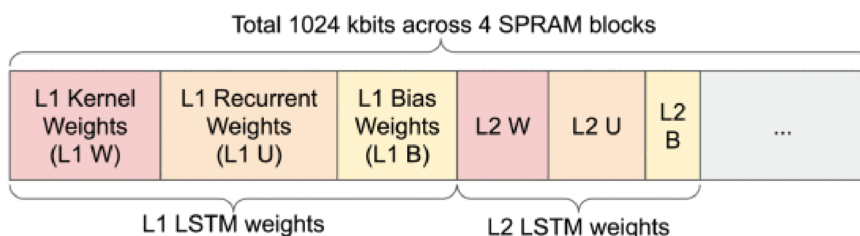


Figura 3.20 ordine di memorizzazione pesi dei vari layers

3.3.3 *Unità di calcolo*

Nella pipeline MAC viene eseguita la maggior parte dell'aritmetica per LSTM. Ogni passaggio viene elaborato in una sequenza non sovrapposta in modo tale da poter riutilizzare efficientemente un piccolo numero di unità aritmetiche quantizzate. Il calcolo, come detto sopra, è organizzato in base all'ordine di memorizzazione del peso (kernel, riccorrenti, bias). Per ogni input di 8 bit, le MAC con i 4 pesi kernel possono essere eseguite in modo indipendente per ogni cella dello stesso layer dell'LSTM. Ne risulta un numero Unit di tuple a 4-byte che vengono inseriti in una BRAM FIFO, dove Unit è il numero di celle che compongono il layer. Durante l'esecuzione delle MACs, i pesi ricorrenti vengono caricati in coda di lettura in modo tale da essere pronti, una volta terminata l'esecuzione, per eseguire la stessa operazione nello

stesso hardware con i dati di hidden state (se presenti). Una volta terminate entrambe le fasi i risultati di ciascuna vengono inoltrati alla fase di bias, quindi ai due flussi vengono aggiunti i pesi dei bias già letti nella pipeline di lettura. Al termine di questa fase ci saranno un numero Unit di 4 tuple di valori memorizzate in una BRAM FIFO.

Una volta eseguite le operazioni MAC e generato un numero Unit di tuple da 4 byte, nell'altra pipeline vengono calcolati lo stato della cella e gli stati nascosti, per il passaggio temporale successivo in base all'algoritmo LSTM. Questo processo riutilizza lo stesso modulo ALU quantizzato per eseguire la moltiplicazione, l'addizione quantizzata e il sigmoide rigido. Anziché utilizzare la memoria complessa, i risultati vengono memorizzati in una singola FIFO implementata nella BRAM dove verranno mantenuti per i layer successivi. Una volta terminata l'esecuzione di tutti i layer quindi il calcolo torna al primo layer per il passaggio temporale successivo, lo stato della cella per tutti i layers attenderà in ordine nella FIFO.

L'unità ALU quantizzata fornisce tre funzioni: addizione quantizzata, moltiplicazione e sigmoide rigido. Mentre l'addizione tra due valori quantizzati è semplice, la moltiplicazione è leggermente più complicata in quanto anche il risultato deve essere riportato al fattore di scala originale tramite divisione. Sebbene la divisione hardware richieda molte risorse, fortunatamente il fattore di scala è un valore statico determinato durante la quantizzazione. Precalcoliamo una divisione normalizzata (1≪16)/scala durante la quantizzazione. Questo valore può essere semplicemente moltiplicato per i risultati della moltiplicazione utilizzando il blocco DSP UP5K, che supporta la moltiplicazione a 16 bit in un valore a 32 bit. Il valore risultante può essere spostato verso il basso di 16 bit per ottenere i risultati di divisione corretti. Poiché i risultati della moltiplicazione originale sono larghi 16 bit e i pesi e l'input sono tutti a 8 bit, non viene persa alcuna correttezza. Di conseguenza, la moltiplicazione quantizzata richiede due blocchi DSP, uno per la moltiplicazione e uno per la divisione per il fattore di scala.

Anche il modulo sigmoide rigido richiede un moltiplicatore, ma dal momento che non vengono mai richieste contemporaneamente la moltiplicazione e il calcolo sigmoideo, l'ALU è progettata per condividere il moltiplicatore interno.

Poiché l'interfaccia di memoria SPRAM è larga 16 bit, vengono forniti due pesi per ciclo. Dal momento che i quattro pesi in una cella vengono elaborati indipendentemente l'uno dall'altro, la nostra ALU quantizzata implementa internamente due moduli separati per operare in parallelo, utilizzando 4 degli 8 blocchi DSP disponibili.

Il nucleo denso è un'entità separata dal nucleo LSTM. La sua architettura è simile all'architettura della cella LSTM, ma molto più semplice. Il nucleo denso ha solo tre fasi di calcolo: kernel MAC, aggregazione di bias e un sigmoide rigido finale. Poiché la quantità di

calcolo dello strato denso è quasi trascurabile rispetto allo strato LSTM, l'ALU quantizzata dell'ALU del blocco denso è progettata per utilizzare solo un'unità MAC, utilizzando solo due blocchi DSP e meno logica di supporto.

