

*UNIVERSITÀ POLITECNICA DELLE MARCHE*

*FACOLTÀ DI INGEGNERIA*



*Corso di Laurea Triennale in  
Ingegneria Informatica e dell'Automazione*

***Progettazione e Sviluppo di un algoritmo per il lane  
detection nell'ambito della corsa di atleti ipovedenti basato  
su immagini RGB***

*Design and development of an algorithm for lane detection in the field of blind athletes  
based on RGB images*

Relatore:  
DOTT. MANCINI ADRIANO

Laureando:  
TRIBUIANI LORENZO

ANNO ACCADEMICO 2020-2021

# Indice

<b>Indice</b>	<b>i</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 Analisi del progetto . . . . .	1
<b>2 Introduzione all'AI</b>	<b>3</b>
2.1 Cenni storici . . . . .	3
2.2 Classificazione dell'AI . . . . .	5
2.2.1 Sistemi esperti . . . . .	5
2.2.2 Machine Learning . . . . .	6
2.2.3 Deep Learning . . . . .	9
<b>3 Reti Neurali</b>	<b>11</b>
3.1 Il Percettrone . . . . .	12
3.1.1 Modello McCulloch-Pits . . . . .	12
3.1.2 Funzioni di attivazione . . . . .	13
3.1.3 I limiti del percettrone . . . . .	15
3.2 MLP o Reti Neurali . . . . .	21
3.2.1 Modello Matematico . . . . .	22
3.2.2 Soluzione del problema dello XOR . . . . .	23
3.2.3 Back-propagation . . . . .	25
<b>4 Reti Neurali Convolute</b>	<b>31</b>
4.1 Introduzione alla Computer Vision . . . . .	31
4.1.1 Object Detection . . . . .	33
4.1.2 Image Segmentation . . . . .	34
4.2 Tensori . . . . .	35
4.3 CNN (Convolutional Neural Networks) . . . . .	37
4.3.1 Convolutional Layers . . . . .	39
4.3.2 Pooling . . . . .	42
4.3.3 Struttura delle reti convolute . . . . .	43
4.3.4 Vantaggi e limiti delle reti convoluzionali . . . . .	45
4.3.5 R-CNN . . . . .	47
4.3.6 Fast R-CNN . . . . .	48
4.3.7 Faster R-CNN . . . . .	49

4.4	FCN (Fully Convolutional Networks) . . . . .	51
4.4.1	Down-sampling e classificazione . . . . .	51
4.4.2	Up-Sampling . . . . .	53
<b>5</b>	<b>Sviluppo Del Progetto</b>	<b>59</b>
5.1	Architettura Della Rete . . . . .	60
5.1.1	U-Net . . . . .	60
5.1.2	Implementazione della U-Net . . . . .	61
5.2	Creazione Del Dataset . . . . .	66
5.2.1	Dati ed Etichette . . . . .	67
5.3	Addestramento della rete . . . . .	68
5.4	Ottimizzazioni . . . . .	69
5.4.1	Modello Ridotto . . . . .	70
5.4.2	Modello Lite . . . . .	73
5.4.3	Quantizzazione a int8 . . . . .	75
<b>6</b>	<b>Risultati</b>	<b>77</b>
6.1	Modello Completo . . . . .	78
6.1.1	Analisi delle previsioni . . . . .	79
6.1.2	Tempi di esecuzione . . . . .	81
6.2	Modello Ridotto . . . . .	82
6.2.1	Analisi delle previsioni . . . . .	82
6.2.2	Tempi di esecuzione . . . . .	84
6.3	Modello Ridotto Lite . . . . .	85
6.3.1	Analisi delle Previsioni . . . . .	86
6.3.2	Tempi di esecuzione . . . . .	88
6.4	Modello Ridotto Lite Quantizzato . . . . .	89
6.4.1	Analisi delle previsioni . . . . .	89
6.4.2	Tempi di esecuzione . . . . .	91
<b>7</b>	<b>Conclusioni e Sviluppi Futuri</b>	<b>95</b>
7.0.1	Rilevamento Della Traiettorie . . . . .	96
7.0.2	Conclusioni . . . . .	100
	<b>Bibliografia</b>	<b>101</b>
	<b>Sitografia</b>	<b>105</b>
	<b>Elenco delle figure</b>	<b>107</b>
	<b>Elenco delle tabelle</b>	<b>111</b>

# Capitolo 1

## Introduzione

Le disabilità fisiche e sensoriali hanno da sempre rappresentato un limite talvolta insormontabile nella pratica di discipline sportive di vario genere. Sebbene le varie federazioni e società sportive e tutto il mondo dello sport abbiano negli anni costruito un *circuito parallelo* sportivo interamente dedicato ad atleti con disabilità fisiche o sensoriali (dalla fondazione delle *Paralimpiadi* e i vari sport non inclusi) questi sport restano fortemente legati a guide esterne che possono essere persone o sussidi di vario tipo e che rendono di conseguenza l'atleta non del tutto autonomo nella pratica sportiva.

Parallelamente l'avanzare della tecnologia e della ricerca legata allo sviluppo dell'intelligenza artificiale e della robotica hanno permesso la modellazione e lo sviluppo di macchine autonome in grado di compiere compiti complessi con precisione pari se non superiore a quella di un essere umano. Macchine autonome, sistemi di guida assistita, riconoscimento e classificazione delle immagini sono solo alcuni dei risultati ottenuti negli ultimi anni.

Il progetto *Blind Athletes* ha come obiettivo quello di utilizzare la tecnologia come sussidio per l'attività fisica di atleti con disabilità visiva o *ipovedenti*. Il campo della *Computer vision* è uno dei rami dell'intelligenza artificiale più prolifici che, grazie a reti neurali ad apprendimento profondo ottimizzate per il lavoro sulle immagini, consente di discriminare in modo sufficientemente preciso il sistema visivo umano. È possibile quindi utilizzare un sistema intelligente in grado di distinguere gli elementi e gli ostacoli frontali (tramite l'utilizzo di una telecamera) per guidare un atleta, tramite dei segnali ben specifici, all'interno di un determinato percorso o di una determinata corsia.

### 1.1 Analisi del progetto

L'obiettivo del progetto è quello di sviluppare un algoritmo in grado di riconoscere la corsia su cui si sta muovendo un atleta rilevando eventuali spostamenti verso l'esterno della corsia e guidandolo, tramite dei guanti vibranti, attraverso tutto il circuito impedendogli di invadere la corsia degli altri atleti (quest'ultima precisazione vale non solo a livello regolamentare, ma anche a livello fisico, ovvero per



evitare incidenti o collisioni durante la corsa). L'algoritmo deve essere in grado di analizzare immagini in formato RGB<sup>1</sup> riconoscendo i confini e la posizione della corsia rispetto all'atleta in tempi sufficientemente veloci.

L'algoritmo è pensato per sfruttare un sistema di apprendimento profondo (una rete neurale) allenata su un dataset sufficientemente ampio e specifica per le corsie di atletica.



Figura 1.1: Esempio di rilevamento corsia. Sulla sinistra l'immagine originale, sulla destra in giallo viene evidenziata la corsia centrale rilevata dalla rete (Risultati del progetto).

Nel capitolo 2 dell'elaborato viene presentata un'introduzione generica all'*Intelligenza Artificiale* con alcuni cenni storici e una divisione dei campi di studio della disciplina stessa. Nei capitoli 3 e 4 verranno approfonditi i concetti relativi alle *Reti Neurali*, dal *perceptrone* ai sistemi di addestramento (capitolo 3) con particolare attenzione alla *computer vision* e alle *reti neurali convolutive* (capitolo 4). Il capitolo 5 viene interamente dedicato allo sviluppo del progetto riportando i vari modelli sviluppati, i *tool* utilizzati e le varie ottimizzazioni. Nel capitolo 6 vengono riportati ed analizzati i risultati ottenuti dai vari modelli sviluppati sia relativamente all'accuratezza delle previsioni effettuate che ai tempi di inferenza di ciascun modello. Nell'ultimo capitolo dell'elaborato verranno esposti un possibile algoritmo di rilevamento della traiettoria dell'atleta, a partire dalle previsioni del modello, e le conclusioni relative a tutto il progetto.

---

<sup>1</sup>Dall'inglese *Red Green Blue*, è un modello di colori additivo per la rappresentazione dell'intero spazio cromatico

# Capitolo 2

## Introduzione all'AI

L'intelligenza artificiale, generalmente indicata con l'acronimo AI (dall'inglese *Artificial Intelligence*), è un ramo della *computer science* che ha come obiettivo quello di creare macchine o algoritmi in grado di emulare i comportamenti e/o i processi cognitivi tipici degli esseri umani. Questa prima e piuttosto generica definizione implica un campo di azione dell'AI estremamente vasto, alcuni esempi di intelligenza artificiale sono la *Computer Vision*, ovvero l'insieme dei processi mirati alla riproduzione della visione umana, lo *Speech Recognition*, il campo dell'AI che si occupa del riconoscimento e della traduzione del linguaggio umano, sistemi di guida automatizzati, previsioni statistiche e molti altro.

Sebbene ad oggi l'intelligenza artificiale sia una disciplina strettamente, se non esclusivamente, legata all'informatica è bene precisare che molti dei modelli dell'AI sono basati su sistemi prettamente di tipo logico matematico, molti dei quali teorizzati molto prima dell'avvento dell'intelligenza artificiale (il *Perceptron* o *Perceptrone*, elemento alla base dello sviluppo di sistemi di apprendimento profondo come le *Reti Neurali* venne per la prima volta teorizzato nel 1943 e implementato nel 1968, analogamente le basi della *Regressione Lineare*, fondamento della soluzione dei problemi di classificazione, risalgono ai primi dell'ottocento). Il largo sviluppo delle tecnologie legate all'informatica e ai calcolatori, in particolare l'avvento del *Pipelining*, la Multiprogrammazione e la larga diffusione di architetture *Multicore* o *GPU* (dall-inglese *Graphics Processing Unit*), hanno consentito una crescita esponenziale della ricerca sull'AI che, negli ultimi anni, sta diventando uno tra i principali e più prolifici argomenti di studio legati all'informatica.

### 2.1 Cenni storici

L'intelligenza artificiale come disciplina ha avuto il suo principale sviluppo a partire dal secolo scorso. Le innovazioni tecnologiche, le ricerche e le scoperte fatte all'inizio del XX secolo relative alle reti neurali biologiche hanno posto le fondamenta per l'avvento di modelli mirati alla costruzione di sistemi artificiali intelligenti.

Nel 1950 il matematico **Alan Turing**, uno dei padri dell'informatica, nell'incipit dell'articolo *Computing machinery and intelligence* dice:

Mi propongo di considerare la questione: “Possono pensare le macchine?” Si dovrebbe cominciare col definire il significato dei termini “macchina” e “pensare”. Le definizioni potrebbero essere elaborate in modo da riflettere il più possibile l'uso normale delle parole, ma questo atteggiamento è pericoloso. [...] Invece di tentare una definizione di questo tipo sostituirò la domanda con un'altra, che le è strettamente analoga e che è espressa in termini non troppo ambigui. La nuova forma del problema può essere descritta nei termini di un gioco, che chiameremo “il gioco dell'imitazione”. [1]

Turing propone quindi un test in grado di valutare se una macchina può o meno essere intelligente. Il test di Turing consiste nel porre una persona (definita *giudice*) davanti ad un terminale che comunica con un essere umano e con una macchina. Il giudice deve porre delle domande al terminale e, se le risposte date dalla macchina e quelle date dall'uomo sono indistinguibili dal giudice, allora la macchina può essere considerata intelligente.

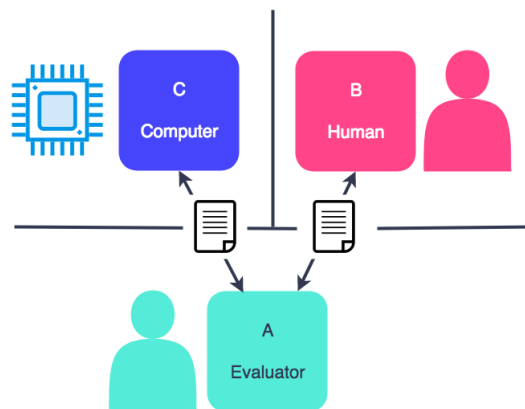


Figura 2.1: Rappresentazione del test di Turing [2]

Dalla seconda metà del secolo scorso la ricerca e l'interesse nei confronti della ricerca sull'intelligenza artificiale hanno mostrato andamenti altalenanti alternando periodi di grande interesse per la materia a periodi di scarso interesse. A partire dagli anni '90 in poi l'interesse è stato sempre crescente e l'avvento della tecnologia ha consentito l'ottenimento di risultati sempre migliori. Oggi l'intelligenza artificiale rappresenta uno dei principali campi di interesse nella ricerca tecnologica e molte delle principali multinazionali del settore hanno sviluppato progetti legati all'AI (il principale esempio è *Google* con i progetti *Deepmind*, lo sviluppo delle *Tensor Processing Unit TPU*, unità di calcolo pensate per lavorare con i *Tensori*, e le librerie *Tensorflow*)

## 2.2 Classificazione dell'AI

Come detto in precedenza l'intelligenza artificiale è un campo di studio estremamente vasto che comprende sistemi e tecniche differenti per giungere al medesimo risultato. È tuttavia possibile fare una divisione logica dei principali campi di lavoro dell'AI in base al tipo di apprendimento utilizzato.

La prima classificazione dell'intelligenza artificiale distingue il campo di studi in due macrocategorie divise principalmente dagli obiettivi raggiungibili dall'AI stessa, ovvero un'intelligenza generale, in grado di risolvere problemi di qualsiasi natura, e una specifica capace di trovare soluzione a problemi ben definiti:

- **Intelligenza Artificiale Forte** detta anche AGI (*Artificial General Intelligence*) basata sull'ipotesi che le macchine possano effettivamente essere intelligenti, ovvero una AGI è un'intelligenza artificiale esattamente analoga all'intelligenza umana
- **Intelligenza Artificiale Debole** detta anche ANI (*Artificial Narrow Intelligence*) basata invece sull'ipotesi che le macchine possano comportarsi come se fossero intelligenti

Come detto in precedenza questa distinzione giace interamente sul piano logico e non implementativo dell'intelligenza artificiale. Nella pratica effettiva tutti gli algoritmi e i processi di sviluppo di sistemi intelligenti oggi risiedono nell'intelligenza artificiale debole, ovvero siamo in grado di creare macchine altamente specializzate capaci di trovare soluzione a un ben specifico tipo di problema. Questo dipende da ragioni di tipo implementativo e dai limiti tecnologici attuali che rendono molto difficile, se non impossibile, creare macchine con un'intelligenza generale, ovvero in grado di risolvere problemi di varia natura distinguendo informazioni e richieste analogamente a quanto farebbe un essere umano.

Nelle successive sezioni, partendo da questa prima divisione, andremo a dividere ulteriormente il campo dell'intelligenza artificiale prendendo come metro di giudizio i metodi di implementazione e i sistemi di apprendimento distinguendo quelli che oggi sono i principali campi di lavoro dell'intelligenza artificiale debole, ovvero i *Sistemi Esperti*, il *Machine Learning* e il *Deep Learning*.

### 2.2.1 Sistemi esperti

I più semplici tipi di AI sono i *sistemi esperti* basati su *grafici della conoscenza*. In questi casi, parlando in termini generali, quello che si ottiene è un algoritmo che compie una serie di passi per giungere alla soluzione di un ben determinato problema. Sebbene spetti al programmatore implementare il sistema risolutivo la macchina sarà poi in grado di risolvere un problema deterministico analogamente a quanto farebbe un essere umano.

Un esempio di sistema esperto è l'algoritmo *Minimax* applicato ai giochi a turni. Supponendo di giocare un gioco con due giocatori dove ogni giocatore può effettuare una mossa per turno, l'algoritmo minimax crea un albero di gioco visualizzando

tutte le possibili combinazioni di mosse che possono essere effettuate dal giocatore e dall'avversario a partire dal turno corrente fino ad un n-esimo turno (dipende dall'implementazione e dalla complessità del gioco) e assegna a ciascuna di esse un peso numerico. Presupponendo che il giocatore voglia massimizzare il suo punteggio e l'avversario voglia minimizzare il punteggio del giocatore ripercorre l'albero a ritroso partendo dalle foglie e scegliendo per ogni turno il minimo o il massimo fra i valori (in base al giocatore di turno) e in base a questo effettua la decisione sulla giocata da fare.

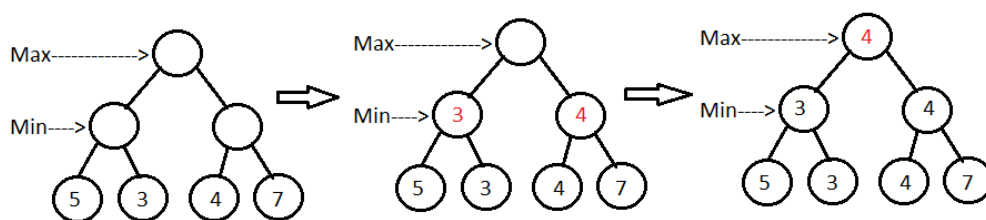


Figura 2.2: Albero di gioco a tre turni

Se applicato a giochi piuttosto semplici, come ad esempio il *tris*, l'algoritmo non solo si comporta analogamente ad un essere umano, ma risulta imbattibile, il miglior punteggio che si possa ottenere è un pareggio. In giochi più complessi, come ad esempio gli scacchi, non è possibile analizzare l'intero albero di gioco e di conseguenza l'algoritmo deve arrestare la sua analisi in anticipo. Nonostante questo è comunque possibile sviluppare algoritmi di tipo minimax per giochi complessi.

### 2.2.2 Machine Learning

I *Sistemi esperti* basati su *alberi o grafi della conoscenza* hanno dominato il panorama dell'AI per almeno trent'anni, dalla loro prima formulazione negli anni '50 fino alla fine degli anni '80. Questi sistemi tuttavia soffrono di alcuni problemi legati principalmente alla difficoltà nel trovare algoritmi soddisfacenti per la soluzione di problemi complessi e un sistema di apprendimento non autonomo. I sistemi esperti di fatto non sono in grado di migliorare i loro risultati basandosi sui loro errori; sono semplicemente in grado eseguire un algoritmo pensato ad hoc per la soluzione di uno specifico problema.

Il *Machine Learning*, di contro, è una branca dell'intelligenza artificiale mirata all'ottenimento di algoritmi risolutivi con un sistema di apprendimento autonomo, ovvero, sistemi informatici in grado di elaborare dati al fine di migliorare le prestazioni di risoluzione di problemi basandosi sugli errori commessi. Questo ramo dell'intelligenza artificiale, sebbene teorizzato quasi in contemporanea con i principali concetti dell'AI (1959 ca.), ha visto il suo sviluppo a partire dall'ultimo decennio del secolo scorso. Questo dipende principalmente dal rapporto che

il *machine learning* ha con i *Big Data*. Affinché sia possibile un apprendimento autonomo i sistemi intelligenti devono avere a disposizione grandi quantità di dati in modo da poter migliorare la propria accuratezza e, di conseguenza, lo sviluppo del *Machine Learning* ha dovuto attendere l'avanzamento di altre tecnologie.

Verso la fine degli anni '80 del secolo scorso la grande ricerca sui *Sistemi di dati e Database* e lo sviluppo del *World Wide Web* hanno quindi fornito al *Machine Learning* tutti gli strumenti di cui aveva bisogno e quello che era inizialmente un sistema di difficile, se non impossibile realizzazione ha ben presto dato ottimi risultati fino a dominare, quasi esclusivamente, il campo dell'intelligenza artificiale.

### Sistemi di apprendimento

Gli algoritmi di *Machine Learning*, come detto, devono analizzare grandi quantità di dati al fine di creare un modello predittivo sufficientemente accurato per la risoluzione di un dato problema. Il processo di analisi dei dati viene generalmente definito come *Apprendimento* o *Addestramento* ed è una componente comune a tutti i tipi di algoritmi del ML<sup>1</sup>. Tuttavia algoritmi differenti possono adottare sistemi differenti di *apprendimento* e, in base al tipo di dati e gli input e gli output dell'algoritmo, possiamo distinguere tre principali categorie di addestramento per sistemi ML:

- **Apprendimento Supervisionato**

L'apprendimento supervisionato, o addestramento supervisionato, costruisce un modello predittivo a partire da *Dataset*<sup>2</sup> dei quali conosciamo già gli output desiderati. Informalmente forniamo al nostro modello gli input a partire dal dataset, il modello produrrà quindi un output che verrà confrontato con l'output desiderato e ne verrà valutato l'errore. L'addestramento del modello consiste quindi di due parametri fondamentali ovvero **l'accuratezza** (parametro che deve essere massimizzato) e **l'errore** (parametro che deve essere invece minimizzato).

Questo tipo di addestramento viene generalmente applicato a due grandi classi di problemi quelli di *Classificazione* e quelli di *Regressione*:

- **Classificazione.** I problemi di classificazione utilizzano algoritmi per assegnare con precisione i dati di input a categorie specifiche. È possibile utilizzare algoritmi di apprendimento supervisionato per classificare lo spam in una cartella separata dalla posta in arrivo. Classificatori lineari, macchine a vettori di supporto e alberi decisionali sono tutti tipi comuni di algoritmi di classificazione.
- **Regressione.** I problemi di regressione lavorano invece in campo continuo e hanno l'obiettivo di mappare gli input dati in un precisa funzione

---

<sup>1</sup>Acronimo di Machine Learning

<sup>2</sup>In generale con il termine Dataset andiamo ad indicare l'insieme dei dati con cui il modello verrà addestrato

continua. Un esempio di regressione la predizione dell'andamento di un determinato valore monetario.

- **Apprendimento non Supervisionato**

L'apprendimento non supervisionato utilizza algoritmi che sono in grado di analizzare e raggruppare i dataset senza conoscere gli output desiderati. Questo tipo di apprendimento classifica i dati instaurando relazioni interne di similitudine e attinenza e i dati utilizzati nell'addestramento sono privi di etichette (non se ne conosce l'output desiderato). Questi modelli trovano largo utilizzo nei motori di ricerca (che associano pagine web a determinate parole chiave) o negli algoritmi che consigliano determinati contenuti sulla base delle preferenze dell'utente.

I principali campi di applicazione di questi algoritmi sono i problemi di *Clustering* e il *Ridimensionamento dei dati*:

- **Clustering.** I problemi di data Clustering utilizzano algoritmi in grado di raggruppare dati privi di etichette in gruppi omogenei fra loro in base alle loro somiglianze o differenze. Questa classe di problemi presenta molte similitudini con i problemi di classificazione, tuttavia in questi ultimi sono note, in numero e specifiche, le possibili classi di suddivisione, mentre nel *Clustering* non conosciamo quali e quante classi di associazione potranno essere create, dipende esclusivamente dai dati forniti e dal modello. Per questo motivo si parla a volte di clustering come di *Classificazione non supervisionata*.
- **Ridimensionamento dei dati.** Il ridimensionamento è una tecnica di apprendimento utilizzata quando il numero di caratteristiche (o dimensioni) in un dato insieme di dati è troppo alto. Questi algoritmi hanno l'obiettivo di ridurre la dimensione o il numero di dati senza però avere perdita di informazione.

- **Apprendimento Rinforzato**

I sistemi ad *Apprendimento Rinforzato* rappresentano una variazione piuttosto significativa rispetto ai due precedenti modelli di addestramento dei modelli. In questo caso l'addestramento del modello non avviene tramite l'utilizzo di dataset di grandi dimensioni, anzi, l'apprendimento rinforzato prevede pochi se non nessun dato alla base. Il sistema di apprendimento si basa su quella che viene definita **Funzione di rinforzo** e sui meccanismi di *premio* e *punizione*. Inizialmente la nostra macchina viene inserita in un ambiente (fisico o virtuale) con l'input di raggiungere un obiettivo. I sistemi ad *Apprendimento rinforzato* osservano l'ambiente che li circonda e generano un vettore definito come **vettore delle caratteristiche** all'interno del quale vengono inserite le informazioni rilevate sull'ambiente. Ogni decisione presa dalla nostra macchina (quindi ogni output prodotto) viene poi valutato dalla funzione di rinforzo che ha l'obiettivo di premiare la macchina se si è

avvicinata all'obiettivo (ad esempio assegnando un valore positivo) o punirla nel caso si sia allontanata (ad esempio assegnando un valore negativo).

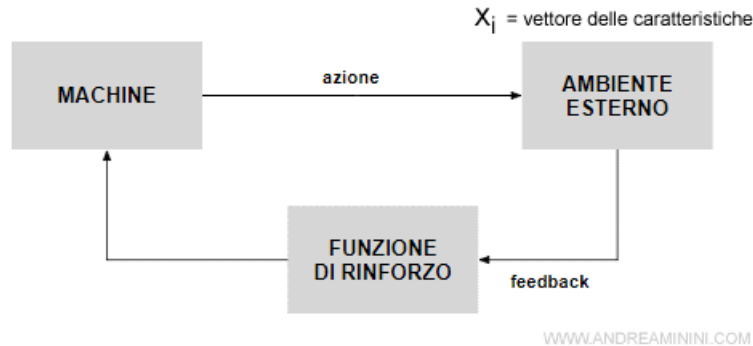


Figura 2.3: Schema a blocchi dell'apprendimento rinforzato

L'ultimo passo compiuto da questi sistemi è quello di andare a memorizzare i dati migliori all'interno di una banca dati definita come *Knowledge Base (KB)*. L'obiettivo del nostro sistema è quindi quello di massimizzare la funzione di rinforzo per ottenere un modello analitico accurato.

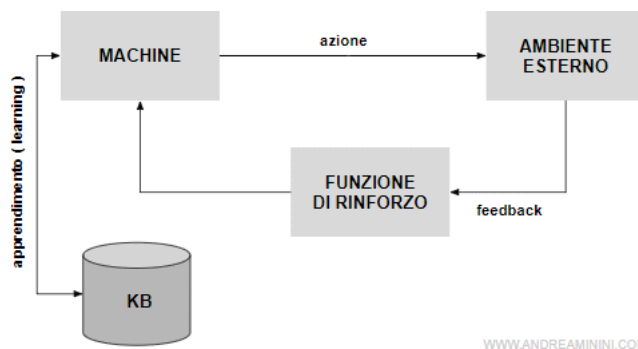


Figura 2.4: Schema a blocchi dell'apprendimento rinforzato con KB

### 2.2.3 Deep Learning

Il *Deep Learning*, o *Apprendimento Profondo*, ultima classificazione per l'intelligenza artificiale, rappresenta un ramo del *Machine Learning* e basa il suo lavoro sulla classificazione gerarchica dei dati in architetture multilivello che vengono generalmente definite *Reti Neurali*. Il *Deep Learning* pone quindi la sua attenzione allo sviluppo di sistemi intelligenti basati sulle reti neurali e, negli ultimi anni, ha avuto sempre più larga diffusione fino a diventare uno dei principali campi di ricerca dell'AI.

Come detto in precedenza l'apprendimento profondo rappresenta un ramo del *Machine Learning*, di conseguenza tutte le considerazioni fatte in precedenza relative ai sistemi di apprendimento sono tutt'ora valide (*Apprendimento supervisionato*,



*Apprendimento non supervisionato e Apprendimento Rinforzato).*

Tutte le classi di intelligenza artificiale viste rappresentato una rappresentazione di inclusione come può essere notato dal seguente grafico.

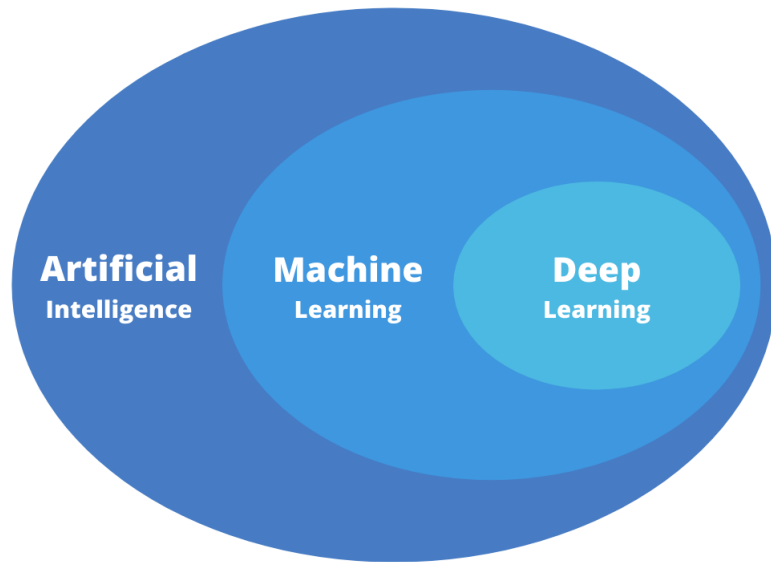


Figura 2.5: Relazione fra la classi di intelligenza artificiale

# Capitolo 3

## Reti Neurali

Le *Reti Neurali Artificiali* o semplicemente *Reti Neurali* (ci si riferisce ad esse anche con l'acronimo *ANN*, dall'inglese *Artificial Neural Network*) offrono sistemi intelligenti ad apprendimento autonomo in grado di risolvere problemi di classificazione, regressione e controllo non-lineare e rappresentano l'elemento alla base del *Deep Learning* o *Apprendimento Profondo*. Come suggerito dal nome stesso questi modelli prendono ispirazione, seppur semplificando, dalle reti neurali biologiche e in particolar modo dal cervello umano.

Lo sviluppo delle tecnologie legate al *Calcolo Parallelo* e il calcolo mediante GPU ha permesso a questi modelli di vedere un grande sviluppo negli ultimi anni ottenendo così risultati sempre migliori e un costante interesse di ricerca.

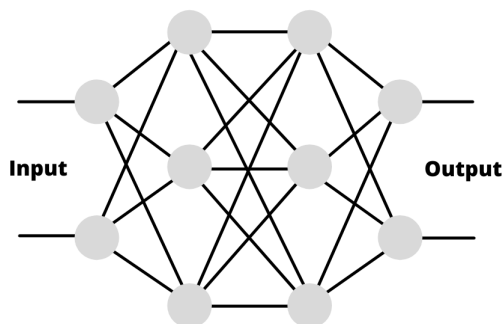


Figura 3.1: Schema elementare di una rete Neurale

Dal punto di vista modellistico matematico alla base delle reti neurali vi è la possibilità di esprimere una desiderata funzione  $f(x)$  come una composizione di altre funzioni  $g(x)$  che a loro volta possono essere espresse in termini di composizione di funzioni più semplici. Questo concetto porta quindi ad una prima formulazione del concetto di rete neurale come un *insieme interconnesso di funzioni elementari i cui output sono gli input delle successive funzioni*. L'elemento alla base delle reti

neurali sono quindi le funzioni elementari che formano poi le connessioni, quello che in una rete biologica chiameremmo *Neurone* e che, nelle reti neurali artificiali, viene definito *Percettrone*.

## 3.1 Il Percettrone

Il *Percettrone* è l'elemento alla base delle reti neurali. In termini grafici nella fig. 3.1 ogni pallino grigio rappresenta un percettrone. Questi elementi sono delle funzioni che prendono  $n$  elementi in input e forniscono una singola uscita che viene poi inviata come input ai successivi percettroni (questo chiaramente in *Reti neurali stratificate*). Il primo modello di percettrone venne presentato nel 1943 da *Warren McCulloch* e *Walter Pitts* e prende appunto il nome di *Modello di McCulloch-Pitts*.

### 3.1.1 Modello McCulloch-Pitts

Il modello del *percettrone* di McCulloch-Pitts rappresenta la prima formalizzazione teorica di un neurone artificiale.

