UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Progettazione e Sviluppo di un algoritmo basato su Deep Learning per il conteggio di frutti da immagini RGB

Design and Development of an algorithm based on Deep Learning to count fruits in RGB images

Relatore:
Dott. Mancini Adriano

Laureando:
Pierigè Giacomo

Anno Accademico 2019-2020

Indice

Introduzione

L'esponenziale evoluzione dell'elettronica e dell'informatica nell'ultimo decennio ha portato alla crescita del numero di dispositivi connessi in rete. Internet si è esteso quindi anche al mondo degli "oggetti", come elettrodomestici, automobili, orologi, che ora sono in grado di comunicare tra di loro in modo autonomo. Nelle città sono presenti lampioni e semafori intelligenti in grado di adattarsi alle condizioni del traffico. Questi dispositivi acquisiscono quotidianamente una grossa mole di dati memorizzandole e analizzandole in remoto attraverso l'utilizzo di servizi *cloud*.

La nascita dell'internet delle "cose" ha influenzato diversi settori economici tra cui l'industria e l'agricoltura. L'impiego delle moderne tecnologie dell'informazione in agricoltura ha portato ad una vera e propria rivoluzione, tanto che è stato coniato il termine *Agricoltura 4.0* per descrivere il radicale cambiamento. Le caratteristiche dei terreni agricoli sono in continuo mutamento a causa delle variazione climatiche, alla presenza di piante infestanti e malattie. Terreni vicini possono inoltre presentare caratteristiche diverse a seconda degli attributi del suolo e dei tipi di coltivazione. La presenza di questa variabilità spaziale e temporale all'interno dei campi rende necessaria una gestione diversificata di sostanze come fertilizzanti, pesticidi, acqua e semi.

La gestione delle risorse agricole rappresenta un complesso problema decisionale i cui obiettivi sono la massimizzazione dei raccolti, la minimizzazione dei costi e la salute delle colture. Attraverso l'acquisizione e l'analisi di dati è possibile individuare la giusta quantità di risorse da destinare ad ogni area del terreno minimizzando le spese e limitando l'impatto ambientale. Infatti, l'utilizzo controllato di sostanze chimiche come pesticidi ed erbicidi permette di ridurre l'inquinamento dei terreni e delle acque sotterranee. Le informazioni ottenute dalle attività di monitoraggio permettono a macchinari agricoli come seminatrici, irroratrici e spandiconcime di eseguire un'applicazione a tasso variabile razionando le sostanze in base alle necessità del terreno. Esse inoltre possono essere impiegate per automatizzare molte operazioni come la raccolta dei frutti, il controllo delle pianti infestanti, l'irrigazione e la semina dei campi... [1]

Le operazioni più costose e dispendiose all'interno delle aziende agricole sono sicuramente la raccolta dei frutti e la sua pianificazione. esse solitamente costituiscono circa la metà della spesa totale. Gli alberi da frutto devono essere costantemente monitorati osservando il numero e il grado di maturazione dei frutti, in modo da determinare il momento più opportuno per la raccolta. L'ispezione manuale risulta molto dispendiosa ed è irrealizzabile su vasti campi. Per questo motivo sono stati parecchi i tentativi di automatizzare la fase di

raccolta.

In passato sono state impiegate macchine agricole in grado di raccogliere i frutti in modo autonomo attraverso lo scuotimento degli alberi. Queste macchine lavorano alla "cieca" facendo cadere i frutti con la forza, poiché non non conoscono la loro posizione. Le tecniche di *agricoltura 4.0* prevedono l'utilizzo di macchinari intelligenti che grazie ai dati acquisiti attraverso sofisticati sistemi di sensori sanno perfettamente dove e come agire. I macchinari per la raccolta di ultima generazione impiegano telecamere ad alta risoluzione che permettono loro di "vedere". In questo modo essi sono in grado di prelevare i frutti ad uno ad uno senza danneggiarli ne ammaccarli. [2]

Con questa tesi si vuole sperimentare l'efficacia delle tecniche di *deep learning* nell'analisi delle immagini acquisite dai sistemi per la raccolta automatica dei frutti. In particolare, vengono descritte in modo dettagliato la fase di progettazione e la fase di sviluppo di un algoritmo per il conteggio di olive. Gli algoritmi di *deep learning* si basano su modelli matematici chiamati reti neurali, che schematizzano in maniera molto semplificata il comportamento del cervello umano. Essi tentano di riprodurre il funzionamento della vista umana per poter rendere le macchine capaci di vedere. Il cervello umano riesce a riconoscere gli oggetti con estrema facilità, per questo motivo è il miglior modello da seguire.

Gli algoritmi descritti nel corso della tesi sono implementati utilizzando *Python*, un linguaggio di programmazione interpretato e orientato agli oggetti caratterizzato da forte astrazione. Viene considerato come uno dei linguaggi di più alto livello, poiché tutte le variabili e i metodi sono implementati come puntatori ad oggetti. Questa caratteristica lo rende uno dei linguaggi più utilizzati per lo sviluppo di modelli basati sul *deep learning*.

- Nel primo capitolo vengono presentate le principali tecniche per il monitoraggio degli alberi da frutto, in questo caso gli olivi, analizzando la fasi di acquisizione ed analisi dei dati.

- Nel secondo capitolo viene analizzato il problema del rilevamento di oggetti, che è alla base dell'operazione di conteggio. In particolare, viene descritta la fase di progettazione del modello per il riconoscimento e e la localizzazione delle olive attraverso tecniche di apprendimento automatico.

- Nel terzo capitolo vengono presentate in maniera approfondita le reti neurali, il modello matematico alla base degli algoritmi di *deep learning*.

- Nel quarto capito vengono presentati gli algoritmi di *deep learning* all'avanguardia nell'ambito del rilevamento di oggetti. In particolare verranno descritte due diverse tipologie di algoritmi, le reti basate su regioni e le reti a passata singola. Le tecniche descritte vengono poi utilizzate per implementare modelli per il conteggio delle olive.

- Nel quinto ed ultimo capitolo vengono valutate le prestazioni dei modelli sviluppati, analizzando pregi e difetti.

Capitolo 1

Pianificazione della della raccolta

Le tecniche di *agricoltura 4.0* sono basate sull'acquisizione di grosse quantità di dati. Per poter prendere decisioni in merito alla raccolta delle olive i dati grezzi acquisiti dai sensori devono essere analizzati in modo tale da estrarre conoscenza. L'elaborazione dei dati spesso risulta estremamente onerosa dal punto di vista computazionale e per questo motivo viene svolta utilizzando servizi di *cloud computing*. L'*hardware* alla base del sistema di monitoraggio si occupa quindi esclusivamente di eseguire le operazioni di base del sistema operativo e permettere la connessione in rete.

Durante il processo di maturazione le olive passano gradualmente da un colore verde chiaro al nero, la polpa inizialmente dura si ammorbidisce e la quantità di olio contenuta aumenta. Il momento migliore in cui effettuare la raccolta è circa metà del processo di maturazione poiché si ottiene l'olio di maggiore qualità. Il completamento del processo di maturazione garantisce una resa migliore a discapito però della qualità.

Si pone quindi il problema di raccogliere le olive al momento giusto del processo di maturazione. Risulta inoltre necessario determinare la quantità di manodopera necessaria per l'attività di raccolta e pianificare le vendite e lo stoccaggio. A tale scopo sono necessarie informazioni sul numero di olive presenti sugli alberi e sul loro grado di maturazione.

1.1 Pianificazione della raccolta

1.1.1 Monitoraggio

Solitamente le attività di monitoraggio vengono svolte installando sensori sui mezzi agricoli o su appositi furgoni di monitoraggio. Spesso però gli olivi sono situati in terreni scoscesi e non strutturati che difficilmente possono essere raggiunti dagli ingombranti mezzi agricoli come trattori e mietitrebbie. In questo caso l'attività di monitoraggio deve essere svolta direttamente dagli agricoltori, installando i sensori su un apposita imbracatura.

Negli ultimi anni con il progresso della tecnologia si è diffuso l'utilizzo di veicoli a guida autonoma nell'agricoltura. In particolare si distinguono due categorie:

- veicoli terrestri senza pilota (*Unmanned Ground Vehicle* - UGV);
- aeromobili a pilotaggio remoto (*Unmanned Aerial Vehicle* - UAV), comunemente detti droni.

Tra le due tipologie i droni sono quelli che presentano i maggiori vantaggi. Essi, infatti, consentono di raccogliere immagini aeree da diverse prospettive, altezze e angolazioni, riuscendo a raggiungere qualsiasi zona con estrema facilità, agilità e velocità. L'unica limitazione nell'utilizzo dei droni è rappresentata dal carico che sono in grado di sopportare.

L'utilizzo dei veicoli a guida autonoma ha riscontrato parecchio successo, poiché consente di automatizzare il processo di monitoraggio evitando agli agricoltori di scendere sul campo, rendendo la procedura più efficiente e più economica. Inoltre il loro impiego permette di evitare spostamenti coi mezzi agricoli per le attività di monitoraggio, riducendo i consumi di carburate e le emissioni di anidrite carbonica. [1, 3]

Molte sono le aziende che si sono interessate allo sviluppo di veicoli a guida autonoma per il monitoraggio agricolo. Uno dei progetti di maggiore successo degli ultimi anni è il *robot VINBOT* [4], sovvenzionato dall'Unione Europea, per il monitoraggio dei vigneti. *VINBOT* è un veicolo elettrico a quattro ruote che impiega telecamere a colori per conteggiare i grappoli d'uva e stimare il loro grado di maturazione.

1.1.2 Robot per la raccolta automatica

Le informazioni acquisite dai sistemi di monitoraggio possono essere impiegate, oltre che per stimare la resa dei raccolti, anche per automatizzare la fase della raccolta vera e propria. A tale scopo vengono utilizzati veicoli a guida autonoma terrestri dotati di un braccio meccanico articolato per raccogliere le olive e un contenitore per depositarle. Questi *robot* sono in grado di individuare le olive attraverso una o più telecamere e di afferrarle attraverso un dispositivo simile ad una mano.

Prima di eseguire la raccolta viene svolta una fase di ricognizione con lo scopo di mappare la posizione delle piante all'interno del terreno, in modo tale che il *robot* possa muoversi in maniera autonoma. Per ogni pianta vengono localizzate e contate tutte le olive contenute. Per ogni oliva viene valutata la posizione all'interno dell'albero, le dimensioni e il grado di maturazione.

Una volta terminata la fase di ricognizione è necessario determinare l'ordine di raccolta. Le prime olive ad essere raccolte saranno quelle più esterne, mentre le ultime saranno quelle all'interno della chioma. In questo modo si può scegliere l'ordinamento ottimale, cioè quello che consente di compiere il minimo numero di spostamenti del braccio meccanico senza danneggiare i frutti. Risulta inoltre necessario determinare, se esiste, la posizione ottimale del *robot* all'interno del terreno, in modo tale che riesca a raggiungere tutte le olive evitando di riposizionarsi più volte.

Utilizzando i dati raccolti durante la fase di ricognizione il *robot* è in grado di classificare le olive raccogliendo solamente quelle con il giusto grado di maturazione e le giuste dimensioni, lasciando le altre a maturare sull'albero.

Nei *robot* che presentano più bracci meccanici sono necessari algoritmi per la gestione della concorrenza e il bilanciamento del carico. Bisogna, infatti, evitare che i vari bracci interferiscano tra di loro e che uno sia inattivo mentre gli altri sono sovraccarichi.

Solitamente l'attività di monitoraggio viene svolta prima della raccolta, attraverso i metodi descritti nella sezione 1.1.1, pianificando la strategia di raccolta in anticipo. Per ottenere informazioni più precise e recenti l'acquisizione e l'analisi dei dati potrebbero essere svolte in tempo reale durante la raccolta. In questo caso è però necessario che gli algoritmi per l'elaborazione dei dati abbiano un tempo di risposta estremamente piccolo. [2]

L'impiego di macchine per la raccolta automatica è piuttosto recente ed ancora in fase di sviluppo. Tra i progetti più interessanti c'è il *robot Ladybird* [5] sviluppato in Australia da un professore dell'Università di Sydney. Il *robot* è alimentato ad energia solare ed è in grado di svolgere in maniera autonoma diverse funzioni di monitoraggio. Gli sviluppatori hanno mostrato l'intenzione di modificare il *robot* aggiungendo un braccio meccanico in grado di estirpare erbacce ed automatizzare la raccolta.

Figura 1.1: *VINBOT* all'opera all'interno di un vigneto. [4]

Figura 1.2: *Robot* di monitoraggio *Ladybird*. [5]

1.2 Acquisizione e analisi dei dati

1.2.1 Telerilevamento

Le informazioni ottenute durante il monitoraggio vengono utilizzate per creare mappe dei terreni, che possono essere utilizzate per studiare la variabilità dei campi o impiegate dai veicoli a guida autonoma per spostarsi all'interno dei terreni. La creazione delle mappe richiede di localizzare la posizione delle piante e contarle. Questa operazione viene svolta impiegando tecniche di telerilevamento, che consentono di ricavare informazioni su oggetti posti a distanza da un sensore misurando le radiazioni elettromagnetiche riflesse, trasmesse o emesse.

Tra i dispositivi maggiormente impiegati vi sono i sensori *Lidar*, che sono in grado di individuare la posizione delle piante all'interno dei terreni emettendo un impulso laser. Quando l'impulso incontra un oggetto viene riflesso indietro e ricevuto dal sensore. La distanza dell'oggetto, in questo caso la pianta, viene calcolata misurando il tempo trascorso tra l'emissione dell'impulso e la ricezione del segnale riflesso. La scansione può essere svolta sia per via aerea tramite droni sia via terra.

Un'altra tecnica molto utilizzata è l'acquisizione di immagini con telecamere multispettrali. Queste particolari telecamere permettono di acquisire oltre alle radiazioni appartenenti alle bande dello spettro visibile anche quelle ultraviolette e quelle appartenenti al vicino infrarosso. L'analisi delle immagini multispettrali consente di individuare all'interno dei terreni le aree contenenti la vegetazione fornendo una misura del vigore delle piante.

Questa tecnica è basata sul fatto che la clorofilla contenuta nelle foglie degli alberi assorbe solo una porzione della radiazione solare, detta radiazione fotosinteticamente attiva, che viene impiegata per eseguire la fotosintesi. Il resto dello spettro elettromagnetico, appartenente principalmente alla banda del vicino infrarosso, viene riflesso e trasmesso. Analizzando la radiazione riflessa ed emessa si possono quindi individuare le aree che contengono la vegetazione. A tale scopo viene definito il *Normalized Difference Vegetation Index* (NDVI):

$$NDVI = \frac{NIR - VIS}{NIR + VIS}, \qquad NDVI \in [-1, 1] \tag{1.1}$$

dove NIR sta ad indicare il valore della riflettanza spettrale acquisita nella banda del vicino infrarosso, mentre VIS sta ad indicare il valore della riflettanza spettrale acquisite nella banda della radiazione visibile visibile (rosso). La riflettanza spettrale è definita come il rapporto tra la radiazione riflessa da un oggetto e quella incidente, quindi assume un valore nell'intervallo $[0, 1]$. La misurazione del vigore della piante permette di prendere decisioni in merito alla fertilizzazione della pianta e la gestione della sua irrorazione. [6]

I dati acquisiti con le tecniche di telerilevamento vengono combinati con le informazioni sulla posizione geografica del veicolo di monitoraggio attraverso l'impiego di sensori di geolocalizzazione *GPS*. In questo modo i veicoli sono in grado di spostarsi in modo autonomo sapendo dove agire.

1.2.2 Problema del conteggio

Una volta terminata la fase di mappatura il *robot* agricolo può procedere con l'acquisizione di immagini in modo da poter contare le olive e stimare il loro grado di maturazione. Il sistema è in movimento durante l'acquisizione delle immagini in modo da inquadrare l'albero in diverse posizioni e da diverse angolazioni, quindi l'algoritmo che implementa l'operazione di conteggio deve essere in grado di lavorare in tempo reale.

La telecamera genera un flusso di immagini i cui fotogrammi sono distanziati di un intervallo di tempo ΔT. L'algoritmo deve quindi avere un tempo di risposta inferiore a ΔT per poter effettuare il conteggio in tempo reale. L'algoritmo è strutturato in due moduli differenti:

- un *detector* (rilevatore), cioè un modello per identificare e localizzare le olive presenti nell'immagine;
- un contatore che viene incrementato per ogni oliva identificata.

Risulta molto probabile che due fotogrammi successivi possano contenere alcune olive in comune, per questo motivo l'algoritmo deve effettuare opportune sequenze di controllo in modo da evitare di contare uno stesso elemento più volte. A tale scopo è necessario che l'algoritmo conosca la velocità del veicolo e la direzione in cui si muove in modo tale che possa calcolare lo spostamento delle olive all'interno dell'inquadratura della fotocamera tra due fotogrammi successivi. Queste informazioni vengono acquisite impiegando sensori come accelerometri e giroscopi. [3]

1.2.3 Rappresentazione e interpretazione delle immagini

Le telecamere a colori sono sensori in grado di acquisire l'intensità della radiazione elettromagnetica appartenente allo spettro della luce visibile. Questo segnale è di tipo analogico e per poter essere elaborato al calcolatore deve essere digitalizzato, operando la discretizzazione delle coordinate spaziali e quantizzando le ampiezze.

L'immagine viene suddivisa in tante celle, dette *pixel* (*picture element*), applicando una griglia di dimensioni $w \times h$ e associando uno o più *byte* per ogni cella. La codifica *RGB*, una tra le più utilizzate per rappresentare le immagini a colori, prevede di utilizzare tre *byte* per ogni cella, uno per il rosso, uno per il verde e uno per il blu.

L'immagine viene quindi rappresentata attraverso un tensore tridimensionale di dimensioni $d \times h \times w$, dove d è detta profondità e rappresenta il numero di canali di colore. Questa rappresentazione prende il nome di immagine *raster*.

A differenza di come potrebbe sembrare, il riconoscimento e la localizzazione delle olive all'interno di una pianta rappresentano un problema estremamente complesso da risolvere detto *object detection* (rilevamento di oggetti).

Dato un insieme contenenti k diverse classi di oggetti, un algoritmo di rilevamento ha il compito di individuare all'interno dell'immagine di input tutti gli oggetti appartenenti ad una delle k classi, tracciando intorno ad ognuno un riquadro di delimitazione che ne indichi la posizione e la categoria di appartenenza. Nel caso del conteggio di olive si ha $k = 1$. Questi riquadri prendono il nome di *bounding box* e rappresentano la "scatola" di dimensioni più piccole in grado di contenere tutti i *pixel* che appartengono ad un oggetto.

Il rilevamento di oggetti viene studiato da una branca dell'informatica, che prende il nome di visione artificiale (*computer vision*), il cui obiettivo è sviluppare algoritmi in grado di simulare la vista umana, ricostruendo un modello approssimato dello spazio tridimensionale a partire dall'analisi di immagini bidimensionali.

A differenza degli oggetti matematici, definiti in modo rigoroso e preciso, gli oggetti appartenenti al mondo reale, come le olive, sono vaghi e sfumati. Basti pensare, che due olive pur appartenendo alla stessa classe di oggetti possono risultare estremamente diverse tra di loro, per dimensioni, colore e altre

caratteristiche. Inoltre un'oliva potrebbe essere soggetta a diversi effetti di illuminazione, potrebbe essere coperta da un ramo, da una foglia o sovrapposta ad un'altra.

Le tecniche di programmazione tradizionali basate su passi elementari sistematici e ben definiti risultano inadatte a trattare oggetti imprecisi come quelli appartenenti al mondo reale. A tale scopo si utilizzano le cosiddette tecniche di *soft computing*, definite così in contrapposizione con i metodi tradizionali detti di *hard computing*, il cui compito è quello di imparare nuovi concetti a partire dall'esperienza, emulando la logica di ragionamento dell'uomo.

1.2.4 Stima del grado di maturazione di un frutto

Una volta localizzata un'oliva il *robot* di monitoraggio ne determina il grado di maturazione, calcolando la quantità di luce riflessa. Sia T il numero di giorni rimanenti per la raccolta e sia ΔM la durata del processo di maturazione. Il grado di maturazione di un frutto G è definito come segue:

$$G = \frac{T}{\Delta M} \tag{1.2}$$

Ogni oliva individuata dall'algoritmo di rilevamento è rappresentata da un *bounding box*. Per poter stimare il grado di maturazione di ciascun oliva è necessario ritagliare la regione dell'immagine contenuta nel riquadro di delimitazione ed eliminare i *pixel* appartenenti allo sfondo. Quest'operazione viene svolta utilizzando una tecnica che prende il nome di segmentazione, che consiste nel partizionamento dell'immagine in un insieme di regioni significative, individuate da gruppi di *pixel* con caratteristiche simili come colore, intensità luminosa o trama, chiamate segmenti.

Il metodo più semplice per eseguire la segmentazione consiste nel convertire l'immagine RGB in un'immagine a livelli di grigio ed eseguire un'operazione di sogliatura, classificando i vari punti in *pixel oggetto* e *pixel sfondo* attraverso il confronto con una soglia prestabilita. Il risultato è un'immagine binaria in bianco e nero in cui gli oggetti vengono messi in risalto. [7, 8]

Ai fini della stima del grado di maturazione potrebbero essere necessarie tecniche più sofisticate rispetto alla sogliatura, poiché le olive e le foglie hanno un colore simile. A partire dall'immagine segmentata, cui è stato rimosso lo sfondo, vengono calcolati tre diversi indici:

- la differenza tra il valor medio della componente rossa e di quella verde nella regione che individua il frutto;
- il rapporto tra il valor medio della componente rossa e di quella verde nella regione che individua il frutto;
- il rapporto tra il prodotto tra deviazione standard e valor medio della componente blu e il valor medio della componente rossa nella regione che individua il frutto;

Questi indici sono funzione del grado di maturazione e aumentano con il trascorrere del tempo. Ad esempio, la componente di rosso diventa sempre più

dominante con il trascorrere dei giorni rispetto a quella di verde, di conseguenza la differenza tra i loro valori medi aumenta. Confrontando il loro valore con dei valori di soglia ottenuti sperimentalmente si può quindi risalire al grado di maturazione. [9]

Capitolo 2

Rilevamento di oggetti

Il principale problema nella progettazione di un algoritmo di *object detection* sta nel trovare un modello matematico in grado di fornire informazioni a partire da concetti vaghi e imprecisi come quelli del mondo reale. Questo problema viene risolto impiegando particolari algoritmi, che sono in grado di imparare nuovi schemi direttamente dai dati, senza che il programmatore dia loro istruzioni.

Questi metodi vengono studiati da una branca dell'intelligienza artificiale che prende il nome di apprendimento automatico (*machine learning*), il cui ambito di ricerca è basato su tre settori principali:

- apprendimento supervisionato, basato su algoritmi capaci di generare un modello a partire da insiemi di dati opportunamente etichettati;
- apprendimento non supervisionato, basato su algoritmi capaci di individuare schemi all'interno di dati non etichettati;
- apprendimento per rinforzo, basato su algoritmi capaci imparare dall'esperienza attraverso l'ottenimento di una ricompensa quando compiono azioni corrette. [10]

2.1 Scomposizione del problema

2.1.1 Algoritmi sliding window

Il problema del rilevamento di olive all'interno di un'immagine è estremamente complesso e per poter essere risolto deve essere scomposto e ricondotto a sottoproblemi più semplici. A tale scopo l'immagine di ingresso viene suddivisa in una serie di regioni e per ogni regione vengono eseguiti algoritmi per il riconoscimento e la localizzazione.

Dato un insieme contenenti k diverse classi di oggetti, viene definito *object recognition* (riconoscimento di oggetti) il problema di associare ad un'immagine di input l'etichetta corrispondente alla classe dell'oggetto in essa contenuto. Un algoritmo di *object recognition* (riconoscimento di oggetti) esegue una classificazione delle immagini di input basandosi sull'oggetto posto in primo piano.

Un algoritmo di riconoscimento ha il compito di capire se una delle n regioni dell'immagine contenga o meno un'oliva, mentre l'algoritmo di localizzazione ha il compito di stimare la posizione e le dimensioni del riquadro di delimitazione nel caso in cui la regione contenga effettivamente un'oliva.

Il metodo più semplice e intuitivo per suddividere un'immagine in regioni è quello di scansionarla utilizzando la forza bruta. Questo metodo prevede l'utilizzo di una finestra scorrevole, cioè un riquadro che ad ogni iterazione individua una diversa regione dell'immagine. La scansione termina quando è stata analizzata tutta l'immagine. Gli algoritmi che seguono questa logica vengono detti algoritmi a finestra scorrevole (*sliding window*). Le prestazioni di questi algoritmi dipendono dalla scelta della dimensione della finestra e dal passo con cui essa viene fatta scorrere. L'algoritmo di riconoscimento sarà in grado di riconoscere solamente oggetti di dimensioni simili a quelle della finestra. Il rilevamento è quindi limitato ad oggetti di particolari scale, forme e dimensioni.

Per superare questo limite vengono solitamente impiegate due strategie:

- ridimensionare gli input con diverse scale e proporzioni ottenendo una *piramide di immagini* e applicare una finestra scorrevole di dimensione fissa ad ogni immagine;
- utilizzare finestre scorrevoli di diverse dimensioni, in questo caso si parla di *piramidi di filtri*.

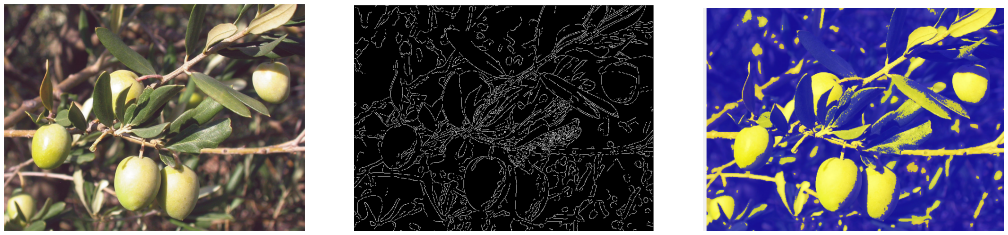
2.1.2 Estrazione di caratteristiche

La creazione di un modello di riferimento per il riconoscimento e la localizzazione di olive viene effettuata a partire dai dati attraverso tecniche di apprendimento automatico supervisionato. Sono necessarie quindi delle immagini di esempio da cui l'algoritmo di apprendimento automatico possa imparare. Delle volte, però, queste immagini presentano troppi dettagli e informazioni inutili ai fini della creazione del modello. Alcuni dati sono ridondanti o correlati tra di loro non esprimendo informazioni aggiuntive. Risulta quindi necessario estrarre dall'immagine i punti chiave che consentono di distinguere gli oggetti dallo sfondo riducendo la dimensionalità dei dati e ottenendo una rappresentazione semplificata. Quest'operazione prende il nome di estrazione di caratteristiche e viene solitamente eseguita applicando opportuni filtri alle immagini.