3.3.4 *Risultati e Conclusione*

Eciton è stato implementato utilizzando *toolchain* opensource quali Bluespec e IceStorm. Il primo è un set di strumenti EDL per la progettazione di ASIC e FPGA basato sulla semantica hardware tradizionale SystemVerilog il quale, utilizza il modello standard della struttura hardware Verilog (e VHDL).[29] Il secondo invece fornisce gli strumenti per l'analisi e la creazione dei file bitstream (flusso Verilog-to-Bitstream) per FPGA iCE40.[30] Come MCU è stato utilizzato Arduino Nano.

Per la valutazione delle prestazioni dell'acceleratore, quest'ultimo viene dimostrato su due scenari di manutenzione predittiva:

1. Manutenzione del motore turbofan

   Viene utilizzato un set di dati che simula il degrado e quindi guasto del motore utilizzando dati provenienti da 25 diversi sensori. Il modello utilizza due strati LSTM con 100 e 50 nodi nonché uno strato denso a cellula singola. Il numero totale di pesi ammonta a 80.651.

2. Manutenzione del motore elettrico

   Il modello è addestrato su quattro flussi di dati input, tre assi dell'accelerometro e un sensore di umidità raccolti da un motore elettrico in fase di spegnimento. Gli accelerometri sono campionati a 2 KHz, mentre il sensore di umidità è campionato a 50 Hz. Il sistema è alimentato da un Power harvester che genera energia dal campo magnetico mentre il motore gira. Poiché la raccolta di energia diventa non disponibile dopo lo spegnimento del motore, i dati raccolti devono essere elaborati o trasmessi dopo il rilevamento dell'arresto; entro un budget di 0,6 joule di energia raccolta. Il modello utilizza tre strati ciascuno con 128, 32 e 16 nodi e uno strato di denso a cella singola per un totale di 91.873 valori di peso.

In Figura 3.21 sono stati riportati i risultati dell'accuratezza del modello utilizzando 3 diverse quantizzazioni. Nonostante il modello quantizzato a 16 bit del motore turbofan fosse sufficientemente piccolo per adattarsi alla SPRAM su chip, per entrambi i motori è stata scelta

la quantizzazione a 8 bit. In questo modo è possibile utilizzare meglio i blocchi DSP limitati, eseguendo più operazioni a 8 bit per ciclo.

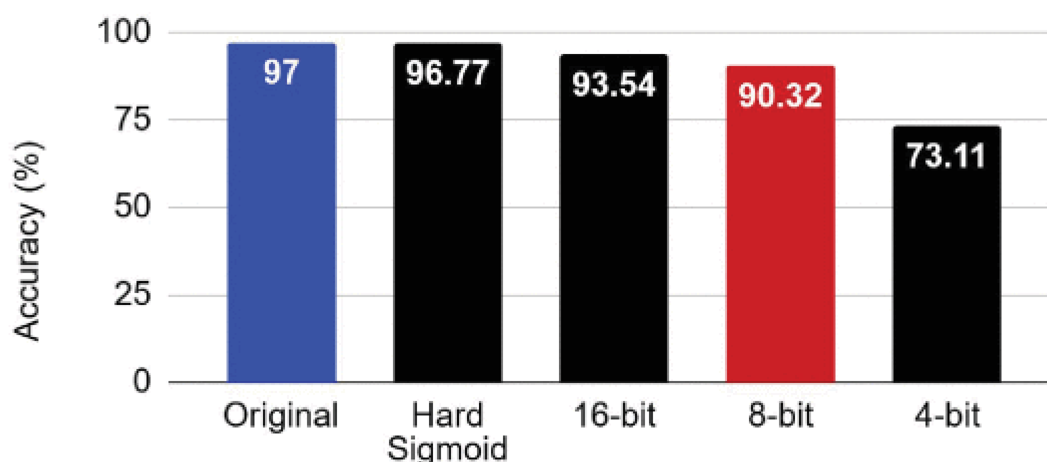


Figura 3.21 Accuratezza del modello turbofan con diverse quantizzazioni

Le risorse dell'FPGA utilizzate sono riportate in Tabella III.XI. Come si evince dalla Tabella, vengono utilizzati tutti e 4 i blocchi SPRAM disponibili, nonché 6 degli 8 blocchi DSP. L'acceleratore utilizza il 94,5% delle celle logiche della scheda, raggiungendo una frequenza di clock di 17MHz. La memoria richiesta dai modelli turbofan e del motore elettrico è rispettivamente 80.651 e 91.873 byte; pertanto, sono perfettamente contenuti all'interno del dispositivo che ha una capacità totale su chip dei quattro blocchi SPRAM di 256 KB.

Resource	Resource Utilization	Percent Utilization (%)
LC	4987 / 5280	94.5
SPRAM	4 / 4	100
BRAM	22 / 30	73.3
DSP	6 / 8	75

TABELLA III.XI Utilizzo delle risorse su chip Eciton

Valutando le prestazioni ed efficienza energetica di Eciton rispetto a più configurazioni di sistema, per entrambe le applicazioni, ne risulta che l'implementazione su FPGA è l'unica a mantenere un'elevata performance rispettando i requisiti sul consumo di energia all'edge (Figura 3.22 e 3.23). L'acceleratore FPGA di Eciton, infatti, ha misurato un consumo energetico di circa 17 mW a pieno carico ed efficienza un ordine di grandezza migliore rispetto ad altre opzioni ad alte prestazioni e bassa potenza.