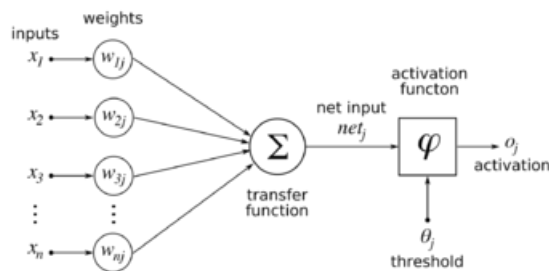


Figura 3.2: Modello di McCulloch-Pitts

Esattamente come un neurone biologico il percettrone prende un insieme di input che viene, nel modello matematico, descritto come un vettore  $X = (x_1, \dots, x_n)$  e produce una singola uscita  $y$ . Tuttavia, come è possibile osservare nella figura 3.2, questi non sono gli unici parametri coinvolti nella funzione interna del percettrone. Possiamo notare un secondo vettore di elementi  $W = (w_1, \dots, w_n)$  definito come vettore dei pesi. I pesi, nel percettrone come nelle reti neurali, rappresentano gli elementi fondamentali nell'addestramento del percettrone stesso. Questi elementi sono dei pesi moltiplicativi che, nel modello più semplice, erano numeri naturali e potevano assumere i valori  $-1$  o  $1$ , tuttavia, nei modelli moderni (più complessi), i pesi sono numeri reali che possono assumere qualsiasi valore nell'intervallo  $[-1, 1]$ . Questi valori giungono poi all'interno di un nodo sommatore che, appunto, ne effettua la somma pesata. Il modello fin qui descritto del percettrone può quindi essere formalizzato come:

$$y = \left( \sum_{i=1}^n x_i w_i \right) + b$$

o analogamente:

$$y = w^T \cdot x + b$$

Dove l'operatore  $\cdot$  rappresenta il *Prodotto scalare*. Il termine  $b$  viene detto *bias* del perceptrone. Il *bias* può essere considerato un peso a tutti gli effetti (può di fatti essere modificato nella fase di addestramento) svolge tuttavia un compito differente rispetto agli altri pesi. Il suo compito è quello di modificare (aumentandola o diminuendola) la *Soglia di attivazione*, il concetto diventerà più chiaro quando parleremo delle funzioni di attivazione.

L'ultimo elemento del perceptrone, appunto, è la *funzione di attivazione*. Analogamente a quanto accade nei neuroni biologici, dove l'informazione viene trasferita da un neurone ad un altro solo se il potenziale supera una certa soglia, la *funzione di attivazione* del perceptrone consente il passaggio dell'informazione al successivo perceptrone solo se l'output generato supera una certa soglia (soglia che, come detto in precedenza, può essere manipolata dal volere di *bias*).

Il modello del nostro perceptrone quindi può essere formalizzato come segue:

$$y = \phi\left(\sum_{i=1}^n x_i w_i + b\right)$$

analogamente:

$$y = \phi(w^T \cdot x + b)$$

dove  $\phi$  rappresenta appunto la *funzione di attivazione*.

### 3.1.2 Funzioni di attivazione

Come detto in precedenza le *funzioni di attivazione* rappresentano una parte fondamentale del perceptrone e di conseguenza delle reti neurali. Abbiamo bisogno di una funzione che invii il segnale solo se il livello di output del perceptrone supera una certa soglia. Per comodità possiamo fissare questa soglia a zero (sarà poi compito del bias modificare l'output del perceptrone affinché la soglia sia quella desiderata) e analizzare alcune funzioni che possono svolgere questo compito.

#### Funzione di Heaviside o gradino unitario

La funzione di *Heaviside*, detta anche gradino unitario, è una funzione largamente utilizzata in campo ingegneristico per via di alcune sue proprietà e viene generalmente descritta nella seguente forma:

$$y(x) = \begin{cases} 0 & \text{se } x \leq 0 \\ 1 & \text{se } x > 0 \end{cases}$$

e presenta l'andamento in figura 3.3.

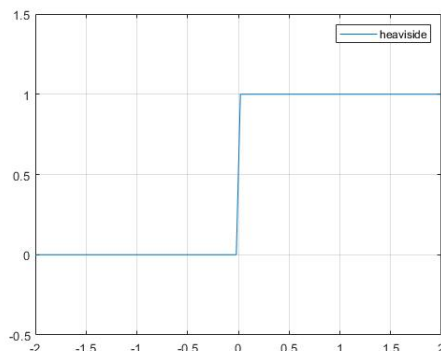


Figura 3.3: Andamento della funzione di heaviside (Da MATLAB)

Questa funzione "attiva" il perceptrone, ovvero consente la trasmissione dell'output, solo se questo è maggiore di zero. Inoltre si tratta di una funzione che produce un'uscita discreta (0 o 1) a prescindere dall'uscita effettiva del perceptrone. Questa funzione, seppur valida, non è derivabile nel punto 0 e questo rappresenta un grave problema nei sistemi di addestramento che hanno bisogno di conoscere la pendenza della curva nel punto desiderato (ovvero la derivata della funzione stessa) e di conseguenza non viene utilizzata.

### Sigmoide

La funzione *Sigmoide* viene introdotta per ovviare al problema della non derivabilità della funzione di *Heaviside* nel punto zero. La sigmoide presenta un andamento non lineare e viene descritta come segue:

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

e ha l'andamento descritto nella figura 3.4

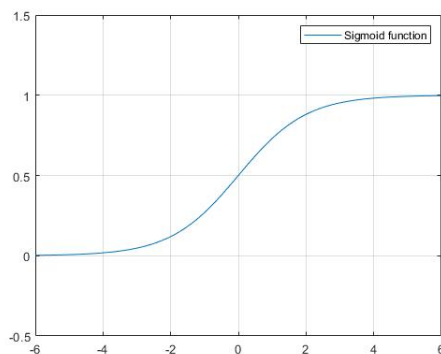


Figura 3.4: Andamento della Sigmoide (da MATLAB)

La funzione sigmoide presenta numerosi vantaggi rispetto al gradino. Innanzitutto è derivabile, e gode di una particolare proprietà, ovvero:

$$y(x)' = y(x)(1 - y(x))$$

il che rende il calcolo del valore della derivata molto più semplice. Inoltre la sigmoide normalizza il valore dell'uscita riportandolo ad un numero reale compreso fra 0 e 1.

## ReLU

La funzione *ReLU*, acronimo per *Rectifier Linear Unit*, è una funzione di attivazione che ha trovato largo utilizzo nelle reti neurali moderne, specialmente nei livelli intermedi, descritta come:

$$y(x) = \begin{cases} 0 & \text{se } x \leq 0 \\ x & \text{se } x > 0 \end{cases}$$

e presenta l'andamento in figura 3.5

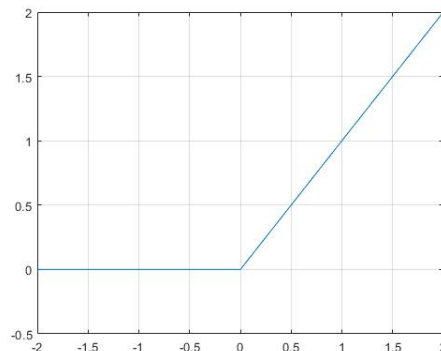


Figura 3.5: Andamento della funzione ReLU (da MATLAB)

Sebbene anche questa funzione non sia derivabile in 0 si assume per convenzione che la sua derivata in quel punto valga zero. La funzione risulta essere molto semplice e ha il vantaggio di mantenere invariato il valore dell'output nel caso questo sia maggiore di zero.

### 3.1.3 I limiti del perceptrone

Vediamo ora un esempio di utilizzo del perceptrone in modo da esplorarne in modo più approfondito le caratteristiche e analizzarne i limiti che hanno portato allo sviluppo di sistemi di perceptroni stratificati.

Supponiamo di voler implementare, tramite l'utilizzo di un perceptrone, alcune delle funzioni booleane come *AND*, *OR* e *NOT*. Implementare un operatore tramite l'utilizzo di un perceptrone significa trovare un insieme di pesi e una funzione

di attivazione tali che le uscite del perceptrone relative a determinati ingressi corrispondano alle uscite della funzione logica che vogliamo implementare relative agli stessi ingressi.

Poichè stiamo lavorando con operatori binari utilizzeremo come funzione di attivazione la *Funzione di Heaviside*, questo perché, seppur non più utilizzata, si presta bene alla nostra analisi ed è di facile implementazione:

---

```

1  def heaviside(x):
2      if x > 0: return 1
3      else: return 0

```

---

Una volta definita la funzione di attivazione andiamo a definire una classe che implementa (in modo elementare) il modello del perceptrone:

---

```

1  class Perceptron:
2
3      def __init__(self, weights, bias):
4          self.weights = weights
5          self.bias = bias
6
7      def predict(self, inputs):
8          out = np.dot(self.weights, inputs) + self.bias
9          out = heaviside(out)
10         return out

```

---

A questo punto abbiamo tutti gli strumenti necessari per procedere.

### Funzione NOT

La funzione NOT è un operatore binario descritto dalla seguente tabella di verità:

A	$\bar{A}$
0	1
1	0

e viene generalmente indicata con il simbolo  $\bar{A}$ . Si tratta di un operatore binario con un ingresso ed un'uscita ed è possibile implementarlo tramite l'utilizzo del perceptrone utilizzando i pesi  $w_1 = -1$  e  $b = 0.5$ :

---

```

1  NOT_p = Perceptron(-1, 0.5)
2
3  print("NOT 0 = {}".format(NOT_p.predict(0)))
4  print("NOT 1 = {}".format(NOT_p.predict(1)))

```

---

che fornisce il seguente output:

```

NOT 0 = 1
NOT 1 = 0

```

corrispondente appunto alla funzione NOT.

### Funzione AND

La funzione *AND* è un operatore booleano rappresentato dalla seguente tabella di verità:

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

e viene generalmente indicato come  $AB$  (questo anche per la somiglianza con la funzione prodotto). La funzione AND è una funzione a due ingressi e una uscita e, analogamente a quanto fatto in precedenza, possiamo implementarla con un perceptrone tramite l'utilizzo dei pesi  $w_1 = 1, w_2 = 1$  e  $b = -1.5$ :

---

```

1  AND_p = Perceptron([1,1], -1.5)
2
3  print("0 AND 0 = {}".format(AND_p.predict([0,0])))
4  print("0 AND 1 = {}".format(AND_p.predict([0,1])))
5  print("1 AND 0 = {}".format(AND_p.predict([1,0])))
6  print("1 AND 1 = {}".format(AND_p.predict([1,1])))

```

---

e produce il seguente output:

```

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

```

### Funzione OR

La funzione OR è un operatore binario rappresentato dalla seguente tabella di verità:

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

e viene generalmente indicato come  $A + B$  (anche in questo caso il simbolo ricorda la somiglianza fra l'operatore OR e l'operatore somma). La funzione OR



è una funzione a due ingressi e una uscita e può essere implementata tramite l'utilizzo di un perceptrone utilizzando i pesi  $w_1 = 1, w_2 = 2$  e  $b = -0.5$ :

---

```

1 OR_p = Perceptron([1,1], -0.5)
2
3 print("0 OR 0 = {}".format(OR_p.predict([0,0])))
4 print("0 OR 1 = {}".format(OR_p.predict([0,1])))
5 print("1 OR 0 = {}".format(OR_p.predict([1,0])))
6 print("1 OR 1 = {}".format(OR_p.predict([1,1])))

```

---

che producono l'output:

```

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

```

### Il problema dello XOR

Come abbiamo appena visto è possibile, tramite l'utilizzo del medesimo modello (perceptrone), implementare le funzioni booleane *AND*, *OR* e *NOT*. Tuttavia nel 1969 i matematici Marvin Minsky e Seymour Papert misero in mostra alcuni dei limiti del perceptrone. In particolare riuscirono a dimostrare che le classi di funzioni che il perceptrone era in grado di discriminare era limitata alle forme linearmente separabili [3].

Per comprendere meglio questo concetto riprendiamo l'equazione in forma vettoriale che descrive il perceptrone:

$$y = \phi(w^T \cdot x + b)$$

e supponiamo che la nostra funzione di attivazione  $\phi$  si di tipo gradino (anche in questo caso la scelta della funzione di heaviside viene effettuata solo per semplicità di calcolo, ma i risultati rimangono i medesimi per diverse funzioni di attivazione). Possiamo quindi riscrivere l'equazione del modello come:

$$y(x) = \begin{cases} 0 & \text{se } w^T \cdot x > -b \\ 1 & \text{se } w^T \cdot x < -b \end{cases}$$

Il perceptrone si comporta quindi come un *Classificatore binario*, ovvero divide le uscite in due classi separate (che in questo caso corrispondono ai valori binari *True* e *False*). Inoltre l'equazione  $w^T \cdot x = -b$  che divide le due classi rappresenta l'equazione caratteristica di un *Iperpiano*. L'algoritmo di apprendimento di un perceptrone, quindi, risulta essere del tutto analogo ad un algoritmo di *Regressione lineare*. Per comprendere meglio questo concetto analizziamo le funzioni fin'ora implementate. Supponiamo di visualizzare su un grafico le possibili coppie di ingressi di un perceptrone con due input:

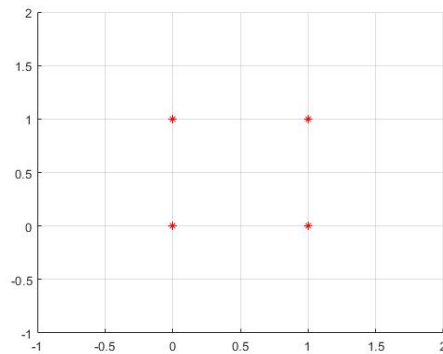


Figura 3.6: Disposizione ingressi sul piano cartesiano (da MATLAB)

Essendo il perceptrone analogo ad un algoritmo di *regressione lineare* modificare i suoi pesi equivale a modificare i coefficienti di una retta di equazione  $w_1x_1 + w_2x_2 = -b$  affinché divida gli ingressi in due classi distinte di output.

Per implementare la funzione AND avevamo utilizzato i seguenti pesi  $w_1 = 1$ ,  $w_2 = 1$  e  $b = -1.5$ , il che descrive una retta di equazione  $x_1 + x_2 = 1.5$  che, se rappresentata nel piano, fornisce la seguente classificazione:

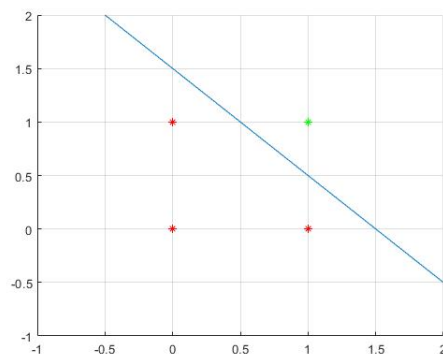


Figura 3.7: Retta caratteristica della funzione AND (da MATLAB)

Dove i punti verdi vengono classificati come 1 mentre quelli rossi come 0. Il che rappresenta esattamente la nostra funzione AND. Ragionando analogamente per la funzione OR otteniamo il grafico in fig. 3.8.

Tuttavia, se cerchiamo di implementare la funzione XOR, descritta dalla tabella di verità:

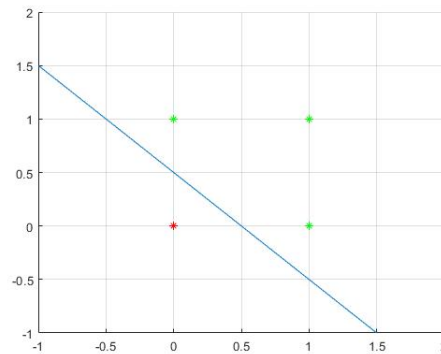


Figura 3.8: Retta caratteristica della funzione OR (da MATLAB)

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

tramite l'utilizzo di un perceptrone troveremo che nessuna terna di pesi  $(w_1, w_2, b)$  sarà in grado di rappresentare la funzione. Questo perché non esiste una singola retta in grado di classificare gli ingressi della funzione come descritto nella Fig. 3.9. Ovvero la funzione XOR non è linearmente separabile.

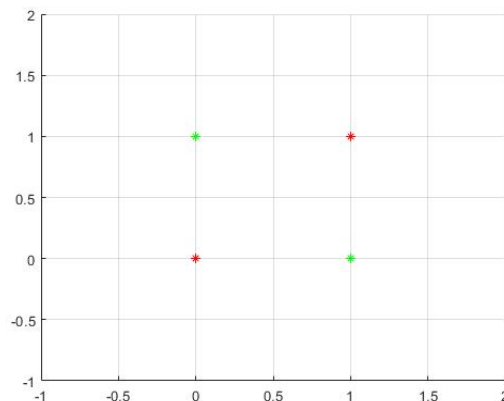


Figura 3.9: Classificazione ingressi per la funzione XOR (da MATLAB)

## 3.2 MLP o Reti Neurali

Il problema dello XOR e la discretizzazione lineare del perceptrone rappresentano un limite che non può essere superato senza l'introduzione del *Multi-layer Perceptron (MLP)* o *Perceptrone Multistrato*. Un perceptrone multistrato rappresenta la formalizzazione più generica possibile di una *Rete Neurale*. Il perceptrone, come abbiamo visto, presenta dei limiti nella discriminazione di funzioni relegate all'ambiente delle funzioni linearmente separabili, tuttavia il *MLP*, come vedremo, non soffre di questo svantaggio.

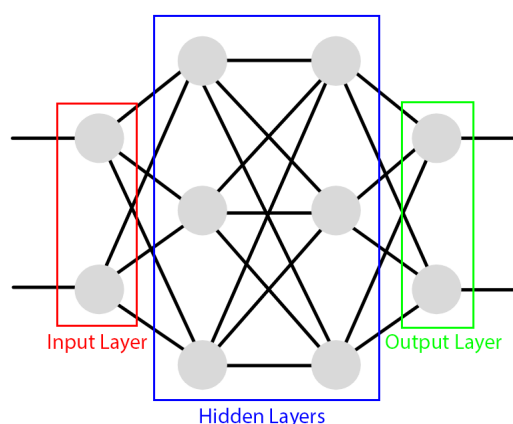


Figura 3.10: Struttura degli strati del MLP

Il perceptrone multistrato, come si deduce dal nome, è una rete interconnessa di perceptroni organizzata su vari livelli detti *strati*. Ogni MLP possiede almeno due strati distinti, uno strato di input e uno di output, ma, generalmente, possiede un numero arbitrario di strati intermedi detti *Hidden Layers*, come mostrato in Fig. 3.10. Sebbene i termini *Multi Layer perceptron* e *Rete Neurale* vengano molto spesso utilizzati come sinonimi si preferisce parlare di MLP per reti con due o tre strati mentre parleremo di reti Neurali per strutture con più strati. In linea generale, sulla base del tipo di interconnessione dei perceptroni fra i vari strati possiamo distinguere due principali categorie di MLP o reti neurali:

- **Fully Connected Networks (FCN)**

Una *Fully Connected Network* è una rete neurale dove ogni perceptrone di ogni livello è connesso con tutti i perceptroni del livello successivo. Queste reti presentano un elevato numero di pesi per ogni strato e vengono di solito utilizzate in applicazioni con input di dimensioni piccole

- **Locally Connected Networks (LCN)**

Le *Locally Connected Networks* sono reti neurali dove ogni perceptrone è connesso solo ad una parte dei perceptroni dello strato successivo. Questo implica

un minor numero di pesi per ogni strato e ne riduce di conseguenza i tempi computazionali. Sono largamente usate nei campi della *Computer Vision* e dell'elaborazione delle immagini dove i dati di input hanno tendenzialmente grandi dimensioni.

Possiamo inoltre, sempre sulla base de tipo di connessione, distinguere due ulteriori classi di reti neurali:

- **Reti Neurali Feed-Forward**

La *Reti Neurali Feed-Forward* sono reti neurali in cui le connessioni fra i vari percettroni scorrono solo in una direzione, ovvero, le uscite di uno strato rappresentano gli ingressi dello strato successivo e non sono presenti cicli.

- **Reti Cicliche**

Le *Reti Cicliche* sono reti neurali dove le connessioni fra i vari percettroni possono avvenire in entrambe le direzioni, ovvero, le uscite di uno strato possono essere gli ingressi dello strato successivo o anche dello strato precedente, in questo modo vengono a crearsi dei cicli all'interno della rete

### 3.2.1 Modello Matematico

Analogamente a quanto fatto per il percettrone andiamo ora a dare un modello matematico che definisca le uscite della rete neurale a partire dai suoi ingressi. Per poter procedere con l'analisi del modello dobbiamo fare un'ipotesi:

**Ipotesi 1** *I percettroni appartenenti allo stesso strato hanno tutti la stessa funzione di attivazione*

Sebbene questa ipotesi serva per semplificare l'analisi stessa del modello va precisato che si tratta di un'ipotesi sufficientemente generale. Questo perché, nella pratica, i percettroni dello stesso strato hanno effettivamente la stessa funzione di attivazione, generalmente tutta la rete sfrutta due differenti funzioni di attivazione, una per *Input Layer* e *Hidden Layers* e una per l'*Output Layer*.

A questo punto possiamo analizzare la rete partendo da un generico strato  $k$ . Ogni strato di una rete neurale può essere visto come un insieme di  $m$  percettroni collegati in parallelo con le uscite degli  $n$  percettroni precedenti. Ogni percettrone fornisce un'uscita del tipo:

$$y = \phi(w^T \cdot x + b)$$

Quindi l'uscita dello strato  $k$  sarà un vettore del tipo:

$$y_k = \begin{pmatrix} \phi_k(w_{k1}^T \cdot y_{k-1} + b_{k1}) \\ \phi_k(w_{k2}^T \cdot y_{k-1} + b_{k2}) \\ \vdots \\ \phi_k(w_{km}^T \cdot y_{k-1} + b_{km}) \end{pmatrix} \quad (3.1)$$

dove  $\phi_k$  rappresenta la funzione di attivazione del livello  $k$ ,  $w_{km}$  e  $b_{km}$  rispettivamente il vettore dei pesi e il bias del perceptrone  $m$ -esimo dello strato  $k$ . Possiamo semplificare questa scrittura andando ad introdurre due semplificazioni:

- **La matrice**  $W_k$  definita come la matrice dove la colonna  $j$ -esima rappresenta il vettore dei pesi del perceptrone  $j$ -esimo dello strato  $k$ :

$$W_k = \begin{vmatrix} w_1 & w_2 & \dots & w_m \end{vmatrix}$$

- **Il vettore**  $B_k$  definito come il vettore dei bias dello strato  $k$ -esimo (l'elemento di posto  $j$  rappresenta il bias del  $j$ -esimo perceptrone dello strato  $k$ ).

Utilizzando questi due elementi possiamo riscrivere la relazione 3.1 che descrive l'uscita dello strato  $k$  come:

$$y_k = \phi_k(W_k^T \cdot y_{k-1} + B_k) \quad (3.2)$$

Questa rappresentazione ci da delle informazioni strutturali della rete. Il numero di ingressi di ogni perceptrone dello strato  $k$  deve essere uguale al numero di perceptron presenti nello strato  $k - 1$ , altrimenti il prodotto  $W_k^T \cdot y_{k-1}$  non sarebbe possibile (quindi questo modello si applica bene solo alle reti *Feed-forward*), ma due strati adiacenti possono avere un numero differente di perceptron. Com'è possibile vedere dalla 3.2 l'uscita del  $k$ -esimo strato viene rappresentata in una forma ricorsiva (questo perché gli ingressi di una strato sono le uscite dello strato precedente). Il modello completo può essere ricavato semplicemente aggiungendo al modello l'uscita del primo livello che quindi risulta essere:

$$\begin{cases} y_1 = \phi_1(W_1^T \cdot x + B_1) \\ y_k = \phi_k(W_k^T \cdot y_{k-1} + B_k) \end{cases} \quad (3.3)$$

dove  $x$  rappresenta il vettore degli ingressi. Supponendo che la rete neurale abbia in totale  $k$  strati, l'uscita della rete in forma estesa è:

$$y = \phi_k(W_k^T \cdot [\phi_{k-1}(W_{k-1}^T \cdot \dots \cdot [\phi_1(W_1^T \cdot x + B_1)] + B_2)] + \dots + B_k) \quad (3.4)$$

### 3.2.2 Soluzione del problema dello XOR

L'introduzione del *Multilayer Perceptron* consente quindi di avere un modello in grado di discriminare classi di funzioni non linearmente separabili differentemente dal perceptrone. Vediamo quindi in che modo tramite l'utilizzo di *MLP* è possibile discretizzare la funzione XOR.

Utilizzando il modello del perceptrone utilizzato in precedenza possiamo implementare lo XOR come segue:

---

```

1 def XOR_predict(input):
2     layer1_p1 = Perceptron([1,1], -0.5)
3     layer1_p2 = Perceptron([-1,-1], 1.5)
4
5     layer1_out = np.array([layer1_p1.predict(input), layer1_p2.
6         predict(input)])
7
8     layer2_p1 = Perceptron([1,1], -1.5)
9
10    return layer2_p1.predict(layer1_out)
11
12 print("0 XOR 0 = {}".format(XOR_predict([0,0])))
13 print("1 XOR 0 = {}".format(XOR_predict([1,0])))
14 print("0 XOR 1 = {}".format(XOR_predict([0,1])))
15 print("1 XOR 1 = {}".format(XOR_predict([1,1])))

```

---

che produce il seguente output:

```

0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0

```

Dal codice possiamo notare che il *MLP* presenta i seguenti parametri caratteristici:

- *Input Layer*

$$W_1 = \begin{vmatrix} 1 & -1 \\ 1 & -1 \end{vmatrix}, B_1 = \begin{vmatrix} -0.5 \\ 1.5 \end{vmatrix}$$

- *Output Layer*

$$W_2 = \begin{vmatrix} 1 \\ 1 \end{vmatrix}, b = -1.5$$

Utilizzando il modello 3.2 possiamo quindi andare a vedere come si comporta l'uscita del primo strato del *Multi-layer perceptron*:

$$y_{l1} = \begin{vmatrix} \begin{cases} 1 & x_1 + x_2 > 0.5 \\ 0 & x_1 + x_2 < 0.5 \end{cases} \\ \begin{cases} 1 & x_1 + x_2 < 1.5 \\ 0 & x_1 + x_2 > 1.5 \end{cases} \end{vmatrix} = \begin{vmatrix} y_1 \\ y_2 \end{vmatrix}$$

Quello che accade in uno strato di un *Multi-Layer perceptron* o di una rete neurale è che ogni perceptrone appartenente a quello strato traccia una linea lungo il diagramma cartesiano indipendentemente dagli altri come possiamo vedere in figura 3.11

L'uscita complessiva del modello viene descritta come:

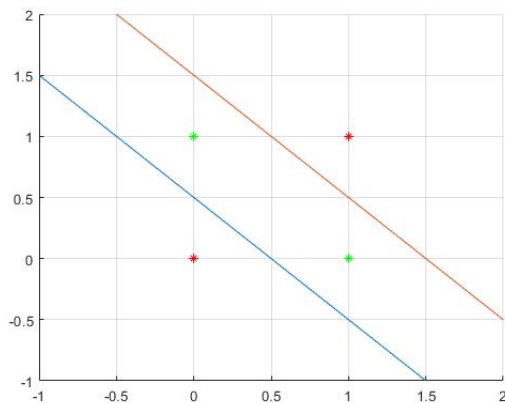


Figura 3.11: Diagramma del primo strato del Multi-Layer Perceptron (da MATLAB)

$$y = \begin{cases} 1 & y_1 + y_2 > 1.5 \\ 0 & y_1 + y_2 < 1.5 \end{cases}$$

Il compito del livello di output è quello di classificare le aree divise dai precedenti iperpiani in due classi distinte. L'uscita del modello è 1 solo se la somma delle uscite del livello di input è maggiore di 1.5, questo, essendo un sistema booleano è possibile solo se entrambi gli ingressi sono 1, ovvero nella sezione del grafico cartesiano fra le due rette, che è esattamente quello che descrive graficamente la funzione XOR.

### 3.2.3 Back-propagation

Fin'ora abbiamo utilizzato, nella trattazione degli esempi legati ai perceptor e alle reti neurali, un insieme di pesi e bias prestabiliti. La semplicità dei problemi proposti consentiva di ricavare analiticamente questi valori, tuttavia i *MLP* e le *Reti neurali* sono algoritmi di *Machine Learning* e quindi ci aspettiamo che siano in grado, tramite un adeguato processo di addestramento, di aggiornare automaticamente i loro pesi in modo da ottimizzare i risultati ottenuti.

La fase di addestramento di una rete neurale viene divisa in due sezioni distinte:

- *Model evaluation.* Durante questa fase i pesi e i bias della rete neurale vengono mantenuti invariati e la rete effettua operazioni di predizione su una parte del dataset di riferimento. Questa sezione viene definita *Batch* ed è una quantità arbitraria, dipende dal tipo di modello, di hardware, ecc.
- *Model training.* Una volta terminata la fase di *Model evaluation* la rete ha a disposizione una serie di valori predetti e rispettive predizioni desiderate (il numero dipende dalla dimensione del *batch*). In questa fase gli input e gli output vengono tenuti invariati e la rete effettua delle operazioni di aggiornamento dei pesi e dei bias



Queste due operazioni vengono ripetute ciclicamente fino al termine del dataset di addestramento. Una volta giunti a questo punto diremo che la rete neurale ha compiuto un'epoca o *epoch* di addestramento. La dimensione del *batch* e il numero di *epoch* di addestramento rappresentano due iperparametri che influenzano il modo in cui la rete apprende e verranno trattati più approfonditamente dopo aver descritto la *discesa del gradiente*.

### Discesa del gradiente

Abbiamo detto che nella fase di *Model training* i pesi e i bias della rete neurale vengono aggiornati in modo da ottimizzare i risultati ottenuti. Questo processo di aggiornamento dei pesi avviene grazie ad un algoritmo detto *Discesa del gradiente* o *Gradient descent* che mira a minimizzare il valore di una funzione detta *funzione di costo* o *perdita*. Sappiamo che la rete neurale ha a disposizione un insieme di uscite predette dalla rete e le corrispondenti uscite desiderate. Possiamo quindi andare a definire la funzione di costo come l'errore medio commesso nella valutazione dei dati, ovvero:

$$C(w, b) = \frac{1}{2n} \sum_{i=1}^n \|y_{di} - y_i\|^2 \quad (3.5)$$

dove  $n$  è il numero di elementi nel *batch*,  $y_{di}$  è l'uscita desiderata per l'ingresso  $i$ -esimo e  $y_i$  è l'uscita predetta dalla rete per l'ingresso  $i$ -esimo. Il parametro  $y_i$  è dipendente dai parametri  $w$  e  $b$  come sappiamo. Questa funzione quindi rappresenta l'errore medio commesso nella predizione del *batch* e, di conseguenza, l'obiettivo è quello di minimizzare il valore di questa funzione (tanto più  $C$  è vicina a zero tanto più la rete sarà accurata nelle predizioni).

questa funzione prevede un numero di parametri arbitrario, ma che, nelle reti moderne, è tendenzialmente molto elevato. trovare un minimo di questa funzione numericamente può quindi diventare piuttosto complicato. L'algoritmo di *Discesa del gradiente* consente di trovare il minimo di questa funzione in modo iterativo. consideriamo la seguente relazione:

$$\frac{\partial C}{\partial w_1} \Delta w_1 + \dots + \frac{\partial C}{\partial w_n} \Delta w_n = \nabla C \cdot \Delta W \quad (3.6)$$

dove per semplicità di scrittura abbiamo incluso i bias all'interno dei pesi. La quantità  $\nabla C$  viene detto *gradiente* e rappresenta un vettore che punta sempre nella direzione di massima crescita della funzione  $C$ . La quantità  $\Delta w$  rappresenta invece una variazione molto piccola della matrice dei pesi  $W$ . Inoltre il *Teorema della derivate parziali* ci dice che:

$$\Delta C = \nabla C \cdot \Delta W \quad (3.7)$$

ovvero ci dice che la variazione della funzione di costo è uguale al prodotto scalare del gradiente della funzione per la variazione della matrice dei pesi. Visto

che la variazione della matrice dei pesi è quello che dobbiamo scegliere possiamo fare in modo di scegliere un  $\Delta W$  tale che la variazione della funzione di costo sia negativa (e quindi l'errore diventi più piccolo). In particolare se scegliamo:

$$\Delta W = -\mu \nabla C \quad (3.8)$$

dove il termine  $\mu$  viene detto *Learning rate*. Sostituendo questo risultato nell'eq. 3.7 otteniamo:

$$\Delta C = -\mu \nabla^2 C \quad (3.9)$$

In questo modo siamo sicuri che la variazione della funzione di costo sia negativa (poiché si muove nella direzione opposta del gradiente della funzione) e, iterando ad ogni ciclo, possiamo ridurla fino a giungere al minimo della funzione stessa.

