UNIVERSITÀ POLITECNICA DELLE MARCHE
DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Elettronica

Sviluppo di un sensore CNN-based per il pedestrian detection

Development of a CNN-based sensor for pedestrian detection

Relatore
Prof. Claudio Turchetti

Correlatore
Dott.ssa Laura Falaschetti

Studente
Simone Crisologo

Anno Accademico 2020/2021

Indice

Elenco delle tabelle

Elenco delle figure

Sommario

Questo elaborato rappresenta uno studio di fattibilità e conseguente implementazione di un sistema di pedestrian detection, in ambito Automotive, su dispositivi che si contraddistinguono per un quantitativo di risorse computazionali limitate e un costo ridotto. Il problema appartiene al dominio dell'Object Detection, sotto categoria della Computer Vision, e viene affrontato con tecniche di Deep Learning. Viene quindi proposta un'architettura convoluzionale adoperante tecniche di compressione innovative, caratterizzata da una numerosità dei parametri ridotta, e se ne valutano le prestazioni relative rispetto all'attuale stato dell'arte.

Capitolo 1

Introduzione

Nell'ambito della Computer Vision, il task dell'Object Detection rappresenta oggi una sfida aperta. Sebbene, grazie al crescente interesse del pubblico e dell'industria, siano stati sviluppati metodi di detection sempre più avanzati, il focus delle tecniche proposte è rivolto soprattutto all'accuratezza con cui si dichiara la categoria dell'oggetto tracciato e la sua posizione. Questo ha portato all'abbandono di strategie derivanti dalla Computer Vision classica, quali modelli Haar-Cascade[1], in favore dell'uso di reti neurali, di base convoluzionale, via via più articolate e stratificate e in grado di astrarre meglio l'informazione di input loro fornita.

Negli ultimi anni, data la crescente necessità di hardware sempre più performante e tempi d'inferenza marginalmente ridotti, questa tendenza è cambiata in favore di una rivisitazione dell'architettura tipica di un object detector, che passa così da reti con region proposal (fast/faster rcnn) a modelli definiti *single shot*, ovvero che producono dei candidati, frame per frame, in un unico step.

Sebbene alcuni modelli di rete neurale siano oggigiorno disponibili per un processo di fine-tuning rapido e mirato, questi non coprono spesso le esigenze di ambiti industriali e casi d'uso specifici ma si propongono, piuttosto, come soluzioni generiche al problema. Conseguentemente, dal punto di vista della pedestrian detection, la principale problematica che ne emerge è un tempo d'inferenza eccessivo e inadeguato, quindi, a situazioni quotidiane quali la sicurezza nella navigazione autonoma dei

veicoli.

In questo elaborato si prende in esame il problema dell'object detection con un'attenzione particolare al rispetto dei requisti di operazione real time e si valutano le possibilità implementative su hardware specializzato o con risorse limitate che consenta un'adeguata flessibilità economica nello scaling.

L'elaborato è organizzato come segue: nel Capitolo 2 si illustrano alcuni principi base della Computer Vision, concentrando l'attenzione sul ruolo che le moderne tecniche di Deep Learning hanno nel dominio della Object Detection; nel Capitolo 3 si introducono i dispositivi hardware presi in esame, il comparto software loro associato comprensivo delle librerie per lo sviluppo degli algoritmi di machine learning, i dataset di riferimento e le architetture dei modelli scelti; nel Capitolo 4 viene illustrata la metodologia con la quale si è eseguito il training delle reti selezionate e quali metriche si sono osservate, viene quindi riportato quanto conseguito sperimentalmente per i singoli dispositivi; nel Capitolo 5 infine si discutono l'efficacia dei metodi testati e le implicazioni che questi hanno dal punto di vista dell'ottimizzazione delle reti in ambito *embedded*.

Capitolo 2

Object Detection Tramite Tecniche di Deep Learning

La Computer Vision consiste nell'analisi automatizzata d'immagini allo scopo di estrapolarne dati utili al processo in esame. Queste informazioni possono essere di basso livello, quali la ricerca di contorni, vettori di movimento dei pixel o l'individuazione di elementi peculiari dell'immagine, o di alto livello come nel caso dell'interpretazione del contesto e la segmentazione semantica. Più in generale, il compito della Computer Vision può essere così riassunto:

> *"At an abstract level, the goal of computer vision problems is to use the observed image data to infer something about the world."*[1]

È chiaro come la possibilità di automatizzare un simile processo apra a un numero considerevole di applicazioni, sia a fini industriali, dove è d'interesse limitare la necessità d'intervento umano o sono richieste azioni e risposte immediate, sia per usi più vicini all'individuo, dove l'assistenza fornita da un sistema autonomo consente una maggior sicurezza e, generalmente, un ridotto impegno da parte dell'uomo.

Oggi, tecniche di *computer vision* vengono applicate a telecamere monoculari, telecamere stereo e a sistemi integrati in grado di percepire frequenze al di fuori

[1] Prince, Simon JD. Computer vision: models, learning, and inference. Cambridge University Press, 2012.

dello spettro visibile, fornendo una ricostruzione della profondità. A seconda del task a cui sono dedicati, questi apparati posseggono un *Field of View* che spazia dalla visione puntuale del dettaglio, utilizzata tipicamente in processi di misurazione *contactless* dove la perturbazione del sistema tramite misurazione renderebbe il processo impreciso, lento o più complesso, a una visione assimilabile a quella umana, adoperata ad esempio per le rilevazioni sul territorio, fino ad angoli superiori a 180° per l'analisi dell'intero ambiente circostante.

Negli ultimi anni, in particolare, la crescita del settore dell' *automotive*, indirizzato sempre più verso la produzione e l'uso di mezzi elettrici, ha portato con sé anche una forte domanda per metodi di navigazione avanzati e al servizio dell'uomo. Quanto un tempo disponibile solo in ambito di ricerca per la robotica viene ora integrato nei sistemi *ADAS* per fornire meccanismi di sicurezza e cruising in grado di reagire all'ambiente circostante con tempi ridotti, garantendo al conducente un aiuto immediato nelle situazioni di potenziale pericolo (assistenti al sorpasso, sistemi di rilevamento della carreggiata, etc.) o un maggiore comfort nella guida (*adaptive cruise control*, guida autonoma).

Punto cruciale di queste applicazioni è la necessità di garantire la sicurezza sia di chi è alla guida del mezzo, sia di chi si trova nei suoi pressi, con interventi tempestivi. Per questa ragione un ambito di studio particolarmente d'interesse è oggi la *Pedestrian Detection*[2][3][4][5][6], ovvero il processo di rilevamento della presenza di pedoni unito al riconoscimento della loro posizione nello spazio.

2.1 Tecniche classiche di Object Detection

2.1.1 Il concetto di *feature*

I primi metodi di analisi di un'immagine come matrice di valori sono nati, in passato, nel tentativo d'individuare alcune caratteristiche salienti della stessa adoperando algoritmi deterministici. A partire da una definizione a priori dell'elemento cercato all'interno del frame, si sono sviluppate nel tempo tecniche per l'estrazione d'infor-

mazione tali da fornirne una rappresentazione immediata e facilmente processabile. Un esempio è il delineamento degli "edge" di un oggetto, ovvero i contorni: tramite l'analisi dei gradienti di colore o luminosità individuabili nella matrice di pixel è infatti possibile individuare variazioni repentine indicanti, a seguito di opportuni filtraggi, la posizione e direzione dei contorni di ciò che si sta osservando.

Quando individuati, intorni di valori con caratteristiche rilevanti all'elaborazione dell'immagine vengono definiti *feature* e raccolti in vettori descrittori (*Feature Descriptor*). Un esempio sono gli Histogram of Oriented Gradients[7] per la rappresentazione dei gradienti e del loro orientamento, espressi in vettori di 3780 elementi.

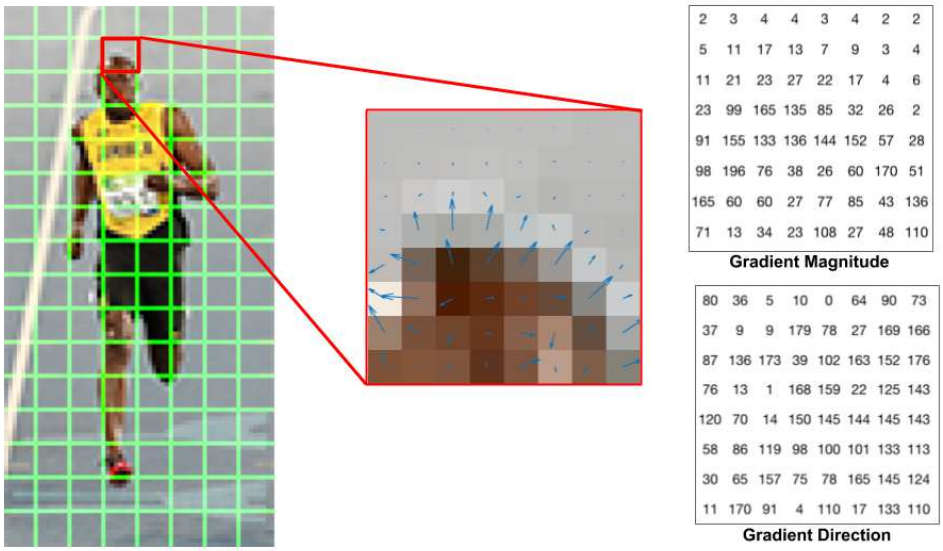


Figura 2.1: Histogram of Oriented Gradients

2.1.2 Detector classici

Nei sistemi di detection antecedenti la diffusione delle tecniche convoluzionali, il processo di estrazione dell'informazione e seguente uso della stessa per la determinazione della posizione di un target nell'immagine richiedeva, quindi, una chiara definizione delle *feature* d'interesse e della loro correlazione con l'oggetto da individuare.

Metodi come questi consentono un'elaborazione rapida e una stima accurata

del carico computazionale massimo proposto dal task, rendendoli particolarmente adeguati a implementazioni compatte e adattabili a sistemi con ridotte performance. Nel tempo si sono dimostrati, però, limitati dalla capacità dell'uomo di indirizzare opportunamente la ricerca verso quanto richiesto a seconda del dominio di appartenenza del problema affrontato. Non sempre è possibile, infatti, esprimere la caratteristica d'interesse in termini facilmente traducibili in algoritmi a diretta integrazione. Alcune tecniche di Object Detection così elaborate hanno consentito, ad esempio, processi di *face detection* estremamente rapidi in quanto realizzati "su misura" per l'obiettivo preposto, ma che non consentirebbero una detection adeguata di oggetti differenti dal volto umano, mostrando un chiaro limite nella flessibilità applicativa.

2.1.3 Haar-Cascade

All'estremo delle implementazioni classiche nella Computer Vision troviamo il detector Haar-Cascade[1]. Esso si basa sull'individuazione di oggetti a partire da una conoscenza pregressa delle edge-feature considerate significative. Tale conoscenza viene costruita con tecniche di machine larning, sulla base di dataset annotati, e utilizzata in un processo a cascata di filtraggio delle caratteristiche descrittive dell'immagine assimilabile a un albero decisionale.

Il deep learning ha oggi delegato, come visto in precedenza, l'uso di queste tecniche, un tempo stato dell'arte della object detection, all'uso in dispositivi con capacità elaborative estremamente limitate o dove il tempo d'inferenza è più rilevante dell'accuratezza con cui si effettua la detection.

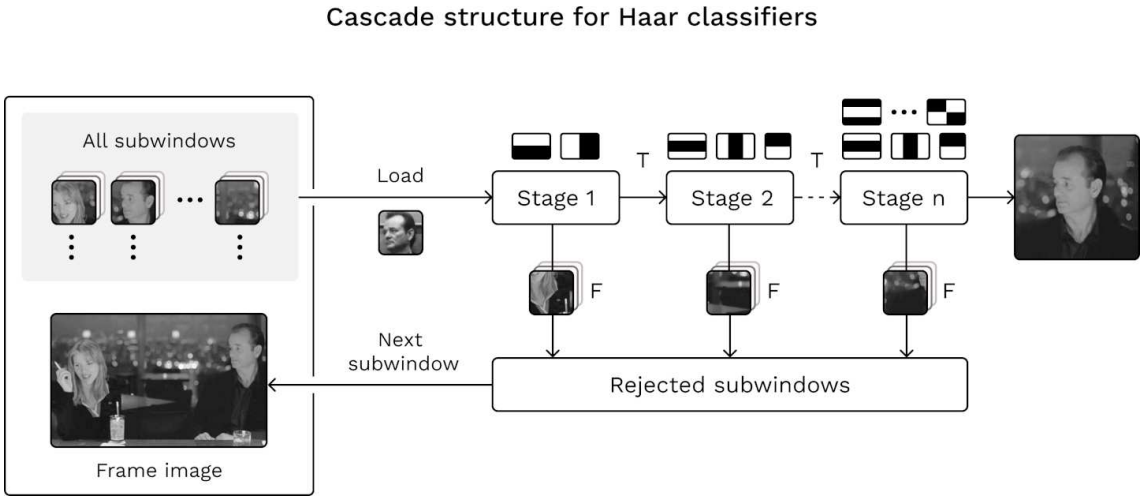


Figura 2.2: Struttura dell'Haar Cascade

2.2 Modelli convoluzionali e deep learning

2.2.1 Artificial Neural Network

Le reti neurali nascono come imitazione del processo di apprendimento per rafforzamento caratteristico dalle loro equivalenti biologiche. Ne conservano l'architettura a nodi e sinapsi, modellati rispettivamente come funzioni di attivazione e pesi di propagazione, costituendo, nella loro struttura più semplice di *Dense Neural Network*, un approssimatore universale di funzioni.

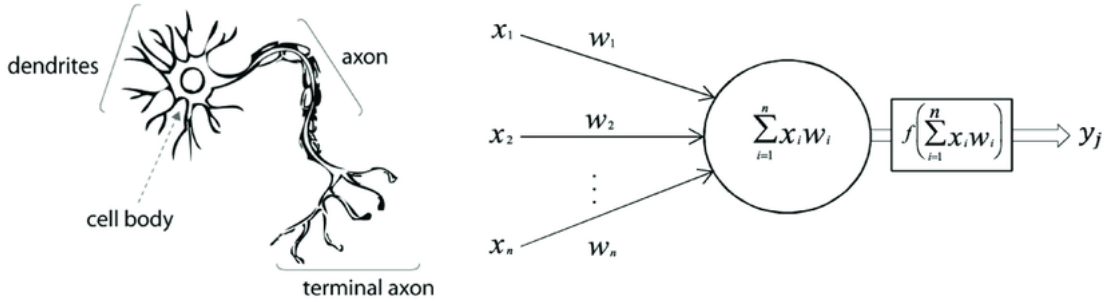


Figura 2.3: Dalle reti biologiche al modello matematico

I segnali in ingresso alla rete, così, si propagano a partire dai primi neuroni costituenti il *layer* di input, subendo lungo il percorso semplici operazioni di somma,

moltiplicazione per parametri e l'applicazione di meccanismi di soglia scelti in funzione della tipologia di problema affrontato, per arrivare in output, spesso alterati in numero, a rappresentare la risposta del modello allo stimolo fornito.

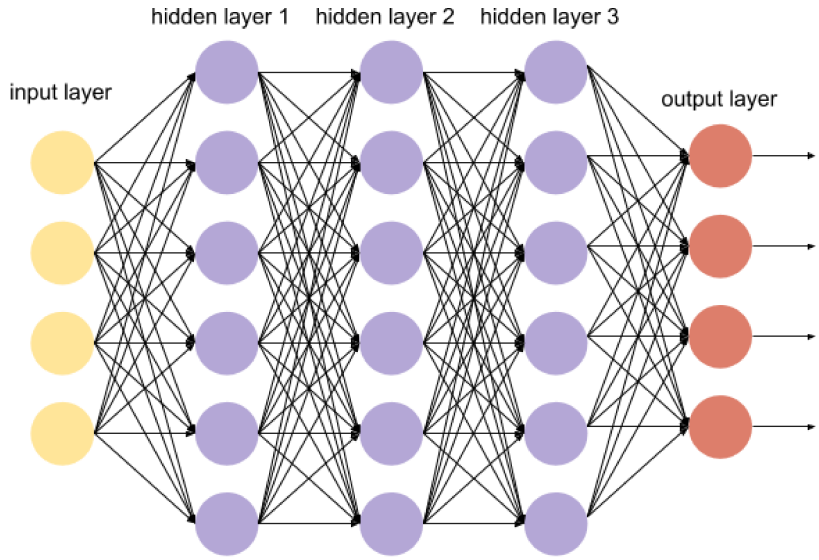


Figura 2.4: Dense Neural Network

Come l'equivalente biologico, la rete necessita di un'importante fase di apprendimento iterativa. A partire da un *dataset* contenente elementi in ingresso e output desiderati, mediante funzioni dette di *loss* e modellate sulla tipologia di compito che la rete andrà a svolgere, si elabora la distanza tra quanto prodotto dalla DNN e ciò che viene considerato *ground-thruth*. Matematicamente, questo consiste nel valutare il gradiente locale della superficie descritta dall'errore, con l'obiettivo di spostarsi all'interno dello spazio delle soluzioni nella direzione di un minimo dello stesso, in un'operazione detta, appunto, *discesa del gradiente*. L'esito di questa valutazione comporta un aggiornamento dei parametri interni del modello, quali i pesi delle interconnessioni, con un meccanismo di propagazione inversa, fino a raggiungere una condizione che produca risultati considerati accettabili secondo delle metriche scelte.

Questa implementazione mostra facilmente un punto di forza dell'approccio già sperimentato nel machine learning e ora esteso all'intera architettura di rete: è

possibile realizzare uno strumento che sia in grado di svolgere un task, definito da input fornito e output desiderato, senza la conoscenza a priori di quali siano, come nel caso della Computer Vision, le feature che il sistema debba saper riconoscere.

È importante evidenziare come non sia quindi necessario sviluppare un processo decisionale a partire da feature note, quali ad esempio gli HOG[7], ma è possibile invece far sì che sia il metodo di ottimizzazione del modello di rete scelto a estrapolare quanto di significativo, relativamente al compito preposto, ci sia nell'informazione in ingresso.