Eciton è anche in grado di soddisfare i requisiti di limite di energia del motore elettrico impiegando 31,3 secondi per elaborare completamente i 25 secondi di 1/4 di flusso di dati campionati secondo i parametri del modello, con un consumo energetico costante di 17 mW da parte dell'FPGA. Supponendo che il resto del sistema possa essere messo in stop durante questo

periodo, l'acceleratore Eciton su FPGA termina il lavoro entro il limite di energia di 0,6 Joule imposto dal power harvester. Nessun'altra configurazione di sistema è stata in grado di raggiungere questo traguardo.

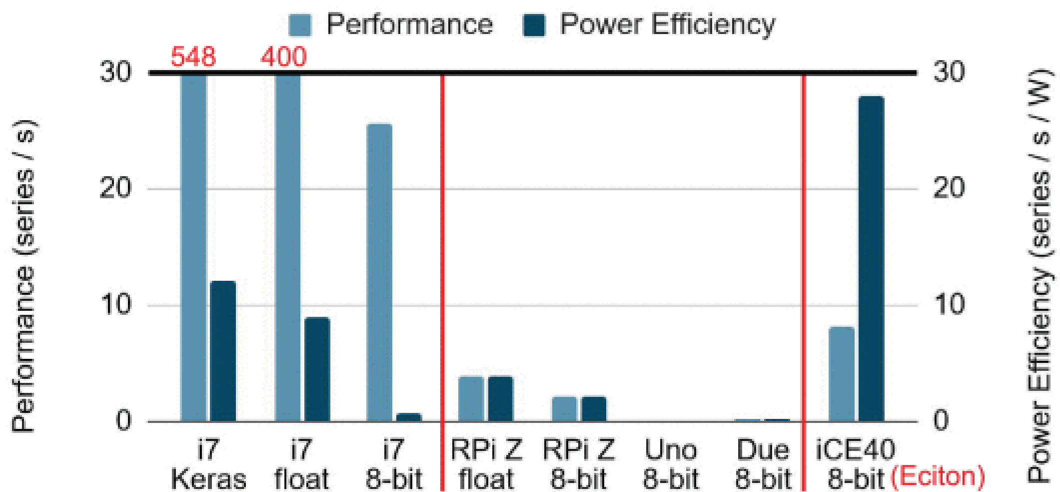


Figura 3.22 Prestazioni ed efficienza energetica per turbofan

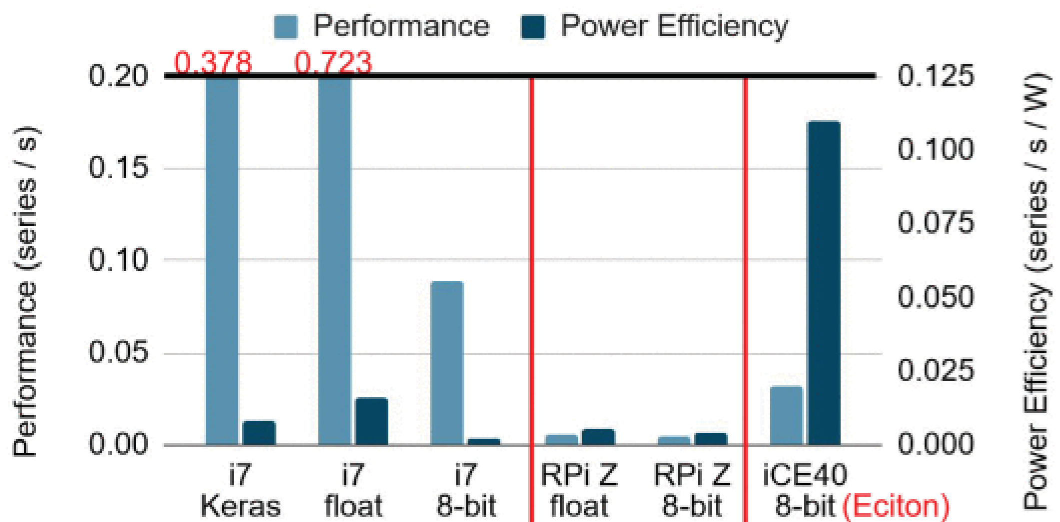


Figura 3.23 Prestazioni ed efficienza energetica per il motore elettrico

La Tabella III.XII mostra il confronto di prestazione ed efficienza energetica con implementazioni FPGA all'avanguardia di LSTM su diverse piattaforme hardware con scala e capacità diverse. Nonostante i vincoli della piattaforma utilizzata Eciton raggiunge prestazioni ed efficienza energetica competenti rispetto ad altre implementazioni a basso consumo. Inoltre, sebbene raggiunga prestazioni ed efficienza inferiori rispetto agli FPGA molto più grandi, esso occupa comunque un ruolo importante nel catalogo degli acceleratori FPGA. I chip più grandi, infatti, possono sicuramente offrire prestazioni migliori per watt, ma le implementazioni edge hanno risorse limitate che potrebbero non essere in grado di utilizzare affatto a causa del budget energetico. D'altro canto, nel caso di Eciton, il consumo di energia è sufficientemente basso da

poter essere alimentato in modo sostenibile da molte unità di raccolta dell'energia non intrusive proposte, il che significa che può funzionare continuamente indefinitamente raccogliendo energia dall'ambiente circostante.

