UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Implementazione di una CNN per la semantic segmentation in applicazioni di guida autonoma su piattaforma embedded

Implementation of a CNN-based semantic segmentation network for autonomous driving in an embedded system

Candidato:
Carlo Castagnari

Relatore:
Prof. Claudio Turchetti

Correlatore:
Dott.ssa Laura Falaschetti

Anno Accademico 2019-2020

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Oggi giorno, l'intelligenza artificiale (IA) è uno dei temi più caldi in cui gli scienziati e i ricercatori stanno investendo, principalmente dovuto alla sua grande versatilità e capacità nell'aiutare a compiere innumerevoli compiti in qualsiasi professione e non in modo automatizzato. Difatti possiamo trovare applicazioni di software intelligente nell'automatizzazione di routine lavorative, nella comprensione di immagini, testi o dialoghi, nelle diagnosi in medicina e come ausilio alla ricerca scientifica.

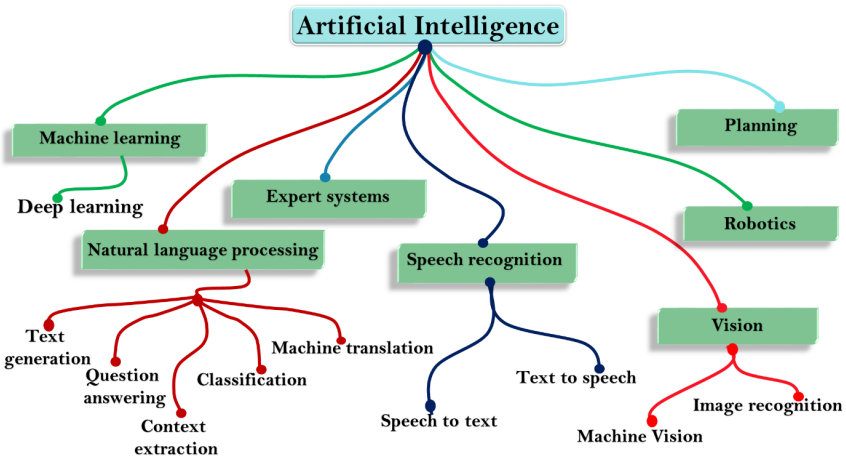

Figura 1.1: Categorie di applicazione dell'intelligenza artificiale

Sin dagli albori dell'intelligenza artificiale, attraverso il suo utilizzo, vennero affrontati e risolti problemi di natura intellettualmente complessa per l'uomo, ma relativamente semplici per i calcolatori. Proprio per questo motivo, la vera sfida per i ricercatori fu quella di far eseguire dei compiti facili per gli esseri umani, per lo più risolti in maniera intuitiva o automatica, ai calcolatori, tra i quali il riconoscimento delle parole in un discorso o dei volti delle persone nelle immagini. Per affrontare questa sfida, i ricercatori pensarono di fornire ai calcolatori la possibilità di apprendere tramite esperienza la conoscenza

del mondo, catalogandola in gerarchie di concetti, dove ogni macro area è definita per mezzo di nozioni più semplici; proprio grazie alla definizione di queste ultime, i calcolatori furono in grado di apprendere argomenti complicati descrivendoli tramite una composizione di concetti più semplici. In aggiunta, automatizzando la raccolta della conoscenza attraverso l'esperienza diretta, fu possibile eludere la definizione formale ed esplicita della conoscenza stessa da parte dei ricercatori, realizzando un metodo di apprendimento automatizzato. Questo approccio nel definire un'intelligenza artificiale attraverso l'apprendimento con esperienza diretta, determinò la nascita del *Machine Learning*. Per acquisire la conoscenza necessaria nell'eseguire quei compiti immediati per l'uomo, gli scienziati tentarono inizialmente un approccio a codifica fissa ma i risultati furono insoddisfacenti e ciò suggerì che i sistemi di IA necessitassero di apprendere la conoscenza in modo autonomo, estraendola direttamente dai dati forniti in ingresso. Questa capacità di *Machine Learning* consentì ai calcolatori di poter affrontare problemi richiedenti conoscenza dal mondo reale servendosi direttamente dell'esperienza accumulata durante la fase di apprendimento, per poi valutare quali fossero le decisioni adeguate ad ogni particolare circostanza. In aggiunta, sfruttando una struttura gerarchica per la classificazione della conoscenza, il calcolatore è appunto in grado di apprendere questi concetti complicati e se volessimo disegnare la struttura con cui essi vengono appresi, otterremmo un grafo molto esteso o profondo e composto da numerosi strati. Tale approccio venne definito *Deep Learning*.

Il connubio tra *Deep Learning* e *Machine Learning* portò alla definizione di numerose rappresentazioni di IA specializzate nell'aiutare l'uomo nei più svariati campi di applicazione. Uno dei più esplorati e fertili è proprio quello della *Computer Vision*, la quale consente di approssimare il più fedelmente possibile la visione del mondo reale attraverso l'utilizzo di dati 2D (fotografie), LIDAR, sonar; in breve ciò che si vuole ottenere è la riproduzione artificiale della vista umana. Però è necessario non fraintendere quest'ultima affermazione, infatti il termine vista non corrisponde alla possibilità di rappresentare ciò che si vede tramite una fotografia, ma bensì arrivare a comprendere le immagini e più precisamente la scena catturata dall'immagine stessa.

Attraverso l'utilizzo di strumenti per la cattura di immagini ed altri per la loro elaborazione, la *Computer Vision* riesce a superare di gran lunga le capacità percettiva dell'occhio umano. Difatti, come già introdotto precedentemente, l'unica mancanza è l'attuale capacità di comprendere correttamente ciò che le immagini riportano, in modo da poter classificare tramite la conoscenza pregressa cosa si sta al momento osservando. Per questo motivo, si va spesso ad affiancare ai mezzi di acquisizione, delle reti neurali di *Deep Learning* allenate

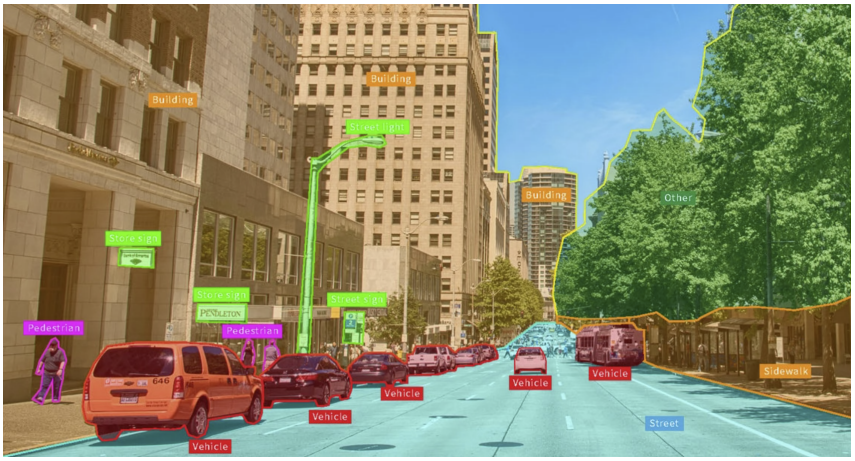

Figura 1.2: Esempio utilizzo Computer Vision per la classificazione della scena

adeguatamente al fine di svolgere l'attività di riconoscimento e classificazione. In sintesi, possiamo affermare che il *Machine Learning* e più in particolare il *Deep Learning* sono tra i campi più fiorenti dell'ultima decade ed hanno inoltre riscontrato una notevole crescita in popolarità come anche in utilità, specialmente grazie all'aumento della potenza computazionale dei calcolatori, alla presenza di grandi data-set per la fase di allenamento e allo sviluppo di nuove architetture, *Deep Network*.

1.1 Struttura tesi

La tesi verrà strutturata come segue: nell'attuale capitolo 1 il lettore avrà la possibilità di approcciarsi ad una panoramica del campo di applicazione della tesi, conoscendone brevemente i principali rami toccati.

Nel capitolo Reti Neurali nella Computer Vision, verrà accennata la storia dell'intelligenza artificiale, il passaggio dai primi approcci al *Machine Learning* a quelli più attuali, soffermandoci in particolare sulle reti neurali convolutive, la segmentazione semantica e l'edge ai.

Nel capitolo Definizione architettura della rete verrà illustrata la struttura della rete neurale, soffermandoci in particolare sulla *UNet* e sulla *MobileNet* e argomentando le scelte prese in fase di definizione dell'architettura stessa.

Nel capitolo Risultati e discussioni, verrà introdotto la struttura del dataset utilizzato, alcuni porzioni di codice relativi alla configurazione della procedura di apprendimento della rete e infine presentati i risultati ottenuti.

La tesi termina con il capitolo Conclusioni e sviluppi futuri, nel quale viene riassunto brevemente ciò che si è ottenuto e verranno discussi gli sviluppi futuri volti ad estendere il progetto di tesi.

Capitolo 2

Reti Neurali nella Computer Vision

2.1 Cenni storici e definizione

Quando la maggior parte delle persone sente la parola "Machine Learning", il primo pensiero va alla parola robot, confrontandoli ad esempio con quelli già visti nei film fantascientifici. Il *Machine Learning* (ML) non è solo una fantasia futuristica o un'utopia, ma bensì, come già introdotto precedentemente, è già presente nella vita quotidiana e ormai da decadi. Per essere più precisi, la prima applicazione di ML che diventò popolare, migliorando la vita di milioni di persone, risale agli anni '90, ovvero l'invenzione del *filtro spam*. Non è esattamente una IA con la consapevolezza della sua esistenza, ma può comunque essere tecnicamente definita un *Machine Learning*, dato che svolge il suo compito così bene che l'utente non ha più la necessità di etichettare personalmente una email come spam. Con il passare degli anni, furono realizzate numerose applicazioni ML che tuttora svolgono il loro lavoro silenziosamente in centinaia dei prodotti o servizi usati regolarmente dall'uomo. Ma esattamente che cosa significa *Machine Learning*?
Il *Machine Learning* è una scienza che descrive la programmazione dei computer affinché essi possano apprendere una conoscenza dai dati forniti. Più precisamente, una sua definizione più generica può essere:

> "Il Machine Learning è un campo di studio che offre ai computer la capacità di apprendere senza essere esplicitamente programmati."
>
> [Arthur Samuel, 1959]

Ritornando all'esempio del filtro spam, che come già sappiamo è un programma di ML, quando vengono forniti esempi di email classificate come spam ed esempi di quelle classificate come "regolari", esso sarà in grado di apprendere e segnalare lo spam. Gli esempi che il sistema utilizza nell'apprendimento vengono chiamati *istanze di allenamento* (*training instances*) o semplicemente *campioni*, mentre

nel loro insieme formano il *set di allenamento* (*training set*). Per valutare le performance del filtro spam è necessario definire un'unità di misura; per esempio, è possibile usare il rapporto tra le email classificate correttamente e non. L' unità di misura dell'esempio viene chiamata *precisione* (*accuracy*) ed è la più utilizzata nei compiti di classificazione svolti da un programma di *Machine Learning*.

Ora, la successiva lapalissiana domanda è: perché il Machine Learning?

La risposta è semplice e usando nuovamente l'esempio del filtro spam, possiamo dire che realizzare un programma manuale che riproduca esattamente lo stesso comportamento del servizio anti-spam basato su ML è tutt'altro che banale. Basti pensare che le email considerate spam vengono rilevate perché contengono un determinato pattern che le identifica. Però qualora gli spammer si accorgessero dell'eventuale blocco, ne cambierebbero prontamente il contenuto. Dunque considerando questa evenienza, lo sviluppatore di un filtro spam dovrebbe codificare una lunga lista di regole complesse che devono essere costantemente aggiornate. Contrariamente un filtro spam basato su tecniche di *Machine Learning* apprende automaticamente quali parole e frasi sono riconducibili a spam e riesce, appunto, ad individuare questi frequenti pattern di parole inusuali.

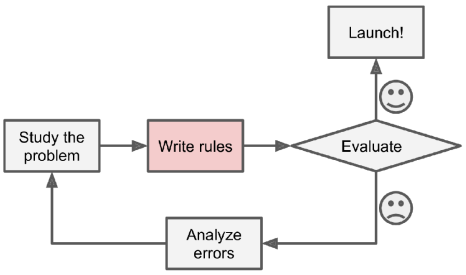

Fig. 2.1: Approccio tradizionale

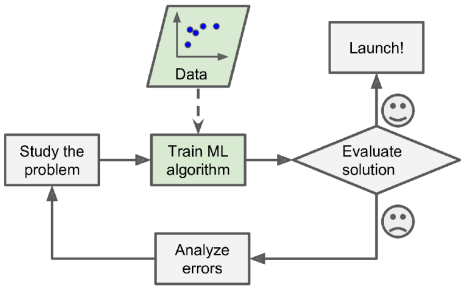

Fig. 2.2: Approccio ML

Come è possibile vedere nelle immagini 2.1 e 2.2 la codifica ML risulta molto più efficiente, semplice da mantenere e molto probabilmente più accurata, rispetto a quella realizzata manualmente da un programmatore.

Un'altra area in cui il *Machine Learning* è molto utilizzato è quella relativa ai problemi che risultano troppo complessi per gli approcci tradizionali o per cui non esistono algoritmi conosciuti; un esempio è il riconoscimento vocale. Infine, il *Machine Learning* può essere usato per aiutare l'uomo. Ritornando all'esempio del filtro spam, ispezionando tutto ciò che viene appreso ed estrapolandone le parole o combinazioni delle stesse che risultino essere i migliori indicatori di spam, è possibile, per di più, scoprire nuove correlazioni o trend che portino ad una migliore comprensione del problema stesso. Applicare tecniche di ML per

analizzare queste grosse moli di dati, può aiutare nello scoprire nuovi pattern che non sono immediatamente riconoscibili e prende il nome di *Data Mining*.