Ai fini del riconoscimento di un oliva non è importante mantenere informazioni sui colori, ma é sufficiente rilevare angoli, contorni, linee, gradienti. Queste informazioni prendono il nome di caratteristiche (*features*)[1]. Nell'estrazione di caratteristiche l'immagine viene completamente ristrutturata, trasformando il tensore tridimensionale di *bit* in un vettore di caratteristiche (*features vector*), che contiene solo le informazioni utili ai fini della creazione del modello. In questo modo, oltre a ridurre il volume dei dati da analizzare, è possibile generalizzare a

[1] L'estrazione di caratteristiche risulta sostanzialmente diversa rispetto alla compressione *lossy*, nonostante entrambe riducano le dimensioni dell'immagine. Lo scopo della compressione è infatti quello di ridurre lo spazio occupato in memoria dall'immagine, senza danneggiarne la qualità e facendola rimanere il più possibile fedele all'originale.

partire dai dati di input effettuando un'operazione di astrazione. Oggetti simili come due olive, presentano molti dettagli differenti all'interno dell'immagine, dovuti alla luce, all'ombra, alla profondità. La rappresentazione mediante vettore di caratteristiche permette di avere una rappresentazione simile per entrambi gli oggetti. [11]



(a) Immagine originale. (b) Rilevamento dei contorni. (c) Segmentazione.

Figura 2.1: Segmentazione e rilevamento dei contorni applicati ad un'immagine di olive.

Il processo di estrazione di caratteristiche vero e proprio consiste nel rilevamento di proprietà degli oggetti come ad esempio contorni, angoli, creste, *blob*[2]. Questi elementi sono il risultato della proiezione di oggetti tridimensionali, le olive, su un piano bidimensionale, l'immagine, e danno informazioni sulla disposizione e sull'orientamento degli oggetti nello spazio. A partire da queste poi possono essere generate caratteristiche di livello più alto come trame, forme geometriche e gli oggetti stessi.

Le variazioni di profondità e di illuminazione sono caratterizzate da brusche variazioni dell'intensità luminosa dei *pixel* all'interno dell'immagine. Per questo motivo le tecniche di riconoscimento dei contorni spesso si basano sull'analisi del gradiente dell'intensità luminosa all'interno dell'immagine con lo scopo di individuare i punti di massimo e di minimo (*tecniche search based*), o in alternativa i punti in cui la derivata seconda passa per lo zero (*tecniche zero crossing*).

I contorni vengono poi riconosciuti eseguendo la sogliatura del gradiente. Nell'ambito dell'elaborazione di immagini spesso l'operazione di derivazione viene discretizzata usando il metodo delle differenze finite centrali e implementata come convoluzione[3] tra l'immagine e un filtro derivativo. Sia $I(x)$ l'intensità luminosa di un pixel x. Applicando la tecnica delle differenze centrali si ha:

$$\frac{dI}{dx} = -\frac{1}{2}I(x-1) + \frac{1}{2}I(x+1) = I(x) * (-\frac{1}{2} \quad 0 \quad \frac{1}{2}) \tag{2.1}$$

$$\frac{d^2I}{dx^2} = I(x-1) - 2I(x) + \frac{1}{2}I(x+1) = I(x) * (1 \quad -2 \quad 1) \tag{2.2}$$

[12]

[2]Il termine blob sta ad indicare regioni all'interno di un'immagine che differiscono dallo sfondo per luminosità e colore.

[3]La convoluzione verrà descritta in maniera approfondita nel terzo capitolo in relazione alle reti neurali convoluzionali. Il simbolo $*$ sta ad indicare l'operazione di convoluzione tra due matrici.

2.1.3 Riconoscimento

Il problema dell'associazione di un'etichetta al vettore di caratteristiche è un problema di classificazione e viene risolto utilizzando tecniche di apprendimento supervisionato. In particolare ad ogni diversa classe di oggetti da riconoscere viene associato un numero naturale in modo tale da ottenere un insieme delle classi $0, 1, 2, ..., k$. In maniera formale, il riconoscimento di un oggetto consiste nell'associare ad vettore $x \in \mathbb{R}^n$, il vettore di caratteristiche, un numero $y \in \mathbb{N}$, che rappresenta la classe di appartenenza dell'oggetto considerato.

Un algoritmo in grado di eseguire una classificazione di oggetti prende il nome di classificatore e per poter essere utilizzato deve prima essere addestrato. Con addestramento si intende il processo con cui l'algoritmo riesce ad estrarre conoscenza dai dati, imparando. Durante la fase di addestramento, vengono dati in ingresso all'algoritmo diversi esempi rappresentati da coppie ingresso-uscita, che vengono elaborati per generare un modello. In un problema di *object recognition* ogni oggetto di esempio è descritto dal suo vettore di caratteristiche, l'ingresso, e dalla sua classe di appartenenza, l'uscita. L'insieme di tutti gli oggetti di esempio proposti all'algoritmo è detto dataset di addestramento (*training set* o *training dataset*).

Quando l'algoritmo ha processato un sufficiente numero di esempi genera in output un modello di classificazione, rappresentato da una funzione $y = f(x)$ tale che $f : \mathbb{R}^n \rightarrow 0, 1, 2, ..., k$. La fase di addestramento procede per tentativi, proponendo più volte in ingresso lo stesso set di dati e valutando ogni volta l'accuratezza e le prestazioni del modello prodotto. L'addestramento prosegue finché l'errore commesso dal modello nella classificazione possa essere considerato trascurabile. La presentazione dell'intero dataset di addestramento all'algoritmo prende il nome di epoca. Lo stadio in cui vengono valutate le prestazioni del modello dopo ogni epoca è detta fase di test e spesso viene eseguita su un dataset diverso da quello di addestramento, chiamato *test set* o *test dataset*. La fase di test è necessaria per determinare quando interrompere l'addestramento.

Un classificatore, durante l'addestramento, a partire da singoli casi particolari, gli esempi, tenta di ricavare una legge generale valida anche in altre circostanze. L'obiettivo è quello di elaborare una legge matematica a partire dai dati di addestramento che sia valida anche per i dati di test. La logica di apprendimento del classificatore è quindi basata sul ragionamento induttivo.

Il modello prodotto durante l'apprendimento può essere utilizzato per classificare nuovi input, cioè dati non etichettati. Il classificatore tenta di "indovinare" la giusta classe da associare ai dati di input, elaborando delle ipotesi a partire da una regola più generale, cioè il modello ricavato durante l'addestramento. Questa fase prende il nome i inferenza e segue la logica deduttiva.

L'assegnazione di etichette ad un'immagine viene effettuata estraendo le caratteristiche dall'immagine ed eseguendo l'inferenza sul vettore di caratteristiche ottenuto. L'inferenza restituisce come previsione un numero intero, che deve essere associato ad un'opportuna stringa che identifichi la classe corrispondente. Sia l'operazione di codifica che di decodifica può essere eseguita attraverso l'utilizzo di opportune strutture dati come gli *array* o i dizionari. Queste strutture

dati consento l'accesso diretto ai valori in esso contenuti attraverso una chiave, che può essere un indice numerico, nel caso degli *array*, o una stringa, nel caso dei dizionari. [10]

2.1.4 Localizzazione

Le tecniche di apprendimento supervisionato consentono di risolvere, oltre al problema di riconoscimento, anche il problema della previsione delle coordinate dei *bounding box*. Ogni riquadro viene descritto utilizzando quattro coordinate: le due coordinate spaziali del centro o dell'angolo in alto a sinistra, l'altezza e la lunghezza. La stima di ognuno di questi parametri rappresenta un problema di regressione. In maniera formale, la previsione di ognuna di queste coordinate consiste nell'associare ad un vettore $x \in \mathbb{R}^n$, il vettore di caratteristiche, un numero $y \in \mathbb{R}$, che ne rappresenta la stima. Rispetto alla classificazione, che è un problema discreto, la regressione è un problema continuo. Un algoritmo che consente di risolvere un problema di regressione prende il nome di regressore e per poter essere utilizzato deve essere prima addestrato. Le fasi di addestramento e inferenza nei regressori seguono la stessa logica di quelle nei classificatori.

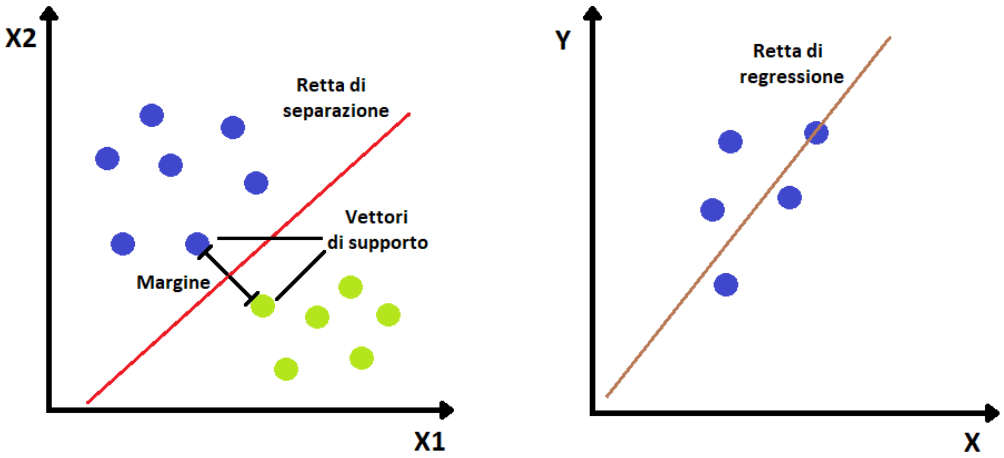
Ogni esempio contenuto nel dataset di addestramento è rappresentato da un insieme di variabili indipendenti a cui è associata una variabile dipendente. La variabile dipendente viene considerata come somma di due contributi, un termine funzione delle variabili indipendenti e una variabile aleatoria, rappresentante un errore incontrollabile e imprevedibile. Questo errore tiene conto dell'impossibilità di associare ad un oggetto un *bounding box* in modo univoco durante la creazione del dataset di addestramento. Diverse persone potrebbero etichettare uno stesso oggetto con riquadri di dimensioni o posizioni leggermente differenti.

Durante l'apprendimento un regressore tenta di ricavare dai dati di esempio una curva, espressa da una funzione $y = f(x)$ tale che $f : \mathbb{R}^n \to \mathbb{R}$, che modelli l'andamento dei dati di addestramento. Nella fase di inferenza la relazione ottenuta viene utilizzata per effettuare una stima delle coordinate dei *bounding box* su immagini diverse da quelle di addestramento. [10]

2.1.5 Tecnica basata sull'istogramma dei gradienti orientati

L'utilizzo degli algoritmi *sliding window* risulta parecchio oneroso dal punto di vista computazionale se i modelli di classificazione sono complessi e non lineari. Per questo motivo per risolvere i problemi di *object detection* vengono spesso utilizzati classificatori e regressori lineari. Nel 2005 da Navneet Dalal e Bill Trigg[13] proposero un algoritmo *sliding window* basato ull'idea che l'aspetto locale degli oggetti possa essere descritto dalla distribuzione delle direzioni dei bordi, rappresentata da istogrammi di gradienti orientati.

Per ogni posizione della finestra scorrevole l'algoritmo estrae le caratteristiche della regione calcolando gli istogrammi dei gradienti orientati (*Histogram of Oriented Gradients* - HOG). Questa tecnica prevede di dividere la regione in tante celle di dimensioni 8×8, contenenti ognuna 64 *pixel*. Ogni *pixel* presenta una diversa intensità luminosa e una diversa direzione del gradiente, espressa da un angolo $\alpha \in [0, 180]$. Ad ogni cella viene associato un istogramma, dove ogni



(a) Classificazione binaria di *features vector* bidimensionali tramite macchina a vettori di supporto.

(b) Regressione lineare con una sola variabile dipendente.

Figura 2.2: Esempi di classificazione e regressione.

colonna individua una particolare direzione dl gradiente. In totale si hanno 64 intensità e 64 gradienti, e le direzioni vengono campionate con un passo di 20, ottenendo 9 colonne. Le colonne dell'istogramma vengono riempite aggiungendo il valore dei *pixel* contenuti nella cella:

- se la direzione del gradiente associata ad un *pixel* è uguale a quella di una colonna, viene aggiunto il valore della sua intensità al *bin* corrispondente;
- se la direzione è intermedia tra due colonne viene aggiunto un contributo ad entrambe le colonne.

Figura 2.3: Esempio di estrazione di caratteristiche basata sulla tecnica degli istogrammi dei gradienti orientati. Le frecce stanno ad indicare la direzione del gradiente in una specifica regione dell'immagine.

Il vettore di caratteristiche generato viene classificato utilizzando una macchina a vettori di supporto (Support Vector Machine - SVM). L'algoritmo si basa sul presupposto che i punti associati ad una stessa classe si trovino in posizioni vicine sul piano. Questo classificatore tratta il vettore di caratteristiche, di

lunghezza n, come un punto nell'iper-spazio dimensionale superiore (di dimensione $n + 1$), con l'obiettivo di trovare un iperpiano (di dimensione $n - 1$) che riesca a partizionare lo spazio separando gli esempi appartenenti a due classi diverse.

La macchina a vettori di supporto esegue una classificazione binaria suddividendo le regioni che contengono oggetti da quelle che appartengono allo sfondo. Si consideri il caso più semplice, un vettore di caratteristiche bidimensionale ($n = 2$), che può essere rappresentato graficamente come un punto nel piano. Durante l'addestramento ad ogni punto è associata una classe, oggetto o sfondo, che può essere rappresentata colorando i punti di colore diverso. Si ottiene un punto nello spazio tridimensionale.

La macchina a vettori di supporto cerca di trovare la migliore retta di separazione tra punti di classi diverse. Per ciascuna classe, i punti più vicini alla retta di separazione prendono il nome di vettori di supporto e la loro distanza è detta margine. La suddivisione tra le classi deve essere netta, per questo motivo viene scelta la retta che presenta il margine maggiore.

Durante la fase di inferenza i nuovi punti vengono classificati valutando in quale semipiano si trovino rispetto alla retta di separazione.

Combinando insieme più classificatori binari si possono riconoscere anche oggetti di classi diverse. [11, 13]

Il più semplice algoritmo di regressione è il regressore lineare. Nel caso più semplice, cioè quando gli esempi di addestramento presentano una sola variabile indipendente, i dati di input sono individuati da punti nel piano bidimensionale. Durante la fase di apprendimento un regressore lineare cerca di trovare una retta che aderisca all'andamento dei dati, stimando il valore del coefficiente angolare e del termine noto. Nella fase di inferenza la retta trovata viene utilizzata per effettuare previsioni su nuovi dati di input.

2.2 Addestramento

2.2.1 Preparazione del dataset

Per poter addestrare un algoritmo di object detection è necessario realizzare un insieme di esempi opportunamente etichettati e classificati, in modo che l'algoritmo possa imparare a riconoscere gli oggetti in modo autonomo. Il primo passo è quello di reperire le immagini.

Due importanti fattori da tenere in considerazione sono la dimensione e la quantità delle immagini da inserire nel dataset. Per ottenere un buon modello sono necessarie immagini sufficientemente grandi di dimensioni almeno 1000×800 *pixel*. I campioni scelti devono inoltre essere rappresentativi della popolazione di partenza e diversi tra di loro, in modo che l'algoritmo di apprendimento riesca a generalizzare.

Nel caso le immagini a disposizione siano poche si possono utilizzare tecniche di *data augmentation*, che consentono di generare nuove immagini a partire da quelle già esistenti. Queste tecniche eseguono sulle immagini di partenza trasformazioni come rotazioni, inversioni, ingrandimenti, ritagli e traslazioni scelte in modo casuale così da aumentare il volume dei dati. Gli algoritmi di *machine*

learning analizzando queste immagini riescono a creare modelli invarianti alle trasformazioni e maggiormente robusti anche con pochi dati.

2.2.2 Assegnazione delle etichette

Una volta scelto un insieme di campioni questi vanno etichettati in modo da generare delle coppie input-output, dove l'input è l'immagine e l'output è una lista di *bounding box*.

Esistono diverse applicazioni e servizi online per etichettare le immagini. Tra i servizi online più utilizzati c'è Labelbox, un tool particolarmente semplice ed efficiente che consente di annotare immagini in modo veloce. Il servizio è disponibile al link:

`https://labelbox.com/`

Per utilizzare Labelbox è sufficiente registrarsi e creare un nuovo progetto. La lista delle immagini da etichettare viene gestita attraverso una coda di ingresso. Le prime immagini ad essere inserite sono anche le prime ad essere etichettate. Labelbox consente anche di saltare un'immagine che si vuole etichettare in seguito o scartare. Per annotare un immagine è sufficiente selezionare la classe relativa all'oggetto da etichettare e tracciare un riquadro intorno alla regione dell'immagine che lo contiene. Se l'annotazione non è precisa il *bounding box* può essere eliminato per poi tracciarne uno nuovo. Una volta terminato il progetto,

Figura 2.4: Interfaccia grafica fornita del *tool* Labelbox per l'annotazione di immagini.

Labelbox consente di generare un file contenente i metadati relativi al progetto. Il file può essere in formato JSON (*JavaScript Object Notation*) o CSV (*Comma Separated Values*). Entrambi i formati presentano dei vantaggi. Nella progettazione dell'algoritmo per il conteggio di olive si è scelto di utilizzare il formato JSON, poiché risulta maggiormente leggibile a occhio nudo, anche se occupa maggiore spazio in memoria.

Nel formato JSON le informazioni sono rappresentate attraverso una struttura dati chiamata oggetto costituita da una lista di attributi e delimitata da due parentesi graffe. Ogni attributo viene rappresentato come una coppia chiave-valore, dove la chiave è una stringa indicante il nome dell'attributo e il valore

è un tipo di dato di base (*int, float, string, bool*). Gli oggetti possono essere raggruppati insieme in una struttura dati chiamata vettore, individuata da due parentesi quadre. Gli stessi vettori possono essere inclusi in altri vettori in modo da formare una struttura gerarchica.

Il file JSON generato da Labelbox è strutturato come un vettore di oggetti, dove ogni oggetto rappresenta una specifica immagine etichettata. Ogni oggetto presenta diversi attributi tra cui:

- il nome del progetto e dell'immagine, identificati rispettivamente dalle stringhe "**Project Name**" e "**External ID**";
- l'indirizzo da cui ottenere l'immagine, identificato dalla stringa "**Labeled Data**";
- la data di creazione e di ultima modifica dell'immagine, identificate rispettivamente dalle stringhe "**Created At**" e "**Updated At**";
- l'utente che ha aggiunto l'immagine al dataset, identificato dalla stringa "**Created By**";
- la lista dei *bounding box* contenuti nell'immagine, identificata dalla stringa "**Label**";

La lista dei *bounding box* contenuti nell'immagine viene rappresentata come un oggetto, dove ogni attributo è un vettore contenente gli oggetti relativi alla stessa classe[4]. Ogni elemento di questi vettori è un oggetto che ha come attributi, il nome dell'etichetta associata, il colore e le coordinate del riquadro... In particolare, il JSON contiene le coordinate dell'angolo in alto a sinistra e le dimensioni del riquadro.

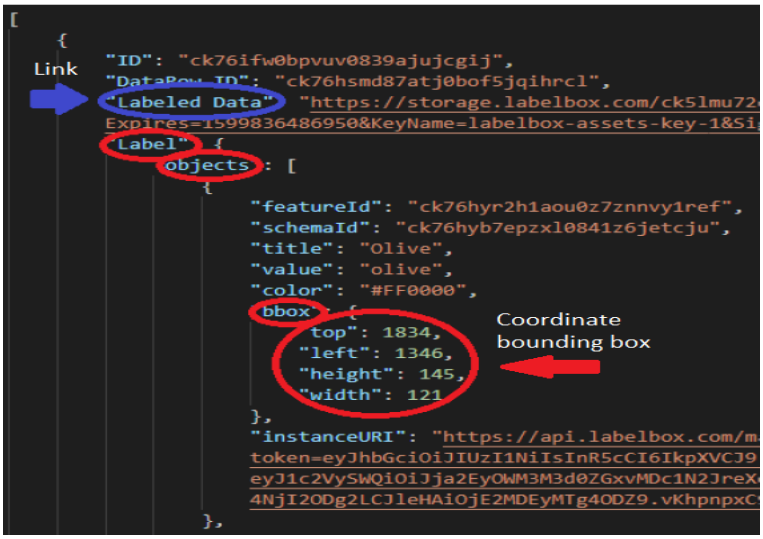


Figura 2.5: Esempio di file JSON generato da Labelbox.

[4]Nel caso del riconoscimento di olive si ha una sola classe, quindi un solo vettore

I dati contenuti nel JSON per poter essere elaborati devono prima essere estratti e rappresentati utilizzando le strutture dati tipiche del linguaggio di programmazione. Questo processo prende il nome di *parsing* e in Python può essere eseguito utilizzando una specifica libreria, la libreria **json**.

Prima di eseguire il *parsing* è necessario effettuare il download del JSON da Labelbox. Quest'operazione in Python viene eseguita utilizzando il metodo **urlretrieve()**, fornito dalla libreria **urllib.request**, cui bisogna passare come parametro l'*url* da cui scaricare la risorsa e il percorso in cui salvarla, entrambi rappresentati come stringhe. Una volta terminato il download, è sufficiente caricare il file in memoria utilizzando il metodo **open()**, cui bisogna passare come parametro l'indirizzo del file, ed eseguire il *parsing*. Il metodo **open()** restituisce un puntatore al file da passare come parametro alla funzione **load()**, fornita dalla libreria **json**, che ne esegue il *parsing*. Una volta effettuato il *parsing* è necessario chiudere il file, utilizzando la funzione **close()**.

La funzione **load()** estrapola le informazioni contenute nel JSON e le rappresenta con strutture dati tipiche di Python, come dizionari e liste. In questo caso, l'output della funzione **load()** sarà una lista di dizionari, dove ogni dizionario corrisponde ad un immagine del dataset.

```
<annotation>
        <folder>JPEGImages</folder>
        <filename>olives0.jpg</filename>
        <path>/content/faster_RCNN/dataset/JPEGImages/olives0.jpg</path>
        <source>
                <database>Olives2019</database>
        </source>
        <size>
                <width>800</width>
                <height>600</height>
                <depth>3</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>olives</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>203</xmin>
                        <ymin>368</ymin>
                        <xmax>221</xmax>
                        <ymax>397</ymax>
                </bndbox>
        </object>
</annotation>
```

Figura 2.6: Esempio di annotazione Pascal Visual Object Classes.

2.2.3 Struttura del dataset

Gli elementi del dataset di addestramento devono essere strutturati come coppie ingresso-uscita per poter essere elaborati dall'algoritmo di apprendimento. A tale scopo vengono utilizzati due diversi tipi di file: un file per rappresentare le immagini e un file per rappresentare le etichette. I file corrispondenti alla stessa coppia ingresso-uscita devono avere lo stesso nome ed essere posti in due cartelle differenti. Il dataset di addestramento sarà quindi composto da due cartelle speculari, una contenente le immagini e una contenente i relativi file di etichetta.

Il dataset di addestramento può essere generato a partire dai metadati estratti nella fase di *parsing*. Il download delle immagini viene effettuato allo stesso modo del JSON, prestando però particolare attenzione al nome da dare ai file.

La creazione dei file di etichetta risulta più complessa e dipende dalla particolare architettura scelta. Una delle più utilizzate è quella di tipo **XML**, introdotta dal dataset di immagini **Pascal Visual Object Classes (Pascal VOC)**, che rappresenta le etichette utilizzando dei *tag*. I file **XML** contengono, oltre alle informazioni relative ai *bounding box*, anche metadati dell'immagine, come il nome, il percorso, la dimensione... Nell'architettura Pascal Visual Object Classes un *bounding box* è rappresentato dalla posizione dei sui lati all'interno dell'immagine.

Dato l'i-esimo *bounding box* contenuto nell'immagine, sia x_{min} l'ascissa relativa del bordo sinistro, y_{min} l'ordinata relativa al bordo in alto, x_{max} l'ascissa relativa al bordo destro e y_{max} l'ordinata relativa al bordo in basso. Siano invece x, y, h, w le quattro coordinate contenute nel file JSON. La conversione viene effettuata con le seguenti formule:

$$\begin{aligned} x_{min} &= x \\ y_{min} &= y \\ x_{max} &= x_{min} + w = x + w \\ y_{max} &= y_{min} + h = y + h \end{aligned} \tag{2.3}$$

Il file di annotazione deve contenere anche le dimensioni dell'immagine di input, informazione che non è contenuta tra i metadati presenti nel JSON. A tale scopo è necessario caricare l'immagine in memoria attraverso la funzione **open()**, fornita dalla libreria **PIL.Image**[5], che restituisce un riferimento all'immagine aperta. Per ricavare le dimensioni è sufficiente accedere al campo **size** del riferimento all'immagine. Le dimensioni sono rappresentate come una tupla di due elementi. Una volta ottenute le dimensioni è necessario chiudere l'immagine, utilizzando la funzione **close()**, fornita dalla libreria **PIL.Image**.

Spesso l'algoritmo può richiedere che le immagini abbiano una dimensione fissa, in questo caso esse devono essere ridimensionate. La libreria **PIL.Image** permette di ridimensionare un'immagine utilizzando il metodo **resize()**, che richiede come parametro le nuove dimensioni dell'immagine sotto forma di tupla e restituisce un riferimento all'immagine.

2.2.4 Algoritmo di apprendimento

Un algoritmo di apprendimento supervisionato riesce ad imparare a risolvere problemi di classificazione o regressione procedendo per tentativi ed errori. L'algoritmo per ogni esempio del dataset di addestramento tenta di "indovinare" la risposta corretta, rappresentata dall'uscita associata all'esempio considerato, per poi verificare se la sua predizione fosse giusta.

[5]La funzione descritta è diversa da quella utilizzata per aprire il JSON, che al contrario è fornita direttamente da Python

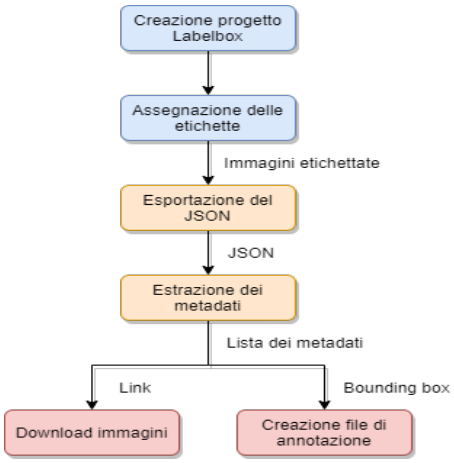


Figura 2.7: Logica seguita nella preparazione del dataset di addestramento.

Inizialmente, l'algoritmo commetterà tanti errori, poiché le risposte sono date a caso senza seguire una logica precisa. Le risposte iniziali dipendono fortemente dall'inizializzazione dei parametri del modello. L'algoritmo dopo aver osservato un certo numero di esempi modifica i parametri del modello in modo tale da ridurre il numero di errori commessi negli esempi successivi.

In maniera formale, l'obiettivo dell'algoritmo di apprendimento è quello di minimizzare un funzionale di errore, detto funzione di costo o di perdita, che esprime la discrepanza tra i valori stimati dal modello e quelli contenuti nel dataset di addestramento.

Al variare dei parametri del modello varia anche il valore delle risposte predette. L'errore di predizione può essere espresso in funzione dei parametri del modello, poiché ogni uscita predetta è una loro funzione composta. Sia n il numero di parametri contenuti nel modello. Allora la funzione di perdita è una funzione $C = f(x)$ tale che $f : \mathbb{R}^n \to \mathbb{R}$.

L'apprendimento di un modello di *object detection* viene riformulato come un problema di ottimizzazione la soluzione non è banale, poiché le funzioni di perdita solitamente non sono convesse presentando diversi minimi locali. L'ottimizzazione viene eseguita attraverso l'utilizzo di algoritmi iterativi, poiché spesso non esistono metodi analitici che consentano di trovare soluzioni esatte. Tra i metodi iterativi quello che garantisce i migliori risultati è la discesa del gradiente *batch* (*Batch Gradient Descend - BGD*), che modifica i parametri del modello analizzando la direzione di crescita del gradiente della funzione di costo[6].