	Low-power		High-power	
	Eciton	[26]	[6]	[8]
Platform	**iCE40**	Artix-7	Zynq-7000	Arria 10
mW	**17**	109	280	19,100
GOP/s	**0.067**	0.055	7.51	304.1
GOP/s/W	**3.9**	0.5	26.84	15.9

TABELLA III.XII Confronti con lo stato dell'arte

È stata presentata la progettazione e la valutazione di Eciton, un acceleratore di rete neurale per la manutenzione predittiva in tempo reale all'edge. I requisiti di alimentazione e rete ridotti consentiti da Eciton consentiranno implementazioni CPS/IoT più ampie abbinate a reti WAN a bassa potenza e tecnologie di raccolta dell'energia. [28]

3.4 **Classificazione del Testo**

La classificazione del testo è il processo di classificazione di una parte di testo in classi predefinite e la sua esecuzione in dispositivi edge IoT è un'applicazione che suscita molto interesse. Per la *Natural Language Processing* (NLP) vengono più comunemente utilizzate due tipologie di reti neurali: CNN e RNN. La recente esplorazione dei metodi basati sulla CNN ha dimostrato che queste sono molto veloci ed efficaci, pertanto, in questo esempio, viene discussa la classificazione del testo multiclasse utilizzando le reti neurali convoluzionali temporali TCNN 1D (sottocapitolo 1.2.5) implementate sulla scheda FPGA Xilinx PYNQ-ZI (Python Productivity for ZYNQ). Nella maggior parte dei casi i modelli finora utilizzati si basavano sui due metodi più comuni a livello di parola e a livello di frase. Questi ultimi però si sono dimostrati non sufficientemente robusti per gestire parole sbagliate, nuove o errate, per questo motivo sono stati introdotti dei metodi più efficienti ovvero, quelli a livello di carattere. La classificazione del testo a livello di carattere, infatti, si rivela perfetta per le applicazioni edge grazie all'adattabilità a parole errate, infrequenti o testo informale e per i suoi modelli leggeri e a bassa latenza.

3.4.1 *Architettura del Sistema*

Il sistema utilizzato è un SoC (System on Chip) della Xilinx che sfrutta la cooperazione tra un processore ARM Cortex-A9 e una logica programmabile FPGA per implementare il sistema completo di inferenza. Nello specifico, la scheda è costituita da: 630 KB BRAM, 512 MB di memoria DDR3, 220 sezioni DSP, 13.300 sezioni logiche (ciascuna con 6 ingressi LUT e 8 flip-flop) e 8 canali DMA con 5 porte slave AXI3 ad alte prestazioni.

Il testo a livello di carattere viene trattato come un segnale grezzo, codificato in una matrice e classificato utilizzando un TCNN 1D. A causa della disponibilità limitata di risorse hardware, solo le attività ad alta intensità di calcolo vengono modellate su FPGA. Questi moduli accelerati funzionano quindi in coerenza con l'architettura complessiva distribuita per classificare il testo.

L'uso più importante di FPGA è per il parallelismo che consente l'accelerazione delle applicazioni hardware. L'utilizzo di blocchi DSP, infatti, fornisce tempi di sviluppo ridotti, migliori prestazioni in virgola mobile e un'elevata efficienza delle risorse.

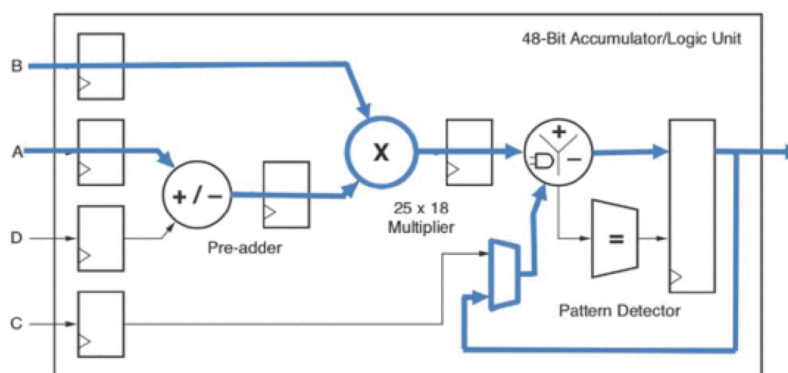


Figura 3.24 Struttura DSP come MAC

Questi includono un moltiplicatore a complemento a due 25*18, un accumulatore a 48 bit, un pre-adder per il risparmio energetico e un'unità aritmetica SIMD che consente operazioni aritmetiche doppie a 24 bit o quadruple a 12 bit (Figura 3.24). Supportano il *pipelining* e hanno bus dedicati per la cascata. Dal momento che lavorare con numeri in virgola mobile sull'hardware è piuttosto sofisticato e dispendioso, viene utilizzata la quantizzazione dei numeri in virgola fissa a 8 bit (per la parte decimale e frazionaria viene assegnato un certo numero di bit in modo tale da ridurre al minimo la perdita di precisione). In questo modo, l'FPGA consente operazioni aritmetiche più semplici sull'hardware e riduce l'utilizzo di memoria. A seguito della quantizzazione quindi, l'unità DSP viene utilizzata come MAC per moltiplicare due numeri a 8 bit e accumulare con il risultato precedente, riducendo della metà il numero totale di fette presenti nel blocco DSP e migliorando di quattro volte l'utilizzo della memoria su FPGA. Attraverso la quantizzazione anche la lunghezza della logica circostante viene ridotta, utilizzando così una lunghezza del bus inferiore e un numero inferiore di LUT. Ciò, consente di eseguire la convoluzione sul blocco logico stesso che si traduce in un'accelerazione dell'intera CNN.