2.2.2 Convoluzione nell'image classification

Nella Object Detection e, ancor prima, nella Image Classification, il dominio entro il quale viene definito il problema consente una preventiva conoscenza della tipologia di input processato e, di conseguenza, l'introduzione di ulteriori vincoli sulla tipologia di operazioni svolte all'interno della rete e di meccanismi di relazione spaziale tra i pixel dell'immagine. L'uso dell'operatore di convoluzione, infatti, porta una mutua influenza tra valori tra loro prossimi all'interno della matrice, comportamento che meglio rispecchia la realtà di ciò che si osserva dove, tipicamente, sussiste forte correlazione tra zone contigue appartenenti al medesimo oggetto.

La convoluzione, concretamente, consiste nell'applicazione di un filtro quadrato scorrevole, di dimensione variabile, sull'intero input. A ogni step la regione evidenziata dalla matrice filtrante vede i suoi valori moltiplicati per l'entità del filtro e sommati in un unico valore che andrà a popolare la matrice di output. Questa operazione aiuta a delineare l'informazione contenuta nell'immagine esaminata, condensandone i valori contenuti lungo le sue dimensioni di *altezza*, *larghezza* e numero di *canali* (generalmente associati ai colori).

Oltre al ridurre la dimensione dello spazio delle soluzioni ottime per una rete neurale fornendo un vincolo relazionale, questa tipologia di elaborazione si presta particolarmente, inoltre, all'implementazione hardware. Ogni operazione di convoluzione n-dimensionale può essere vista, semplificando, come una successione di

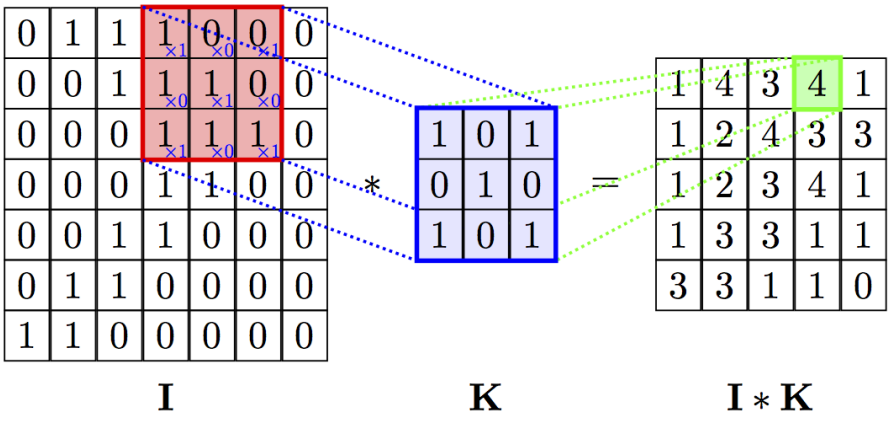


Figura 2.5: Operazione di convoluzione

moltiplicazioni e *somme* (indicate con il termine *MAC - Multiply and Accumulate*), facilmente mappabili a equivalenti hardware forniti dall'architettura in uso.

2.2.3 Astrazione dell'informazione

La ricerca di modelli sempre più articolati e performanti ha portato a una convergenza verso architetture definibili *"deep"*, costituite cioè da un numero elevato di strati, o layer, tra loro consecutivi, con l'obiettivo di ottenere una proporzionale capacità astrattiva nei riguardi dell'input immesso.

È infatti noto come l'abilità di una rete nell'individuare elementi significativi in scenari complessi derivi dalla possibilità che la sua struttura offre di costruire associazioni complesse tra le informazioni che ne percorrono i rami di processamento. Tante più le operazioni coinvolte in successione, quanto più è il livello di astrazione che il dato finale matura rispetto al dato in ingresso.

Una topologia comune, più volte rivisitata, è quella adoperata nella rete convoluzionale VGG-16 o nella sua espansione VGG-19, derivante dalla prima proposta AlexNet: l'operatore principale è il layer di convoluzione, presente in più fasi della catena input-output e applicato a tensori in ingresso via via più compatti e numerosi, seguito ciclicamente da layer di MaxPooling.

Quest'ultima tipologia di layer attua uno scaling delle dimensioni dei tensori in

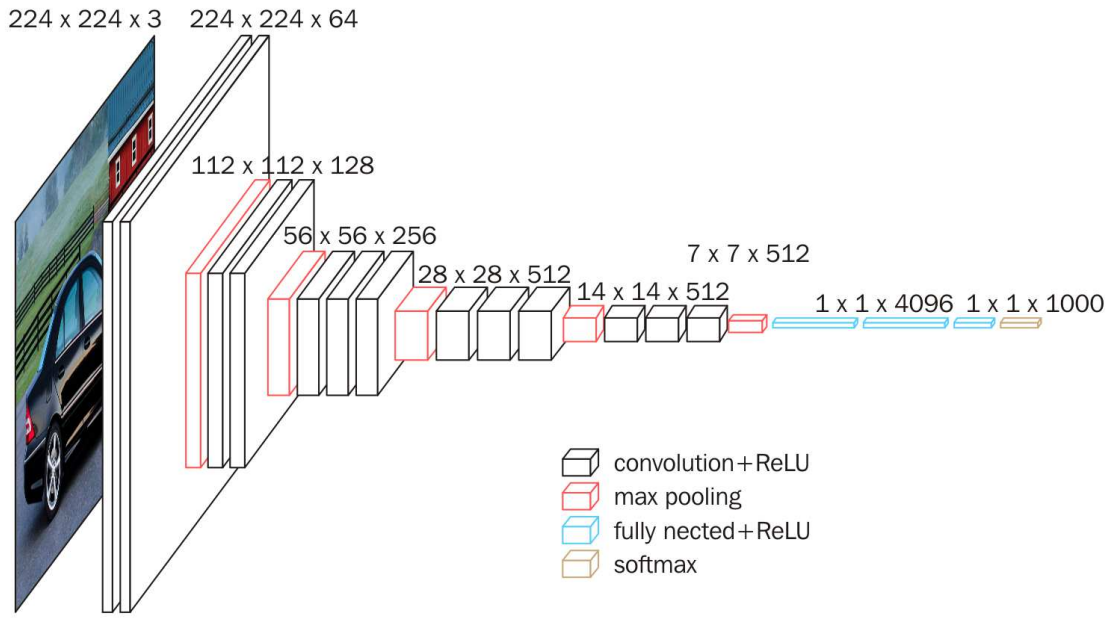


Figura 2.6: Architettura VGG-16

ingresso secondo una regola di conservazione dei massimi, andando cioè a campionare sotto intervalli regolari dei valori presenti e preservandone solo l'elemento costituente il massimo. L'operazione così ottenuta è a tutti gli effetti una compressione *lossy* dell'informazione che dà la possibilità alla rete di preservare solo quanto significativo tramite un adattamento dei parametri in fase di allenamento, ma che consente soprattutto di mantenere un carico computazionale contenuto all'aumentare della numerosità dei filtri introdotti.

L'ultimo step di effettiva classificazione viene eseguito da una rete *fully connected* il cui output, sottoposto a funzioni di attivazione *softmax*, rappresenta la probabilità, in range [0..1], che quanto osservato appartenga alla determinata classe.

2.2.4 Skip-connection e blocchi residuali

Nel processo di training di una rete neurale è l'ottimizzazione iterativa dei parametri interni che porta il modello ad adattare la propria risposta al task individuato. Questo metodo, *backpropagation*, necessita della definizione di una funzione di *loss*, rappresentante l'errore di predizione, sulla quale adeguare i pesi delle connessioni

interne tra i layer. Nota la curva di errore, iterativamente se ne valuta il gradiente lungo le direzioni dello spazio delle soluzioni individuato, con l'intento di *discendere* la curva verso un minimo che sia *assoluto*.

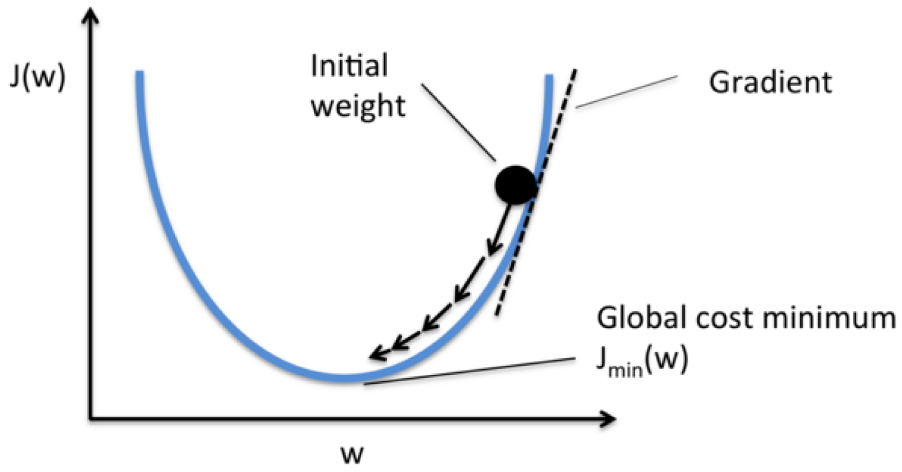


Figura 2.7: Procedura di discesa del gradiente della funzione di errore o costo

Un problema comune che si può riscontrare nello sviluppo di modelli neurali complessi è, però, l'*evanescenza del gradiente*, fenomeno per il quale il processo di *training* di una rete diventa via via più arduo e soggetto al rischio di falsi minimi con l'aumentare della profondità della rete e, conseguentemente, della profondità di propagazione dei gradienti.

L'errore calcolato sulla predizione, infatti, in fase di apprendimento, va ad aggiornare il valore dei parametri (o pesi) associati alle operazioni presenti nel modello, con un processo che può essere ricondotto a una catena di moltiplicazioni. Ciò che spesso emerge è che l'entità dell'errore, valore tipicamente molto inferiore all'unità, diventa, man mano che viene propagato, troppo piccola per apportare un contributo significativo all'apprendimento della rete.

Una soluzione a questo problema è data dalle skip-connection, ovvero delle connessioni tra i layer del modello che "saltano" degli step di propagazione e si congiungono direttamente al livello target. L'utilità di questa architettura a rami paralleli può essere spiegata in termini di contributo alla backpropagation o di

apporto informativo.

Dal punto di vista della propagazione dell'errore, quello che risulta è che i layer immediatamente successivi all'input, normalmente poco soggetti ad aggiornamenti significativi dei parametri per via della profondità della catena di moltiplicazioni, vengono resi artificialmente più *vicini* all'origine della propagazione e quindi più facilmente adattabili tramite processi di ottimizzazione.

In maniera più intuitiva si può osservare come l'informazione che entra nel modello, nel discenderne l'architettura e subirne le alterazioni dettate dai layer coinvolti, perde sempre più un legame diretto con quanto originariamente descritto nel tensore di input. Benché questo aiuti significativamente nel processo di astrazione, è utile alle volte conoscere contemporaneamente l'origine e quanto prodotto dalle operazioni di convoluzione e scaling per attuare un processo decisionale adeguato al compito preposto.

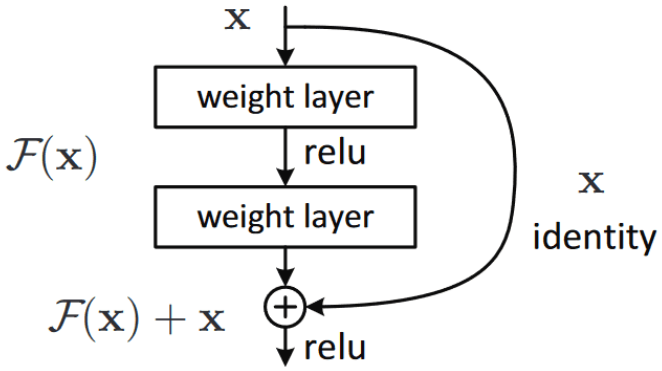


Figura 2.8: Skip-connection nel *blocco residuale*

Per tali ragioni l'architettura convoluzionale pura di modelli come VGG-16 ha visto l'introduzione, in varie pubblicazioni, di rami di propagazione paralleli con ricombinazione dell'output, fino alla definizione di una unità elaborativa chiara che implementasse sistematicamente il concetto delle skip-connection.

Il blocco, introdotto quindi nella rete *ResNet*, viene detto *residuale* e propone, in uscita, una somma dei rami come alternativa alla concatenazione, presente in

architetture come la DenseNet, per la regressione mediante upscaling delle feature, come la U-Net.

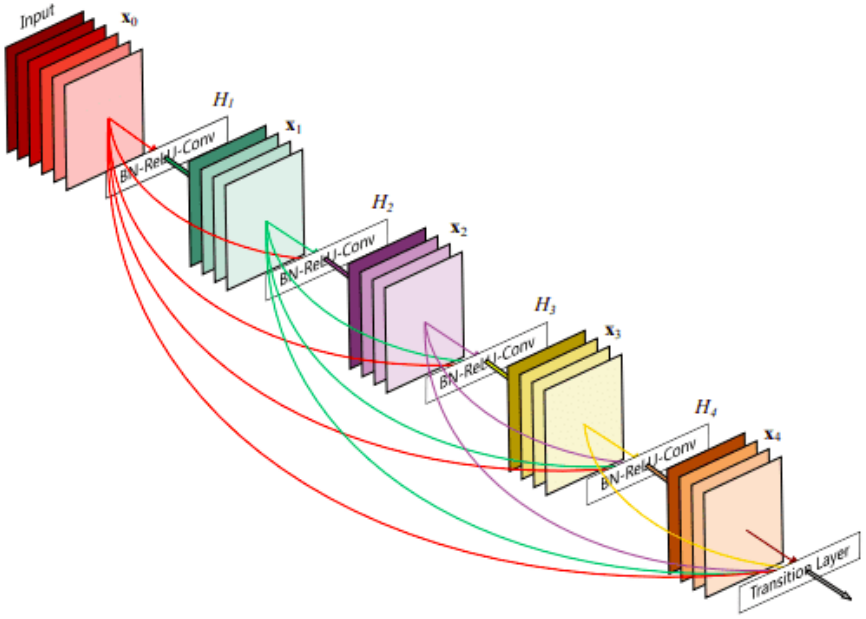


Figura 2.9: DenseNet skip-connection con concatenazione

Da qui si sono susseguiti vari tentativi di evoluzione scaturiti spesso in architetture note per l'elevata capacità di astrazione, come le *Inception V2* e *Inception V3* proposte da *Google*, ma note anche per l'elevato numero di parametri coinvolti nel processo di *training* e di operazioni necessarie durante l'inferenza, tanto da obbligarne l'uso solamente su hardware prestazionale o attraverso soluzioni *cloud* dedicate. Negli ultimi anni l'attenzione si è spostata verso l'esecuzione di modelli su dispositivi portatili, nel tentativo di una riduzione del traffico e del carico dei server fino ad allora impiegati, della latenza d'inferenza e, sotto alcuni aspetti, per questioni riguardanti la privacy degli utenti.

Questa nuova necessità implementativa, con il nome di *edge computing*, ha portato a reti con un ridotto numero di parametri e che impiegano spesso tecniche di compressione della struttura, o riduzione del carico computazionale, come nel caso delle proposte *MobileNet V1*[8] e *MobileNet V2*[9], oggi standard *de facto* della

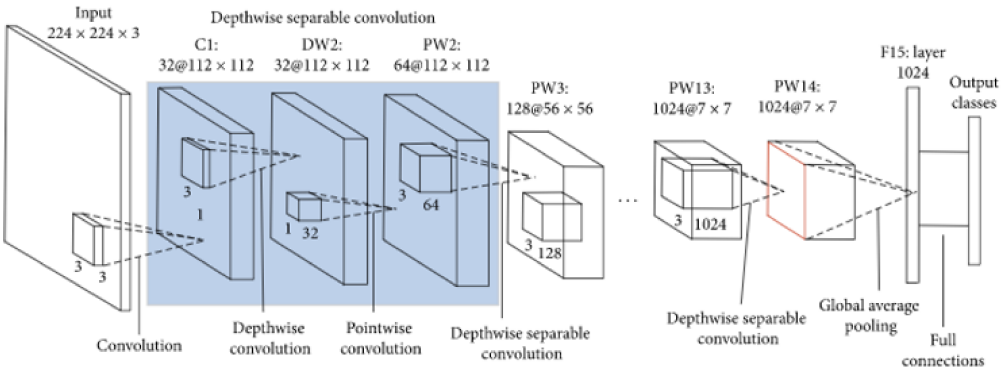


Figura 2.10: Architettura MobileNet

classificazione su dispositivi *low power*.

2.2.5 Region Proposal Networks

Il passo successivo alla classificazione dell'immagine nella Computer Vision è sicuramente il processo di *object detection*, che consiste nell'individuare la posizione del soggetto riconosciuto all'interno del frame dato dall'immagine che si sta esaminando. Il task così individuato vede, nell'ambito del deep learning, metodi e modelli prendere forte spunto dal lavoro svolto in precedenza dalle reti convoluzionali.

L'evidente abilità nell'estrapolare feature utili alla descrizione dell'input, ed eventuale suo riconoscimento, diventa la base di partenza di modelli come la *R-CNN*[10] e le sue derivate *Fast*[11] e *Faster*[12].

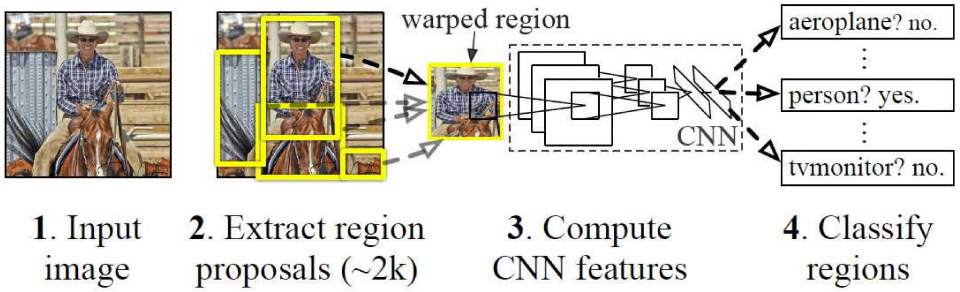


Figura 2.11: R-CNN con Region Proposal

Nella *R-CNN*[10] il processo di classificazione tipico delle Convolutional Neural

Network segue un'analisi dell'immagine detta "selective search." Questo consiste in un metodo di *region proposal* che estrae un numero prefissato di aree dell'immagine con più elevata probabilità di contenere oggetti, da poi sottoporre, ridimensionate, al classificatore.

La strategia implementata, considerata fondamentale alla sua introduzione, è però divenuta nota per i tempi di elaborazione considerevoli di cui necessita e il quantitativo di memoria su disco richiesta dall'archiviazione temporanea delle *ROI* (*region of interest*) prodotte.