Nel 2006, Geoffrey Hinton [1] pubblicò un articolo in cui venne mostrato come allenare una deep neural network capace di riconoscere caratteri scritti a mano e per di più con una precisione eccezionale (>98%). Questo articolo pose le fondamenta alla rinascita del "*Deep Learning*". Per quanto riguarda una deep neural network, può essere definito come un modello semplificato della nostra corteccia cerebrale, composta da numerosi strati di neuroni artificiali. Sebbene la conoscenza di queste tipologie di reti fosse presente già negli anni '90, allenarle ad apprendere un determinato compito era considerato quasi impossibile e difatti numerosi ricercatori abbandonarono tale idea. Successivamente alla pubblicazione dell'articolo, il *Deep Learning* riscosse nuovamente un importante interesse da parte della comunità scientifica e da lì a poco non solo vennero redatte numerose pubblicazione, ma in aggiunta si conclamò la sua sbalorditiva capacità nell'ottenere risultati che nessun altro approccio di *Machine Learning* potesse minimamente uguagliare (con l'aiuto di una notevole maggiore potenza di calcolo e grandi moli di dati). Questo entusiasmo si estese molto presto a molte altre aree del *Machine Learning* e un decennio più tardi conquistò, addirittura, il settore dell'industria; infatti è divenuto l'elemento centrale di tutti i prodotti high-tech attuali, passando dal classificare le ricerche su web, permettere il riconoscimento vocale negli smartphone fino ad addirittura sconfiggere il più forte giocatore di Go al mondo. Dopo questa panoramica relativa al concetto di *Machine Learning*, è necessario approfondire alcuni termini fondamentali; come già accennato brevemente poco sopra, il concetto di rete è alla base della costruzione di deep network e consiste in un modello computazionale ispirato alle reti neurali biologiche presenti negli esseri umani e animali. Esse sono costituite da un insieme di interconnessioni entranti e uscenti dai neuroni artificiali e da processi che tentano di emulare il passaggio delle informazioni attraverso quest'ultimi. Nella maggior parte dei casi una rete neurale artificiale è un sistema adattivo che cambia in base ad informazioni esterne e/o interne che viaggiano attraverso la rete stessa nella la fase di apprendimento. Una rete neurale artificiale riceve segnali esterni su uno strato di nodi (*neurons*) detti d'ingresso, ciascuno dei quali è collegato a numerosi nodi interni (*hidden neurons*), organizzati in più livelli o strati. Ogni nodo elabora i dati ricevuti e trasmette il risultato ai successivi. Riassumendo, possiamo quindi definire le reti neurali come strutture non-lineari utilizzate per effettuare una modellazione statistica del fenomeno reale in modo da fornire la soluzione ad uno o più problemi specifici.

Come già brevemente anticipato nel capitolo 1, uno dei campi più esplorati è

quello della *Computer Vision*, ovvero la riproduzione della vista umana unita alla capacità di comprendere le immagini o più precisamente della scena catturata dall'immagine stessa. Per comprendere la scena presente in un immagine è necessario costruire una rete che permetta di estrapolare automaticamente quante più informazioni possibili, partendo dalle caratteristiche di più basso livello, come ad esempio un piccolo dettaglio, fino ad arrivare a quelle di più alto livello, quali ad esempio la comprensione della scena nella sua totalità. Idealmente, aspireremmo ad algoritmi di apprendimento che consentano la scoperta di queste caratteristiche con il minor sforzo umano possibile, ovvero senza la necessità di dover specificare manualmente tutte le astrazioni necessarie e in aggiunta senza dover fornire un enorme mole di esempi già etichettati; purtroppo al momento questa è la soluzione più utilizzata durante l'apprendimento. Ritornando invece alla *Computer Vision*, le prime reti di *Deep Learning* vennero utilizzate per riconoscere singoli oggetti in immagini estremamente piccole, ma con il passare degli anni, fu possibile introdurre un graduale aumento nelle dimensioni delle immagini che una rete neurale avesse potuto processare. Difatti nell'ultimo decennio esse furono in grado di processare immagini ad altissima risoluzione, senza necessitare un adeguato ritaglio nei pressi dell'oggetto da riconoscere; per di più, a differenza delle prime, le attuali reti di *Deep Learning* hanno la capacità di riconoscere un numero elevato di oggetti.

Nell'ultimo decennio, vista l'esponenziale crescita nell'utilizzo di reti di *Deep Learning*, vennero alla luce molteplici competizioni rivolte all'intelligenza artificiale, tra cui diverse relative *Deep Learning* applicato alla *Computer Vision*. Infatti, proprio al più grande evento nel campo della *Object Recognition*, l'*ImageNet Large Scale Visual Recognition Challenge* (ILSVRC), fu presentata una delle reti che rivoluzionò l'intero panorama della *Computer Vision*: la rete a convoluzioni. Per la prima volta la rete abbassò l'errore top-5 dello stato dell'arte da 25.8% al 16.4%. In altre parole, questo errore rappresenta una lista ordinata decrescente della probabilità di ogni oggetto riconosciuto dalla rete e il corretto risulta essere tra i primi 5 oggetti della lista tranne nel 16.4% dei casi. Negli anni successivi l'avanzamento tecnologico nel *Deep Learning* portò tale errore top-5 fino al 2.3%, come mostrato in figura 2.3.

In aggiunta, le *Deep Networks* riuscirono anche a distinguersi nel riconoscimento dei pedoni, nella segmentazione dell'immagine, così come nel riconoscimento dei segnali stradali. Per di più, allo stesso tempo le dimensioni e la precisione delle *Deep Networks* continuarono a crescere e come anche la complessità dei compiti da esse risolvibili; infatti le reti neurali arrivarono a riconoscere una sequenza di oggetti presenti nell'immagine, restituendoli in uscita. Un altro importante traguardo ottenuto dal *Deep Learning*, si riscontra anche nel campo

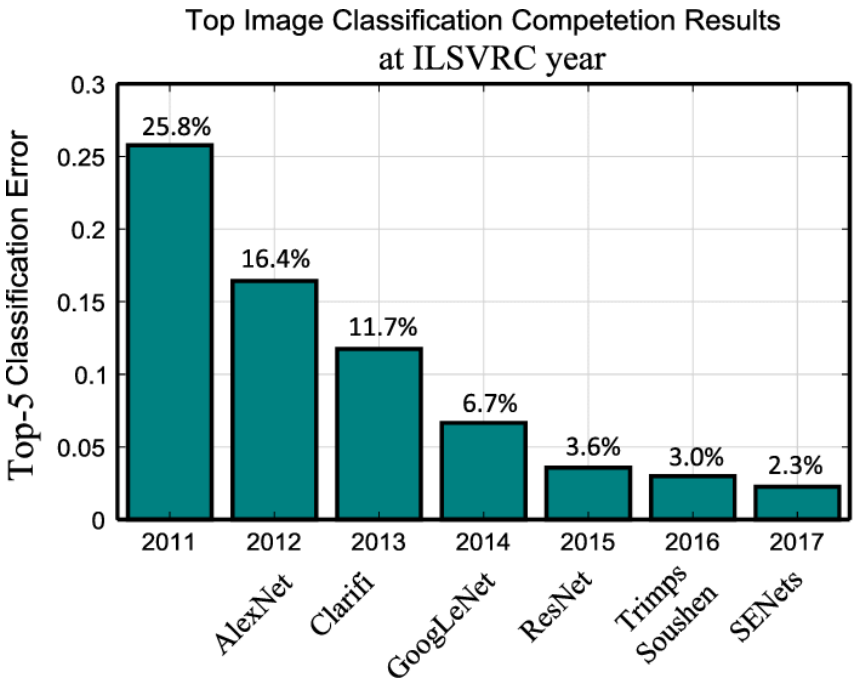

Figura 2.3: Errore Top-5 ILSVRC

del *Reinforcement Learning*; più precisamente in questo contesto si ha un agente autonomo che deve apprendere come eseguire un compito basandosi solamente sulla sua interazione con l'ambiente in cui è immerso. L'agente sceglie le azioni da eseguire, in base alla ricompensa ricevuta dal sistema per aver compiuto una determinata azione. Questa ricompensa va a valutare la bontà dell'azione scelta ed ha lo scopo di incoraggiare l'agente ad intraprendere comportamenti corretti. Con l'introduzione del *Deep Learning*, fu possibile raggiungere delle prestazioni decisionali superiori, in alcuni casi, a quelle del livello umano, accrescendone dunque il loro prestigio.

2.2 Convolutional Neural Network

Come accennato in 2.1, una delle *Deep Neural Network* (DNN) che rivoluzionò il campo della *Computer Vision* fu proprio la *Convolutional Neural Network* (CNN) introdotta da Krizhevsky in [2]. Le performance di questa DNN furono sorprendenti in termini di accuratezza nel riconoscere oggetti nelle immagini, e per di più siglò l'inizio dell'ascesa delle CNN nella *Computer Vision*. Grazie a questa nuova possibilità, nell'ultima decade, iniziò un'accelerazione allo sviluppo, da parte dei costruttori di auto, di veicoli a guida autonoma, i quali condurranno, in un futuro non lontano, ad un cambiamento radicale nell'efficienza dei sistemi di trasporto. In aggiunta, considerando la condizione delicata del nostro clima e

l'aumento di inquinanti nella nostra atmosfera, questa nuova strada intrapresa dall'industria nello sviluppo di veicoli e di trasporti elettrificati, così da ridurre al minimo il rilascio di inquinanti durante lo spostamento, fu il vero e proprio trampolino di lancio per l'adozione di tecniche di *Deep Learning* volte alla guida autonoma.

I veicoli a guida autonoma fanno uso di sistemi di *Computer Vision* basati sull'utilizzo di telecamere, radar e LIDAR per percepire l'ambiente limitrofe al veicolo e soprattutto avere la conoscenza necessaria per intraprendere un'azione ben specifica basata sulle evenienze riscontrate nei quasi illimitati scenari di guida. Come tutti ben sappiamo, le immagini o i frame di un video sono identificati come dati strutturati a forma di griglia, dove ogni rettangolo che la compone indica un pixel; per pixel intendiamo l'unità minima di un'immagine digitale (picture element, pict-el). Proprio per questa sua struttura a griglia, i sistemi più idonei nella *Computer Vision* per manipolare questa tipologia di dati ricadono proprio nelle CNN. Infatti, esse possono essere definite come delle reti neurali specializzate per trattare dati che hanno una ben specifica topologia a griglia. Vengono chiamate reti neurali convolutive semplicemente perché utilizzano la convoluzione al posto della più generale moltiplicazione tra matrici. Le CNN giocarono un ruolo molto importante nella storia del *Deep Learning* e infatti sono uno esempio chiave della corretta implementazione, in applicazioni di *Machine Learning*, di intuizioni ottenute studiando il cervello umano e animale. Furono, inoltre, uno dei primi modelli a operare correttamente in ambito commerciale, molto prima che le DNN si affermassero. Ad ogni modo, non è ancora completamente chiaro il perché le reti convolutive ebbero successo mentre le più generali reti neurali furono considerate un "fallimento". Forse può concernere il semplice fatto che le reti convolutive fossero molto più efficienti da un punto di vista puramente computazionale rispetto alle reti completamente connesse, e dunque risultò più facile realizzare numerosi esperimenti al fine di ottenere una migliore configurazione implementativa. Per di più, le reti molto estese (*Deep Network*) risultarono molto più propense a produrre risultati soddisfacenti durante la fase di allenamento, grazie anche all'introduzione di un hardware più performante ed adeguato, seppur utilizzando gli stessi data-set con i quali le reti completamente connesse furono reputate non correttamente funzionanti. Qualunque fosse il motivo del successo delle CNN, fu chiaro che esse furono le fondamenta per la rapida crescita nell'interesse nel *Deep Learning* e spianarono la via alle reti neurali in generale.

Esistono diversi campi di applicazione delle CNN, ma quella più popolare resta comunque l'identificazione probabilistica di cosa un'immagine rappresenta. La *semantic segmentation* è uno degli algoritmi di *Deep Learning* più utilizzati nel

mondo della *Computer Vision*. Infatti, esso viene utilizzato per capire cosa sia presente in un'immagine e di conseguenza va ad associare un'etichetta o una categoria ad ogni pixel che si sta analizzando. La segmentazione semantica viene utilizzata in numerose applicazioni, quali la guida autonoma, analisi di immagini medicali, ausilio alla guida di un robot e nelle immagini satellitari. Attraverso l'utilizzo di *Deep Architecture*, è stato possibile migliorare la classificazione dei pixel di un'immagine sia in termini di accuratezza che di efficienza, superando di gran lunga le vecchie architetture; una delle rete più utilizzate nel compito della *semantic segmentation* è proprio la *Convolutional Neural Network*.
Abbia introdotto il concetto di rete convolutiva, ma prima cosa è necessario introdurre l'operazione di convoluzione. Nella forma più generale possibile, la convoluzione è un operazione definita su due funzioni (f e g) a valori reali. Più precisamente consiste nell'integrale del prodotto tra queste due funzioni, di cui la seconda (g) è rovesciata e traslata di un valore t.

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

In questo contesto, la convoluzione può essere descritta come la media pesata della funzione $f(\tau)$ all'istante t dove $g(-\tau)$ è la sua funzione peso traslata di un intervallo t; al cambiare di t la funzione peso enfatizza parti diverse di $f(\tau)$. L'operazione di convoluzione è spesso identificata con l'asterisco:

$$s(t) = f(\tau) * g(t - \tau)$$

Ritornando alle CNN, usualmente la convoluzione viene effettuata tra un *input* e un *kernel*, il cui risultato viene identificato come *feature map*. Nelle applicazioni di ML, l'*input* è comunemente definito da un vettore di dati a più dimensioni, mentre il *kernel* è un vettore di parametri a più dimensioni, dove tali parametri sono regolati da uno specifico algoritmo di apprendimento. Questi vettori multidimensionali vengono definiti come *tensori*. La convoluzione può essere definita su più assi, come ad esempio nel caso delle CNN relative alla *Computer Vision*; infatti, visto l'utilizzo di dati strutturati a griglia, viene realizzata una convoluzione su due dimensioni.
Un strato convolutivo di una CNN (fig. 2.4) consiste essenzialmente di tre stadi:

- nel primo, lo strato iniziale esegue convoluzioni in parallelo producendo un insieme di soluzioni lineari
- nel secondo, ogni soluzione lineare viene utilizzata come ingresso di una funzione di attivazione non lineare, ad esempio utilizzando un rettificatore.