### Valutazione degli iperparametri

Nell'analisi della *back-propagation* fin qui abbiamo messo in evidenza alcuni parametri che abbiamo detto essere arbitrari, ovvero il *Learning rate*, la dimensione del *batch* e il numero di *epoch*. Vediamo quindi come questi parametri influenzano l'apprendimento del modello.

- **Learning Rate  $\mu$**

Il *Learning rate* è un iperparametro che ci dice quanto velocemente la funzione di costo convergerà (o si muoverà verso) il suo minimo. La prima ipotesi che potremmo fare è quella di prendere il parametro più grande possibile per far convergere la funzione al minimo nel minor tempo possibile. Tuttavia questa scelta non è ottima poiché il valore della funzione di costo potrebbe mettersi ad oscillare attorno al punto di minimo senza mai giungere ad esso. D'altro canto, se il *Learning rate* viene scelto troppo piccolo la funzione potrebbe convergere al punto di minimo troppo lentamente ed inoltre potrebbe fermarsi in un punto di minimo locale ignorando eventuali punti di minimo globale. Generalmente il valore del learning rate viene aggiornato ad ogni step secondo una tabella detta *schedule* in base alla vicinanza della funzione di perdita dal minimo.

- **Dimensione del batch** La dimensione del batch è un parametro che ci dice ogni quanto viene effettuato l'aggiornamento dei parametri della rete. Questo parametro presenta dei limiti ben precisi e in base a questi limiti possiamo distinguere tre classi:

- **Batch Gradient Descend**

In questo caso la dimensione del batch coincide con la dimensione dei dati di input. Questo significa che i pesi della rete neurale vengono aggiornati una volta ogni epoca. Questo sistema presenta il miglior andamento della curva di costo poiché valuta la discesa del gradiente relativamente all'errore commesso su tutto il dataset.

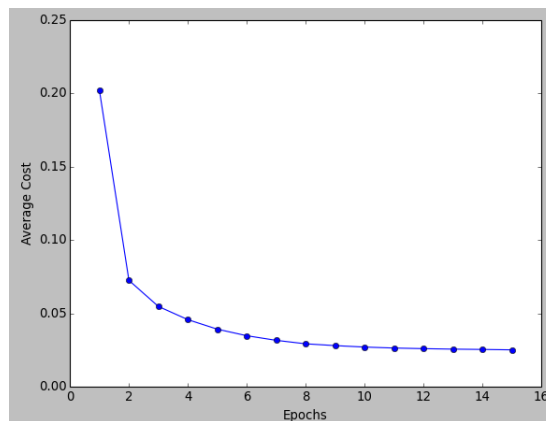


Figura 3.12: Andamento della funzione di costo nel Batch Gradient Descent [4]

Tuttavia nel caso di dataset di grandi dimensioni questo algoritmo impiega molto tempo per il calcolo relativo all'aggiornamento dei valori (deve calcolare il gradiente per ogni coppia di valori del dataset) e di conseguenza rappresenta un utilizzo poco ottimizzato della backpropagation.

#### – Stochastic Gradient Descent

In questa situazione invece la dimensione del batch viene presa uguale ad uno. L'aggiornamento dei parametri avviene quindi dopo ogni predizione della rete neurale. Questo sistema consente di migliorare i tempi di apprendimento per dataset di grandi dimensioni. Calcolando il gradiente per ogni coppia di valori la funzione di perdita non sarà costante (tenderà a scendere e salire), ma complessivamente questa tenderà a diminuire.

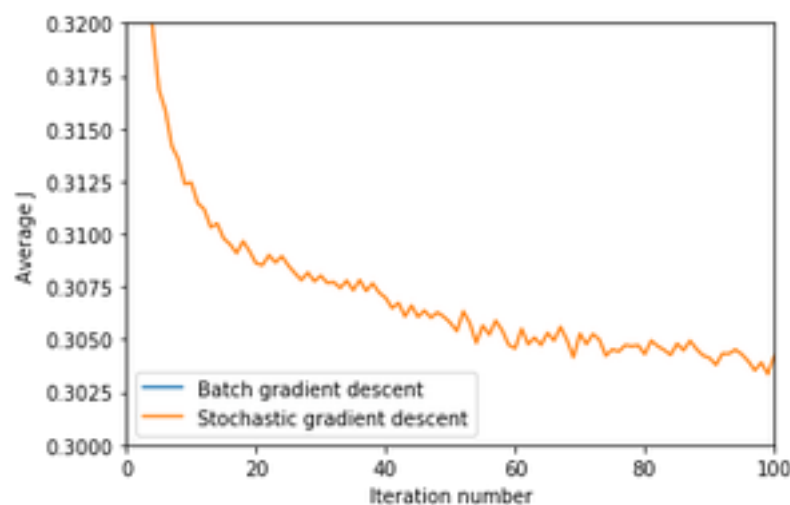


Figura 3.13: Andamento della funzione di costo nello Stochastic gradient descent [5]

**Mini-batch Gradient Descend** Questo algoritmo rappresenta un'unione dei due algoritmi precedenti. La dimensione del batch viene scelta ad un numero arbitrario più piccolo della dimensione del dataset stesso. Questo implica che i parametri della rete vengono aggiornati un numero di volte pari a  $size_{data}/size_{batch}$ . Questo algoritmo presenta i principali vantaggi dei precedenti, una funzione di perdita che converge più velocemente a zero e tempi di esecuzione ridotti. La dimensione del batch è arbitraria anche se generalmente oscilla da qualche decina fino a poche centinaia di elementi.

- **Numero di epoche** Il Numero di epoche è un parametro che definisce per quanto la rete neurale dovrà allenarsi sul dataset di allenamento. Anche in questo caso si potrebbe pensare di utilizzare un numero di epoche molto elevato in modo da permettere alla rete neurale di ottimizzare i propri pesi e minimizzare la funzione di costo. Tuttavia un numero di epoche troppo elevato può portare a problemi di **overfitting** ovvero la situazione in cui la rete diventa troppo sensibile alle variazioni parametriche e di conseguenza, sebbene fornisca ottimi risultati sui dati di allenamento, perde accuratezza nell'applicazione generale. In pratica la rete non riesce più a generalizzare il problema e riesce a lavorare correttamente solo su specifici set di dati. Il caso opposto invece è rappresentato dall'**underfitting** ovvero l'eccessiva generalità della rete dovuta ad un allenamento su dataset ristretti o, appunto, ad un numero di epoche di allenamento insufficienti.

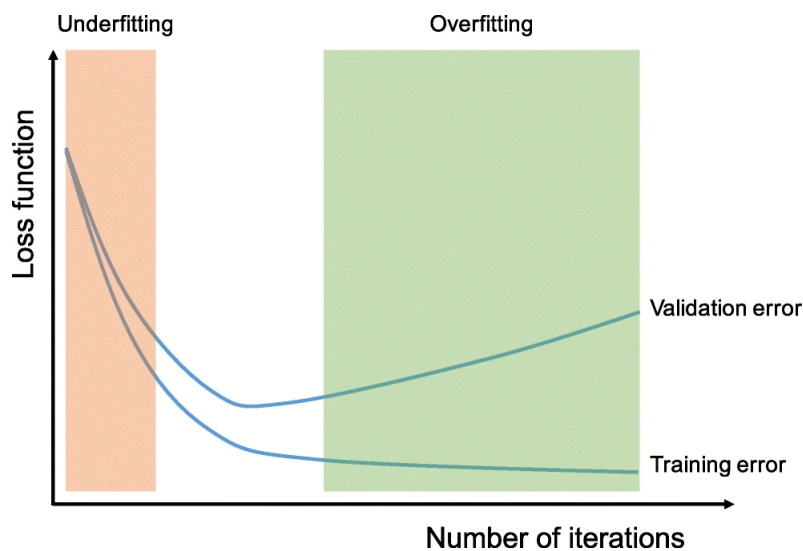


Figura 3.14: Zone di underfitting e overfitting relativamente alla funzione di costo [6]



# Capitolo 4

## Reti Neurali Convolutive

Particolare attenzione verrà dedicata in questo capitolo alle *Reti Neurali convolutive* o *CNN* (acronimo dall'inglese *Convolutional Neural Network*). Le reti neurali convolutive (CNN) sono una classe di reti neurali artificiali divenute dominanti in vari compiti di visione artificiale e che stanno suscitando interesse in una varietà di domini molto estesa. La CNN è progettata per apprendere automaticamente e in modo adattivo le gerarchie spaziali delle funzionalità tramite la *back-propagation* utilizzando più blocchi predefiniti, come livelli di convoluzione, livelli di pooling e livelli completamente connessi [6].

Queste reti sono quindi pensate e ottimizzate per lavorare con immagini e rappresentano, come vedremo più avanti, una sottoclasse delle *Locally Connected Network* o LCN con una struttura bidimensionale di ogni strato. La loro particolare struttura le differenzia dalle reti fin qui viste non solo nelle operazioni, ma anche nel modello stesso che le rappresenta, più complesso da analizzare ma molto più efficiente se si lavora con un elevato numero di dati e che riesce ad ottenere ottimi risultati utilizzando, a parità di dimensione di input, un numero di pesi minore rispetto alle reti *Fully Connected*.

Grazie a questo le *Convolutional Neural Networks* sono ampiamente utilizzate in molti dei principali campi dell'intelligenza artificiale e, come poteva essere scontato pensare, sono principalmente note per il loro utilizzo nel campo della *Computer Vision*.

### 4.1 Introduzione alla Computer Vision

La *Computer Vision* o *CV* è l'insieme dei processi che mirano a creare un modello approssimato del mondo reale tridimensionale partendo da immagini bidimensionali. Lo scopo principale della visione artificiale è quello di riprodurre la vista umana. Vedere è inteso non solo come l'acquisizione di una fotografia bidimensionale di un'area ma soprattutto come l'interpretazione del contenuto di quell'area. L'informazione è intesa in questo caso come qualcosa che implica una decisione automatica[7].

Possiamo quindi vedere la *Computer Vision* come la branca dell'intelligenza artifi-

ciale che cerca di riprodurre artificialmente il sistema visivo biologico di un essere umano. Non è un caso che la struttura delle *CNN*, che come detto sono largamente utilizzate in questo campo, sia modellata a partire dalle reti biologiche di alcuni animali delegate alla percezione visiva.

Sebbene la disciplina sia molto ampia questa si pone fundamentalmente di fronte a due distinti problemi:

- **Classificazione**

I problemi di classificazione nella *CV* sono problemi legati al riconoscimento di un determinato oggetto all'interno di un'immagine. Vogliamo quindi che, data un'immagine, la rete sia in grado di dirci cosa è presente all'interno di quell'immagine.



CAT

Figura 4.1: Esempio di classificazione [8]

- **Localizzazione**

I problemi di *localizzazione* sono problemi legati invece al posizionamento di un determinato oggetto all'interno di un'immagine.



Figura 4.2: Esempio di localizzazione[8]

Questo rispecchia pienamente il sistema visivo umano. Tutto quello di cui si occupa la visione umana è di *classificare e localizzare* gli oggetti che ci circondano definendone confini e posizioni. L'unione di questi due problemi e la *granularità* con cui si vuole applicare la localizzazione (a livello locale o a livello dei singoli pixel) ci permette di distinguere due principali classi di problemi: **Object Detection** e **Image Segmentation**.

### 4.1.1 Object Detection

l'*Object Detection* è un termine generico per descrivere una raccolta di attività di visione artificiale correlate che implicano l'identificazione di oggetti nelle fotografie digitali. La classificazione delle immagini implica la previsione della classe di un oggetto in un'immagine. La localizzazione degli oggetti si riferisce all'identificazione della posizione di uno o più oggetti in un'immagine e al disegno di un riquadro intorno al loro contorno. Il rilevamento degli oggetti combina queste due attività e localizza e classifica uno o più oggetti in un'immagine. [9]

I principali elementi coinvolti nei processi di *object detection* sono le *classi* e quelli che vengono definiti *Bounding Box*. Le classi sono, come è facile capire, i possibili oggetti che la rete deve essere in grado di riconoscere all'interno di un'immagine. Sebbene sia possibile rimuovere queste classi trasformando il problema dell'*object detection* in un problema di *clustering* questa tecnica è utilizzata molto di rado (prevedendo anche sistemi di analisi e di allenamento delle reti differenti) e non rientra negli obiettivi di questo scritto. I *Bounding Box* sono l'elemento distintivo dell'*object detection*. Si tratta di sezioni della superficie dell'immagine circoscritte e descritte da tre elementi: un punto, una larghezza e un'altezza. Questi box vengono utilizzati per localizzare gli oggetti all'interno delle immagini.

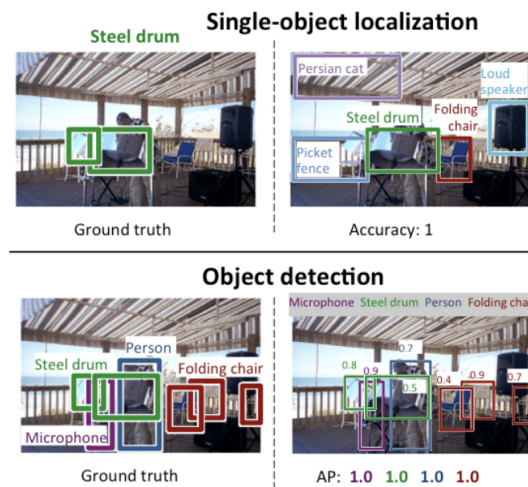


Figura 4.3: Esempio di bounding box in processi di object detection e single-object detection [9]

Si tende generalmente a distinguere la *single-object detection* dalla *object detection*, ma questo principalmente perché la prima rappresenta una semplificazione della seconda ed è possibile ottenere modelli accurati anche con reti più semplici, ma a livello teorico rappresentano lo stesso problema.

Alla luce di quanto detto è facile immaginare come possa essere strutturato l'output di una rete neurale in questo campo. In base al numero di oggetti rilevati l'output sarà costituito da una serie di vettori (uno per ogni oggetto) contenenti quattro elementi distinti: un numero che identifica la classe e tre valori che identificano il *bounding box*.



### 4.1.2 Image Segmentation

l'*Image Segmentation* rappresenta una classe avanzata di *object detection*. Sappiamo che un'immagine non è altro che una raccolta di pixel. L'*image segmentation* è il processo di classificazione di ogni pixel in un'immagine appartenente a una certa classe[8].

Sebbene, come abbiamo detto, questo problema rientri nella classificazione degli oggetti presenti all'interno di un'immagine, questo si pone un obiettivo in più, oltre a classificarli e localizzarli all'interno dell'immagine pone la sua attenzione a livello dei singoli pixel andando a distinguere quindi i contorni veri e propri degli elementi rilevati. In questo ambito il modo in cui la rete discrimina gli elementi rilevati ci



Figura 4.4: Esempio di *Image segmentation* [8]

permette di distinguere due classi differenti di segmentazione dell'immagine:

- **Semantic Segmentation**

La segmentazione semantica è il processo di classificazione di ogni pixel appartenente a una particolare etichetta senza distinguere fra le varie istanze della stessa etichetta [8]. Quindi se ad esempio il compito della nostra rete è quello di localizzare le auto questo applicherà la stessa etichetta per tutte le macchine, senza distinguerne il numero.

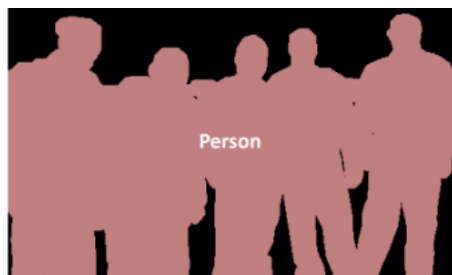


Figura 4.5: Esempio di segmentazione semantica

- **Instance Segmentation**

La segmentazione delle istanze differisce dalla segmentazione semantica poiché fornisce un'etichetta univoca a ogni istanza di un particolare oggetto nell'immagine.[8]

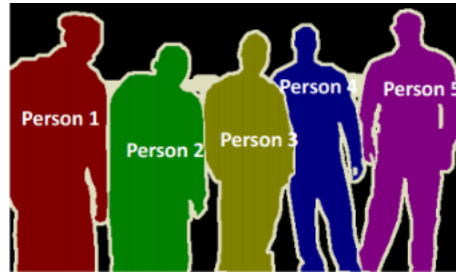


Figura 4.6: Esempio di segmentazione per istanza

Nei processi di segmentazione dell'immagine l'output previsto, a differenza di quanto detto per l'*object detection*, non è un insieme di vettori, ma una nuova immagine i cui pixel rappresentano una mappa della posizione, dei contorni e della classificazione degli elementi rilevati. Questa distinzione nei tipi di uscite delle reti ci permette di capire quanto questi problemi siano differenti fra loro e, di conseguenza, come non sia possibile utilizzare lo stesso modello per la soluzione dei due problemi.

## 4.2 Tensori

Prima di procedere con l'analisi strutturale delle reti neurali convolutive è necessario introdurre il concetto di *Tensor*. Il tensore è una nozione matematica appartenente al campo dell'algebra lineare. Viene generalmente definito a partire da uno spazio vettoriale e un campo, tuttavia, ai fini dell'analisi dei modelli delle *CNN* è più utile introdurli come particolari strutture dati. Un tensore risulta essere un'estensione *n-dimensionale* del concetto di array.

I tensori vengono identificati da una serie di parametri particolari:

- **Forma**, ovvero il numero di elementi presenti in ogni asse del tensore
- **Rango**, ovvero il numero di assi del tensore
- **Dimensione**, il numero di elementi totali nel tensore

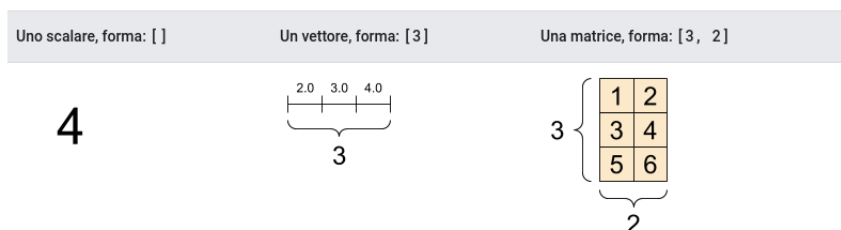


Figura 4.7: rappresentazione tensori elementari [10]

In particolare il rango e la forma ci danno le informazioni strutturali del tensore. Un tensore di rango 0 rappresenta semplicemente uno scalare, un tensore di rango 1 rappresenta un vettore e la forma ci dice quanti elementi sono presenti al suo interno. Analogamente un tensore di rango 2 rappresenta una matrice la cui forma ne specifica la dimensione (fig. 4.7).

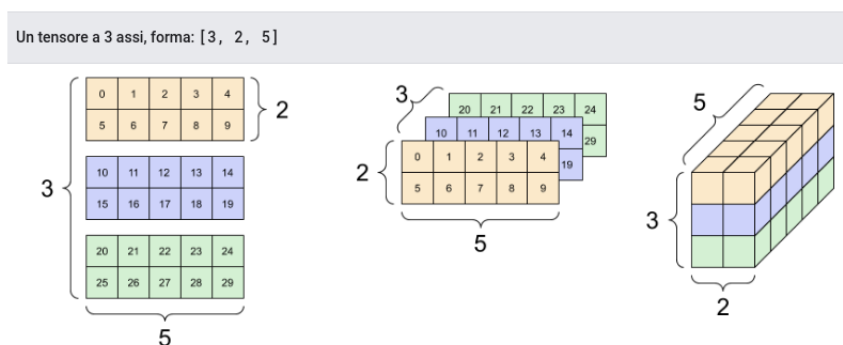


Figura 4.8: Esempi di differenti forme dei tensori [10]

I tensori con rango maggiore di due, a differenza dei precedenti, non presentano rappresentazioni standard e possono essere intesi in modi differenti sulla base del loro utilizzo (fig. 4.8). Queste strutture dati consentono di rappresentare in maniera compatta particolari input delle reti e le operazioni ad esse connesse. Inoltre le reti possono essere rappresentate come operatori lineari che lavorano con tensori in ingresso e in uscita.

Un particolare esempio di tensore sono le immagini. Sebbene queste siano rappresentate da due dimensioni spaziali (larghezza e altezza) che identificano una matrice di pixel, ogni pixel è rappresentato a sua volta da una terna di valori che identificano i colori RGB. Quindi in generale un'immagine RGB è un tensore di forma [larghezza, altezza, 3].

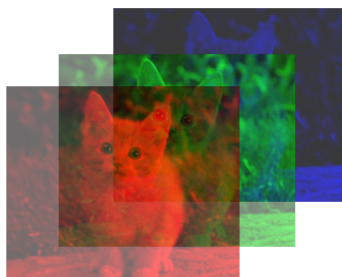


Figura 4.9: Canali RGB di un'immagine

## 4.3 CNN (Convolutional Neural Networks)

Le reti neurali convolutive sono molto simili alle reti neurali ordinarie viste in precedenza: sono costituite da neuroni che hanno pesi e bias allenabili. Ogni neurone riceve alcuni input, esegue un prodotto scalare e lo trasforma tramite una funzione di attivazione. L'intera rete esprime ancora un'unica funzione: dai pixel dell'immagine grezza su un'estremità ai punteggi delle classi dall'altra. E hanno ancora una funzione di perdita sull'ultimo livello e tutti i modelli sviluppati per l'apprendimento sono ancora validi.

Allora cosa cambia? Le architetture *ConvNet* presuppongono esplicitamente che gli input siano immagini, il che ci consente di codificare determinate proprietà nell'architettura. Queste quindi rendono la rete più efficiente per implementare e ridurre notevolmente la quantità di parametri[11].

Come abbiamo visto in precedenza le reti *Fully Connected* sono reti che prendono in input un tensore e lo trasformano, tramite una serie di *hidden layers* in un altro tensore in uscita. Ogni strato della rete è composto di perceptron indipendenti connessi interamente con tutti i perceptron dello strato precedente. Queste reti, sebbene estremamente funzionali, non si rapportano bene a lavorare con le immagini. Se supponiamo di avere in ingresso un'immagine di piccole dimensioni (ad esempio  $32 \times 32 \times 3$ ) i perceptron del primo strato dovranno avere un tensore di input di dimensioni analoghe, questo significa che ognuno di loro avrà esattamente 3072 pesi allenabili ( $32 \times 32 \times 3 = 3072$ ) e, supponendo di avere solo 10 perceptron sul primo strato (tendenzialmente se ne hanno molti di più) i pesi del primo strato saranno 30072, e questo per un'immagine di piccole dimensioni.

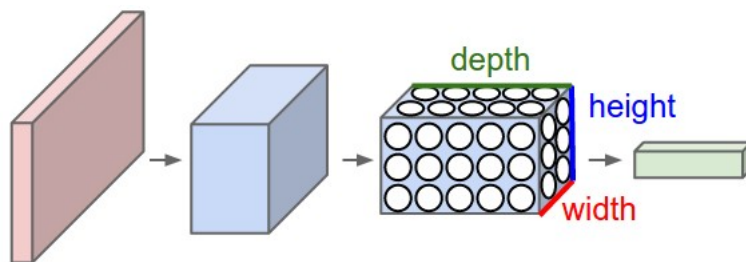


Figura 4.10: Differenza nella "forma" delle reti FCN e CNN [11]

A livello strutturale le *ConvNet*, diversamente dalle reti *FCN*, sono delle reti strutturate tridimensionalmente, ogni strato è quindi costituito da matrici di perceptron e presentano un'altezza, una larghezza e una profondità (dove con profondità si intende la dimensione spaziale e non il numero di strati). Le *ConvNet* sono, fondamentalmente, un particolare tipo di *Locally Connected Networks*, questo implica che ogni neurone appartenente ad uno strato non è completamente connesso con lo strato precedente o con gli input. Questo meccanismo rispecchia il sistema visivo biologico di alcuni animali dove i neuroni che si occupano di processi di visione non analizzano l'immagine in se, ma solo una sua piccola porzione

e, tramite l'utilizzo di neuroni disposti diversamente, permettono di analizzare l'intera immagine.

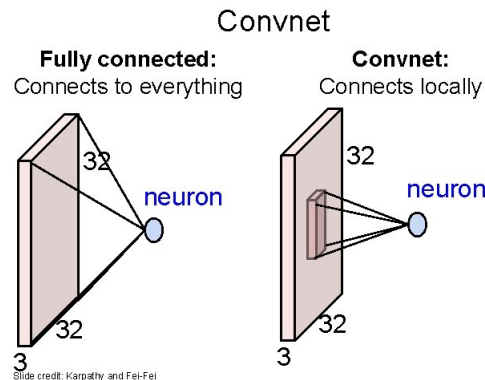


Figura 4.11: Differenza nelle connessioni dei perceptor in reti FCN e ConvNet [12]

Inoltre, ogni strato di una *ConvNet* può essere scomposto in "sotto-strati" (sebbene utilizzeremo il nome "sotto-strati" va precisato che solo uno di questi è effettivamente composto da perceptor, gli altri eseguono semplicemente delle operazioni sugli output dello strato principale, ma a livello teorico vale la pena analizzarli separatamente per comprenderne meglio il funzionamento e l'utilizzo)

- Convolutional Layer
- Activation Layer
- Pooling

A questi tre livelli, nelle *ConvNet*, si aggiunge un **Fully Connected Layer** come strato di output della rete.

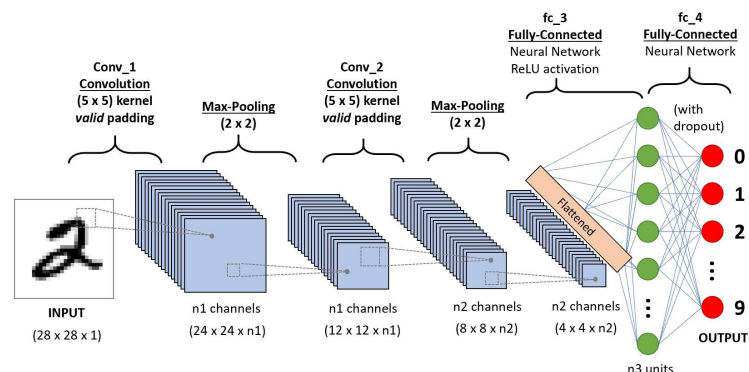


Figura 4.12: Distribuzione degli strati nelle ConvNet

### 4.3.1 Convolutional Layers

Gli strati convolutivi rappresentano il sistema caratteristico delle *ConvNet*. Come abbiamo detto questi strati sono formati da matrici di neuroni localmente connessi agli input o agli strati precedenti. Queste connessioni vengono organizzate utilizzando una serie di parametri specifici detti *Sliding window*, *Stride* e *Zero Padding*. La **Sliding Window** rappresenta la dimensione e la forma della porzione di tensore cui il perceptrone viene connesso. Questo parametro è costante per tutti quanti i perceptroni ed è un parametro fondamentale per definire la dimensione del tensore dei pesi del perceptrone (che viene generalmente detto *Kernel*)

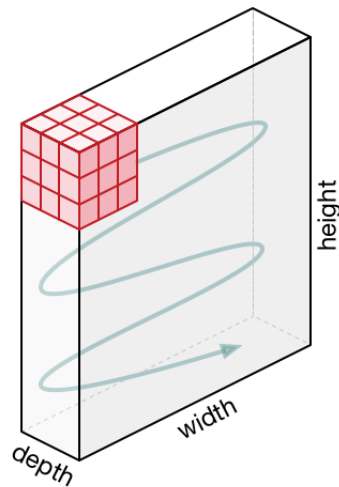


Figura 4.13: rappresentazione della sliding window su un tensore [13]

Questa sezione di tensore, come suggerisce il nome, viene fatta scorrere lungo tutto il tensore di ingresso e ogni porzione del tensore di ingresso corrispondente ad uno spostamento della *sliding window* rappresenta gli ingressi di un differente perceptrone. Il numero di elementi (o pixel nel caso di immagini) che ne definisce lo spostamento viene detto **Stride**.

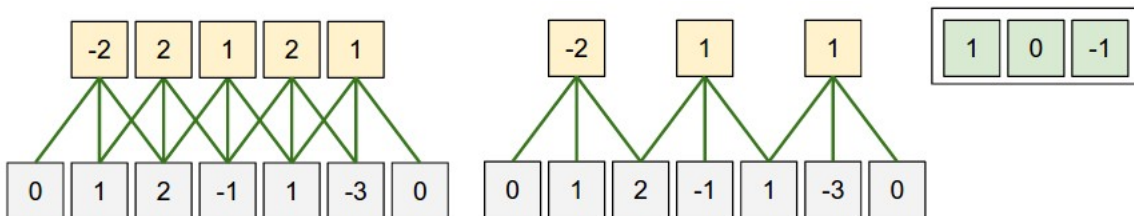


Figura 4.14: Visualizzazione di differenti tipi di stride

Supponiamo di avere un vettore di forma  $[1, 7]$  (considereremo un vettore solo per semplicità, ma in generale questo procedimento avviene con sliding window tridimensionali) con una *sliding window* di forma  $[1, 3]$ . nella Fig. 4.14 possiamo

vedere il comportamento per la rete relativo a differenti valori di stride. Nella figura a sinistra abbiamo uno stride unitario a cui corrispondono 5 sezioni del vettore e, quindi, 5 percettroni nello strato di input. Nel secondo caso invece lo stride viene posto a due, abbiamo quindi 3 sezioni di vettore e altrettanti percettroni nello strato di input.

La forma e dimensione della sliding window e lo stride sono quindi parametri che ci consentono di definire, a partire dalla forma e dimensione del tensore di input, la forma e dimensione del tensore di uscita di un determinato strato. Tuttavia, sebbene le ConvNet effettuino effettivamente operazioni di ridimensionamento dell'immagine, si preferisce che il tensore di uscita dal *Convolutional Layer* abbia la stessa forma del tensore di ingresso (Questo perché, per evitare perdita di informazione durante le operazioni di convoluzione, il ridimensionamento del tensore avverrà nel livello di *Pooling*). Per ottenere questo risultato si utilizza un ulteriore parametro caratteristico detto **Zero Padding**.

Il compito del parametro *Zero Padding* è fondamentalmente quello di aumentare la dimensione del tensore di ingresso aggiungendo degli zeri nei suoi contorni.