A essa sono infatti seguite revisioni che, nella più recente *Faster R-CNN*[12], portano la fase di *region proposal* a essere parte integrante del modello, nella forma di una rete disposta sul ramo parallelo alla diretta propagazione dell'input. Non operando più sull'immagine in ingresso ma sulla mappa di feature generate dalla porzione convoluzionale che la precede, la rete è in grado di generare regioni candidate molto più rapidamente e con più elevata precisione, rendendo oggi la *Fast R-CNN*[11] una delle implementazioni di riferimento per un Object Detector *accurato*.

2.2.6 Single shot detection

Ciò nonostante, il tempo d'inferenza richiesto da architetture derivate da reti con meccanismi di *region proposal* è tipicamente elevato e richiede ingenti capacità di elaborazione, elementi non sempre compatibili con quanto disponibile in ambiti come il mobile computing, la guida autonoma e, in generale, situazioni dove la priorità è mantenere consumi energetici ridotti.

Lo studio fino ad allora condotto sulla Object Detection cambia ottica con l'introduzione dell'architettura YOLO (You Only Look Once) e il principio della *single shot detection*. Esso si basa sull'idea che la ricerca di regioni d'interesse all'interno dell'immagine osservata, processo che richiede implicitamente valutazioni multiple sullo stesso frame, può essere sostituita da un set predefinito di aree osservate, purché se ne specifichi il rapporto d'aspetto e, affinché siano competitive in termini

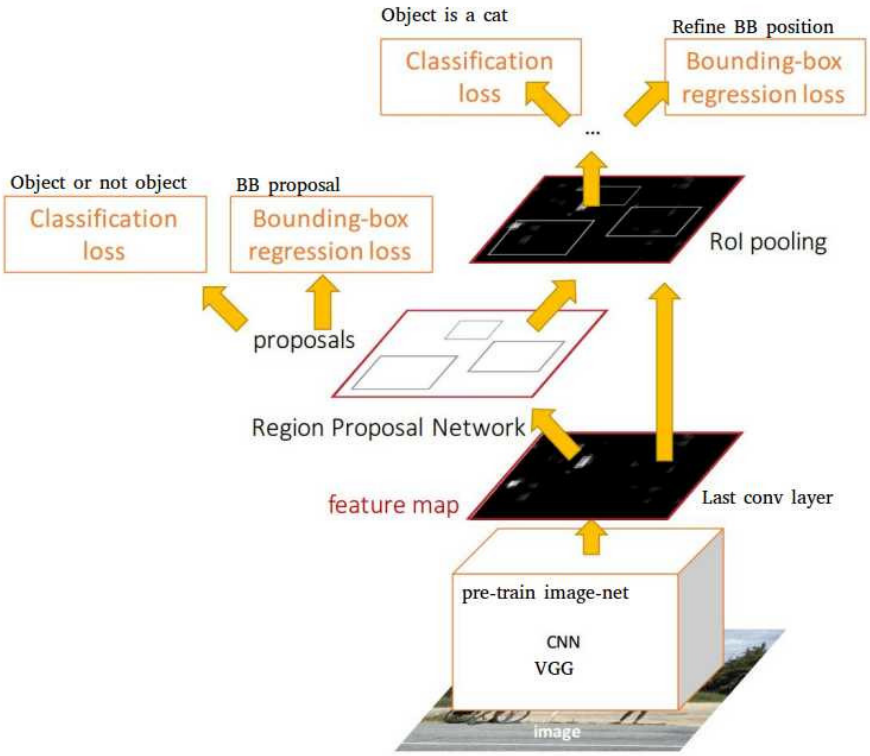


Figura 2.12: Faster R-CNN

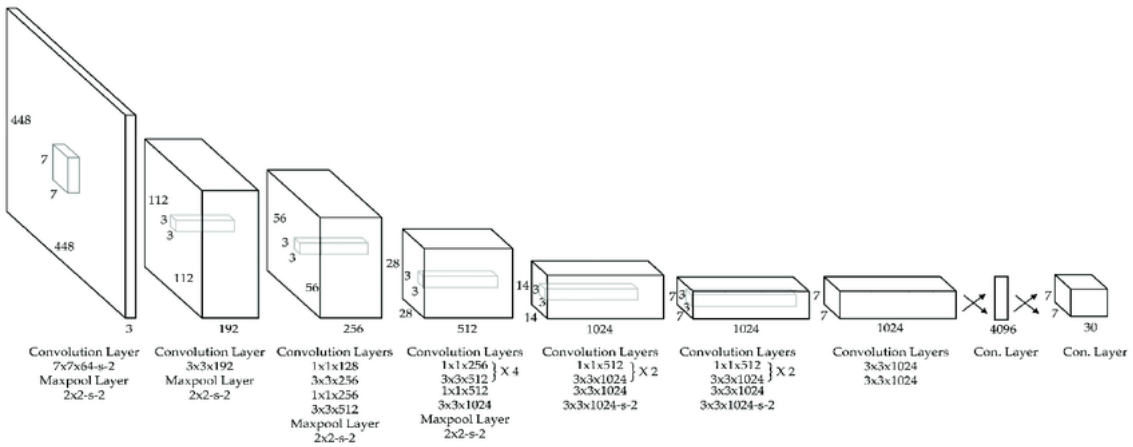


Figura 2.13: YOLO, architettura della prima revisione

di accuratezza, le si scelga in funzione della probabilità che queste contengano un oggetto valido.

Tali regioni d'interesse vengono definite, a seconda dello specifico modello in esame, *anchor boxes* o *prior boxes*, e ricevono l'output di convoluzioni effettuate in punti diversi lungo la catena di processing del modello impiegato. In questa maniera si riesce a esaminare l'input con aree di dimensione *relativa* diversa e a massimizzare le capacità di detection.

Le coordinate delle *bounding box* prodotte dal modello, sono associate, in fase d'inferenza, a uno o più termini indicanti la probabilità che ognuna delle aree contenga una classe nota e di quale oggetto si tratti.

È però probabile, in questa maniera, che le predizioni ottenute siano costituite da un elevato numero di ROI con esito positivo ma tra loro sovrapposte. Segue quindi un processo di filtraggio e selezione dei migliori risultati con tecniche di *Non-Maximum Suppression* e di *Top-k Sorting*.

I modelli oggi più diffusi appartengono a tre principali rami di sviluppo degli Object Detector, ovvero:

- YOLO[13][14][15], nella versione originale e nella variante *Tiny*;
- SSD[16], con estrattori di feature (reti *backbone*) ereditate da MobileNet[8][9], ResNet[17] e CenterNet[18];
- EfficientDet, con architettura derivata da EfficientNet.

Come anticipato, questi si propongono non tanto come reti concorrenti in accuratezza, sebbene ottengano oggi punteggi paragonabili allo stato dell'arte, quanto soluzioni caratterizzate da un'estrema rapidità di esecuzione e, in alcune varianti, da una dimensione ridotta tale da permetterne un'implementazione funzionale in ambito *embedded*.

2.3 Tecniche di compressione

2.3.1 Depthwise-Pointwise convolution

Parte della complessità di un modello convoluzionale deriva dalla necessità di operare su tensori tridimensionali, dati da altezza, larghezza e numero di canali di un'immagine, richiedendo, quindi, un elevato numero di operazioni *Multiply and Accumulate*. Questo comporta una relazione di proporzionalità, attesa, tra le capacità della rete e il costo, in termini di latenza e memoria, che ha un suo processo di predizione.

Una alternativa viene presentata con l'introduzione delle architetture Xception e MobileNet[8], delle quali l'ultima con evidente focus sull'esecuzione in dispositivi mobili in ottica di *edge computing*. Esse sostituiscono l'operazione di convoluzione spaziale classica, operante sui tre canali, tipicamente RGB (*red*, *green*, *blue*), dell'immagine, con una sua decomposizione in due fasi.

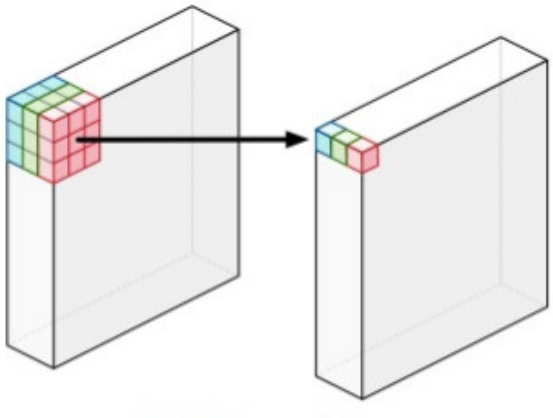


Figura 2.14: Convoluzione "depth-wise"

La prima consiste in una convoluzione applicata canale per canale, detta *"Depthwise Convolution"*, seguita poi da una seconda di ricombinazione degli output mediante la *"Pointwise Convolution"*, una convoluzione cioè che interessa puntualmente i singoli valori delle matrici così ottenute e agisce, dimensionalmente, lungo tutti i canali.

Riformulare l'operazione in questi due *fattori* consente un risparmio computa-

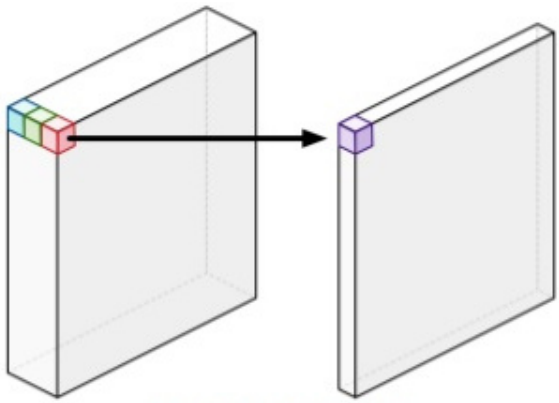


Figura 2.15: Convoluzione “point-wise” sui canali

zionale importante e, soprattutto, proporzionale al numero di canali su cui si vuole operare. Il computo di una convoluzione spaziale classica genera una *feature map* G con la seguente operazione:

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m}$$

e possiede, quindi, una complessità esprimibile come:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

dove D_K è la dimensione del kernel, D_F la dimensione della *feature map* e M, N rappresentano la numerosità dei canali d’input e output.

La convoluzione Depthwise viene invece espressa per il singolo canale come:

$$G_{k,l,n} = \sum_{i,j} K_{i,j,m} \cdot F_{k+i-1,l+j-1,m}$$

Osservando quest’ultima si può evidenziare una complessità data da:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

con assenza del contributo N dato dalla combinazione dei canali di output. Questa fase viene sostituita quindi dalla successiva convoluzione Pointwise, con una complessità lineare che porta il costo dell’operazione a essere pari a:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

Ciò che ne risulta è un netto guadagno computazionale dell'ordine di $\frac{1}{N} + \frac{1}{D_K^2}$ che consente l'esecuzione di architetture come la MobileNet anche su hardware dalle risorse limitate quali i dispositivi mobili, con un minimo effetto sulla complessiva accuratezza dei risultati.

2.3.2 Decomposizione CP

Questa tecnica di compressione dell'informazione consente di rappresentare un tensore tramite la sua decomposizione in fattori a rango unitario. In figura 2.16 si ha una rappresentazione visiva del processo per un tensore tridimensionale.

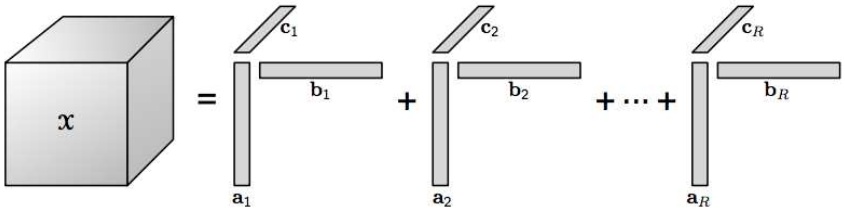


Figura 2.16: Decomposizione CANDECOMP/PARAFAC

Se i fattori sono espressi in forma normalizzata, con dimensione unitaria, la loro composizione può essere quindi espressa come:

$$\boldsymbol{\mathcal{X}} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}^{\mathbf{r}} \circ \mathbf{b}^{\mathbf{r}} \circ \mathbf{c}^{\mathbf{r}} \tag{2.1}$$

dove λ_r è uno scalare opportuno. È quindi possibile esprimere un dato tensore $\boldsymbol{\mathcal{X}}$ come combinazione lineare tramite *prodotto esterno* dei suoi elementi $\mathbf{a}, \mathbf{b}, \mathbf{c}, ...$ a meno di un errore di approssimazione. Per il generico $\S_{ijk}$ avremo infatti:

$$x_{ijk} \approx \hat{x}_{ijk} = \sum_{r=1}^{R} \lambda_r a_i^r b_j^r c_k^r \tag{2.2}$$

La tecnica di decomposizione consiste, concretamente, nell'applicazione di un algoritmo di minimizzazione dell'errore introdotto dal troncamento, noto come *alternating least square* (ALS), che, fissato un certo numero di tensori, permette di trovare l'espressione dei rimanenti. Il procedimento è iterativo e, ad ogni step, il set di elementi fissi viene ridefinito fino a completa ottimizzazione. Ulteriori metodologie includono gli algoritmi *moment-based*, la *simultaneous diagonalization* e l'algoritmo di *Levenberg-Marquardt*.

2.3.3 Decomposizione di Tucker

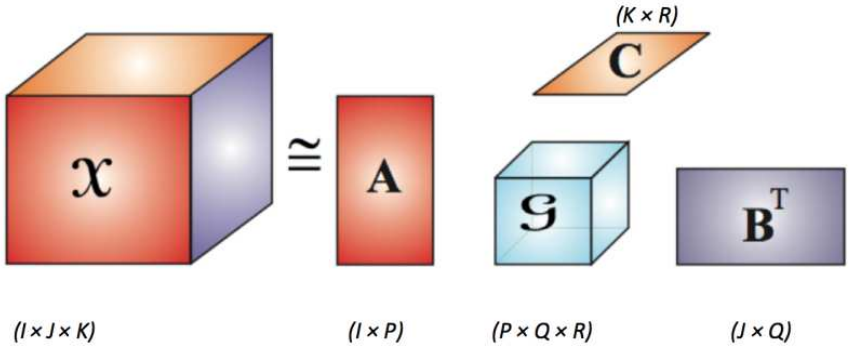


Figura 2.17: Decomposizione di Tucker

Un'altra possibilità nella decomposizione di tensori al fine della compressione è l'applicazione dell'algoritmo di Tucker. Eliminando infatti il vincolo sulla diagonalità determinato dalla decomposizione CP, permette di esprimere il tensore tramite un suo *core* ed una successione di matrici che, tramite l'applicazione di prodotti *n-mode*, ricostruiscono il tensore originale.

La decomposizione di Tucker, supponendo $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$, si può quindi scrivere come:

$$\boldsymbol{\mathcal{X}} \approx \boldsymbol{\mathcal{G}} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = \sum_{\mathbf{p=1}}^{\mathbf{P}} \sum_{\mathbf{q=1}}^{\mathbf{Q}} \sum_{\mathbf{r=1}}^{\mathbf{R}} \mathbf{g_{pqr}}\, \mathbf{a^p} \circ \mathbf{b^q} \circ \mathbf{c^r} \tag{2.3}$$

dove G rappresenta il tensore di *core* di dimensioni $P \times Q \times R$ e i fattori $\mathbf{A}, \mathbf{B}, \mathbf{C}$ sono matrici di dimensione, rispettivamente, $I \times P$, $J \times Q$, $K \times R$.

I vantaggi offerti da questa tipologia di decomposizione, come introdotto precedentemente, sono l'assenza di vincoli importanti sulla uniformità dei fattori o l'essere, per il *core tensor*, un elemento *superdiagonale* e la non unicità della rappresentazione ottenuta.

È chiaro il vantaggio introdotto dall'uso di queste tecniche di compressione nell'ambito del Deep Learnign, dove operatori convoluzionali, applicati tramite un processo computazionalmente intenso, possono essere scomposti in una molteplicità di elementi più semplici e adatti all'esecuzione su piattaforme embedded[19].

Capitolo 3

Implementazione di un sensore CNN-based per il pedestrian tracking

Il processo di prototipazione e implementazione di un modello di rete convoluzionale ottimizzato per dispositivi *embedded*, volti all'*edge computing*, ha visto lo studio di due ipotesi d'integrazione rappresentanti scenari caratterizzati da *hard constraints* e *soft constraints*.

Nel primo caso la piattaforma scelta, una scheda di sviluppo OpenMV Cam H7 Plus, ha evidenziato dei vincoli assoluti sulla realizzabilità del progetto dettati da limitazioni hardware e dalla parziale disponibilità di framework di supporto all'esecuzione delle reti scelte.

Per la seconda piattaforma, un Raspberry Pi 4, le capacità computazionali si sono dimostrate un potenziale ostacolo nel raggiungimento di prestazioni *real time* adeguate, ma hanno beneficiato dei processi di ottimizzazione e compressione del modello adoperate in fase d'integrazione.

Di seguito viene riportata una descrizione dei dispositivi e del processo d'integrazione delle soluzioni scelte, oltre a una breve introduzione alla libreria adoperata durante lo sviluppo.

3.1 Software

3.1.1 TensorFlow

Nato come progetto interno per l'uso nel servizio di posta elettronica *Gmail*, presso l'azienda Google, TensorFlow[1] è oggi tra i più noti framework per il Machine Learning e Deep Learning, grazie soprattutto alla sua diffusione a seguito della pubblicazione del suo codice sorgente sotto i termini delle license *Open Source*.

L'ecosistema che costituisce, in continua evoluzione e, al tempo della redazione di questo documento, giunto alla versione stabile 2.4.2, raccoglie in sé numerosi strumenti che spaziano dalla gestione dei dati in pipeline di elaborazione, alla costruzione di modelli di reti neurali con architetture varie, al *training* degli stessi con successiva validazione, fino al *deployment* su piattaforme tra loro varie come *web engine* e dispositivi mobili.

Complessivamente, il framework TensorFlow si presenta come una soluzione *end-to-end* in grado di fornire sistemi completi e integrati con i quali è possibile interagire, tramite le API fornite, a più livelli di astrazione, disponibile in alcuni tra i più comuni linguaggi di programmazione come Python, C++, Java, Go, JavaScript e altri.