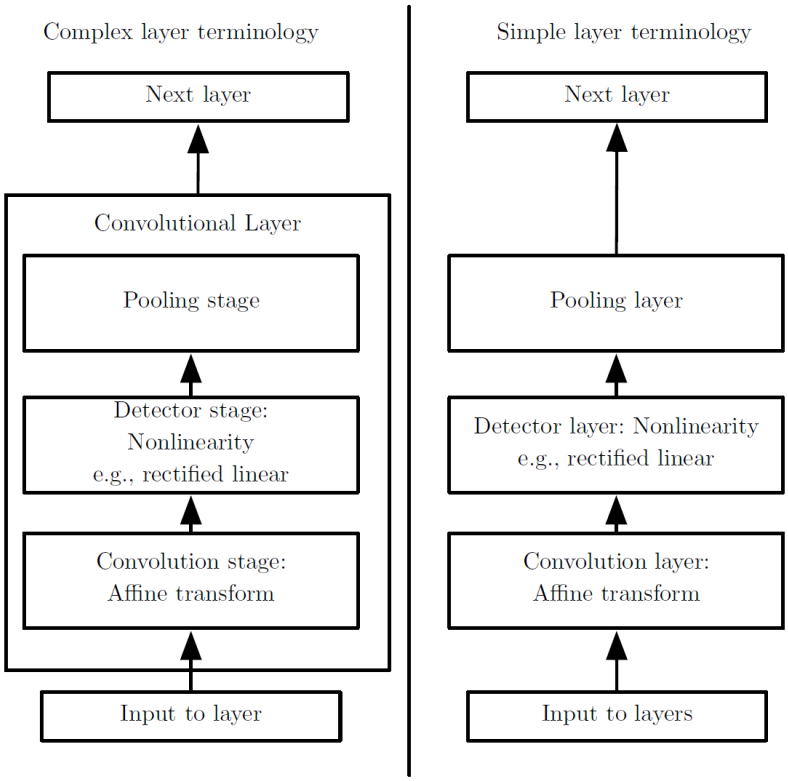

Figura 2.4: Composizione di uno strato convolutivo di una rete neurale.

Questo stadio è anche chiamato stadio rivelatore

- nel terzo ed ultimo, viene utilizzata una funzione di *pooling* per modificare ulteriormente l'output dello strato convolutivo. La funzione di *pooling* viene utilizzata per sostituire l'output della rete in un qualsiasi punto con un statistica riassuntiva di tutti gli output delle convoluzioni eseguite in parallelo.

Tutte le differenti tipologie di *pooling* hanno lo scopo di mitigare la rappresentazione dell'output finale rendendolo invariante alle differenze introdotte dall'input; ciò significa che se l'input subisce delle variazioni dovute al non perfetto pre-processamento dei dati, la maggior parte dei valori in uscita dallo stadio di *pooling* rimarrà inalterato. L'utilizzo del *pooling* migliora notevolmente l'efficienza statistica della rete.

2.3 La segmentazione semantica

Dopo una introduzione al *Machine Learning* e un approfondimento alle CNN, è necessario approfondire le aree della *Computer Vision* con il più alto utilizzo di queste reti neurali convolutive, ovvero la *semantic segmentation*

e la *object detection*. Ci focalizzeremo principalmente sulla prima, visto che lo scopo della tesi è appunto quello di realizzare una rete a convoluzioni specializzata nella segmentazione semantica nell'ambito della guida autonoma. Nella segmentazione semantica, ogni pixel, relativo ad un'immagine, viene classificato in base alla classe dell'oggetto a cui appartiene, come ad esempio strada, automobile, pedone, costruzione, visibile nella figura 2.5.

Figura 2.5: Esempio di segmentazione semantica

Però è necessario puntualizzare che differenti oggetti della stessa classe non possono essere distinti. La principale difficoltà di questo approccio è dovuta alla propagazione delle immagini nella CNN; man mano che esse proseguono negli strati convolutivi della rete, la risoluzione spaziale dell'immagine va a perdersi (dovuto principalmente a quegli strati con *stride* di dimensioni maggiori a 1). Dunque, anche se una CNN è in grado di definire che in un qualsiasi punto dell'immagine è presente una persona, non sarà in grado riferire ulteriori informazioni a riguardo. Per poter risolvere questo problema, alcuni ricercatori proposero diverse metodologie al fine di recuperare le informazioni spaziali perse durante la propagazione dell'immagine nella rete; un esempio è quello riportato dall'articolo [3], dove viene proposto, un approccio che parte dalla trasformazione della CNN in una FCN (Fully Convolutional Network), cioè sostituendo tutti gli strati non convolutivi della CNN a favore di strati convolutivi 1x1 (con *stride* = 1). Successivamente, dato che la CNN da loro proposta provoca uno *stride* complessivo pari a 32 nei confronti dell'immagine in input, riducendo appunto le sue dimensioni di appunto 32 volte, l'immagine risulta essere diventata troppo grezza; per questo motivo, gli autori hanno inserito uno strato di ri-campionamento finale per riportala alle dimensioni originali. Oltre quello proposto nell'articolo di cui sopra, esistono molteplici approcci di *upsampling*, come ad esempio l'interpolazione bilineare, la quale opera abbastanza bene fino a x4 o x8. Per vincere queste limitazioni, viene usualmente utilizzata una convoluzione trasposta, la quale è equivalente ad introdurre righe e colonne vuote all'interno della struttura a griglia, in modo da

ampliare la dimensione dell'immagine, e successivamente eseguire una convoluzione normale (alternativamente può essere vista come una convoluzione con *stride* frazionario). Dato che la convoluzione trasposta è definita come strato di una rete, le sue performance migliorano man mano che la rete viene allenata.

Network	MeanIoU	Inference Time
FCN-8s [3]	P.VOC2011 test: 62.7	175 ms (GPU)
DeepLab-CRF(ResNet101) [4]	Cityscapes test: 70.4	_
EfficientNet-B7(self training) [5]	P.VOC2012 val: 86.7	_
GALDNet(ResNet50) [6]	Cityscapes test: 80.8	_
DeepLabv3(ResNeSt50) [4]	Cityscapes test: 79.87	_
MobileNetv3 [7]	Cityscapes test: 72.6	2.47 s (Pixel3)
EfficientSeg (depth: 1.5) [8]	Cityscapes test: 51.5	_
EfficientSeg (depth: 6) [8]	Cityscapes test: 58.1	_
FC-HarDNet68 [9]	CamVid: 62.9	13.3 ms (GPU)
FC-HarDNet76 [9]	CamVid: 65.8	21 ms (GPU)
FC-HarDNet84 [9]	CamVid: 67.7	28 ms (GPU)

Tabella 2.1: Stato dell'arte CNN

Nella tabella 2.1 sono evidenziate le attuali reti di *Deep Learning* che rappresentano lo stato dell'arte nella *Semantic Segmentation* nella comunità scientifica. Sfortunatamente esse sono state allenate e testate su diversi data-set e dunque la tabella non risulta essere una puntuale ed omogenea rappresentazione delle performance di ogni rete, ma comunque quelle selezionate risultano essere le più accurate e performanti in termini di precisione e tempo di inferenza. Nella tabella sono state specificate le diverse metriche per i differenti task cui le DNN sono state definite, quali:

1. la più alta precisione in termini di meanIoU (*Mean Intersection over Union*), unità di misura utilizzata nella *Semantic Segmentation* per quantificare la percentuale di sovrapposizione fra la vera maschera dell'oggetto (target) e la predizione effettuata della rete;

2. tempo di inferenza per effettuare la *segmentazione semantica* di un'immagine, in cui è anche specificato la tipologia di hardware utilizzata.

In merito all'obiettivo del progetto di tesi, ovvero quello di realizzare una rete CNN per la *Semantic Segmentation* in applicazioni di guida autonoma su piattaforme embedded, diviene indispensabile investigare lo stato dell'arte relativo alle CNN per la *Semantic Segmentation* che richiedano le minori performance di inferenza, vista la limitata capacità di calcolo e di memoria a disposizione. Difatti, le reti che soddisfano questo requisito risultano raggiungere

una precisione minore in termini di accuratezza, ma garantiscono un tempo di inferenza molto più basso, riducendone per di più sia il quantitativo di memoria che di potenza di calcolo indispensabile. Questo compromesso è la base delle DNN realizzate per le applicazioni real-time, le quali vanno a rinunciare all'accuratezza per garantire gli stretti tempi di inferenza richiesti dal determinato tipo di applicazione cui utilizzate. Le DNN che riescono ad ottenere un ottimo rapporto tra accuratezza e velocità di inferenza sono MobileNetv3 [7] ed EfficientSeg [5]. La MobileNet [10] è una DNN pensata per essere sfruttata da dispositivi con una bassa potenza di calcolo come dispositivi mobili e embedded, ma comunque orientati alla *Computer Vision*. Invece la EfficientSeg è una DNN che sfrutta diversi paradigmi utilizzati nella segmentazione semantica, cioè un'architettura di base chiamata *U-Net* fusa con quella della MobileNetv3. Nello specifico, con fusione si intende la sostituzione degli strati convolutivi standard con quelli proposti dalla MobileNetV3, ovvero la fattorizzazione della convoluzione in *depthwise* e *pointwise* e l'inserimento di strati chiamati *bottleneck*. Nel capitolo Definizione architettura della rete verrà approfondito il motivo dell'utilizzo di tali strati e i benifici che ne conseguono.

2.4 Edge AI

Con la proliferazione di dispositivi elettronici e di archiviazione, dai cloud datacenter ai computer, smartphone, indossabili e dispositivi IoT (Internet of Things), siamo entrati in un era centrata sullo scambio di dati ed informazioni (Fig. 2.6).

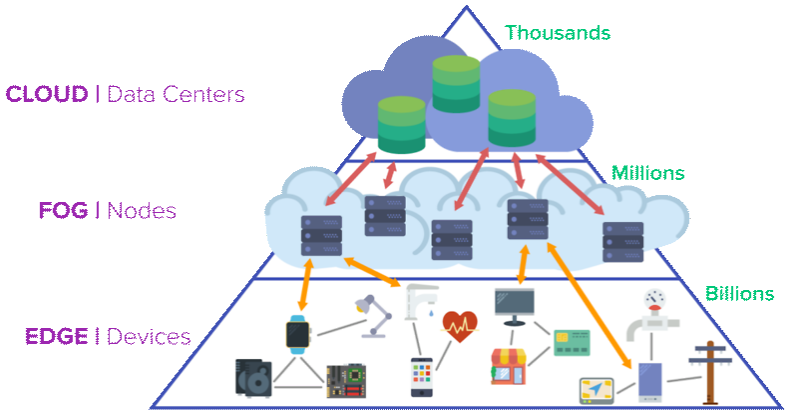

Figura 2.6: Struttura gerarchica dispositivi

Secondo uno studio Cisco [11], si è stimato che i dispositivi attualmente presenti saranno in grado di generare ogni anno una quantità dati locali nell'ordine

di 850 Zettabytes (ZB), a fronte di una capacità globale dei cloud datacenter di circa 20.6 ZB. Ciò ha difatti portato ad una conseguente trasformazione nelle sorgenti dati, passando dai grandi datacenter ad un sempre più alto numero di edge device, ovvero qualsiasi dispositivo capace di generare dati localizzato ai bordi delle reti di telecomunicazione. Al momento l'attuale cloud computing non è in grado né di gestire questa massiva quantità di dati generati né di disporre della quantità necessaria di potenza di calcolo e le cause di tale impossibilità possono essere identificate da:

- il grande numero di dispositivi edge, i quali necessitano di potenza di calcolo dal cloud per processare i dati generati, ponendo indubbiamente una esosa responsabilità e un'importante sfida alla capacità delle infrastrutture di telecomunicazioni e di conseguenza allo stesso cloud,
- le molte delle nuove applicazioni emerse, come ad esempio quelle per la guida autonoma cooperativa, richiedenti stringenti vincoli in termini di ritardo, mettendo a dura prova le possibilità del cloud nel soddisfare tali requisiti.

Queste limitazioni hanno però favorito la nascita di alcune soluzioni, tra cui quella dell'*edge computing*, focalizzata nell'offrire un'opzione capace di eseguire la computazione dei dati il più vicino possibile alla sorgente e all'utente finale. Certamente, l'*edge computing* e il cloud computing non sono mutualmente esclusivi, ma bensì la si può pensare come una l'estensione dell'altra. I principali vantaggi sono tre:

- alleggerimento del carico sulle infrastrutture di telecomunicazioni,
- prontezza nel fornire risposta ai servizi, diminuendo il ritardo implicato trasmissione e migliorandone la velocità di risposta,
- l'eventuale utilizzo delle potenti capacità computazionali e l'enorme spazio di archiviazione offerto del cloud da parte dei dispositivi edge solo se indispensabile.

Dunque grazie a questa collaborazione tra l'edge computing e il cloud computing, si vennero a sviluppare nuove importanti tecnologie. Queste emergenti tecnologie aiutarono l'edge computing ad accelerare il suo sviluppo, portandolo ad ottenere una vasta adozione industriale e non. Allo stesso tempo, grazie al continuo miglioramento di queste nuove tecnologie e alla loro successiva standardizzazione, l'edge computing raggiungerà una maturità tale da permettergli di essere direttamente integrato nelle stesse tecnologie. Tra i numerosi

rami derivanti dall'adozione dell'edge computing, quello che sposa al meglio le aspirazioni di questo progetto di tesi è relativo all'*Edge Intelligence*. In particolare, esso sfrutta l'edge computing al fine di spingere l'integrazione dell'intelligenza artificiale nei dispositivi edge. In aggiunta, attraverso il continuo avanzamento tecnologico e alla maggiore potenza computazionale disponibile, l'adozione dell'AI è in continua espansione, raggiungendo così un'ampia gamma di nuovi scenari di applicazione; per di più, combinandola insieme all'edge computing, i dati generati possono essere più velocemente processati senza la necessità di comunicazione con il cloud, dimezzandone considerevolmente costi di comunicazione e di conseguenza i ritardi di trasferimento. Quindi sfruttando le tecnologie AI, è possibile ottimizzare la gestione e la pianificazione di tutti i dispositivi edge, così come espandere il loro mercato di adozione garantendo all'utente finale servizi sempre più efficienti e solidi. Sulla base di queste considerazioni, è possibile affermare che l'*Edge Intelligence* diventerà una importante tecnologia per la società futura e inoltre permetterà lo sviluppo di numerosi nuovi dispositivi prima impensabili.