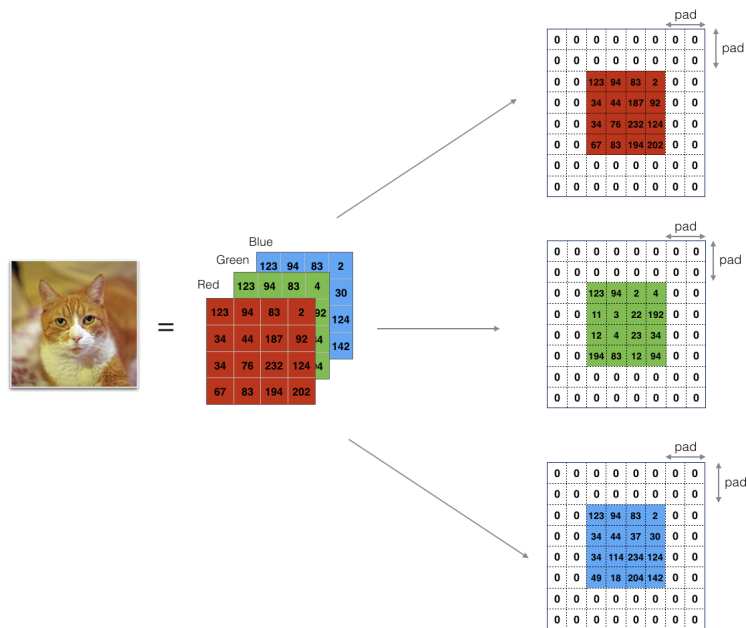


Figura 4.15: Visualizzazione del padding di immagini RGB

La dimensione del tensore di output a questo punto può essere calcolata come segue:

$$\frac{W - F + 2P}{S - 1} \quad (4.1)$$

dove  $W$  rappresenta la dimensione del tensore di input,  $F$  la dimensione della sliding window,  $P$  lo zero padding e  $S$  lo stride.

### Kernel e Filters

I pesi dei perceptron all'interno delle ConvNet sono organizzati in strutture particolari dette *Kernel*. Il kernel di un perceptrone è un tensore di pesi che presenta la stessa forma della *sliding window*. Questo assunto risulta fondamentale poiché il kernel rappresenta l'insieme dei pesi moltiplicativi di un dato perceptrone e, affinché sia possibile il prodotto scalare fra due tensori, questi devono avere la medesima forma.

Sebbene il kernel debba condividere la forma della *sliding window* è possibile pensare di "allineare" una serie di kernel, producendo quindi un tensore con una profondità maggiore (il che non va in contrasto con quanto detto precedentemente poiché il prodotto avverrà sempre kernel per kernel) in modo da produrre un tensore di output strutturato diversamente (approfondiremo il discorso quando analizzeremo il modello funzionale delle ConvNet) Queste strutture vengono invece definite **Filtri**.

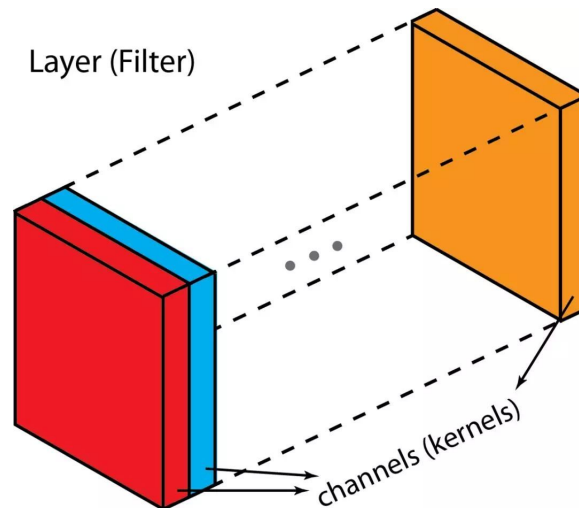


Figura 4.16: Illustrazione del rapporto fra kernel e filtri [14]

La differenza tra filtro e kernel è quindi sottile. Tuttavia, Un "kernel" tende ad essere una matrice di dimensione  $N$  mentre "filtro" si riferisce alla struttura di dimensione  $N+1$  di più stack del kernel. Ciascun kernel di convoluzione è unico e ha il principale compito di enfatizzare diversi aspetti del canale di input [14].

### Condivisione dei parametri

Un'ulteriore step nella riduzione del numero di parametri addestrabili della rete viene effettuato con la *condivisione dei parametri*. Ogni kernel del filtro di uno strato convolutivo è un recettore specializzato nel riconoscere determinati pattern all'interno della sezione del tensore in analisi (contorni, colori, superfici, ecc.). Quindi se un kernel di un perceptrone è in grado di riconoscere un pattern specifico in una sezione del tensore è molto probabile che sia in grado di farlo anche in una sezione differente.



In generale i percettroni appartenenti al medesimo strato condividono tutti lo stesso filtro, questo consente di evitare ripetizioni e ridondanza nei parametri addestrabili migliorando i tempi e le prestazioni della *back-propagation* che a questo punto dovrà occuparsi di un numero minore di pesi.

### 4.3.2 Pooling

I *Pooling Layer* sono degli strati inseriti generalmente fra uno strato convolutivo e l'altro, anche se è possibile inserire questi strati in sezioni differenti della rete, e hanno il compito di ridurre la dimensione spaziale dei tensori in uscita dai vari livelli. Il pooling agisce tuttavia solo sulle dimensioni trasversali dei tensori lasciandone inalterata la profondità (Fig.4.17).

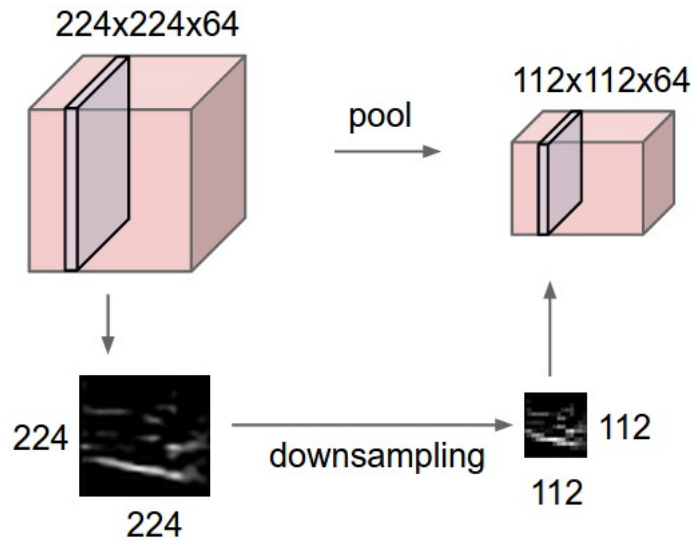


Figura 4.17: Downsampling di un tensore tramite Pooling [6]

Queste operazioni sono fondamentali sia per ridurre il numero di parametri addestrabili della rete (e quindi migliorare le prestazioni), ma anche per comprimere l'informazione in aree più piccole rimuovendo i dati ridondanti. Anche in questo caso queste operazioni si basano su due parametri ovvero una *sliding window* e uno *stride* che sono esattamente analoghi ai parametri visti negli strati convolutivi. Gli strati di pooling quindi:

1. prendono in input un tensore di forma  $[W_1, H_1, D_1]$
2. a partire da una *sliding window* di dimensione  $F$  e uno stride  $S$  producono in uscita un tensore di forma:

$$\left[ \frac{W_1 - F}{S + 1}, \frac{H_1 - F}{S + 1}, D_1 \right] \quad (4.2)$$

Esistono diverse tipologie di pooling o, più precisamente, esistono diversi modi in cui il pooling estrae informazioni relative alla *sliding window*, ma quello maggiormente utilizzata è il **Max Pooling**.

### Max Pooling

Il *Max Pooling* è un'operazione di downsampling dell'immagine che basa la sua valutazione sul valore massimo presente all'interno di una determinata area del tensore definita dalla *sliding window*.

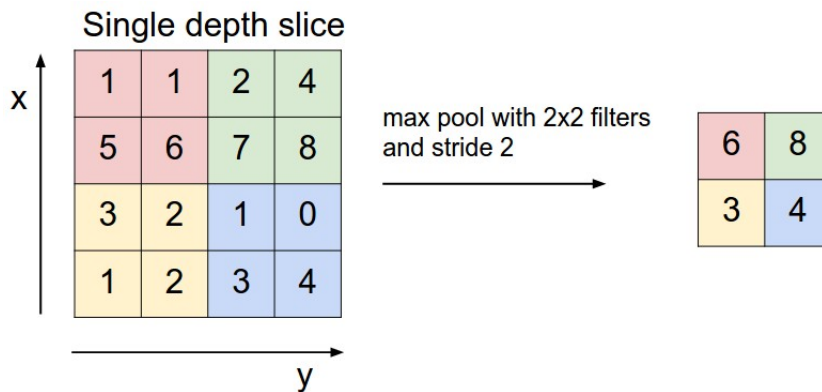


Figura 4.18: Visualizzazione del Max Pooling

Nella Fig. 4.18 l'algoritmo di Max Pooling posiziona la finestra all'inizio del tensore, considera il valore massimo presente all'interno della finestra e lo inserisce nel nuovo tensore. La finestra viene poi fatta scorrere e il processo viene ripetuto.

### 4.3.3 Struttura delle reti convolutive

A questo punto abbiamo tutti gli strumenti necessari per analizzare il funzionamento delle reti convolutive. Come abbiamo detto in precedenza queste reti sono costituite da strati convolutivi e strati di pooling che convergono ad un ultimo livello *fully connected* che produce il tensore di uscita (generalmente in forma vettoriale).

La rete prende in input un tensore di forma  $[w, h, d]$  ed effettua inizialmente le operazioni di convoluzione. I parametri caratteristici di queste operazioni (*sliding window*, *stride* e *zero padding*) sono costanti in tutto lo strato e vengono definiti al momento della definizione del modello. Viene quindi effettuato un processo ricorsivo definito come segue:

1. viene effettuato il prodotto vettoriale fra la sezione del tensore di ingresso definito dalla *sliding window* e il primo kernel del filtro dello strato
2. Il risultato della precedente operazione viene inserito nella posizione relativa del tensore di output

3. la finestra viene fatta scorrere in relazione allo *stride* e il processo si ripete

Al termine di questo processo viene prodotto in output un primo tensore detto **Feature Map**. Una volta che questo processo è terminato viene eseguito nuovamente con il secondo kernel del filtro. Quest'ultimo processo si ripete un numero  $N$  di volte pari al numero di kernel all'interno del filtro.

Il tensore in uscita avrà quindi una forma differente rispetto al tensore di ingresso, in particolare, supponendo di aver impostato lo *zero padding* in modo da conservare la forma trasversale del tensore, quello che avremo in uscita sarà un tensore con profondità uguale al numero di kernel del filtro (composto come uno stack di *feature map*).

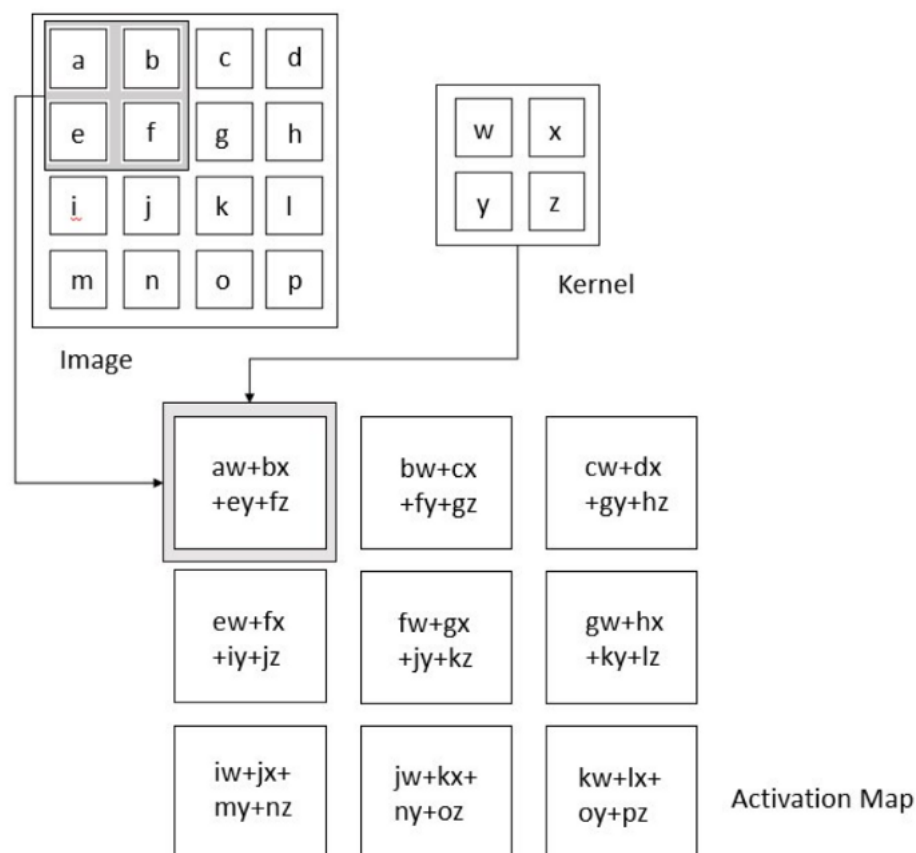


Figura 4.19: Operazioni di convoluzione [15]

Il processo in questione viene visualizzato nelle immagini 4.19 e 4.20 che specificano come viene effettuata la singola operazione di convoluzione (Fig. 4.19) e come questa si ripeta tramite la *sliding window* lungo tutto il tensore di input (Fig. 4.20) fino a produrre la *feature map* corrispondente al kernel. Strato dopo strato il numero complessivo *feature map* aumenta (analogamente alla profondità del tensore). Questo processo può essere visto come la divisione di un immagine in varie immagini a cui sono stati applicati vari filtri (da qui deriva il nome *Filtro*) che mettono in evidenza varie sezioni dell'immagine stessa.

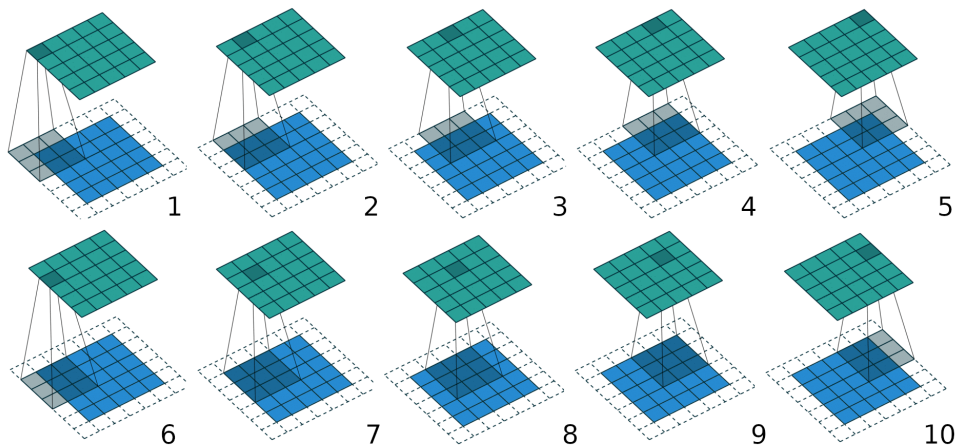


Figura 4.20: Visualizzazione del processo di convoluzione in una ConvNet. Ogni sezione dell'immagine d'ingresso viene moltiplicata per il kernel e il risultato fornisce una cella del tensore di uscita. Il processo viene ripetuto interamente su tutta l'immagine. La presenza del padding esterno consente di mantenere le dimensioni originali, in questi casi si parla di *Valid padding*.

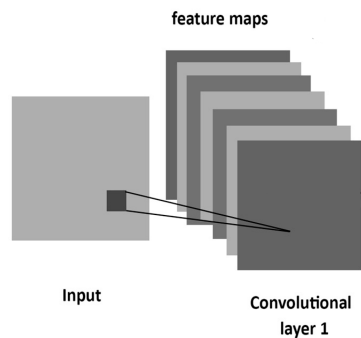


Figura 4.21: Ogni kernel del filtro dello strato convolutivo produce in output una *feature map*, complessivamente l'output dello strato sarà un tensore costituito da uno stack di feature map

Come abbiamo detto in precedenza la rete esegue delle operazioni di pooling del tensore andandone a diminuire la dimensione. L'immagine viene quindi ridotta fino ad giungere alla dimensione di un singolo vettore di profondità dipendente dalle convoluzioni precedenti.

Il vettore così ottenuto rappresenta l'ultimo step della rete neurale diventando l'input di un *Fully connected layer*. Quest'ultimo strato della rete è del tutto analogo agli strati delle *FCN* e ha il compito di discretizzare il vettore andandone ad effettuare una classificazione.

#### 4.3.4 Vantaggi e limiti delle reti convoluzionali

Le reti neurali convolutive rappresentano un punto chiave nello sviluppo della *Computer Vision*. La possibilità di analizzare immagini tramite reti neurali in grado di essere addestrate (utilizzando oltretutto modelli di addestramento in tutto coerenti)

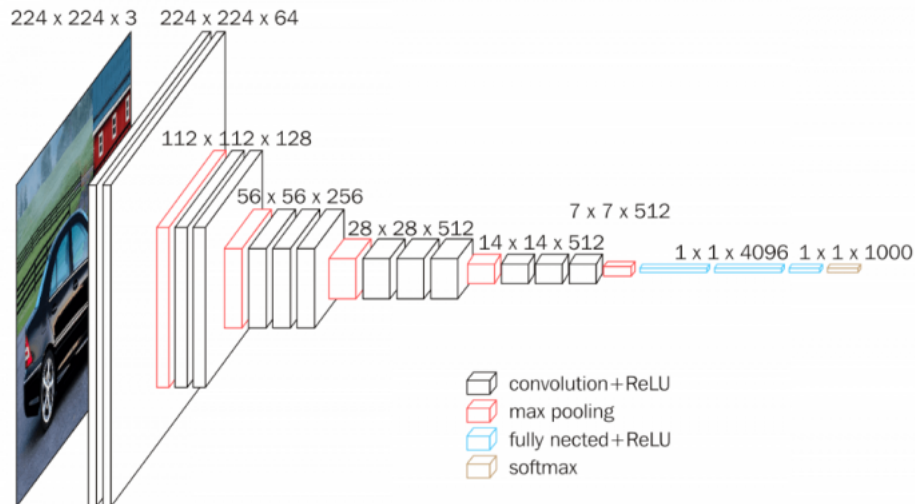


Figura 4.22: struttura completa di una ConvNet

ti con quelli delle reti FCN) con un numero ridotto di parametri hanno consentito di sviluppare soluzioni a problemi complessi di intelligenza artificiale che non potevano essere risolti tramite l'utilizzo di sistemi classici di *Machine Learning*.

Tuttavia, malgrado i loro grandi pregi, le ConvNet presentano dei limiti piuttosto importanti legati al modo in cui analizzano i dati. Come abbiamo visto nelle sezioni precedenti queste reti sfruttano sistemi di convoluzione e pooling (quest'ultimo può anche essere rimosso andando ad applicare la riduzione dell'immagine tramite lo *zero padding*) che modificano strutturalmente il tensore che definisce l'immagine. Questo processo consente alla rete di lavorare bene in task di classificazione poiché riesce ad estrapolare informazioni relative alla classe dell'oggetto, ma non riesce ad ottenere informazioni relative alla posizione degli oggetti. Inoltre, essendo presente uno strato *fully connected* in uscita il numero di vettori prodotti non può variare, questo significa che non è possibile utilizzare una ConvNet in problemi di *Object Detection* dove non abbiamo un numero fissato di possibili elementi in un'immagine. Per risolvere questi problemi devono essere introdotti il concetto di **Region of Interest (RoI)** e gli **Algoritmi di selezione**.

### Region of Interest (RoI)

le *Region of Interest* o *RoI* vengono introdotte come possibile soluzione al problema della non scalabilità delle reti convolutive. Sappiamo che le ConvNet sono in grado di portare a termine processi di classificazione con ottimi risultati, ma non possiamo utilizzarle nei problemi di *Object Detection*. Tuttavia possiamo pensare di suddividere la nostra immagine in delle regioni ben specifiche di spazio e far analizzare alla rete convolutiva solo le singole regioni in modo da ottenere una classificazione multipla su una singola immagine.

L'algoritmo più semplice in questo caso sarebbe quello di suddividere l'immagine in un numero arbitrario di regioni e andare poi a far analizzare alla rete ciascuna di

queste regioni per verificare la presenza o meno di un determinato oggetto. Il problema con questo approccio è che gli oggetti di interesse potrebbero avere differenti posizioni all'interno dell'immagine e differenti proporzioni. Quindi, dovremmo selezionare un numero enorme di regioni e questo rallenterebbe le prestazioni della computazione complessiva [16].

Per ovviare a questo problema vengono introdotti degli algoritmi di ricerca delle regioni di interesse detti **Algoritmi di selezione** (*Exhaustive Search*, *Selective Search*, ecc.). Questi algoritmi consentono di dividere l'immagine in un numero ridotto di regioni di interesse riducendo la computazione complessiva della rete.

### 4.3.5 R-CNN

La *R-CNN* o *Region Based Convolutional Neural Network* è un particolare tipo di architettura di rete neurale proposta da *Ross Girshick* come soluzione al problema dell'*Object Detection* [16].

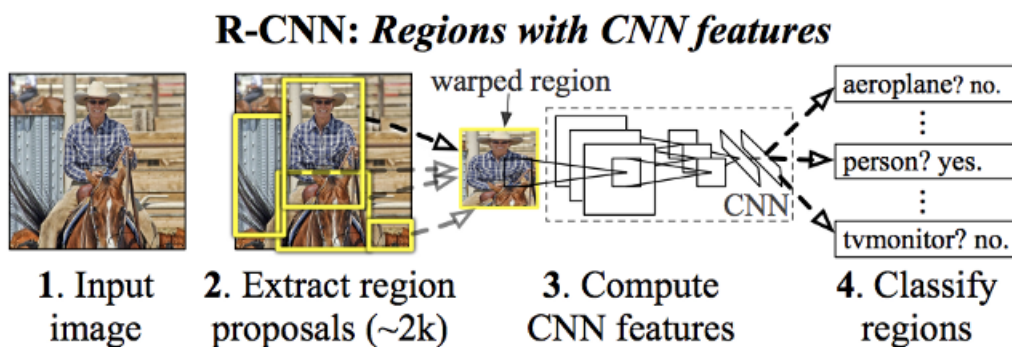


Figura 4.23: Schema funzionale di una rete R-CNN [16]

Le R-CNN sfruttano un algoritmo di selezione detto *Selective Search* per la distinzione delle regioni di interesse. Il *Selective Search* combina l'utilizzo di una ricerca esaustiva (una ricerca di tipo naturale) con i vantaggi della segmentazione dell'immagine ed è in grado di produrre un numero discreto di RoI da poter analizzare. Una volta che l'algoritmo di selezione ha selezionato le varie RoI questa vengono fatte passare attraverso una rete convolutiva pre-allenata. Questa rete si occupa di trasformare le varie regioni di immagine in un vettore delle caratteristiche. Lo step successivo della *R-CNN* è uno strato fully connected che si occupa di eseguire un algoritmo di *softmax*<sup>1</sup>, quindi una classificazione delle immagini. A questo punto ciascuna regione è stata classificata con un punteggio probabilistico

<sup>1</sup>L'algoritmo softmax è un algoritmo che trasforma un vettore n-dimensionale con valori arbitrari in un vettore n-dimensionale i cui valori sono compresi fra 0 e 1 e la loro somma è 1. Fondamentalmente il *softmax* è analogo ad una predizione probabilistica

relativo alla classe, l'ultima operazione compiuta dalla rete è quella di ridurre il numero di regioni. Questo avviene tramite un sistema definito *greedy non-maximum suppression*. L'algoritmo controlla le intersezioni delle varie regioni e scarta quelle con un punteggio probabilistico più basso. Alcune reti *R-CNN* aggiungono un *Bounding Box Regressor* che si occupa di migliorare la dimensione del box rilevato dalla *selective search* al fine di includere pienamente l'oggetto rilevato. Le reti *R-CNN* tuttavia sono estremamente lente. Questo è dovuto principalmente al fatto che l'algoritmo di *selective search* elabora un numero variabile di RoI che dipendono da alcuni parametri di ricerca. I dati della tabella 4.1 sono stati estrapolati utilizzando la libreria Python `selective_search` [17].

immagine	modalità	boxes rilevati	tempo di ricerca (s)
cat.jpeg	single	134	1.46
cat.jpeg	fast	638	6.27
cat.jpeg	quality	3031	31.66

Tabella 4.1: Tempi di esecuzione e numero di RoI per differenti modalità di *selective search*

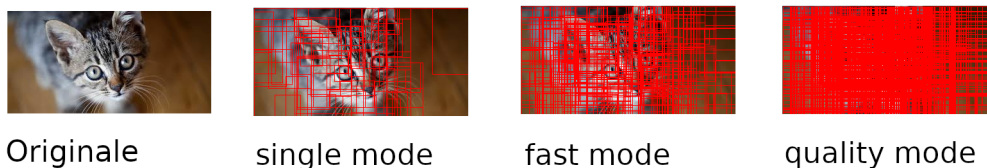


Figura 4.24: Visualizzazione delle RoI per varie modalità di *selective search*

Com'è possibile vedere l'algoritmo rileva un differente numero di RoI, un maggior numero di regioni di interesse equivale ad una maggiore precisione della rete e una maggiore probabilità di localizzare gli oggetti all'interno dell'immagine, tuttavia questo porta a tempi di esecuzione eccessivamente lunghi (la sola *selective search* impiega 31.66 s), analogamente l'utilizzo di un minor numero di regioni di interesse può portare a poca precisione nella localizzazione. Inoltre queste reti sono composte da differenti livelli che non possono essere addestrati contemporaneamente. gli strati *fully connected* che si occupano di classificare le immagini prendono in input i vettori di caratteristiche prodotti dalla rete convolutiva, questo impone che gli strati di classificazione vengano addestrati dopo la rete convolutiva, che appunto, deve essere pre-addestrata.

### 4.3.6 Fast R-CNN

Come abbiamo visto le reti *R-CNN* presentano dei problemi fondamentali legati ai tempi di esecuzione e alla difficoltà legata all'addestramento della rete stessa.



Nel 2015 una prima soluzione per migliorare le prestazioni delle *R-CNN* viene proposto dallo stesso autore tramite le *Fast R-CNN*. Il ragionamento posto alla base dell'architettura *Fast R-CNN* è molto semplice e viene costruito a partire dalle debolezze delle reti precedentemente sviluppate. Il problema dei tempi di esecuzione era generalmente causato dall'elevato numero di RoI che l'algoritmo di *selective search* produceva e al fatto che per ognuna di esse doveva essere effettuata una classificazione tramite rete neurale. Nelle *Fast R-CNN* l'operazione di convoluzione viene eseguita una sola volta sull'intera immagine in modo da produrre in uscita una *feature map* su cui poi andare a discriminare le varie possibili regioni di interesse.

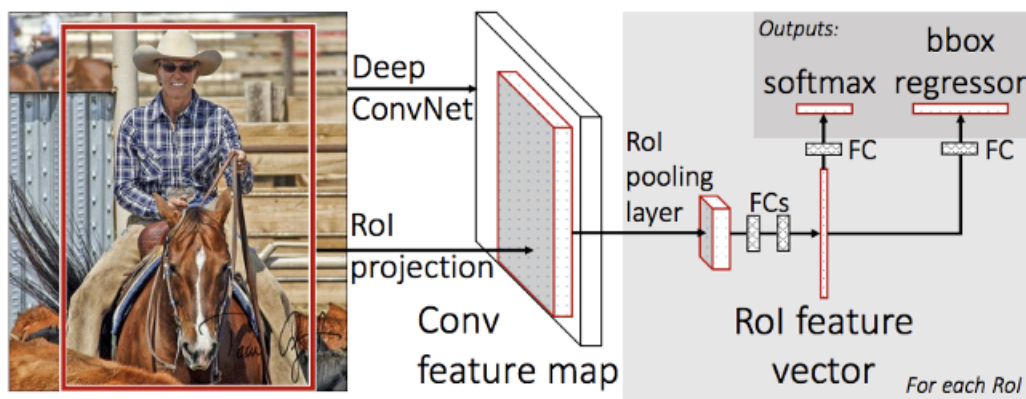


Figura 4.25: Architettura della fast R-CNN [16]

La rete convolutiva utilizzata nelle *Fast R-CNN* non utilizza strati di pooling, questo perché, una volta prodotta la feature map e discriminate le regioni di interesse, queste verranno passate attraverso un sistema di pooling detto *RoI Pooling Layer* che si occupa di ridimensionare la varie RoI ad una dimensione specifica. Questa dimensione deve coincidere con le dimensioni degli input dei livelli successivi, ciascun elemento ridimensionato dal *RoI Pooling Layer* viene fatto passare contemporaneamente in una serie di strati *fully connected* specializzati nell'effettuare la classificazione tramite *softmax* e nel *Bounding Box Regressor*.

Com'è possibile vedere nella Fig. 4.26 l'architettura *Fast R-CNN* ottiene prestazioni temporali nettamente migliori rispetto alla precedente architettura sia nell'addestramento che nell'inferenza. Altra nota a favore di quest'architettura è la maggior facilità nell'allenare la rete che, a differenza della *R-CNN*, può essere addestrata direttamente senza dover suddividere il processo in più fasi.

### 4.3.7 Faster R-CNN

Un'ultima ottimizzazione alle architetture di tipo *R-CNN* proposte da *Ross Girshick* viene presentato da *Shaoqing Ren* pochi anni dopo la *Fast R-CNN* con una nuova architettura detta *Faster R-CNN*. Come abbiamo visto in precedenza,



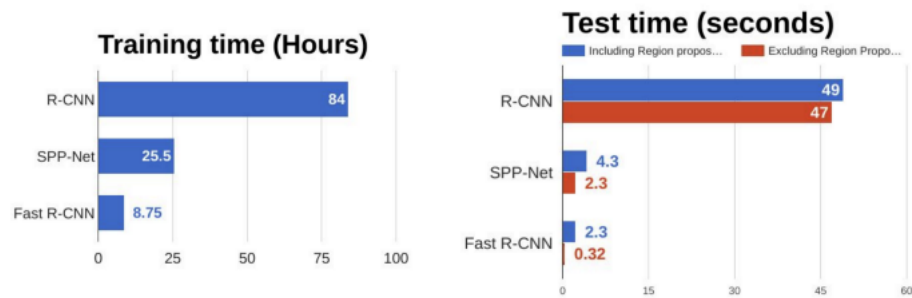


Figura 4.26: Analisi delle prestazioni di allenamento e previsione di R-CNN e Fast R-CNN [16]

l'algoritmo di *selective search* è un algoritmo piuttosto lento la cui prestazioni influenzano quelle della rete complessiva [16], la rete *Faster R-CNN* sostituisce l'algoritmo con una seconda rete addestrata per svolgere un compito del tutto analogo a quello della *Selective search*.

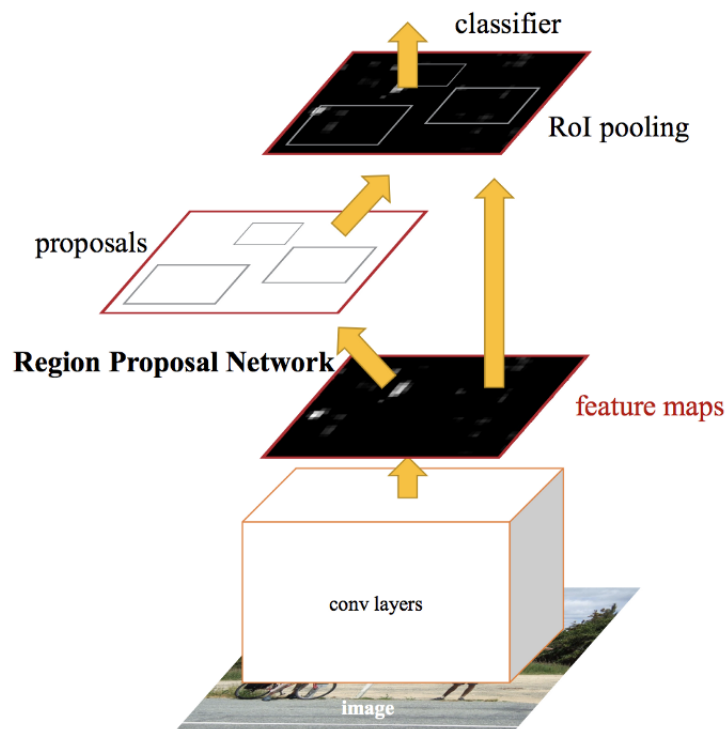


Figura 4.27: Architettura Fast R-CNN[16]

La *Faster R-CNN*, come suggerisce il nome stesso, presenta le prestazioni migliori e la presenza di una seconda rete neurale per la localizzazione degli oggetti permette di poter minimizzare tempi di ricerca e di permettere l'apprendimento anche nel campo della localizzazione, cosa che nelle precedenti architetture era

sempre compiuta tramite algoritmi analoghi a quelli dei sistemi esperti e quindi non addestrabili.