3.1.2 Keras

L'approccio più semplice alla realizzazione di modelli è l'uso del modulo *Keras*[2] per la prototipazione rapida delle più comuni architetture. Oltre a includere implementazioni pronte di reti note per vari campi del Deep Learning, l'API Keras dispone di utility per la semplificazione di step tipici di un processo di *training*, come l'applicazione di tecniche di *data augmentation* per l'integrazione dei dataset che risultano non adeguati al task che ci si prepone, o il *logging* del progresso ottenuto durante l'ottimizzazione dei parametri.

[1] https://www.tensorflow.org/
[2] https://keras.io/

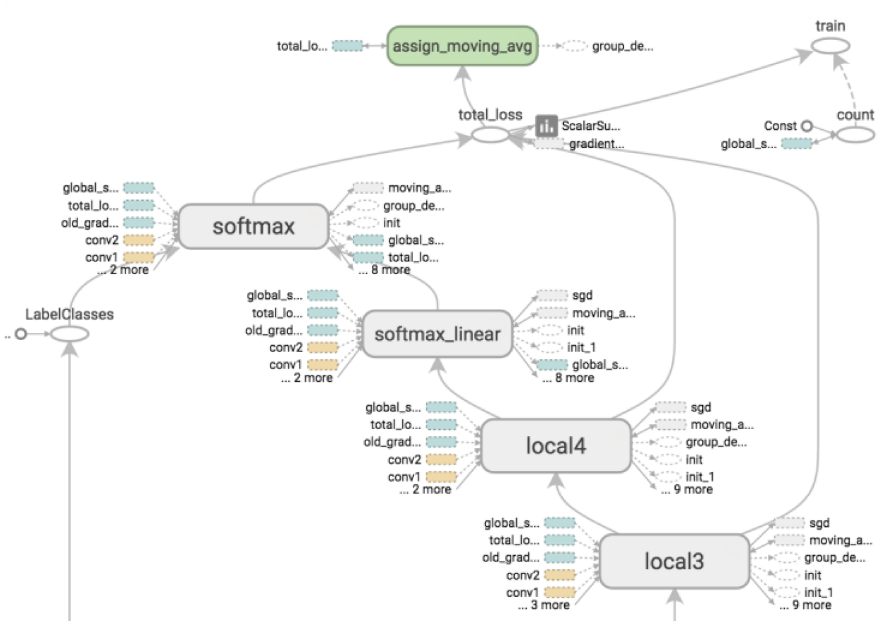


Figura 3.1: Esempio di grafo generato da un modello di rete neurale

Come prevedibile, l'elevata semplificazione presente comporta alcune limitazioni nella flessibilità che ci si può attendere durante l'implementazione di reti più articolate. È quindi possibile procedere con l'uso del set di API, definito *Functional*, che consente, al costo della produzione di codice più prolisso e dettagliato, l'interazione più diretta con la struttura descrittrice dei modelli per il Machine Learning, ovvero il *Grafo*. Questo è un formato convenzionale per l'espressione, priva di ambiguità, di algoritmi di varia natura e descrive al meglio il flusso che segue il dato durante la fase di processamento.

3.1.3 Piattaforma Google Colaboratory

Al fine di una più rapida analisi delle reti, per le fasi di training e valutazione si è eseguito parte della sviluppo sulla piattaforma Google Colaboratory. Il sistema consiste nell'accesso, gratuito per gli utenti registrati ai servizi Google ma soggetto ad alcune limitazioni temporali, ad istanze di Virtual Machine generate su richiesta ed in grado di eseguire Notebook Jupyter con codice Python, compreso il framework

TensorFlow per il Machine Learning.

Google Colaboratory, offrendo un sistema di interazione dipendente solamente dalla disponibilità di accesso ad un browser web, rappresenta oggi il principale riferimento per lo sviluppo di sistemi intelligenti, sia in ambito educativo con la ridotta barriera di ingresso, sia nella ricerca grazie all'accesso gratuito che esso fornisce a schede grafiche (GPU) e unità tensoriali (TPU) per l'accelerazione dei processi di allenamento dei modelli.

Lo svolgimento di parte dell'elaborato è stato condotto, quindi, all'interno di questa utile piattaforma, consentendo tempi di sviluppo più rapidi e indipendenza dalle disponibilità di macchine fisiche dalle prestazioni sufficientemente elevate.

3.2 Hardware

3.2.1 OpenMV Cam

L'OpenMV Cam[3] è una board di sviluppo nata con l'obiettivo di avvicinare il mondo della elaborazione embedded alla Computer Vision. Essa consiste in una piattaforma ARM M7 a 32 bit prodotta dalla ST Microelectronics, abbinata a una fotocamera e a moduli di espansione della memoria RAM e Flash.

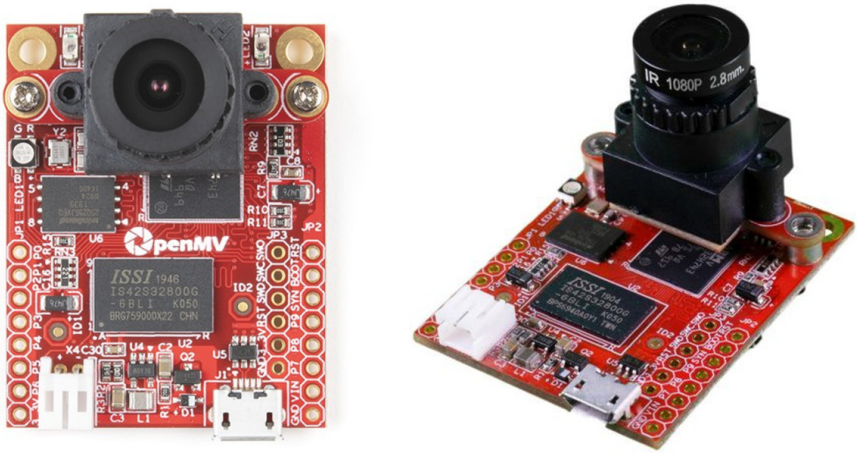


Figura 3.2: Scheda di sviluppo OpenMV Cam H7 Plus

[3]https://openmv.io/

3.2.1.1 Specifiche

Il modello scelto è, in questo caso, la variante OpenMV Cam H7 Plus con le caratteristiche di seguito riportate:

- Processore ARM Cortex M7 STM32F743II
  - frequenza di clock: 480 MHz, 1027 DMIPS
  - memoria flash integrata: 2 MB
  - memoria RAM integrata: 1 MB
- SDRAM esterna: 32 MB
- Flash esterna: 32 MB
- Slot microSD con 100 Mbps in lettura e scrittura
- Sensore ottico OV5640 da 5 Mpx con supporto per lenti M12
- 10 GPIO, 1 ADC, 1 DAC, 2 canali PWM
- Possibilità di espansione delle periferiche tramite interfaccia unificata per moduli ausiliari

3.2.1.2 Interprete MicroPython

Il progetto OpenMV include un IDE sviluppato appositamente, corredato di numerosi esempi, e un porting dell'interprete MicroPython che facilita l'approccio alla piattaforma e riduce i tempi di prototipazione. MicroPython è, a tutti gli effetti, un compilatore Python completo comprensivo di una runtime che viene eseguita su bare-metal. Fornisce accesso interattivo tramite prompt (REPL) per l'esecuzione immediata dei comandi tramite comunicazione USART, ma consente anche l'importazione di script da filesystem, nel caso della OpenMV quindi dalla memoria ausiliaria Flash/microSD.

Complessivamente il prompt fornisce un'esperienza molto prossima a quanto si può riscontrare su macchine più avanzate e fornite di sistema operativo, facilitando l'approccio alla progettazione su dispositivi embedded ma consentendo anche, nel

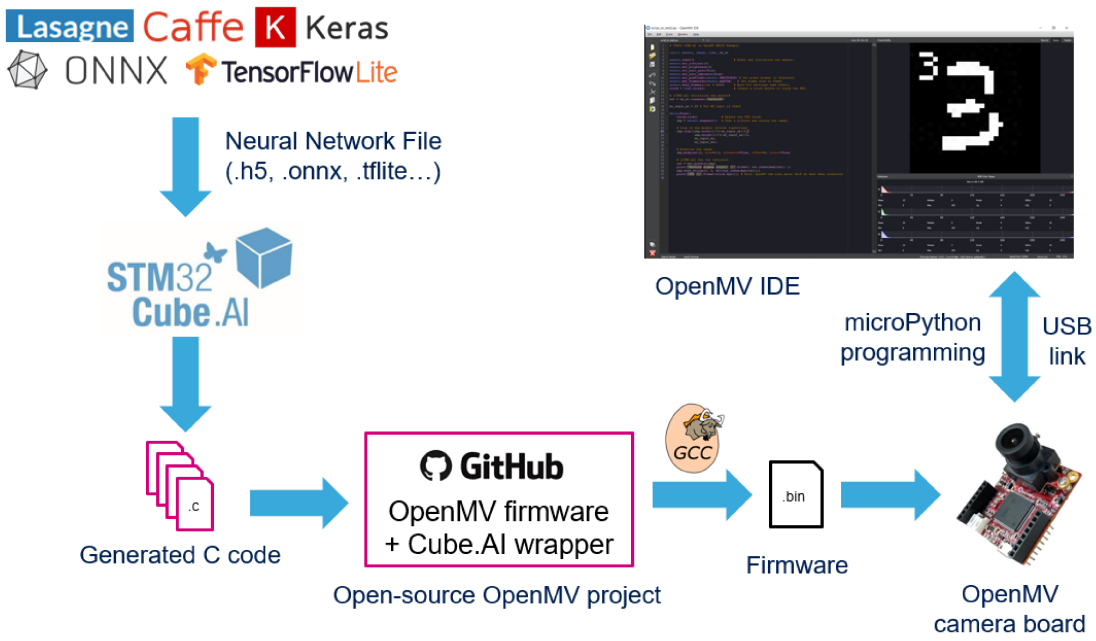


Figura 3.3: Processo di integrazione di un modello di rete neurale tramite OpenMV e STM32CubeAI

caso di programmi complessi, una maggior astrazione del codice essendo il linguaggio di programmazione un subset di Python.

L'interprete viene fornito come firmware precompilato modellato sulle capacità della variante di OpenMV Cam scelta, con limitazioni riguardanti la dimensione dell'*heap* e una frazione consistente della memoria complessiva dedicata all'archiviazione temporanea dei *frame buffer*.

Viene resa disponibile inoltre una libreria dedicata alla Computer Vision con algoritmi, altamente ottimizzati, per tecniche quali la *edge detection*, il *frame differencing*, il tracking di marker appositi (e.g. *AprilTags*), accompagnata da un modulo MicroPython che ricalca alcune delle principali funzionalità della nota libreria NumPy. Nonostante la complessità delle funzionalità incluse, sono evidenti i passi effettuati in fase di sviluppo della libreria per portare le performance di esecuzione a non risentire dei livelli di astrazione che l'interprete consente, fornendo tempi di risposta per operazioni come la detection di marker prossimi al *real time*.

Nel firmware è presente una variante dell'interprete di modelli TFLite Flat-

buffer per l'inferenza tramite framework TensorFlow: questo segue i progressi del progetto ufficiale *TensorFlow Lite for Microcontrollers* portato avanti da *Google* a meno di minimi adattamenti volti al semplificare l'interazione mediante interfaccia MicroPython.

3.2.2 Raspberry Pi

Il Raspberry Pi[4] è un ormai famoso progetto nato dalla omonima fondazione britannica come tentativo di portare il computing nel mondo dell'educazione abbattendo le barriere costituite dai costi di sviluppo, produzione e distribuzione che progetti simili avevano precedentemente sviluppato. Si presenta come una board di sviluppo altamente versatile, accompagnata da una fiorente community, e, nella sua quarta revisione principale, mette oggi in evidenza le prestazioni computazionali e bassi consumi che un'architettura ARM può offrire in numerosi scenari d'uso quotidiano.

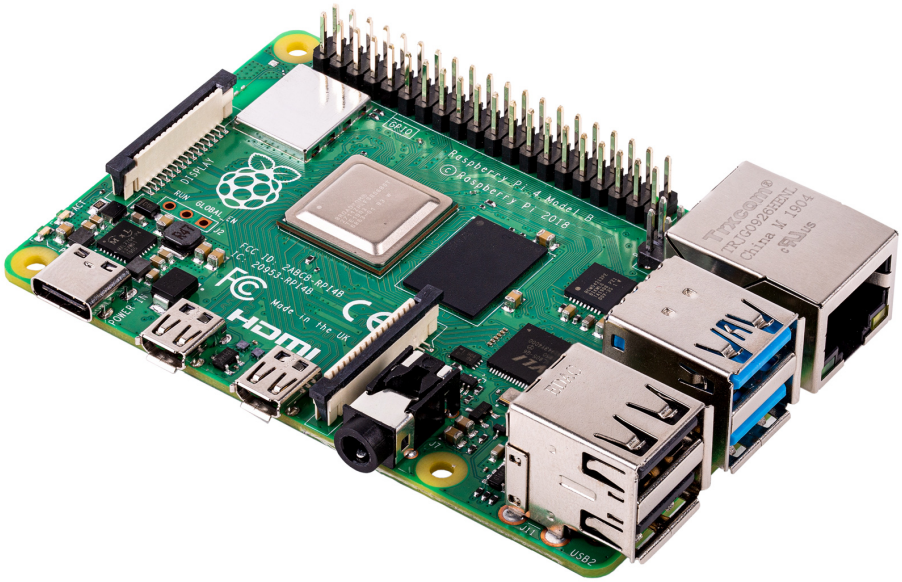


Figura 3.4: Raspberry Pi nella sua quarta revisione

[4]https://www.raspberrypi.org/

È il cuore, infatti, di numerosi progetti (headless computing, networking, robotica, etc.) che nel tempo hanno reso questa piattaforma il punto di riferimento per la prototipazione e lo sviluppo di soluzioni di moderata capacità computazionale, ma caratterizzate dal focus sul *low power*.

Come tale, si presta particolarmente allo scopo di sviluppare e integrare algoritmi di Object Detection nell'ottica della scalabilità e dell'elevata integrabilità, come ad esempio avviene nella ricerca tramite l'uso del framework, nonché sistema operativo, ROS (Robotic Operating System), del quale viene fornita una *distribuzione* per la piattaforma RPi.

3.2.2.1 Specifiche

Il Raspberry Pi 4 è una scheda di sviluppo costituita da un processore ARM Cortex A72 a 64bit, on-board RAM a partire da 2 GB, supporto HDMI multischermo, bus PCIe e USB3.0, capacità di networking integrate e accesso diretto a GPIO e periferiche per protocolli di comunicazione a basso livello (I2C, USART, SPI, CSI, etc.). La variante adoperata presenta le seguenti caratteristiche:

- Processore: Broadcom BCM2711 Quad-core
  - frequenza di clock: 1.5 GHz
  - cache: 80 kB L1, 1 MB L2
  - supporto RAM: LPDDR4-3200 SDRAM fino a 8 GB indicizzabili
- Memoria RAM: 4 GB
- Memoria Flash: lettore microSD e supporto avvio da EEPROM bootloader
- Networking: Gigabit Ethernet, 2.4/5.0 GHz 802.11ac, Bluetooth 5.0, BLE
- Multimedia:
  - output video: 2 micro HDMI, port DSI 2-lane
  - camera: porta CSI 2-lane
  - grafica: supporto a H265, H264, OpenGL ES 3.1, Vulkan 1.0

3.2.2.2 Interprete TFLite

Il sistema operativo adoperato, Raspbian OS, una *distribuzione* derivata da Debian, fornisce accesso a *repository* software contenenti il framework TensorFlow e l'interprete per TFLite Flatbuffer sia come modulo integrato, sia come eseguibile a parte per una ridotta occupazione in memoria e un minor tempo di esecuzione. In questo caso la più ampia disponibilità di risorse consente di ottenere un supporto completo al set di operazioni previste dalla versione Lite del framework, estendendo la lista di topologie architetturali realizzabili. Quando non direttamente supportata, una *Op* (operazione) può essere ereditata dall'implementazione più complessa di TensorFlow, a costo di una minor ottimizzazione e una conseguente dilatazione dei tempi di processing.

La procedura di conversione di un modello di rete neurale da formato *h5*, caratteristico di Keras, o *pb* (*protobuffer*, a formato TFLite *flatbuffer*, consiste in un'ottimizzazione iterativa del grafo delle operazioni: lo strumento di conversione, incluso nel modulo `tflite`, traversa le varie *signature* presenti nel modello, rappresentanti i possibili percorsi che il dato in input può seguire a seconda della sua natura e di processi decisionali interni, per ricostruirne una versione, funzionalmente equivalente, costituita da singole operazioni compatibili con la runtime *Lite*. Dove possibile, gruppi di *Op* costituenti pattern noti e ottimizzabili vengono sostituiti da composizioni in grado di accelerare l'esecuzione o ridurne lo spazio occupato in memoria.

Se predisposto, inoltre, viene applicato un sistema di compressione per quantizzazione dei parametri dei layer presenti: da numeri in virgola mobile se ne calcolano i corrispondenti espressi, a meno di un errore di risoluzione, come interi a *8 bit*. Questa operazione si è dimostrata essere estremamente efficacie nella riduzione delle risorse necessarie durante l'inferenza al costo di, tipicamente, una minima perdita di accuratezza. Per mitigare questo questa perdita di precisione è possibile fornire un dataset di riferimento con cui guidare il processo di quantizzazione, ottenendo una compressione applicata, quindi, *su misura*.

È da notare come non tutte le operazioni man mano integrate nella runtime

posseggano la possibilità di una totale quantizzazione, e come a volte sia necessario ricorrere a step per la riduzione della profondità in bit dei parametri effettuati online, in una formulazione della compressione definita *quantizzazione dinamica*.

3.2.3 Acceleratore Google Coral

L'acceleratore di reti neurali Google Coral[5] è costituito da una piattaforma hardware specializzata, corredata da una suite software per la corretta compilazione dei modelli sviluppati.

Figura 3.5: Acceleratore USB Google Coral

È disponibile in diversi formati per la rapida integrazione in scenari tipici d'uso: dalle *development board* o *single board computer* per soluzioni *all-in-one* a periferiche compatte, in grado di comunicare tramite bus *PCIe* e *USB*, come dispositivi ausiliari per sistemi preesistenti, fino al singolo *IC* per la progettazione di schede custom.