2.4.1 Algoritmi di compressione per Deep Neural Network

Nel corso degli anni, le *Deep Neural Network* si sono evolute fino a raggiungere lo stato dell'arte in merito alla *Computer Vision*, ma sebbene risultino così potenti, la loro struttura di molti livelli impiega un quantitativo considerevole di memoria di archiviazione e di banda necessaria al trasferimento dei dati. Per questo motivo, la possibilità di integrare una DNN all'interno di dispositivi edge risulta alquanto remota e per di più i vantaggi introdotti da questi dispositivi, come maggior privacy, minor consumo della banda di rete e l'utilizzo in applicazioni real time, non possono essere sfruttati appieno delle loro potenzialità. In aggiunta, l'utilizzo di DNN performanti in un dispositivo edge comporta un ulteriore inconveniente, ovvero il quantitativo di energia necessario a fine di eseguire inferenza sui nuovi dati; infatti sarebbero necessari grandi quantitativi in termini di banda di memoria e di potenza di calcolo, consumandone di conseguenza una considerevole porzione di energia del dispositivo stesso. Perciò, essendo i dispositivi edge spesso limitati dalla capacità della batteria stessa, l'introduzione di applicazioni energeticamente esigenti come le DNN, risulta al quanto difficile e se non quasi proibitivo. Per poter affrontare queste limitazioni al fine di accrescere l'adozione dell'*edge intelligence*, molti ricercatori hanno iniziato a studiare soluzioni che consentano la drastica riduzione nelle dimensioni delle DNN e di conseguenza del quantitativo di energia necessaria al fine di poterle usare, rendendone difatti possibile la loro integrazione all'in-

terno di tali dispositivi. La soluzione più utilizzata è quella della compressione della DNN, con l'obiettivo di limitare sia la dimensione di memoria necessaria per il suo salvataggio nel dispositivo che l'impatto computazionale necessario, mantenendone comunque l'accuratezza originale.
Esistono numerosi algoritmi che permettono di eseguire la compressione del modello e, durante lo studio dello stato dell'arte, si sono maggiormente approfonditi gli approcci più promettenti, specialmente quelli che usano le potenzialità dell'intelligenza artificiale e del *Machine Learning* per determinare una particolare soluzione ad-hoc.
In [12] gli autori propongono un algoritmo di compressione, chiamato"Deep Compression", che introduce una pipeline di tre stadi consecutivi:

- *pruning*: questo stadio rimuove le connessioni ridondanti mantenendo solamente quelle più importanti per la propagazione delle informazioni attraverso la DNN,
- *quantizzazione*: i pesi della DNN vengono quantizzati affinché connessioni multiple condividano gli stessi valori di peso e possa dunque essere memorizzato una sola volta, accedendovi per mezzo di un determinato puntatore,
- *codifica di Huffman*: codifica i valori dei pesi della rete.

Attraverso questo algoritmo è possibile ottenere un rapporto di compressione della rete originale che varia tra 35x e 49x senza intaccare largamente la sua accuratezza originale, introdurre inoltre una migliore velocità tra gli strati della rete (da 3x a 4x) e una migliore efficienza energetica (da 3x a 7x).
In [13] gli autori propongono una tecnica di *Auto-ML* per la compressione dei modelli (AMC) che sfrutta il *Reinforcement Learning* per realizzare la policy di compressione. Essa, basandosi sull'auto-apprendimento, va a superare di gran lunga qualsiasi altra policy basata su regole predefinite, permettendole di ottenere un più alto livello di compressione, preservandone comunque la sua accuratezza originale. Attraverso questa tecnica, sono riusciti ad ottenere una considerevole riduzione in termini di FLOP pari a 4x e un incremento dell'accuratezza del 2.7%, rispetto ad una policy ad-hoc creata da un esperto per comprimere la stessa rete sul data-set ImageNet.
In [14] gli autori hanno introdotto un nuovo framework che sfrutta, anche in questo caso, le tecniche di *Reinforcement Learning* per determinare automaticamente la politica di quantizzazione, sfruttando direttamente come dato di input il feedback dell'acceleratore hardware, in termini di latenza e di energia. In questo modo, il framework è completamente automatizzato ed è in grado di

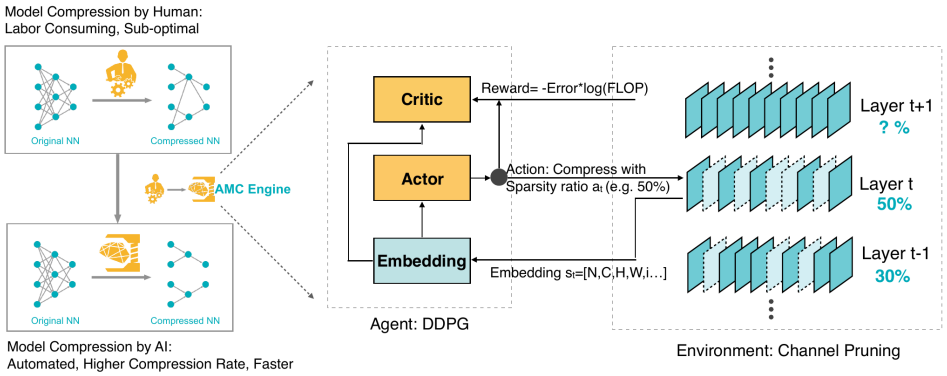

Figura 2.7: Schema generale dell'AMC. Sinistra: AMC sostituisce completamente l'uomo ottenendo un livello di compressione migliore e in modo completamente automatizzato. Destra: strutturazione dell'AMC come un problema di RL. Viene processata una rete pre-allenata lavorando su ogni suo layer. L'agente del RL riceve informazioni riguardante il layer sotto analisi e restituisce in uscita un valore di compressione applicabile. Successivamente viene valutata la precisione del modello dopo la compressione dei layer, fornendo in uscita la ricompensa per l'agente sulla base di quanto prodotto, durante quell'iterazione, e considerando precisione e FLOP ottenuti.

definire una policy di quantizzazione in base al tipo di architettura della rete e dell'hardware, considerando inoltre anche le restrizioni sulle risorse imposte dall'utente stesso; questa tipologia di approccio realizza una compressione che varia da livello a livello nella , chiamata appunto "Mixed Precision". Il framework riduce effettivamente la latenza di 1.4-1.95x e il consumo di energia di 1.9x con una trascurabile perdita di accuratezza rispetto alla quantizzazione statica imposta da una policy fatta ad-hoc.

In [15] gli autori introducono una nuova tecnica di *Neural Architecture Search* (NAS) che può direttamente apprendere l'architettura di reti definite su grandi data-set, senza appoggiarsi su proxy task per l'apprendimento. L'utilizzo di algoritmi NAS permette di ridurre il numero di ore di GPU durante la fase di training, andando a definire un'architettura più efficace per il compito, sebbene introducendo un alto consumo di memoria nella GPU con una crescita lineare rispetto alla dimensione della rete stessa. Per questo motivo, al fine di evitare questa limitazione imposta dalla crescita lineare dell'occupazione di memoria, si tende ad utilizzare i proxy task, come per esempio eseguire il training della rete su una porzione del data-set, o eseguendolo per poche iterazioni, al fine di contenere l'esplosione della memoria occupata; in questo modo si trova però un'architettura di rete non ottimale. Dunque gli autori hanno introdotto ProxylessNAS che sfrutta la grande dimensione del data-

set per allargare lo spazio di ricerca dell'architettura di rete in modo tale da ottenere migliori performance. In aggiunta, per poter gestire i differenti obiettivi imposti dall'hardware utilizzato, essi hanno definito un approccio che sfrutta la discesa del gradiente o il RL. Più precisamente, data una differente piattaforma hardware, ProxylessNAS, permette di consapevolizzare la DNN sulla tipologia di hardware utilizzato e ottimizzandola rispetto alle limitazioni imposte dalla piattaforma hardware stessa.
Lo studio di questi algoritmi di compressione, effettuato durante il periodo di tirocinio, ha permesso di comprendere gli attuali approcci utilizzati, così da avere la possibilità di introdurre queste soluzioni in una successiva iterazione del progetto di tesi. Quello che più ha suscitato interesse è quello proposto da [13], i quali risultati promettono di creare una compressione ottimale per le applicazioni integrabili su dispositivi edge. Inoltre offre la possibilità di poter sfruttare l'algoritmo sia per ridurre le dimensioni del modello, mantenendo quasi inalterata la sua precisione a meno di un piccolo delta di differenza, che per incrementare la velocità di inferenza del modello, riducendone le operazioni a virgola mobile (FLOP) necessarie per la computazione dei dati. A differenza degli altri framework proposti, quello selezionato risulta un buon compromesso tra l'avanguardia delle tecniche utilizzate per la compressione e l'effettiva complessità dell'algoritmo stesso, come è possibile vedere dalla struttura dell'algoritmo in figura 2.7. In particolare in [14], il framework va a sfruttare nel RL i dati provenienti da specifici acceleratori hardware per reti neurali, così da eseguire la quantizzazione della rete in modo eterogeneo sui vari layer che la compongono. Per tale motivo, questo approccio risulterebbe non attuabile rispetto alla tipologia di hardware scelto in fase di progettazione e all'assenza di un acceleratore tensoriale. Invece per quanto riguarda [15] utilizzando un approccio *Differentiable NAS* premette di ottenere degli ottimi risulti sia in termini di riduzione delle rete (*pruning*) che in termini di latenza, anche se il metodo utilizzato per ottenere un'architettura ad-hoc per l'hardware, necessita di una ricerca attraverso algoritmi di NAS su uno spazio molto grande, richiedendo quindi un quantitativo di ore considerevoli per la fase di definizione dell'architettura stessa.

2.5 AutoML: algoritmo per la compressione di modelli

In merito all'algoritmo di compressione introdotto nell'articolo [13], possiamo riassumere in poche parole che AMC sfrutta il *Reinforcement Learning* per

campionare lo spazio di definizione della rete con lo scopo di migliorare la qualità della compressione applicata al modello. Attraverso gli esperimenti, gli autori hanno riscontrato che la precisione del modello compresso è molto sensibile alla sparsità di ogni layer della rete e per questo è necessario utilizzare una policy comprendente azioni ad-hoc definite per l'agente del RL. Per questo motivo, AMC applica continuamente una strategia di controllo del livello di compressione, lasciando l'agente apprendere per tentativi la giusta policy di compressione applicabile ad uno specifico layer, penalizzandone la perdita di precisione nel modello e incoraggiandone invece sia la riduzione del modello che la sua velocizzazione. L'algoritmo introdotto da AMC non è semplicemente limitato a diminuire la dimensione del modello, ma bensì può essere utilizzato per raggiungere degli obiettivi differenti, quali la riduzione dei FLOP o il tempo di inferenza del dispositivo. In aggiunta, la sua strategia di ricerca può essere cambiata andando a variare il sistema di ricompensa dell'agente; infatti, sfruttando quest'ultima opzione, AMC può essere utilizzato anche per determinare il limite massimo di compressione raggiungibile senza perdite in precisione.

	Million MAC	top-1 acc	top-5 acc	GPU latency	GPU speed	Android latency	Android speed	memory
100% MobileNet	569	70.6%	89.5%	0.46 ms	2191 fps	123.3 ms	8.1 fps	20.1 MB
75% MobileNet	325	68.4%	88.2%	0.34 ms	2944 fps	72.3 ms	13.8 fps	14.8 MB
AMC (50% FLOPS)	285	70.5%	89.3%	0.32 ms	3127 fps	68.3 ms	14.6 fps	14.3 MB
AMC (50% latency)	272	70.2%	89.2%	0.30 ms	3350 fps	63.3 ms	16.0 fps	13.2 MB

Tabella 2.2: Impatto AMC sulla MobileNet

Come è possibile vedere nella tabella 2.2, AMC permette di ridurre notevolmente l'impatto computazionale della rete, diminuendone contemporaneamente la latenza. Dato che le MobileNet sono delle reti molto compatte, risulta quindi più difficile comprimerle ulteriormente. Per questo motivo, gli autori hanno applicato l'algoritmo seguendo una riduzione uniforme sui canali di ogni layer, attraverso un valore fisso di compressione, ma comunque ottenendo un ottimo risultato senza ridurre o quasi la sua precisione di inferenza.

Per eseguire l'algoritmo di compressione AMC su una rete, è necessario eseguire 3 passi in successione: la ricerca della strategia, il salvataggio dei pesi ridotti e infine il fine-tuning di quest'ultimi. Lo script 2.1 richiama il file *amc_search.py* per eseguire il RL e ridurre il modello di partenza, contenuto nella *ckpt_path*, ritornando in uscita la strategia opportuna.

```
python amc_search.py \
    --job=train \
    --model=mobilenet \
    --dataset=imagenet \
    --preserve_ratio=0.5 \
    --lbound=0.2 \
    --rbound=1 \
    --reward=acc_reward \
    --data_root=/dataset/imagenet \
    --ckpt_path=./checkpoints/mobilenet_imagenet.pth.tar \
    --seed=2018
```

Listing 2.1: Script per la ricerca dei pesi da ridurre nella rete.

Successivamente viene richiamato lo script 2.2 per esportare il modello ridotto della rete nella *export_path*. In aggiunta, è necessario creare una classe alternativa alla rete che si sta comprimendo, altrimenti non si usufruirebbe delle ottimizzazioni ottenute durante la fase di ricerca; per questo gli autori hanno codificato i modelli relativi alla MobileNet e MobileNetV2.

```
python amc_search.py \
    --job=export \
    --model=mobilenet \
    --dataset=imagenet \
    --data_root=/dataset/imagenet \
    --ckpt_path=./checkpoints/mobilenet_imagenet.pth.tar \
    --seed=2018 \
    --n_calibration_batches=300 \
    --n_worker=32 \
    --channels=3,24,48,96,80,192,200,328,352,368,360,328,400,736,752 \
    --export_path=./checkpoints/mobilenet_0.5flops_export.pth.tar
```

Listing 2.2: Script per salvare il valore dei pesi ridotti.

Infine lo script 2.3 esegue il fine-tuning dei pesi, in modo da migliorare ulteriormente il valore della precisione di inferenza della rete, partendo dal modello esportato nella fase precedente. L'operazione di fine-tuning viene eseguita andando a sfruttare un *learning rate* che segue un andamento di tipo cosinusoidale per un numero totale di epoche pari a 150.

```
python -W ignore amc_fine_tune.py \
    --model=mobilenet_0.5flops \
    --dataset=imagenet \
    --lr=0.05 \
    --n_gpu=4 \
    --batch_size=256 \
    --n_worker=32 \
    --lr_type=cos \
    --n_epoch=150 \
    --wd=4e-5 \
```

```
    --seed=2018 \
    --data_root=/dataset/imagenet \
    --ckpt_path=./checkpoints/mobilenet_0.5flops_export.pth.tar
```

Listing 2.3: Script per eseguire il fine-tuning della rete

Infine è necessario puntualizzare le problematiche incontrate dall'analisi del codice relativo all'algoritmo di compressione AMC. In particolare, il codice realizzato dagli autori di [13] utilizza una piattaforma diversa da quella utilizzata nel progetto; dunque sarà necessario esportare il modello allenato in un formato compatibile con AMC oppure integrare del codice custom per la corretta estrapolazione della rete. Infine, AMC richiede la definizione di una classe custom della rete che si vuole comprimere per poter usufruire dei risultati ottenuti durante la ricerca.