Ogni combinazione dei parametri del modello rappresenta un punto nello spazio n-dimensionale a cui è associato un particolare valore dell'errore. La funzione di perdita può essere quindi rappresentata da una curva nello spazio dimensionale superiore. In un modello con due soli parametri l'errore è rappresentabile in uno spazio tridimensionale come una superficie caratterizzata da "valli", in corrispondenza dei punti di minimo, e "colline", in corrispondenza

[6]Il gradiente della funzione di perdita è un vettore le cui componenti sono le derivate parziali rispetto ai parametri del modello.

dei punti di massimo. Ad ogni iterazione, l'algoritmo di apprendimento tenta di spostarsi in direzione della valle più profonda, dove si troverà il punto di minimo globale corrispondente al più piccolo errore possibile.

Alla fine di ogni epoca, dopo aver analizzato l'intero dataset di addestramento, l'algoritmo calcola il funzionale di errore e il suo gradiente. I parametri del modello vengono modificati in modo che il punto da loro individuato si sposti nella direzione del minimo globale, cioè la direzione opposta a quella del gradiente. L'algoritmo termina quando la soluzione si stabilizza intorno ad un punto di minimo. Solitamente l'addestramento viene interrotto prematuramente in modo da evitare l'*overfitting*.

Il calcolo del gradiente alla fine di ogni epoca richiede che tutti gli esempi di addestramento siano contemporaneamente caricati in memoria, rendendo l'algoritmo lento e pesante dal punto di vista computazionale. Per questo motivo in presenza di dataset di grandi dimensioni solitamente si usa una variante del *batch gradient descend*, la discesa stocastica del gradiente (*Stochastic Gradient Descend - SGD*), che aggiorna i parametri del modello ogni volta che viene analizzato un nuovo esempio del dataset. In questo modo viene caricato in memoria un solo esempio alla volta, diminuendo lo sforzo computazionale, anche se la varianza dell'errore in alcuni casi potrebbe aumentare anziché diminuire. [10, 14]

2.2.5 Overfitting e underfitting

La durata dell'addestramento, espressa come numero di epoche, e la quantità di esempi proposti sono parametri fondamentali per ottenere un buon modello. Un addestramento troppo breve o l'utilizzo di un dataset poco numeroso potrebbe portare al fenomeno dell'*underfitting*. In queste circostanze l'algoritmo genera un modello troppo semplice, che non presenta abbastanza parametri per descrivere l'andamento dei dati.

Un modello affetto da *underfitting* restituisce pessimi risultati nella fase di inferenza, riuscendo a rilevare correttamente pochi oggetti.

L'algoritmo di addestramento potrebbe avere difficoltà a generalizzare o a costruire un modello corretto quando i dati di input, in questo caso le immagini, sono molto rumorosi. Per questo motivo le immagini in alcune situazioni vengono pre-elaborate prima di essere etichettate, riducendo gli effetti legati a luminosità e contrasto e ritagliando le zone di interesse all'interno dell'immagine. Una tecnica molto utilizzata è quella della normalizzazione. che prevede di sottrarre ad ogni *pixel* il valor medio dell'intensità luminosa sull'intera immagine, per poi dividere il risultato per la deviazione standard.

All'aumentare del numero di epoche l'algoritmo di addestramento genera un modello sempre più preciso e accurato per rappresentare il particolare andamento dei dati di addestramento. Se il numero di epoche è troppo elevato, durante la fase di test, eseguita su un diverso set di dati, il modello produrrà risultati scadenti. Questo fenomeno prende il nome di *overfitting* o *overtraining* e si verifica quando il modello ottenuto risulta troppo complesso, adattandosi eccessivamente al particolare insieme di dati usato per l'addestramento, senza riuscire a generalizzare.

L'*overfitting* si verifica quando tra due più funzioni che rappresentano l'andamento dei dati viene scelta quella più complessa. Il modello così ottenuto presenta troppi gradi di libertà modellando anche il contributo introdotto dal rumore. Tra due curve che modellano entrambe bene l'andamento dei dati, senza generare *underfitting*, bisogna quindi scegliere sempre quella più semplice. [10]

2.3 Inferenza e test

2.3.1 Intersection over Union

Il numero di epoche risulta essere un parametro critico nella creazione di un buon modello di *object detection*. La scelta solitamente ricade su un valore elevato, in modo tale da evitare il fenomeno dell'*underfitting*. All'aumentare del numero di epoche trascorse le prestazioni sul dataset di addestramento migliorano sempre di più e l'errore commesso nel rilevamento tende a zero.

Al contrario, le prestazioni sul dataset di test iniziano a degradarsi dopo un certo numero di epoche n_0. Per evitare l'*overfitting* bisogna fermare l'addestramento prima di eseguire n_0 epoche, cioè prima che l'algoritmo di apprendimento smetta di generalizzare. A tale scopo, dopo ogni epoca, durante la fase di test, vengono valutate le prestazioni del modello sul dataset di addestramento e su quello di test, in modo da determinare il momento più opportuno per interrompere l'addestramento. Le prestazioni di un modello per il rilevamento di oggetti vengono valutate analizzando la sovrapposizione tra la previsione generata dall'algoritmo e il *ground truth box*, cioè il riquadro associato all'oggetto reale.

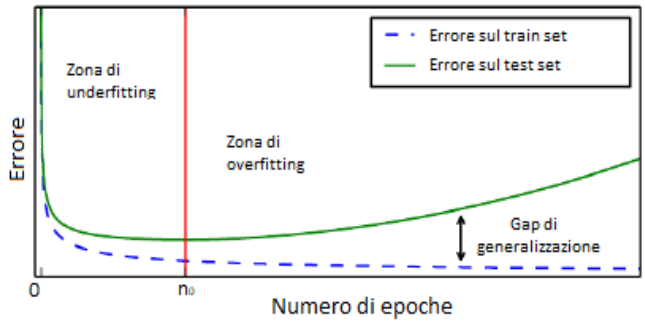


Figura 2.8: Il grafico mostra l'andamento dell'errore di rilevamento sul test set e sul train set in funzione del numero di epoche. [10]

La sovrapposizione tra due *bounding box* viene valutata utilizzando una metrica che prende il nome di *Intersection over Union* (IoU) o coefficiente di similarità di Jaccard. In particolare, questo indice fornisce una misura della similarità tra le caratteristiche di due insiemi campionari, valutando il rapporto tra la dimensione dell'intersezione e quella dell'unione dei due insiemi.

Nell'ambito dell'*object detection* i due insiemi sono rappresentati dalle aree delimitate dai *bounding box*. Sia A l'insieme dei punti che rappresentano la su-

perficie individuata dall'oggetto predetto, sia B l'insieme dei punti che rappresentano la superficie individuata dall'oggetto reale[7], l'*Intersection over Union* è definito come segue:

$$IoU(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}, \qquad IoU(A,B) \in [0,1] \tag{2.4}$$

Spesso per misurare la dissimilarità tra due insiemi A e B si utilizza la distanza di Jaccard, definita come il complementare dell'*Intersection over Union*. Entrambi gli indici forniscono una misura della distanza e della lontananza tra le caratteristiche di due insiemi di punti.

Due *bounding box* si dicono sovrapposti se il valore del loro *Intersection over Union* risulta superiore ad una soglia stabilita a priori. Quando tra due *bounding box* $IoU = 0$ si ha assenza di sovrapposizione, mentre quando $IoU = 1$ si sovrappongono perfettamente.[15]

Siano i e j i due *bounding box* di cui si vuole valutare la sovrapposizione. Si consideri un sistema di riferimento con origine nell'angolo in alto a sinistra dell'immagine. Sia $\{x, y, w, h\}$ l'insieme delle quattro variabili che descrivono un generico *bounding box*, dove $x \in \mathbb{R}$ e $y \in \mathbb{R}$ sono le coordinate del centro, $w \in \mathbb{R}$ è la lunghezza e $h \in \mathbb{R}$ l'altezza. Prima di calcolare l'intersezione tra i e j è necessario valutare se i due riquadri si sovrappongono.

Sia $d_{ij}^{(x)}$ la distanza tra le ascisse dei centri dei due *bounding box* e sia $d_{ij}^{(y)}$ la distanza tra le corrispondenti ordinate. Allora la distanza tra i due centri d_{ij} viene definita come segue:

$$d_{ij} = \begin{pmatrix} d_{ij}^{(x)} \\ d_{ij}^{(y)} \end{pmatrix} = \begin{pmatrix} |x_i - x_j| \\ |y_i - y_j| \end{pmatrix} \tag{2.5}$$

Viene definita distanza massima tra i due centri il valore limite che può assumere d_{ij} affinché i e j siano sovrapposti ed è data dalla somma delle distanze dei due centri dai bordi del riquadro. In particolare si ha:

$$d_{ij}^{max} = \begin{pmatrix} \frac{1}{2}w_i \\ \frac{1}{2}h_i \end{pmatrix} + \begin{pmatrix} \frac{1}{2}w_j \\ \frac{1}{2}h_j \end{pmatrix} = \frac{1}{2}\begin{pmatrix} w_i + w_j \\ h_i + h_j \end{pmatrix} \tag{2.6}$$

Se $d_{ij} > d_{ij}^{max}$ i due riquadri non sono sovrapposti e di conseguenza l'*intersection over union* risulta nullo. Se uno dei due riquadri è interamente contenuto nell'altro, allora l'area dell'intersezione è pari a quella del riquadro più piccolo.

L'area di intersezione tra due *bounding box* è rappresentata da un rettangolo i cui vertici sono i punti di intersezione tra i lati dei due riquadri. La lunghezza $w_{i \bigcap j}$ del rettangolo individuato dall'intersezione è pari alla distanza tra due vertici consecutivi lungo l'asse delle ascisse, mentre l'altezza $h_{i \bigcap j}$ del rettangolo individuato dall'intersezione è pari alla distanza tra due vertici consecutivi lungo l'asse delle ordinate. Il loro prodotto è pari all'area dell'intersezione $A_{i \bigcap j}$.

[7] Con "oggetto" si sottintende il *bounding box* relativo all'oggetto.

Per poter calcolare le coordinate dei vertici del riquadro di intersezione è necessario valutare la posizione dei lati dei due riquadri i e j.

Siano l_i^{right} e l_i^{left} rispettivamente le ascisse del bordo destro e del bordo sinistro di i e siano l_j^{right} e l_j^{left} rispettivamente le ascisse del bordo destro e del bordo sinistro di j. Le ascisse dei vertici posti sul lato sinistro del riquadro di intersezione saranno pari all'ascissa del bordo sinistro posto più a destra tra l_i^{left} e l_j^{left}, mentre le ascisse dei vertici posti sul lato destro del riquadro di intersezione saranno pari all'ascissa del bordo destro posto più a sinistra tra l_i^{right} e l_j^{right}.

Siano l_i^{up} e l_i^{down} rispettivamente le ordinate del bordo in alto e del bordo in basso di i e siano l_j^{up} e l_j^{down} rispettivamente le ordinate del bordo in alto e del bordo basso di j. Le ordinate dei vertici posti sul lato in alto del riquadro di intersezione saranno pari all'ordinata del bordo in alto posto più in basso tra l_i^{up} e l_j^{up}, mentre le ordinate dei vertici posti sul lato in basso del riquadro di intersezione saranno pari all'ordinata del bordo in basso posto più in alto tra l_i^{down} e l_j^{down}.

Sia v_{left}^{top} la posizione del vertice in alto a sinistra, sia v_{right}^{top} la posizione del vertice in alto a destra e sia v_{left}^{down} la posizione del vertice in basso a sinistra del riquadro di intersezione. Formalmente si ha:

$$v_{left}^{top} = \begin{pmatrix} max(l_i^{left}, l_j^{left}) \\ max(l_i^{up}, l_j^{up}) \end{pmatrix} = \begin{pmatrix} max(x_i - \frac{1}{2}w_i, x_j - \frac{1}{2}w_j) \\ max(y_i - \frac{1}{2}h_i, y_j - \frac{1}{2}h_j) \end{pmatrix} \tag{2.7}$$

$$v_{right}^{top} = \begin{pmatrix} min(l_i^{right}, l_j^{right}) \\ max(l_i^{up}, l_j^{up}) \end{pmatrix} = \begin{pmatrix} min(x_i + \frac{1}{2}w_i, x_j + \frac{1}{2}w_j) \\ max(y_i - \frac{1}{2}h_i, y_j - \frac{1}{2}h_j) \end{pmatrix} \tag{2.8}$$

$$v_{left}^{down} = \begin{pmatrix} max(l_i^{left}, l_j^{left}) \\ min(l_i^{down}, l_j^{down}) \end{pmatrix} = \begin{pmatrix} max(x_i - \frac{1}{2}w_i, x_j - \frac{1}{2}w_j) \\ min(y_i + \frac{1}{2}h_i, y_j + \frac{1}{2}h_j) \end{pmatrix} \tag{2.9}$$

L'area dell'intersezione tra i e j viene calcolata come segue:

$$A_{i \bigcap j} = w_{i \bigcap j} h_{i \bigcap j} = (v_{right\,x}^{top} - v_{left\,x}^{top})(v_{left\,y}^{down} - v_{left\,y}^{top}) \tag{2.10}$$

L'area dell'unione tra i e j viene calcolato come segue:

$$A_{i \bigcup j} = A_i + A_j = w_i h_i + w_j h_j \tag{2.11}$$

[16]

2.3.2 Matching di bounding box

Prima di valutare le prestazioni di un algoritmo di rilevamento è necessario associare ad ogni previsione generata il corrispondente *ground truth bounding box* dell'immagine di addestramento. Quest'operazione prende il nome di *matching di bounding box*. Alcuni oggetti potrebbero non essere rilevati, mentre altri

potrebbero essere rilevati più volte. Risulta necessario che ad ogni oggetto sia associata una sola previsione.

I risultati dell'inferenza sono spesso soggetti a due tipi di errore:

- errore di prima specie, che porta a classificare erroneamente una regione dello sfondo come oggetto;
- errore di seconda specie, che porta a classificare erroneamente un oggetto come parte dello sfondo;

Nel primo caso i risultati vengono classificati come falsi positivi, mentre nel secondo caso i risultati vengono classificati come falsi negativi. Le previsioni che corrispondono al rilevamento effettivo di un oggetto prendono il nome di veri positivi, mentre le regioni dell'immagine che vengono correttamente individuate come parte dello sfondo sono dette veri negativi.

La fase di *matching* consiste nel trovare tutte le coppie di indici (i, j) tali per cui ad ogni oggetto j sia associata una sola previsione i. Sia N l'insieme di tutte le coppie previsione-oggetto, inizialmente vuoto. Gli elementi contenuti in N rappresentano i veri positivi generati dall'algoritmo di rilevamento. Il criterio secondo cui formare le coppie è basato sulla sovrapposizione dei riquadri predetti con quelli reali.

L'algoritmo per la formazione delle coppie prevede due passi: il calcolo della sovrapposizione tra i riquadri predetti e quelli reali e la formazione delle coppie in base al valore della sovrapposizione. L'algoritmo seleziona ad una ad una le previsioni generate durante il rilevamento per poi calcolare l'*intersection over union* con ogni oggetto presente nell'immagine. Una volta stabilito un valore di soglia IoU_{min}, di solito pari a 0.5, l'algoritmo valuta se l'intersection over union tra la previsione i e l'etichetta j risulta superiore alla soglia, cioè $IoU(i, j) > IoU_{min}$. In caso affermativo, il valore $IoU(i, j)$ viene aggiunto all'insieme M, inizialmente vuoto, tale che:

$$M = \{(i, j, IoU(i, j)) | IoU(i, j) > IoU_{min}, i \in \{1, 2, ..., n\}, j \in \{1, 2, ..., m\}\}$$

dove n è il numero di previsioni ottenute con il rilevamento e m è il numero di oggetti presenti nell'immagine.

L'operazione precedente associa ad ogni previsione gli oggetti che si sovrappongono meglio. In questo caso, è possibile che ad un oggetto sia associata più di una previsione e viceversa. La non univocità del legame tra oggetti e previsioni può essere causata da errori dell'algoritmo, come rilevamenti multipli di uno stesso oggetto, o dalla vicinanza degli oggetti all'interno dell'immagine. Ad esempio, due olive molto vicine probabilmente saranno caratterizzate da una forte sovrapposizione dei loro riquadri di delimitazione. Una previsione generata in quella zona dell'immagine verrà quindi associata ad entrambe le olive.

Per ottenere una corrispondenza biunivoca tra previsioni ed oggetti bisogna mantenere solamente i migliori accoppiamenti migliori per ogni oggetto. A tale scopo, ad ogni elemento dell'insieme M si associa un indice in modo tale da stabilire tra le terne un ordinamento decrescente, in base al grado di sovrapposizione. La terna con l'indice più basso presenterà l'*intersection over union*

più alto in assoluto, quindi la coppia di riquadri (i, j) ad esso associata viene inserita in N. L'algoritmo scandisce tutte le terne seguendo l'ordinamento dell'insieme M, valutando per ogni terna se l'oggetto in essa contenuto sia già stato associato ad un'altra previsione e viceversa. Se la previsione i individuata dalla terna e l'oggetto j individuato dalla terna non sono ancora stati collegati, allora la coppia viene aggiunta ad N.

Sia TP il numero dei veri positivi (*True Positive*), sia FP il numero di falsi positivi (*False Positive*), sia FN il numero dei falsi negativi (*False Negative*), sia p il numero di tutte le previsioni generate dalla rete, sia o il numero di tutti gli oggetti presenti nell'immagine. Allora si ha:

$$\begin{aligned} FP &= p - TP \\ FN &= o - TP \end{aligned} \tag{2.12}$$

Un modello poco performante potrebbe scambiare foglie o altre parti dello sfondo come olive, generando falsi positivi, o potrebbe non riuscire a rilevare olive parzialmente coperte o poco illuminate, generando falsi negativi. [16]

2.3.3 Matrice di confusione

I risultati del rilevamento vengono organizzate in una tabella chiamata matrice di confusione binaria, composta da due righe e due colonne.

La prima colonna contiene gli elementi non previsti dall'algoritmo di rilevamento, i negativi, mentre la seconda contiene tutti gli elementi previsti, i positivi.

La prima riga contiene tutti gli elementi classificati come sfondo, mentre la seconda contiene tutti gli elementi classificati come oggetti.

L'intersezione tra una riga e una colonna individua una particolare tipologia di risultati. Ad esempio, l'intersezione tra la prima riga e la prima colonna individua i risultati classificati come veri negativi. Nei problemi di *object recognition*

		Valori predetti	
		n'	***p'***
Valori reali	***n***	Veri negativi	Falsi positivi
	p	Falsi negativi	Veri positivi

(a) Matrice di confusione binaria.

		Predetti			Somma
		Uva	Oliva	Ciliegia	
Reali	Uva	5	2	0	7
	Oliva	3	3	2	8
	Ciliegia	0	1	11	12
Somma		8	6	13	27

(b) Matrice di confusione multiclasse.

Figura 2.9: Esempi di matrici di confusione. [17]

con più categorie di oggetti da riconoscere i risultati vengono spesso organizzati in una matrice di confusione multi-classe, una matrice quadrata la cui dimensione è uguale al numero delle classi del problema. Ad ogni riga e ad ogni colonna viene associata una classe. La somma dei valori contenuti in una colonna indica

il numero di previsioni generate per quella classe, mentre la somma dei valori contenuti in una riga indica il numero di oggetti appartenenti ad una determinata classe effettivamente presenti nell'immagine. Sulla diagonale principale si può leggere per ogni classe il numero di veri positivi, che è dato dall'intersezione di una colonna e una riga con lo stesso indice. [17]

Sia TP il numero dei veri positivi (*True Positive*), sia FP il numero di falsi positivi (*False Positive*), sia FN il numero dei falsi negativi (*False Negative*) e sia TN il numero dei veri negativi (*True Negative*). Le prestazioni di un algoritmo di rilevamento vengono valutate a partire dalla matrice di confusione binaria, utilizzando le seguenti metriche:

- tasso di errore, rappresenta l'errore commesso dall'algoritmo di *object detection*, indica la percentuale di rilevamenti errati rispetto al totale, ed è dato da:
$$e = \frac{FP + FN}{FP + FN + TP + TN} \tag{2.13}$$

- accuratezza, rappresenta l'accuratezza delle previsioni effettuate dall'algoritmo di *object detection*, indica la percentuale di rilevamenti corretti rispetto al totale, ed è data da:
$$acc = \frac{TP + TN}{FP + FN + TP + TN} \tag{2.14}$$

- precisione, rappresenta l'abilità dell'algoritmo di *object detection* di non rilevare regioni dello sfondo come oggetti, indica la percentuale di rilevamenti corretti tra i positivi, ed è data da:
$$p = \frac{TP}{FP + TP} \tag{2.15}$$

- recupero, rappresenta l'abilità dell'algoritmo di *object detection* di rilevare tutti gli oggetti, indica la percentuale di rilevamenti corretti rispetto al numero di oggetti presenti nell'immagine, ed è dato da:
$$r = \frac{TP}{FN + TP} \tag{2.16}$$

Un'elevata precisione sta ad indicare che tra i rilevamenti effettuati quasi tutti rappresentano effettivamente un oggetto. Nonostante l'elevata precisione potrebbe capitare che molti oggetti non siano rilevati. Ciò corrisponde ad un elevato numero di falsi negativi.

Un elevato recupero sta ad indicare che l'algoritmo di rilevamento è in grado di individuare quasi tutti gli oggetti presenti nell'immagine. Il problema è che l'algoritmo oltre agli oggetti presenti nell'immagine potrebbe rilevare anche un gran numero di falsi positivi.

Per questo motivo precisione e recupero non vengono mai usate da sole come metriche di valutazione. Per misurare l'accuratezza dell'algoritmo solitamente si

esegue una media armonica ponderata delle due grandezze, che prende il nome di F1-score. Esso è definito come segue:

$$F1 = \frac{2}{\frac{1}{p} + \frac{1}{r}} = \frac{2pr}{p + r} \tag{2.17}$$

[16]

2.3.4 Mean average precision

Gli algoritmi di rilevamento per ciascun *bounding box*, oltre alle coordinate e alle dimensioni del riquadro, predicono anche una quantità detta livello di confidenza, che rappresenta una stima della fiducia che il modello ha nella correttezza della previsione. Il livello di confidenza viene espresso come una probabilità, quindi il suo valore varia nell'intervallo $[0, 1]$.

L'utilizzo delle metriche appena descritte non permette di tener conto dei livelli di confidenza associati ai vari rilevamenti. Questo problema viene risolto calcolando la precisione e il recupero per diversi livelli di soglia. Si stabilisce una soglia e si eliminano tutti i rilevamenti con livello di confidenza minore della soglia. Poi si esegue il *matching* dei *bounding box*, si crea la matrice di confusione e si calcolano i valori della precisione e del recupero.

I valori della precisione e del recupero calcolati per diversi livelli di soglia vengono utilizzati per tracciare il grafico della precisione in funzione del recupero (*precision at recall graph* o *PR graph*).

Ad ogni valore del recupero sull'asse delle ascisse si associa il corrispondente valore della precisione sull'asse delle ordinate, calcolato per lo stesso livello di soglia, ottenendo un insieme di punti sul piano cartesiano. I vari punti vengono interpolati per formare una curva, il cui andamento è sempre decrescente a meno di piccole oscillazioni.

Sia p la precisione e r il recupero. L'area sottesa dal grafico prende il nome di *average precision* (AP) ed è definita come segue:

$$AP = \int_0^1 p(r)d(r) \tag{2.18}$$

Nella pratica viene utilizzata una forma discreta, data da:

$$AP = \sum_k^n = 1p(k)\Delta r(k) \tag{2.19}$$

Spesso al posto del grafico precisione su recupero si utilizza una sua variante chiamata *11 points precision recall curve*, ottenuta suddividendo l'asse delle ascisse in dieci intervalli della stessa lunghezza e associando ad ognuno un diverso livello di recupero. Ciò equivale a mappare l'asse reale positivo sull'insieme $\{0, 0.1, 0.2, ..., 1.0\}$.

Per ogni livello di recupero viene calcolata una precisione interpolata, pari al valore più alto tra quelli che si trovano alla sua destra sul grafico discretizzato. In particolare si ha:

$$p_{interp}(r) = max_{\hat{r}:\hat{r}>r} p(\hat{r}) \tag{2.20}$$

Il grafico ottenuto presenta 11 coppie di punti che posso essere interpolate per ottenere una curva decrescente e priva di oscillazioni. L'andamento decrescente sta ad indicare che all'aumentare del numero degli oggetti rilevati diminuisce la precisione dei rilevamenti. Maggiore è il valore delle ascisse, maggiore sarà il numero di falsi positivi. Minore è il valore delle ascisse maggiore sarà la percentuale di rilevamenti corretti, anche se vengono effettuati pochi rilevamenti.

A partire dal grafico *11 points precision recall curve* l'*average precision* viene calcolata come segue:

$$AP = \frac{1}{11} \sum_{r \in \{0,0.1,\ldots,1.0\}} p_{interp}(r) \tag{2.21}$$

La media dei valori dell'*average precision* calcolati per ciascuna classe viene detta *mean average precision* (mAP). Nel rilevamento di olive le due metriche coincidono, poiché il problema ha una sola classe. [16]

2.3.5 Creazione dei bounding box

Se il modello ottenuto dall'addestramento, dopo averne valutato le prestazioni, viene giudicato accettabile esso può essere impiegato per l'inferenza. Il modello durante l'inferenza è in grado di localizzare la posizione degli oggetti all'interno dell'immagine, classificandoli in modo opportuno, ma i risultati forniti sono codificati in forma numerica. Risulta quindi necessario disegnare i riquadri all'interno dell'immagine in modo che i risultati siano facilmente comprensibili per l'essere umano.

```
1  def DrawBoundingBox(image,bbox_list):
2    draw=ImageDraw.Draw(image)
3    for bbox in bbox_list:
4        if bbox["conf"]>0.1:
5          text=bbox["classe"]+" "+str(int(bbox["conf"]*100)/100)
6          text_lenght,text_height=draw.textsize(text)
7          draw.rectangle([bbox["xmin"],bbox["ymin"],bbox["xmax"],bbox["
       ymax"]],width=5,outline="rgb(255,255,255)")
8          draw.rectangle([bbox["xmin"],bbox["ymin"]-text_height,bbox["
       xmin"]+text_lenght,bbox["ymin"]],width=5,fill="rgb(255,255,255)",
       outline="rgb(255,255,255)")
9          draw.text((bbox["xmin"],bbox["ymin"]-text_height),text,fill="
       rgb(0,0,0)")
10   return image
```

Spesso i riquadri generati dagli algoritmi di *object detection* sono ridondanti. Per questo motivo è necessario filtrare i risultati del rilevamento in modo che ad ogni oggetto sia associato un solo riquadro. A differenza della fase di *matching* dei *bounding box* durante l'inferenza non si hanno informazioni sulla posizione e la dimensione degli oggetti. Questo problema viene risolte applicando un algoritmo che prende il nome di soppressione dei non massimi (*Non Max Suppression - NMS*.