Il cuore del sistema è un ASIC, di ridotte dimensioni ($5mm \times 5mm$), in grado di eseguire fino a 4 TOPS con un consumo di potenza estremamente ridotto e pari a circa 0.5W per TOPS per un totale di 2W. Il nome commerciale del singolo chip è *Tensor Processing Unit* (TPU) e richiama appunto l'abilità nell'elaborazione di

[5]https://coral.ai

operazioni su tensori. L'incremento di prestazioni che si ottiene dall'uso di questo dispositivo varia a seconda dell'architettura integrata e della pipeline di processing che si va a delineare, ma è generalmente significativo. Si riporta un confronto tra i tempi d'inferenza ottenuti su processore ARM Cortex-A53 (quad-core) e gli stessi relativi all'uso di una Coral Dev Board.

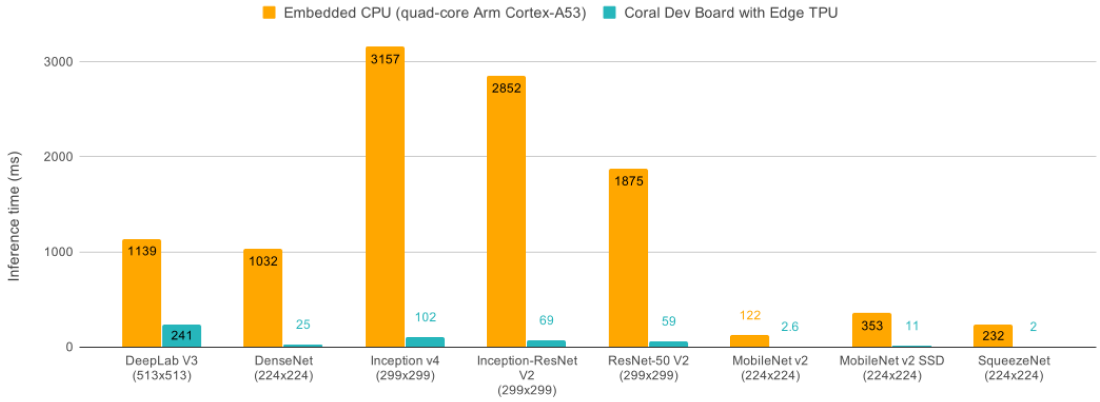


Figura 3.6: Tempi d'inferenza per architetture eseguite su ARM con o senza dispositivo Coral

La flessibilità del sistema consente l'uso delle risorse fornite dal chip sia per l'esecuzione di modelli semplici, sia per la disposizione di pipeline accelerate per parziale parallelismo, sia per l'inferenza di più modelli in contemporanea con condivisione della memoria di *scratchpad*.

Benché non sia prevista una soluzione completa per il processo di training, che avviene tipicamente offline su sistemi general purpose, è possibile anche in questo caso ottenere un incremento delle prestazioni della macchina nel caso specifico di *transfer learning* con l'esecuzione della *backbone* direttamente su dispositivo Coral e con la produzione dei parametri di adattamento solo per l'ultimo layer, per il quale deve essere disposta la possibilità di training.

Per il deployment dei modelli esistono, inoltre, diverse reti pre-allenate che coprono gli *use case* più comuni e che possono facilmente essere adattate tramite step di *fine tuning*.

Data la natura dell'architettura del dispositivo ASIC, i modelli implementabili hanno però due principali limitazioni:

- Il flatbuffer può contenere solamente parametri con profondità, in bit, pari a

    8. Non sono quindi previsti pesi di tipo `float` o, in generale, parametri non quantizzati.

- Il grafo della rete deve rispettare la compatibilità delle operazioni, in quanto non tutte le *Op* disponibili nella versione principale del framework TensorFlow sono oggi integrate e adattate all'esecuzione su TPU.

Il modello scelto per questo elaborato è la variante con comunicazione tramite protocollo USB: pur essendo la rapidità di trasferimento dati un potenziale collo di bottiglia nell'esecuzione del processo d'inferenza, la versatilità dello standard USB ha consentito un più rapido inserimento dell'acceleratore nella catena di sviluppo dell'Object Detector.

3.3 Dataset

L'operazione di detection è stata condotta su dati costituiti da immagini in risoluzione variabile preventivamente sottoposte a una fase di *preprocessing*, e i risultati valutati sui corrispettivi *ground thruth* parallelamente forniti e adattati a seconda della convenzione adoperata dall'architettura scelta.

I dataset disponibili per il task di Object Detection differiscono dal numeroso set di quanti dedicati alla semplice Classification per la disponibilità, nei file delle annotazioni relative, oltre che delle classi di appartenenza degli oggetti presenti per ogni elemento, anche della posizione che questi occupano all'interno del frame espressa come coordinate della *bounding box* nella quale sono racchiusi.

Oggi sono presenti molte raccolte con queste caratteristiche, delle quali le più note, il Pascal Visual Object Classes[20] e il Common Objects in Context[21], contengono la classe associata alla figura umana, utile per la *Pedestrian Detection*[2][3][4][5][6],

Figura 3.7: Elementi individuati dalle rispettive bounding box

obiettivo di questo elaborato. Di seguito si riportano alcune caratteristiche dei dataset adoperati e delle metriche scelte per la valutazione delle performance e la compilazione dei risultati.

3.3.1 Pascal Visual Object Classes (VOC)

Il dataset Visual Object Classes[20] nasce come riferimento per la classificazione e l'individuazione di oggetti ed è stato proposto in varie *challenge* annuali, raccolte, con l'obiettivo di una più elevata copertura del set possibile nel rilevamento delle classi presenti al suo interno e della loro posizione. È poi divenuto termine standard di paragone nella valutazione delle performance dei modelli classificatori e dei detector e viene oggi distribuito in due collezioni principali prodotte nel 2007 e nel 2012.

Gli oggetti individuati vengono rappresentati all'interno del contesto in cui è possibile trovarli, senza quindi una prima fase di segmentazione e isolamento: questo rende il compito della categorizzazione più arduo, in principio, ma la presenza d'informazioni ausiliarie nell'immagine fa sì che il modello opportunamente allenato sia più robusto ed effettui distinzioni a volte sottili, o mostri, generalmente, maggior

resilienza in condizioni di parziale ostruzione della classe presente nel frame.

La numerosità delle classi è ridotta, con sole 20 categorie presenti:

- Uomo: persona;
- Animale: uccello, gatto, mucca, cane, cavallo, pecora;
- Veicolo: aereo, bicicletta, barca, autobus, auto, moto, treno;
- Oggetto: bottiglia, sedia, tavolo da pranzo, pianta in vaso, divano, tv/monitor.

Questa condizione mostra la propensione del dataset a task di *detection* piuttosto che di classificazione, in quanto quest'ultima viene principalmente eseguita oggi solo in fase di transfer-learning e fine-tuning da reti con parametri precedentemente calcolati sulla base di dataset dediti alla sola *classification*, come l'ImageNet. Il numero d'immagini incluse è stato però incrementato nel corso degli anni fino ai circa 17000 elementi nella raccolta del 2012 e comprende situazioni dove più classi possono essere individuate nella stessa inquadratura.

Le annotazioni sono presenti in formato *XML* e seguono la struttura dell'esempio qui riportato:

```
<annotation>
    <filename>2012_004331.jpg</filename>
    <folder>VOC2012</folder>
    <object>
        <name>person</name>
        <actions>
            <jumping>1</jumping>
            <other>0</other>
            <phoning>0</phoning>
            <playinginstrument>0</playinginstrument>
            <reading>0</reading>
            <ridingbike>0</ridingbike>
            <ridinghorse>0</ridinghorse>
            <running>0</running>
            <takingphoto>0</takingphoto>
            <usingcomputer>0</usingcomputer>
            <walking>0</walking>
        </actions>
```

```
        <bndbox>
            <xmax>208</xmax>
            <xmin>102</xmin>
            <ymax>230</ymax>
            <ymin>25</ymin>
        </bndbox>
        <difficult>0</difficult>
        <pose>Unspecified</pose>
        <point>
            <x>155</x>
            <y>119</y>
        </point>
    </object>
    <segmented>0</segmented>
    <size>
        <depth>3</depth>
        <height>375</height>
        <width>500</width>
    </size>
    <source>
        <annotation>PASCAL VOC2012</annotation>
        <database>The VOC2012 Database</database>
        <image>flickr</image>
    </source>
</annotation>
```

Il file di annotazioni contiene, oltre al nome dell'immagine alla quale è associato, 4 sezioni principali:

1) **sbject**: descrive la classe o le classi presenti nel frame, le azioni in corso di svolgimento se riconosciute, le *bounding box* in formato $[x_{max}, x_{min}], [y_{max}, y_{min}]$ e include il *flag* `difficult` per agevolare il filtraggio del set;
2) **size**: riporta le dimensioni, in pixel, dell'immagine associata, compreso il numero di canali colore;
3) **source**: include metadati riguardanti l'origine del file immagine e a quale revisione del dataset appartenga;
4) **segmented**: l'eventuale segmentazione del soggetto mostrato.

Come anticipato, non tutte le classi presenti nella raccolta sono utili al fine del task di Pedestrian Detection ma sono state comunque adoperate per la valutazione e comparazione dei modelli.

3.3.2 Common Objects in Context (COCO)

È un dataset più recente rispetto al Visual Objects Classes[20], contenente una varietà di elementi categorizzati in 90 classi complessive e quindi considerevolmente più ampio. Anche questo dataset nasce dall'istituzione di challenge nel corso degli anni, che lo hanno portato ad includere 245.496 elementi per la raccolta pubblicata nel 2014 (con una integrazione del 2015) e 163.957 per la revisione del 2017. Le classi presenti sono:

- Uomo: *person*
- Accessorio: *hat*, *backpack*, *umbrella*, *shoe*, *eye*, *glasses*, *handbag*, *tie*, *suitcase*
- Animale: *bird*, *cat*, *dog*, *horse*, *sheep*, *cow*, *elephant*, *bear*, *zebra*, *giraffe*
- Veicolo: *bicycle*, *car*, *motorcycle*, *airplane*, *bus*, *train*, *truck*, *boat*
- Elemento stradale: *traffic light*, *fire hydrant*, *street sign*, *stop sign*, *parking meter*, *bench*
- Cibo: *banana*, *apple*, *sandwich*, *orange*, *broccoli*, *carrot*, *hot dog*, *pizza*, *donut*, *cake*
- Oggetto: *bottle*, *plate*, *wine*, *glass*, *cup*, *fork*, *knife*, *spoon*, *bowl*, *frisbee*, *skis*, *snowboard*, *sports*, *ball*, *kite*, *baseball bat*, *baseball glove*, *skateboard*, *surfboard*, *tennis racket*, *chair*, *couch*, *potted plant*, *bed*, *mirror*, *dining table*, *window*, *desk*, *toilet*, *door*, *tv*, *laptop*, *mouse*, *remote*, *keyboard*, *cell phone*, *microwave*, *oven*, *toaster*, *sink*, *refrigerator*, *blender*, *book*, *clock*, *vase*, *scissors*, *teddy bear*, *hair drier*, *toothbrush*, *hair brush*

Come per il Visual Object Classes, il dataset serve non solo come riferimento per il task di *detection*, ma contiene annotazioni utili anche per i processi di *segmentation*,

per l'individuazione di *keypoint* nel corpo umano per la *pose estimation*, per il riconoscimento del contesto e, naturalmente, la *classificazione*.

Nel tentativo di una miglior standardizzazione e fruibilità dei dati, il formato con cui vengono fornite le annotazioni è stato rivisitato rispetto ai precedenti dataset di detection ed è qui proposto come file JSON, un formato particolarmente adatto allo scambio di informazioni tra macchine ma facilmente comprensibile dall'uomo. Riportiamo qui un esempio di etichetta per una generica immagine appartenente al Common Objects in Context:


```
{
    "info": {
        "year": "2014",
        "version": "1",
        "description": "COCO 2014 Dataset",
        "contributor": "COCO Consortium",
        "url": "http://img.cocodataset.org/train2014/2014_000109477.jpg",
        "date_created": "2000-01-01T00:00:00+00:00"
    },
    "licenses": [
        {
            "id": 1,
            "url": "https://creativecommons.org/publicdomain/zero/1.0/",
            "name": "Public Domain"
        }
    ],
    "categories": [
        {
            "id": 0,
            "name": "Workers",
            "supercategory": "none"
        },
        {
            "id": 1,
            "name": "head",
            "supercategory": "Workers"
        },
        {
            "id": 2,
            "name": "helmet",
```

```
            "supercategory": "Workers"
        },
        {
            "id": 3,
            "name": "person",
            "supercategory": "Workers"
        }
    ],
    "images": [
        {
            "id": 0,
            "license": 1,
            "file_name": "0001.jpg",
            "height": 275,
            "width": 490,
            "date_captured": "2014-07-20T19:39:26+00:00"
        }
    ],
    "annotations": [
        {
            "id": 0,
            "image_id": 0,
            "category_id": 2,
            "bbox": [
                45,
                2,
                85,
                85
            ],
            "area": 7225,
            "segmentation": [],
            "iscrowd": 0
        },
        {
            "id": 1,
            "image_id": 0,
            "category_id": 2,
            "bbox": [
                324,
                29,
                72,
                81
            ],
```

```
            "area": 5832,
            "segmentation": [],
            "iscrowd": 0
        }
    ]
}
```


Oltre ad introdurre un metodo differente per la rappresentazione di *bounding box*, che qui vengono riportate in forma $[x, y, width, height]$, il dataset COCO propone una nuova metrica, sempre basata sulla *mean average precision*, come nel VOC, ma calcolata a più livelli di IoU scelta e poi combinata in un solo punteggio.

3.4 Architetture

3.4.1 MobileNet-SSD

L'object detector Mobilenet-SSD viene proposto come evoluzione del Single Shot Multibox Detector presentato nell'articolo di riferimento, il quale basava l'architettura del primo stadio di elaborazione dell'immagine sul modello VGG-16. La scelta è, come precedentemente accennato, dettata dalla volontà di rendere questa architettura più efficiente nell'elaborazione delle immagini in termini di velocità di esecuzione e, quindi, numero di parametri coinvolti nella realizzazione della rete.

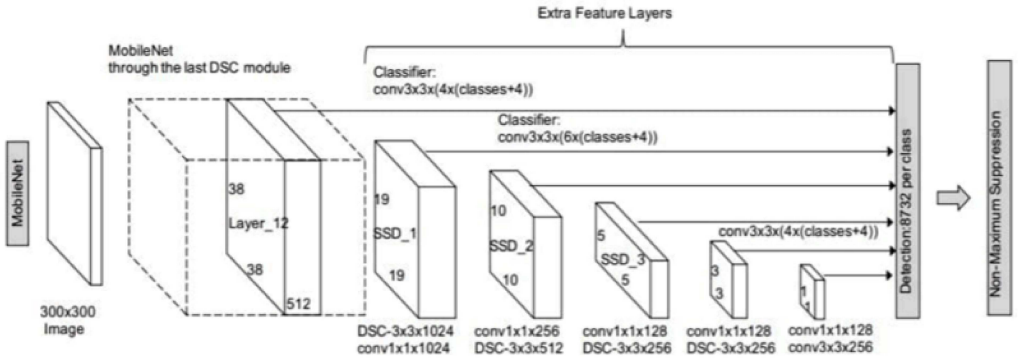


Figura 3.8: Object detector Mobilenet-SSD

La Mobilenet, dunque, applica il concetto di *flattened network* introdotto in [22] sostituendo opportunamente ai layer di convoluzione classica, con filtri 3×3, la combinazione di convoluzioni *depthwise separable* e convoluzioni *pointwise* in un processo di *fattorizzazione*: così facendo si ottiene una drastica riduzione del numero di parametri, da 8 a 9 volte rispetto l'originale, e perciò una minor complessità dell'operazione.

Vengono inoltre forniti, al fine di un miglior *tuning* della rete in funzione del *tradeoff* tra latenza e accuratezza che si vuole perseguire, due parametri di configurazione del modello, *depth multiplier* e *resolution multiplier*, che agiscono rispettivamente, come fattori moltiplicativi nel range [0..1], sulla numerosità dei canali di output a ogni layer della rete e sulla risoluzione dell'immagine in ingresso. È possibile in questa maniera, rinunciando a una miglior precisione nella classificazione, ridurre considerevolmente le dimensioni della rete.

Nella sua prima revisione, la *MobileNet v1* [8] è costituita da una successione di blocchi convoluzionali ad attivazione non lineare $ReLU$, per ognuno dei quali viene attuato un processo di *batch normalization*. La dimensione dell'immagine in ingresso è $224 \times 224 \times 3$. In Tabella 3.1 se ne riporta la struttura.

Tabella 3.1: Architettura MobileNet

Type	Stride	Kernel Shape	Input Size
Conv2D	2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
DepthWise Conv2D	1	$3 \times 3 \times 32$	$112 \times 112 \times 32$
Conv2D	1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
DepthWise Conv2D	2	$3 \times 3 \times 64$	$112 \times 112 \times 64$
Conv2D	1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
DepthWise Conv2D	1	$3 \times 3 \times 128$	$56 \times 56 \times 128$
Conv2D	1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
DepthWise Conv2D	2	$3 \times 3 \times 128$	$56 \times 56 \times 128$
Conv2D	1	$1 \times 1 \times 128 \times 256$	$56 \times 56 \times 128$

Type	Stride	Kernel Shape	Input Size
DepthWise Conv2D	1	$3 \times 3 \times 256$	$28 \times 28 \times 256$
Conv2D	1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
DepthWise Conv2D	2	$3 \times 3 \times 256$	$28 \times 28 \times 256$
Conv2D	1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
DepthWise Conv2D	1	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
DepthWise Conv2D	1	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
DepthWise Conv2D	1	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
DepthWise Conv2D	1	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
DepthWise Conv2D	1	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
DepthWise Conv2D	2	$3 \times 3 \times 512$	$14 \times 14 \times 512$
Conv2D	1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
DepthWise Conv2D	2	$3 \times 3 \times 1024$	$7 \times 7 \times 1024$
Conv2D	1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool	1	Pool 7×7	$7 \times 7 \times 1024$
Fully Connected	1	1024×1000	$1 \times 1 \times 1024$
Softmax	1	Classifier	$1 \times 1 \times 1000$

La seconda revisione, *MobileNet v2*[9], aggiunge dei *bottleneck lineari* e l'uso di *blocchi residuali inversi* per migliorare l'accuratezza del modello, a scapito di un lieve aumento di complessità e costo computazionale.