Capitolo 3

Definizione architettura della rete

3.1 Descrizione del progetto e delle tecnologie

Il progetto di tesi mira ad utilizzare la tecnologia offerta dal *Deep Learning* al fine di implementare una CNN per la *semantic segmentation* in applicazioni di guida autonoma su piattaforma embedded. Dopo una prima analisi della letteratura scientifica descritta nel capitolo 2, è essenziale approfondire ulteriormente le scelte architetturali necessarie, al fine di incontrare le specifiche relative alla piattaforma hardware selezionata. Date le importanti dimensioni in termini computazionali e specialmente di memoria richieste da una rete CNN per la segmentazione semantica, si è scelto di utilizzare la *Raspberry Pi* come dispositivo embedded di testing. Data la generosa capacità hardware offerta dalla piattaforma, si è deciso di utilizzare come architettura base per la realizzazione della CNN, la struttura chiamata *U-Net*. Essa è composta da due rami, il primo va a ridurre la dimensione dell'input in modo da catturare il contesto a cui si riferisce, mentre il secondo va ad espandere l'uscita dal precedente ramo in modo da localizzare correttamente i dettagli; così in uscita si ottiene una migliore contestualizzazione dell'immagine e una maggiore precisione nell'inferenza stessa. In aggiunta alla classica implementazione, la struttura base della *U-Net* è stata modificata in modo da sostituire gli strati convolutivi a favore di quelli definiti dagli autori della MobileNet [10]; ciò ha lo scopo di alleggerire le dimensioni e la potenza di calcolo richieste dalla rete. Questa fusione tra le due architetture è stata definita dagli autori dell'articolo [8]. Dunque, una volta individuati tutti gli ingredienti necessari alla realizzazione della rete CNN in questione, si è scelto come ambiente di sviluppo Python, dove è nota la presenza di numerosi framework e librerie volte alla definizione delle reti di deep learning. Inoltre, data la disponibilità del codice sorgente della rete selezionata, si è scelto di compiere un porting della versione già realizzata dagli autori di [8] passando da *PyTorch* alle librerie di *Tensorflow* e *Keras*.

3.1.1 MobileNet

Come descritto in [10], la MobileNet è una rete di deep learning, con la quale gli autori si sono posti come obiettivo quello di sviluppare la migliore architettura di *Machine Vision* per dispositivi mobili, scegliendo di ottimizzare al meglio il compromesso tra precisione e latenza della rete. La MobileNet è basata su una architettura snella che utilizza la *depthwise separable convolution*, cioè una forma di fattorizzazione della convoluzione stessa. Quest'ultima viene infatti suddivisa in due convoluzioni in sequenza: la *depthwise convolution* e la *pointwise convolution*. La prima fattorizzazione mira a filtrare uno a uno i canali in ingresso alla rete e il suo output viene poi processato dalla seconda per ricombinare i canali appena separati insieme. Sfruttando questa separazione, è possibile ottenere una drastica riduzione sia in termini di computazione che di dimensione del modello stesso. In aggiunta, gli autori hanno introdotto due nuovi *hyper-parameter* che svolgono il compito di bilanciare efficientemente la latenza e la precisione del modello, lasciando all'utente la libertà di realizzare una rete che rispetti i vincoli imposti dal problema. Il primo *hyper-parameter* viene chiamato *width multiplier* e ha lo scopo di assottigliare uniformemente la rete in ogni strato; il secondo, invece, è il *resolution multiplier* che permette invece di ridurre il costo computazionale della rete.
Una prima evoluzione della MobileNet fu introdotta con la sua seconda versione (v2) [16], la quale andò ulteriormente a raffinare la *depthwise separable convolution* in modo da aumentarne la sua efficienza in termini di computazione, fino ad ottenere una riduzione di circa 8-9 volte rispetto alla convoluzione originale, a fronte però di una piccola riduzione nella precisione. Un ulteriore miglioramento riguarda l'introduzione di *bottleneck layer* nell'architettura, i quali consentono di prevenire che le non-linearità introdotte dalla rete possano fortemente distruggere le informazioni propagate nella rete. Un'altra importante aggiunta riguarda l'introduzione di scorciatoie tra i *bottleneck layer*, simili a quelli dei *residual block*; vengono definiti *inverted residual block* e il loro scopo è quello di migliorare la propagazione del gradiente attraverso gli strati moltiplicatori. Vengono definiti "inverted" perché a differenza dei classici residui, essi collegano gli strati di *bottleneck* come visibile in figura 3.1.
Infine, nella terza iterazione (v3) della MobileNet oltre all'utilizzo di algoritmi di ricerca (NAS) per ottimizzare la struttura della rete, vennero introdotte tre ulteriori migliorie: la prima fu l'utilizzo dello *squeeze and excitation block*, che ha l'obiettivo di potenziare la qualità rappresentativa della rete, andando a modellare esplicitamente le dipendenze presenti tra i canali relativi ad ogni proprietà della convoluzione; in pratica apprende come sfruttare informazioni di livello globale per enfatizzare quelle proprietà più rappresentative rispetto alle

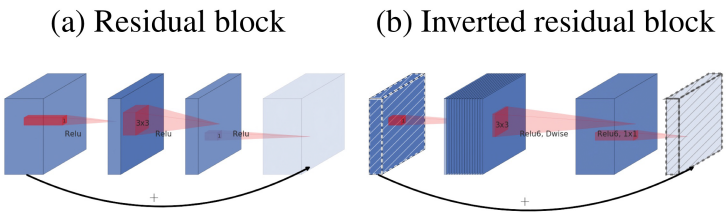

Figura 3.1: Differenza tra residual block e inverted residual block

altre. La seconda miglioria riguarda invece l'introduzione della non-linearità *swish* [17], che offre una migliore efficienza e compatibilità con le reti rivolte ai dispositivi mobili rispetto alla classica *ReLU* [18]. La terza ed ultima miglioria è quella relativa alla sostituzione della funzione di attivazione a sigmoide con la *hard sigmoid* [19]. Gli autori hanno realizzato due differenti configurazioni, la *MobileNetV3-Large* e *MobileNetV3-Small* presentate nella tabella 3.1. Come già detto all'inizio di questa sezione, gli autori hanno realizzato questa architettura al fine di abbattere l'utilizzo di risorse computazionali e di memoria, così da rendere possibile il loro utilizzo su dispositivi edge. Difatti entrambe le soluzioni sono in grado di ottenere un buon compromesso tra velocità di inferenza e precisione, dove in particolare la configurazione "*small*" ottiene il miglior risultato in termini di latenza, riducendo di quasi 3 volte il tempo di inferenza a discapito di una minor precisione rispetto alla versione "*large*".

Input	Operator	exp size	#out	SE	NL	s
224^2x3	conv2d	-	16	-	HS	2
112^2x16	bneck, 3x3	16	16	-	RE	1
112^2x16	bneck, 3x3	64	24	-	RE	2
56^2x24	bneck, 3x3	72	24	-	RE	1
56^2x24	bneck, 5x5	72	40	✓	RE	2
28^2x40	bneck, 5x5	120	40	✓	RE	1
28^2x40	bneck, 5x5	120	40	✓	RE	1
28^2x40	bneck, 3x3	240	80	-	HS	2
14^2x80	bneck, 3x3	200	80	-	HS	1
14^2x80	bneck, 3x3	184	80	-	HS	1
14^2x80	bneck, 3x3	184	80	-	HS	1
14^2x80	bneck, 3x3	480	112	✓	HS	1
14^2x112	bneck, 3x3	672	112	✓	HS	1
14^2x112	bneck, 5x5	672	160	✓	HS	2
7^2x160	bneck, 5x5	960	160	✓	HS	1
7^2x160	bneck, 5x5	960	160	✓	HS	1
7^2x160	conv2d, 1x1	-	960	-	HS	1
7^2x960	pool, 7x7	-	-	-		1
1^2x960	conv2d 1x1, NBN	-	1280	-	HS	1
1^2x1280	conv2d 1x1, NBN	-	k	-		1

Input	Operator	exp size	#out	SE	NL	s
224^2x3	conv2d	-	16	-	HS	2
112^2x16	bneck, 3x3	16	16	✓	RE	2
56^2x16	bneck, 3x3	72	24	-	RE	2
28^2x24	bneck, 5x5	88	24	-	HS	1
28^2x24	bneck, 5x5	96	40	✓	HS	2
14^2x40	bneck, 3x3	240	40	✓	HS	1
14^2x40	bneck, 3x3	240	40	✓	HS	1
14^2x40	bneck, 3x3	120	48	✓	HS	1
14^2x48	bneck, 3x3	144	48	✓	HS	1
14^2x48	bneck, 5x5	288	96	✓	HS	2
7^2x96	bneck, 5x5	576	96	✓	HS	1
7^2x96	bneck, 5x5	576	96	✓	HS	1
7^2x96	conv2d, 1x1	-	576	✓	HS	1
7^2x576	pool, 7x7	-	-	-		1
1^2x576	conv2d 1x1, NBN	-	1024	-	HS	1
1^2x1024	conv2d 1x1, NBN	-	k	-		1

Tabella 3.1: Specifiche MobileNetV3-Large e MobileNetV3-Small: SE *Squeeze-And-Excite*, NL la non-linearità usata tra HS *hard-swish* e RE *ReLU*, NBN no batch normalization e *s* lo stride.

3.1.2 U-Net

Nell'articolo [20] viene presentata una nuova architettura di DNN e una strategia di allenamento che si affida fortemente sull'utilizzo di *data augumentation* (tecnica per l'aumento dei dati) con l'obiettivo di sfruttare più efficientemente i dati disponibili. Come già accennato all'inizio del capitolo, l'architettura è composta da due rami: il primo va a ridurre la dimensione dell'input in modo da catturare il contesto descritto, mentre il secondo va ad espandere l'uscita del precedente ramo in modo da localizzare correttamente i dettagli racchiusi. La DNN introdotta dagli autori viene definita come un'implementazione più elegante dell'architettura *fully convolutional network* dell'articolo [3]. Infatti la struttura è stata modificata ed estesa cosicché può operare sfruttando poche immagini di allenamento, ma rendendo comunque più precisa la loro segmentazione. L'architettura è rappresentata nella figura 3.2.

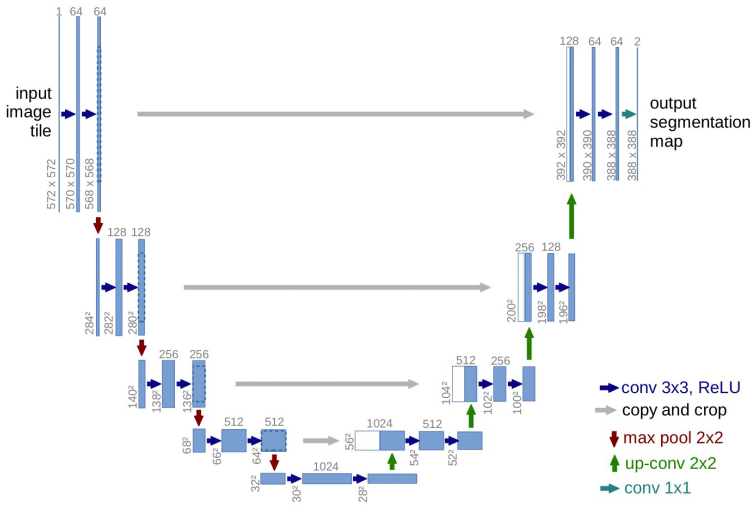

Figura 3.2: Architettura della U-net. Le box blu corrispondono alle feature map multicanale e il numero dei canali è indicato sotto ogni box, mentre la dimensione x-y è descritta in basso a sinistra. Le box bianche rappresentano le copie delle feature map. Le frecce, invece, rappresentano le differenti operazioni, specificate nella legenda.

L'idea principale in [3] è quella di integrare ad una normale *contracting network* (rete che riduce la dimensione dell'input) degli strati (layer) consecutivi, sostituendo inoltre tutti gli operatori di pooling con quelli di upsampling; in questo modo i layer aggiuntivi permettono alla rete di ottenere una risoluzione maggiore nell'output. In aggiunta, concatenando l'uscita dei contracting layer con quella dei upsampling layer e successivamente passandola attraverso un layer convolutivo, è possibile per la rete apprendere come localizzare alcuni

importanti dettagli al fine di migliorare la qualità dell'output stesso. Ora, una importante modifica introdotta nell'architettura *U-Net* è che la parte rivolta all'upsampling ha un grande numero di canali per la propagazione di informazioni sul contesto ai layer che operano con input con maggior risoluzione. Infatti, seguendo questo approccio, la parte relativa all'espansione risulta essere pressoché simmetrica con la parte relativa alla riduzione, creando un'architettura a forma di *u*. La rete non ha nessun layer completamente connesso, ma utilizza solo convoluzioni e in particolare solo la parte valida della convoluzione; in altre parole la mappa creata attraverso la segmentazione dell'immagine contiene solo quei pixel per i quali il contesto completo è disponibile. Questa strategia permette una segmentazione di immagini con grandi dimensione sfruttando un approccio chiamato *overlap-tile*, visibile in figura 3.3. Infine per predire la mancanza del contesto di quei pixel localizzati ai bordi dell'immagine viene specchiata l'immagine in input. Questa strategia è di fondamentale importanza quando vengono utilizzate immagini con grande risoluzione, altrimenti la memoria disponibile nella GPU limiterebbe le risoluzioni usufruibili. Come già

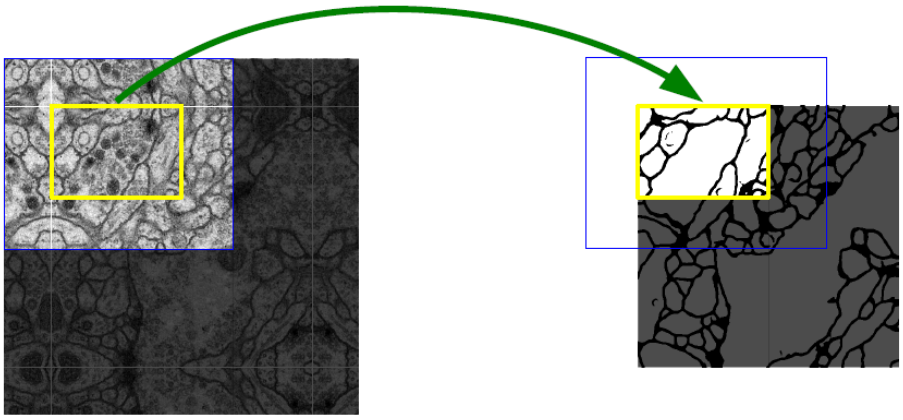

Figura 3.3: La strategia *overlap-tile*. La predizione della segmentazione riguardante l'area delimitata in giallo, richiede dati di input relativi all'area delimitata in blu. Nel caso tali dati mancassero, essi vengono estrapolati specchiando l'immagine in input.

prima accennato, gli autori, disponendo di poche immagini campione per la fase di allenamento della rete, hanno fatto uso di uno smodato numero di dati ottenuti tramite tecniche di *data augmentation*. Applicando una deformazione alle immagini, la rete ha la possibilità di apprendere un'invarianza a tali deformazioni, senza la necessità di vederle nelle immagini già annotate.