Sia P l'insieme di tutti i *bounding box* predetti dalla rete e sia F l'insieme dei riquadri ottenuti dopo il filtraggio, inizialmente vuoto.L'algoritmo seleziona il riquadro con il livello di confidenza più alto dall'insieme S. Questo riquadro è

quello che aderisce meglio di tutti ad un generico oggetto, quindi viene aggiunto all'insieme N ed eliminato da S. Il *bounding box* viene confrontato con tutti gli altri presenti nell'insieme S valutandone la sovrapposizione con l'*intersection over union*. I riquadri che presentano un valore di *intersection over union* superiore ad una certa soglia vengono classificati come ridondanti, quindi vengono eliminati da S. L'algoritmo ripete l'operazione ad ogni iterazione fino a quando l'insieme S non risulta vuoto. [18]

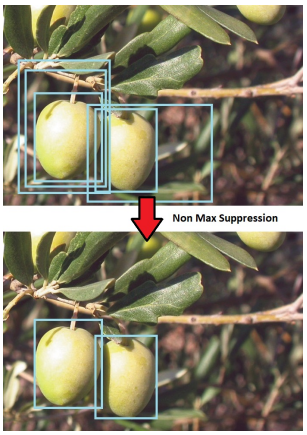


Figura 2.10: Soppressione dei non massimi su un'immagine di olive.

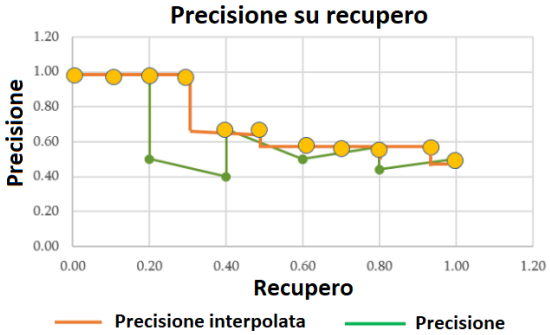


Figura 2.11: Grafico della precisione in funzione del recupero valutata a diverse soglie di confidenza.

Capitolo 3

Reti neurali

La tecnica proposta nella sezione 2.1.5 risulta particolarmente efficacie nel rilevamento di pedoni ed è in grado di eseguire rilevamenti in tempo reale. Nonostante ciò questa tecnica, come le altre basate su modelli di apprendimento automatico, presenta un grosso limite, risulta difficile da estendere ad altre classi di oggetti. Ogni classe di oggetto richiede infatti un diverso metodo per l'estrazione di caratteristiche, questo richiede che per rilevare diversi oggetti in un'immagine servono tanti algoritmi diversi quanto sono gli oggetti.

Per anni non ci furono progressi e gli algoritmi di *object detection* rimasero limitati a poche applicazioni, come videosorveglianza e riconoscimento facciale. La svolta si ebbe quando si iniziarono ad usare modelli di rilevamento estremamente complessi e non lineari, detti reti neurali.

L'applicazione delle reti neurali al rilevamento di oggetti è molto recente e risale all'ultimo decennio, nonostante i primi modelli siano stati sviluppati già intorno la seconda metà del Novecento.

3.1 Cenni storici e basi neuroscientifiche

3.1.1 Neurone biologico

Negli esseri umani il sistema nervoso è l'unita che si occupa di ricevere, trasmettere ed elaborare gli stimoli, rappresentati da segnali bioelettrici, consentendogli di relazionarsi con l'ambiente in cui si trova. Esso si occupa di funzioni muscolari, sensoriali, psichiche ed intellettive. I componenti elementari che lo costituiscono sono i neuroni, cellule nervose connesse tra di loro in modo da formare delle reti chiamati reti neurali.

Le cellule nervose sono costituite da un corpo cellulare chiamato soma, rivestito da una membrana polarizzata in grado di mantenere una concentrazione di cariche elettriche. I neuroni riescono a comunicare tra di loro attraverso due tipi di prolungamenti, che hanno origine dal soma:

- i dendriti, fibre nervose che si diramano ramificandosi come le radici di un albero trasportando i segnali in ingresso al neurone;
- l'assone, il prolungamento principale, che trasporta il segnale in uscita dal neurone.

Il sistema nervoso è in grado di acquisire degli input dal mondo esterno sotto forma di piccole quantità di energia attraverso degli organuli chiamati recettori. Questi stimoli per essere elaborati devono essere convertiti in segnali elettrici attraverso un processo di trasduzione. I recettori possono quindi essere immaginati come dei sensori, capaci di acquisire segnali di diversa natura e di trasformarli in segnali elettrici.

Le capacità di apprendimento e memoria sono legate al modo in cui si attivano le reti nervose quando sono soggette a degli stimoli. Il neurone riceve in ingresso, attraverso i dendriti, degli impulsi costituiti da cariche elettriche in movimento, il cui compito è quello di eccitare la membrana che avvolge il soma. Le cariche che raggiungono un neurone vengono accumulate grazie alla polarizzazione della membrana esterna. Quando la concentrazione di cariche elettriche nella membrana raggiunge la soglia massima, il neurone si attiva, generando una scarica elettrica attraverso l'assone.

I dendriti trasportano impulsi in direzione centripeta rispetto alla cellula, mentre l'assone li trasporta in direzione centrifuga. Nei punti di interazione tra due cellule nervose, detti sinapsi, il segnale proveniente da un assone viene trasmesso attraverso un dendrite. Lo spessore di una sinapsi all'interno di una rete non è una grandezza costante, ma può variare nel tempo in relazione alla frequenza con cui comunicano i due neuroni associati.

L'aumento dello spessore di una sinapsi sta ad indicare il rafforzamento della connessione tra i due neuroni coinvolti, mentre la sua diminuzione sta ad indicare un suo indebolimento. Questo fenomeno genera dei percorsi preferenziali all'interno della rete, dove gli impulsi viaggiano con maggiore frequenza. [19, 20]

3.1.2 Il primo modello

In una rete neurale è l'area di memoria viene condivisa con quella di elaborazione. I neuroni, infatti, sono responsabili sia di immagazzinare informazioni sia di elaborarle. Nell'informatica classica dati e codice sono sempre stati gestiti in maniera separata, sia dal punto di vista software[1] che hardware[2].

Le cellule nervose appartenenti ad una stessa rete hanno la capacità di attivarsi in maniera simultanea, consentendo l'elaborazione in parallelo di più impulsi. Se un neurone non funziona in maniera corretta, l'impulso segue un percorso diverso e l'elaborazione prosegue senza bloccare la rete. Al contrario, i programmi classici solitamente presentano una struttura sequenziale dove i dati vengono elaborati in serie.

Durante i primi anni Cinquanta, lo studio e la ricerca nel campo delle reti neurali suscitarono un notevole interesse, che portò ai primi tentativi di modellare in termini matematici il comportamento del cervello umano.

Nel 1943 Warren McCulloch e Walter Pitts presentarono il primo modello formale di neurone artificiale, basandosi sulle teorie neuroscientifiche dell'epoca, tutt'ora valide. Il modello era basato su un combinatore lineare a soglia di tipo

[1] In un programma tradizionale i blocchi di codice e i dati sono caricati in segmenti diversi della memoria.

[2] L'architettura di un calcolatore prevede un'unità di elaborazione, CPU, e una memoria volatile, RAM, fisicamente distinte.

MISO(Multiple Input Single Output), che presentava un'uscita binaria. Questi neuroni potevano essere combinati insieme in modo da formare una rete in grado di calcolare semplici funzioni booleane. Il sistema si auto-migliorava con l'esperienza procedendo per tentativi ed errori, come il cervello umano.

Nel 1949 Donald Hebb ipotizzò che l'apprendimento biologico avvenisse a livello sinaptico. Secondo Hebb, il rafforzamento della connessione tra due neuroni interconnessi era dovuto alla loro attivazione simultanea e ricorrente durante un certo evento, in modo tale da conservare nel tempo il ricordo di quel particolare evento.[19]

3.1.3 Percettrone

Nel 1957 Frank Rosenblatt presentò un nuovo modello di neurone artificiale, chiamato percettrone. Il percettrone è un classificatore binario che mappa i suoi ingressi, rappresentati da un vettore $u \in \mathbb{R}^{\mathbb{N}}$ di numeri reali, in un valore di output $y \in [0, 1]$, calcolato mediante un'opportuna funzione.

Il valore dell'uscita non dipende solo dal vettore degli ingressi, ma anche da un vettore di variabili di stato w, le cui componenti prendono il nome di pesi, e uno scalare $b \in \mathbb{R}$ detto *bias*. L'uscita del percettrone è data da:

$$y = \chi(< w, u > + b) \tag{3.1}$$

La somma tra il *bias* e il prodotto scalare del vettore dei pesi con quello degli ingressi viene detta attivazione o stato di attivazione del percettrone. L'uscita del percettrone si ottiene applicando la funzione di output $\chi : \mathbb{R} \to [0, 1]$ allo stato di attivazione. Il percettrone viene considerato come il più semplice modello di rete neurale.

Gli impulsi che viaggiano all'interno di una rete neurale biologica, vengono rappresentati in forma matematica come delle grandezze variabili nel tempo. Ogni variabile e ogni vettore del percettrone rappresenta un elemento del neurone biologico, mentre ogni funzione ne rappresenta un diverso comportamento:

- nel percettrone i dendriti vengono modellati come canali di input, rappresentati dal vettore u, capaci di acquisire impulsi in ingresso[3];
- nel percettrone la scarica elettrica lungo l'assone viene modellata dallo scalare $y \in \mathbb{R}$, che rappresenta l'uscita del modello;
- nel percettrone la polarizzazione della membrana elettrica viene modellata dalla funzione di output, che ha il compito di generare l'uscita quando il suo argomento supera una certa soglia;
- nel percettrone la funzione delle sinapsi viene svolta dal prodotto scalare tra il vettore dei pesi w e quello degli ingressi u[4];

[3] Ogni impulso in ingresso nel percettrone viene rappresentato da un numero reale e costituisce una componente del vettore u. La dimensione di u rappresenta il numero di dendriti associati al percettrone.

[4] Le componenti del vettore dei pesi w associato ad un percettrone, rappresentano lo spessore delle sue "sinapsi".

- il rafforzamento e l'indebolimento della connessione tra due percettroni avviene modificando il valore del peso ad esso associato[5];

Il modello presentato da Rosenblatt rappresenta una rete neurale supervisionata a due livelli di neuroni, detti strati:

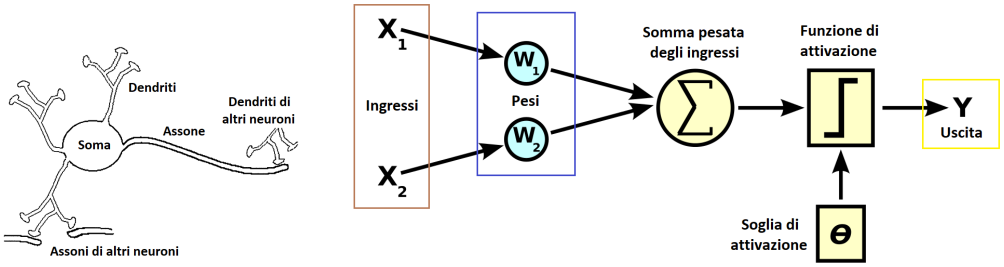
- uno strato di input, rappresentato dal vettore degli ingressi u;
- uno strato di output, rappresentato dal valore dell'uscita y

Questa struttura si rivelò essere troppo semplice, poiché capace di riconoscere solamente funzioni linearmente separabili. [19, 21] Due insiemi di punti sono separabili da un iperpiano essi vengono detti linearmente separabili. Siano X_0 e X_1 due insiemi di punti in uno spazio euclideo n-dimensionale. Allora X_0 e X_1 si dicono linearmente separabili se esistono $n+1$ numeri reali $w_1, w_2, \ldots, w_n, k$ tali che:

$$\begin{aligned} \forall x \in X_0 \sum_{i=1}^{n} w_i x_i > k \\ \forall x \in X_1 \sum_{i=1}^{n} w_i x_i < k \end{aligned} \tag{3.2}$$

[22]

Nel 1969 Marvin Minsky e Seymour A. Papert dimostrano che le reti a due strati basate sui percettroni non erano in grado di calcolare l'OR esclusivo (XOR), funzione alla base della logica booleana, facendo presto diminuire l'interesse nello studio dello reti neurali. Ben presto fu intutito che il problema potesse essere risolto aggiungendo dei livelli di neuroni intermedi tra lo strato di input e quello di output. Questa soluzione risultò però impraticabile per via della crescente complessità del modello all'aumentare del numero di strati della rete. [19, 21]



(a) Struttura del neurone biologico. [19]

(b) Modello matematico del percettrone. [21]

Figura 3.1: Analogie tra percettrone e neurone biologico.

3.1.4 Cognitrone

Intorno alla metà degli anni Cinquanta David Hubel e Torsten Wiesel fecero nuove scoperte sul funzionamento della corteccia visiva animale. In particolare,

[5] Questo è il principio alla base dell'apprendimento delle reti neurali

dimostrarono che i neuroni della corteccia cerebrale dei gatti e delle scimmie sono associati a piccole regioni del campo visivo, dette campi recettivi.

I neuroni vicini sono caratterizzati da campi recettivi simili e sovrapposti tra di loro. La sovrapposizione dei vari campi recettivi va a formare una mappa completa del campo visivo. La corteccia dell'emisfero destro contiene una mappa del campo visivo sinistro e viceversa.

Nel 1968 gli stessi studiosi identificarono due diversi tipi di cellule visive presenti nel cervello:

- cellule semplici, la cui uscita generata è massima quando sono presenti bordi rettilinei con particolari orientamenti all'interno del loro campo recettivo;
- cellule complesse, che presentano grandi campi recettivi e generano un'uscita insensibile alla posizione dei bordi che riescono a rilevare.

[10] Nel 1975 lo scienziato giapponese Kunihiko Fukushima, ispirato dalle scoperte di Hubel e Wiesel, propose un modello di rete neurale non supervisionata chiamata cognitrone, specializzata nel riconoscimento di pattern. La rete è costituita da quattro strati organizzati tra di loro in modo gerarchico. Il primo strato acquisisce gli stimoli dall'esterno della rete, simulando il comportamento della retina, mentre il terzo strato simula il ruolo della corteccia celebrale.

I neuroni di ogni strato sono organizzati sotto forma di matrice, in modo tale che ognuno di loro sia spazialmente circondato da altri neuroni. Ogni gruppo di neuroni vicini individua una sottomatrice che prende il nome di area di connessione. La connessione tra due neuroni è rinforzata solamente se il neurone post-sinaptico presenta un livello di attivazione maggiore rispetto a quelli dell'area di connessione del neurone pre-sinaptico.

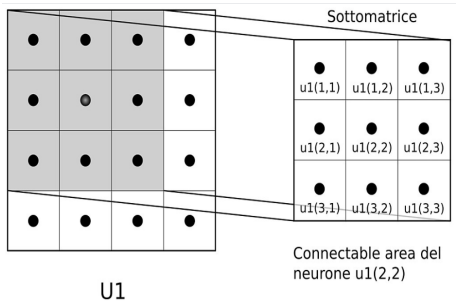


Figura 3.2: Esempio di area di connessione di un neurone del primo strato. [23]

A differenza del percettrone, nel cognitrone le connessioni sinaptiche tra le varie unità hanno la capacità di riorganizzarsi in modo autonomo reagendo in modo selettivo quando viene presentato un nuovo pattern. La ripetizione di uno stesso stimolo induce il cognitrone ad associare lo stimolo stesso a uno specifico insieme di neuroni della corteccia.

Questo modello tiene conto delle limitazioni biologiche dei neuroni, poiché presenta un numero ridotto di connessioni rispetto alle reti tradizionali dove tutti i neuroni erano completamente connessi.

Lo stesso Fukushima nel 1979 propose un nuovo modello basato sul cognitrone, detto neocognitrone, che era in grado di garantire invarianza a trasformazioni come rotazione, traslazione e deformazione,i dei componenti di un immagine fornendo gli stessi risultati. [23]

3.1.5 Reti neurali profonde

Il problema dell'"incapacità delle reti neurali di elaborare funzioni non linearmente separabili fu risolto solo nel 1986, quando David E. Rumelhart, G. Hinton e R. J. Williams introdussero il concetto di *hidden layers* e l'algoritmo di retropropagazione dell'errore (*backpropagation*) portando alla nascita del percettrone multistrato (*Multi-Layer-Perceptron, MLP*).

Il concetto di *hidden layers* prevede di inserire un insieme di livelli di neuroni nascosti tra lo strato di input e quello di output del percettrone, mentre la (*backpropagation*) è l'algoritmo che consentì per la prima volta di addestrare reti così complesse.

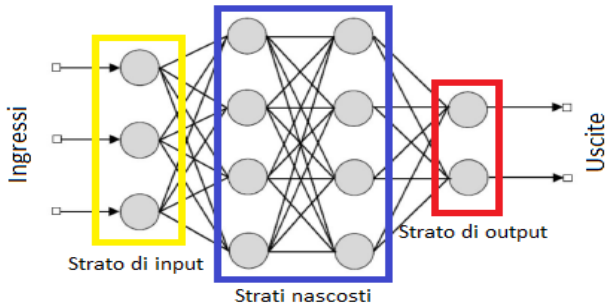


Figura 3.3: Architettura del percettrone multistrato. [14]

Il modello ha una struttura gerarchica, dove ogni percettrone ha come ingressi le uscite dei neuroni dello strato precedente. Il funzionamento del percettrone multistrato è basato sul fatto che le capacità intellettuali possono emergere dal comportamento collettivo di un gran numero di entità considerate di per sé stupide. [19, 21]

L'evento fu talmente importante da far nascere un nuovo campo di studi, all'interno dell'apprendimento automatico, detto apprendimento profondo (*deep learning*). Esso si basa su modelli che contengono diversi strati non lineari in cascata, detti reti neurali profonde (*deep neural network*)[6], dove ogni strato calcola i valori per quello successivo elaborando l'informazione in maniera sempre più completa.

Gli algoritmi di apprendimento profondo rappresentano i concetti in maniera gerarchica, esprimendoli mediante più livelli di astrazione. I concetti e le caratteristiche di alto livello sono definiti sulla base di quelli di basso livello. Questo struttura risulta particolarmente adatta per l'estrazione di caratteristiche dai dati. Ogni strato della rete si occupa di imparare a rilevare una particolare caratteristica dei dati, caratteristiche che vengono combinate insieme strato per strato fino a formare l'output.

[6]Il percettrone multistrato appartiene alla categoria delle reti neurali profonde.

Nel 1998 Yann LeCun propose **LeNet 5**[7], una particolare architettura di rete neurale in grado di riconoscere caratteri scritti a mano a partire da un'immagine in scala di grigi. [24]

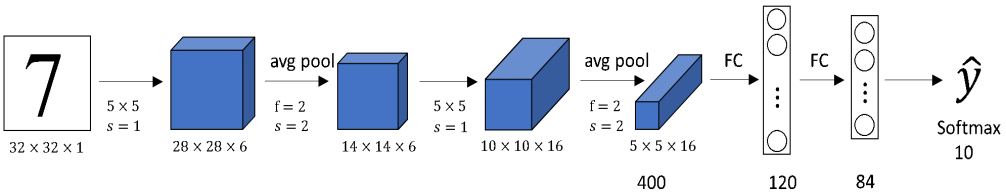


Figura 3.4: Architettura della rete neurale **LeNet 5**. L'input è rappresentato da un tensore di dimensioni $32 \times 32 \times 1$, cioè un'immagine in scala di grigi. La rete presenta due strati convoluzionali alternati a due strati di *average pooling* per estrarre le caratteristiche dall'immagine ed utilizza due strati *fully-connected* combinati con uno *softmax* per la classificazione dei pattern. [24]

Nel campo della visione artificiale l'introduzione delle reti profonde consentì per la prima volta di estrarre caratteristiche in maniera automatica dalle immagini. La fase di apprendimento delle reti profonde veniva svolta su dataset di dimensioni molto ridotte e spesso veniva interrotta prima di ottenere un modello soddisfacente, a causa dell'elevato numero di connessioni sinaptiche che rendeva l'addestramento estremamente lento. Questo fatto rese impraticabile l'applicazione delle reti profonde ai problemi di visione artificiale. [25]

3.1.6 Innovazioni hardware

Nell'ultimo decennio la diffusione dei *social network* e l'avvento dell'*internet of things* hanno portato ad un aumento esponenziale del volume dei dati scambiati in rete, rendendo necessari nuovi componenti *hardware* in grado di immagazzinare e processare i cosiddetti *big data*.

Il grande interesse intorno al mondo dei videogiochi e della fotografia in alta definizione ha reso necessario lo sviluppo di coprocessori grafici specializzati, detti *Graphics Processing Unit* (*GPU*). Questi coprocessori altamente specializzati consentono di ridurre il carico di lavoro del processore quando le applicazioni grafiche richiedono di eseguire una stessa operazione su gruppi di *pixel*, come nel *rendering* o nelle accelerazioni grafiche.

Lo sviluppo dell'elettronica e dei componenti *hardware* influenzò però anche il mondo del *deep learning*. L'avvento di dispositivi di memorizzazione di capacità sempre più elevate permise di realizzare dataset di immagini etichettate di dimensioni sempre più grandi. Nel 2009 nacque il dataset **ImageNet**, realizzato raccogliendo ed etichettando le immagini attraverso il servizio di *crowdsourcing* *Amazon Mechanical Turk* (*MTurk*).

Nel 2015 la Microsoft presentò **COCO** (*Common Objects in Context*), un dataset di 328.000 immagini contenente un totale di 2,5 milioni di oggetti etichettati appartenenti a 91 classi diverse.

[7] L'architettura **LeNet 5** rappresenta uno dei primi modelli di reti neurali convoluzionali, che saranno descritte nella terza sezione.

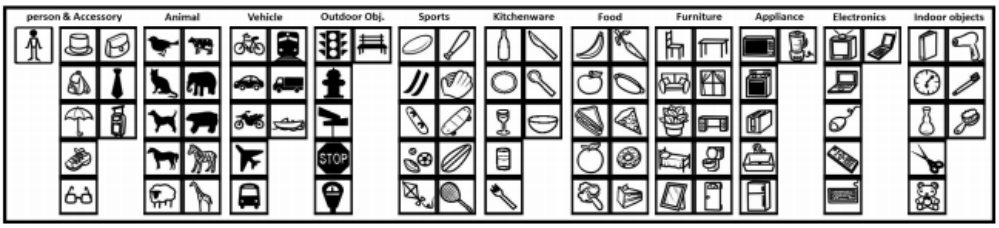


Figura 3.5: Insieme delle classi contenute nel dataset **Microsoft COCO**. [26]

L'utilizzo di dataset numerosi e diversificati nell'addestramento delle reti profonde dimostrò che gli scarsi risultati ottenuti fino a quel momento fossero dovuti alla scarsa quantità di esempi proposti. L'apprendimento di una quantità così vasta di dati fu resa possibile impiegando le *graphics processing unit* durante l'addestramento.

I coprocessori grafici vengono classificati come *SIMD* (*Single Instruction Multiple Data*) secondo la tassonomia di Flynn, poiché sono in grado di eseguire lo stesso flusso di istruzioni su molteplici flussi di dati risultando molto abili nel calcolo in parallelo. Questa loro caratteristica risulta estremamente adatta per l'addestramento delle reti neurali, caratterizzato dall'esecuzione della stessa operazione, cioè l'aggiornamento dei pesi e dei *bias*, su tante unità diverse, i neuroni.

Nel 2012 Alex Krizhevsky propose **AlexNet**, uno dei primi modelli di rete neurale ad essere implementato su *graphics processing unit*. Durante l'*ImageNet Large Scale Visual Recognition Challenge* (*ILSVRC*)[8] del 2012, **AlexNet** surclassò i vari algoritmi concorrenti, rendendo per la prima volta nella storia una rete profonda più performante rispetto ai metodi tradizionali. **AlexNet** è stata una delle prime reti neurali in grado di processare immagini a colori diventando un punto di riferimento per i modelli sviluppati successivamente. [24]

La nascita di questi modelli combinata con l'avvento di **COCO**, **ImageNet** e altri dataset permise per la prima volta di estendere il riconoscimento di oggetti ad un dominio di classi potenzialmente infinito. Mentre i metodi tradizionali basati sul *machine learning* erano in grado di riconoscere in maniera efficiente esclusivamente pedoni o volti, le reti neurali si dimostrarono incredibilmente versatili per riconoscere qualunque tipo di oggetto.

Tutto questo portò alla diffusione del *transfer learning*, una pratica che consiste nell'eseguire l'addestramento su modelli pre-allenati modificando solo i parametri associati agli strati finali. I modelli pre-allenati sono ottenuti addestrando una rete neurale su dataset numerosi e vari per svariate epoche, utilizzando potenti architetture *hardware*. Questa tecnica permette di ridurre i tempi di addestramento e al tempo stesso permette di ottenere buoni modelli anche utilizzando pochi dati.

L'addestramento e l'inferenza di un algoritmo di apprendimento profondo oggi possono essere eseguiti su qualunque dispositivo, compresi *smarthphone* e

[8] L'*ImageNet Large Scale Visual Recognition Challenge* è una competizione in cui vengono valutate le prestazioni di diversi algoritmi di *object detection* sul dataset ImageNet, con lo scopo di stimolare la ricerca e valutare i progressi nel campo della visione artificiale

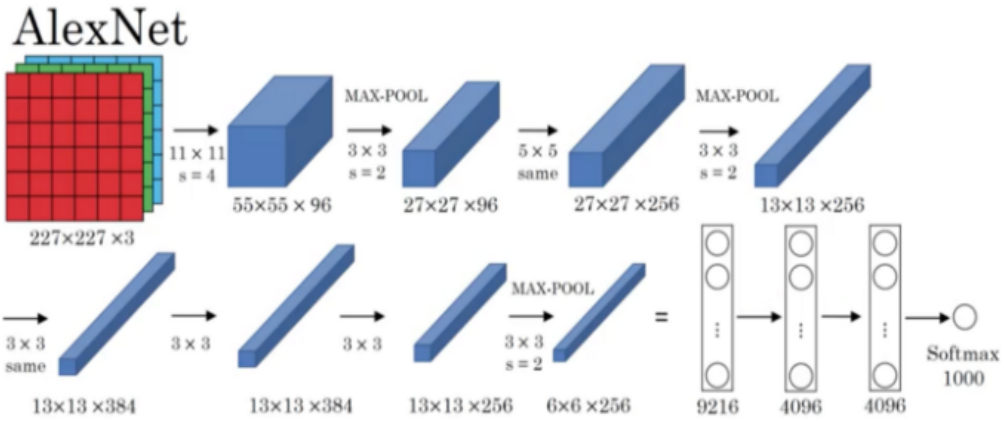


Figura 3.6: Architetura della rete neurale **AlexNet**. La rete presenta 8 strati per l'estrazione di caratteristiche, 5 di tipo convoluzionale e 3 di tipo *pooling*, e 3 strati per la classificazione, 2 di tipo *fully-connected* e uno strato finale *softmax*. All'uscita di uno strato convoluzionale viene prima applicata la funzione di attivazione e poi esegue il *pooling*. A differenza delle reti tradizionali, in alcuni casi, l'operazione di *pooling* viene omessa per preservare la risoluzione spaziale dell'immagine. [24]

tablet, sfruttando gli strumenti di *cloud computing*.