## 4.4 FCN (Fully Convolutional Networks)

Tutte le reti convoluzionali presentate fin'ora consentono di sviluppare sistemi efficienti per diversi task specialmente all'interno della *Computer Vision*. Tuttavia, tutte queste reti terminano con un livello *Dense*<sup>2</sup> che produce quindi un vettore delle caratteristiche dell'immagine analizzata. Se questo approccio consente di lavorare con problemi di classificazione e, con le varianti viste, di *object detection* non permette di approfondire l'analisi ai singoli pixel e, di conseguenza, le ConvNet soffrono nello sviluppo di algoritmi risolutivi per problemi di *segmentazione*. Sebbene esistano alcuni tentativi di utilizzare ConvNet per la segmentazione andando a classificare ogni singolo pixel dell'immagine questi processi hanno presto rilevato la loro inefficienza sia nei risultati che nei tempi di esecuzione [18].

La principale soluzione ai problemi di segmentazione dell'immagine viene rappresentata dalle *Fully Convolutional Network*. Queste reti rappresentano una particolare estensione delle ConvNet per strutture mirate alla classificazione di ogni pixel. Come abbiamo visto in precedenza le reti CNN compiono operazioni di *convoluzione* e *pooling* andando progressivamente a ridurre la dimensione dell'immagine creando vari *filtri* (rappresentati dalla profondità del tensore) che vengono in ultima fase passati attraverso uno strato *fully connected*. Nelle FCN non abbiamo strati densi, queste reti sono dette *fully convolutional* (pienamente convoluzionali) perché appunto eseguono solo operazioni di convoluzione e pooling (oltre ad un ulteriore operazione detta *up-sampling*).

### 4.4.1 Down-sampling e classificazione

Come detto in precedenza le reti completamente convoluzionali non utilizzano livelli densi per la classificazione. Se proviamo a rimuovere l'ultimo livello di una CNN otteniamo una serie di livelli di convoluzione e pooling che scalano l'immagine e la filtrano andando ad estrapolare informazioni utili. Quello che otteniamo come output di questa *rete troncata* è un tensore che ha dimensioni minori dell'immagine originale, ma una profondità molto più elevata (dipendente dal numero di filtri applicati nei precedenti strati convolutivi).

Quanto fatto fin'ora dalla rete è quindi semplicemente ridurre la dimensione del tensore di ingresso andando a creare dei filtri distinti in grado di mettere in evidenza particolari caratteristiche. Al fine di effettuare la classificazione dell'immagine (operazione che veniva effettuata dai *fully connected layers* nelle ConvNet) dobbiamo utilizzare un particolare meccanismo di convoluzione detto **1x1 Convolution**.

---

<sup>2</sup>Si parla di livelli *Densi* in riferimento a livelli *Fully Connected*

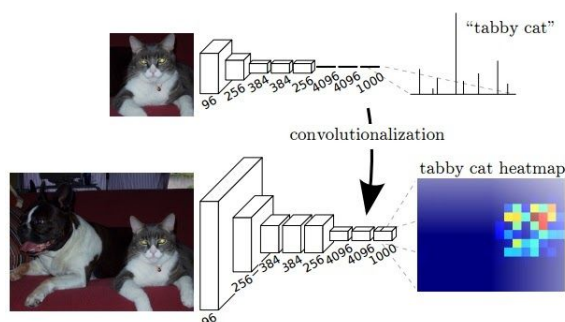


Figura 4.28: Struttura di una CNN e della rete FCN (troncata) [18]

### 1x1 Convolution

Le operazioni di *convoluzione 1x1* rappresentano uno step fondamentale nello sviluppo delle FCN. Introdotte per la prima volta nell'articolo *Network In Network* [19], le convoluzioni 1x1 vennero maggiormente utilizzate per ridurre le dimensioni spaziali come alternativa ai livelli di *Pooling*. Un livello di convoluzione 1x1 è del tutto analogo ad un classico livello convolutivo precedentemente descritto, l'unica differenza sta nella dimensione del kernel che, appunto, è di 1x1. La profondità del kernel nelle convoluzioni 1x1 permette di ridimensionare il tensore da una profondità maggiore ad una minore senza modificare le rimanenti dimensioni spaziali del tensore stesso consentendo così di avere un ridimensionamento dei parametri addestrabili.

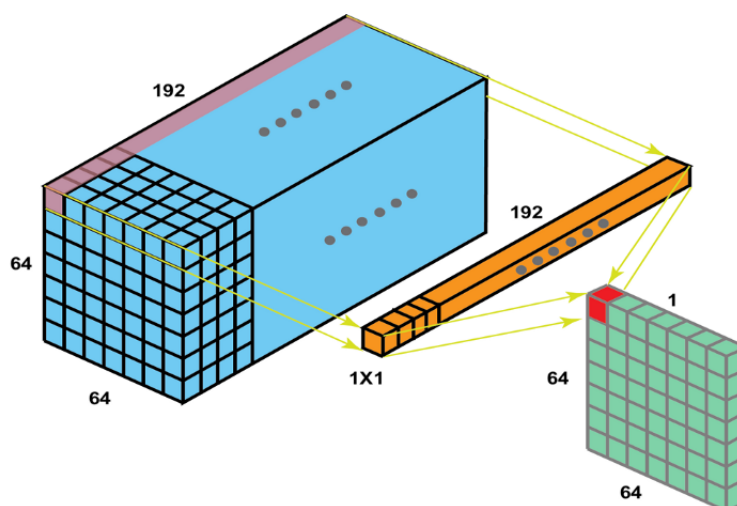


Figura 4.29: Esempio di 1x1 convolution [20]

Nelle *FCN* questi strati svolgono tuttavia un compito differente. Come riportato in Fig.4.29 è possibile utilizzare la convoluzione 1x1 per trasformare un tensore con profondità elevata in un tensore con profondità singola. Questo permette quindi di sfruttare tutti i filtri ottenuti nelle operazioni di convoluzione precedenti per ottenere una classificazione pixel per pixel e riportarci quindi ad una dimensione

tipica di un'immagine (1 dimensione in profondità per immagini in toni di grigio e 3 per le immagini RGB).

Le FCN sostituiscono quindi i livelli densi utilizzati per la classificazione nelle ConvNet con una serie di livelli di convoluzione  $1 \times 1$  che classificano i singoli pixel dell'immagine. Tuttavia questo processo non è sufficiente per la segmentazione. L'output della rete fin qui ottenuta è un'immagine di dimensioni ridotte i cui pixel sono stati classificati singolarmente. Questa immagine viene generalmente definita *Heat Map*. [18]

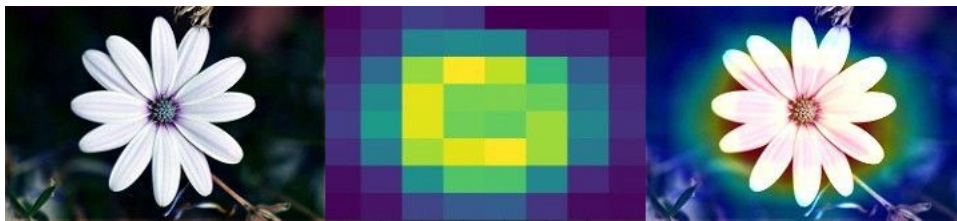


Figura 4.30: Esempio di *HeatMap*

La *Heat Map* è fondamentalmente in grado di dirci dove si trova un determinato elemento all'interno dell'immagine sottoposta, tuttavia le *heat map* sono molto piccole (rispetto all'immagine originale) e, prima di poter produrre un output, abbiamo bisogno di riportarla alla dimensione del tensore di ingresso.

#### 4.4.2 Up-Sampling

Il processo di *Up-sampling* è l'ultima serie di step compiuti da una rete neurale di tipo *fully convolutional*. Come detto in precedenza lo scopo di questo processo è di riportare le dimensioni della *heat map* ottenuta nei processi precedenti alla dimensione originale del tensore di ingresso. Le operazioni di *up-sampling* possono essere viste come l'operazione inversa del *pooling* e, di conseguenza, è possibile implementarla a partire da quest'ultimo. Possiamo tuttavia distinguere due differenti classi di up-sampling: *up-sampling statico* e *transposed convolution*.

##### Up-sampling statico

L'*up-sampling statico* è una classe di algoritmi di up-sampling che utilizza processi non addestrabili per il ridimensionamento dell'immagine. Chiaramente questi processi non dipendono dal tipo di dati coinvolti nelle operazioni e non sono quindi in grado di *apprendere* da questi. I principali algoritmi di *up-sampling statico* sono:

- **Nearest Neighbors**

L'algoritmo *Nearest Neighbors* è il più semplice algoritmo di up-sampling. L'algoritmo copia semplicemente i valori di una cella di dimensione  $n \times n$  in una di dimensione maggiore. Questo algoritmo (inizialmente utilizzato all'interno di problemi di classificazione) risulta tuttavia molto impreciso e non

è in grado di riprodurre transizioni di valore sfumate rendendo l'immagine in uscita molto rigida.

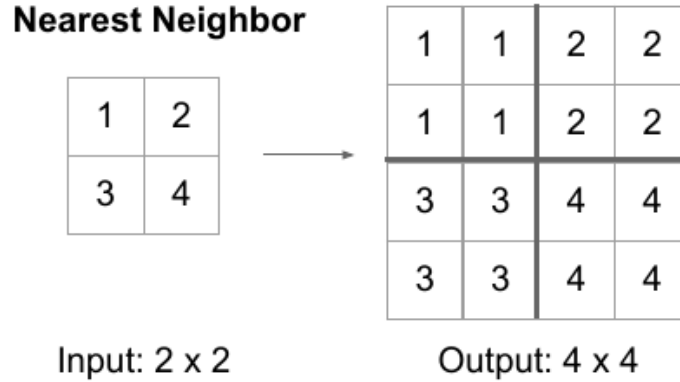


Figura 4.31: Visualizzazione dell'algoritmo *Nearest Neighbors* [21]

- **Interpolazione Bi-lineare**

L'interpolazione bi-lineare è una particolare estensione dell'interpolazione<sup>3</sup> lineare di funzioni di due variabili [22]. Supponendo di conoscere i valori di una determinata funzione in quattro distinti punti:

$$Q_{11}(x_1, y_1), Q_{21}(x_2, y_1), Q_{12}(x_1, y_2), Q_{22}(x_2, y_2)$$

applicando l'interpolazione lineare su uno dei due assi (in questo caso l'asse x) otteniamo:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \quad (4.3)$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \quad (4.4)$$

A questo punto viene effettuata un'ulteriore interpolazione lungo il rimanente asse (y nel nostro caso) utilizzando i risultati 4.3 e 4.4, ottenendo:

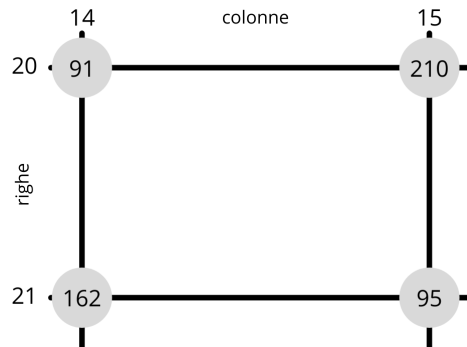
$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) =$$

$$\frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{vmatrix} x_2 - x & x - x_1 \\ f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{vmatrix} \begin{vmatrix} y_2 - y \\ y - y_1 \end{vmatrix}$$

In questo modo è possibile calcolare il generico valore di una funzione f in qualsiasi punto. Questo processo viene utilizzato nei processi di up-sampling partendo da celle di dimensione 2x2 utilizzando i singoli pixel come punti di partenza per l'interpolazione bi-lineare.

<sup>3</sup>In matematica, calcolo mediante il quale si determinano, secondo una certa legge, valori di una funzione all'interno di un intervallo nel quale ne sono noti solo alcuni.

Supponiamo di avere la seguente cella  $2 \times 2$ :

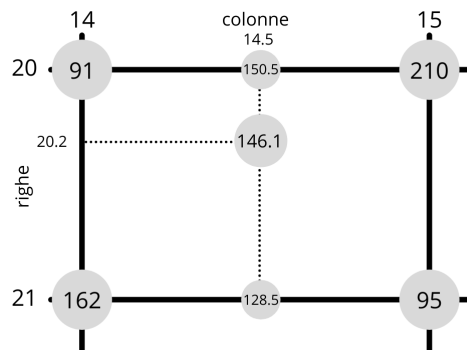


supponendo di voler calcolare l'elemento di posizione  $(14.5, 20.2)$ , seguendo i passi dell'interpolazione bi-lineare otteniamo:

$$I_{20,14.5} = \frac{15 - 14.5}{15 - 14} \cdot 91 + \frac{14.5 - 14}{15 - 14} \cdot 210 = 150.5,$$

$$I_{21,14.5} = \frac{15 - 14.5}{15 - 14} \cdot 162 + \frac{14.5 - 14}{15 - 14} \cdot 95 = 128.5,$$

$$I_{20.2,14.5} = \frac{21 - 20.2}{21 - 20} \cdot 150.5 + \frac{20.2 - 20}{21 - 20} \cdot 128.5 = 146.1.$$



Ripetendo il processo per ogni cella aggiuntiva è possibile scalare l'immagine da un dimensione ridotta fino a giungere alla dimensione originale. Il principale vantaggio rispetto all'algoritmo *Nearest Neighbors* è la possibilità di ottenere cambi di valore gradualmente che rendono l'immagine in uscita meno rigida.

- **Interpolazione Bi-cubica**

L'*interpolazione bi-cubica* è un'estensione dell'interpolazione cubica per funzioni di due variabili [23]. Analogamente a quanto detto per l'interpolazione bi-lineare questa esegue l'interpolazione sui due assi indistintamente per poi

calcolare il risultato finale a partire da quelli precedentemente calcolati. Nell'elaborazione delle immagini, l'interpolazione bi-cubica viene spesso scelta rispetto all'interpolazione bilineare o al *Nearest Neighbors* nel ricampionamento delle immagini, quando la velocità non è un problema (questo poiché, sebbene più preciso, l'algoritmo d'interpolazione bi-cubica è molto più lento). A differenza dell'interpolazione bi-lineare, che prende in considerazione solo 4 pixel ( $2 \times 2$ ), l'interpolazione bi-cubica considera 16 pixel ( $4 \times 4$ ). Le immagini ricampionate con l'interpolazione bi-cubica sono più uniformi e presentano meno artefatti di interpolazione<sup>4</sup> [23].

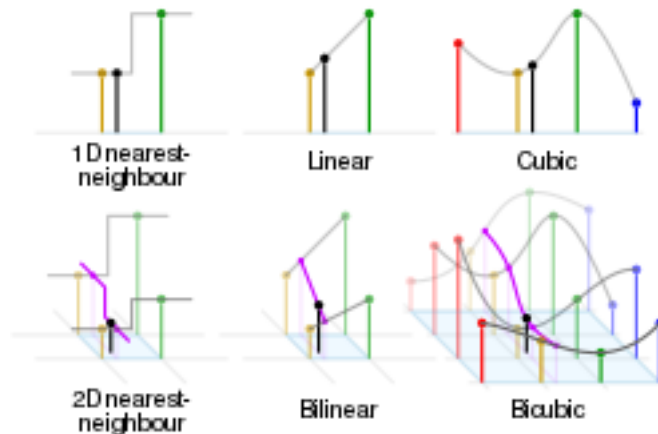


Figura 4.32: Paragone fra gli algoritmi *Nearest Neighbor*, interpolazione bi-lineare e interpolazione bi-cubica. I punti neri e rossi/gialli/verdi/blu corrispondono rispettivamente al punto interpolato e ai campioni vicini. Le loro altezze dal suolo corrispondono ai loro valori[23].

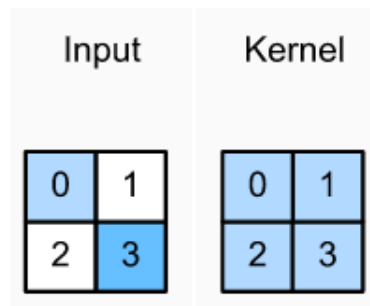
### Transposed Convolution

Tutti gli algoritmi di up-sampling visti fin'ora sono algoritmi statici privi di parametri addestrabili e quindi non in grado di migliorare le loro prestazioni basandosi sull'errore commesso. Per migliorare la qualità delle predizioni nelle reti FCN moderne il processo di up-sampling viene delegato ad un'ulteriore serie di strati convolutivi detti *Transped Convolutional Layers*. Questi strati effettuano delle operazioni di convoluzione "inversa", ovvero, a partire da un singolo pixel producono in output una matrice di pixel a partire da un *kernel* addestrabile.

Supponiamo di avere in input una matrice di pixel di dimensione  $2 \times 2$  e di voler ottenere, tramite un *transposed convolutional layer*, una matrice di pixel di dimensione  $3 \times 3$  in uscita. Lo strato presenta un kernel di dimensione  $2 \times 2$ .

Ogni elemento della matrice dei pixel viene moltiplicato per il kernel. Supponiamo di moltiplicare l'elemento di posto  $(0, 0)$ , la matrice prodotta dall'operazione

<sup>4</sup>Con artefatti d'interpolazione si intendono tutte le distorsioni dell'immagine dovute all'impossibilità di ricampionare fedelmente un'immagine



andrà quindi a posizionarsi, a partire dall'angolo in alto a sinistra, sull'elemento (0,0) della matrice  $3 \times 3$  dell'output. Poiché alcuni elementi potrebbero sovrapporsi (dipende dalla dimensione del kernel e della dimensione della matrice di output) questi verranno sommati fra loro andando così ad ottenere una matrice  $3 \times 3$  in output.

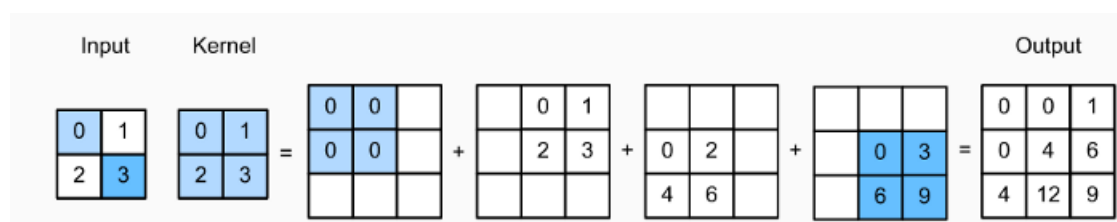


Figura 4.33: Visualizzazione delle operazioni di *transposed convolution* [21]

Nel caso di immagini RGB la matrici vengono sostituite da tensori e le operazioni vengono eseguite ugualmente per ogni livello di profondità del tensore. La *transposed convolution* rappresenta un sistema di up-sampling addestrabile (il kernel è un tensore di parametri addestrabili) e, com'è possibile osservare in fig.4.34, presenta risultati e tempi di esecuzione migliori rispetto agli algoritmi fin qui trattati.

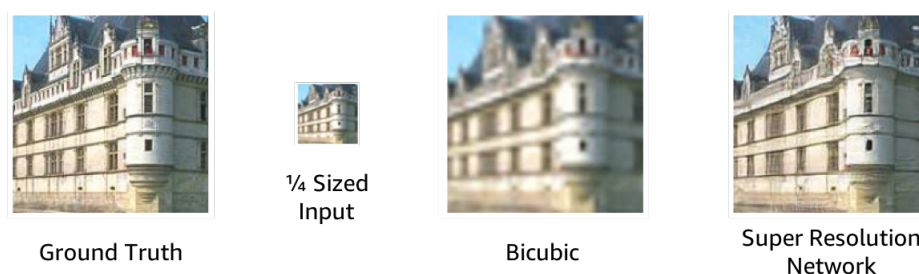


Figura 4.34: Up-sampling della medesima immagine tramite interpolazione bicubica e *transposed convolution* [21]



### Artefatti a scacchiera

Uno dei principali problemi nell'upsampling tramite *transposed convolution* viene definito *artefatto a scacchiera* ovvero la tendenza, nelle immagini ridimensionate, in sezioni omogenee di colore a presentare un pattern a scacchiera con varie intensità del medesimo colore.

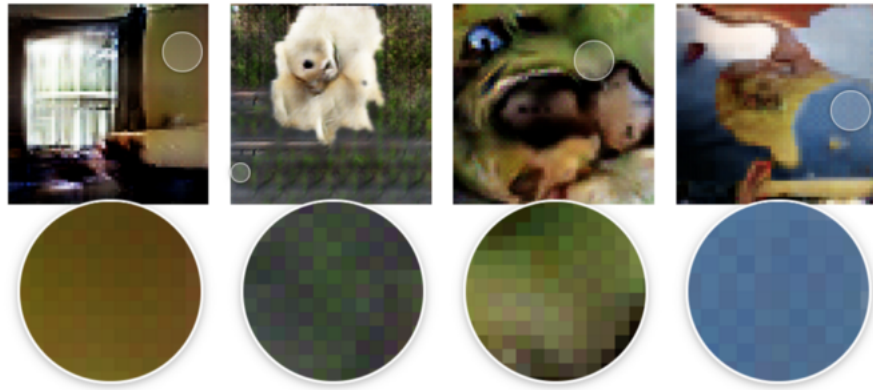


Figura 4.35: Esempi di artefatti a scacchiera

Questo problema è principalmente dovuto alle aree di sovrapposizione che vengono sommate durante le operazioni di *transposed convolution* e che generano il tipico pattern. Il modo più semplice per ridurre questo problema è quello di scegliere la dimensione del kernel e dello stride in modo da evitare la formazione di aree di sovrapposizione fra le matrici.

# Capitolo 5

## Sviluppo Del Progetto

L'obiettivo del progetto di sviluppo *Blind Athletes* consiste nell'implementare, tramite l'utilizzo di un sistema di *Deep Learning* con apprendimento supervisionato, un algoritmo in grado di rilevare la corsia di una pista di atletica sulla quale un atleta si sta muovendo. L'implementazione del progetto si è svolta in più fasi relativamente alla scelta del tipo di architettura della rete, la creazione di un dataset sufficientemente ampio (tramite l'utilizzo di particolari *tool*), l'addestramento e l'ottimizzazione della rete per permettere tempi di inferenza compatibili con un'applicazione *real time*.

Prima di poter definire l'architettura della rete e definire quindi un dataset per l'addestramento è necessario stabilire in che modo la rete avrebbe dovuto predire la posizione della corsia centrale. Il progetto inizialmente pensato era un algoritmo di *Lane Detection*, ovvero un algoritmo in grado di predire la posizione delle sole linee che tracciano il confine della corsia stessa. Tuttavia una soluzione del genere comporta poi un maggiore peso computazionale nel processo di calcolo della traiettoria. Abbiamo quindi optato per una rete in grado di riconoscere non solo le linee, ma l'intera corsia centrale relativamente all'inquadratura (ripresa da una telecamera posta sul petto dell'atleta) in modo da rendere più agevole il calcolo della traiettoria e ridurre i tempi necessari all'analisi di ogni singolo frame.



Figura 5.1: Esempio di rilevamento della corsia

Come mostrato in Fig.5.1 l'obiettivo della rete è quello di ottenere una maschera della corsia di percorrenza in modo da conoscere pienamente la zona che l'atleta può percorrere senza dover effettuare ulteriori calcoli sull'immagine (operazione che può essere dispendiosa su un hardware con capacità di calcolo limitate). Una volta

appurato il tipo di risultato che si vuole ottenere dalla rete è possibile procedere nella progettazione e nello sviluppo di un'architettura.

## 5.1 Architettura Della Rete

Come detto nei precedenti capitoli la *Computer Vision* sfrutta differenti tipologie di architetture di reti neurali (principalmente di tipo convolutivo) nella soluzione di problemi legati all'analisi delle immagini basandosi sulla tipologia del problema da risolvere e sulle ottimizzazioni possibili. Il nostro modello rientra nella classe di problemi legati alla *semantic segmentation*, questo perchè abbiamo bisogno di individuare posizione e perimetro della corsia su cui l'atleta si sta muovendo. Le reti che meglio si prestano alla soluzione di problemi di segmentazione, da quanto discusso in precedenza, sono le *Fully Convolutional Networks* in grado di ottenere ottimi risultati con tempi di inferenza brevi. Il progetto sviluppa quindi una particolare *FCN* generalmente definita **U-Net**.

### 5.1.1 U-Net

La U-Net [24] è un particolare tipo di *fully connected network* sviluppata per ottimizzare i processi di *image segmentation*. Come tutte le reti di tipo FCN è composta da due distinti percorsi definiti come *Encoder* e *decoder*.

- Il percorso encoder non mostra differenze significative rispetto a quello classico delle reti FCN
- Il percorso del decoder utilizza invece un sistema differente per l'up-sampling dell'immagine. Oltre ad utilizzare dei *transposed convolutional layers* per ingrandire l'immagine vengono effettuati dei concatenamenti con il nodo rispettivo del percorso dell'encoder.

la Fig.5.2 mostra la struttura generale di una rete U-Net. È possibile vedere che il percorso di up-sampling esegue degli specifici passi ad ogni iterazione, in particolare:

1. Il tensore in ingresso viene fatto "passare" attraverso un *transposed convolutional layer* i cui parametri (*kernel* e *stride*) sono organizzati in modo che il tensore di uscita abbia dimensione e profondità coincidenti con il tensore di uscita del passo precedente nel percorso dell'*encoder*. Questa operazione risulta fondamentale affinché possa essere effettuato il passo successivo.
2. Il tensore ridimensionato viene quindi concatenato con il tensore in uscita del rispettivo passo nel cammino di *encoder* (la freccia grigia nella fig.5.2). Questa operazione produce in uscita un tensore di profondità  $2n$ , dove  $n$  è la profondità dei tensori in uscita dai livelli di *encoder* e *decoder*. Fondamentalmente il tensore ridimensionato dal *transposed convolutional layer* e quello in uscita dal relativo passo del cammino di *encoder* vengono uniti assieme.

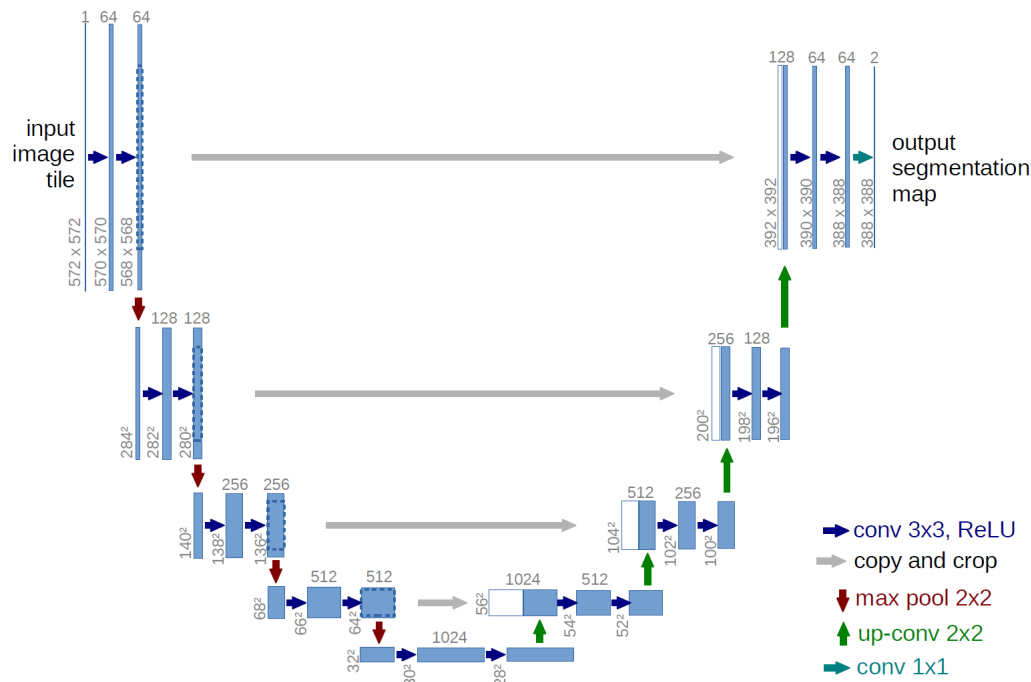


Figura 5.2: Struttura della rete U-Net [24]

3. Quest'ultimo tensore viene passato attraverso uno o più livelli convolutivi che hanno il compito di ridurre la profondità del tensore e migliorare la classificazione effettuata dall'up-sampling tramite l'utilizzo della *feature map* prodotta dal cammino di *encoder*

Questo processo viene ripetuto fin quando la dimensione del tensore di uscita non combacia con quella del tensore in ingresso. A questo punto viene applicata una convoluzione 1x1 che classifica i singoli pixel dell'immagine andando così a produrre in output una maschera degli elementi presenti nell'immagine di ingresso.

L'utilizzo di strati convolutivi nel percorso di up-sampling e l'utilizzo della concatenazione consentono alla rete di ottenere un ridimensionamento più preciso dell'immagine e, di conseguenza, una maggiore precisione nella segmentazione stessa.

### 5.1.2 Implementazione della U-Net

La prima fase della realizzazione del progetto è stata interamente implementata tramite l'ausilio della piattaforma *Google Colaboratory* [25] e delle librerie *Tensorflow* [26]. *Google Colaboratory* è una piattaforma online sviluppata da *Google* che consente, tramite l'utilizzo di particolari ambienti interattivi detti *Blocchi Note*, di sviluppare ed eseguire codice Python combinato con RTF, HTML, LaTeX e altro senza alcuna configurazione. *Google Colaboratory* è pensato per lo sviluppo negli ambiti della *Data Science* e *Artificial Intelligence*, permette quindi di utilizzare

le principali librerie Python quali *Numpy* [27], *Tensorflow* e *OpenCV* [28] senza la necessità di installarle su un computer locale, ma utilizzando semplicemente il browser. Il codice viene inoltre eseguito sui *Serve Cloud* di Google permettendo l'utilizzo di accelerazioni hardware quali GPU e TPU<sup>1</sup> indipendentemente dal tipo di macchina locale.

*Tensorflow* è una libreria *open source* per l'intelligenza artificiale fra le più utilizzate nello sviluppo di reti neurali e algoritmi per il deep learning. Anch'essa sviluppata da *Google* è una raccolta di librerie open sources (tra le quali *Keras*, fondamentale per lo sviluppo di reti neurali) alla base di molti prodotti della casa informatica (*Google Assistant*, *Gmail*, *Google foto*).