Per ridurre la ridondanza delle reti neurali, che generalmente sono sovraparametrate, è stato usato il metodo della potatura, il quale elimina i neuroni che contribuiscono meno al risultato complessivo, mantenendo comunque un'accuratezza ottimale.

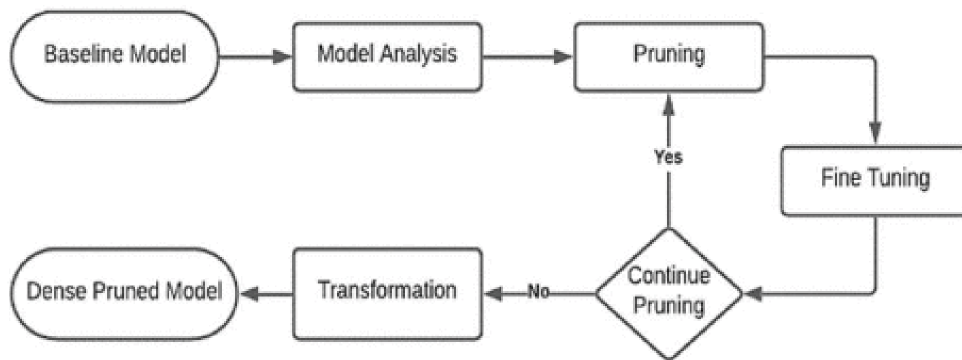


Figura 3.25 Flusso iterativo per eseguire la potatura

Il processo di potatura è di tipo iterativo e viene ripetuto fino ad ottenere una riduzione del peso del 60%, successivamente viene generato un modello denso dai pesi ridotti (processo di trasformazione).

3.4.2 *Implementazione Logica Programmabile*

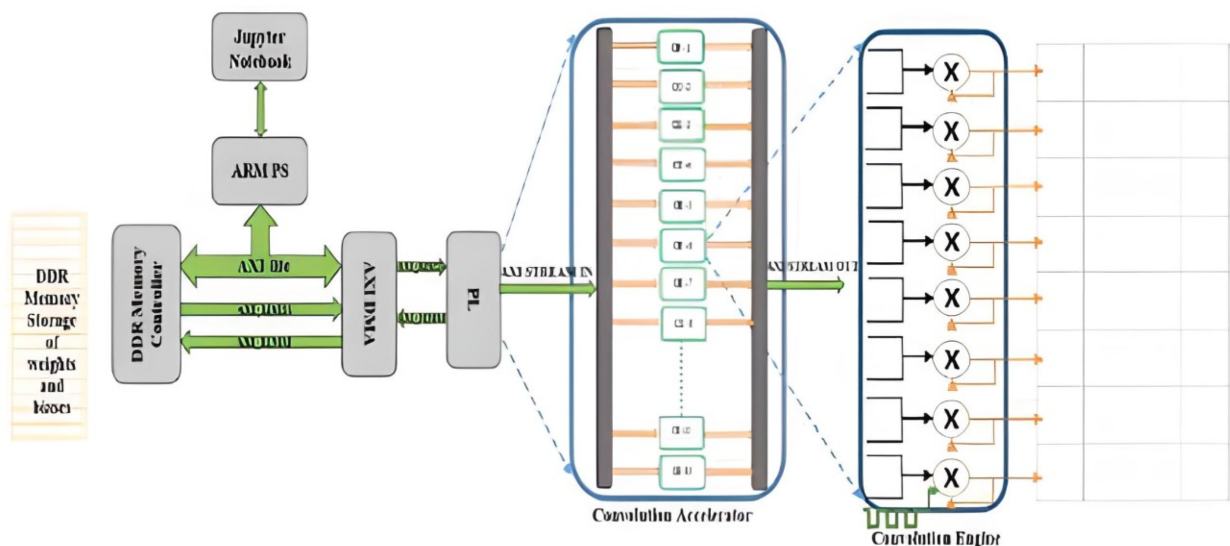


Figura 3.26 Architettura di inferenza completa

L'acceleratore di convoluzione implementato in logica programmabile è costituito da 16 motori di convoluzione ciascuno dei quali è costituito da 8 blocchi DSP che aiutano a parallelizzare l'operazione di convoluzione completa. I dati in ingresso vengono inviati dal PS tramite un'interfaccia di flusso AXI che a sua volta necessita di un DMA AXI per mapparli sulla memoria esterna DDR della FIFO. In sostanza, i dati vengono inviati da PS all'acceleratore di convoluzione tramite l'interfaccia AXI e dopo che il calcolo è stato eseguito dall'acceleratore, i risultati vengono inviati nuovamente al PS per un ulteriore flusso di inferenza attraverso la stessa interfaccia AXI. Ogni riga nell'operazione di convoluzione è assegnata a un particolare blocco DSP che funziona come unità di moltiplicazione e accumulatore e calcola il risultato della convoluzione con i pesi del kernel e le mappa delle caratteristiche di input ricevute dalla

memoria DDR. A seconda della dimensione del kernel, che è 1x7 o 1x3, viene deciso il numero di cicli necessari per produrre la funzione di output. L'acceleratore di convoluzione completo calcolerà 128 righe in parallelo e altre 128 righe in successione riducendo la latenza drasticamente.