3.1.3 EfficientSeg

L'architettura utilizza nella definizione della CNN per la segmentazione semantica è quella relativa alla *EfficientSeg*, presentata nell'articolo [8]. Tipicamente gli approcci utilizzati nel *machine learning* e specialmente nel *deep learning* traggono forza dall'utilizzo di un grande numero di esempi già annotati. Purtroppo affidarsi a grandi training set restringe il panorama di applicazione delle soluzioni basate su *deep learning*, visto che molti problemi non dispongono di grandi quantità di dati annotati. Per questo motivo, generalmente si fa utilizzo di reti pre-allenate su grandi dataset, come *ImageNet*, per soppiantare alla mancanza di dati e inoltre permette di velocizzare l'allenamento. Comunque, per tutte quelle reti definite su domini differenti dal contesto di questi grandi dataset, come ad esempio la segmentazione semantica applicata alla guida autonoma, il pre-allenamento non risulta affatto significativo. Per questo motivo, avere la possibilità di apprendere da zero utilizzando un ristretto numero di campioni, rappresenta un importante obiettivo, tanto da arrivare alla definizione di un nuovo campo chiamato *data-efficient deep learning*.

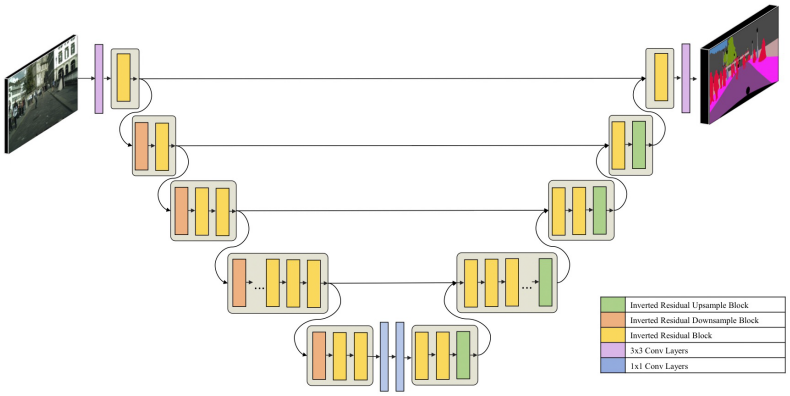

Figura 3.4: Architettura della EfficientSeg. Sono presenti 5 tipologie di blocchi: *Inverted Residual Block* definiti dagli autori della MobileNetV3, *1x1* e *3x3* sono normali blocchi di convoluzione che hanno una funzione di attivazione e un layer di batch normalization, i blocchi di *downsample* sono realizzati con layer convolutivi con stride maggiore di 1 e infine *upsampling* attraverso interpolazione bilineare.

Nella segmentazione semantica, è risaputo che definire architetture di rete con molti layer o con backbone profonde, influenzano positivamente i risultati; d'altro canto, però, reti molto profonde portano anche inconvenienti. Idealmente, supponendo di prendere una rete come riferimento e sottoporla ad un processo di ridimensionamento in modo da renderla compatibile con parecchi ambiti,

necessita la presenza di efficienza sia in termini di memoria che temporale. Quest'ultima, in particolare, è molto rilevante, dato che si riferisce al fatto che incrementando la dimensione della rete, di conseguenza aumenterà anche il tempo necessario all'allenamento. Per questi motivi, hanno deciso di avvalersi dei blocchi definiti nell'articolo della MobileNetV3 [7] al fine di definire un modello base ad ogni modo performante e profondo, ma con un considerevole inferiore numero di parametri. In questo modo, gli autori hanno realizzato la *EfficientSeg*, un'architettura fondata sulla *U-net*, avvalendosi dell'utilizzo di *inverted residual block* definiti dalla MobileNetv3 (figura 3.4).
La *U-Net* è composta da un *encoder* e da un *decoder*; nella *EfficientSeg*, il primo, che si occupa della fase di downsampling dell'input, è strutturato come una *MobileNetV3-Large* senza i layer finali che si occupano della classificazione, mentre il secondo è la versione simmetrica dell'encoder dove l'operazione di downsampling è sostituita dall'upsampling. In particolare, nel decoder viene utilizzato il ricampionamento dell'input in modo tale da ritrovare le stesse dimensioni dell'immagine in ingresso; ciò viene ottenuto attraverso una interpolazione bilineare con una fattore di scala pari a 2 applicato ad ogni layer simmetrico a quello di downsample. In aggiunta, si hanno 4 connessioni scorciatoia tra l'encoder e il decoder e ognuna è realizzata attraverso la concatenazione dell'input proveniente dai due layer collegati. In questo modo, viene reso possibile alla rete di catturare quei minimi dettagli che altrimenti verrebbero persi tra i vari layer. Infine, come nella MobileNetV3, è stato inserito un parametro per scalare la dimensione della rete, rendendolo adatto per la creazione di versioni personalizzate capaci di soddisfare i vincoli del problema.

3.2 Architettura e implementazione

Dopo aver introdotto tutti gli strumenti necessari alla realizzazione della CNN per la segmentazione semantica, verrà ora finalmente presentata l'architettura implementata. Il linguaggio di programmazione utilizzato per la realizzazione della rete è *Python* e come ausilio alla definizione dei vari layer, *Tensorflow* e *Keras*; TensorFlow è una piattaforma open-source per il *machine learning* e comprende un ecosistema completo e flessibile di strumenti, librerie e risorse progettate e sviluppate dalla comunità scientifica. Ciò consente ai ricercatori di continuare ad evolvere l'attuale stato dell'arte nel ML, offrendo inoltre la possibilità di creare e distribuire facilmente applicazioni basate sul ML. Keras invece è un API recentemente inglobata dall'ecosistema Tensorflow e permette di estendere le funzionalità base della piattaforma andando a coprire ogni step

della definizione di soluzioni per *machine learning* e *deep learning*, incluse quelle relative al data management e agli hyper-parameter della fase di training.

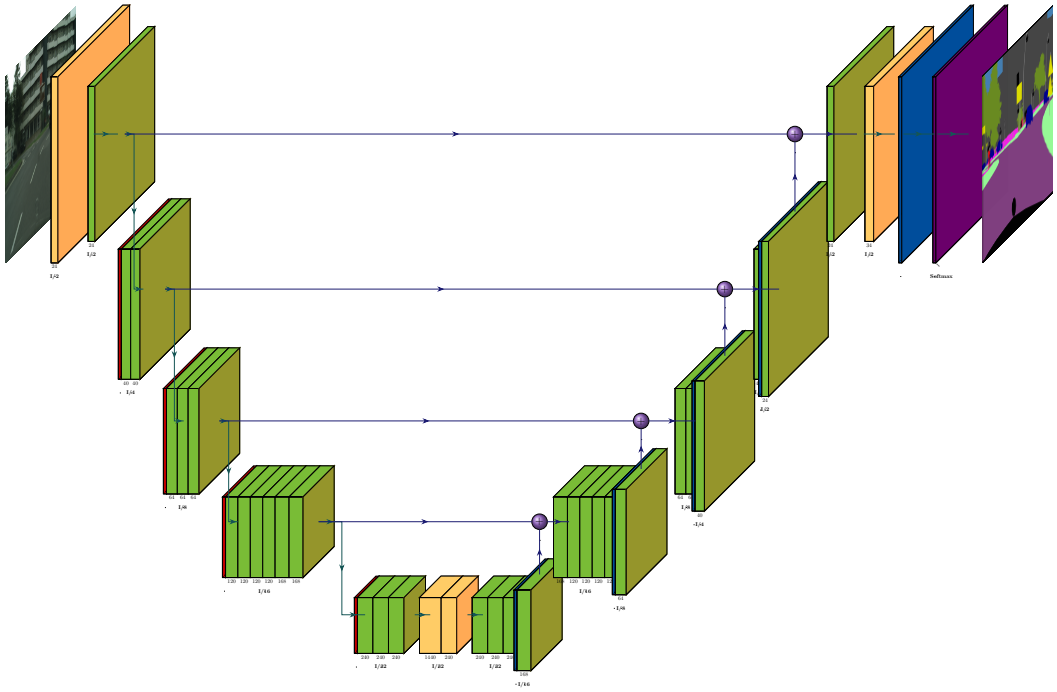

Figura 3.5: Architettura della rete implementata

L'architettura di rete è strutturata come in figura 3.5 dove in ingresso viene fornita un'immagine e la sua segmentazione semantica, necessaria per poter iniziare il training supervisionato. Con supervisionato si intende una rete al cui ingresso viene fornito oltre all'immagine del training set anche il corretto output realizzato dai creatori dei dataset; ciò permette, alla fine di ogni iterazione, di effettuare la comparazione tra quello fornito in ingresso e quello generato dalla rete, propagando all'indietro l'eventuale errore. Con la diffusione a ritroso dell'errore, i pesi calcolati durante la fase di training vengono alterati in accordo con l'errore riscontrato e tale che nella successiva iterazione la rete sia in grado di migliorare la sua performance nel conseguire, come ad esempio in questo caso, la segmentazione semantica di un'immagine. Nel capitolo Risultati e discussioni verranno discussi tutti i dettagli riguardanti il dataset utilizzato, la creazione delle strutture, funzioni utilizzate per fornire i dati alla rete.
Il primo layer incontrato è di tipo convolutivo e il suo compito è quello di andare ad effettuare la prima *feature exctraction* dall'immagine in ingresso ed di eseguire un primo taglio dell'input dimezzandone la sua dimensione. L'implementazione è riportata nel listato 3.1 e come si può vedere è composto da un layer convolutivo seguito da una *Batch Normalization*, utilizzata per normalizzare l'input, e infine dalla non-linearità.

```
# Definizione della hard-sigmoid, utilizzata negli InvertedResidualBottleneck layer
def h_sigmoid(input_tensor):
    return tf.nn.relu6(input_tensor + 3) / 6

# Definizione della hard-swish, utilizzata negli InvertedResidualBottleneck layer
def h_swish(input_tensor):
    return input_tensor*h_sigmoid(input_tensor)

# Layer convolutivo utilizzato all'ingresso e all'uscita della rete
def conv_3x3_bn(input_tensor, oup, stride):
    conv = tf.keras.layers.Conv2D(filters = oup, kernel_size = 3, padding='same', strides =
    ↪  stride, use_bias=False)(input_tensor)
    conv = tf.keras.layers.BatchNormalization()(conv)
    conv = h_swish(conv)
    return conv
```

Listing 3.1: Layer convolutivo in ingresso

La non-linearità utilizzata dalla rete è definita qui sopra ed è per lo più utilizzata negli *Inverted Residual Bottleneck* layer. Più precisamente, sono state specificate due non-linearità [17, 21, 22] in sostituzione alla classica *ReLU*, con lo scopo di migliorare significativamente la precisione della rete neurale. La funzione non-lineare *swish* è definita come:

$$swish\, x = x \cdot \sigma(x)$$

Anche se permette di migliorare la precisione, questa non-linearità ha un impatto non indifferente nella computazione, risultando esosa se utilizzata in dispositivi mobili. Per affrontare questo problema, gli autori dell'articolo [7] hanno pensato di utilizzare due precisi accorgimenti:

1. sostituire la funzione *sigmoid* (σ) con una sua implementazione lineare a tratti, ottenendo di conseguenza la funzione lineare a tratti *h-swish*:

$$h\text{-}swish[x] = x\frac{\text{ReLU6}(x+3)}{6}$$

   La scelta di modificare la non-linearità *swish* è principalmente dovuta ai vantaggi introdotti in fase di deployment della rete: il primo riguarda la vasta adozione della ReLU6 nei maggior framework software e hardware, mentre il secondo è che permette di eliminare delle potenziali perdite in precisione numerica, riscontrate specialmente nelle reti quantizzate, causate dalle diverse implementazioni della sigmoide ed infine riduce il numero di accessi alla memoria e di conseguenza la latenza complessiva;

2. il costo nell'applicare una non-linearità va via via decrescendo con la profondità della rete, dato che in ogni layer in cui avviene un taglio nella

risoluzione, anche la memoria utilizzata viene tipicamente dimezzata. Difatti gli autori hanno usufruito di questa non-linearità solo nei layer più profondi della rete.

```
def InvertedResidualBottleneck(inp, input_channel, hidden_dim, oup, kernel_size, use_se,
↪  use_hs, stride):
    assert stride in [1, 2]
    identity = stride == 1 and input_channel == oup

    if input_channel == hidden_dim:
        invertedResidualBtk = tf.keras.layers.ZeroPadding2D(padding=(kernel_size - 1) //
        ↪  2)(inp)
        invertedResidualBtk = tf.keras.layers.DepthwiseConv2D(kernel_size, stride,
        ↪  padding='valid')(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.BatchNormalization()(invertedResidualBtk)
        if use_hs:
            invertedResidualBtk = h_swish(invertedResidualBtk)
        else:
            invertedResidualBtk = tf.nn.relu6(invertedResidualBtk)
        if use_se:
            invertedResidualBtk = Squeeze_excitation_layer(invertedResidualBtk, hidden_dim,
            ↪  4)
        invertedResidualBtk = tf.keras.layers.Conv2D(oup, 1, 1, padding="valid",
        ↪  use_bias=False)(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.BatchNormalization()(invertedResidualBtk)
    else:
        invertedResidualBtk = tf.keras.layers.Conv2D(hidden_dim, 1, 1, padding="valid",
        ↪  use_bias=False)(inp)
        invertedResidualBtk = tf.keras.layers.BatchNormalization()(invertedResidualBtk)
        if use_hs:
            invertedResidualBtk = h_swish(invertedResidualBtk)
        else:
            invertedResidualBtk = tf.nn.relu6(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.ZeroPadding2D(padding=(kernel_size - 1) //
        ↪  2)(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.DepthwiseConv2D(kernel_size, stride,
        ↪  padding='valid')(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.BatchNormalization()(invertedResidualBtk)
        if use_se:
            invertedResidualBtk = Squeeze_excitation_layer(invertedResidualBtk, hidden_dim,
            ↪  4)
        if use_hs:
            invertedResidualBtk = h_swish(invertedResidualBtk)
        else:
            invertedResidualBtk = tf.nn.relu6(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.Conv2D(oup, 1, 1, padding="valid",
        ↪  use_bias=False)(invertedResidualBtk)
        invertedResidualBtk = tf.keras.layers.BatchNormalization()(invertedResidualBtk)

    if identity:
        return inp + invertedResidualBtk

    return invertedResidualBtk
```