Il primo sviluppo della U-Net segue perfettamente lo schema riportato in fig.5.2. Ogni livello dell'*encoder* è formato da due strati convolutivi con un kernel di dimensione (3,3) e funzione di attivazione di tipo *ReLU*. A questi segue uno strato di Max Pooling con dimensione del kernel (2,2) con uno stride pari a 2 che ne dimezza la dimensione. I livelli del percorso di *decoder* sono invece formati da uno strato *transposed convolution* a cui seguono la concatenazione e di nuovo due strati convolutivi. L'architettura complessiva viene descritta come segue:

---

```

1 def conv2d_block(input_tensor, n_filters, kernel_size =3,
2                 batchnorm=True):
3     #primo livello
4     x = Conv2D(filters=n_filters,
5               kernel_size=(kernel_size, kernel_size),
6               kernel_initializer='he_normal', padding='same')(
7         input_tensor)
8     if batchnorm:
9         x = BatchNormalization()(x)
10
11    x = Activation('relu')(x)
12
13    #secondo livello
14    x = Conv2D(filters=n_filters,
15              kernel_size=(kernel_size, kernel_size),
16              kernel_initializer='he_normal', padding='same')(x)
17
18    if batchnorm:
19        x = BatchNormalization()(x)
20
21    x = Activation('relu')(x)
22
23    return x

```

---

```

1 def get_unet(n_filters = 16, dropout = 0.1, batchnorm = True):
2
3     input_img = Input(shape=(img_height, img_width, 1))

```

---

<sup>1</sup>*Tensor Processing Unit*. Particolari unità di calcolo sviluppate da Google ed ottimizzate per lavorare con tensori multidimensionale

```
4 #Contracting path
5 c1 = conv2d_block(input_img, n_filters*1, kernel_size=3,
6   batchnorm=batchnorm)
7 p1 = MaxPooling2D((2,2))(c1)
8 p1 = Dropout(dropout)(p1)
9
10 c2 = conv2d_block(p1, n_filters*2, kernel_size=3, batchnorm=
11   batchnorm)
12 p2 = MaxPooling2D((2,2))(c2)
13 p2 = Dropout(dropout)(p2)
14
15 c3 = conv2d_block(p2, n_filters*4, kernel_size=3, batchnorm=
16   batchnorm)
17 p3 = MaxPooling2D((2,2))(c3)
18 p3 = Dropout(dropout)(p3)
19
20 c4 = conv2d_block(p3, n_filters*8, kernel_size=3, batchnorm=
21   batchnorm)
22 p4 = MaxPooling2D((2,2))(c4)
23 p4 = Dropout(dropout)(p4)
24
25 c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3,
26   batchnorm=batchnorm)
27 p5 = MaxPooling2D((5,5))(c5)
28 p5 = Dropout(dropout)(p5)
29
30 c6 = conv2d_block(p5, n_filters*32, kernel_size=3, batchnorm=
31   batchnorm)
32
33 #Expansive path
34 u7 = Conv2DTranspose(n_filters*16, (3,3), strides=(5,5), padding
35   ='same')(c6)
36 u7 = concatenate([u7, c5])
37 u7 = Dropout(dropout)(u7)
38 c7 = conv2d_block(u7, n_filters*16, kernel_size=3, batchnorm=
39   batchnorm)
40
41 u8 = Conv2DTranspose(n_filters*8, (3,3), strides=(2,2), padding=
42   'same')(c7)
43 u8 = concatenate([u8, c4])
44 u8 = Dropout(dropout)(u8)
45 c8 = conv2d_block(u8, n_filters*8, kernel_size=3, batchnorm=
46   batchnorm)
47
48 u9 = Conv2DTranspose(n_filters*4, (3,3), strides=(2,2), padding=
49   'same')(c8)
50 u9 = concatenate([u9, c3])
51 u9 = Dropout(dropout)(u9)
52 c9 = conv2d_block(u9, n_filters*4, kernel_size=3, batchnorm=
53   batchnorm)
54
55 u10 = Conv2DTranspose(n_filters*2, (3,3), strides=(2,2), padding
56   ='same')(c9)
```

```

44 u10 = concatenate([u10, c2])
45 u10 = Dropout(dropout)(u10)
46 c10 = conv2d_block(u10, n_filters*2, kernel_size=3, batchnorm=
    batchnorm)
47
48 u11 = Conv2DTranspose(n_filters*1, (3,3), strides=(2,2), padding
    ='same')(c10)
49 u11 = concatenate([u11, c1])
50 u11 = Dropout(dropout)(u11)
51 c11 = conv2d_block(u11, n_filters*1, kernel_size=3, batchnorm=
    batchnorm)
52
53 outputs = Conv2D(1, (1,1), activation='sigmoid', dtype='float32'
    )(c11)
54 model = tf.keras.Model(inputs=[input_img], outputs=[outputs])
55 return model

```

nell'implementazione del modello sono stati aggiunti due strati definiti come **Batch Normalization** e **Dropout**. La *Batch Normalization* è un metodo algoritmico che rende più veloce e stabile l'addestramento delle reti neurali [29]. Questo algoritmo utilizza la media (5.1) e la varianza (5.2) del vettore delle uscite di un determinato strato:

$$\mu = \frac{1}{n} \sum_i Z_i \quad (5.1)$$

$$\sigma = \frac{1}{n} \sum_i (Z_i - \mu) \quad (5.2)$$

dove  $Z_i$  rappresenta il vettore delle uscite, e ne effettua una normalizzazione:

$$Z_i^{norm} = \frac{Z_i - \mu}{\sqrt{\sigma^2 - \epsilon}} \quad (5.3)$$

In questo modo le uscite di un determinato strato seguono una distribuzione normale (Gaussiana):

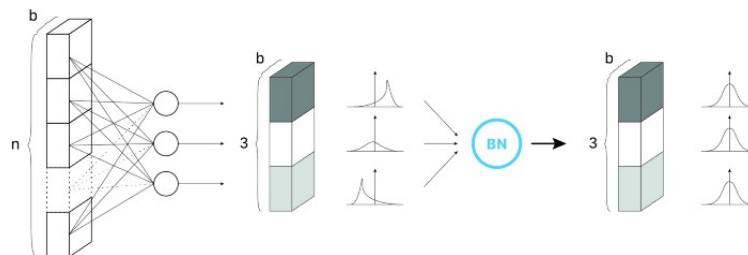


Figura 5.3: Visualizzazione della batch normalization [29]

Come ultimo step viene calcolata l'uscita del livello come una funzione di due parametri addestrabili:

$$Z_{out} = \gamma \cdot Z_i^{norm} + \beta \quad (5.4)$$

Questi due parametri applicano una trasformazione lineare che permette alla rete di scegliere la distribuzione normale ottimale per quello strato regolandoli [29].

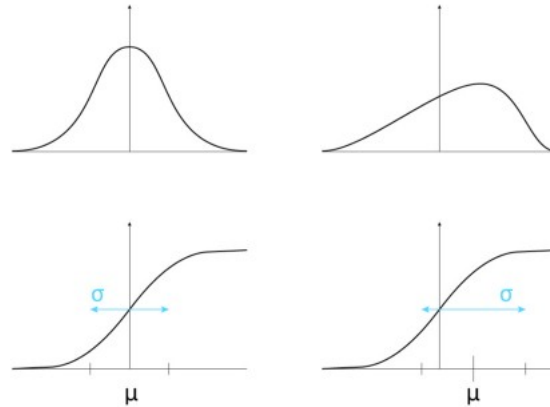


Figura 5.4: modifica della distribuzione normale tramite modifica dei parametri  $\gamma, \beta$  [29]

Il *Dropout*, invece, è una tecnica che riduce la possibilità dello sviluppo di problemi di *overfitting* durante la fase di allenamento della rete. Per evitare che i neuroni diventino eccessivamente dipendenti dai risultati degli altri il *dropout* rimuove temporaneamente e in modo causale alcuni dei neuroni dalla rete durante le fasi di addestramento.

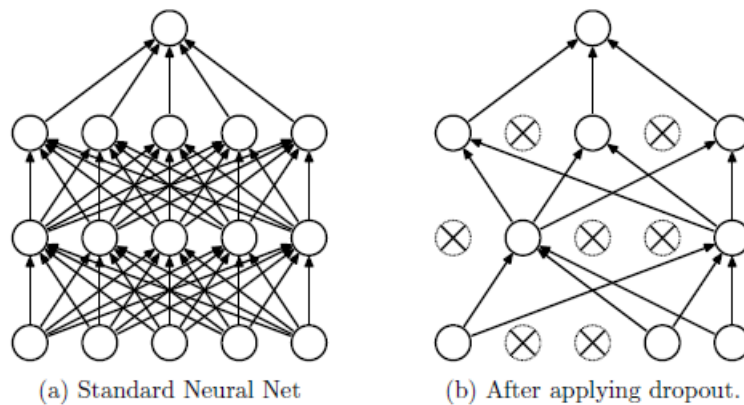


Figura 5.5: Durante l'allenamento alcuni neuroni vengono disconnessi e rimane attiva solo una sottoparte della rete [30]



La rete così sviluppata presenta quindi le seguenti caratteristiche:

	U-Net 1
Numero strati (esclusi Dropout e Concatenazioni)	54
Parametri totali	8.660.305
parametri addestrabili	8.654.289
parametri non addestrabili	6.016

Tabella 5.1: Riassunto implementazione U-net

La rete prende in input un tensore di forma  $(240, 400, 1)$ , la dimensione del tensore di ingresso (quindi dell'immagine da analizzare) è un parametro importante poiché l'utilizzo di immagini eccessivamente grandi può portare la rete ad essere troppo lenta e non adatta ad un'applicazione *real-time* mentre l'utilizzo di immagini eccessivamente piccole può comportare una perdita di informazione eccessiva e una conseguente imprecisione nelle predizioni del modello.

La rete attuale presenta un elevato numero di parametri addestrabili e di strati che, sebbene possa avere tempi di inferenza accettabili sulle GPU messe a disposizione da *Google Colab*, può essere inadatta all'inferenza su dispositivi con minore capacità computazionale, più avanti nella discussione verranno introdotte differenti tipologie di implementazione della rete.

## 5.2 Creazione Del Dataset

La rete U-Net implementata sfrutta un sistema di addestramento supervisionato, ha bisogno quindi di dati etichettati per poter addestrare i propri parametri. Il dataset è stato creato tramite l'ausilio di un *tool* online chiamato *Label Box* [31]. Labelbox è una piattaforma per la produzione di dati di allenamento con strumenti di etichettatura veloci abilitati all'intelligenza artificiale, automazione dell'etichettatura, forza lavoro umana, gestione dei dati, una potente API per l'integrazione e un SDK Python per l'estensibilità.

*Label Box* consente di utilizzare l'archiviazione su server per la produzione di dati etichettati. L'intero dataset viene quindi archiviato sui server di *Label Box* ed esportato in formato *JSON*<sup>2</sup> o *CSV*<sup>3</sup> e i singoli dati sono accessibili tramite i corrispettivi link all'interno del documento esportato. Questo rende la produzione e l'utilizzo di dataset (anche di grandi dimensioni) molto più semplice, eliminando

<sup>2</sup>*Javascript Object Notation*. È un formato file adattato per lo scambio di dati fra *client* e *server* originariamente sviluppato per l'utilizzo con javascript come descrittore di classi, ma poi divenuto quasi universale

<sup>3</sup>*Comma Separated Values*. È un formato file per l'importazione e l'esportazione di dati in formato tabulare

la necessità di salvare in locale grandi quantità di dati e, nel caso di *Google colab*, quella di caricare il dataset su server cloud (operazione dispendiosa che dovrebbe essere ripetuta ad ogni accesso).

### 5.2.1 Dati ed Etichette

Lo sviluppo di un dataset quanto più semplice possibile è una prerogativa importante nello sviluppo di algoritmi di apprendimento profondo. Tanto più complesso e articolato è il dataset, tanto più complessa e profonda dovrà essere la rete per poter eseguire predizioni coerenti e con poco errore. La nostra rete deve essere in grado di distinguere la posizione e la forma della corsia su cui si muove il nostro atleta. Il dataset più semplice può essere realizzato etichettando la sola corsia di percorrenza e mantenendo tutto il resto non etichettato. L'output che ci aspettiamo dalla rete quindi è una semplice immagine binaria con i pixel impostati a 1 (analagamente 255) per la corsia e 0 per il resto.

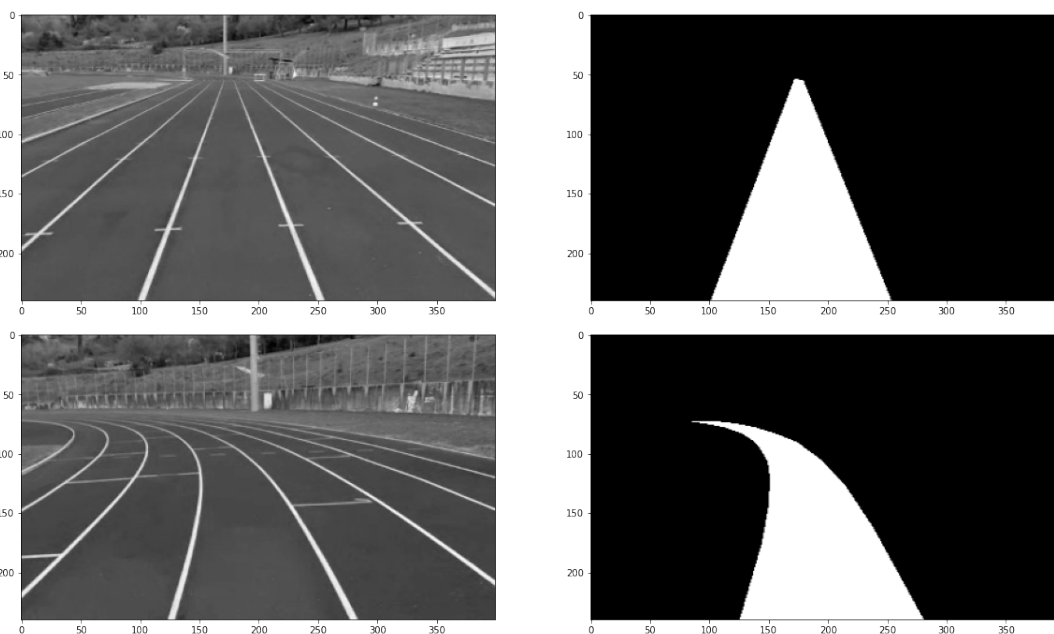


Figura 5.6: Esempi di etichette del dataset

Le immagini del dataset sono state estrapolate dai frame di alcuni video girati in piste d'atletica con videocamere o *smartphone* posizionati sul petto della persona. Questi video comprendono varie condizioni di ripresa, come ad esempio riprese notturne o durante la corsa. Il dataset complessivo conta di circa 700 immagini e rispettive etichette. Sebbene non numerose (tendenzialmente per compiti di image segmentation si utilizzano dataset contenenti qualche migliaio di immagini per permettere alla rete di esplorare quanto più approfonditamente varie angolazioni e posizioni dell'inquadratura) il dataset è stato sufficiente per l'addestramento della rete mettendo in mostra punti di forza e debolezze della stessa.

### 5.3 Addestramento della rete

Il modello sviluppato è stato addestrato sulla piattaforma *Google Colaboratory* sfruttando l'accelerazione GPU. Per l'addestramento sono stati utilizzati alcuni parametri detti *Callbacks*. I *Callbacks* sono un insieme di funzioni che vengono eseguite in determinati istanti della fase di addestramento. Questi sono molto utili per controllare l'andamento dell'apprendimento, ma anche per migliorarne le prestazioni e ridurre i problemi di *overfitting*. In questo progetto sono stati utilizzati i seguenti *callbacks*:

---

```

1 callbacks = [
2     EarlyStopping(patience=20,
3                   verbose=1),
4     ReduceLROnPlateau(factor=0.1,
5                       patience=5,
6                       min_lr=0.00001,
7                       verbose=1),
8     ModelCheckpoint('model-weights.h5',
9                    verbose=1,
10                   save_best_only=True,
11                   save_weights_only=True),
12     tf.keras.callbacks.TensorBoard(log_dir='/content/logs',
13                                    histogram_freq=0,
14                                    write_graph=True,
15                                    write_images=False,
16                                    write_steps_per_second=False,
17                                    update_freq='epoch',
18                                    profile_batch=2,
19                                    embeddings_freq=0,
20                                    embeddings_metadata=None)
21 ]
```

---

- **Early Stopping**

Questa Callback consente di monitorare l'apprendimento della rete e fermare l'aggiornamento dei parametri in anticipo quando non ci sono miglioramenti nella funzione di perdita o nell'accuratezza del modello. Il parametro principale è il parametro *patience* che indica il numero di epoche consecutive senza miglioramenti necessarie all'arresto anticipato del processo di apprendimento. In questo caso, se per 20 epoche consecutive non ci sono miglioramenti nella funzione di perdita, l'addestramento viene interrotto.

- **Reduce LR On Plateau**

Questo Callback consente di modificare il *learning rate* dell'algoritmo di apprendimento (Cap. 3) quando non ci sono miglioramenti nell'apprendimento della rete. A differenza del *Learning Rate Schedule* (altro Callback di Tensorflow) che consente di ridurre il *learning rate* dopo un certo numero di epoche, questo callback basa la sua attivazione sull'apprendimento della rete con l'obiettivo di evitare eventuali situazioni di stallo nell'addestramen-

to. In questo caso abbiamo due parametri caratteristici: il *factor*, ovvero il fattore di riduzione del *learning rate*, e il *patience*, analogo all'*Early Stopping*.

- **Model Checkpoint**

Il *Model Checkpoint* consente di salvare i parametri della rete in determinati istanti dell'addestramento. Questo Callback può essere utile sia per avere dei punti di salvataggio durante l'addestramento dei modelli (specialmente in caso di lunghi tempi di apprendimento), ma anche per salvare i pesi del modello relativi ai risultati migliori ottenuti (al termine del processo i pesi del modello sono quelli dell'ultima epoca, ma la rete potrebbe aver ottenuto risultati migliori in epoche precedenti).

- **TensorBoard**

*TensorBoard* è un *tool* di *Tensorflow* che monitora l'andamento dell'addestramento della rete e consente la creazione di grafici e tabelle utili per l'analisi.

L'ultima operazione compiuta prima dell'addestramento della rete è stata quella di dividere il dataset in due parti, una per l'addestramento vero e proprio e una per la validazione. Questa operazione viene effettuata per migliorare l'apprendimento della rete:

- il *Test Set* è una parte (più grande) del dataset che viene utilizzata durante l'addestramento vero e proprio della rete. Le immagini vengono fatte analizzare, i risultati predetti vengono confrontati con le etichette e i pesi della rete vengono aggiornati.
- il *Validation Set* è una seconda parte del dataset (più piccola) che viene utilizzata al termine di ogni epoca. Dopo un primo addestramento del modello questo effettua una predizione sul *Validation Set* (composto da elementi che non facevano parte del processo di addestramento e che quindi la rete non ha mai analizzato) e se ne valutano l'accuratezza e la perdita. Questi valori sono quelli utilizzati poi per valutare le prestazioni della rete e modificare alcuni parametri tra cui il *learning rate*.

Questo approccio consente di minimizzare ulteriormente i problemi di *overfitting* riducendo la dipendenza fra gli elementi di addestramento e l'aggiornamento dei pesi. La rete è stata a questo punto addestrata producendo i seguenti grafici di perdita e accuratezza (fig. 5.7).

## 5.4 Ottimizzazioni

Come detto in precedenza, la dimensione e il numero di parametri della rete fin qui analizzata possono essere eccessivi per permettere l'inferenza su dispositivi con capacità computazionale ridotta in tempi che supportano l'esecuzione *Real Time*. Per questo è possibile ridurre o ottimizzare la rete in modo da avere perdite di

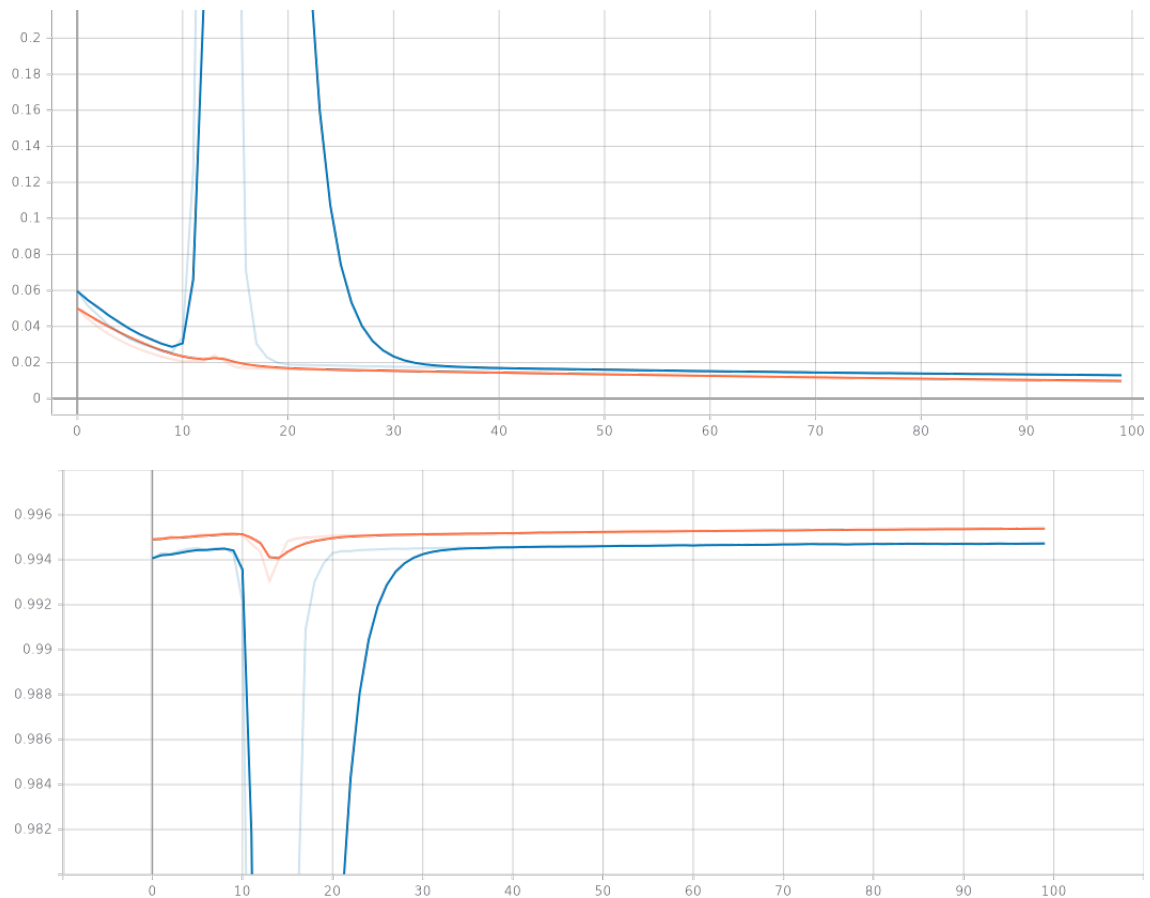


Figura 5.7: Andamento delle curve di perdita e accuratezza relativamente alle epoche di addestramento. In arancione vengono riportati i valori di perdita/accuratezza sul *Test Set* mentre in blu quelli sul *Validation Set*. Da notare come nelle epoche fra 11 e 21 alcune variazioni dei pesi che hanno prodotto variazioni minime nella perdita e accuratezza sul *Test Set* abbiano portato ad una grande perdita nel *Validation Set*.

accuratezza minime al fine di migliorare i tempi di esecuzione su GPU. Tutti i risultati ottenuti dalle varie reti verranno analizzati nel capitolo successivo.

### 5.4.1 Modello Ridotto

Un primo approccio all'ottimizzazione può essere effettuato andando a modificare leggermente l'architettura della rete in modo da renderla più efficiente. È possibile produrre una rete di dimensioni ridotte andando a ridurre la dimensione del tensore di input (senza influenzare eccessivamente l'accuratezza). Una dimensione di input minore comporta una dimensione ridotta dei percorsi di *encoder* e *decoder* dovuta ai minori strati necessari. La nuova rete ha quindi un tensore di input di forma (144, 256, 1) ed è definita come segue:

---

```
1 def get_unet(n_filters = 16, dropout = 0.1, batchnorm = True):
```

```
2
3 input_img = Input(shape=(img_height, img_width, 1))
4 #Contracting path
5 c1 = conv2d_block(input_img, n_filters, kernel_size=3, batchnorm
6   =batchnorm)
7 p1 = MaxPooling2D((2,2))(c1)
8 p1 = Dropout(dropout)(p1)
9
10 c2 = conv2d_block(p1, n_filters*2, kernel_size=3, batchnorm=
11   batchnorm)
12 p2 = MaxPooling2D((2,2))(c2)
13 p2 = Dropout(dropout)(p2)
14
15 c3 = conv2d_block(p2, n_filters*4, kernel_size=3, batchnorm=
16   batchnorm)
17 p3 = MaxPooling2D((2,2))(c3)
18 p3 = Dropout(dropout)(p3)
19
20 c4 = conv2d_block(p3, n_filters*8, kernel_size=3, batchnorm=
21   batchnorm)
22 p4 = MaxPooling2D((2,2))(c4)
23 p4 = Dropout(dropout)(p4)
24
25 c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3,
26   batchnorm=batchnorm)
27
28 #Expansive path
29 u6= Conv2DTranspose(n_filters*8, (3,3),
30   strides=(2,2), padding='same')(c5)
31 u6 = concatenate([u6, c4])
32 u6 = Dropout(dropout)(u6)
33 c6 = conv2d_block(u6, n_filters*8, kernel_size=3,
34   batchnorm=batchnorm)
35
36 u7 = Conv2DTranspose(n_filters*4, (3,3),
37   strides=(2,2), padding='same')(c6)
38 u7 = concatenate([u7, c3])
39 u7 = Dropout(dropout)(u7)
40 c7 = conv2d_block(u7, n_filters*4, kernel_size=3,
41   batchnorm=batchnorm)
42
43 u8 = Conv2DTranspose(n_filters*2, (3,3),
44   strides=(2,2), padding='same')(c7)
45 u8 = concatenate([u8, c2])
46 u8 = Dropout(dropout)(u8)
47 c8 = conv2d_block(u8, n_filters*2, kernel_size=3,
48   batchnorm=batchnorm)
49
50 u9 = Conv2DTranspose(n_filters, (3,3),
51   strides=(2,2), padding='same')(c8)#
52 u9 = concatenate([u9, c1])
53 u9 = Dropout(dropout)(u9)
54 c9 = conv2d_block(u9, n_filters, kernel_size=3,
```

```

50         batchnorm=batchnorm)
51
52     outputs = Conv2D(1, (1,1), activation='sigmoid',
53                    dtype='float32')(c9)
54     model = Model(inputs=[input_img], outputs=[outputs])
55     return model

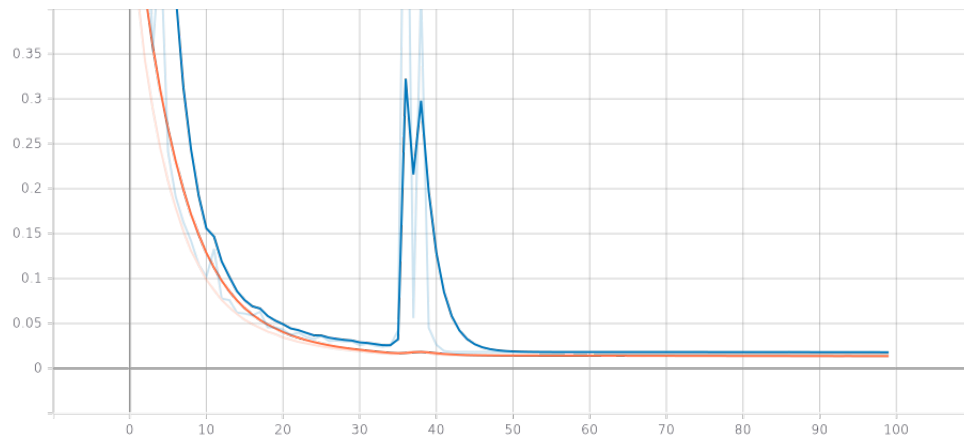
```

con la seguente tabella di caratteristiche:

	U-Net small
Numero strati (esclusi Dropout e Concatenazioni)	45
Parametri totali	543.017
parametri addestrabili	541.545
parametri non addestrabili	1.472

Tabella 5.2: Riassunto implementazione U-net ridotta

Com'è possibile notare dalla tabella 5.2 la riduzione della dimensione del tensore di input comporta una riduzione degli strati (da 55 a 45) ed una sensibile riduzione dei parametri complessivi della rete (da oltre 8 milioni a 500.000 ca.). Il modello ridotto è stato poi addestrato sul dataset utilizzando gli stessi parametri del modello completo. Di seguito vengono riportati gli andamenti dei valori di perdita e accuratezza durante l'addestramento.



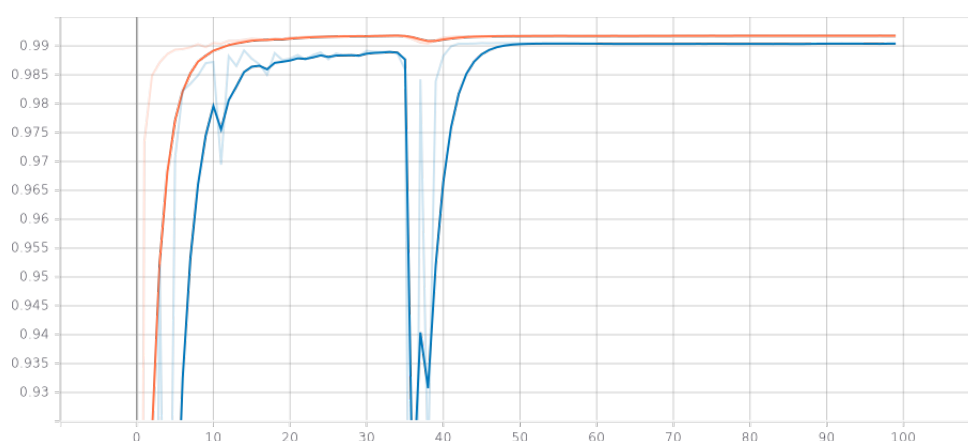


Figura 5.8: Andamento dei valori di perdita e accuratezza nella fase di addestramento della rete ridotta. In blu viene riportata la perdita/accuratezza sul *Validation set* e in arancione sul *Test Set*.

### 5.4.2 Modello Lite

Prima dell'introduzione della versione 2.0 delle librerie *Tensorflow* il principale sistema di esecuzione di algoritmi di *Machine Learning* e in particolare di *Deep Learning* (il più oneroso) era l'*Edge AI*. Questa tecnica consiste nell'utilizzare dispositivi a ridotta capacità di calcolo (come microcontrollori, smartphone, ecc.) semplicemente come dispositivi di acquisizione dei dati. Questi dati vengono poi inviati a *Cloud Server* ad elevata capacità di calcolo su cui viene effettuata l'inferenza e i risultati vengono poi inviati indietro al dispositivo mittente.



Figura 5.9: Rappresentazione schematica del sistema EdgeAI. [32]

Questo sistema deve rapportarsi con i tempi di trasmissione dei dati tramite una rete e può incorrere in problemi di latenza, banda, ma anche di privacy e sicurezza.

Tuttavia l'alternativa all'*Edge AI* comporta il dover lavorare con dispositivi con limitate capacità computazionali, memoria ridotta ed eventuali problemi di consumo energetico. Le librerie *Tensorflow 2.0* introducono quindi un *Toolkit* di sviluppo chiamato **Tensorflow Lite** pensato per ottimizzare il modello per lavorare con macchine a ridotta capacità computazionale riducendone dimensione e tempi di inferenza. *Tensorflow lite* consiste fondamentalmente di tre strumenti:



- **Convertitore**

Il convertitore ha il compito principale di convertire il modello creato (generalmente salvato con estensione *.pb* o *.tf*) in un modello tensorflow lite (salvato con estensione *.tflite*). Questo mette in luce la prima dinamica delle librerie *TFLite*, ovvero non è necessario addestrare nuovamente il modello. Quindi, sebbene non tutte le possibili operazioni compiute dalle reti siano supportate, è possibile convertire un modello pre-addestrato in un modello *TFLite* riducendo i tempi di sviluppo.