3.4.3 *Risultati e Conclusioni*

Il progetto è stato scritto in Python e attraverso il framework Pynq-Torch eseguito direttamente nel SoC della Xilinx. L'acceleratore descritto in 3.4.2 ha sfruttato invece la caratteristica della scheda di supportare gli overlay, ovvero, librerie hardware utilizzate per accelerare parte di inferenza e caricate in fase di esecuzione nella logica programmabile tramite le funzioni Python. In particolare, esso è stato progettato in Verilog attraverso il tool Vivado e importato come qualsiasi altra funzione Python (deve essere esportato in file.tcl e .bit utilizzando Vivado Design Suite identificabile direttamente dalla piattaforma PYNQ).

A seguito della potatura e quantizzazione del modello, come si evince dai risultati in Tabella III.XIII, l'accuratezza ottenuta si mantiene accettabile.

	Quantized and Pruned		Original	
	Train	*Test*	*Train*	*Test*
Accuracy	0.9047	0.8824	0.9366	0.9156
F1 Score	0.9398	0.9214	0.9650	0.9450

TABELLA III.XIII Effetto di potatura e quantizzazione nel modello

Per effettuare il confronto con altri metodi sono state utilizzate tutte le parole nel dizionario Oxford 2019 le quali sono state classificate come cattive o buone generando un numero compreso tra 0 (peggiore) e 1 (migliore). È stata calcolata una media di tutti i metodi per ottenere un punteggio finale per ogni parola che è stato poi confrontato con il metodo proposto. Di una selezione di parole sono poi state inserite manualmente delle varianti informali e confrontati i risultati.

Original Word	Average Score	Modified word	Proposed Method
Great	0.9511	Grt	0.8542
Amazing	**0.9763**	**Amazin**	**0.9052**
Legendary	0.9624	Legendry	0.8733
Coffee	0.4683	-	-
Okay	0.6043	-	-
Very	0.4981	-	-
Bad	0.0742	Bd	0.25
Terrible	**0.0173**	**Terribl**	**0.1254**
Pathetic	0.0565	Pathetic	0.1035

TABELLA III.XIV Confronto risultati medi ottenuti da più modelli e quello proposto

Model	Accuracy	
	Original Dataset	*Modified Dataset*
Bag of words[25]	92.40	83.28
nGrams[26]	95.26	87.92
Bag of Means[25]	87.34	81.43
LSTM[27]	94.86	80.34
word2vec CNN[29]	94.44	78.56
Lookup CNN [6]	92.28	87.32
Proposed	93.47	91.82

TABELLA III.XV Confronto accuratezza modelli e quello proposto con dataset originali e modificati

Come si evince dalla Tabella III.XIV le parole buone hanno un punteggio vicino a 1 e le parole cattive più vicino a 0. Per le parole errate il metodo proposto vede un leggero calo nel punteggio (per le parole buone e viceversa), ma quest'ultimo è ancora efficace rispetto ad altri metodi che al contrario salteranno direttamente quelle parole.

Al fine di valutare le prestazioni dell'FPGA, il modello è stato eseguito sia su CPU che su GPU e per ogni livello di convoluzione è stato confrontato il tempo di esecuzione come risultato di una media di 100 reviews in ingresso (Tabella III.XVI e III.XVII).

Layer	*CPU Time (ms)*	*GPU Time (ms)*	*PYNQ (ms)*
Conv1	135.213	10.276	4.791
Conv2	165.736	11.335	5.272
Conv3	88.914	3.897	1.885
Conv4	74.196	3.587	1.652
Conv5	51.023	3.441	1.728
Conv6	81.482	3.652	1.536

TABELLA III.XVI Tempi di esecuzione medi su un campione di 100 prove per ogni livello della rete su CPU, GPU e FPGA

Device	*CPU*	*GPU*	*PYNQ*
Total Time (ms)	604.210	38.458	18.59
Frequency (GHz)	2.8	1.097	0.1
Power (W)	10.51	22.83	1.48

TABELLA III.XVII Confronto tempo di esecuzione, frequenza e energia consumata su piattaforma CPU, GPU e FPGA

I modelli scelti per CPU e GPU sono rispettivamente un Intel i7 7700HQ, quad-core con frequenza di clock media di 2,8 GHz e Nvidia GTX 960m funzionante a 1,097 GHz. Il sistema

operativo utilizzato era Ubuntu 18.04. Il tempo di esecuzione per CPU e GPU è stato calcolato utilizzando la libreria TorchProf e per FPGA è stata utilizzata la libreria Python Time. L'acceleratore di convoluzione progettato aveva un clock di 100 MHz. Come si vede nella Tabella III.XVII, il tempo di esecuzione della GPU è circa 10-25 volte più veloce rispetto alla CPU, mentre l'FPGA offre un miglioramento di 2 volte della latenza rispetto alla pura implementazione del software su GPU. Inoltre, un importante miglioramento si può osservare anche in termini di potenza in quanto la GPU consuma circa 15 volte più energia rispetto all'FPGA. Per un dispositivo embedded, l'alimentazione e la durata della batteria giocano un ruolo più importante della precisione e il consumo di energia è molto inferiore su FPGA rispetto ai modelli CPU e GPU.