Listing 3.2: Definizione Inverted Residual Bottleneck block

Successivamente al layer iniziale, vengono invece inseriti un susseguirsi di *Inverted Residual Bottleneck* layer, i quali andranno a ripetere periodicamente un taglio nelle dimensioni dell'input, così da ampliare la sensibilità verso i piccoli dettagli. L'implementazione è visibile nel dettaglio nel listato 3.2. Rispetto all'implementazione originale, è stato necessario modificare due dei layer convolutivi visto l'utilizzo di una particolare impostazione, chiamata *groups* capace di trasformare la convoluzione in una convoluzione di gruppo e con lo scopo di limitare il numero complessivo di parametri generati; in particolare essa applica un insieme di kernel indipendenti ad un gruppo di canali. Dato che questa operazione non è supportata nella versione *TensorFlow Lite* utilizzata principalmente nei dispositivi embedded o edge, essa è stata sostituita con la convoluzione *Depthwise*. L'unico neo di questo approccio riguarda il maggior utilizzo della memoria per l'allocazione dei tensori.

```
def Squeeze_excitation_layer(input_tensor, out_dim, ratio):
    squeeze_excitation = tf.keras.layers.GlobalAveragePooling2D()(input_tensor)
    squeeze_excitation = tf.keras.layers.Dense(units=out_dim // ratio)(squeeze_excitation)
    squeeze_excitation = tf.nn.relu6(squeeze_excitation)
    squeeze_excitation = tf.keras.layers.Dense(units=out_dim)(squeeze_excitation)
    squeeze_excitation = h_sigmoid(squeeze_excitation)
    squeeze_excitation = tf.reshape(squeeze_excitation, [-1,1,1,out_dim])
    scale = input_tensor*squeeze_excitation

    return scale
```

Listing 3.3: Squeeze e Excitation layer

Ultimo ma non meno importante layer utilizzato dal modello della rete è quello riportato nel listato 3.3 e chiamato *Squeeze and Excitation*. Il suo scopo è quello di andare a migliorare la qualità delle rappresentazioni prodotte dalla rete, agendo esplicitamente sulle interdipendenze tra i canali degli strati convolutivi. Infatti il meccanismo adottato permette alla rete di eseguire una ricalibrazione delle feature, grazie alla quale si può migliorare l'apprendimento di quelle informazioni di livello globale, utili nell'enfatizzare le caratteristiche più rappresentative.

Capitolo 4

Risultati e discussioni

Nel seguente capitolo si presenteranno i risultati sperimentali ottenuti allenando la rete neurale convoluzionale sul dataset *Cityscapes*. L'analisi sperimentale sulle prestazioni è stata realizzata con l'ausilio di tre diverse configurazioni basate sull'utilizzo della rete *EfficientSeg* [8] e ciascuna di esse verrà introdotta da una breve panoramica sui dati utilizzati durante il training. Inoltre verranno mostrati degli esempi di segmentazione semantica ottenuti sul test-set, le metriche di performance considerate nel training del modello quantizzato.

4.1 Cityscapes Dataset

Questa sezione sarà dedicata all'introduzione del dataset selezionato motivandone la scelta, e alle tecniche di *data augmentation* impiegate per accrescere le possibilità di generalizzazione del modello. *Cityscapes* è uno dei principali dataset nel training di reti neurali volte a comprendere gli scenari incontrati lungo le strade urbane; infatti esso si focalizza principalmente nell'aiutare le reti neurali a percepire la semantica dietro a tali scene, come ad esempio aiutare un software di guida autonoma a comprendere l'ambiente circostante percepito dai vari sensori ottici utilizzati. Gli autori di *Cityscapes* [23] hanno deciso di progettare questo grande dataset con lo scopo di focalizzarsi sulla conoscenza semantica degli ambienti urbani. Esso si basa sull'utilizzo di immagini annotate, le quali richiedono un processo di realizzazione dispendioso e tedioso e difatti la scelta di tale approccio è sempre stata ritenuta di scarso interesse. D'altro canto, i vantaggi introdotti da questo metodo sono superiori sia in termini di informazioni racchiuse nelle immagini che nelle possibilità di future estensioni del dataset. Uno dei punti di forza del dataset è la presenza di una grande varietà di immagini annotate che permettono di catturare molteplici scene urbane. Infatti i dati sono stati raccolti in circa 50 diverse città così da ridurre la somiglianza tra le scene, limitandone l'over-fitting; in aggiunta le immagini sono state raccolte nel corso dell'anno così da catturare le scene in diverse

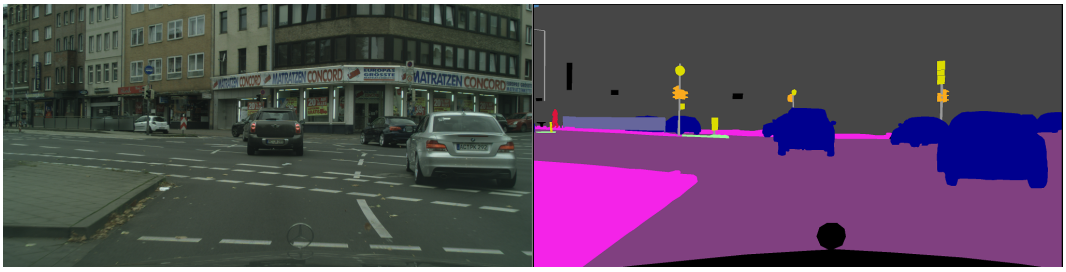

Figura 4.1: Esempio dati contenuti nel dataset.

stagioni. Infine per raggiungere ancora una più alta difformità tra le scene, le immagini sono state selezionate scartando quelle vuote e favorendo quelle con un maggior numero di elementi in scena. Un esempio della tipologia di immagini contenute nel dataset è visibile nella figura 4.1
Cityscapes è un grande dataset che contiene circa 5000 immagini annotate e 20000 con parziale annotazione e per il training del modello si è utilizzato il blocco immagini *LeftImg8bit*; le immagini contenute in esso corrispondono ai soli dati provenienti dalla telecamera di sinistra. In aggiunta il blocco è suddiviso in tre parti:

- *train*, utilizzato di solito per il training, contiene 2975 immagini con annotazioni complete e parziali,
- *validation*, utilizzato per la validazione degli hyper-parameter del modello, contiene 500 immagini con annotazioni complete e parziali,
- *test*, utilizzato per testare le performance del modello sul server offerto dai gestori del dataset; infatti le annotazioni non sono pubbliche.

Proprio per la mancanza di un test set per valutare le prestazioni della rete CNN, si è scelto di suddividere il training set in due parti, estrapolandone 500 immagini annotate per la fase di testing.
Il dataset utilizza 34 classi riassunte nella tabella 4.1. Ogni classe è identificata da un'etichetta e da un identificativo, utilizzati alternativamente dalla rete per classificare i pixel; una volta che la rete ha generato la segmentazione semantica dell'immagine, l'output viene passato ad una funzione che esegue la colorazione dell'immagine con i colori imposti dal dataset, così da distinguere ogni classe. Una buona rete neurale è contraddistinta dalla capacità di generalizzare, la quale è garantita dall'allenamento su un dataset sufficientemente ampio e variegato. Qualora invece il training sia condotto su un insieme limitato, la rete memorizza le caratteristiche specifiche dei campioni (come la disposizione delle classi nelle immagini, se ve n'è una prevalente) e non è in grado di generalizzare: è il caso

Gruppo	**Classi**
flat	road, sidewalk, parking, rail track
human	person, rider
vehicle	car, truck, bus, on rails, motorcycle, bicycle, caravan, trailer
construction	building, wall, fence, guard rail, bridge, tunnel
object	pole, pole group, traffic sign, traffic light
nature	vegetation, terrain
sky	sky
void	unlabeled, ego vehicle, rectification border, out of roi, ground, dynamic, static

Tabella 4.1: Classi annotate nel dataset Cityscapes

dell'*overfitting* (sovradattamento). Dunque per aiutare il modello a raggiungere una migliore generalizzazione ed aumentare il numero di immagini disponibili nella fase di training, si è deciso di utilizzare tecniche di *data augmentation*. Questi metodi consentono di manipolare e modificare parzialmente le immagini presenti nel dataset, così da creare dei nuovi dati sintetici derivanti da quelli veri. Come mostrato nel listato 4.1, le tecniche utilizzate per generare data augmentation sono principalmente di due tipi:

- *geometriche*, cioè l'immagine in ingresso viene modificata nella sua forma e dimensioni attraverso rotazioni, ritagli dell'immagine e ribaltamenti orizzontali,

- *cromatiche*, dove l'immagine subisce delle variazioni in termini di luminosità, contrasto, saturazione dei colori e tonalità.

In questo modo è possibile alterare le immagini in modo da aumentare la conoscenza della rete neurale e dandole la possibilità di potersi adattare a situazioni particolari incontrate in ambito urbano. Però è necessario ricordare che le tecniche di data augmentation devono essere applicate con attenzione e considerando soprattutto il caso in esame. Ad esempio, per un problema di segmentazione di immagini urbane per navigazione autonoma di veicoli, variazioni di contrasto, saturazione, luminosità delle immagini irrobustiscono il modello rendendolo resistente a differenti condizioni di illuminazione, come possono essere la presenza di un raggio di luce diretto verso il sensore della telecamera, riflesso dovuto ad un fondo bagnato o innevato.

```
def data_augmentation(img, label):
    colorjitter_factor = 0.05
    bf = np.random.uniform(1-colorjitter_factor,1+colorjitter_factor)
    cf = np.random.uniform(1-colorjitter_factor,1+colorjitter_factor)
    sf = np.random.uniform(1-colorjitter_factor,1+colorjitter_factor)
    hf = np.random.uniform(1-colorjitter_factor,1+colorjitter_factor)
    rotate_angle = random.uniform(0,15)

    img = read_png(img)
    label = read_png_label(label)

    if tf.random.uniform(()) > 0.5:
        img, label = rand_crop(img, label)
    elif tf.random.uniform(()) > 0.5:
        img = tf.image.flip_left_right(img)
        label = tf.image.flip_left_right(label)
    elif tf.random.uniform(()) > 0.5:
        img = tfa.image.rotate(img, tf.constant(rotate_angle))
        label = tfa.image.rotate(label, tf.constant(rotate_angle))
    elif tf.random.uniform(()) > 0.7:
        img = tf.image.adjust_brightness(img, bf)
    elif tf.random.uniform(()) > 0.7:
        img = tf.image.adjust_contrast(img, cf)
    elif tf.random.uniform(()) > 0.6:
        img = tf.image.adjust_saturation(img, sf)
    elif tf.random.uniform(()) > 0.9:
        img = tf.image.adjust_hue(img, hf)

    img = tf.image.resize(img, [384, 768])
    label = tf.image.resize(label, [384, 768])

    return norm(img, label)
```

Listing 4.1: Data augmentation utilizza nel training

4.2 Metriche di valutazione

In questa sezione verranno presentate le metriche di valutazione utilizzate durante le fase di training e di testing. Le valutazioni del modello su training-set e test-set sono realizzate per mezzo di "*classification metric functions*". Queste funzioni accettano in input le annotazioni fornite come *ground truth* e quelle stimate dalla rete, fornendo in uscita un insieme di punteggi. Ora verranno presentate le funzioni di classificazione più utilizzate.

Accuracy. La più semplice funzione metrica utilizzata nella classificazione è senza alcun dubbio l'accuratezza. Essa corrisponde alla percentuale o frazione di campioni classificati correttamente. In un problema di segmentazione semantica:

- per ogni classe, l'accuratezza è il rapporto tra il numero di pixel correttamente classificati e il numero totale della singola classe, secondo quanto specificato nell'annotazione fornita in ingresso,
- per ogni immagine, il valore di accuratezza è l'accuratezza media di tutte le classi per la singola immagine,
- per il dataset, il valore di accuratezza è l'accuratezza media di tutte le classi rispetto a tutte le immagini processate.

In generale però, l'accuratezza esprime una semplice e grossolana valutazione delle potenzialità del modello; è indicativa e non completamente affidabile, soprattutto in presenza di dataset sbilanciati, ovvero quando alcune classi hanno più istanze rispetto ad altre, falsandone quindi la percentuale di accuratezza.

IoU. Intersection over Union, conosciuto anche come coefficiente di similarità di *Jaccard*, è la metrica di valutazione più utilizzata.

- Per ogni classe, l'IoU rappresenta il rapporto tra i pixel correttamente classificati e la somma del numero complessivo di pixel etichettati e classificati:
$$IoU = \frac{TP}{TP + FP + FN}$$
- Per ogni immagine, l'IoU medio ($mIoU$) è la media dei valori di tutte le classi per la singola immagine,
- per l'intero dataset, $mIoU$ è la media dei valori di tute le classi in tutte le immagini.

Nel caso in cui il dataset fosse particolarmente sbilanciato, è più opportuno calcolare un valore di $mIoU$ pesato secondo il numero di pixel totale di ogni singola classe. Lo IoU, anch'esso variabile da 0 ad 1, non soffre del problema da cui è afflitta l'accuratezza ed è il singolo parametro che meglio descrive la qualità della segmentazione.

Tempo di Inferenza Medio. Il tempo medio necessario ad eseguire un'inferenza su nuovi dati:

$$Tempo\,Inferenza\,Medio = \frac{Tempo\,Totale}{Numero\,Campioni}$$

FPS. Frame per second è un'unità di misura che mostra le performance di un dispositivo; nel dettaglio consiste nel numero di immagini, chiamate appunto

frame, computate in un secondo.

$$FPS = \frac{1}{Tempo\,Inferenza}$$

Questa misura è direttamente legata alla potenza di calcolo del dispositivo, ma anche dalla velocità delle periferiche, quali memoria, bus e sistemi di acquisizione. Nella segmentazione semantica volta alla guida autonoma, la capacità di processare immagini in real-time è uno compiti più importanti e la misura degli fps va a determinare la bontà del modello in termini di ottimizzazione e "leggerezza".