In questo caso il ruolo delle convoluzioni *pointwise*, prima adoperate solo per la combinazione dei canali lasciandone inalterata la numerosità o semplicemente duplicandola, diviene ora fondamentale per la riduzione dei parametri, andando a

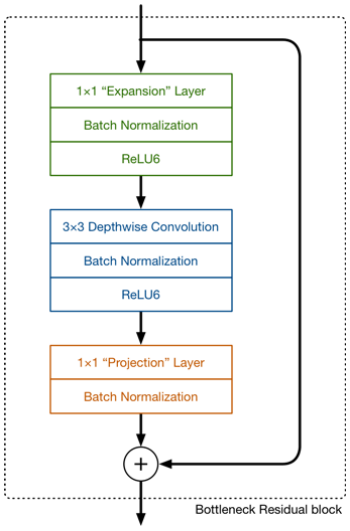


Figura 3.9: Blocco di bottleneck con struttura residuale.

trasferire l'informazione proveniente da un numero considerevole di canali in ingresso verso un tensore con dimensioni proporzionalmente più basso in un'operazione detta di *proiezione*.

L'effetto è l'introduzione di un collo di bottiglia che riduce il quantitativo di dati che attraversa la rete, ma che viene parzialmente bilanciato dall'introduzione di una ulteriore convoluzione 1×1 in ingresso al blocco che va invece a espandere il numero di canali presenti prima dell'applicazione della convoluzione *depthwise*. A questa architettura interna si affianca una linea diretta di trasporto dell'input, che procede parallelamente alle convoluzioni e si ricombina al loro termine, secondo la strategia del blocco residuale, per mitigare i problemi legati all'evanescenza del gradiente.

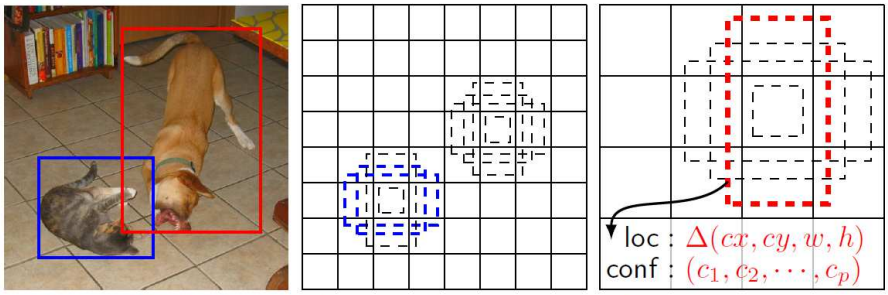
Tabella 3.2: Architettura della seconda revisione di MobileNet

Type	Stride	Expansion	Output	n	Input Size
Conv2D	2	-	32	1	$224 \times 224 \times 3$
Bottleneck	1	1	16	1	$112 \times 112 \times 32$
Bottleneck	2	6	24	2	$112 \times 112 \times 16$
Bottleneck	2	6	32	3	$56 \times 56 \times 24$
Bottleneck	2	6	64	4	$28 \times 28 \times 32$

Type	Stride	Expansion	Output	n	Input Size
Bottleneck	1	6	96	3	$14 \times 14 \times 64$
Bottleneck	2	6	160	3	$14 \times 14 \times 96$
Bottleneck	1	6	320	1	72×160
Conv2D	1	-	$1 \times 1 \times 1280$	1	72×320
Avg Pool	-	-	Pool $7\times$ 7	1	72×1280
Conv2D		-	$1 \times 1\times$k	-	$1 \times 1 \times 1280$

3.4.1.1 Multibox Detector

Il meccanismo di detection si basa sull'applicazione di ulteriori layer convoluzionali in grado di produrre una serie di box fisse alle quali è associato un punteggio indicante la confidenza con la quale si ritiene che al loro interno sia presente una classe nota. La struttura estende la backbone con *feature layer* basati su convoluzioni a dimensione via via via più piccola per consentire la predizione su diversi livelli di scala e ottimizzarne l'accuratezza.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figura 3.10: Single Shot Multibox Detector: architettura dello stadio di detection

Ognuno di questi è quindi seguito dall'applicazione di un kernel $3 \times 3 \times p$, con p il numero di canali presente nel *feature layer*, che andrà a generare uno *score* per la classe oppure l'offset rilevato rispetto alla *anchor box* fornita. La posizione delle *box* considerate di *default* è determinata manualmente in fase di costruzione della rete e

può seguire tipicamente alcune caratteristiche degli oggetti da individuare, come ad esempio gli *aspect-ratio* più probabili.

Nello specifico, per k box si produce un output contenente il vettore, in codifica *one-hot*, delle c classi rilevate, seguito da 4 elementi indicanti gli offset rispetto alla *bounding box* predefinita, il tutto applicato a ogni posizione della *feature map*, per un totale di $(c+4)kmn$ elementi di output per ogni mappa $m \times n$.

Questa strategia di determinazione delle *anchor box* fa si che, durante la fase di allenamento, diventi necessario assegnare output esatti al *groundthruth* scegliendoli tra i predefiniti disponibili. Una successiva predizione attuata dalla rete viene valutata quindi in base all'*overlap di Jaccard* ottenuto, con soglia minima posta pari a 0.5, mentre l'operazione di filtraggio dei candidati multipli viene eseguita in ultimo, secondo un processo di *Non Maximum Suppression* seguito da *sorting*.

3.4.2 Tiny YOLOv3

La rete *Tiny YOLOv3* nasce, ispirata dalla nota architettura *YOLOv3*, come un'alternativa competitiva in termini di rapidità d'inferenza, a scapito di una ridotta accuratezza, per scenari dove si necessita di tempi di risposta estremamente ridotti ma la disponibilità di risorse computazionali risulta limitata.

3.4.2.1 YOLOv3

You Only Look Once, nella sua terza edizione, propone un metodo di *object detection* che elimina, al pari di altri metodi *single shot*, la necessità d'integrare un sistema di *region proposal* che fornisca candidati alla classificazione.

L'idea originaria, infatti, vede l'immagine suddivisa in una griglia regolare di $n \times n$ riquadri, ognuno dei quali responsabile solamente del riconoscimento dell'oggetto il cui centro cade al suo interno e da cui viene estratta la predizione per B *bounding box* predefinite. A queste ultime corrisponde un punteggio, detto *confidenza*, basato sul calcolo della *Intersection over Union* tra il perimetro generato dalla predizione e

il *ground-thruth*, e la *probabilità* con la quale il classificatore ritiene vi sia contenuto un oggetto noto.

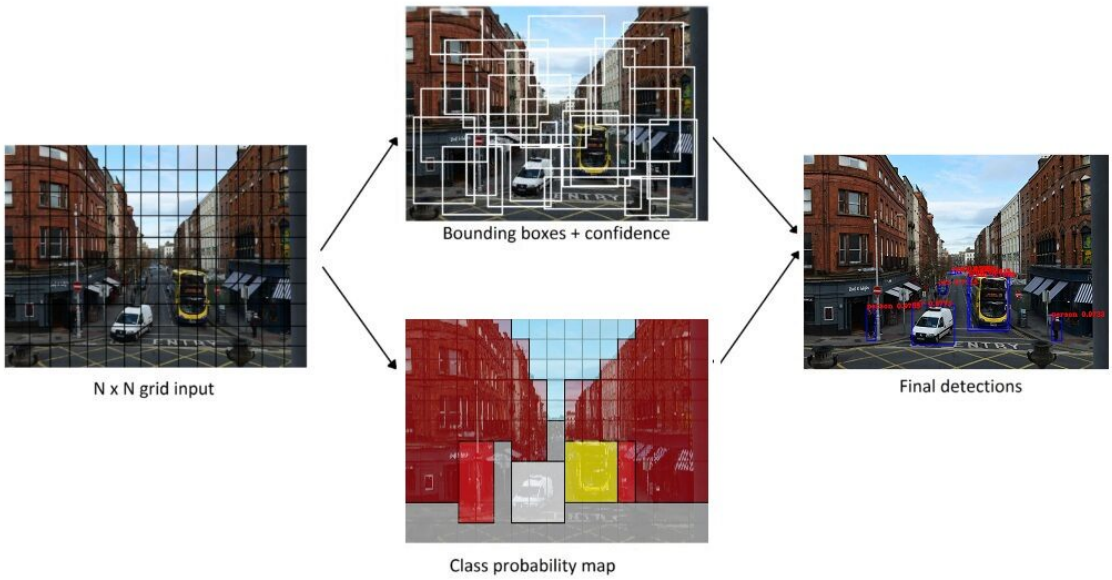


Figura 3.11: Divisione in NxN dell'immagine per la detection nelle reti YOLO

Questa tecnica da un lato consente l'esecuzione parallela di n^2 detection, dall'altro introduce il problema della *close object detection*, ovvero l'impossibilità di riconoscere più di una classe per elemento della griglia e una ridotta accuratezza nel posizionamento delle *box* candidate.

Con le successive revisioni, YOLOv2 e YOLOv3, si va a mitigare questa problematica introducendo alcune importanti variazioni:

- per ogni elemento della griglia vengono rese disponibili un numero k di *prior box* (equivalenti alle *anchor box*) la cui forma e disposizione viene calcolata sulla base del dataset d'interesse tramite *k-mean clustering*. La griglia è quindi in grado di prevedere un numero $n^2 \times k$ di oggetti dove k è il numero di *prior box* disponibili per riquadro;
- il metodo di predizione per condizioni di *multi-label*, ovvero oggetti con più di un'etichetta assegnata, introduce l'uso di classificatori *logistic regressor* separati

in sostituzione della funzione di attivazione *softmax*;

- la predizione, prima effettuata in ultimo stadio a seguito della rete di classificazione, viene ora eseguita a partire da *feature map* estratte in 3 punti differenti lungo la catena di convoluzioni, corrispondenti a livelli di scala diversi per l'ottimizzazione della detection di oggetti di dimensione varia;
- si adotta l'uso di blocchi residuali tramite l'inserimento di *skip-connection*, ottimizzando i risultati riscontrati nella versione precedente nei task di classificazione e ottenendo un punteggio *Top-5* di 93.8% sul dataset ImageNet. Questo consente, inoltre, una miglior accuratezza nella detection di oggetti di piccola dimensione relativa.

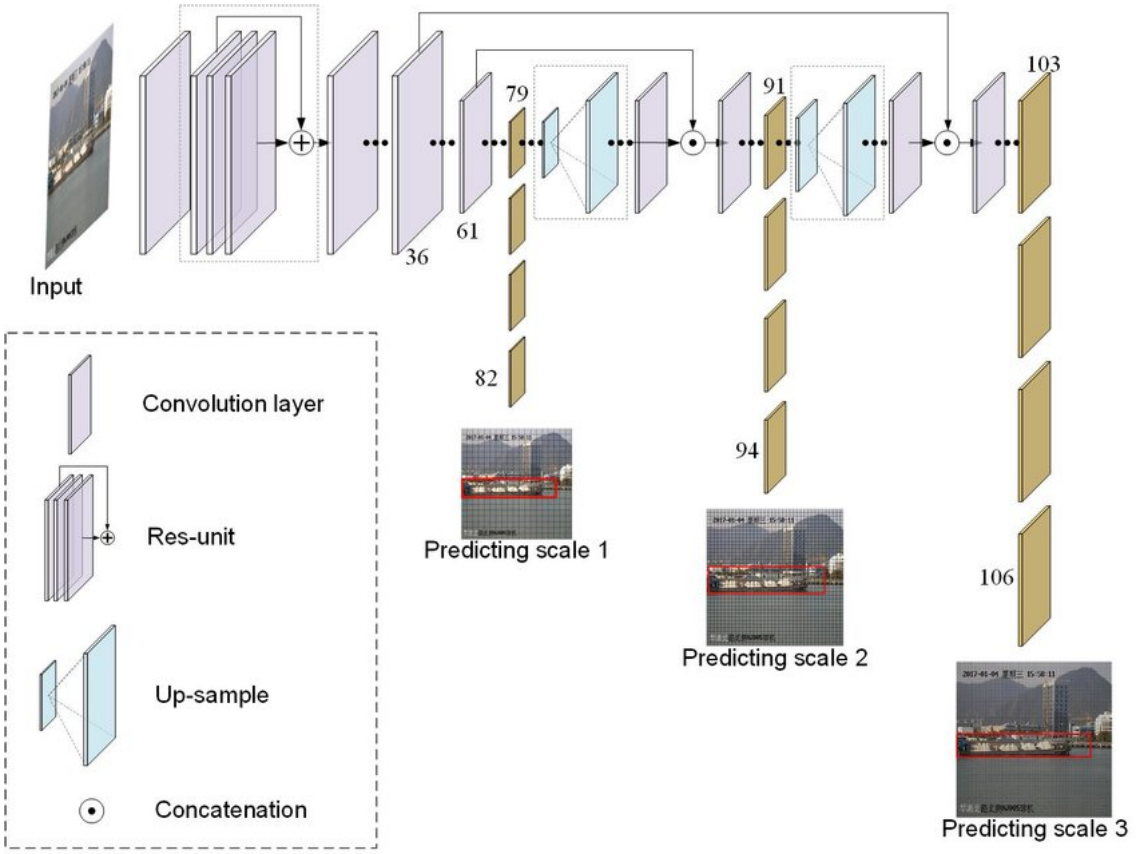


Figura 3.12: Architettura YOLOv3, backbone darknet-53

Complessivamente, l'architettura YOLOv3, riportata in figura [3.12], compete in

accuratezza con l'equivalente proposto in [16] fornendo una riduzione significativa dei tempi d'inferenza, a scapito di un lieve abbassamento della precisione nell'esatto posizionamento delle *bounding box*.

3.4.2.2 Variante *Tiny*

Nella sua versione originale, il modello YOLOv3 adopera come feature detector la backbone *darknet-53*, nome derivante dal numero complessivo di layer presenti, sviluppata tramite framework Darknet[6][23].

Tabella 3.3: Backbone di feature extraction darknet-53.

Type	Filters	Size	Output
Convolutional	32	3 × 3	256 × 256
Convolutional	64	3 × 3/2	128 × 128
Convolutional	32	1 × 1	
Convolutional	64	3 × 3	
Residual			128 × 128
Convolutional	128	3 × 3/2	64 × 64
Convolutional	64	1 × 1	
Convolutional	128	3 × 3	
Residual			64 × 64
Convolutional	256	3 × 3/2	32 × 32
Convolutional	128	1 × 1	
Convolutional	256	3 × 3	
Residual			32 × 32
Convolutional	512	3 × 3/2	16 × 16
Convolutional	256	1 × 1	
Convolutional	512	3 × 3	
Residual			16 × 16

[6]https://pjreddie.com/darknet/

Type	Filters	Size	Output
Convolutional	1024	$3 \times 3/2$	8×8
Convolutional	512		
Convolutional	1024	1×1 3×3	
Residual			8×8
Avgpool		Global	
Softmax		1000	

Nonostante sia stato introdotto allo scopo di ottenere una miglior capacità di classificazione, l'uso di questo *feature extractor*, dato l'elevato numero di parametri contenuti e la necessità di preservare copie in memoria dei tensori nelle sezioni con *skip-connection*, risulta non adeguato laddove non si disponga di risorse hardware dedicate.

La Tiny YOLOv3 riprende quindi la struttura *fully convolutional* della backbone presente nella versione YOLOv2, denominata *darknet-19*[14] (Tabella 3.4), caratterizzata da una successione di convoluzioni con filtro 3×3, seguite da processi di *batch normalization* per accelerarne il training e layer di *max pooling* per lo scaling delle *feature map*.

Per la fase di detection la rete replica parzialmente quanto definito nella YOLOv3, preservando due dei tre blocchi dediti alla predizione delle *bounding box*. Tra questi viene inoltre mantenuto il meccanismo di rinforzo delle *multi-scale prediction* con l'interconnessione mediante *up-sampling* delle feature. Questa scelta, seppur riducendo la sensibilità della rete nelle predizioni di oggetti con dimensione ridotta, permette una considerevole diminuzione del numero di parametri e, unitamente alla linearità della rete di *backbone*, un numero di operazioni necessarie contenuto e adatto all'inferenza *real time*.

Tabella 3.4: Struttura Tiny YOLOv3

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	Prediction				
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	Prediction				

Esistono oggi, inoltre, altre varianti “Tiny” basate sulla quarta revisione dell’archi-

tettura YOLO: a seguito di valutazione delle prestazioni e delle risorse necessarie alla loro esecuzione si è scelto di adoperare il modello *Tiny YOLOv3* perché caratterizzato da una struttura di complessità e numero di parametri inferiore, che più si adatta allo scenario d'uso previsto in questo elaborato.

3.4.2.3 Variante Custom Tiny YOLOv3

Si propone di seguito una variazione dell'architettura vista nella rete Tiny YOLOv3 che si caratterizza per l'applicazione di tecniche di compressione tramite decomposizione per i *layer* convoluzionali dove l'elevato numero di canali presente comporta un altrettanto elevato numero di filtri applicati in fase di inferenza.

Benché sia infatti costituita da un grafo semplice e da operazioni facilmente implementabili su piattaforme embedded con risorse limitate, l'accuratezza nella detection che il modello Tiny YOLOv3 garantisce deriva, più che da processi astrattivi avanzati, dalla numerosità dei filtri introdotti in fase di pre-detection, ovvero nella produzione delle *feature map*.

Sviluppando una decomposizione tramite rappresentazione di Tucker è stato quindi possibile ridimensionare il quantitativo di parametri coinvolti nella descrizione del modello di un fattore pari a 2.18 al costo di una riduzione del punteggio ottenuto, secondo metrica VOC, del 4% sull'equivalente non compresso.

Tabella 3.5: Struttura Custom Tiny YOLOv3 dopo decomposizione.