- **Interprete** Per eseguire l'inferenza di un modello *TensorFlow Lite* è necessario eseguirlo tramite un interprete. L'interprete si occupa quindi dell'esecuzione del modello e della gestione della memoria del dispositivo in modo da garantirne l'utilizzo minimo. La libreria è pensata per microcontrollori e smartphone e, di conseguenza, l'interprete è ottimizzato per lavorare con processori *ARM*<sup>4</sup> e può risultare quindi più lento nell'inferenza, rispetto al modello originale, se eseguito su processori di tipo *x86*<sup>5</sup> o simile.

- **Ottimizzazioni** Nella fase di conversione del modello è possibile applicare delle ottimizzazioni riguardanti principalmente la precisione dei pesi e le connessioni dei neuroni, le principali ottimizzazioni sono:

- *Quantizzazione a int8*
- *Riduzione dei pesi a float16*
- *Pruning*
- *Clustering*

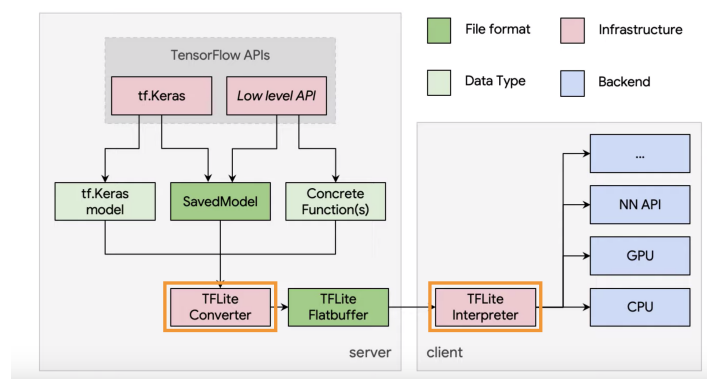


Figura 5.10: Panoramica del *ToolKite* di *Tensorflow Lite*. [33]

<sup>4</sup>L'architettura ARM (*Advanced RISC Machine*) è una famiglia di microprocessori RISC (*Reduced Instruction Set Computer*) a basso consumo energetico. Grazie all'ottimo rapporto fra consumo energetico e prestazioni, i processori ARM sono largamente diffusi in ambito Mobile e nei sistemi *Embedded*.

<sup>5</sup>l'architettura *x86* è un'architettura per microprocessori prodotti da *Intel* largamente utilizzata in PC Desktop e laptop.

### 5.4.3 Quantizzazione a int8

La quantizzazione a `int8` è una particolare tecnica di conversione del modello che *riduce* i pesi dal formato `float32` ad un formato `int8`. Questa operazione consente di ridurre i tempi di *latenza* e la dimensione totale del modello con una perdita di precisione minima. L'utilizzo di una precisione `int8` comporta vantaggi che sono strettamente legati a come le CPU eseguono i calcoli e lavorano con le loro *cache*, in particolare:

- Sebbene su quelli moderni la differenza sia minima, i processori, specialmente gli ARM, lavorano molto più velocemente con operazioni aritmetiche a `int8` rispetto a `float32`
- la rappresentazione `int8` è quattro volte più piccola rispetto alla `float32`, il che significa che il modello complessivo avrà dimensioni ridotte. Inoltre questo consente un maggiore utilizzo di registri e cache da parte del processore che può quindi ridurre il numero di accessi alla RAM (con tempi di accessi molto più lunghi) e migliorare le prestazioni
- l'aritmetica a virgola mobile è molto complessa e alcuni microcontrollori potrebbero non essere ottimizzati per questo tipo di lavoro.

Le reti neurali profonde si sono dimostrate sufficientemente insensibili al *Rumore* o altri tipi di perturbazione (dopo essere state addestrate) e i pesi di ogni strato rientrano in un range non molto elevato. È quindi possibile ridurre la precisione di rappresentazione senza intaccare l'intervallo completo dei pesi. Questo implica che la quantizzazione di un modello riesce a ridurre i tempi di latenza con perdite minime nell'accuratezza del modello complessivo.

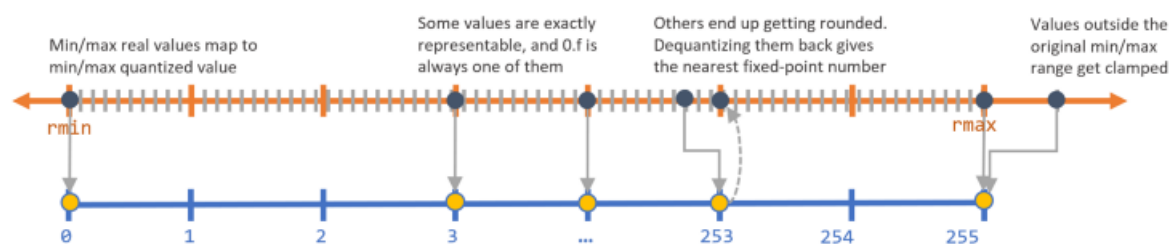


Figura 5.11: Esempio di quantizzazione. La linea arancione rappresenta l'intervallo completo dei valori, mentre la linea blu i valori quantizzati. [34]

Affinché il processo di conversione sia quanto più preciso possibile, *Tesnorflow Lite* richiede un *Representative Dataset*, ovvero una parte del dataset di allenamento che verrà utilizzato per stimare i valori massimi e minimi entro quale ricadono i tensori di ingresso e uscita della rete. Di seguito viene riportato il codice python per la conversione del modello *TFLite* e la quantizzazione a `int8`.

```
1 def representative_dataset():
2     for data in images_tot:
```

```

3     data = np.expand_dims(data, axis=0)
4     yield [data]

```

---

```

1 converter = tflite.TFLiteConverter.from_saved_model('model')
2 converter.optimizations = [tflite.Optimize.DEFAULT]
3 converter.representative_dataset = tflite.RepresentativeDataset(
4     representative_dataset)
5 converter.target_spec.supported_ops = [tf.lite.OpsSet.
6     TFLITE_BUILTINS_INT8]
7 converter.inference_input_type = tf.int8
8 converter.inference_output_type = tf.int8
9 tfl_model_quant = converter.convert()
10
11 with open('model_small_int8.tflite', 'wb') as f:
12     f.write(tfl_model_quantGD)

```

---

Nella tabella 5.3 vengono riportati i principali parametri dei quattro modelli implementati.

	Modello completo	Modello Ridotto	Modello Lite	Modello Lite Quant
Forma Input	(240x400x1)	(144x256x1)	(144x256x1)	(144x256x1)
Dimensione Modello	106.5Mb	8.6Mb	2.2Mb	581.4Kb
N° Strati	55	45	45	45
N° Parametri	8.660.305	543.017	543.017	543.017
Precisione	float32	float32	float32	int8

Tabella 5.3: Tabella riassuntiva dei vari modelli

# Capitolo 6

## Risultati

Nel presente capitolo vogliamo andare ad analizzare i risultati ottenuti dalle diverse reti in termini di accuratezza e tempi di esecuzione. Per eseguire le valutazioni sull'accuratezza della rete sono state scelte dieci immagini di piste di atletica da internet (che rispecchiassero la tipologia di angolazione e pista delle immagini utilizzate per lo sviluppo del progetto) che la rete non aveva mai analizzato. Ne sono state create delle etichette per poterne valutare l'accuratezza rispetto alla previsione desiderata e sono state poi fatte analizzare dalle varie reti. Alcune delle immagini sottoposte alle reti presentano elementi di confusione, come ad esempio linee tratteggiate o scritte sulla pista. Queste tipologie di immagini non erano presenti nel dataset di allenamento (la pista era uniforme, senza linee tratteggiate o scritte) ed è quindi interessante vedere come le reti reagiscano a questo tipo di immagini per avere un'idea di come queste riescano a generalizzare il problema ricavando soluzioni quanto più precise possibile.

L'analisi dei tempi di esecuzione delle varie reti si è svolta su differenti piattaforme. Questo perché al fine pratico la rete dovrà essere messa in esecuzione su dispositivi a ridotte capacità computazionali quindi, sebbene i test siano stati svolti anche sulla piattaforma *Google Colaboratory* questi risultano ininfluenti nell'analisi dei tempi di esecuzione, ma sono comunque un buon punto di riferimento per comprendere la differenze nei tempi di esecuzione per differenti tipologie di hardware. I test più rilevanti sono stati svolti sulle seguenti schede:

- **NVIDIA Jetson Nano** [35]

La *Jetson Nano* è una scheda prodotta dalla società informatica **NVIDIA**, principalmente nota per lo sviluppo di GPU, ottimizzata per il lavoro nel campo dell'intelligenza artificiale con consumi energetici ridotti. La scheda è dotata di una *CPU ARM* con 4 core ed una frequenza di lavoro massima di 1.43Ghz affiancata da una *GPU Maxwell* basata sul chip *Erista* di NVIDIA (*Tegra X1*) con un ridotto numero di core (128 cores rispetto ai 256 di un *tegra x1* completamente attivo). Questo consente alla scheda di avere Buone capacità computazionali (grazie soprattutto all'accelerazione GPU) con un ridotto consumo energetico stimato intorno ai 5 watt.

- **Google Coral Dev Board Mini** [36]

La *Dev Board Mini* di *Google coral* è una scheda prodotta ed ottimizzata per il lavoro con le reti neurali e, in particolare, con la libreria *Tensorflow*. Il progetto *Coral* di *Google* nasce a metà del decennio scorso insieme allo sviluppo, sempre da parte di *Google*, di particolari unità di calcolo dette *TPU* (*Tensor Processing Unit*).

Le *TPU* vengono presentate da *Google* nel 2016 come *Acceleratori per l'AI di tipo ASIC*<sup>1</sup> pensati ed ottimizzati per il lavoro con dati di tipo tensoriale [38]. A differenza delle GPU (che sono comunque dei dispositivi ASIC ottimizzati per i calcoli con matrici e il lavoro con la renderizzazione 2D e 3D) le *TPU* sono progettate per eseguire un elevato numero di calcoli a bassa precisione (fino a 8 bit) motivo per cui vengono molto spesso utilizzate con modelli *Tensorflow Lite* quantizzati.

Sebbene lo sviluppo delle *TPU* da parte di *Google* sia iniziato nel 2016, queste sono rimaste per diversi anni proprietarie e non disponibili alla vendita. Solo nel 2018, con lo sviluppo delle *Edge TPU* e la nascita del brand *Coral*, queste sono divenute disponibili per il mercato. La *Dev Board Mini* è provvista di un processore ARM *MediaTek 8167s SoC* con GPU integrata e, chiaramente, un acceleratore *TPU Google Edge TPU coprocessor* in grado di compiere fino a 4 TOPS<sup>2</sup> a int8 al secondo con un consumo energetico di 0.5 watts per TOPS.

Tutte le analisi sui tempi di inferenza del modello sono state svolte sulle medesime immagini fornite per il test di valutazione dell'accuratezza. A fini pratici è stato valutato come tempo di esecuzione il solo tempo necessario alla predizione da parte del modello mentre sono stati esclusi i tempi necessari alla preparazione delle immagini (ridimensionamento e conversione a *grayscale*), questo perché possiamo supporre che le immagini fornite dalla telecamera del dispositivo siano già nel formato e nello spazio di colore corretto (riducendo i tempi di esecuzione).

## 6.1 Modello Completo

Iniziamo l'analisi dei risultati a partire dal modello più grande e privo di ottimizzazioni. I risultati ottenuti per questo modello verranno utilizzati come punto di riferimento nella valutazione delle reti ottimizzate in modo da valutare quali siano la perdita nell'accuratezza e il guadagno nei tempi di inferenza rispetto al modello completo.

Ricordiamo che il modello prevede un input di forma  $(240 \times 400 \times 1)$  con 55 strati e oltre 8 milioni di parametri addestrabili.

---

<sup>1</sup>ASIC, *Application-specific Integrated Circuit* è una classe di *Circuiti Integrati* ottimizzati per uno specifico utilizzo a discapito di una funzionalità *General Purpose* [37]

<sup>2</sup>*Tera operation*, equivalente di un *trilione di operazioni* ( $10^{18}$ )

### 6.1.1 Analisi delle previsioni

Riportiamo nella tabella 6.1 i risultati ottenuti relativamente alle 10 immagini di prova.

Tabella 6.1: Tabella dei risultati del Modello Completo




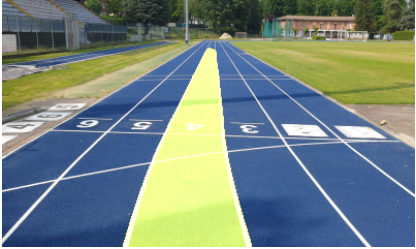





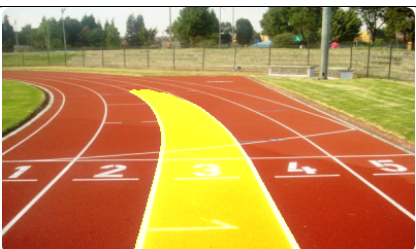










Immagine	Previsione	Acc.
		0.99
		0.99
		0.99
		0.99
		0.99



Tabella 6.1: Tabella dei risultati del Modello Completo

Immagine	Previsione	Acc.
		0.99
		0.99
		0.99
		0.99
		0.99

Come è possibile osservare la rete riesce a predire con elevata accuratezza la posizione della corsia centrale presentando una riduzione di precisione (in alcuni casi) solo nella sezione più lontana della pista (dove i pixel dell'immagine cominciano a confondersi). Questa riduzione della precisione è tuttavia irrilevante poiché, al fine del progetto, è importante conoscere con precisione la posizione della pista nell'area prossima all'atleta.

Particolare interesse va alle immagini 6 e 9. Queste presentano rispettivamente un banner con una scritta sulla pista e la sovrapposizione di due piste con delle linee tratteggiate. Entrambi questi casi non rientrano nel dataset e le predizioni effettuate mostrano quindi un'ottima capacità di generalizzazione del problema da parte della rete stessa (bisogna ricordare che il numero di immagini del dataset era piuttosto piccolo).

### 6.1.2 Tempi di esecuzione

Passiamo ora ad analizzare i tempi di inferenza della rete su differenti dispositivi andando quindi a valutarne le prestazioni. Questo modello, poiché non convertito in un modello *Tensorflow lite*, privo quindi di quantizzazione, non è compatibile con l'esecuzione su *edge TPU* e, di conseguenza, la *Dev Board Mini* ne sposterebbe l'esecuzione su CPU. Per questi motivi omettiamo i risultati sul dispositivo *Coral*

modello	img. size	Google			
		GPU		CPU	
Modello Completo	240	0.05	0.05	0.3	0.29
		0.05	0.04	0.29	0.28
		0.04	0.04	0.28	0.28
		0.05	0.04	0.29	0.3
		0.04	0.06	0.28	0.3

Tabella 6.2: Tempi di esecuzione del modello completo su CPU e GPU di *Google Colaboratory*

modello	img. size	Jetson Nano			
		GPU		CPU	
Modello Completo	240	0.33	0.33	0.91	0.89
		0.31	0.31	0.89	0.89
		0.31	0.33	0.89	0.89
		0.31	0.31	0.90	0.89
		0.33	0.30	0.89	0.89

Tabella 6.3: Tempi di esecuzione del modello completo su CPU e GPU della Jetson Nano



Come è possibile vedere nella tabella 6.2 i tempi di inferenza del modello sui server cloud di *Google Colaboratory* su GPU (*NVIDIA Tesla T4*) risultano accettabili per un'applicazione di tipo *realtime*. Con tempi di esecuzione medi di circa 45 ms abbiamo una possibilità di analisi fino a 22 frame al secondo (circa) il che produce comunque un margine di errore relativamente troppo piccolo (supponendo i 30 frame al secondo di ripresa di una fotocamera standard) e, come detto in precedenza, questi test sono irrilevanti per la nostra applicazione.

La tabella 6.3 mostra i tempi di esecuzione del modello sulla scheda *Jetson Nano*. La degradazione dei tempi di inferenza risulta evidente (da 45 ms su GPU Google a 300 ms su GPU NVIDIA) con margini largamente fuori scala rispetto ad un'applicazione *realtime*.

## 6.2 Modello Ridotto

Procediamo quindi con l'analisi delle previsioni e dei tempi di inferenza del modello ridotto. Anche in questo caso i test sono stati effettuati sulle immagini di test e l'esecuzione sulla scheda *Dev Board Mini* viene omessa per le medesime problematiche del modello Completo.

Ricordiamo che il modello è caratterizzato da un input di forma (144x256x1) con 45 strati e poco più di 500.000 parametri addestrabili.

### 6.2.1 Analisi delle previsioni

Nella tabella 6.4 è possibile osservare le previsioni del modello ridotto sulle immagini di test.

Tabella 6.4: Tabella dei risultati del Modello Ridotto





Immagine	Previsione	Acc.
		0.99
		0.97

Tabella 6.4: Tabella dei risultati del Modello Ridotto


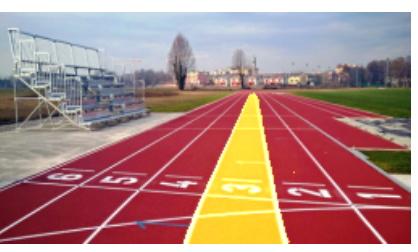


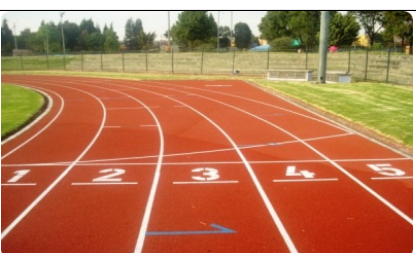
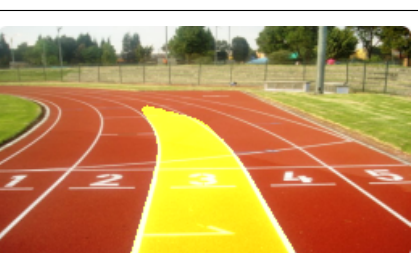
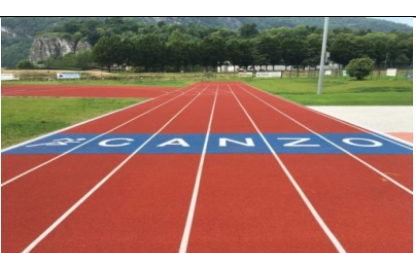
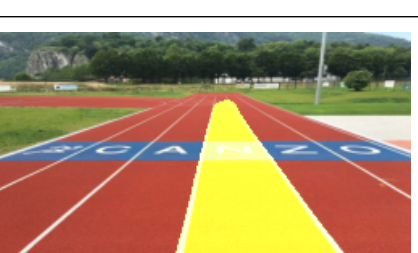
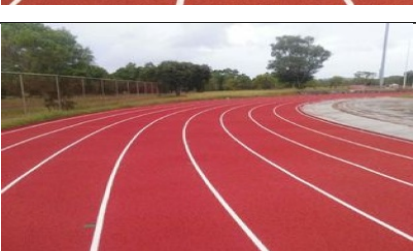
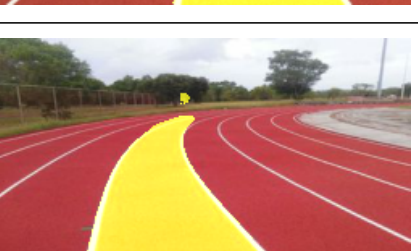
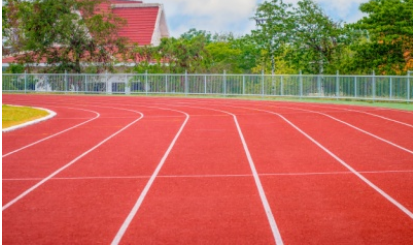
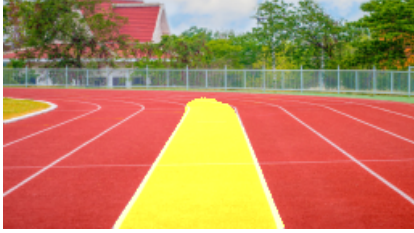




Immagine	Previsione	Acc.
		0.99
		0.98
		0.99
		0.98
		0.98

Tabella 6.4: Tabella dei risultati del Modello Ridotto

Immagine	Previsione	Acc.
		0.98
		0.99
		0.98

La riduzione della dimensione di input e quindi della profondità della rete produce una perdita di accuratezza minima che, anche in questo caso, si presenta principalmente nella parte finale della pista e, come abbiamo detto in precedenza, la perdita di accuratezza in quest'area rappresenta un problema limitato. Anche il modello ridotto, esattamente come il modello completo, produce previsioni corrette per le immagini 6 e 9 confermando la capacità della rete di generalizzare anche nel caso ridimensionato.

### 6.2.2 Tempi di esecuzione

Di seguito vengono riportate le tabelle relative ai tempi di inferenza del modello ridotto sui server cloud Google e sulla scheda Jetson Nano. Com'è possibile osservare nella tabella 6.5 i tempi di esecuzione del modello ridotto su GPU nei cloud Google sono leggermente migliorati, passando da una media di 45 ms ad una media di 37 ms. Nella tabella 6.6 è possibile osservare i tempi di esecuzione del modello ridotto su Jetson Nano. Anche in questo caso l'esecuzione su GPU NVIDIA è migliorata, passando da una media di 300 ms ad una media 200 ms. Questi miglioramenti sono tuttavia ancora insufficienti per un'applicazione *realtime* (con una media di 200 ms abbiamo una possibilità di analisi fino 5 frame al secondo, il che è chiaramente insufficiente).

Un particolare interessante è invece l'elevato miglioramento dei tempi di esecuzione su CPU che passano da una media di 900ms (quasi un secondo ad immagine) ad una media di circa 265 ms per immagine (quasi prossimi ai tempi di GPU).

modello	img. size	Google			
		GPU		CPU	
Modello Ridotto	144	0.05	0.03	0.1	0.09
		0.03	0.03	0.08	0.08
		0.03	0.05	0.08	0.08
		0.05	0.04	0.08	0.08
		0.03	0.03	0.08	0.1

Tabella 6.5: Tempi di esecuzione del modello ridotto su GPU e CPU Google

modello	img. size	Jetson Nano			
		GPU		CPU	
Modello Ridotto	144	0.20	0.21	0.27	0.27
		0.22	0.20	0.27	0.26
		0.21	0.20	0.26	0.26
		0.19	0.20	0.27	0.26
		0.19	0.20	0.26	0.27

Tabella 6.6: Tempi di esecuzione del modello ridotto su Jetson Nano

### 6.3 Modello Ridotto Lite

Procediamo adesso ad analizzare i risultati ottenuti dal modello ridotto ottimizzato tramite l'utilizzo delle librerie *Tensorflow Lite*. Prima di procedere è necessaria una precisazione rispetto all'inferenza del modello. Le librerie *TFLite* offrono la possibilità di produrre un modello leggero ed ottimizzato per il lavoro su hardware meno potenti; tuttavia, il progetto nasce inizialmente come strumento di supporto per le applicazioni di *Artificial Intelligence* su dispositivi mobile. Affinché l'interprete di tensorflow lite possa interfacciarsi con la GPU di sistema è necessario utilizzare dei *GPU Delegate*, ovvero degli strumenti, parte delle API di *Tensorflow Lite*, che permettono l'esecuzione del modello su GPU. Come detto in precedenza questi strumenti sono pensati per interfacciarsi con dispositivi mobile



e, attualmente, non sono compatibili con le GPU NVIDIA. Il modello lite è stato quindi eseguito solo su CPU sia su Jetson Nano che su cloud Google (che sfrutta comunque GPU NVIDIA).

### 6.3.1 Analisi delle Previsioni

Nella tabella 6.7 vengono riportate le predizioni e l'accuratezza del modello lite relativamente alle immagini di test

Tabella 6.7: Tabella dei risultati del Modello Ridotto Lite









Immagine	Previsione	Acc.
		0.99
		0.97
		0.99
		0.98

Tabella 6.7: Tabella dei risultati del Modello Ridotto Lite


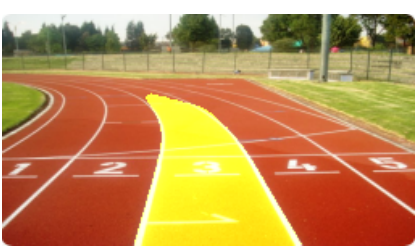

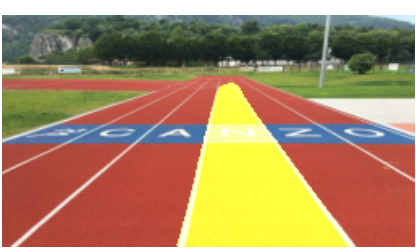

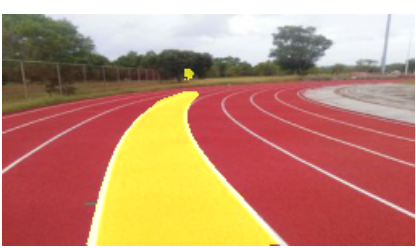



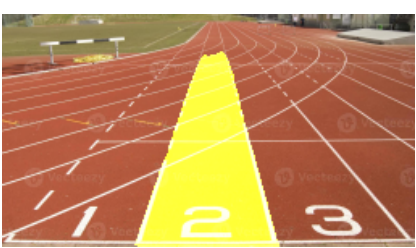


Immagine	Previsione	Acc.
		0.99
		0.98
		0.98
		0.98
		0.99

Tabella 6.7: Tabella dei risultati del Modello Ridotto Lite

Immagine	Previsione	Acc.
		0.98

La conversione del modello ridotto in un modello *Tensorflow Lite* mostra perdite nell'accuratezza praticamente irrilevanti. Possiamo concludere che la conversione del modello non ha mostrato alcun tipo di perdita.

### 6.3.2 Tempi di esecuzione

modello	img. size	Google			
		GPU		CPU	
Modello Ridotto Lite	144	/	/	0.05	0.05
				0.04	0.04
				0.05	0.04
				0.05	0.05
				0.05	0.05

Tabella 6.8: Tempi di esecuzione del modello ridotto lite su CPU Google

modello	img. size	Jetson Nano			
		GPU		CPU	
Modello Ridotto Lite	144	/	/	0.14	0.1
				0.1	0.1
				0.1	0.1
				0.1	0.1
				0.1	0.1

Tabella 6.9: Tempi di esecuzione del modello ridotto lite su CPU ARM Jetson



Le tabelle 6.8 e 6.9 mostrano i risultati dei tempi di inferenza su CPU nei server cloud Google e sulla Jetson Nano. L'impossibilità di eseguire questi modelli su GPU rende l'inferenza dei modelli piuttosto complessa. Tuttavia, la scheda Jetson, malgrado l'esecuzione su CPU, ottiene un notevole miglioramento nei tempi di inferenza passando dai 200 ms della GPU sul modello ridotto ai 100 ms su CPU del modello ridotto lite. Questo miglioramento è ancora insufficiente rispetto alla specifiche di un'applicazione *realtime* (in questo caso riusciamo a raggiungere velocità di analisi di circa 10 frame al secondo), ma il fatto che questi tempi siano stati calcolati su CPU lascia comunque un buon margine per un eventuale esecuzione del modello su GPU laddove venisse introdotta la compatibilità con GPU NVIDIA per *Tensorflow Lite*.

## 6.4 Modello Ridotto Lite Quantizzato

Come ultima fase andremo ad analizzare i risultati ottenuti con il modello *tensorflow Lite* quantizzato a `int8`. Anche in questo caso il modello non presenta compatibilità per l'esecuzione su GPU NVIDIA, tuttavia il modello è perfettamente compatibile con l'accelerazione edge TPU coral e ne verranno riportati i tempi di esecuzione sulla *Dev Board Mini*.

### 6.4.1 Analisi delle previsioni

Nella tabella 6.10 vengono riportate le previsioni e l'accuratezza del modello quantizzato sulle immagini di test.

Tabella 6.10: Tabella dei risultati del Modello Ridotto Lite Quantizzato





Immagine	Previsione	Acc.
		0.99
		0.97



Tabella 6.10: Tabella dei risultati del Modello Ridotto Lite Quantizzato






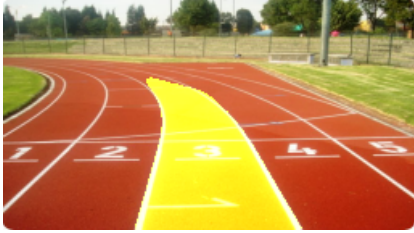

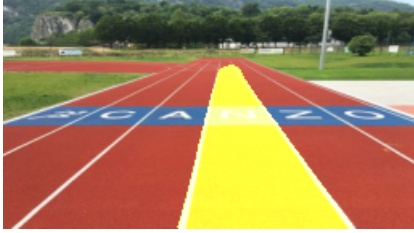

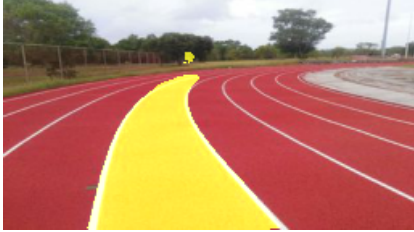
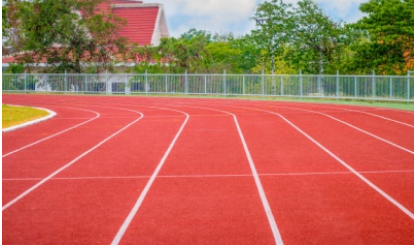


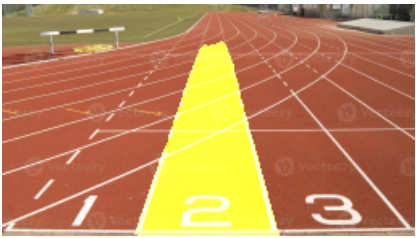

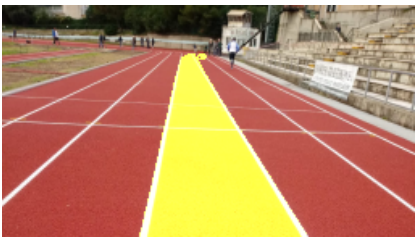
Immagine	Previsione	Acc.
		0.99
		0.98
		0.99
		0.98
		0.98

Tabella 6.10: Tabella dei risultati del Modello Ridotto Lite Quantizzato

Immagine	Previsione	Acc.
		0.98
		0.99
		0.98

Com'è possibile notare la rete non mostra predite di precisione visibili e i risultati di previsione sono del tutto simili a quelli ottenuti dal modello ridotto lite e dal modello ridotto.