3.2 Reti neurali feed-forward

3.2.1 Grafo di rete

Le reti neurali possono essere rappresentate attraverso un grafo orientato, descritto da un insieme finito di punti chiamati nodi e un insieme finito di frecce chiamate archi. Ogni nodo corrisponde ad un neurone, mentre ogni arco corrisponde ad una sinapsi. Il verso associato ad un arco indica la direzione del flusso di informazioni, che è sempre diretto da un assone verso i dendriti. [27]

Un insieme di archi e nodi non ripetuti che costituiscono un cammino chiuso prende il nome di ciclo. Una rete neurale dove le connessioni tra i neuroni non formano cicli viene detta *rete neurale feed-forward*.

Vengono, invece, detta *rete neurale ricorrente* una rete che consente la presenza di cicli all'interno del suo grafo.

Le reti *feed-forward* sono caratterizzate da un flusso di informazioni unidirezionale. I dati si propagano in avanti, dall'input verso l'output, attraversando gli strati nascosti. Ogni strato riceve in ingresso informazioni solo dagli strati precedenti e invia informazioni solo agli strati successivi.

Queste reti non hanno memoria dei valori degli ingressi agli strati precedenti, quindi l'uscita è determinata esclusivamente dal valore attuale dell'ingresso. Il percettrone e il percettrone multistrato rientrano nella categoria di rete *feed-forward*. [28]

3.2.2 Modello in forma matriciale

Sia data una rete neurale formata da N neuroni artificiali. Ogni unità riceve segnali di input, rappresentati dalle uscite dei neuroni dello strato precedente.

Sia $x_j^{(k-1)} \in \mathbb{R}^n$ l'output del neurone j, appartenente al (k-1)-esimo strato della rete. Il valore di $x_j^{(k-1)}$ viene trasmesso al neurone i dello strato successivo k, opportunamente pesato.

La forza sinaptica della connessione tra i e j è rappresentata dall'elemento $W_{ij}^{(k)}$ della matrice $W^{(k)}$. Sia m il numero di neuroni nello strato k e sia n il numero di neuroni nello strato $k-1$. La matrice $W^{(k)} \in \mathbb{R}^{mxn}$ viene detta matrice dei pesi associata allo strato k. Questa matrice rappresenta i pesi delle connessioni tra lo strato $k-1$ e lo strato k. Le m righe individuano i neuroni che ricevono i segnali in ingresso, mentre le n colonne individuano il neurone che producono le attivazioni. L'elemento dell'i-esima riga e della j-esima colonna della matrice è rappresentato dal peso sinaptico che scala il valore in uscita da j e in ingresso ad i. [19]

- Se $0 < |W_{ij}^{(k)}| < 1$, il segnale in ingresso viene attenuato sul segnale in ingresso.
- Se $W_{ij}^{(k)} > 1$, il segnale in ingresso viene amplificato.
- Se $W_{ij}^{(k)} < 0$, il segnale in ingresso viene inibito.

Gli ingressi di un neurone concorrono a formare il suo stato di attivazione, ognuno con una diversa influenza. Il valore di attivazione è calcolato come la somma pesata di tutti gli ingressi. Sia $z_i^{(k)} \in \mathbb{R}^m$ lo stato di attivazione del neurone i, allora:

$$z_i^{(k)} = \sum_{j=1}^{n} W_{ij}^{(k)} x_j^{(k)} \tag{3.3}$$

Sia T la funzione di attivazione del neurone. Applicando la funzione di attivazione allo stato di attivazione si ottiene l'uscita del neurone. Questa funzione ha lo scopo di trattenere l'attivazione all'interno del neurone fino ad un certo livello di soglia, oltre il quale avviene la scarica. L'uscita del neurone i è data da:

$$x_i^{(k)} = T(\sum_{j=1}^{n} W_{ij}^{(k)} x_j^{(k)}) \tag{3.4}$$

Alcune reti neurali modificano lo stato di attivazione dei neuroni di uno strato sommando al prodotto tra la matrice dei pesi e il vettore degli ingressi un vettore $b^{(k)} \in \mathbb{R}^m$, detto *bias*. Poiché la funzione di attivazione è la stessa per tutti i neuroni appartenenti allo stesso strato, l'uscita dello strato k è data da:

$$x^{(k)} = T \begin{pmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_n^{(k)} \end{pmatrix} = T \left(\begin{pmatrix} W_{11}^{(k)} & W_{12}^{(k)} & \dots & W_{1n}^{(k)} \\ W_{21}^{(k)} & W_{22}^{(k)} & \dots & W_{2n}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{m1}^{(k)} & W_{m2}^{(k)} & \dots & W_{mn}^{(k)} \end{pmatrix} \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ \vdots \\ x_n^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(k-1)} \\ b_2^{(k-1)} \\ \vdots \\ b_n^{(k-1)} \end{pmatrix} \right) \tag{3.5}$$

L'uscita di ogni neurone risulta essere una funzione delle uscite di quelli dello strato precedente. Modificando il valore di un peso del primo strato, quindi, non viene alterata solo l'uscita del neurone corrispondente, ma anche le uscite di tutti i neuroni degli strati successivi. L'uscita generata della rete risulta quindi essere una funzione composta delle uscite degli strati nascosti e più in particolare dei parametri associati alle connessioni tra i vari neuroni, come pesi e *bias*.

Sia y l'uscita della rete neurale, rappresentata dal valore delle uscite dei neuroni dell'ultimo strato, e sia u il suo ingresso. Essa può essere espressa come funzione dei pesi e dei *bias* dei vari strati:

$$\begin{aligned} y &= T(W^{(N)}x^{(N)} + b^{(N)}) \\ y &= T(W^{(N)}T(W^{(N-1)}x^{(N-1)} + b^{(N-1)}) + b^{(N)}) \\ y &= T(W^{(N)}T(\dots T(W^{(1)}u + b^{(1)}) + \dots) + b^{(N)}) \end{aligned} \tag{3.6}$$

Uno strato della rete k in cui ogni neurone è collegato con tutti i neuroni dello strato successivo, viene definito *fully-connected layer* (strato pienamente connesso). Il percettrone multistrato è un rete composta esclusivamente da strati *fully-connected*.

3.2.3 Funzione di attivazione

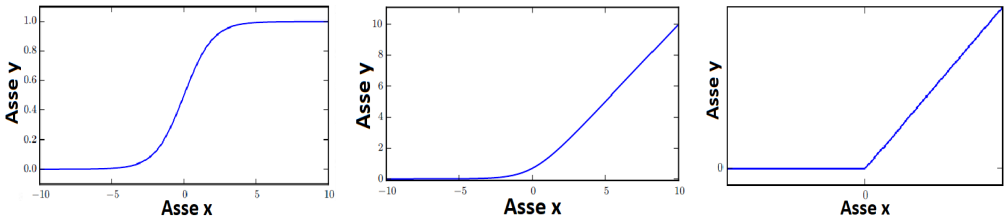
La funzione di attivazione è stata definita come una generica funzione T applicata allo stato di eccitazione di un neurone. La scelta di T solitamente ricade su una funzione non lineare, poiché la non linearità consente alla rete di imparare caratteristiche più complesse a partire dai dati. Le più utilizzate sono:

- la *sigmoide logistica*[9], definita da $\mathbb{R} \to [0, 1]$, una funzione che per valori elevati in modulo del suo argomento va in saturazione, diventando piatta e insensibile a piccole variazioni dell'ingresso:
$$\delta(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x} \tag{3.7}$$
- la *rectified linear unit (ReLU)*, definita da $\mathbb{R} \to [0, \infty]$, una funzione a rampa che restituisce la parte positiva di un numero:
$$x^+ = max(0, x) \tag{3.8}$$
- la *softplus*[10], definita da $\mathbb{R} \to [0, \infty]$, una funzione che ha un andamento più morbido e smussato rispetto alla *ReLU*.
$$\zeta(x) = \ln(1 + e^x) \tag{3.9}$$

[14, 10]

[9] La *sigmoide logistica* viene utilizzata per stimare il valore del parametro di una distribuzione Bernoulliana, che indica la probabilità di successo della prova. Il particolare andamento della funzione può essere sfruttato per eseguire una classificazione binaria dei dati di input.

[10] La *softplus* viene utilizzata per stimare i valori dei parametri β e σ di una distribuzione normale standard, che rappresentano rispettivamente la media e lo scarto quadratico medio associati alla distribuzione.



(a) Funzione *sigmoide logistica*. (b) Funzione *softplus*. (c) Funzione *ReLU*.

Figura 3.7: Esempi di funzione di attivazione. [10]

3.2.4 Backpropagation

La fase di apprendimento nelle reti neurali *feed forward* avviene in maniera supervisionata, applicando l'algoritmo di retropropagazione dell'errore, inizialmente introdotto per l'addestramento del percettrone multistrato.

In particolare, le coppie di valori ingresso-uscita contenute nel dataset di addestramento vengono analizzate dalla rete neurale per diverse volte fino a quando essa non riesce ad apprendere la relazione che lega gli input e gli output.

La rete modifica sistematicamente il valore delle sue connessioni sinaptiche, rinforzandone alcune e indebolendone altre, in modo tale che la sua funzione di trasferimento somigli quanto più possibile alla relazione tra gli ingressi e le uscite degli esempi. Al tempo stesso, la rete deve essere in grado di generalizzare apprendendo una funzione di trasferimento che sia valida anche per altri *set* di esempi, evitando situazioni di *overfitting*.

Il risultato è la formazione di percorsi preferenziali all'interno della rete individuati dalla connessioni con pesi maggiori. I segnali che viaggiano attraverso questi percorsi avranno una maggiore influenza sull'uscita generata dalla rete.

L'algoritmo di *backpropagation* prevede due fasi:

- il *forward pass* (propagazione in avanti), in cui viene calcolato il valore degli stati di eccitazione dei neuroni per poi applicare ad essi le funzioni di attivazione;

- il *backward pass*[11](propagazione all'indietro), in cui viene alterato il valore dei pesi e dei *bias*, sommando ad essi un termine correttivo, in modo da ridurre l'errore tra l'uscita predetta e quella desiderata facendo aderire il modello ai dati di addestramento;

Idealmente, il valore dell'errore, calcolato alla fine del *forward pass*, viene fatto propagare all'indietro fino a raggiungere le varie connessioni sinaptiche modificando il loro stato. In realtà, l'errore non si propaga all'indietro, ma viene utilizzato per calcolare i termini correttivi con cui modificare il valore delle connessioni attraverso una derivazione a catena.

L'algoritmo di retropropagazione dell'errore utilizza come metodo di ottimizzazione della funzione di perdita la discesa del gradiente *mini-batch* (*mini-batch*

[11]Durante il *forward pass* le informazioni si propagano dall'ingresso verso l'uscita, mentre durante il *backward pass* le informazioni si propagano dall'uscita verso l'ingresso.

Gradient Descend), una combinazione tra la discesa del gradiente stocastica e quella *batch*.

Nella tecnica *mini-batch* il calcolo del gradiente e l'aggiornamento dei parametri viene eseguito dopo aver analizzato un certo numero di esempi, che prende il nome di lotto (*batch*). La dimensione del lotto, detta *batch size*, è un parametro arbitrario della rete che viene scelto a seconda del problema da risolvere e a seconda dell'*hardware* a disposizione[12].

Sia b il valore del *batch size*. L'algoritmo esegue il *forward pass* per ogni esempio contenuto nel lotto. Dopo aver processato b calcola la funzione di perdita, a partire dalle discrepanze tra le uscite desiderate e quelle stimate, e ne valuta il gradiente. Ogni componente del gradiente esprime la variazione della funzione di perdita in relazione alla variazione del parametro della rete ad essa associato.

L'algoritmo altera il valore delle varie connessioni sottraendo un termine proporzionale alla derivata parziale della funzione di perdita rispetto al parametro considerato. Il peso della connessione tra il j-esimo neurone dello strato $k-1$ e l'i-esimo neurone dello strato k viene aggiornato nel seguente modo:

$$W_{ij}^{(k)} = W_{ij}^{(k)} - \epsilon \frac{\partial C}{\partial W_{ij}^{(k)}} \tag{3.10}$$

Il *bias* associato all'i-esimo neurone dello strato k viene aggiornato nel seguente modo:

$$b_{i}^{(k)} = b_{i}^{(k)} - \epsilon \frac{\partial C}{\partial b_{i}^{(k)}} \tag{3.11}$$

La costante di proporzionalità, detta velocità o tasso di apprendimento (*learning rate*) e indicata con ϵ, determina l'influenza del gradiente sulla correzione del parametro considerato. La scelta di questo parametro è cruciale per ottenere buoni risultati durante l'apprendimento. Un valore troppo alto potrebbe causare l'oscillazione dell'algoritmo intorno ad un punto di minimo, senza raggiungere mai la convergenza, mentre un valore troppo basso renderebbe la ricerca della soluzione molto lenta con il rischio che si blocchi in un punto di minimo locale.

Le oscillazioni della soluzione intorno al punto di minimo possono essere smorzate applicando la tecnica *momentum*, che calcola i coefficienti correttivi tenendo conto anche dei valori ottenuti nell'iterazione precedente. Sia t una generica iterazione dell'algoritmo di apprendimento e sia $\gamma \in [0, 1]$. L'equazione (3.10) viene modificata come segue:

$$\begin{aligned} \Delta W_{ij}^{(k)}(t) &= \gamma \Delta W_{ij}^{(k)}(t-1) + \epsilon \frac{\partial C}{\partial W_{ij}^{(k)}} \\ W_{ij}^{(k)} &= W_{ij}^{(k)} - \Delta W_{ij}^{(k)}(t) \end{aligned} \tag{3.12}$$

[12] L'utilizzo di memorie RAM di capacità ridotta consente di tenere in memoria solamente lotti di esempi di piccole dimensioni. Il problema risulta particolarmente grave quando i dati di esempio sono rappresentati da immagini a colori ad alta risoluzione.

Sia P il punto che individua la soluzione dopo una generica iterazione. Quando l'algoritmo prosegue la ricerca della soluzione attraverso una "gola"[13], la componente del gradiente diretta verso una parete inverte il suo segno tra due iterazioni successive, facendo oscillare P da una parte all'altra del burrone e rallentando così la ricerca della soluzione ottimale. Usando la tecnica *momentum* il termine correttivo associato alla direzione delle pareti tende ad annullarsi, mentre gli altri tendono ad aumentare velocizzando la ricerca. In alcune situazioni vengono utilizzate tecniche più sofisticate come, *AdaGrad*, *AdaDelta* e *Adam*, che utilizzano velocità di apprendimento diverse per ogni parametro. [29, 14]

3.2.5 Calcolo del gradiente

L'algoritmo di *backpropagation* richiede di calcolare il gradiente di una funzione composta, la funzione di perdita. Il gradiente può essere quindi calcolato applicando la regola della catena sui vari strati a cascata.

Per semplicità si consideri una rete neurale con un solo neurone nello strato di output in cui il gradiente viene ricalcolato dopo ogni esempio[14]. Sia $y \in \mathbb{R}$ l'uscita predetta dalla rete e sia $y_d \in \mathbb{R}$ la corrispondente uscita desiderata.

Applicando la regola della catena, la derivata della funzione di perdita rispetto a un generico peso $W_{ij}^{(k)}$ viene scomposta come prodotto di due termini:

$$\frac{\partial C}{\partial W_{ij}^{(k)}} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial W_{ij}^{(k)}} \tag{3.13}$$

Si consideri il primo termine del prodotto, $\frac{\partial C}{\partial y}$. Utilizzando come misura dell'errore di predizione il quadrato della distanza euclidea[15] tra y e y_d, si ottiene:

$$C = \frac{1}{2}||y - y_d||^2 \tag{3.14}$$

$$\frac{\partial C}{\partial y} = y - y_d \tag{3.15}$$

$$\frac{\partial C}{\partial W_{ij}^{(k)}} = (y - y_d)\frac{\partial y}{\partial W_{ij}^{(k)}} \tag{3.16}$$

Si consideri ora il secondo termine del prodotto, $\frac{\partial y}{\partial W_{ij}^{(k)}}$:

$$\frac{\partial y}{\partial W_{ij}^{(k)}} = \frac{\partial y}{\partial z^{(N)}}\frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}} \tag{3.17}$$

[13] Il termine "gola" è riferito all'analogia presentata nel secondo capitolo nell'ambito della discesa del gradiente *batch*.

[14] Si usa un' ottimizzazione basata sulla discesa stocastica del gradiente standard

[15] Viene aggiunto il termine correttivo $\frac{1}{2}$ per rendere più efficiente l'operazione di derivazione. In questo caso, essendo y e y_d due scalari, la distanza euclidea è uguale al modulo e la derivata della funzione di costo è pari alla loro differenza.

Il termine $\frac{\partial y}{\partial z^{(N)}}$ rappresenta la variazione dell'uscita della rete al variare dello stato di attivazione del neurone dello strato di output. Scegliendo come funzione di attivazione la sigmoide logistica[16], si ha che:

$$\frac{\partial y}{\partial W_{ij}^{(k)}} = \frac{\partial y}{\partial z^{(N)}} \frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}} = y(1-y) \frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}} \tag{3.18}$$

Lo strato $(N-1)$-esimo della rete è uno strato nascosto e, a differenza dello strato di output, contiene più di un neurone. Durante la derivazione bisogna tener conto del contributo di ogni neurone dello strato. Sia $n^{(k)} \in \mathbb{N}$ il numero di neuroni contenuti in un generico strato k. Il termine $\frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}}$ è dato da:

$$\frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}} = \sum_{l}^{n^{(N-1)}} \frac{\partial z^{(N)}}{\partial x_l^{(N-1)}} \frac{\partial x_l^{(N-1)}}{\partial W_{ij}^{(k)}} \tag{3.19}$$

Derivando $z^{(N)}$ rispetto ad uno degli ingressi del neurone si ottiene il peso della connessione ad esso associata. Infatti, $z^{(N)}$ è una combinazione lineare degli ingressi. I termini che non contengono l'ingresso $x_l^{(N-1)}$ hanno derivata nulla, mentre il termine che contiene l'ingresso ha come derivata il peso. Si ha quindi che:

$$\frac{\partial z^{(N)}}{\partial W_{ij}^{(k)}} = \sum_{l}^{n^{(N-1)}} \frac{\partial z^{(N)}}{\partial x_l^{(N-1)}} \frac{\partial x_l^{(N-1)}}{\partial W_{ij}^{(k)}} = \sum_{l}^{n^{(N-1)}} W_{1l}^{(N)} \frac{\partial x_l^{(N-1)}}{\partial W_{ij}^{(k)}} \tag{3.20}$$

$$\frac{\partial z^{(N)}}{\partial x_l^{(N-1)}} = W_{1l}^{(N)} \tag{3.21}$$

Questo procedimento può essere applicato per ogni strato a cascata fino a raggiungere il peso considerato. Si definisce il termine $\delta_j^{(k)}$ come segue:

$$\delta_j^{(k)} = \begin{cases} (y - y_d)y(1-y) & se = N \\ (\sum_l^{n^{(k+1)}} W_{1l}^{(k+1)}) x_l^{(k)}(1 - x_l^{(k)}) & altrimenti \end{cases} \tag{3.22}$$

Allora si ottiene:

$$\frac{\partial C}{\partial W_{ij}^{(k)}} = \delta_j^{(k)} x_i^{(k-1)} \tag{3.23}$$

L'aggiornamento del peso $W_{ij}^{(k)}$ utilizzando l'equazione (3.24) viene eseguito come segue:

$$\begin{aligned} \Delta W_{ij}^{(k)}(t) &= \gamma \Delta W_{ij}^{(k)}(t-1) + \epsilon \delta_j^{(k)} x_i^{(k-1)} \\ W_{ij}^{(k)} &= W_{ij}^{(k)} - \Delta W_{ij}^{(k)}(t) \end{aligned} \tag{3.24}$$

[16] L'utilizzo della sigmoide logistica come funzione di attivazione ha il vantaggio di rendere particolarmente efficiente l'operazione di derivazione.

La legge può essere facilmente estesa al caso della discesa stocastica *mini-batch*, usando come funzione di perdita la media degli errori calcolati per ogni esempio. Per un lotto contenente b esempi si ha:

$$C = \frac{1}{2n} \sum_{i=1}^{b} ||y(i) - y_d(i)||^2 \tag{3.25}$$

[29]

3.3 Reti neurali convoluzionali

3.3.1 Convoluzione

In *computer vision* l'estrazione delle caratteristiche da un'immagine viene eseguita applicando filtri opportunamente progettati alle immagini, attraverso un'operazione chiamata convoluzione.

La convoluzione è un'operazione tra due funzioni che dipendono dalla stessa variabile, che consiste nell'integrare il prodotto tra la prima funzione e la seconda traslata di un certo valore, detto ritardo. Siano $f(t)$ e $g(t)$ due funzioni definite in $\mathbb{R}$ e integrabili secondo Lebesgue. La convoluzione tra f e g viene definita come segue:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{+\infty} f(t-\tau)g(\tau)d\tau \tag{3.26}$$

La formula (3.26) viene impiegata nell'elaborazione dei segnali per applicare filtri lineari[17] a segnali analogici.

Nell'elaborazione digitale dei segnali la convoluzione viene implementata in forma discreta. I filtri digitali consentono di eseguire elaborazioni molto più complesse e sofisticate rispetto a quelli analogici. Siano f e g due funzioni definite nell'insieme dei numeri interi $\mathbb{Z}$. La convoluzione discreta tra f e g viene definita come segue:

$$(f * g)(k) = \sum_{i=-\infty}^{+\infty} f(i)g(k-i) = \sum_{i=-\infty}^{+\infty} f(k-i)g(i) \tag{3.27}$$

Assumendo che le due funzioni siano uguali a zero in tutto $\mathbb{Z}$ tranne che in un numero finito di punti esse possono essere rappresentate da due vettori, $F \in \mathbb{R}^n$ e $G \in \mathbb{R}^m$. Si consideri il caso in cui $n \geq m$. Sia $k \in [(m-1), (n-1)]$. La formula (3.27) diventa:

$$(F * G)(k) = \sum_{i=k-m+1}^{k} F_i \; G_{k-i} \tag{3.28}$$

Il segnale filtrato viene ricavato eseguendo la convoluzione per valori successivi di k.

[17] L'uscita di un sistema lineare tempo invariante stimolato da un segnale è data dalla convoluzione del segnale con la risposta impulsiva del sistema.

La formula (3.28) può essere estesa a funzioni di più variabili, come le immagini, ottenendo una convoluzione multidimensionale. Sia $Im \in \mathbb{R}^{n \times o}$ un'immagine in scala di grigi e sia $Ker \in \mathbb{R}^{m \times p}$ il filtro che si vuole applicare. Il filtro, anche detto *kernel*, ha sempre dimensione uguali o inferiori a quelle dell'immagine, quindi $n \geq o$ e $m \geq p$. Sia $k \in [(m-1), (n-1)]$ e sia $l \in [(p-1), (o-1)]$. La convoluzione tra l'immagine e il filtro è data da:

$$(Im * Ker)(k, l) = \sum_{i=k-m+1}^{k} \sum_{j=l-p+1}^{l} Im_{i,j} \; Ker_{(k-i),(l-j)} \tag{3.29}$$

La convoluzione prevede l'inversione di una delle due funzioni rispetto alle due variabili indipendenti[18], quindi una delle matrici viene trasposta prima di eseguire il prodotto tra le componenti.

3.3.2 Mappa di caratteristiche

Anche le reti neurali utilizzano la convoluzione per estrarre le caratteristiche da un'immagine, impiegando filtri imparati durante la fase di addestramento. A differenza delle tecniche tradizionali, i filtri usati dalle tecniche di apprendimento profondo non sono stabiliti a priori dai progettisti, ma vengono generati a seconda del tipo di oggetto da riconoscere.

Il principale difetto è rappresentato dall'estrazione manuale di caratteristiche dalle immagini. In questo caso, si parla di caratteristiche artigianali o fatte a mano (*hand-crafted*). Infatti, a seconda del tipo di oggetto che deve essere riconosciuto è necessario scegliere quali siano le caratteristiche più importanti da estrarre dall'immagine. La scelta delle caratteristiche non è quindi univoca.

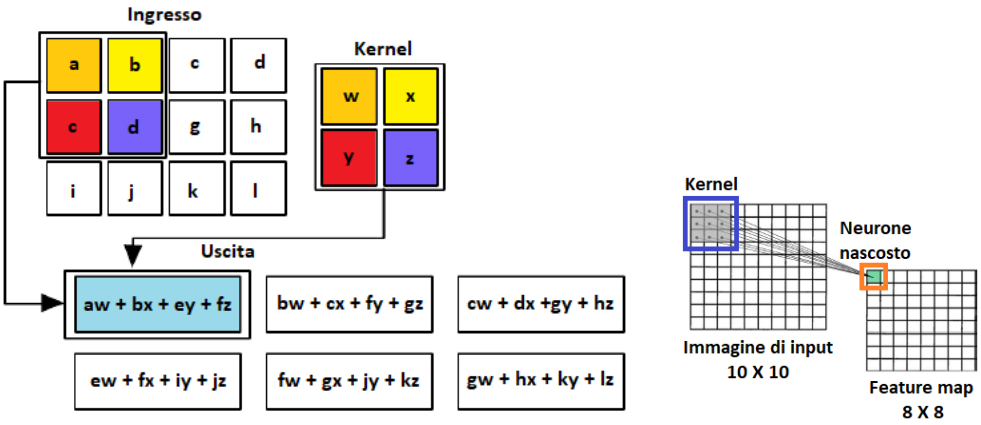
Gli algoritmi all'avanguardia nel rilevamento di oggetti sono ormai tutti basati su reti neurali, poiché hanno il vantaggio di estrarre caratteristiche in maniera automatica. Una rete neurale impara a riconoscere gli attributi di un determinato oggetto direttamente dai dati durante l'addestramento. Se nell'approccio tradizionale il problema veniva risolto solo parzialmente utilizzando tecniche di *soft computing*, in questo secondo approccio tutto il lavoro è basato sul *soft computing*. I modelli prodotti dalle reti neurali risultano inoltre maggiormente più accurati e robusti.

Una rete neurale impara inizialmente caratteristiche di basso livello linee, contorni, circonferenze, angoli, creste e blobs, per poi combinarle in modo gerarchico generando caratteristiche di livello più alto come trame e forme, fino a costituire gli oggetti veri e propri.

Gli strati che eseguono l'estrazione di caratteristiche vengono detti *convolutional layer* (strato convoluzionale), poiché eseguono la convoluzione tra un'immagine in scala di grigi e una matrice di pesi. L'uscita dello strato è rappresentata da una matrice, detta *feature map* (mappa delle caratteristiche), contenete informazioni su alcune caratteristiche dell'immagine.

[18] Per la proprietà commutativa è indifferente quale delle due funzioni venga invertita. La funzione ottenuta applicando la finestra scorrevole senza capovolgere il *kernel* viene detta cross-correlazione.

Quando l'input della rete è un'immagine a colori, le caratteristiche vengono estratte eseguendo una convoluzione tridimensionale tra due tensori. In questo caso i neuroni di uno strato convoluzionale sono organizzati in modo da formare un volume corrispondente a una piccola regione del tensore di ingresso, detta campo recettivo.



(a) Convoluzione tra una matrice 3×3 e un *kernel* 2×2. [10]

(b) *Features map*. [14]

Figura 3.8: Esempio di estrazione di caratteristiche mediante convoluzione in una rete neurale.