Layer	Type	Filters	Size/Stride	Input	Output
1	Convolutional	16	3 × 3/1	320 × 320 × 3	320 × 320 × 16
2	MaxPooling	16	2 × 2/2	320 × 320 × 16	160 × 160 × 16
3	DecomposedConv	9	1 × 1/1	160 × 160 × 16	160 × 160 × 9
4		19	3 × 3/1	160 × 160 × 9	160 × 160 × 19
5		32	1 × 1/1	160 × 160 × 19	160 × 160 × 32
6	MaxPooling	32	2 × 2/2	160 × 160 × 32	80 × 80 × 32
7	DecomposedConv	19	1 × 1/1	80 × 80 × 32	80 × 80 × 19

Layer	Type	Filters	Size/Stride	Input	Output
8		37	3 × 3/1	80 × 80 × 19	80 × 80 × 37
9		64	1 × 1/1	80 × 80 × 37	80 × 80 × 64
10	MaxPooling	64	2 × 2/2	80 × 80 × 64	40 × 40 × 64
11	DecomposedConv	37	1 × 1/1	40 × 40 × 64	40 × 40 × 37
12		74	3 × 3/1	40 × 40 × 37	40 × 40 × 74
13		128	1 × 1/1	40 × 40 × 74	40 × 40 × 128
14	MaxPooling	128	2 × 2/2	40 × 40 × 128	20 × 20 × 128
15	DecomposedConv	74	1 × 1/1	20 × 20 × 128	20 × 20 × 74
16		149	3 × 3/1	20 × 20 × 74	20 × 20 × 149
17		256	1 × 1/1	20 × 20 × 149	20 × 20 × 256
18	MaxPooling	256	2 × 2/2	20 × 20 × 256	10 × 10 × 256
19	DecomposedConv	149	1 × 1/1	10 × 10 × 256	10 × 10 × 149
20		298	3 × 3/1	10 × 10 × 149	10 × 10 × 298
21		512	1 × 1/1	10 × 10 × 298	10 × 10 × 512
22	MaxPooling	512	2 × 2/2	10 × 10 × 512	10 × 10 × 512
23	DecomposedConv	298	1 × 1/1	10 × 10 × 512	10 × 10 × 298
24		596	3 × 3/1	10 × 10 × 298	10 × 10 × 596
25		1024	1 × 1/1	10 × 10 × 596	10 × 10 × 1024
26	Convolutional	256	3 × 3/1	10 × 10 × 1024	10 × 10 × 256
27	DecomposedConv	149	1 × 1/1	10 × 10 × 256	10 × 10 × 149
28		298	3 × 3/1	10 × 10 × 149	10 × 10 × 298
29		512	1 × 1/1	10 × 10 × 298	10 × 10 × 512
30	Prediction	75			10 × 10 × 75

Si è optato, inoltre, per la rimozione del ramo di detection derivante dalla concatenazione di convoluzioni con upscaling delle feature. La scelta, volta all'ulteriore riduzione del numero di parametri coinvolti nella catena di detection, è risultata compatibile con l'uso di una risoluzione in ingresso scelta pari a 320 × 320, inferiore

ai 416 $\times$ 416 pixel dell'originale. Quest'ultima ha infatti portato a un conseguente ridimensionamento dei tensori di output dei singoli layer, rendendo meno significativo l'uso di grid di detection a più elevata densità.

Capitolo 4

Risultati Sperimentali

Viene qui introdotta la metodologia di addestramento e successiva valutazione delle implementazioni selezionate per le rispettive piattaforme, seguita dai risultati ottenuti e alcune criticità riscontrate nel processo d'integrazione su hardware.

4.1 Training

Si è scelto di eseguire il processo di training in modalità *offline* per mancanza di soluzioni software adeguate o semplice vincolo dettato da ridotte capacità computazionali, a seconda della *board* di destinazione. Benché sia infatti presente l'interesse da parte degli sviluppatori della libreria TensorFlow per l'inclusione di strumenti per l'esecuzione di *training on the edge*, attualmente gli interpreti TFLite e TFLite Micro includono solamente le funzionalità necessarie al processo d'inferenza.

4.1.1 Ottimizzatore Adam

La fase di training si è basata sull'uso di funzioni di ottimizzazione derivate dal processo di *gradient descent*: come introdotto precedentemente, questa tecnica può aggiornare ogni parametro di un modello, valutare come l'errore sia influenzato dalla variazione di questi ultimi, scegliere una "direzione" che andrebbe a ridurre il tasso di errore e continuare a iterare fino a convergere verso il suo minimo.

Nello specifico si è scelto l'algoritmo di *Adaptive Moment Estimation*[24], ovvero *Adam*, che differisce dall'implementazione classica dello *Stochastic Gradient Descent* per due caratteristiche fondamentali che ne distinguono l'approccio al processo di ottimizzazione.

La progressione dell'algoritmo è infatti variabile, o *adattiva*, ed è funzione dell'elemento soggetto ad aggiornamento: diversi parametri ricevono quindi diversi *rate* di apprendimento, rendendo il processo dinamico. Inoltre introduce il concetto di *momento*, di *primo* e *secondo ordine*, del gradiente. Il momento n-esimo è definito, per una variabile casuale, come il valore atteso di tale variabile elevato alla potenza di n. Formalmente:

$$m_n = E[X^n]$$

e corrisponde, nel suo primo ordine, alla *media* della variabile, e, nel secondo ordine, alla sua varianza.

La stima di questi valori è condotta dall'ottimizzatore Adam adoperando *moving average* esponenziali calcolate su intervalli ridotti di parametri del gradiente generalmente corrispondenti alla dimensione del *mini-batch* sul quale si effettua il training:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.1}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \tag{4.2}$$

con m, v *moving average* calcolate sul gradiente g e β iperparametro caratteristico di Adam. Le variabili m e v sono quindi *estimatori* dei momenti di primo e secondo ordine, per i quali è possibile cioè determinare la relazione:

$$E[m_t] = E[g_t] \tag{4.3}$$

$$E[v_t] = E[g_t^2] \tag{4.4}$$

Per ottenere questa proprietà e per delinearne meglio la connessione che sussiste

tra queste variabili è possibile procedere allo sviluppo del singolo termine di *moving average* per contributi di ordine i-esimo. Otteniamo quindi, per indice $i = [0, 1, \ldots, t]$:

$$
\begin{aligned}
m_0 &= 0 \\
m_1 &= \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1 \\
m_2 &= \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2 \\
&\vdots \\
m_t &= (1-\beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i
\end{aligned} \tag{4.5}
$$

Riprendendo la relazione precedente sul momento di primo ordine si può scrivere:

$$E[m_t] = E[(1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i] = \tag{4.6}$$

$$= E[g_i](1-\beta_1^t) + \sigma \tag{4.7}$$

dove σ include l'errore di approssimazione. Lo stimatore individuato possiede quindi un *bias* e richiede un fattore di correzione che porta a esprimere le due variabili di *moving average* come:

$$\cap m_t = \frac{m_t}{1-\beta_1^t} \tag{4.8}$$

$$\cap v_t = \frac{v_t}{1-\beta_2^t} \tag{4.9}$$

Da questi estimatori l'ottimizzatore quindi procede, durante le iterazioni, all'aggiornamento dei parametri del modello con un andamento che tiene conto anche degli step immediatamente precedenti e che risulta quindi più difficilmente perturbabile dalle minime variazioni presenti in ogni processo di valutazione del gradiente.

Va ricordato che proprio la maggior resilienza alle perturbazioni momentanee si manifesta occasionalmente, in Adam, con l'incapacità di convergere a soluzione,

problema per il quale viene affiancato da altre strategie di ottimizzazione che vengono introdotte tipicamente nelle fasi successive, dove il gradiente della curva assume valori più piccoli e sono richieste correzioni più precise.

L'iperparametro più comune di *learning rate*, indicante indirettamente la rapidità con cui si tenta la discesa del gradiente, è stato impostato infine a 0.0001, valore che si è dimostrato essere una soluzione sufficientemente equilibrata tra velocità di convergenza e robustezza nei confronti di fenomeni di *overshooting*.

4.1.2 Funzione di *loss*

La peculiarità del task di Object Detection è nella necessità di una valutazione congiunta della confidenza con la quale il modello assegna o meno un oggetto riconosciuto alla classe considerata *ground-thruth* e della precisione con la quale viene delimitata la porzione del frame che che lo contiene, ovvero la precisione di collocamento delle coordinate della *bounding box*.

Per questa ragione il computo dell'errore in fase di training, che avviene con una funzione detta *funzione di loss*, va a combinare, in misura opportuna, i contributi di questi due termini, al fine di ottenere un unico indicatore sul quale basare l'ottimizzazione dei pesi e dei bias della rete. Possiamo quindi esprimere l'errore come:

$$L = \frac{L_{class} + \alpha L_{loc}}{N} \tag{4.10}$$

dove N è un fattore di normalizzazione ed indica il numero di *box* a esito positivo. Gli elementi L_{class} e L_{loc}, indicanti rispettivamente gli errori di classificazione e localizzazione appena introdotti e bilanciati dal parametro α, dipendono tipicamente dalla natura della rete neurale che si implementa e dalle scelte effettuate in fase di sua progettazione. Per questo motivo, nel caso delle due famiglie prese in esame in questo elaborato, essi differiscono pur fornendo, in entrambi i casi, una misura coerente delle discrepanze tra dato etichettato e predizione.

Nella YOLOv3 la valutazione dell'errore derivato dall'associazione dell'oggetto a una certa classe è data dalla *cross-entropia binaria*, o *log loss*, e comprende un termine aggiunto che tiene conto della confidenza della predizione.

$$BinaryCrossEntropy = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(y_1^{prob}) + (1-y_i)log(1-y_i)] \tag{4.11}$$

La *localization loss* viene invece misurata come *mean square error* (MSE) sulle coordinate della *bounding box*.

$$MeanSquareError = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{n} \tag{4.12}$$

Per il modello MobileNet-SSD, la funzione per la valutazione dell'errore mantiene una *log loss* per il task di classificazione, mentre prevede una variazione dell'*errore assoluto medio*, definito comunemente *L1*, chiamata *smooth L1 loss*:

$$g(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{if } |x| > 1 \end{cases} \tag{4.13}$$

4.2 Metriche

Con l'evoluzione dello stato dell'arte per la *object detection* si sono sviluppate diverse metriche per la valutazione delle performance ottenute, spesso dettate da quanto definito nelle *challenge* svolte negli anni e divenute poi riferimento per i modelli futuri. Sebbene spesso simili a causa del comune obiettivo, queste metriche si distinguono per l'uso di parametri adattati per lo specifico campo di applicazione.

In generale, è frequente l'uso delle definizioni introdotte nei dataset VOC[20] e COCO[21], con accorgimenti riguardo l'uso, o meno, di metodi di normalizzazione delle dimensioni dell'oggetto individuato per la corretta interpretazione dell'abilità, mostrata dalla rete, di effettuare il riconoscimento nel caso di *bounding box* con

dimensioni, relative al frame, variabili. La stima della prestazione di un *detector* è quindi frequentemente affidata a un parametro che tenga conto della varietà delle classi presenti, come la *Mean Average Precision*. Di seguito se ne discute brevemente l'origine e la sua espressione in forma matematica.

4.2.1 Mean Average Precision (mAP)

4.2.1.1 Precision e Recall

La capacità di fornire risposte corrette in funzione del compito disposto è, dalle origini del Machine Learning, affidata alla valutazione degli indicatori di *Precision* e *Recall*, i quali si propongono come parametri normalizzati e universali per task dove è possibile delineare con certezza la *positività* o *negatività*[1] del risultato ottenuto dal processo d'inferenza.

$$Precision = \frac{TP}{TP + FP} \tag{4.14}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.15}$$

I termini TP, FP e FN indicano rispettivamente *veri positivi*, *falsi positivi* e *falsi negativi*. La loro espressione mostra un rapporto di proporzionalità inversa per il quale un punteggio eccessivo ottenuto da un parametro ne implica uno altrettanto basso nell'altro.

È chiaro che, in un sistema discreto dove sia richiesto un processo decisionale, l'esito sia influenzato dalla quantizzazione dell'intervallo delle risposte possibili. Nel caso di un classificatore questo implica la presenza di una soglia, o *threshold*, che permetta di distinguere tra un esito *positivo* e uno *negativo*. La variabilità di questa soglia produce un effetto significativo sulle performance della rete convoluzionale

[1] Un risultato "*vero* positivo" o "*vero* negativo" comporta una corretta detection per, ad esempio, l'appartenenza o meno di un soggetto a una determinata classe. Al contrario, risultati di "*falso* positivo" o "*falso* negativo" individuano errori nel processo di decisione del sistema in esame, con conseguente errata assegnazione della proprietà di appartenenza.

che si sta valutando, ed è quindi necessario effettuare una scelta sensibile affinché si ottenga il miglior risultato possibile in funzione dell'obiettivo del task. Nel caso della Object Detection un metodo ampiamente adoperato per determinare il livello di *threshold* è il parametro *Intersection over Union*.

4.2.1.2 Intersection Over Union

Noto anche come *indice di Jaccard*, questo parametro valuta con quale accuratezza l'area delineata dal modello, durante la produzione delle *bounding box* candidate, corrisponda all'area descritta nel *ground-thruth*.

$$IoU = \frac{|A \bigcup B|}{|A \bigcap B|} \tag{4.16}$$

Essendo questo un compito di *detection* dove i confini delle *regioni d'interesse* descritte sono segmenti, a due a due paralleli, individuati da 4 vertici, e che formano quindi un rettangolo, questa valutazione può essere condotta con semplicità intersecando le *box* ottenute e determinando il rapporto tra l'area data dall'*intersezione* delle due e l'area data dalla loro *unione*.

Figura 4.1: Differenti livelli d'Intersection over Union

Valori più elevati comportano conseguenti requisiti più stringenti e un numero di *detection* considerate valide tendenzialmente più basso, ma garantisce un minor

rischio di generazione di *falsi positivi*.

Per il *Pascal Visual Objects Classes*[20] il valore di soglia per l'*Intersection over Union* è stato determinato in fase d'istituzione della challenge come pari a 0.5.

4.2.1.3 Average Precision

L'*Average Precision* consente di mediare le valutazioni di *Precision* ottenute in funzione dei valori di *Recall*. Concretamente, questo valore corrisponde all'area sottesa dalla curva *Precision-Recall* ed è più rigorosamente descritta come:

$$AP = \int_0^1 p(r) \cdot dr$$

La sua natura comporta un andamento, per valutazioni d'intervalli discreti di p e r, fortemente segmentato e necessita tipicamente di operazioni di *smoothing* per la valutazione nel caso di numero ridotto di campioni.

Il computo del parametro di *mAP*, *Mean Average Precision*, riguarda infine un processo di media dei valori di AP finora ottenuti dipendente dal numero di classi presenti nel dataset e che si *intende riconoscere*. Ovviamente, nel caso di singola classe, le due valutazioni coincidono.

4.3 OpenMV Cam

Le reti selezionate per l'esecuzione su OpenMV Cam appartengono alle famiglie YOLO e SSD e sono le varianti *Tiny YOLOv3* e *MobileNet-SSD*, individuate per la ridotta complessità e potenziale adattabilità alla piattaforma. In entrambi i casi si tratta di Object Detector di tipo *single shot* e che quindi propongono *bounding box* candidate a partire da *anchor box* in posizione predeterminata in fase di costruzione del modello. In questo elaborato si è seguita l'indicazione fornita dai rispettivi articoli d'origine per individuare una metrica comune per il paragone.

Sfortunatamente l'integrazione su OpenMV Cam ha però evidenziato numerose problematiche che ne hanno, in ultimo, determinato il temporaneo abbandono in

attesa di futuri sviluppi delle librerie e degli strumenti impiegati. Nondimeno è stata possibile una valutazione di massima delle prestazioni previste, in caso di successo nell'esecuzione, che ha mostrato l'inadeguatezza del dispositivo al compito della Pedestrian Detection in ambito automotive.

4.3.1 Strumenti di conversione

4.3.1.1 CubeAI 6.0

La suite di conversione fornita dalla ST Microelectronics rappresenta un sistema completo per l'esecuzione di reti neurali su dispositivi della medesima casa di produzione. Il software nello specifico, CubeAI[2], presente come *pack* o espansione del noto framework CubeMX, predispone una catena di preparazione, elaborazione e conversione di alcuni tra i più noti formati file adoperati nel mondo del *machine learning* e proveniente da librerie come Caffe, Keras e la più recente TFLite.

Il formato descrittore dell'architettura sviluppata viene quindi interpretato per una ricostruzione tramite runtime interna, e vengono trasferiti i parametri dei layer corrispondenti al fine di ripristinare le capacità originali del modello. A tal proposito è disponibile uno strumento di validazione dell'accuratezza con cui la rete è stata riprodotta, processo effettuabile sia *online* che indirettamente, od *offline*, tramite l'uso di un computer al quale collegare il microcontrollore in questa fase.

Utile ai fini dell'adattamento alla piattaforma hardware è la possibilità di attuare tecniche di compressione per quantizzazione su una parte dei layer attualmente riconosciuti, con una riduzione dell'occupazione in memoria pari a 4 od 8 volte l'originaria. L'impatto di questa strategia può essere riscontrato in minime alterazion dell'accuratezza che tipicamente non influenzano il corretto funzionamento della rete. La suite infine genera del codice, definibile *di boilerplate*, dal quale l'utente può iniziare lo sviluppo della soluzione *embedded* scelta.

[2]https://www.st.com/en/embedded-software/x-cube-ai.html

Nella fase d'integrazione dei modelli scelti è però emersa una mancanza di compatibilità con alcuni operatori coinvolti nell'inferenza. Pur comprendendo numerose operazioni, o layer, correttamente riconosciuti nelle loro varianti più comuni, come riportato nella documentazione del pack CubeAI, esistono infatti diversi vincoli sui parametri che la runtime è poi in grado d'interpretare correttamente.

Questo ha implicato un supporto parziale delle architetture scelte, delle quali è stato possibile analizzare, ai fini della conversione, per quanto riguarda la MobileNet-SSD, solo la porzione di *backbone*. I risultati, conseguiti per le varianti MobileNet con parametro *depth multiplier* pari a $[0.75, 1]$, hanno indicato una proiezione di occupazione RAM e Flash non compatibile con l'hardware selezionato. Nelle varianti con *depth multiplier* pari a $[0.15, 0.25, 0.5]$ si è invece riscontrato, in fase di allenamento, un livello di accuratezza insufficiente al giustificare il compromesso di compressione.

Nel caso della Tiny YOLOv3 la semplicità architetturale ne ha consentito piena conversione ma l'uso di un ingente numero di filtri, in ultimo stadio, prima della fase di detection, ha reso impossibile l'integrazione nel firmware a causa dei limiti di memoria del microcontrollore. In ultimo si è quindi rinunciato all'esecuzione in modalità *bare-metal* con la sola memoria Flash integrata.