4.3 Risultati

In questa sezione verranno presentati i risultati sperimentali ottenuti durante la fase di testing. Sono stati utilizzati tre diversi modelli della rete *EfficientSeg* con una fattore di espansione fisso pari a *1.3*, il quale indica l'espansione delle feature nei layer *inverted residual bottleneck* ereditati dalla MobileNetV3; i tre modelli sono allenati sul training set del dataset *Cityscapes* e circa il 20% di esso è stato estrapolato, prima dell'allenamento, per realizzare un sottoinsieme contenente le prime 500 immagini da usare come test set. La fase di training è stata eseguita su *batch* di 5 immagini di dimensioni *384x678*, ritagliate rispetto alla dimensione originale, dato l'alto utilizzo di memoria causato dall'utilizzo della convoluzione *depthwise* e dai limiti di memoria della scheda grafica. La rete utilizza una algoritmo di ottimizzazione *Adam* [24] con un decadimento dei pesi pari a $1\,e^{-6}$ e un *learning rate* iniziale pari a $1\,e^{-3}$; successivamente viene decrementato periodicamente sulla base di una pianificazione *Piecewise Constant Decay* che ne dimezza il valore dopo 25 epoche e successivamente ogni 10 epoche, fino a raggiungere il valore minimo di $5\,e^{-7}$. Infine come *loss function* è stata utilizzata la funzione *Sparse Categorical Crossentropy*. Come già anticipato all'inizio di questo capitolo, avendo un dataset con numerosi elementi della stessa categoria e soprattutto con texture molto simili, servirsi di tecniche di data augmentation consente di ridurre la dipendenza da questi fattori e di conseguenza dalla presenza di over-fitting. Infine nella fase di training si è fatto utilizzo di alcune *callback* offerte dalla piattaforma Tensorflow:

- **Early Stopping.** L'obiettivo della fase di training è quello di minimizzare la loss function e difatti è anche la principale metrica da monitorare. Quando viene avviato l'allenamento della rete, la callback controlla puntualmente alla fine di ogni epoca se la loss è diminuita. Impostando un

ulteriore parametro chiamato *patience* si va a stabilire il numero massimo tollerabile di epoche successive in cui non sono presenti dei miglioramenti nella loss prima di arrestare il training,

- **Tensorboard.** Estensione di Tensorflow che permette di acquisire i dati del training sotto forma di log. Tensorboard permette la visualizzazione dei dati raccolti durante la fase di training ed offre, inoltre, la possibilità di monitorare l'andamento dell'allenamento della rete. Viene spesso utilizzato dai ricercatori e ingegneri per comprendere particolari comportamenti della rete di *Machine Learning* che si sta allenando; inoltre tiene traccia delle metriche utilizzate negli esperimenti, mostra graficamente il modello implementato, gli hyper-parameter e gli istogrammi dei layer,

- **Model Checkpoint.** Questa callback è utilizzata per salvare il modello o i suoi pesi in un file di checkpoint con un certo intervallo di frequenza. In questo modo è possibile in un secondo momento caricare nuovamente il modello e riprendere la fase di training partendo dallo stato salvato.

Per completare l'elenco delle unità di misura utilizzate, si devono aggiungere il peso in memoria del modello (**Memory Cost**) calcolato in megabyte (MB) e il numero dei parametri (**Number of Parameters**).
Prima di illustrare i risultati ottenuti durante la fase di testing è necessario specificare l'hardware utilizzato; il training è stato effettuato su scheda grafica GTX 1080Ti con 11 GB di memoria grafica, mentre per il testing si è utilizzata la Raspberry Pi 4 con il modello esportato in formato Tensorflow Lite. La misura dell *mIoU* è stata effettuata su tutte le classi presenti nel dataset e quindi, a differenza dell'implementazione realizzata nell'articolo [8], risulta minore. Inoltre ci si aspetta una riduzione ulteriore nella precisione una volta che il modello sarà convertito al Tensorflow Lite, dato che verrà inoltre applicata una quantizzazione opportuna per renderlo compatibile con la Raspberry Pi.
É bene ribadire che lo scopo di questa tesi è quello di implementare una CNN che sia capace di lavorare su hardware con prestazioni limitate, dato che nello stato dell'arte la maggior parte dei modelli conosciuti sono testati su scheda grafica. Anche se i risultati ottenuti su processore grafico sono decisamente superiori, soprattutto in termini di tempo di inferenza, la possibilità di sfruttare una rete su un processore con architettura *ARM*, permette di abbattere considerevolmente i costi dell'hardware necessario nell'eseguire la segmentazione semantica.
I risultati ottenuti nella fase di testing sono racchiusi nelle due tabelle 4.2 e 4.3. Nella prima abbiamo quelli ottenuti dalla Raspberry Pi e si può vedere che il costo in termini di memoria, una volta esportato il modello in TFLite, ha una riduzione pari a circa 10x rispetto a quello su desktop; ovviamente il

	Geometry Data Aug.	Geometry Chromatic Data Aug.	Memory Cost (MB)	Number of Parameters	Accuracy	mIoU	Inference Time (s)	FPS
EfficientSeg			12.486	11,882,090	80.95	27.12	2.962	0.338
		x	12.486	11,882,090	81.10	27.90	2.968	0.336
	x		12.486	11,882,090	82.37	27.36	2.592	0.385

Tabella 4.2: Risultati ottenuti su Raspberry Pi 4 con modello TFLite

	Geometry Data Aug.	Geometry Chromatic Data Aug.	Memory Cost (MB)	Number of Parameters	Accuracy	mIoU	Inference Time (s)	FPS
EfficientSeg			137.451	11,882,090	85.40	31.35	0.0309	32.35
		x	137.451	11,882,090	88.45	33.96	0.0335	29.89
	x		137.451	11,882,090	87.10	32.52	0.0322	31.25

Tabella 4.3: Risultati ottenuti su GTX 1080Ti con modello H5

tempo di inferenza è sicuramente più alto e l'accuracy più bassa, ma ribadiamo che la tesi vuole evidenziare che il modello è capace di svolgere operazioni di segmentazione semantica anche su un microprocessore basato su architettura *ARM*. Anche se le prestazioni sono inferiori, il dislivello non è poi così tanto marcato ed è principalmente causato dalla quantizzazione necessaria affinché il modello sia compatibile con la Raspberry Pi. Inoltre la precisione può essere migliorata attraverso un ulteriore tuning dei parametri per cercare di ottenere il massimo potenziale dal modello di partenza. Nella figura 4.2 sono visibili degli esempi di predizione di segmentazione semantica ottenuti dalla rete.
Le seguenti immagini 4.3, 4.4, 4.5 mostrano i dati riguardanti la fase di training

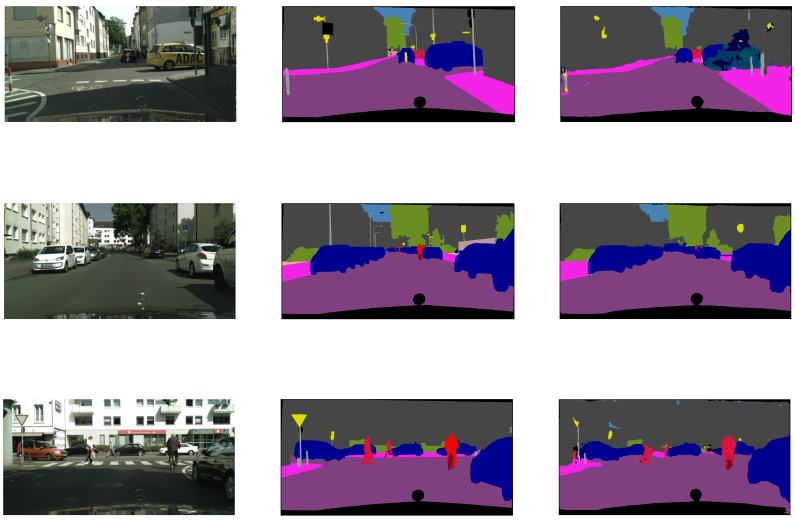

Figura 4.2: Esempio tre predizioni dalla rete EfficientSeg. Partendo da sinistra abbiamo l'immagine vera, la seconda mostra *ground truth* e infine nella terza la predizione della rete.

dei tre modelli. Come si può vedere dal primo grafico, i modelli con maggiore data augmentation ottengono un'accuracy inferiore, dovuta alla maggiore varietà di immagini; nel grafico centrale, invece viene messa in evidenza la loss function e anche in questo caso una maggiore data augmentation implica una loss più "rumorosa"; infine nell'ultimo a sinistra è presentato la mean Intersection over Union, dove anche qui si ritrova lo stesso trend dell'accuracy.

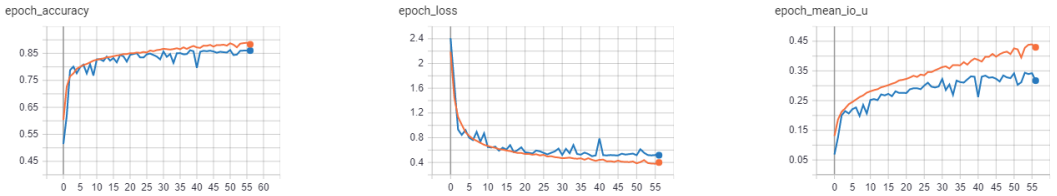

Figura 4.3: Grafici training del modello con data augmentation (geometrica, cromatica): da sinistra *accuracy*, *loss*, *mIoU*

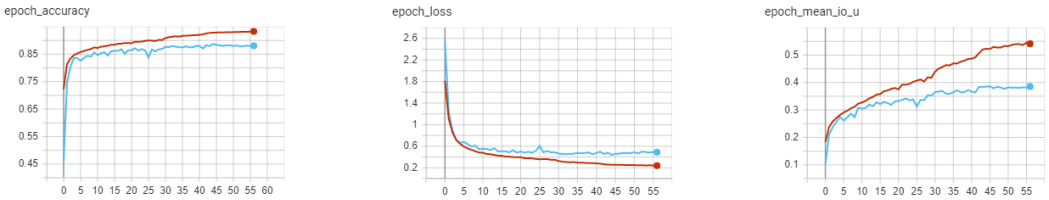

Figura 4.4: Grafici training del modello senza data augmentation: da sinistra *accuracy*, *loss*, *mIoU*

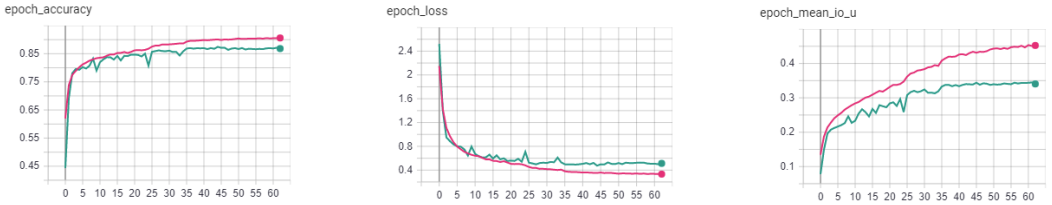

Figura 4.5: Grafici training del modello con data augmentation (geometrica): da sinistra *accuracy*, *loss*, *mIoU*

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stata presentata una rete neurale convolutiva (CNN) per la segmentazione semantica in applicazioni di guida autonoma su dispostivi embedded e ne è stata verificata la sua realizzazione così come le sue prestazioni. É stato preso in considerazione il dataset *Cityscapes* che rappresenta uno dei più ricchi dataset contenente scene urbane riprese da veicolo. L'architettura utilizzata è quella relativa alla rete *EfficientSeg*, della quale ne è stato realizzato un porting in Tensorflow. Si è verificata la possibilità di integrare tale rete all'interno di un dispositivo edge, quale la Raspberry Pi e se ne sono mostrate le prestazioni effettuando tre diversi esperimenti: il primo utilizza un training set privo di data augmentation, il secondo utilizza data augmentation con solo applicate trasformazioni geometriche e infine l'ultimo contenente sia trasformazione geometriche che cromatiche delle immagini. I risultati mostrano la fattibilità di sfruttare queste reti di *deep learning* in dispositivi embedded richiedendo un compromesso in termini di tempi di inferenza e di precisione.
Come sviluppi futuri, il progetto prevede l'utilizzo di tecniche di compressione, come quelle mostrate nel capitolo 2, per ottenere una riduzione del numero dei parametri del modello e conseguentemente del tempo di inferenza e dell'occupazione di memoria, rendendo la rete più fluida e performante. Un'altra possibilità da esplorare è quella relativa alle tecniche di pruning e decomposizione tensoriale, quali per esempio la *Tucker decomposition* o *CANDECOMP/PARAFAC (CP) decomposition*. Infine, a fine di migliorare la precisione del sistema, si potrà eseguire un "fine tuning" degli hyper-parametri del modello utilizzando ad esempio simulazioni con *random search* per trovare una configurazione ottimale dei parametri.

Bibliografia

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[3] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation."

[4] H. Zhang, C. Wu, Z. Zhang, Y. Zhu, Z. Zhang, H. Lin, Y. Sun, T. He, J. Mueller, R. Manmatha *et al.*, "Resnest: Split-attention networks," *arXiv preprint arXiv:2004.08955*, 2020.

[5] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[6] X. Li, L. Zhang, A. You, M. Yang, K. Yang, and Y. Tong, "Global aggregation then local distribution in fully convolutional networks," *arXiv preprint arXiv:1909.07229*, 2019.

[7] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," *arXiv e-prints*, pp. arXiv–1905, 2019.

[8] V. B. Yesilkaynak, Y. H. Sahin, and G. Unal, "Efficientseg: An efficient semantic segmentation network," *arXiv preprint arXiv:2009.06469*, 2020.

[9] P. Chao, C.-Y. Kao, Y.-S. Ruan, C.-H. Huang, and Y.-L. Lin, "Hardnet: A low memory traffic network," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 3552–3561.

[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[11] Cisco, "Fog computing and the internet of things: extend the cloud to where the things are," 2015. [Online]. Available: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[13] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[14] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 8612–8620.

[15] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[16] 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.

[17] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[18] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[20] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *arXiv preprint arXiv:1505.04597*, 2015.

[21] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," 2016.

[22] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, vol. 107, pp. 3–11, 2018.

[23] M. Cordts, M. Omran, S. Ramos, T. Scharwächter, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset."

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