Il documento ha implementato un modello CNN di classificazione del testo a livello di carattere per applicazioni embedded con l'aiuto di un co-design hardware-software su un SoC PYNQ. In generale, è stato utilizzato Python per l'implementazione completa dell'inferenza con la libreria PyTorch per un processo di sviluppo più rapido. La scelta della classificazione del testo a livello di carattere si adatta perfettamente all'applicazione edge grazie a un numero ridotto di parametri rispetto ad altri modelli di classificazione del testo e alla sua natura di adattamento a nuove parole, testi informali ed errori di ortografia. In conclusione, l'FPGA si dimostra ottimo come acceleratore hardware per ridurre il consumo energetico e la latenza per le attività più complesse. [21]

CONCLUSIONI

Attraverso l'analisi di una serie di esempi implementativi, in questa tesi si è voluto tracciare lo stato dell'arte sull'uso dell'FPGA come acceleratore hardware per reti neurali artificiali in applicazioni edge IoT.

Dai risultati ottenuti dapprima nel tracciamento facciale, poi nel riconoscimento di oggetti e testo e infine nella manutenzione predittiva, si evince quanto l'FPGA rappresenti effettivamente una soluzione molto promettente per tali scopi.

Per uno scenario edge, infatti, in cui le risorse sono limitate, l'FPGA, oltre ad essere facilmente implementabile grazie alle sue piccole dimensioni, ha dimostrato un elevato throughput, bassa latenza, basso consumo energetico e prestazioni molto elevate. Il raggiungimento di tali risultati si deve alle caratteristiche del dispositivo e ai vantaggi che da queste ne sono stati tratti. In seguito, verrà effettuata una rassegna sintetica dei principali vantaggi utilizzati. In primo luogo, l'elevato parallelismo architetturale sfruttato dalle altrettante parallelizzabili reti neurali. L'elevata flessibilità in termini di design di hardware specifico, con la possibilità di essere riconfigurato innumerevoli volte. Ancora, la personalizzazione del flusso di dati che ne favorisce il riutilizzo, in aggiunta alla presenza della memoria on-chip su cui archiviarli, portando ad una riduzione degli accessi alla memoria esterna e quindi dell'energia consumata. Infine, sfruttando tutti i diversi gradi di libertà offerti dagli FPGA, è possibile implementare un'ampia gamma di ottimizzazioni sia a livello di miglioramenti della progettazione architetturale, sia di algoritmi di ottimizzazione *hardware-oriented* per il calcolo approssimativo. Tra i più comuni utilizzati anche negli esempi troviamo la quantizzazione, che, come visto, riduce la precisione delle computazioni e la compressione del modello (es. tecnica della potatura) che diminuisce il numero di operazioni MAC necessarie e di conseguenza la dimensioni del modello.

Attualmente, l'FPGA viene utilizzato nei dispositivi perimetrali con il solo scopo di accelerare il processo di inferenza online, riservando al cloud il processo di allenamento in esecuzione offline, a causa del suo elevato costo computazionale e di memoria. Uno degli scenari futuri in cui si sta concentrando la ricerca è quello di riuscire ad eliminare l'interazione con il cloud, realizzando anche la fase di training all'edge. In questo modo, infatti, è possibile adattare il modello al meglio all'ambiente che lo circonda, migliorando le performance in

termini di velocità ed efficienza energetica.[6] Spostare l'esecuzione del training dal cloud all'edge su FPGA richiede l'adozione di nuove tecniche di ottimizzazione, sia per migliorare la gestione del numero limitato di risorse hardware disponibili, sia per ridurre i costi computazionali e di memoria. Nello specifico, infatti, la quantizzazione e la comprensione del modello non possono essere applicate in questo contesto in quanto degraderebbero le performance del modello di troppo durante l’inferenza. Pertanto, al fine di poter realizzare l’intera esecuzione dell’algoritmo di DL all’edge, è necessario ancora molto lavoro di ricerca sia in termini di miglioramento di design architetturale, sia di sviluppo di tecniche di ottimizzazione. [6]

BIBLIOGRAFIA

[1] Milenkovic, M. (2020). Internet of Things: Concepts and System Design. Springer Nature.

[2] Gosh, A., Chackraborty, D., & Law, A. (2018). Artificial intelligence in Internet of things. https://doi.org/10.1049/trit.2018.1008

[3] Nanni, L. (2021). IA_12. Lecture notes in Intelligenza Artificiale, 6. Retrieved 13 August 2022, Università degli Studi di Padova from https://elearning.dei.unipd.it/pluginfile.php/798805/mod_resource/content/15/IA_12.pdf.

[4] Applications of artificial intelligence - Wikipedia. En.wikipedia.org. (2022). Retrieved 11 August 2022, from https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence.

[5] Ng A., Charikar M., Ma T., and Re C., (2020) Lecture notes in CS229 - machine learning, from http://cs229.stanford.edu/.

[6] Casale, A. (2021) FPGA-based Deep Learning Inference Acceleration at the Edge. [MD Thesis, Politecnico di Torino], from https://webthesis.biblio.polito.it/17925/1/tesi.pdf

[7] Vogrig, D. (2021). 17_CNN.pdf Lecture notes in Digital Circuits for Neural Networks Università degli Studi di Padova.

[8] Applications of artificial intelligence - Wikipedia. En.wikipedia.org. (2022). Retrieved 11 August 2022, from https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence.

[9] Rete neurale feed-forward - Wikipedia. (2022). Retrieved 12 August 2022, from https://it.wikipedia.org/wiki/Rete_neurale_feed-forward

[10] Soriano, D. (2019). RNN - Come funzionano le Recurrent Neural Network - Domenico Soriano. Domenico Soriano. Retrieved 18 August 2022, from https://www.domsoria.com/2019/11/rnn-recurrent-neural-network/.