### 6.4.2 Tempi di esecuzione

Riportiamo quindi i risultati relativi ai tempi di esecuzione del modello quantizzato sulle piattaforme server cloud di Google, sulla Jetson Nano e sulla Dev Board Mini Coral. Nella tabella 6.11 vengono riportati i tempi di inferenza su CPU nei server cloud di *Google Colaboratory* ed è interessante vedere come il modello abbia mostrato un sensibile rallentamento. Come detto in precedenza i modelli *Tensorflow lite* e la quantizzazione sono operazioni di ottimizzazione per dispositivi a ridotta capacità computazionale e sono quindi pensati per interfacciarsi principalmente con Processori ARM. L'inferenza su una CPU di tipo x86, sebbene con una maggiore capacità di calcolo, produce tempi di inferenza più lunghi dovuti alla conversione delle istruzioni macchina prodotte dall'interprete per processori ARM a istruzioni macchina per processori x86.

modello	img. size	Google			
		GPU		CPU	
Modello Ridotto Lite	144	/	/	1.19	1.17
				1.15	1.15
				1.16	1.16
				1.15	1.15
				1.16	1.17

Tabella 6.11: Tempi di esecuzione del modello ridotto quantizzato su CPU Google

Nella tabella 6.12 vengono riportati i tempi di inferenza sulla CPU della Jetson Nano. In questo caso, come era possibile aspettarsi, abbiamo un miglioramento dei tempi di esecuzione del modello che passa da 100 ms per immagine a circa 70 ms per immagine. Ancora una volta, questo miglioramento, con una capacità di analisi di circa 14 frame al secondo, non è sufficiente e fornisce margini di errore non adatti ad un'applicazione *realtime*. Tuttavia, quanto detto per il modello lite è ancora valido, ovvero rimane incertezza sulla GPU e su una possibile futura compatibilità.

modello	img. size	Jetson Nano			
		GPU		CPU	
Modello Ridotto Lite	144	/	/	0.09	0.07
				0.07	0.07
				0.07	0.07
				0.07	0.07
				0.07	0.07

Tabella 6.12: Tempi di esecuzione del modello ridotto quantizzato su CPU ARM jetson

Molto interessanti sono invece i risultati ottenuti dal modello in esecuzione su TPU nella *Dev Board Mini* riportati nella tabella 6.13. I tempi di inferenza subiscono un miglioramento netto dice volte superiore ai precedenti passando da tempi di inferenza di 70 ms per immagine a soli 7 ms per immagine. Questo miglioramento fornisce tempi di inferenza più che sufficienti per un'applicazione *realtime* con una possibilità di analisi di quasi 140 immagini al secondo che forniscono margini di errore molto larghi per ulteriori computazioni legate alla natura del progetto. Il modello messo in esecuzione sulla Dev Board Mini è un modello

modello	img. size	Dev Board Mini			
		TPU		CPU	
Modello Ridotto	144	0.01	0.007	0.15	0.16
		0.007	0.007	0.16	0.16
		0.007	0.007	0.16	0.16
		0.007	0.007	0.16	0.16
		0.007	0.007	0.16	0.15

Tabella 6.13: Tempi di esecuzione del modello ridotto quantizzato su TPU e CPU Dev Board Mini

basato interamente sul modello quantizzato, ma subisce una compilazione ulteriore per la compatibilità con l'*Edge TPU*. Questa compilazione fornisce una mappatura dei vari livelli selezionando quali possono essere eseguiti su TPU e quali invece non sono supportati e vengono di conseguenza eseguiti su CPU. Nel nostro caso tutti i livelli erano compatibili con la TPU e di conseguenza l'intero modello viene eseguito da questa. I tempi di inferenza mostrano come l'accelerazione TPU, con le dovute ottimizzazioni del modello, possa migliorare le prestazioni in maniera sensibile rendendo l'esecuzione di algoritmi di *Deep Learning* possibile anche per macchine a ridotto capacità computazionale.



# Capitolo 7

## Conclusioni e Sviluppi Futuri

Lo sviluppo di un algoritmo di *deep learning* per il *lane detect* nell'ambito della corsa di atleti ipovedenti è stata un'esperienza interessante ed altamente formativa che ha messo in mostra le potenzialità di una tecnologia relativamente nuova con una ricerca costante ed in continua evoluzione.

La progettazione dei vari modelli pensati in funzione delle varie ottimizzazioni effettuate ha portato allo sviluppo di differenti reti che si sono mostrate sufficientemente stabili nelle previsioni e, con le giuste ottimizzazioni, efficienti in termini di tempi di inferenza e consumi energetici. Sebbene ci si possa ritenere soddisfatti del modello complessivamente ottenuto è doveroso precisare come sia ancora possibile e, in alcuni casi, doveroso effettuare ulteriori test ed ottimizzazioni. Gli aspetti di maggiore interesse sono:

- **Ampliamento del Dataset**

Come detto in precedenza il dataset di riferimento utilizzato nell'addestramento delle reti è piuttosto piccolo e, sebbene queste abbiano mostrato un'ottima capacità di generalizzare durante i test, ampliare il dataset con varie immagini, specialmente riprese durante le fasi di corsa di atleti professionisti e con piste con caratteristiche differenti possono permettere alle reti di ottenere risultati accurati in varie piste e con immagini non perfettamente a fuoco o angolazioni particolari.

- **Test su hardware**

Anche in questo caso i test relativi ai tempi di inferenza dei modelli sono stati svolti su due schede (una principalmente) che, sebbene sufficienti per capire l'andamento temporale delle previsioni, possono essere ampliati a vari dispositivi per valutare in modo più completo l'inferenza dei modelli. Inoltre molti dei test sui modelli *Tensorflow Lite* non hanno potuto sfruttare le accelerazioni hardware della GPU e risulta quindi interessante fare dei test su schede compatibili con i *GPU Delegate* di *Tensorflow*.

Un'ulteriore aspetto d'interesse è l'implementazione di un algoritmo per il rilevamento della traiettoria dell'atleta. L'obiettivo del progetto è quello di sfruttare

le predizioni ottenute dal modello, relativamente alla corsia, per valutare l'andamento dell'atleta. Questo significa che l'algoritmo di rilevamento della traiettoria non può prescindere dal tipo di dato fornito in output dalla rete del modello descritto. Di seguito riportiamo un possibile sistema di predizione della traiettoria (non implementato) basato sulla tipologia di dati prodotti in uscita dal nostro modello.

### 7.0.1 Rilevamento Della Traiettoria

Come ampiamente discusso nei capitoli precedenti l'output delle reti sviluppate nel progetto sono immagini in *grayscale* normalizzate<sup>1</sup> dalle quali è possibile, tramite l'utilizzo di un valore di soglia, ottenere delle immagini binarie<sup>2</sup>. Sappiamo inoltre che per ogni frame siamo interessati a conoscere la posizione dell'atleta relativamente alla pista solo in una determinata regione dello spazio posta nell'area in basso dell'immagine (in prossimità dell'atleta). Possiamo quindi pensare di estrarre dall'output della rete una sotto-matrice di dimensione  $(B \times H)$  e posizione  $(x, y)$  prefissate come mostrato nella fig. 7.1.

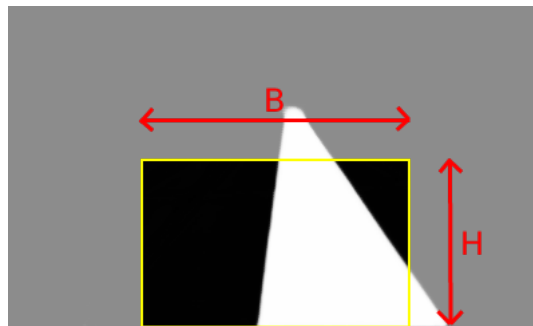


Figura 7.1: Sotto-matrice estratta dall'immagine binaria. Questa sotto-matrice può essere vista analogamente ad una *Region Of Interest (ROI)* la cui dimensione e posizione sono note a priori.

Questa regione rappresenta l'area all'interno della quale vogliamo andare a valutare la posizione e la traiettoria dell'atleta.

Anche in questo caso possiamo ridurre ulteriormente il numero di dati andando ad estrarre due ulteriori sotto-matrici rispettivamente di dimensione  $(B \times c)$  e  $(c \times H)$  posizionate come mostrato nella fig. 7.2. La posizione dei valori 1 e 0 all'interno di queste due sotto-matrici è sufficiente per discriminare la posizione dell'atleta sia lungo l'asse orizzontale che lungo l'asse verticale. Tuttavia questo sistema riesce a fornire informazioni relative alla sola posizione attuale dell'atleta. Affinché sia possibile discriminare la traiettoria abbiamo bisogno di informazioni ulteriori relative alla posizione dell'atleta in istanti di tempo precedenti. Possiamo quindi

<sup>1</sup>ogni pixel è rappresentato da un numero intero compreso fra 0 e 1

<sup>2</sup>Immagini i cui pixel possono assumere solo i valori 0 o 1

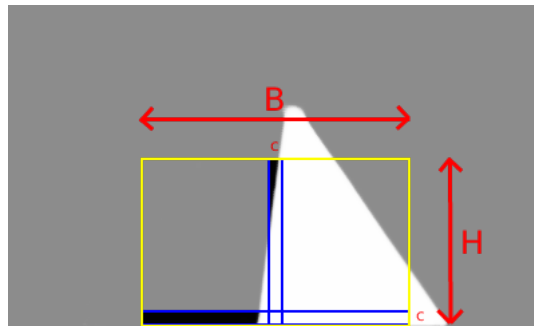


Figura 7.2: Evidenziate in blu le sotto matrici di posizione

impostare un numero di frame  $F$  e memorizzare le sotto-matrici di posizione di ciascuno di essi in modo che, nel momento dell'analisi del  $F$ -esimo frame avremo a disposizione  $2F$  sotto-matrici di posizione per valutare la traiettoria dell'atleta.

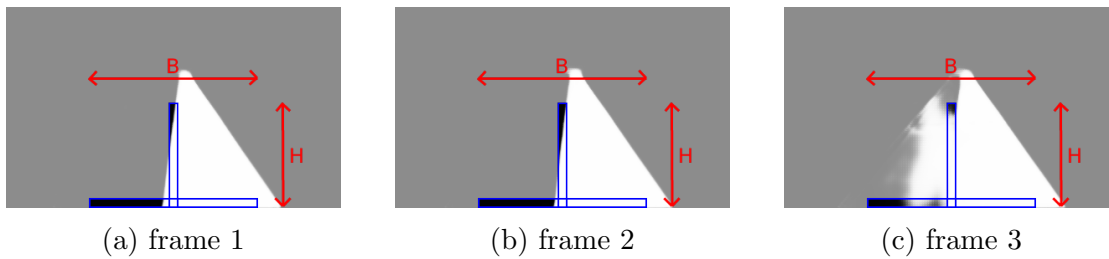


Figura 7.3: Analisi di tre frame consecutivi in una situazione di traiettoria uscente dalla corsia. Com'è possibile vedere la posizione dei valori 1 e 0 all'interno delle due sotto-matrici di posizione può essere sfruttata per determinare la traiettoria dell'atleta

A questo punto abbiamo a disposizione tutti i dati necessari per l'analisi della traiettoria dell'atleta. Possiamo procedere in modi differenti, ma per semplicità possiamo pensare a due distinti approcci, ovvero un **Algoritmo Iterativo** o una **Rete Neurale Fully Connected**.

### Algoritmo Iterativo

Un primo possibile processo di analisi della traiettoria può essere basato sullo sviluppo di un algoritmo iterativo che basa la sua analisi sulla posizione dei valori 1 e 0 all'interno delle sotto-matrici precedentemente estratte che, una volta determinato se la traiettoria rimane all'interno della pista o diverge da essa, produca in output un valore identificativo in grado di definire la direzione per rientrare all'interno della corsia. Questo algoritmo utilizza, oltre alle sotto-matrici di input, alcuni valori che fungono da iperparametri come le dimensioni  $B$ ,  $H$  e  $c$  delle matrici e l'intervallo di frame dopo cui viene avviata l'analisi. Per semplicità di analisi riportiamo di seguito i principali vantaggi e svantaggi di questo sistema di analisi, ma non una possibile implementazione:



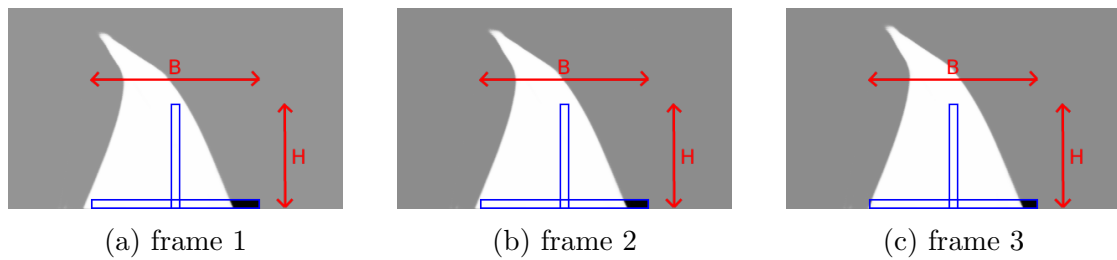


Figura 7.4: Analisi di tre frame consecutivi in una situazione di traiettoria no uscente. Anche in questo caso le posizioni dei valori di 0 e 1 all'interno delle sotto-matrici, differenti dalle precedenti, permettono di comprendere la traiettoria dell'atleta.

- **Vantaggi**

- **Semplicità di modifica degli iperparametri**

Uno dei principali vantaggi dell'utilizzo di un algoritmo iterativo è la possibilità di variare il valore degli iperparametri presentati in precedenza senza necessità di modificare l'algoritmo stesso. Questo implica un minor tempo di sviluppo, implementazione e ottimizzazione del codice.

- **Tempi di esecuzione ridotti**

Sebbene non sia possibile stimare i tempi di esecuzione di un algoritmo senza un'implementazione è verosimile pensare che questo abbia tempi di esecuzione più piccoli rispetto a quelli di una rete neurale completa.

- **Svantaggi**

- **Difficoltà di implementazione**

Uno dei principali svantaggi nell'utilizzo di un algoritmo iterativo sta nella complessità dell'algoritmo stesso. Non possiamo sapere, senza alcuni test, se sia possibile trovare un algoritmo in grado di svolgere questo compito e, laddove fosse possibile, non possiamo sapere con certezza quale sia la sua accuratezza.

- **Scarsa generalizzazione**

Un algoritmo iterativo non è in grado di modificarsi in funzione dell'errore commesso sulle previsioni, questo significa che questo approccio presenta una scarsa elasticità nella generalizzazione del problema e potrebbe portare a risultati imprevedibili in caso di variazioni sensibili dei dati di input.

## Rete Neurale FC

Un'ulteriore possibilità nello sviluppo di un sistema di riconoscimento della traiettoria è l'utilizzo di una *rete neurale fully connected* che prenda in input il  *tensore sfilacciato*<sup>3</sup> formato dall'unione delle sotto-matrici di posizione precedentemente estratte e che produca in output un valore identificativo in grado di definire la probabilità di traiettoria uscente e la direzione di rientro in corsia. Anche in questo caso riportiamo i possibili vantaggi e svantaggi di questo approccio:

- **Vantaggi**

- **Semplicità di sviluppo**

Come discusso in precedenza, una rete neurale è un particolare tipo di algoritmo in grado di modificare i propri parametri sulla base dell'errore commesso rispetto alle predizioni. Questo implica quindi una maggiore semplicità nello sviluppo dell'algoritmo stesso che non prevede lo sviluppo di passi particolari per la predizione.

- **Miglior generalizzazione**

Le reti neurali sono algoritmi con un'elevata capacità di generalizzazione del problema (sempre nei limiti imposti dell'*overfitting*) e di conseguenza una miglior capacità di previsione per variazioni sensibili dei dati di input.

- **Compatibilità con l'accelerazione hardware**

Il dispositivo di esecuzione dell'algoritmo è ottimizzato per lavorare con reti neurali e ne sfrutta le principali accelerazioni. L'utilizzo di una seconda rete neurale per il rilevamento della traiettoria può quindi sfruttare le medesime ottimizzazioni e accelerazioni hardware utilizzate per il modello principale.

- **Svantaggi**

- **Difficoltà nello sviluppo del dataset**

La produzione di un dataset per questo tipo di rete neurale è un processo più complesso relativamente a quanto visto per il modello completo. In questo caso bisognerebbe estrarre le sotto-matrici di posizione da vari video e assegnare loro un'etichetta sulla base dell'andamento della traiettoria visibile nelle immagini. Questo processo può essere dispendioso e richiedere tempi dilatati. Analogamente è possibile pensare ad un sistema differente di addestramento come l'*addestramento con rinforzo*, ma questo richiederebbe tempi di allenamento molto lunghi.

---

<sup>3</sup>Un *Tensore Sfilacciato* o *Ragged Tensor* è un particolare tipo di tensore che presenta dimensioni variabili lungo uno degli assi

– **Sensibilità agli iperparametri**

Differentemente da quanto visto per l'algoritmo iterativo, gli iperparametri  $B$ ,  $H$ ,  $c$  e il numero di frame di analisi sono valori che modificano la dimensione del tensore di input della rete. Questo implica che variazioni di questi parametri comportino automaticamente la necessità di utilizzare reti differenti. Questi valori devono quindi essere valutati prima dello sviluppo della rete e devono rimanere costanti durante tutta l'esecuzione.

### 7.0.2 Conclusioni

Alla luce di quanto detto e visto fin'ora risulta evidente come le possibilità di sviluppo e ampliamento di questo progetto siano molte. Dalla creazione di un dataset più approfondito fino allo sviluppo di un sistema di riconoscimento della traiettoria. Il lavoro svolto rappresenta un ottimo punto di partenza e una base solida per un progetto che si trova attualmente agli inizi, ma che può svilupparsi ed inglobare altri sistemi, ottimizzazioni e idee prima di poter effettivamente svolgere il compito per cui è stato designato.

Ad ogni modo l'esperienza di sviluppo nell'ambito dell'intelligenza artificiale svolta con questo progetto è stata altamente formativa, sia nella parte di analisi e di ricerca legata alla comprensione dei sistemi di funzionamento dei principali algoritmi dell'intelligenza artificiale, sia nella fase di sviluppo vero e proprio del modello tramite l'utilizzo di librerie e piattaforme specifiche.

# Bibliografia

- [1] Turing A. M. “COMPUTING MACHINERY AND INTELLIGENCE”. In: *Mind* LIX (1950), pp. 433–460.
- [2] Banfi Vittorio. *What is The Turing Test? A Guide to Running One With Slack*. URL: <https://botsociety.io/blog/2018/03/the-turing-test/>.
- [3] *Perceptrons: an introduction to computational geometry*. 1969.
- [4] *Batch gradient descent vs Stochastic gradient descent*. URL: [https://www.bogotobogo.com/python/scikit-learn/scikit-learn\\_batch-gradient-descent-versus-stochastic-gradient-descent.php](https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php).
- [5] *Stochastic Gradient Descent – Mini-batch and more*. URL: <https://adventuresin%20machinelearning.com/stochastic-gradient-descent/>.
- [6] Yamashita Rikiya et al. “Convolutional neural networks: an overview and application in radiology”. In: *Insights into Imaging* IX (2018), pp. 611–629.
- [7] Wikipedia. *Computer Vision*. URL: [https://it.wikipedia.org/wiki/Visione\\_artificiale](https://it.wikipedia.org/wiki/Visione_artificiale).
- [8] Chandra Anil e Matcha Naidu. *A 2021 guide to Semantic Segmentation*. 2021. URL: <https://nanonets.com/blog/semantic-image-segmentation-2020/>.
- [9] Jason Brownlee. “A Gentle Introduction to Object Recognition With Deep Learning”. In: *Deep Learning for Computer Vision* (2019).
- [10] Google. *Introduzione ai tensori*. URL: <https://www.tensorflow.org/guide/tensor>.
- [11] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: <https://cs231n.github.io/convolutional-networks/>.
- [12] David Fouhey. “Convolutional Neural Nets EECS 442”. In: (2019). URL: <https://slidetodoc.com/convolutional-neural-nets-eeecs-442-prof-david-fouhey/>.
- [13] Sumit Saha. “A Comprehensive Guide to Convolutional Neural Networks”. In: (2018). URL: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [14] *The difference between convolution kernel and filter in visual image processing*. URL: <https://www.programmingsought.com/article/61697696850/>.

- [15] Mayank Mishra. *Convolutional Neural Networks Explained*. URL: <https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>.
- [16] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>.
- [17] *selective\_search*, *GitHub*. URL: [https://github.com/ChenjieXu/selective\\_search](https://github.com/ChenjieXu/selective_search).
- [18] Long Jonathan, Shelhamer Evan e Darrell Trevor. “Fully Convolutional Networks for Semantic Segmentation”. In: (). URL: [https://openaccess.thecvf.com/content\\_cvpr\\_2015/papers/Long\\_Fully\\_Convolutional\\_Networks\\_2015\\_CVPR\\_paper.pdf](https://openaccess.thecvf.com/content_cvpr_2015/papers/Long_Fully_Convolutional_Networks_2015_CVPR_paper.pdf).
- [19] “Network in Network”. In: (). URL: <https://arxiv.org/pdf/1312.4400.pdf>.
- [20] Raj Sakthi. *Comprehensive look at 1X1 Convolution in Deep Learning*. 2020. URL: <https://medium.com/analytics-vidhya/talented-mr-1x1-comprehensive-look-at-1x1-convolution-in-deep-learning-f6b355825578>.
- [21] Divyanshu Mishra. *Transposed Convolution Demystified*. 2020. URL: <https://towardsdatascience.com/transposed-convolution-demystified-84ca81b4baba>.
- [22] Wikipedia. *Bilinear interpolation*. URL: [https://en.wikipedia.org/wiki/Bilinear\\_interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation).
- [23] Wikipedia. *Bicubic Interpolation*. URL: [https://en.wikipedia.org/wiki/Bicubic\\_interpolation](https://en.wikipedia.org/wiki/Bicubic_interpolation).
- [24] Harshall Lamba. *Understanding Semantic Segmentation with UNET*. 2019. URL: <https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47>.
- [29] Johann Huber. *Normalizzazione batch in 3 livelli di comprensione*. URL: <https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338>.
- [30] Srivastava Nitish et al. “Dropout: A Simple Way to Prevent Neural Networks from Neural Networks (Dropout)”. In: *Journal of Machine Learning Research* (2015).
- [32] Mazzia Vittorio. *Intro to TensorFlow Lite*. URL: <https://vittoriomazzia.com/tensorflow-lite>.
- [33] *Tensorflow Lite*. URL: <https://www.tensorflow.org/lite/convert>.

- 
- [34] *Quantizzazione a 8 bit e TensorFlow Lite: accelerazione dell'inferenza mobile con bassa precisione.* URL: <https://ichi.pro/it/quantizzazione-a-8-bit-e-tensorflow-lite-%20ccelerazione-dell-inferenza-mobile-con-bassa-precisione-185280053836733>.
- [37] wikipedia. URL: [https://en.wikipedia.org/wiki/Application-specific\\_integrated\\_circuit](https://en.wikipedia.org/wiki/Application-specific_integrated_circuit).
- [38] wikipedia. URL: [https://en.wikipedia.org/wiki/Tensor\\_Processing\\_Unit](https://en.wikipedia.org/wiki/Tensor_Processing_Unit).



# Sitografia

- [25] *Google Colab*. URL: [https://colab.research.google.com/notebooks/intro.ipynb?utm\\_source=scs-index](https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index).
- [26] *Tensorflow*. URL: <https://www.tensorflow.org/>.
- [27] *Numpy*. URL: <https://numpy.org/>.
- [28] *OpenCV*. URL: <https://opencv.org/>.
- [31] *LabelBox*. URL: <https://labelbox.com/>.
- [35] *NVIDIA Jetson Nano*. URL: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>.
- [36] *Google Coral DevBoard Mini*. URL: <https://coral.ai/products/dev-board-mini/>.





# Elenco delle figure

1.1	Esempio di rilevamento corsia. Sulla sinistra l'immagine originale, sulla destra in giallo viene evidenziata la corsia centrale rilevata dalla rete (Risultati del progetto).	2
2.1	Rappresentazione del test di Turing [2]	4
2.2	Albero di gioco a tre turni	6
2.3	Schema a blocchi dell'apprendimento rinforzato	9
2.4	Schema a blocchi dell'apprendimento rinforzato con KB	9
2.5	Relazione fra la classi di intelligenza artificiale	10
3.1	Schema elementare di una rete Neurale	11
3.2	Modello di McCulloch-Pitts	12
3.3	Andamento della funzione di heaviside (Da MATLAB)	14
3.4	Andamento della Sigmoide (da MATLAB)	14
3.5	Andamento della funzione ReLU (da MATLAB)	15
3.6	Disposizione ingressi sul piano cartesiano (da MATLAB)	19
3.7	Retta caratteristica della funzione AND (da MATLAB)	19
3.8	Retta caratteristica della funzione OR (da MATLAB)	20
3.9	Classificazione ingrassi per la funzione XOR (da MATLAB)	20
3.10	Struttura degli strati del MLP	21
3.11	Diagramma del primo strato del Multi-Layer Perceptron (da MATLAB)	25
3.12	Andamento della funzione di costo nel Batch Gradient Descend [4]	28
3.13	Andamento della funzione di costo nello Sthochastic gradient descend [5]	28
3.14	Zone di underfitting e overfitting relativamente alla funzione di costo [6]	29
4.1	Esempio di classificazione [8]	32
4.2	Esempio di localizzazione[8]	32
4.3	Esempio di bounding box in processi di object detection e single-object detection [9]	33
4.4	Esempio di <i>Image segmentation</i> [8]	34
4.5	Esempio di segmentazione semantica	34
4.6	Esempio di segmentazione per istanza	35

4.7	rappresentazione tensori elementari [10] . . . . .	35
4.8	Esempi di differenti forme dei tensori [10] . . . . .	36
4.9	Canali RGB di un'immagine . . . . .	36
4.10	Differenza nella "forma" delle reti FCN e CNN [11] . . . . .	37
4.11	Differenza nelle connessioni dei perceptron in reti FCN e ConvNet [12] . . . . .	38
4.12	Distribuzione degli strati nelle ConvNet . . . . .	38
4.13	rappresentazione della sliding window su un tensore [13] . . . . .	39
4.14	Visualizzazione di differenti tipi di stride . . . . .	39
4.15	Visualizzazione del padding di immagini RGB . . . . .	40
4.16	Illustrazione del rapporto fra kernel e filtri [14] . . . . .	41
4.17	Downsampling di un tensore tramite Pooling [6] . . . . .	42
4.18	Visualizzazione del Max Pooling . . . . .	43
4.19	Operazioni di convoluzione [15] . . . . .	44
4.20	Visualizzazione del processo di convoluzione in una ConvNet. Ogni sezione dell'immagine d'ingresso viene moltiplicata per il kernel e il risultato fornisce una cella del tensore di uscita. Il processo viene ripetuto interamente su tutta l'immagine. La presenza del padding esterno consente di mantenere la dimensioni originali, in questi casi si parla di <i>Valid padding</i> . . . . .	45
4.21	Ogni kernel del filtro dello strato convolutivo produce in output una <i>feature map</i> , complessivamente l'output dello strato sarà un tensore costituito da uno stack di feature map . . . . .	45
4.22	struttura completa di una ConvNet . . . . .	46
4.23	Schema funzionale di una rete R-CNN [16] . . . . .	47
4.24	Visualizzazione delle RoI per varie modalità di selective search . . . . .	48
4.25	Architettura della fast R-CNN [16] . . . . .	49
4.26	Analisi delle prestazioni di allenamento e previsione di R-CNN e Fast R-CNN [16] . . . . .	50
4.27	Architettura Fast R-CNN[16] . . . . .	50
4.28	Struttura di una CNN e della rete FCN (troncata) [18] . . . . .	52
4.29	Esempio di 1x1 convolution [20] . . . . .	52
4.30	Esempio di <i>HeatMap</i> . . . . .	53
4.31	Visualizzazione dell'algoritmo <i>Nearest Neighbors</i> [21] . . . . .	54
4.32	Paragone fra gli algoritmi <i>Nearest Neighbor</i> , interpolazione bi-lineare e interpolazione bi-cubica. I punti neri e rossi/gialli/verdi/blu corrispondono rispettivamente al punto interpolato e ai campioni vicini. Le loro altezze dal suolo corrispondono ai loro valori[23]. . . . .	56
4.33	Visualizzazione delle operazioni di <i>transposed convolution</i> [21] . . . . .	57
4.34	Up-sampling della medesima immagine tramite interpolazione bi-cubica e transposed convolution [21] . . . . .	57
4.35	Esempi di artefatti a scacchiera . . . . .	58
5.1	Esempio di rilevamento della corsia . . . . .	59

5.2	Struttura della rete U-Net [24] . . . . .	61
5.3	Visualizzazione della batch normalization [29] . . . . .	64
5.4	modifica della distribuzione normale tramite modifica dei parametri $\gamma, \beta$ [29] . . . . .	65
5.5	Durante l'allenamento alcuni neuroni vengono disconnessi e rimane attiva solo una sottoparte della rete [30] . . . . .	65
5.6	Esempi di etichette del dataset . . . . .	67
5.7	Andamento delle curve di perdita e accuratezza relativamente alle epoche di addestramento. In arancione vengono riportati i valori di perdita/accuratezza sul <i>Test Set</i> mentre in blu quelli sul <i>Validation Set</i> . Da notare come nelle epoche fra 11 e 21 alcune variazioni dei pesi che hanno prodotto variazioni minime nella perdita e accuratezza sul <i>Test Set</i> abbiano portato ad una grande perdita nel <i>Validation Set</i> . . . . .	70
5.8	Andamento dei valori di perdita e accuratezza nella fase di addestramento della rete ridotta. In blu viene riportata la perdita/accuratezza sul <i>Validation set</i> e in arancione sul <i>Test Set</i> . . . . .	73
5.9	Rappresentazione schematica del sistema EdgeAI. [32] . . . . .	73
5.10	Panoramica del <i>ToolKite</i> di <i>Tensorflow Lite</i> . [33] . . . . .	74
5.11	Esempio di quantizzazione. La linea arancione rappresenta l'intervallo completo dei valori, mentre la linea blu i valori quantizzati. [34] . . . . .	75
7.1	Sotto-matrice estratta dall'immagine binaria. Questa sotto-matrice può essere vista analogamente ad una <i>Region Of Interest (ROI)</i> la cui dimensione e posizione sono note a priori. . . . .	96
7.2	Evidenziate in blu le sotto matrici di posizione . . . . .	97
7.3	Analisi di tre frame consecutivi in una situazione di traiettoria uscente dalla corsia. Com'è possibile vedere la posizione dei valori 1 e 0 all'interno delle due sotto-matrici di posizione può essere sfruttata per determinare la traiettoria dell'atleta . . . . .	97
7.4	Analisi di tre frame consecutivi in una situazione di traiettoria no uscente. Anche in questo caso le posizioni dei valori di 0 e 1 all'interno delle sotto-matrici, differenti dalle precedenti, permettono di comprendere la traiettoria dell'atleta. . . . .	98



# Elenco delle tabelle

4.1	Tempi di esecuzione e numero di RoI per differenti modalità di selective search . . . . .	48
5.1	Riassunto implementazione U-net . . . . .	66
5.2	Riassunto implementazione U-net ridotta . . . . .	72
5.3	Tabella riassuntiva dei vari modelli . . . . .	76
6.1	Tabella dei risultati del Modello Completo . . . . .	79
6.1	Tabella dei risultati del Modello Completo . . . . .	80
6.2	Tempi di esecuzione del modello completo su CPU e GPU di <i>Google Colaboratory</i> . . . . .	81
6.3	Tempi di esecuzione del modello completo su CPU e GPU della Jetson Nano . . . . .	81
6.4	Tabella dei risultati del Modello Ridotto . . . . .	82
6.4	Tabella dei risultati del Modello Ridotto . . . . .	83
6.4	Tabella dei risultati del Modello Ridotto . . . . .	84
6.5	Tempi di esecuzione del modello ridotto su GPU e CPU Google . . . . .	85
6.6	Tempi di esecuzione del modello ridotto su Jetson Nano . . . . .	85
6.7	Tabella dei risultati del Modello Ridotto Lite . . . . .	86
6.7	Tabella dei risultati del Modello Ridotto Lite . . . . .	87
6.7	Tabella dei risultati del Modello Ridotto Lite . . . . .	88
6.8	Tempi di esecuzione del modello ridotto lite su CPU Google . . . . .	88
6.9	Tempi di esecuzione del modello ridotto lite su CPU ARM Jetson . . . . .	88
6.10	Tabella dei risultati del Modello Ridotto Lite Quantizzato . . . . .	89
6.10	Tabella dei risultati del Modello Ridotto Lite Quantizzato . . . . .	90
6.10	Tabella dei risultati del Modello Ridotto Lite Quantizzato . . . . .	91
6.11	Tempi di esecuzione del modello ridotto quantizzato su CPU Google . . . . .	92
6.12	Tempi di esecuzione del modello ridotto quantizzato su CPU ARM jetson . . . . .	92
6.13	Tempi di esecuzione del modello ridotto quantizzato su TPU e CPU Dev Board Mini . . . . .	93