Solitamente il *kernel* viene scelto in modo da avere la stessa profondità del volume di input, ma dimensioni inferiori, in modo da ridurre la risoluzione spaziale dei dati. La convoluzione di un'immagine o di una *feature map* con un filtro di piccole dimensioni equivale al far scorrere sul volume di ingresso una finestra in modo da elaborare i dati in piccoli gruppi. Gli strati convoluzionali risultano molto più efficienti rispetto a quelli *fully-connected*, sia dal punto di vista dell'efficienza statistica sia da quello dell'occupazione di memoria, poiché sono caratterizzati da:

- connettività sparsa (*sparse connectivity*), cioè una riduzione del numero di connessioni tra i neuroni[19];
- condivisione di parametri (*parameter sharing*), cioè l'utilizzo degli stessi pesi[20] per ogni posizione della finestra scorrevole.

Il passo con cui si fa scorrere la finestra prende il nome di *stride*. Aumentando il valore di questo parametro viene ridotta sempre di più la dimensione della *feauture map*.

In alcuni casi, per non ridurre troppo rapidamente le dimensioni delle *feature map* viene utilizzata una tecnica detta *zero padding*, che prevede di aggiungere al tensore di input uno o più bordi contenenti elementi tutti nulli aumentando l'altezza e la lunghezza del volume da esso individuato.

In uno strato *fully-connected* l'uscita viene calcolata eseguendo il prodotto tra una matrice, la matrice dei pesi, e un vettore, il vettore degli ingressi, mentre

[19] Ogni neurone di uno strato è collegato solamente con un piccolo gruppo di quelli dello strato precedente di dimensioni pari a quelle del *kernel*.

[20] Uno strato *fully-connected utilizza ogni peso solamente una volta.*

in uno strato convoluzionale viene calcolata eseguendo un'operazione simile al prodotto scalare.

La finestra scorrevole individua una regione del tensore di input, rappresentata da tre sottomatrici, una per ogni canale di colore. Ciascuna delle tre sottomatrici viene appiattita ottenendo tre vettori. La stessa operazione viene eseguita sul *kernel*, ma prima questo viene "capovolto". La trasposizione viene sempre eseguita sul *kernel*, poiché subisce meno variazioni rispetto al tensore di ingresso. Per ciascun canale di colore si esegue il prodotto scalare tra il vettore di input e il vettore associato al filtro, ottenendo uno scalare per ogni "fetta" del tensore.

Una rete neurale che contiene almeno uno strato convoluzionale viene detta rete neurale convoluzionale[21] (*Convolutional Neural Network - CNN*. Il loro funzionamento è ispirato al funzionamento della corteccia visiva animale e sono un'evoluzione del Cognitrone introdotto da Fukushima.

Una rete convoluzionale utilizza gli strati più profondi per estrarre le caratteristiche dall'immagine di ingresso, attraverso la convoluzione su *feature map* sempre più piccole, per poi eseguire la classificazione negli strati finali, impiegando strati *fully-connected*. Le *feature map* devono essere appiattite prima di essere processate da uno strato *fully-connected*.

Nei problemi di classificazione binaria solitamente lo strato di output presenta un solo neurone con una funzione di attivazione logistica, mentre nei problemi di classificazione multi-classe lo strato di output presenta k neuroni, dove k è il numero delle classi del problema, le cui uscite vengono normalizzate applicando la funzione *softmax*, definita da $\mathbb{R}^k \to z \in \mathbb{R}^k | z_i > 0, \sum_{i=1}^{k} z_i = 1$, in modo da ottenere come output della rete un vettore contenente le probabilità di appartenenza a ciascuna classe. Sia $z_i^{(N)}$ lo stato di attivazione associato all'i-esimo neurone dello strato di output. L'uscita del neurone $x_i^{(N)}$ viene calcolata applicando a $z_i^{(N)}$ la funzione *softmax*:

$$x_i^{(N)} = softmax(z_i^{(N)}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_k}} \tag{3.30}$$

Nelle reti neurali che impiegano il metodo della discesa del gradiente *mini-batch* l'ingresso di ogni strato è rappresentato da un tensore quadridimensionale, contenente i dati relativi ad un intero lotto di esempi, di dimensioni $b \times d \times h \times w$, dove b è il *batch size*, d la profondità del colore, h l'altezza e w la lunghezza. [10]

3.3.3 Pooling

Nelle reti convoluzionali spesso alle *feature map* generate *convolution layer* viene applicata una funzione di *pooling*, in modo tale da garantire invarianza a piccole traslazioni nell'immagine di input. In questo modo viene generata la stessa mappa di caratteristiche quando vengono analizzate due immagini contenenti uno stesso oggetto in posizioni leggermente differenti.

L'invarianza permette di determinare se un'immagine contiene una determinata caratteristica senza sapere precisamente in quale punto si trovi. La rete in

[21] Le reti neurali convoluzionali sono una particolare architettura di rete *feed-forward*.

questo modo impara a diventare invariante ad alcune trasformazioni. Inoltre, la capacità degli strati di *pooling* di generalizzare permette di ridurre le probabilità di *overfitting* durante l'addestramento della rete.

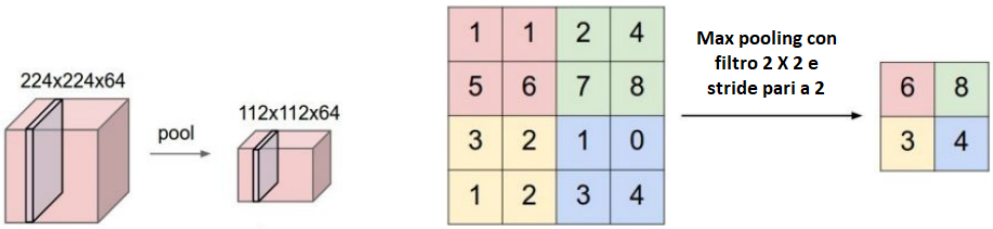
L'operazione viene eseguita applicando una finestra scorrevole in diverse posizioni della *feature map* associando ad ognuna un numero reale, come avviene nella convoluzione, ma anziché eseguire il prodotto scalare con un filtro viene calcolata una statistica sommaria che riepiloga e generalizza le caratteristiche contenute in quel riquadro. Uno strato di *pooling* quindi non contiene nessun peso, quindi il suo impiego non rallenta la fase di addestramento.

Le funzioni di *pooling* più utilizzate sono:

- *max pooling*, che restituisce come uscita il massimo valore tra quelli contenuti in una finestra dell'input;
- *average pooling*, che restituisce come uscita la media pesata, solitamente in base alla distanza dall'elemento centrale, dei valori contenuti in una finestra dell'input.

L'impiego degli strati di *pooling*, insieme a quello degli strati convoluzionali, consente di gestire immagini di input con dimensioni variabili, ridimensionando la *feauture map* in ingresso al primo strato *fully-connected*[22], in modo che essa abbia sempre la stessa dimensione.

Si fa in modo che uno strato *fully-connected* riceva in ingresso sempre lo stesso numero di statistiche riassuntive, variando le dimensioni e il passo della finestra di *pooling*. [10]



(a) Riduzione della risoluzione spaiale tramite *pooling*.

(b) *Max pooling*.

Figura 3.9: Esempio di *pooling layer*. [14]

3.3.4 Apprendimento residuale

La profondità di una rete convoluzionale risulta essere un fattore cruciale per ottenere sofisticati modelli di *object recognition*, soprattutto quando vengono utilizzati dataset estremamente vasti e diversificati come **COCO** e **ImageNet**, poiché l'aggiunta di *convolutional layer* permette di aumentare i livelli di astrazione utilizzati per descrivere le caratteristiche di un'immagine.

Le architetture tradizionali, come **AlexNet**, **VGG** o **GoogLeNet**, non permettono di creare reti troppo profonde. In particolare, è stato dimostrato che

[22] Uno strato *fully-connected* deve avere un input di dimensione fissa

l'errore commesso da una rete convoluzionale durante la fase di addestramento inizia ad aumentare, anziché diminuire, quando ne viene aumentata troppo la profondità[23].

Si consideri una rete neurale A con N strati. Una rete B di $N+1$ strati dovrebbe essere in grado, almeno dal punto di vista teorico, di generare un modello con le stesse identiche prestazioni di A. Sarebbe sufficiente, infatti, che la rete B apprenda gli stessi pesi della rete A nei primi N strati per poi eseguire una mappatura di identità nell'ultimo strato. Tuttavia all'aumentare di N risulta sempre più difficile per una rete imparare ad applicare la funzione di identità.

Il degrado delle prestazioni dei modelli con l'aumento della profondità della rete viene ricondotto principalmente al problema della scomparsa del gradiente (*vanish gradient*).

L'algoritmo di *backpropagation* utilizza la regola della catena per calcolare la derivata della funzione di costo rispetto a un generico peso $W_{ij}^{(k)}$, trasformando l'operazione di derivazione in un prodotto tra n termini, dove n è la profondità della rete. L'utilizzo di funzioni di attivazione non lineari a valori in $[0, 1]$, come la sigmoide logistica e la tangente iperbolica, fa tendere a zero il valore della derivata per piccoli valori di k, poiché il prodotto di n termini con modulo minore di 1 tende a zero all'aumentare di n. In particolare, si dice che il gradiente "scompare" o "fugge".

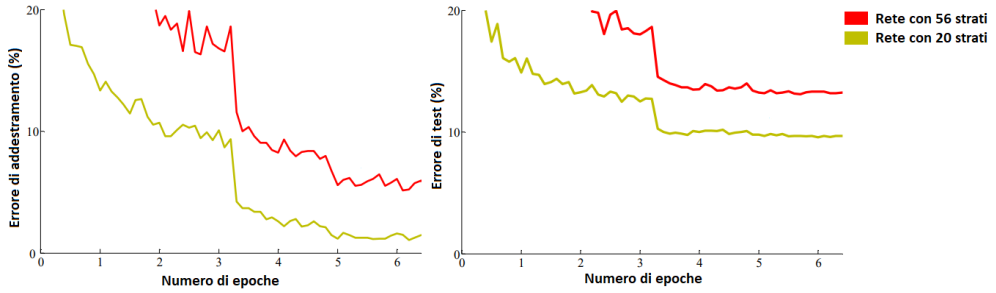


Figura 3.10: Andamento dell'errore durante la fase di addestramento (a sinistra) e durante la fase di test (a destra) calcolato su una rete di 20 strati e su una rete di 56 strati. Le reti addestrate su **ImageNet** più performanti sono quelle che presentano tra i 16 e 30 strati. [30]

In una rete molto profonda, quindi, i termini correttivi utilizzati nell'aggiornamento dei pesi degli strati più vicini all'ingresso risultano molto più piccoli rispetto a quelli degli strati finali, rendendo l'addestramento degli strati più profondi estremamente più lento. Poiché i termini correttivi sono più piccoli è necessario un maggior numero di iterazioni prima di trovare il valore dei pesi che minimizzi la funzione di perdita. L'addestramento delle reti neurali su *graphics processing unit* e l'utilizzo di funzioni di attivazione di tipo $ReLU$ ha permesso di far convergere l'algoritmo di *backpropagation* in tempi non proibitivi, mitigando il problema senza però fornire una soluzione definitiva. [31]

[23] La profondità massima raggiungibile con i modelli tradizionali di solito è intorno ai 50 strati.

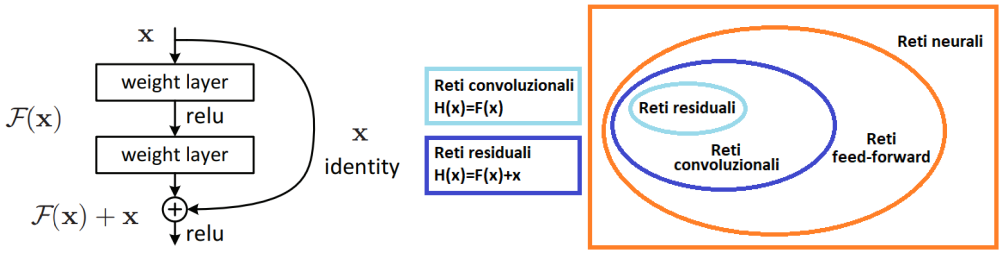
La soluzione al problema è fornita dall'apprendimento residuale, che prevede di aggiungere delle scorciatoie in modo da connettere insieme neuroni appartenenti a due strati non consecutivi.

Nel 2015 viene introdotta la prima architettura di rete neurale basata sul concetto di apprendimento residuale, detta rete neurale residuale (*ResNet*), che permise di raggiungere profondità superiori ai 100 strati, il doppio di quanto possibile coi modelli precedenti.

La peculiarità di questa rete è l'utilizzo dei blocchi residuali (*residual block*), costituiti da due strati convoluzionali separati da uno strato *ReLU*. L'uscita del blocco residuale $H(x)$ viene ottenuta sommando l'input del primo strato convoluzionale x all'uscita del secondo strato convoluzionale $F(x)$, detto residuo.

L'utilizzo dei blocchi residuali permette alla rete di imparare ad eseguire mappature di identità e di preservare l'informazione andando in profondità nella rete, riducendo il problema della scomparse del gradiente.

La convoluzione solitamente riduce la risoluzione spaziale di un'immagine, quindi il tensore di output risulta di dimensioni diverse rispetto a quello di input. Questo problema viene risolto utilizzando la tecnica dello *zero padding* o eseguendo convoluzioni con maschere 1×1. [30]



(a) Struttura di un blocco residuale. [30] (b) Classe di appartenenza delle reti residuali.

Figura 3.11: Reti neurali residuali. [30]

3.3.5 Normalizzazione dei lotti

L'input di uno strato nascosto spesso presenta una distribuzione di probabilità con varianza elevata a causa dei valori di inizializzazione dei pesi e dalla casualità del *train set*. Ad ogni aggiornamento dei pesi l'uscita di uno strato varia innescando una reazione a catena che modifica la distribuzione degli input degli strati successivi. Questo fenomeno, che prende il nome di spostamento della covariata interna (*internal covariate shift*), costringe i neuroni a riadattarsi costantemente a diverse distribuzioni degli ingressi e risulta particolarmente grave nelle reti profonde.

Nelle reti neurali residuali il problema viene risolto applicando la tecnica della normalizzazione dei lotti *batch normalization*, che consiste nel normalizzare i valori contenuti in un *mini-batch* centrando e ridimensionando la distribuzione. Oltre a ridurre lo spostamento della covariata interna questa tecnica mitiga il fenomeno dell'*overfitting*, velocizza l'addestramento e consente di inizializzare i pesi con meno cautela.

Sia $x^{(k)}$ l'uscita di uno strato convoluzionale k, rappresentata da un tensore quadridimensionale di lunghezza w e altezza h contenente le *feature map* relative a un lotto di b esempi. La media $\mu^{(k)} \in \mathbb{R}^{3\times h\times w}$ e la varianza $\sigma^{2(k)} \in \mathbb{R}^{3\times h\times w}$ sul *mini-batch* vengono calcolate con le seguenti formule:

$$\mu^{(k)}_{l,m,n} = \sum_{i=1}^{b} x^{(k)}_{i,l,m,n} \quad dove\, l \in [1,3], m \in [1,h], n \in [1,w] \tag{3.31}$$

$$\sigma^{2(k)}_{l,m,n} = \sum_{i=1}^{b} (x^{(k)}_{i,l,m,n} - \mu^{(k)}_{l,m,n})^2 \quad dove\, l \in [1,3], m \in [1,h], n \in [1,w] \tag{3.32}$$

L'uscita $y^{(k)}$ generata applicando la *batch normalization* è data da:

$$\hat{x}^{(k)}_{i,l,m,n} = \frac{x^{(k)}_{i,l,m,n} - \mu^{(k)}_{l,m,n}}{\sqrt{\sigma^{2(k)}_{l,m,n} + \epsilon}} \quad dove\, i \in [1,b], l \in [1,3], m \in [1,h], n \in [1,w] \tag{3.33}$$

$$y^{(k)}_{i,l,m,n} = \gamma^{(k)}_{l,m,n}\, \hat{x}^{(k)}_{i,l,m,n} + \beta^{(k)}_{l,m,n} \quad dove\, i \in [1,b], l \in [1,3], m \in [1,h], n \in [1,w] \tag{3.34}$$

dove ϵ é una piccola costante arbitraria, mentre $\gamma^{(k)} \in \mathbb{R}$ e $\beta^{(k)} \in \mathbb{R}$ sono due parametri che vengono imparati durante l'addestramento. Il termine $y^{(k)}$ viene utilizzato come ingresso per lo strato successivo della rete, mentre il termine $\hat{x}^{(k)}$ viene salvato nello strato corrente. [32]

Capitolo 4

Approccio basato sul deep learning

Le reti neurali convoluzionali sono estremamente abili nella classificazione di immagini e non risultano affette dal problema della scelta dei filtri per l'estrazione delle caratteristiche, poiché questi vengono imparati in maniera automatica durante la fase di addestramento.

I moderni algoritmi di *object detection* basati sull'apprendimento profondo seguono la stessa logica degli algoritmi tradizionali, eseguendo classificazioni e regressioni per rilevare gli oggetti, ma inglobano tutte le operazioni in un unico modello rappresentato da una rete neurale convoluzionale.

Le reti neurali per il rilevamento di oggetto sono più sofisticate rispetto a quelle utilizzate per la classificazione di immagini. La maggior parte dei modelli è costituita da due blocchi, una *backbone* (spina dorsale), che si occupa dell'estrazione di caratteristiche, e un rilevatore, che predice i *bounding box*. Come *backbone* vengono spesso utilizzati modelli di reti convoluzionali per la classificazione di immagini cui vengono eliminati gli strati finali mantenendo solamente gli strati che si occupano di estrarre le caratteristiche.

4.1 Reti neurali basate su regioni

4.1.1 Ricerca selettiva

Nel novembre 2013 Ross Girshick propose un'architettura di rete neurale specializzata nel riconoscimento di oggetti, le reti neurali convoluzionali basate su regioni (*Region Based Convolutional Neural Network - CNN*).

Gli algoritmi a finestra scorrevole, nonostante siano un metodo di forza bruta, risultano abbastanza efficienti quando vengono impiegati modelli lineari, come le macchine a vettori di supporto, consentendo di risolvere anche problemi in tempo reale. Al contrario, l'esecuzione ad ogni posizione della finestra scorrevole di un modello non lineare particolarmente complesso, come una rete convoluzionale, risulta molto dispendiosa in termini di tempo e difficilmente praticabile.

Nelle reti basate su regioni questo problema viene attenuato applicando alle immagini di addestramento un algoritmo per la proposta di regioni, detto ricerca selettiva, che ha il compito di ridurre considerevolmente la mole delle regioni da processare, in modo da rendere più veloce sia il rilevamento che l'addestramento.

L'obiettivo della ricerca selettiva è quello di individuare all'interno di un'immagine le regioni che hanno una maggiore probabilità di contenere oggetti, dette regioni di interesse (*region of interest* - ROI), scartando quelle che costituiscono lo sfondo.

La semplice segmentazione dell'immagine produce un numero troppo elevato di regioni, generando molti segmenti per uno stesso oggetto[1].

La ricerca selettiva risolve questo problema generando proposte di regioni in maniera gerarchica. L'algoritmo inizialmente esegue una segmentazione[2] dell'immagine in modo tale da ottenere una lista di gruppi di *pixel* contigui.

Ad ogni segmento vengono associati due vettori di caratteristiche:

- un **descrittore dei colori**, rappresentato da un istogramma di $m = 75$ colonne, 25 per ogni canale di colore;
- un **descrittore delle trame**, basato su 8 diverse direzioni del gradiente dell'intensità luminosa, rappresentato da un istogramma di $n = 240$ colonne, 80 per ogni canale di colore[3].

I segmenti ottenuti vengono iterativamente combinati insieme in base alla loro somiglianza in modo tale da generare regioni sempre più grandi e complesse.

Siano i e j due regioni, sia c_k il valore della k-esima colonna del descrittore dei colori, sia t_k il valore della k-esima colonna del descrittore delle trame. Siano $A^{(i)}$ e $A^{(j)}$ le aree delle due regioni, sia $A^{(ij)}$ l'area del più piccolo riquadro contenente le due regioni e sia A l'area dell'immagine[4]. La somiglianza tra le due regioni viene definita come combinazione lineare delle quattro somiglianze:

$$S(i,j) = \alpha(\sum_{k=1}^{m} min(c_k^{(i)}, c_k^{(j)})) + \beta(\sum_{k=1}^{n} min(t_k^{(i)}, t_k^{(j)})) + \\ + \gamma(1 - \frac{A^{(i)} + A^{(j)}}{A}) + \delta(1 - \frac{A^{(ij)} - A^{(i)} - A^{(j)}}{A}) \quad \alpha, \beta, \gamma, \delta \in \mathbb{R} \quad (4.1)$$

I primi due termini tengono conto rispettivamente della somiglianza di colore e di trama tra le due regioni. Il terzo termine misura la somiglianza delle dimensioni favorendo l'unione di regioni delle stesse dimensioni, evitando che le regioni più grandi inghiottiscano quelle più piccole. L'ultimo termine misura la compatibilità delle forme favorendo l'unione di regioni che si adattano bene insieme, in modo tale da riempire i buchi.

[1] Questo potrebbe accadere nel caso di oggetti particolarmente complessi e ricchi di dettagli o nel caso di oggetti parzialmente coperti e nascosti.

[2] Solitamente si utilizzano tecniche di segmentazione basate su grafi. Un'alternativa è rappresentata dall tecniche di *clustering* descritte nel secondo capitolo.

[3] Si hanno 10 colonne per ciascuna direzione del gradiente.

[4] Le aree vengono misurate come numero di *pixel*.

Ad ogni iterazione l'algoritmo calcola la somiglianza per ogni coppia di segmenti adiacenti aggiungendo il valore all'insieme S, inizialmente vuoto, tale che:

$$S = \{(i, j, S(i, j))|\ i\ e\ j\ siano\ vicine\}$$

Viene selezionata la terna $(i, j, S(i, j))$ caratterizzata dalla somiglianza più alta e le regioni associate i e j vengono fuse insieme in un'unica regione k. L'algoritmo elimina dall'insieme tutte le terne contenenti i o j e calcola la somiglianza tra k e le regioni vicine aggiungendo il valore all'insieme S. La regione k viene aggiunta all'insieme R delle regioni di interesse e si procede con l'iterazione successiva, fino a quando non ci sono più regioni da unire.

L'appartenenza di k all'insieme R delle regioni di interesse è condizione necessaria affinché essa contenga un oggetto, ma non è condizione sufficiente.

Gli oggetti da rilevare sono sicuramente tutti presenti all'interno delle regioni di interesse, ma esse potrebbero contenere anche dei falsi positivi, cioè delle porzioni di sfondo. Inoltre c'è il rischio di fondere insieme regioni che identificano due oggetti diversi.

La ricerca selettiva è quindi caratterizzata da un recupero molto elevato, ma questo non è un problema poiché i falsi positivi saranno scartati durante la fase di riconoscimento.

Le regioni di interesse vengono ridimensionate e date in ingresso alla *region based network*. Per ogni regione viene eseguita una diversa rete convoluzionale in modo da ottenere una serie di mappe di caratteristiche, che vengono poi classificate impiegando macchine a vettori di supporto binarie, una per ciascuna classe.

Le regioni che vengono etichettate come negativi da tutte quanti classificatori vengono scartate, mentre per le altre vengono previsti i riquadri di delimitazione relativi agli oggetti, impiegando un regressore per ogni coordinata.

La macchina a vettore di supporto, il regressore e le reti convoluzionali vengono trattati come moduli indipendenti e sono addestrati separatamente. [33, 34]

4.1.2 Fast R-CNN

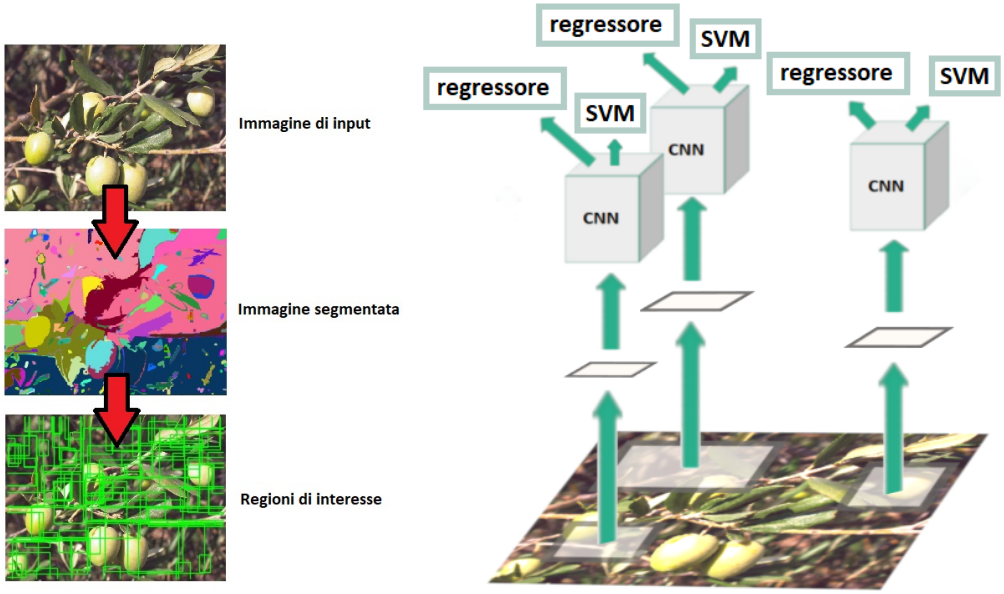
Le reti neurali basate su regioni durante l'inferenza risultano molto lente, impiegando in media circa 50 secondi per processare un'immagine, poiché viene eseguita una rete convoluzionale per ciascuna delle regioni[5] prodotte dalla ricerca selettiva.

Nell'aprile 2015, lo stesso Ross Girshick introduce una versione più veloce delle reti neurali basate su regioni, detta *Fast R-CNN* (Fast Region Based Convolutional Neural Network).

Il problema della lentezza nell'estrazione di caratteristiche viene risolto impiegando un'unica rete neurale[6] per ogni immagine, detta spina dorsale (*backbone*), anziché una per ogni regione.

[5] Solitamente la ricerca selettiva genera 2000 regioni per ogni immagine.

[6] Solitamente viene utilizzata una rete **VGG** addestrata sul dataset **ImageNet**.



(a) Funzionamento dell'algoritmo di ricerca selettiva.

(b) Funzionamento di una rete neurale basata su regioni.

Figura 4.1: Reti neurali basate su regioni. [34]

L'immagine di input viene data in ingresso alla *backbone* che ne estrae le caratteristiche e genera una mappa di caratteristiche. Si esegue l'algoritmo di ricerca selettiva sull'immagine di ingresso, in modo tale da ottenere una lista di regioni di interesse che vengono mappate all'interno della *feature map*. L'utilizzo di un'unica mappa di caratteristiche, anziché una per ogni regione di interesse, permette di condividere le caratteristiche convoluzionali riducendo l'occupazione di memoria.

Le regioni proposte vengono ridimensionate da uno strato di *pooling*, detto *Region of Interest pooling - RoI pooling*, in modo tale da avere una dimensione fissa $W \times H$. Ogni regione di interesse corrisponde ad una finestra rettangolare di dimensioni wxh all'interno della *feature map*. La finestra viene divisa in una griglia $H \times W$, le cui celle hanno dimensioni $\frac{h}{H} \times \frac{w}{W}$. Su ogni cella viene eseguito il *max pooling* ottenendo una regione delle dimensioni richieste, che viene appiattita e data in ingresso a due moduli collegati in parallelo:

- un modulo composto interamente di strati *fully-connected*, che classifica le regioni di interesse utilizzando la funzione *softmax*[7] nello strato finale;
- un modulo che per ciascuna regione di interesse predice le coordinate e le dimensioni dei riquadri di delimitazione.