[11] convoluzionali, R., Casadei, C., OCR, C., minuti, P., & Realizzare una Electron App, s. (2019). Reti convoluzionali. maggiolidevelopers. Retrieved 17 August 2022, from https://www.developersmaggioli.it/blog/reti-convoluzionali/.

[12] T. -H. Tsai, Y. -C. Ho and M. -H. Sheu, "Implementation of FPGA-based Accelerator for Deep Neural Networks," 2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2019, pp. 1-4, doi: 10.1109/DDECS.2019.8724665.

[13] Stewart, M. (2019). Simple Introduction to Convolutional Neural Networks. Medium. Retrieved 16 August 2022, from https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac#:~:text=Convolutional%20layers%20are%20the%20layers,the%20size%20of%20the%20kernels.

[14] K. Choi and G. E. Sobelman, "Optimized Face Detection and Alignment for Low-Cost and Low-Power IoT Systems," 2020 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS), 2021, pp. 129-135, doi: 10.1109/IoTaIS50849.2021.9359713.

[15] Wang, C. (2018). How Does A Face Detection Program Work? (Using Neural Networks). Medium. Retrieved 13 August 2022, from https://towardsdatascience.com/how-does-a-face-detection-program-work-using-neural-networks-17896df8e6ff.

[16] Wendong Gai, Yakun Liu, Jing Zhang & Gang Jing (2021) An improved Tiny YOLOv3 for real-time object detection, Systems Science & Control Engineering, 9:1, 314-321, DOI: 10.1080/21642583.2021.1901156

[17] Miranda, P. R., Pestana, D., Lopes, J. D., Duarte, R. P., Véstias, M. P., Neto, H. C., & de Sousa, J. T. (2021, October 30). Configurable hardware core for IOT object detection. MDPI. Retrieved August 19, 2022, from https://www.mdpi.com/1999-5903/13/11/280/htm

[18] Long Short Term Memory | Architecture Of LSTM. Analytics Vidhya. (2017). Retrieved 19 August 2022, from https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/.

[19] Dobilas, S. (2022). LSTM Recurrent Neural Networks—How to Teach a Network to Remember the Past. Medium. Retrieved 19 August 2022, from https://towardsdatascience.com/lstm-recurrent-neural-networks-how-to-teach-a-network-to-remember-the-past-55e54c2ff22e?gi=346819d85010.

[20] M. Carreras, G. Deriu, L. Raffo, L. Benini and P. Meloni, "Optimizing Temporal Convolutional Network Inference on FPGA-Based Accelerators," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 10, no. 3, pp. 348-361, Sept. 2020, doi: 10.1109/JETCAS.2020.3014503.

[21] S. Saxena, S. Kumari, S. Kumar and S. Singh, "Text Classification for Embedded FPGA Devices using Character-Level CNN," 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), 2021, pp. 802-807, doi: 10.1109/ICACCS51430.2021.9441994.

[22] Silvano, C. (2021). Accelerazione hardware per il Deep Learning, tutte le soluzioni in campo. Agenda Digitale. Retrieved 15 August 2022, from https://www.agendadigitale.eu/cultura-digitale/accelerazione-hardware-per-il-deep-learning-tutte-le-soluzioni-in-campo/.

[23] C. Yang, "FPGA in IoT Edge Computing and Intelligence Transportation Applications," 2021 IEEE International Conference on Robotics, Automation and Artificial Intelligence (RAAI), 2021, pp. 78-82, doi: 10.1109/RAAI52226.2021.9507835.

[24] Wu, R.; Guo, X.; Du, J.; Li, J. (2021) Accelerating Neural Network Inference on FPGA-Based Platforms—A Survey. Electronics 2021, 10, 1025. https://doi.org/10.3390/electronics10091025

[25] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang and Huazhong Yang. 2017. [DL] Survey of FPGA-Based Neural Network Inference Accelerator. ACM Trans. Reconfig. Technol. Syst. 9, 4, Article 11 (December 2017), 26 pages.

[26] FPGAs and eFPGAs Accelerate ML Inference at the Edge (WP026) | Achronix Semiconductor Corporation. Achronix.com. (2021). Retrieved 20 August 2022, from https://www.achronix.com/documentation/fpgas-and-efpgas-accelerate-ml-inference-edge-wp026.

[27] C. Yang, "FPGA in IoT Edge Computing and Intelligence Transportation Applications," 2021 IEEE International Conference on Robotics, Automation and Artificial Intelligence (RAAI), 2021, pp. 78-82, doi: 10.1109/RAAI52226.2021.9507835.

[28] J. Chen, S. Hong, W. He, J. Moon and S. -W. Jun, "Eciton: Very Low-Power LSTM Neural Network Accelerator for Predictive Maintenance at the Edge," 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), 2021, pp. 1-8, doi: 10.1109/FPL53798.2021.00009.

[29] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04., 2004, pp. 69-70, doi: 10.1109/MEMCOD.2004.1459818.

[30] Project IceStorm - Claire's Homepage. (2022). From: https://clifford.at/icestorm

[31] Demystifying LSTM Weights and Biases Dimensions. Medium. (2020). Retrieved 7 September 2022, from https://medium.com/analytics-vidhya/demystifying-lstm-weights-and-biases-dimensions-c47dbd39b30a.