4.3.1.2 TFLite Micro

L'interprete *TFLite for Microcontrollers*, come anticipato, costituisce un'implementazione in divenire dell'interprete di *flatbuffer* ereditato dal runtime TensorFlow Lite. Nonostante il progresso e i risultati conseguiti negli ultimi due anni nello sviluppo di questa soluzione, in grado di eseguire modelli di reti neurali a partire da un'occupazione minima in memoria di appena 16 kB (dato riguardante processori ARM Cortex-M3), la compatibilità con le architetture di rete esistenti è ancora incompleta e si è dimostrata ostacolo per l'esecuzione di un Object Detector completo.

Per la totale inclusione di una soluzione d'intelligenza artificiale su microcontrollore con interprete TFLite Micro, risulta necessario che, noto il grafo rappresentate il flusso dei tensori in ingresso, ogni suo nodo costituisca una *operazione* supportata

dalla versione scelta della runtime. Al contrario però di quanto è possibile fare per la variante *mobile* dell'interprete, non è contemplata l'ereditarietà del set di operazioni dal modulo per architetture AMD64, che tipicamente garantisce maggior compatibilità al costo di complessità e mancanza di ottimizzazione delle istruzioni.

È quindi vincolante che ogni modello che si desidera implementare sia esprimibile completamente tramite le seguenti `Op` riconosciute:

Tabella 4.1: Set di operazioni supportate dall'interprete TFLite Micro

Abs	GreaterEqual	ReduceMax
Add	HardSwish	Relu
AddN	L2Normalization	Relu6
ArgMax	L2Pool2D	Reshape
ArgMin	LeakyRelu	ResizeBilinear
AveragePool2D	Less	ResizeNearestNeighbor
BatchToSpaceNd	LessEqual	Round
Ceil	Log	Rsqrt
Concatenation	LogicalAnd	Shape
Conv2D	LogicalNot	Sin
Cos	LogicalOr	Softmax
CumSum	Logistic	SpaceToBatchNd
DepthToSpace	MaxPool2D	Split
DepthwiseConv2D	Maximum	SplitV
Dequantize	Mean	Sqrt
DetectionPostprocess	Minimum	Square
Elu	Mul	Squeeze
Equal	Neg	StridedSlice
EthosU	NotEqual	Sub
Floor	Pack	Svdf
FloorDiv	Pad	Tanh

Tabella 4.1: Set di operazioni supportate dall'interprete TFLite Micro

FloorMod	PadV2	TransposeConv
FullyConnected	Prelu	Transpose
Greater	Quantize	Unpack

Anche in questo caso la condizione di compatibilità non è stata soddisfatta per l'intera struttura delle reti scelte, delle quali la rete con maggior supporto rimane la Tiny YOLOv3, mentre per la MobileNet-SSD la sola parte supportata è ancora la *backbone* con l'inclusione di ulteriori stadi di convoluzione. Al contrario però della suite CubeAI, è stato possibile eseguire un *mock-up* dei detector selezionati con l'uso di pesi non allenati, mancante solamente, in entrambi i casi, della struttura necessaria alla decodifica e al filtraggio dei candidati (operazioni di Non Maximum Suppression, *sorting* Top-K).

4.3.2 Limiti di memoria ed esecuzione

Benché sia possibile l'esecuzione di modelli con dimensioni significative, data la presenza di memoria ausiliaria sulla quale effettuare l'*offloading* della rete, questa è tendenzialmente sconsigliata, come riportato nella documentazione online per la OpenMV Cam, per le implicazioni che la tecnica comporta, come l'estensione non trascurabile dei tempi d'inferenza.

Si è scelto di condurre comunque una prova di elaborazione come simulazione concreta del processo di detection che si otterrebbe con modelli totalmente supportati. Si riportano alcune combinazioni di parametri scelti per il modello MobileNet-SSD ed il modello Tiny YOLOv3: dove presenti, i *Frame per Second* registrati durante l'esecuzione fungono da indicatore di prestazioni, mentre la loro assenza implica la saturazione della memoria disponibile in fase di caricamento della rete e l'impossibilità nel proseguire l'esecuzione.

Tabella 4.2: Confronto esecuzione delle varianti MobileNet-SSD[3] convertite tramite framework TensorFlow v2.2

Modello	Risoluzione Input	Depth-Multiplier	Flash	FPS
MobileNetv2-SSD	320 × 320 × 3	1	3.98 MB	-
MobileNetv2-SSD	320 × 320 × 3	0.75	3.22 MB	-
MobileNetv2-SSD	320 × 320 × 3	0.5	2.66 MB	-
MobileNetv2-SSD	320 × 320 × 3	0.25	2.31 MB	-
MobileNetv1-SSD	320 × 320 × 3	0.15	0.98 MB	0.73

Tabella 4.3: Confronto esecuzione delle varianti Custom Tiny YOLOv3 convertite tramite framework TensorFlow v2.2

Modello	Risoluzione Input	Flash	FPS
Tiny YOLOv3	416 × 416 × 3	8.59 MB	0.11
Custom Tiny YOLOv3	416 × 416 × 3	8.59 MB	0.13
Custom Tiny YOLOv3	320 × 320 × 3	8.59 MB	0.19

Non è possibile determinare un range di valori *fps* per i quali la soluzione è considerata, in assoluto, *real time*, ma è comune ottenere, in sistemi ADAS, performance comprese tra i 5 e i 60 *frame per second*.

A meno di eventuali evoluzioni significative nell'integrazione e ottimizzazione delle routine necessarie al processo di *detection*, è quindi evidente che la piattaforma risulti inadeguata per il compito scelto, con risultati non all'altezza sia in termini di rapidità d'inferenza, sia, a causa della forte compressione della rete di *backbone* nel caso della MobileNet-SSD, di accuratezza nel riconoscimento della figura umana.

[3]Solo porzione di *backbone* MobileNet, prima e seconda revisione.

4.4 Raspberry Pi con acceleratore Coral

I test sulla piattaforma Raspberry Pi 4 sono stati condotti sia eseguendo l'inferenza direttamente sul processore ARM Cortex A72, sia tramite l'uso dell'acceleratore di reti neurali Google Coral Edge TPU. Questa metodologia consente d'individuare un livello di performance considerabile *baseline*, sufficientemente rappresentativo di ciò che allo stato attuale si può ottenere su hardware accessibile, a basso costo, e utile riferimento per una valutazione dell'incremento di prestazioni dato dall'uso dell'ASIC.

L'interprete TFLite consente, nella revisione corrente, di attuare il processo d'inferenza su un numero variabile di *threads* per far leva sulla oggi consueta architettura multi-core, multi-thread delle CPU. Nel caso del processore ARM si sono quindi effettuati test con calcoli distribuiti, in iterazioni separate, su uno e su quattro threads. Limite di questa tecnologia è la compatibilità dei layer del modello in esecuzione con i meccanismi di parallelizzazione: sono infatti completamente supportate, ad ora, solamente le operazioni di convoluzione.

I modelli sui quali si è concentrato il confronto appartengono ancora alle famiglie YOLO e MobileNet-SSD, con la successiva introduzione della EfficientDet: di questi sono state studiate, oltre all'implementazione *custom tiny YOLOv3* precedentemente illustrata, alcune varianti al fine di massimizzare la compatibilità degli operatori con l'unità TPU e ottenere una riduzione significativa del ritardo di processamento che incorre tra l'acquisizione dell'input, il riconoscimento delle classi presenti e la predizione delle corrispondenti *bounding box*. Come per i test svolti su OpenMV Cam, non si terrà conto, invece, dei meccanismi di filtraggio e ordinamento dei candidati alla detection, processo più strettamente legato alla tipologia di task che si intende svolgere e influenzato, spesso, da vincoli indipendenti dalla piattaforma adoperata.

La valutazione ha quindi riguardato i seguenti aspetti:

- rapidità d'inferenza su CPU e acceleratore di reti neurali;
- accuratezza delle detection, con attenzione particolare sulla classe *person*;

- numero di parametri del modello;
- proiezione dello *storage cost*.

Il *tempo d'inferenza* stimato è, tra queste, una metrica fondamentale, per la quale si è cercato di evidenziare la compatibilità con un'esecuzione considerabile *real time* anche nell'ipotesi di uso su hardware *low power*.

4.4.1 Compilazione per Edge TPU

Ogni modello analizzato ha subito una fase di preprocessamento nella quale è stato sottoposto a compressione tramite applicazione della quantizzazione *dinamica*[4] dei parametri: il formato di destinazione scelto è stato l'*intero a 8 bit*(`int8`), a eccezione degli stadi di *input* ed *output*, espressi in virgola mobile a 32 bit per consentire l'uso di un framework di *evaluation* uniforme per tutte le diverse tipologie testate.

L'uso dell'unità di accelerazione ha richiesto inoltre un'ulteriore procedura di adeguamento della rete che segue la conversione in formato Flatbuffer TFLite. A tale scopo, sul sito di riferimento Google Coral[5], viene reso disponibile un compilatore appositamente sviluppato per l'architettura a elevato parallelismo della Edge TPU.

Qualora parte delle operazioni presenti nel modello scelto non risultasse compatibile con l'esecuzione su unità esterna, il compilatore divide il flusso di elaborazione della rete in più rami e restituisce il controllo alla CPU *host* per il completamento delle porzioni non supportate. Questo scambio può avvenire più volte nel corso dell'inferenza e, a seconda della frequenza con cui ciò avviene, può introdurre ritardi significativi dovuti alle operazioni di trasferimento dei tensori e il *throughput* disponibile sull'interfaccia.

Sono quindi state adoperate, come anticipato, varianti dei modelli scelti, specialmente quelli appartenenti alla famiglia YOLO, nelle quali le operazioni non supportate sono state sostituite da altre, funzionalmente equivalenti, con un impatto minimo sull'accuratezza complessivamente misurata.

[4] https://www.tensorflow.org/lite/performance/post_training_quant
[5] https://coral.ai

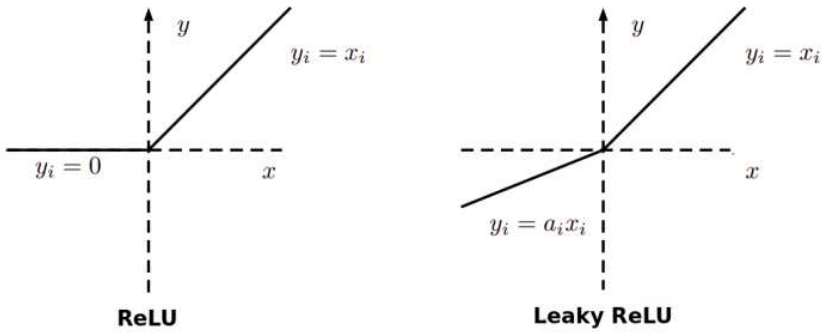


Figura 4.2: Differenza tra ReLU classica e ReLU "Leaky"

Nello specifico, per l'architettura Tiny YOLOv3 è stato necessario sostituire la tipologia di funzione di attivazione, normalmente adoperante una *Leaky ReLU*, con l'approssimazione fornita dall'implementazione *ReLU* classica. L'esclusione del parametro α, introdotto per preservare parzialmente l'effetto di termini negativi penalizzanti per i nodi altrimenti inattivi nella ReLU, ha permesso di riportare correttamente l'esecuzione dell'intero grafo su Edge TPU, riducendo ulteriormente il tempo di esecuzione.

Ne è un esempio l'architettura per la Object Detection nota come *EfficientDet*, considerata stato dell'arte per il task proposto: nella sua descrizione originaria l'elevata complessità e l'uso di operatori non presenti nella lista di compatibilità del compilatore per Edge TPU ha reso impossibile anche una parziale conversione. Essendo un problema attualmente noto, gli stessi ideatori del modello hanno fornito una rete, con la medesima natura della prima versione, ma pensata per l'inferenza su dispositivi con risorse computazionali limitate. L'architettura, chiamata *EfficientDet-Lite0*, prevede infatti una risoluzione d'ingresso ridotta a 320×320 ed è compatibile, nella maggior parte delle sue operazioni, all'esecuzione su Edge TPU: nel confronto effettuato essa funge come nuovo termine di paragone, in accuratezza e tempi d'inferenza misurati, per la variante *Custom Tiny YOLOv3* precedentemente introdotta.

4.4.2 Performance

La tabella 4.4 mostra le prestazioni ottenute sia nel caso di esecuzione dei modelli scelti su Raspberry Pi, sia in caso di offloading sull'acceleratore Google Coral. Per l'inferenza su CPU vengono riportati i dati riguardanti l'uso, rispettivamente, di uno e quattro thread. I tempi rilevati sono calcolati sulla media di dieci iterazioni consecutive del grafo e non prendono in considerazioni ulteriori elaborazioni dei dati d'input e output. Per tutti i modelli considerati, la dimensione del tensore di ingresso è stata scelta pari a $320 \times 320 \times 3$.

Tabella 4.4: Confronto dei tempi d'inferenza per i modelli di rete esaminati.

Modello	$mAP_{IoU=0.5}$	Storage	Param.	CPU (1/4 th.)	TPU
MobileNetv1-SSD	32.9%	6.6 MB	5.8 M	271.16/84.51 ms	20.84 ms
MobileNetv2-SSD	72.7%	5.9 MB	6.1 M	260.30/78.59 ms	23.86 ms
Tiny YOLOv3	60.2%	8.5 MB	8.7 M	517.36/160.16 ms	44.65 ms
Custom Network	48.7%	**4.0 MB**	4.0 M	**177.92/64.79** ms	**13.73 ms**
EfficientDet-Lite0	70.8%	4.4 MB	3.2 M	409.98/163.52 ms	118.94 ms

Il modello *Custom Tiny YOLOv3* si distingue per un'esecuzione rapida anche tramite il solo uso del processore ARM Cortex, con prestazioni compatibili con l'uso per task *real time*, ottenendo tra i 5.5 e i 16 FPS a seconda del numero di thread impiegato. Con l'uso di un'unità di accelerazione Edge TPU è possibile raggiungere un tempo d'inferenza costantemente inferiore ai *14 ms*, equivalenti a **71 FPS**, garantendo quindi una risposta immediata agli eventi di *detection*.

La metrica mAP risente della variazione di architettura, rispetto alla variante Tiny YOLOv3, e dell'applicazione della tecnica di compressione per decomposizione. Si dimostra però, in termini di rapidità d'inferenza, una valida alternativa al riferimento dato dalla MobileNetv1-SSD, riportando una latenza inferiore di 7.1 ms ma totalizzando 14.8 punti percentuali in più nella fase di test, valutazione per la quale si è considerato il parametro IoU=0.5, metrica standard per il dataset Pascal

VOC. Elemento d'interesse per il task di *Pedestrian Detection* è l'AP ottenuto per la categoria "*person*," che supera la media delle classi con un punteggio di 56.0%.

4.4.3 Limiti

Il modello introdotto rinuncia, rispetto alla configurazione riportata nella Tiny YOLOv3 originale, all'uso di una seconda *grid* di detection con un più elevato fattore di divisione e quindi maggior risoluzione, a favore di un costo computazionale più basso e a una latenza minima.

Questo comporta una minor sensibilità della rete alla presenza di oggetti di piccola dimensione, relativamente al frame, e di conseguenza *detection* effettuate su elementi posti a distanze più significative dall'obiettivo.

La strategia scelta è però in linea con la necessaria riduzione della risoluzione con la quale l'immagine viene sottoposta al modello, che per sistemi *embedded* è generalmente bassa per favorire la successiva elaborazione, e per la quale l'uso di una divisione eccessiva del frame per il task di detection porta un beneficio minimo.

Capitolo 5

Conclusioni

Complessivamente, lo studio condotto in questo elaborato ha evidenziato nella *Custom Tiny YOLOv3* un'architettura particolarmente adeguata ad ambienti caratterizzati da *hard constraint*, come nel caso di microcontrollori e processori ARM privi di unità per il calcolo parallelo. La semplicità data dall'uso di un solo ramo di elaborazione e detection si è dimostrata elemento fondamentale, inoltre, per il raggiungimento di tempi d'inferenza estremamente ridotti.

Il valore di *mean Average Precision* più basso, insieme alla consistente performance in accuratezza sulla singola categoria *person*, mostra un modello maggiormente versato al task di *detection* di un numero di classi limitato, con capacità di generalizzazione marginalmente inferiori alle reti concorrenti nel medesimo ambito di applicazione *low power*.

Dal punto di vista dello *storage cost*, l'uso della decomposizione di Tucker ha consentito una riduzione considerevole del numero di parametri coinvolti nella costruzione dei layer di convoluzione, portando a una occupazione in memoria inferiore di un fattore due. È altresì possibile ottenere ulteriore compressione al costo di un maggior errore introdotto nell'approssimazione dei tensori coinvolti con conseguente abbassamento della *Average Precision*, tradeoff plausibile nel *porting* di modelli a elevato numero di filtri convoluzionali verso piattaforme come i microcontrollori.

Per questi ultimi, infine, librerie e strumenti sono oggi in continuo sviluppo e si

prevede, nell'immediato futuro, una maggior compatibilità con la runtime mobile TFLite nel task di *detection*, aprendo a nuove possibili applicazioni nel mondo dell'*edge computing*.

Bibliografia

[1] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, 2001, vol. 1, pp. I–I.

[2] M. Enzweiler and D. M. Gavrila, "Monocular pedestrian detection: Survey and experiments," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 12, pp. 2179–2195, 2008.

[3] P. Dollár, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: A benchmark," in *2009 IEEE conference on computer vision and pattern recognition*, 2009, pp. 304–311.

[4] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: An evaluation of the state of the art," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 4, pp. 743–761, 2011.

[5] D. Tomè, F. Monti, L. Baroffio, L. Bondi, M. Tagliasacchi, and S. Tubaro, "Deep convolutional neural networks for pedestrian detection," *Signal processing: image communication*, vol. 47, pp. 482–489, 2016.

[6] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, "Pedestrian detection with unsupervised multi-stage feature learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 3626–3633.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, 2005, vol. 1, pp. 886–893.

[8] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[11] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *arXiv preprint arXiv:1506.01497*, 2015.

[13] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[14] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[15] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[16] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *European conference on computer vision*, 2016, pp. 21–37.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[18] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, and Q. Tian, "Centernet: Keypoint triplets for object detection," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6569–6578.

[19] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[20] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[21] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*, 2014, pp. 740–755.

[22] J. Jin, A. Dundar, and E. Culurciello, "Flattened convolutional neural networks for feedforward acceleration," *arXiv preprint arXiv:1412.5474*, 2014.

[23] J. Redmon, "Darknet: Open source neural networks in c." http://pjreddie.com/darknet/, 2013--2016.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.