I vari moduli sono integrati in un'unica rete neurale in modo da rendere la fase di addestramento più semplice e veloce.

Sia $p = (p_0, p_1, \ldots, p_{k-1}) \in \mathbb{R}^k$ l'uscita generata dallo strato *softmax*, dove p_i indica la probabilità che la regione di interesse appartenga alla classe i e k è il numero totale delle classi.

[7] Lo strato *softmax* sostituisce le macchine a vettori di supporto binarie.

Per ogni classe i di oggetto il regressore predice i riquadri di delimitazione associati all'oggetto generando un vettore $t^{(i)} = (t_x^{(i)}, t_y^{(i)}, t_h^{(i)}, t_w^{(i)}) \in \mathbb{R}^4$. Sia u la classe reale dell'oggetto e v il vettore delle coordinate del *ground truth box*. La funzione di perdita è data dalla somma di due termini, uno che esprime l'errore nella classificazione e uno che esprime l'errore nella localizzazione:

$$C(p, u, t^{(u)}, v) = -log(p_u) + \lambda[u \geq 1] \sum_{i \in x,y,w,h} smooth_{L1}(t_i^{(u)} - v_i)$$

$$dove,\ smooth_{L1}(x) = \begin{cases} 0.5x^2 & se \quad |x| < 1 \\ |x| - 0.5 & altrimenti \end{cases} \tag{4.2}$$

Il termine $[u \geq 1]$ risulta nullo quando la regione viene classificata come una porzione dello sfondo, cioè $u = 0$, e risulta pari a 1 negli altri casi. Il coefficiente λ è un parametro che controlla il bilanciamento tra le due funzioni di costo. [35]

4.1.3 Faster R-CNN

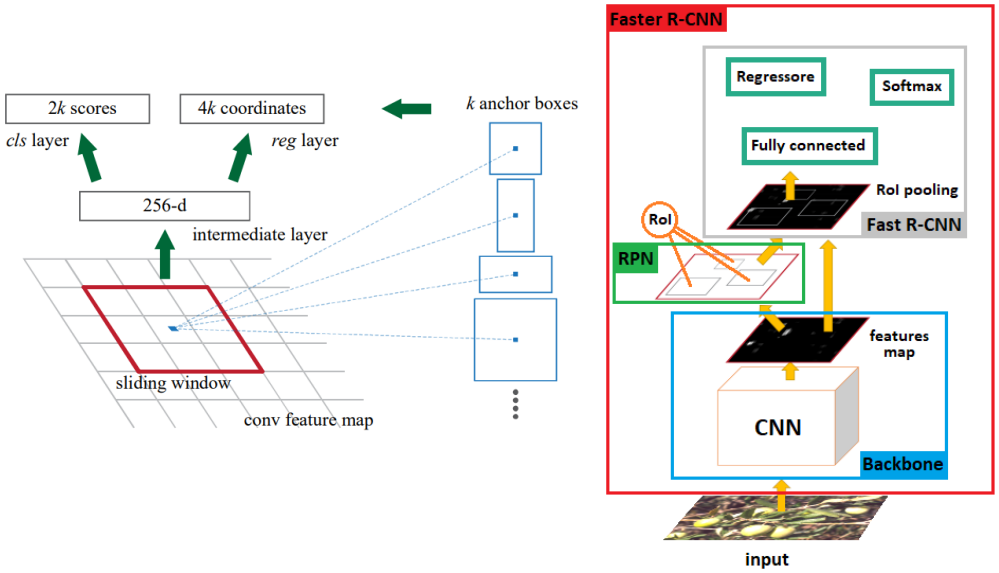
I miglioramenti introdotti con *Fast R-CNN* rendono trascurabile il tempo di esecuzione della rete di rilevamento rispetto a quello della ricerca selettiva, rendendo il metodo per la proposta di regioni un collo di bottiglia. La rete impiega in media 2,3 secondi per processare un'immagine durante l'inferenza, di cui 2 sono spesi per generare proposte di regioni.

Nel giugno 2015, viene introdotta una rete per la proposta di regioni (*Region Proposal Network - RPN* per sostituire l'algoritmo di ricerca selettiva. La rete viene fusa insieme con il rilevatore *Fast R-CNN* ottenendo un'unica rete neurale che prende il nome di *Fater R-CNN*.

La *region proposal network* è una rete neurale completamente convoluzionale in grado di imparare a riconoscere le regioni di interesse direttamente dai dati, garantendo non solo maggiore accuratezza e velocità. Anziché scandire l'immagine in diverse scale o con diverse dimensioni della finestra scorrevole, come gli algoritmi *sliding window* o la ricerca selettiva, la rete per la proposta di regioni esegue una sola scansione utilizzando una tecnica basata su *piramidi di ancore*. Questo metodo garantisce invarianza di traslazione, riducendo le dimensioni del modello e aiutando la rete a generalizzare.

In una rete una rete *Faster R-CNN* la mappa di caratteristiche generata dalla *backbone* viene data in ingresso alla rete per la proposta di regioni. Ogni elemento della *feature map*, detto ancora, viene proiettato sull'immagine originale in modo da corrispondere con il centro del campo recettivo. Ad ogni ancora vengono associati k riquadri di delimitazione, detti priori o riquadri di ancoraggio (*anchor box*), caratterizzati ognuno da diverse forme e dimensioni. Questi riquadri hanno il compito di fornire un riferimento per la scala e le proporzioni di una previsione. Il numero di priori k è un parametro che permette di variare il numero di scale e di proporzioni da impiegare per il rilevamento.

I k riquadri generati ad una posizione della finestra scorrevole vengono dati in ingresso a due strati *fully-connected* posti in parallelo:



(a) Architettura della rete per la proposta di regioni (RPN). (b) Composizione dei vari moduli di *Faster R-CNN*.

Figura 4.2: Architettura di una rete neurale Faster-RCNN. [36]

- lo strato *cls*, che esegue una classificazione binaria dei priori associando ad ognuno la probabilità di contenere un oggetto (*object categories*) e la probabilità di contenere una porzione di sfondo (*background*)[8];
- lo strato *reg*, che ha il compito di far aderire la forma e la scala dei priori all'oggetto contenuto, modificando la posizione del centro e le dimensioni del riquadro.

Lo strato *reg* utilizza un regressore diverso per ognuno dei k priori senza condividere i pesi. In questo modo ogni regressore si abitua a prevedere *bounding box* di determinate scale e proporzioni. Sia a un riquadro di ancoraggio e sia b il bounding box predetto. Siano x e y rispettivamente l'ascissa e l'ordinata del centro di un riquadro di dimensioni $w \times h$. Il regressore associato al riquadro a predice il seguente vettore di coordinate:

$$t^{(b)} = \begin{pmatrix} t_x^{(b)} \\ t_y^{(b)} \\ t_w^{(b)} \\ t_h^{(b)} \end{pmatrix} = \begin{pmatrix} \frac{x_b - x_a}{w_a} \\ \frac{y_b - y_a}{h_a} \\ log(\frac{w}{w_a}) \\ log(\frac{h}{h_a}) \end{pmatrix} \tag{4.3}$$

[36]

La rete per la proposta di regioni viene addestrata minimizzando una funzione di perdita basata sul confronto tra i riquadri di ancoraggio e i *ground truth*

[8]Non viene prevista la probabilità di appartenenza ad una data classe, ma solo la probabilità che il riquadro contenga un oggetto.

box. Sia G l'insieme degli oggetti contenuti nell'immagine di addestramento e sia A l'insieme dei riquadri di ancoraggio. Ad ogni riquadro di ancoraggio $y \in A$ viene associata un'etichetta, che può essere positiva (1) o negativa (0), secondo il seguente criterio:

$$label(y) = \begin{cases} 1 \quad se \quad \forall x \in (G), \forall z \in (A) | \, IoU(y,x) > IoU(z,x) \\ 1 \quad se \quad \exists x \in (G) | \, IoU(y,x) > 0.7 \\ 0 \quad se \quad \nexists x \in (G) | \, IoU(y,x) < 0.3 \end{cases} \tag{4.4}$$

Ad ogni riquadro di ancoraggio viene quindi associato il *ground truth box* che si sovrappone meglio, ma non viene stabilita una corrispondenza biunivoca. I riquadri cui non viene associata nessuna classe non vengono utilizzati nel calcolo della funzione di perdita.

Sia N_{cls} il numero di ancoraggi contenuti in un lotto e sia N_{reg} il numero dei priori totali. Sia p la probabilità che i contenga un oggetto, sia p^* il valore dell'etichetta, sia t il vettore di coordinate previsto dal regressore e sia t^* il vettore di coordinate associato al *ground truth box*[9]. La funzione di perdita è data dalla somma di due contributi, uno che rappresenta l'errore nella classificazione e uno che rappresenta l'errore di localizzazione per ogni riquadro di ancoraggio generato:

$$\begin{aligned} C(p_i, t_i) &= \frac{1}{N_{cls}} \sum_i C_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{reg}} \sum_i p_i^* C_{reg}(t_i, t_i^*) \\ dove, \quad & C_{cls}(p_i, p_i^*) = -p_i^* log(p_i) - (1 - p_i^*) log(1 - p_i) \\ & C_{reg}(t_i, t_i^*) = \textstyle\sum_{k \in x,y,w,h} smooth_{L1}(t_{ik}, t_{ik}^*) \end{aligned} \tag{4.5}$$

Il secondo termine è nullo quando la classe del riquadro considerato è negativa. Il termine $\in \mathbb{R}$ è un parametro di bilanciamento.

La funzione di perdita risulta maggiormente influenzata dai riquadri con classe negativa, poiché è più probabile che l'ancoraggio sia associato a una porzione dello sfondo. Questo problema viene risolto estraendo in modo casuale $N_{cls} = 256$ campioni dall'insieme degli *anchor box*, in modo tale da avere la stessa percentuale di riquadri positivi e negativi. Se nell'immagine non ci sono abbastanza campioni positivi il lotto viene riempito con quelli negativi. La funzione di perdita viene poi calcolata solamente utilizzando i campioni estratti.

La rete per la proposta delle regione possono essere addestrate sia in maniera indipendente che in maniera congiunta. Nel primo caso viene prima addestrata la rete di proposta, poi le regioni di interesse generate vengono utilizzate per addestrare *Faster R-CNN*. La rete di rilevamento viene poi utilizzata per addestrare nuovamente la rete di proposta. Questo processo viene ripetuto più volte e prende il nome di addestramento alternato.

[9]Viene utilizzata la stessa parametrizzazione delle coordinate previste dal regressore.

Nel secondo caso le due reti vengono trattate come un'unica rete, utilizzando le regioni proposte per addestrare la rete di rilevamento. L'algoritmo di *back-propagation* viene eseguito in maniera tradizionale, combinando le due funzioni di perdita negli strati condivisi. Questa tecnica prende il nome di addestramento congiunto. Spesso durante la retropropagazione dell'errore viene trascurata la derivata della funzione di perdita rispetto alle coordinate delle regioni di interesse. In questo caso si parla di addestramento congiunto approssimativo. [36]

4.2 Reti neurali a passata singola

4.2.1 You Only Look Once

Nel 2016 Joseph Redmon presenta *You Only Look Once*(*YOLO*), la prima rete a passata singola. A differenza delle reti basate su regioni, che analizzano l'immagine di input due volte, una per proporre le regioni e una per eseguire il rilevamento vero e proprio, una rete a passata singola stima le coordinate dei riquadri e i livelli di confidenza in un'unica fase. Questa tecnica presenta innumerevoli vantaggi:

- un'elevata velocità di esecuzione, che rende l'algoritmo estremamente adatto per il rilevamento in tempo reale;
- la capacità di ragionare sull'intera immagine, che permette di sfruttare informazioni sul contesto in cui sono inseriti gli oggetti;
- un'elevata generalizzazione, che genera ottimi risultati anche quando i modelli addestrati su immagini naturali vengono utilizzati per eseguire il rilevamento su disegni o opere d'arte;
- basso numero di falsi positivi.

La rete tratta il rilevamento come un problema esclusivamente di regressione. L'immagine di input viene ridimensionata e divisa in una griglia di dimensioni $S \times S$, dove ogni cella è responsabile di predire B di *bounding box* e i relativi livelli di confidenza. Il livello di confidenza riflette la fiducia nel fatto che il *bounding box* associato contenga un oggetto e l'accuratezza nella localizzazione della previsione. Sia $P_i(oggetto)$ la probabilità che il riquadro i contenga un oggetto e sia $IoU(i, oggetto)$ l'intersection over union tra i e il *ground truth box* ad esso associato. Il livello di confidenza o è dato da:

$$o_i = P_i(oggetto)IoU(i, oggetto) \tag{4.6}$$

I regressori sono altamente specializzati e non condividono i pesi in modo che ognuno si adatti a predire riquadri di determinate dimensioni e classi. Se una cella contiene il centro di un oggetto allora è responsabile di stimare la posizione e le dimensioni del suo riquadro di delimitazione. Se una cella non è associata a nessuno oggetto allora il livello di confidenza ad essa associato è nullo.

Le celle, oltre alle coordinate dei riquadri, per ogni classe $j \in \{1, 2, ..., k\}$ del problema stimano la probabilità $P_i(classe_j|oggetto)$ che l'oggetto eventualmente

contenuto al suo interno appartenga a j. La probabilità è condizionata al fatto che la cella contenga effettivamente un oggetto. Nella fase di test viene calcolato il livello di confidenza relativo ad ogni classe k:

$$o_i^{(j)} = P_i(classe_j|oggetto)P_i(oggetto)IoU(i, oggetto) \tag{4.7}$$

Questo punteggio rappresenta la probabilità che un oggetto della classe k appaia nella cella tenendo conto anche dell'aderenza dell'oggetto predetto a quello reale.

L'output prodotto dalla rete è un tensore di dimensioni $S \times S \times (B+1+k)$. La tecnica utilizzata da *YOLO* per il rilevamento genera un numero fisso di riquadri pari a BS^2, indipendentemente dagli oggetti contenuti nell'immagine. Alcuni riquadri potrebbero non contenere nessun oggetto, mentre altri potrebbero essere ridondanti. Risulta quindi necessario eseguire un filtraggio dei *bounding box* predetti dalla rete in modo tale da ottenere un unico riquadro per ogni oggetto.

A tale scopo *YOLO* utilizza il metodo della soppressione dei non massimi (*Non Max Suppression - NMS*). I *bounding box* predetti vengono filtrati applicando la soppressione dei non massimi in modo da eliminare i riquadri ridondanti. Lo strato di output presenta una funzione di attivazione lineare, mentre gli strati nascosti utilizzano la funzione di attivazione:

$$\phi = \begin{cases} x & se \quad x > 0 \\ 0.1x & altrimenti \end{cases} \tag{4.8}$$

[37] La rete *YOLO* viene addestrata minimizzando una funzione di perdita com-

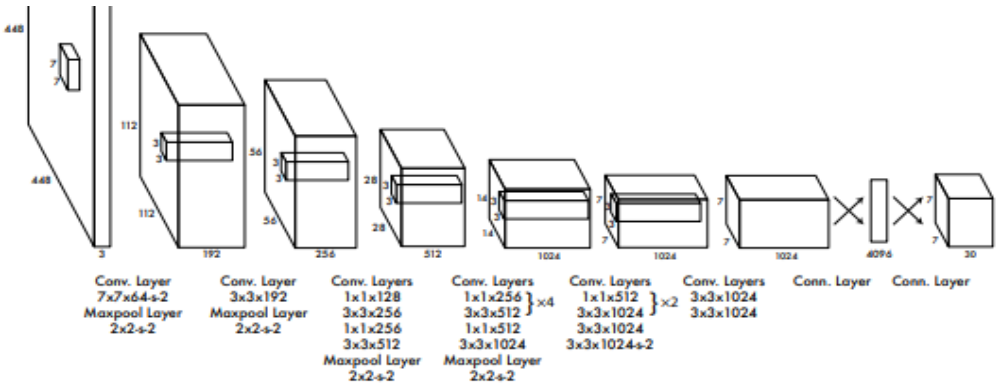


Figura 4.3: Esempio di una rete con architettura *YOLO*, con $S = 7$, $B = 2$ e $k = 20$. La rete presenta 24 strati convoluzionali e due strati pienamente connessi. L'immagine di input viene ridimensionata in modo che abbia una dimensione di 416×416 *pixel*. L'output è un tensore $7 \times 7 \times 30$ di valori. [37]

posta da vari termini. Siano x_i, y_i, w_i, h_i le coordinate dell'i-esima previsione e siano $\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i$ le coordinate del *ground truth box* associato. Sia C_i il livello di confidenza associato all'i-esima previsione e sia $\hat{C}_i$ il suo *intersection over union* con il *ground truth box* associato. Sia $p_i(c)$ la probabilità che l'i-esima predizione appartenga all'c-esima classe e sia $\hat{p}_i(c)$ la probabilità che la c-esima classe ha di comparire all'interno della cella che ha generato l'i-esima predizione. Sia B

il numero di bounding boxes predetti da ogni cella e sia S^2 il numero totale di celle contenute nella griglia in cui viene suddivisa l'immagine. In YOLO viene utilizzata una funzione di perdita composta da tre termini:

- la perdita di classificazione, che misura l'errore compiuta nella classificazione di un oggetto, data da:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{4.9}$$

- la perdita di localizzazione, che misura l'errore commesso nello stimare le coordinate e le dimensioni del riquadro, data da:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \tag{4.10}$$

$$+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{4.11}$$

$$\tag{4.12}$$

- la perdita di confidenza, che misura l'errore commesso nella stima dei livelli di confidenza, data da:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \tag{4.13}$$

Il termine $\mathbb{1}_i^{obj}$ risulta pari a 1 se nell'i-esima cella è presente un oggetto, mentre il termine $\mathbb{1}_{ij}^{obj}$ risulta pari a 1 se il j-esimo regressore associato alla cella i è associato ad un determinato oggetto. Il termine $\mathbb{1}_{ij}^{noobj}$ è l'opposto di $\mathbb{1}_{ij}^{obj}$. La perdita di localizzazione viene calcolata solamente se la cella associata contiene un oggetto.

Il gradiente della funzione di perdita risulta maggiormente influenzato dalle celle che non sono associate a nessun oggetto, cioè quelle con livello di confidenza nullo, poiché sono presenti in maggior quantità. A tale scopo vengono introdotti due coefficienti λ_{coord}, che da un maggior peso alla funzione di perdita di localizzazione rispetto a quella associata alla confidenza, e λ_{noobj}, che da un peso minore alla funzione di perdita associata alle celle dello sfondo.

Gli errori commessi nella stima delle dimensioni dei riquadri piccoli dovrebbero avere un maggior peso rispetto a quelli commessi nei riquadri grandi. Per questo motivo anziché calcolare l'errore quadratico medio sulla lunghezza e sull'altezza di un riquadro *YOLO* utilizza le rispettive radici quadrate. In questo modo le piccole discrepanze calcolate sui riquadri grandi risultano trascurabili, mentre quelle calcolate sui riquadri piccoli restano accentuate. [37]

4.2.2 Single Shot multibox Detector

Nel 2015 Wei Liu propose un nuovo algoritmo di *object detection* in grado di combinare velocità e accuratezza, detto *Single Shot multibox Detector (SSD)*. L'algoritmo appartiene alla categoria delle reti neurali a passata singola, come *YOLO*, ma riesce a raggiungere livelli di accuratezza più elevati.

SSD è basato su una rete convoluzionale *feed-forward*, divisa in due blocchi. La prima parte è la *backbone* e si occupa dell'estrazione di caratteristiche. Il secondo blocco è costituito da una serie di strati convoluzionali in cascata, la cui dimensione diminuisce progressivamente. La rete effettua rilevamenti a più scale diverse usando una tecnica simile alle piramidi di immagini. Anziché ridimensionare l'immagine di input più volte il rilevamento a scale multiple viene effettuato utilizzando *features map* con risoluzioni differenti, in modo da condividere le caratteristiche convoluzionali e risparmiare spazio in memoria. Le diverse *features map* hanno un diverso campo recettivo e sono specializzate a rilevare determinate scale di oggetti. I primi strati sono in grado di acquisire i dettagli più fini, mentre gli ultimi quelli più grossolani. Al contrario *YOLO* aveva problemi col rilevamento di piccoli oggetti, poiché esegue il rilevamento su una singola scala.

SSD discretizza lo spazio di output associando ad ogni cella di una *features map* un insieme di riquadri di ancoraggio di dimensioni predefinite[10]. Ogni strato genera sempre lo stesso numero di riquadri. Il rilevamento avviene applicando alla *features map* di input, di dimensioni $m \times n$ con p canali, un piccolo filtro convoluzionale di dimensioni $3 \times 3 \times p$. La rete usa predittori (filtri) separati per rilevamenti con diverso rapporto d'aspetto e li applica a più mappe di caratteristiche con risoluzioni diverse eseguendo un rilevamento a più scale.

Il compito del filtro è quello di predire per ogni cella della *features map* i punteggi di appartenenza alle classi e gli *offset* che indicano lo scostamento della posizione e delle dimensioni (le differenze tra i centri, la differenza tra le altezze e tra le lunghezze) rispetto al riquadro di ancoraggio associato. Ai riquadri ottenuti con l'inferenza viene poi applicata la soppressione dei non massimi. Sia k il numero di riquadri predefiniti e sia c il numero delle classi. Il rilevamento avviene applicando $(c+4)k$ filtri ad ogni cella. In totale alla *feature map* vengono applicati $(c+4)kmn$ filtri. Gli *offset* vengono utilizzati per far aderire il riquadro di ancoraggio all'oggetto contenuto.

Prima di addestrare la rete è necessario associare a ciascun *ground truth box* un riquadro di ancoraggio. Per fare questo viene calcolata la sovrapposizione tra ciascun riquadro di ancoraggio e i riquadri *target* del dataset utilizzando l'*intersection over union*. Ad ogni *target* viene associato il riquadro che si sovrappone meglio, in questo modo tutti gli esempi sono associati almeno ad un riquadro di ancoraggio. Successivamente tutti i riquadri di ancoraggio che presentano un *intersection over union* superiore a 0.5 vengono associati ai corrispondenti esempi.

La maggior parte dei riquadri di ancoraggio non sarà collegato a nessun *ground truth box*, quindi si ha uno squilibrio tra gli esempi positivi e quelli negativi. Per questo motivo vengono selezionati solamente alcuni campioni tra

[10] Gli autori li chiamano *default boxes*.

gli esempi negativi in modo da avere un rapporto di 3 : 1 con quelli positivi. Sia P l'insieme dei riquadri di ancoraggio positivi e sia N l'insieme dei negativi. La classe $c = 0$ è assegnata allo sfondo. Sia $t \in \mathbb{R}^4$ il vettore di coordinate associato al riquadro predetto e sia $g \in \mathbb{R}^4$ il vettore di coordinate associato al *ground truth box*. Entrambi sono parametrizzati allo stesso modo di *Faster R-CNN*. Sia $c^p \in \mathbb{R}$ il punteggio di appartenenza alla classe p predetto per un riquadro. Al vettore viene applicata la funzione *softmax* ottenendo $\hat{c}^p = softmax(c^p)$. Sia j l'indice associato al *bounding box* predetto e sia k quello associato al corrispondente *ground truth box*. La funzione di perdita C è data dalla somma dell perdita di localizzazione e della perdita di confidenza:

$$
\begin{aligned}
&C(x, c^p, j, k) = \frac{1}{N}(C_{conf}(x, c^p) + \alpha C_{loc}(x, t^j, g^k) \\
&dove, \quad
\begin{aligned}
&C_{conf}(x, c^p) = -\sum_{i \in P} x_{ik}^p log(\hat{c}_i^p) - \sum_{i \in N} x_{ik}^p log(\hat{c}_i^0) \\
&C_{loc}(x, j, k) = \sum_{i \in N} \sum_{m \in x,y,w,h} x_i^p smooth_{L1}(t_m^j, g_m^k)
\end{aligned}
\end{aligned}
\tag{4.14}
$$

Il termine x_{ik} è pari a 1 se il riquadro di ancoraggio i è collegato al *ground truth box* j e pari a 0 altrimenti.

Si considerino m *features map*. La scala s dei riquadri di ancoraggio associati alla *features map* $k \in \{1, ..., m\}$ viene calcolata nel seguente modo:

$$
s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1) \tag{4.15}
$$

dove $s_{min} \in \mathbb{R}$ e $s_{max} \in \mathbb{R}$ sono rispettivamente la scala minima e quella massima. Le varie scale risultano equispaziate.

Per rendere il modello più robusto al variare dei dati di ingresso vengono impiegate tecniche di *data augmentation*. Viene ritagliata una regione dall'immagine di ingresso. La regione viene poi ridimensionata ad una dimensione fissa. L'immagine viene distorta, aumentando e diminuendo luminosità, contrasto, saturazione, e capovolta con una probabilità del 50%. [38]

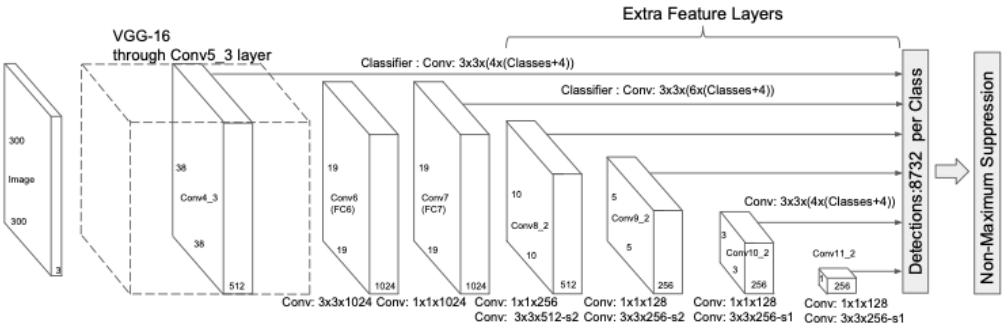


Figura 4.4: Esempio di rete con architettura *SSD* con *backbone VGG-16*. [38]

4.2.3 YOLO 9000

Gli algoritmi di rilevamento basati su *YOLO* risultano meno accurati rispetto a quelli basati su regioni, poiché commettono un maggior numero di errori di

localizzazione riscontrando particolari difficoltà nel rilevamento di piccoli oggetti o di oggetti molto vicini.

Quando i centri di due o più oggetti cadono all'interno della stessa cella questa riuscirà a prevederne solamente uno, poiché ogni cella della griglia è in grado di prevedere i **bounding box** relativi ad un solo oggetto.

Nel dicembre 2017 viene rilasciata una nuova versione di *YOLO* in grado di riconoscere più di 9000 diverse classi di oggetti, detta *YOLO 9000*.

YOLO 9000 risulta più accurata e più veloce rispetto alla versione precedente grazie all'impiego di riquadri di ancoraggio in sostituzione degli strati *fully-connected*. I riquadri di ancoraggio consentono di rilevare più di un oggetto per ogni cella e permettono alla rete di imparare a riconoscere classi di oggetti caratterizzate da dimensioni e proporzioni molto differenti.

I regressori di *YOLO 9000* impiegano una parametrizzazione delle coordinate diversa da quella impiegata nella rete per la proposta di regioni in *Faster R-CNN*. Le coordinate dei riquadri vengono vincolate alla cella di appartenenza, in modo da stabilizzare il modello e rendere l'addestramento più veloce.

Siano C_x, C_y le coordinate dell'angolo in alto a sinistra della cella responsabile di prevedere le coordinate di un determinato oggetto e siano p_w, p_h le dimensioni del relativo riquadro di ancoraggio. I regressori associati alla cella predicono un vettore $t = (t_x, t_y, t_w, t_h, t_o)$, i cui valori rappresentano rispettivamente le coordinate del centro, le dimensioni e il livello di confidenza di un *bounding box*. Le coordinate del centro e il livello di confidenza vengono normalizzati applicando una funzione logistica σ, che mappa i valori delle coordinate nell'intervallo $[0, 1]$. Le coordinate vengono parametrizzate come segue:

$$b = \begin{pmatrix} b_x \\ b_y \\ b_w \\ b_h \\ P_i(oggetto) IoU(i, oggetto) \end{pmatrix} = \begin{pmatrix} \sigma(t_x) + c_x \\ \sigma(t_y) + c_y \\ p_w e^t_w \\ p_h e^t_h \\ \sigma(t_o) \end{pmatrix} \tag{4.16}$$

Le dimensioni e le forme dei riquadri di ancoraggio non vengono scelte in maniera manuale come in *Faster R-CNN*, ma vengono determinate utilizzando tecniche di *clustering*. Queste tecniche sono basate su apprendimento non supervisionato. I *bounding box* appartenenti al dataset di addestramento vengono raggruppati in insiemi detti *cluster* in base alle loro dimensioni e proporzioni. I *cluster* così ottenuti vengono utilizzati come misure di riferimento per i priori, ottenendo i migliori riquadri possibili per ogni classe di oggetto. Il numero di *cluster* predefinito è pari a 5 e rappresenta un compromesso tra complessità e prestazioni.

YOLO 9000 utilizza come *backbone* per l'estrazione di caratteristiche una rete neurale con struttura simile a quella di **VGG** detta **Darknet 19**, caratterizzata da 19 strati convoluzionali. Le prestazioni della rete vengono migliorate applicando la tecnica della normalizzazione dei lotti dopo ogni strato convoluzionali. [39]

4.2.4 YOLO v3

Nonostante i miglioramenti introdotti *YOLO 9000* presenta spesso difficoltà nel rilevamento di oggetti piccoli. La causa di questo comportamento è stata attribuita alla perdita di caratteristiche a grana fine dovuta ad un eccessivo sotto-campionamento durante la propagazione nella rete.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

(a) Architettura di **Darknet 19**. [39]

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

(b) Architettura di **Darknet 53**. [40]

Figura 4.5: *Backbone* usata da *YOLO* per l'estrazione di caratteristiche.

Nell'aprile del 2018 viene rilasciata una nuova versione detta *YOLO v3*, molto più accurata rispetto alla precedente. La rete utilizza una nuova spina dorsale molto più profonda chiamata **Darknet 53**, costituita da 53 strati convoluzionali. L'elevata profondità della rete viene raggiunta impiegando dei blocchi residuali, come nelle reti **ResNet**.

La maggiore profondità della *backbone* rende l'algoritmo maggiormente performante e più robusto, ma allo stesso tempo più lento. La rete effettua il rilevamento su tre scale diverse riuscendo a risolvere il problema del riconoscimento di piccoli oggetti.

L'uscita della rete è generata applicando un kernel $1x1$ di dimensioni $1x1x(B(5+C))$ alla mappa delle caratteristiche, dove B è il numero di *bounding box* rilevati per ogni cella e C è il numero delle classi. L'output della rete è rappresentato da un vettore di $B(5+C)$ elementi, cioè $5+C$ valori per ogni *bounding box* predetto. A differenza di *YOLO 9000* vengono utilizzati 9 riquadri di ancoraggio per ogni cella. [40]

4.3 Sviluppo di un modello

4.3.1 Ambiente di sviluppo

La realizzazione di un modello di *object detection* basato sul *deep learning* richiede i seguenti passi:

- creazione del dataset di addestramento, che avviene allo stesso modo dell'approccio tradizionale;
- scelta degli iperparametri, cioè quei parametri che permettono di controllare il processo di apprendimento;
- addestramento del modello, utilizzando un opportuno *framework*.

Gli iperparametri si suddividono in due categorie:

- i parametri che definiscono la struttura del modello, come il numero e la dimensione degli *hidden layer*, lo *stride* usato nella convoluzione e nel *pooling*, la dimensione del *padding* utilizzato dalla tecnica di *zero padding*, il numero di riquadri di ancoraggio, la dimensione delle finestre di *RoI pooling*...
- i parametri dell'algoritmo di apprendimento, come il *batch size*, il numero di epoche, il tasso di apprendimento, il coefficiente γ utilizzato nella tecnica *momentum*...

Affinché un modello possa essere sufficientemente accurato, esso deve essere addestrato su un dataset molto vasto e diversificato, contenente anche altre classi oltre a quelle strettamente inerenti al problema. Questo significa che per addestrare un modello per il riconoscimento e il rilevamento di olive è necessario costruire un dataset non solo molto numeroso, ma anche contenente classi diverse di oggetti. Per questo motivo in questa sede si preferisce impiegare la tecnica del *transfer learning*.

Nonostante l'impiego di modelli pre-allenati l'addestramento di una rete neurale è comunque oneroso dal punto di vista computazionale, sia per la memoria occupata che per il tempo di esecuzione. Per questo motivo spesso l'addestramento dei modelli viene svolto tramite servizi di *cloud computing*.

Un esempio è **Google Collaboratory** (**Google Colab**) rilasciato da Google nel 2015. Questo servizio permette di scrivere algoritmi di apprendimento automatico in Python direttamente da un *browser web* attraverso l'utilizzo degli *Jupyter nootebook*. Gli *Jupyter nootebook* sono dei particolari documenti, contenenti codici, grafici, link, immagini e testi, che permettono di eseguire *script Python* sui server di Google, disponendo di potenti architetture *hardware*. In questo modo chi lavora nell'ambito dell'apprendimento automatico può concentrarsi esclusivamente sulla programmazione e sulla progettazione svincolandosi dai problemi legati all'*hardware*.

4.3.2 TensorFlow Model Garden

TensorFlow Model Garden è un *software* con licenza *Apache 2.0* che fornisce supporto per la creazione di modelli di apprendimento profondo basati su **TensorFlow** per la soluzione di problemi di *object detection*, *object recognition* e riconoscimento del linguaggio naturale. Per quanto riguarda il rilevamento di oggetti **TensorFlow Model Garden** fornisce diverse architetture di modelli pre-allenati tra cui *Faster R-CNN* e *SSD*.

Per poter installare il *software* sull'ambiente di sviluppo **Google Colab** è sufficiente clonare la *repository* ed aggiungere alcune variabili d'ambiente.

Per terminare la configurazione dell'ambiente è necessario installare le librerie **TensorFlow** e **Tensorflow Slim** utilizzando **PIP** e compilare i *buffer di protocollo*. Un *buffer di protocollo*[11] è un particolare formato di file binario, che viene impiegato per la serializzazione dei dati. **TensorFlow Model Garden** utilizza questo formato per la configurazione e l'addestramento dei modelli di *object detection*. Il processo di compilazione genera uno *script Python* contenente le classi e gli oggetti serializzati.

TensorFlow rappresenta i modelli di apprendimento profondo attraverso grafi, dove ogni nodo corrisponde ad un tensore e ogni arco corrispondo ad un'operazione da eseguire su di essi. Questa struttura risulta particolarmente efficiente, poiché consente di suddividere la computazione in più sottografi indipendenti che possono essere eseguiti in parallelo.

Un classificatore, che sia implementato con un algoritmo di apprendimento automatico come una macchina a vettori di supporto o sia implementato in una rete neurale impiegando una serie di strati *fully-connected*, richiede che le categorie di appartenenza degli oggetti da classificare siano espresso in formato numerico. Risulta quindi necessario mappare i nomi delle classi dal dominio delle stringhe ad un dominio di interi con valori nell'intervallo $[0, 1, ..., k-1$, dove k è il numero delle etichette.

La libreria **Tensorflow** richiede che le etichette siano mappate all'interno di un file in formato *TensorFlow Graph Text*[12], detto mappa delle etichette (*label map*).

Il formato **XML**, introdotto dal dataset **Pascal Visual Object Classes**, è diventato uno standard per la gestione di annotazioni nell'ambito del riconoscimento e rilevamento di oggetti. Nonostante ciò **TensorFlow Model Garden** non utilizza file di annotazione per la gestione dei dataset di addestramento. Tutti gli esempi vengono infatti rappresentati in un formato binario, detto **TensorFlow Record**, dove ogni *record* di *byte* contiene i dati relativi ad un'immagine e alle sue annotazioni.

Questo problema può essere facilmente risolto impiegando le *API* di **TensorFlow Model Garden**, che consentono di convertire un dataset basato sull'architettura **Pascal Visual Object Classes** in formato binario. Per questo motivo risulta conveniente utilizzare annotazioni di tipo **XML** per il datatset di olive.

[11] Un *buffer di protocollo* è un file con estensione **.proto**.
[12] Il *TensorFlow Graph Text* è un formato di file con estensione **.pbtxt**.

La sezione *Detect Model Zoo* di **TensorFlow Model Garden** contiene una serie di modelli per il rilevamento di oggetti addestrati su **COCO**. I modelli sono organizzati in una tabella dove per ogni architettura viene indicata la velocità di inferenza, espressa in millisecondi, su un'immagine di dimensioni 600×600, la *mean average precision* calcolata sul dataset **COCO**, il *link* per il download e il tipo di output supportato. L'uscita di un modello generata durante la fase di inferenza può essere di due tipi:

- *boxes*, cioè una lista dei *bounding boxes* contenuti nell'immagine;
- *masks*, cioè una lista di maschere che individuano gli oggetti[13] contenuti nell'immagine.

I modelli forniti da *framework* sono organizzati in quattro file:

- **model.ckpt.meta**, che contiene la struttura del grafo;
- **model.ckpt.data-00000-of-00001**, che contiene i valori delle variabili;
- **model.ckpt.index**, che contiene una tabelle dove sono salvati i metadati relativi ad ogni tensore;
- **pipeline.config**, che contiene informazioni relative alla configurazione del modello.

I primi tre file sono detti file di *checkpoint* e vengono utilizzati per far ripartire l'addestramento da dove era stato interrotto. Questi file vengono poi modificati durante la fase di apprendimento in modo tale le connessioni e i nodi del grafo. Il *link* fornito consente di scaricare il modello sotto forma di archivio **tar**[14]. L'operazione è identica a quella eseguita per il download della *repository*. L'estrazione di archivi **tar** non è supportata dalla libreria **zipfile**. A tale scopo viene utilizzata la libreria **tarfile**, che fornisce un metodo **open()** per aprire l'archivio, un metodo **extractall()** per estrarne il contenuto e un metodo **close()** per chiuderlo.

Prima di iniziare l'addestramento è necessario modificare il file di configurazione inserendo i percorsi del dataset di addestramento, del dataset di test e delle relative mappe di etichette. Bisogna inoltre specificare il percorso del file **model.ckpt.meta**. Il file può essere utilizzato anche per modificare gli iperparametri della rete.

Il **TensorFlow Model Garden** fornisce delle *API* per riaddestrare i modelli pre-allenati. Durante l'addestramento vengono modificati solamente i pesi relativi agli ultimi strati. L'aggiornamento del modello comporta la creazione di altri file di *checkpoint*. Il *software* consente di monitorare in tempo reale le prestazioni del modello attraverso un *tool* chiamato **TensorBoard**, in modo da valutare la presenza di *overfitting*.

[13] Una maschera è il risultato dell'operazione di segmentazione sul contenuto di un *bounding box*.

[14] Gli archivi **tar** sono dei file con estensione **tar.gz**.

I *checkpoint* consentono di interrompere e riprendere l'addestramento a seconda delle necessità, memorizzando variabili, metadati e operazioni in file separati. Il modello per poter essere utilizzato nella fase di inferenza deve prima essere "congelato".

Durante la fase di inferenza viene eseguita solo la propagazione in avanti. Il congelamento del modello elimina tutte le variabili, i metadati e le operazioni utilizzate per aggiornare i pesi, come i dati relativi al gradiente, o salvare *checkpoint* che non sono necessarie nella fase di rilevamento rilevamento. Tutte le informazioni relative alla struttura e alle variabili del modello vengono unite in un unico file. Spesso alcuni strati vengono fusi insieme in modo da ottimizzare il processo di inferenza. Anche per questa operazione **TensorFlow Model Garden** fornisce delle *API*.

4.3.3 Ultralytics YOLO v3

TensorFlow Model Garden non fornisce supporto per l'addestramento di modelli basati su *YOLO*, per questo motivo è stato deciso di utilizzare un altro *framework*, **Ultralytics YOLO v3**, che implementa l'algoritmo di Joseph Redmon utilizzando la libreria di apprendimento automatico **PyTorch**.

Il *software* può essere installato allo stesso modo di **TensorFlow Model Garden** clonando la *repository*. Non è necessario impostare variabili d'ambiente o installare altre librerie, poiché **PyTorch** è già presente di *default* in **Google Colab**.

Ultralytics YOLO v3 non è compatibile con il formato di annotazioni *Pascal Visual Object Classes*. Le annotazioni sono rappresentate da un file testuale con estensione **.txt** contenente su ogni riga le informazioni relative ad un *bounding box*.

Il primo valore indica la classe di appartenenza del riquadro, codificata con un numero intero, mentre i restanti valori rappresentano rispettivamente le due coordinate del centro, la lunghezza e l'altezza del riquadro. Sia le coordinate che le dimensioni sono normalizzate rispetto alle dimensioni dell'immagine in modo da ottenere valori nell'intervallo $[0, 1]$. Questo formato prende il nome di *YOLO Darknet TXT*.

Dato un generico *bounding box* contenuto in un'immagine di dimensioni $W \times H$ siano x_c e y_c rispettivamente l'ascissa e l'ordinata del centro del riquadro. Siano x, y, h, w le quattro coordinate utilizzate da Labelbox per esprimere il *bounding box*. Siano x'_c, y'_c, h', w' le quattro coordinate normalizzate utilizzate dal formato *YOLO Darknet TXT*. La conversione viene effettuata con le seguenti formule:

$$\begin{aligned} x_c &= \frac{x+\frac{w}{2}}{W} \\ y_c &= \frac{y+\frac{h}{2}}{H} \\ w' &= \frac{w}{W} \\ h' &= \frac{h}{H} \end{aligned} \tag{4.17}$$

Una volta configurato l'ambiente il primo passo da fare è la creazione della mappa delle etichette. La struttura in questo caso è molto semplice. Infatti, è sufficiente elencare i nomi delle classi in un file testuale con estensione **.names**. Ad ogni riga deve corrispondere una stringa. Sarà poi il *software* ad associare un intero ad ogni stringa in base all'ordine in cui sono disposte.

```
"""
Questa funzione crea la mappa delle etichette. Restituisce una
    stringa da scrivere su file. Il file deve avere estensione ".
    names".
"""

def create_label_map():
    return "olives\n"
```

Il *software* richiede che il percorso relativo ad ogni singola immagine del dataset di addestramento sia salvato all'interno di un file di testo con estensione **.txt**. Affinché non ci siano problemi durante l'addestramento è necessario che le immagini e le etichette siano disposte in due cartelle differenti e che i file relativi ad uno stesso esempio abbiano lo stesso nome, come spiegato in precedenza.

I nomi delle immagini contenute nel dataset possono essere ottenuti utilizzando la funzione **listdir()**, fornita dalla libreria **os**, che restituisce una lista di tutti i file contenuti al percorso passato come parametro.

Devono essere creati due file diversi, un per il dataset di addestramento e uno per quello di test.

```
"""
Questa funzione genera una lista dei percorsi relativi alle immagini
    contenute nel dataset. Restituisce una stringa da scrivere su
    file. Il file deve avere estensione ".txt". La funzione richiede
    come parametro il percorso della cartella in cui sono salvate le
    immagini.
"""

def create_olives_txt(path):
    #Lista contenente i nomi delle immagini del dataset
    pictures_name=os.listdir(path)
    #Crea una stringa vuota
    s=""
    #Scorre le immagini del dataset
    for picture in pictures_name:
        #Aggiunge il percorso dell'immagine alla stringa
        s+=path+"/"+picture+"\n"
    return s
```

La *directory* in cui salvare la mappa delle etichette e il file dei percorsi può essere scelta in modo arbitrario. L'importante è specificare il loro percorso in un opportuno file di configurazione con estensione **.data**.

```
"""
Questa funzione genera una lista dei percorsi relativi ai file di
    configurazione e alla mappa delle caratteristiche. La lista
    specifica anche il numero delle classi contenute nel dataset.
    Restituisce una stringa da scrivere su file. Il file deve avere
    estensione ".data". La funzione richiede come argomento i
    percorsi relativi ai file di configurazione, sia per il train set
     che per il test set, e il percorso della mappa delle etichette.
```

```
"""

def create_olives_data(train_path,test_path,label_map_path):
    #Aggiunge il numero di classi
    s=("classes=1\n")
    #Aggiunge il percorso del file di configurazione (estensione .txt
    ) per l'addestramento
    s+=("train="+train_path+"\n")
    #Aggiunge il percorso del file di configurazione (estensione .txt
    ) per la fase di test
    s+=("valid="+test_path+"\n")
    #Aggiunge il percorso della label map (estensione .names)
    s+=("names="+label_map_path+\n")
    return s
```

Ultralytics YOLO v3 fornisce delle *API* per addestrare modelli pre-allenati di *Darknet 53*. Sono disponibili diverse versioni della rete neurale a seconda delle esigenze, che possono essere selezionate scegliendo un opportuno file di configurazione[15]. Ogni file di configurazione descrive la struttura dei vari strati della rete. Questi file possono essere modificati a seconda delle esigenze.

Per addestrare un modello per rilevamento di olive è necessario aggiornare opportunamente il file di configurazione usato. In particolare, bisogna modificare i parametri relativi agli *YOLO layer*, cioè gli strati in cui viene calcolata la funzione di perdita. Il *kernel* utilizzato in questi strati ha dimensioni $1 \times 1 \times (5+C)B$, dove C è il numero delle classi e B è il numero di riquadri di ancoraggio per ogni cella.

I modelle pre-allenati forniti da **Ultralytics** sono addestrati sul dataset **COCO** e utilizzano $C = 80$ e $B = 3$ come valori predefiniti. Nel rilevamento di olive si ha invece $C = 1$, quindi gli iperparametri **filters**, che indica il valore del prodotto $(5 + C)B$, e **classes**, che indica il valore di C, devono essere opportunamente aggiornati. In alternativa, vengono forniti anche file di configurazione per addestrare dataset con una singola classe.

L'addestramento restituisce un file con estensione **.pt** contenente i pesi del modello, che può essere direttamente usato per la fase di inferenza.

4.3.4 Implementazione dell'algoritmo

Una volta terminato l'addestramento del modello si può procedere con lo sviluppo dell'algoritmo vero e proprio. Questa fase consiste nell'implementare la seguente interfaccia **OlivesCounter()**:

```
def OlivesCounter(input_image,model,label_map,delta_x,delta_y,
    prec_bbox_list):
    (bbox_list,output_image)=detector(input_image,model,label_map)
    (bbox_list,conteggio)=contatore(bbox_list,prec_bbox_list,delta_x,
    delta_y)
    return (conteggio,bbox_list,output_image)
```

La funzione **detector()** ha il compito di riconoscere e localizzare le olive presenti all'interno del fotogramma che riceve in ingresso. L'immagine di ingresso viene passata come istanza della classe **Image** implementata dalla libreria

[15] Questi file di configurazione hanno estensione **.cfg**.

PIL. La funzione restituisce il fotogramma su cui sono stati disegnati i *bounding box* e la contenente la le dimensioni e la posizione di ogni oliva all'interno dell'immagine.

```
def detector(input_image,model,label_map):
    output_tensor=RunInference(input_image,model)
    (width,height)=image.size
    bbox_list=ParseResults(output_tensor,label_map,width,height)
    output_image=DrawBoundingBox(image,bbox_list)
    return (bbox_list,output_image)
```

L'algoritmo di rilevamento esegue l'inferenza usando la funzione **RunInference()**, che restituisce il tensore di output in uscita dalla rete neurale contenente le coordinate dei *bounding box* normalizzate. Le coordinate devono poi essere denormalizzate e convertite nel formato *Pascal Visual Object Classes*, usando la funzione **ParseResult()**. Infine è necessario disegnare i riquadri di delimitazione intorno alle olive, utilizzando la funzione **DrawBoundingBox()** descritta nella sezione 2.3.5. Di seguito una possibile implementazione della funzione **ParseResults()**.

```
def ParseResults(output_tensor,label_map,width,height):
  bbox_list=[]
  for *edges,confidence,category in output_tensor:
    bbox={}
    bbox["classe"]=label_map[str(int(category))]
    bbox["xmin"]=edges[1]*width
    bbox["ymin"]=edges[0]*height
    bbox["xmax"]=edges[3]*width
    bbox["ymax"]=edges[2]*height
    bbox["conf"]=confidence
    bbox_list.append(bbox)
  return bbox_list
```

La funzione **contatore()** a partire dalla lista dei *bounding box* presenti nel fotogramma corrente e quella associata al fotogramma precedente effettua il conteggio delle olive, evitando di ricontare quelle già considerate. A tale scopo la funzione necessita di informazioni riguardo lo spostamento in *pixel* delle olive tra due inquadrature successive. Queste informazioni possono essere calcolate a partire dalla velocità e dalla direzione del *robot* e dalla risoluzione della fotocamera, misurata in punti per pollice. Se la telecamera si sposta verso destra, i riquadri all'interno dell'inquadratura si muoveranno verso sinistra e viceversa. La funzione restituisce il conteggio e la lista dei riquadri, cui sono stati tolti quelli ripetuti.

```
    def contatore(bbox_list,prec_bbox_list,delta_x,delta_y):
        for curr in bbox_list:
            curr["xmin"]+=delta_x
            curr["xmax"]+=delta_x
            curr["ymin"]+=delta_y
            curr["ymax"]+=delta_y
            for prec in prec_bbox_list:
                if IoI(prec,curr)>0.9:
                    bbox_list.remove(curr)
        return(bbox_list,len(bbox_list))
```

La precedente funzione individua i riquadri ripetuti utilizzando l'*Intersection over union* e li elimina dalla lista degli oggetti trovati. I riquadri associati ad una stessa oliva nei due diversi fotogrammi sono infatti caratterizzata da una sovrapposizione quasi totale.

4.3.5 Caricare il modello

La funzione **detector()** richiede come argomenti un modello di riferimento e una mappa delle etichette. Queste strutture dati devono quindi essere caricate in memoria prima di eseguire l'inferenza attraverso le funzioni **LoadModel()** e **LoadLabelMap()**. La loro implementazione varia a seconda del *framework* utilizzato per addestrare i modelli.

I modelli realizzati con **TensorFlow** sono rappresentati dal grafo congelato e sono salvati in un file serializzato con estensione **.pb**. Per poter caricare il modello in memoria è necessario istanziare un oggetto della classe **Graph** e uno della classe **GraphDef**.

La classe **Graph** viene utilizzata per rappresentare le operazioni e i calcoli attraverso un grafo, mentre **GraphDef** viene utilizzata per salvare e caricare i file serializzati. Una volta istanziati gli oggetti è necessario aprire e leggere il modello serializzato, con le funzioni **GFile()** e **read()**, trasformarlo in formato testuale con la funzione **ParseFromString()** ed importarlo nel grafo predefinito.

```
def LoadModel(path):
  graph=tf.Graph()
  frozen_graph=tf.GraphDef()
  file=tf.gfile.GFile(path,'rb')
  frozen_graph.ParseFromString(file.read())
  file.close()
  with graph.as_default():
    tf.import_graph_def(frozen_graph,name='')
  return graph
```

Per leggere la mappa di etichette è sufficiente leggere il file effettuando una semplice operazione di *parsing*. I dati devono poi essere organizzati in una struttara dati, come ad esempio un dizionario.

```
def LoadLabelMap(path):
    file=open(path)
    classi=file.readlines()
    file.close()
    label_map={}
    for (i,classe) in enumerate(classi):
        label_map[str(i)]=classe.replace("\n","").replace(" ","")
    return label_map
```

Il caricamento dei modelli generati con **YOLO v3 Ultralytics** risulta invece più complesso, ma fortunatamente il *software* fornisce alcune librerie a riguardo. Per poterle utilizzare è sufficiente aggiornare la variabile d'ambiente ed importarle. Per caricare il modello è sufficiente istanziare un oggetto della classe **Darknet**, fornendo al costruttore il percorso del file di configurazione e la dimensione dell'immagine di input. Una volta creato il modello è necessario caricare il valore dei pesi.

L'immagine di input deve essere quadrata, di dimensioni 512×512, per questo motivo potrebbe essere necessario ridimensionarla. In alternativa, si può utilizzare la tecnica del *padding* per aggiungere dei bordi all'immagine in modo da evitare distorsioni. In quest'ultimo caso è però necessario fare attenzione alla posizione dei *bounding box* all'interno dell'immagine, che saranno scostati di una dimensione pari a quella del bordo.

```
def LoadModel(cfg_path,weights_path):
  device=torch_utils.select_device(device="")
  model=Darknet(cfg_path,512)
  model.load_state_dict(torch.load(weights_path, map_location=device)
    ['model'])
  model.to(device).eval()
  image=torch.zeros((1,3,512,512),device=device)
  return model
```

Il caricamento della mappa di etichette è identico a quello usato con **TensorFlow**, anche se viene adottata una diversa tecnica di *parsing*.

```
def LoadLabelMap(path):
    file=open(path)
    lines=file.readlines()
    file.close()
    label_map={}
    for line in lines:
        if line.find("id")!=-1:
            id=int(line.replace("id:",""))
        if line.find("name")!=-1:
            label_map[str(id)]=line.replace("name:","").replace("\n",
    "").replace(" ","").replace("'","")
    return label_map
```

4.3.6 Implementazione del rilevatore

Affinché l'algoritmo sia in grado di contare effettivamente le olive bisogna definire un'implementazione per la funzione **RunInference()**, che esegue la fase di inferenza. Questa funzione avrà un'implementazione diversa a seconda del *framework* utilizzato per creare il modello.

Nei modelli basati su **Tensorflow** le operazioni contenute in un grafo vengono eseguite all'interno di un ambiente chiamato sessione, cui sono associate tutte le risorse necessarie durante la fase di esecuzione. Il grafo da eseguire contiene il valore dei pesi e le operazioni da eseguire. Poiché i pesi dello strato di ingresso, cioè il tensore di input non sono conosciuti a priori **TensorFlow** utilizza delle speciali variabili "segnaposto" per poter definire comunque il modello. Al momento dell'esecuzione l'immagine di ingresso viene mappata all'interno del grafo e viene effettuata l'inferenza. L'immagine deve però prima essere trasformata in un *array*. A tale scopo si può utilizzare la funzione **getdata()** della libreria **PIL**, che restituisce una lista contenente il valore di tutti i *pixel* contenuti nell'immagine. La lista può essere trasformata in un *array* utilizzando la libreria **NumPy**.

Una sessione viene creata istanziando un oggetto della classe **Session**, passando come parametro al costruttore il grafo da eseguire, e viene eseguita attraverso la funzione **run()**. La funzione **run()** richiede in ingresso il tensore su

cui eseguire l'operazione e un dizionario che permette di mappare l'immagine in ingresso sui segnaposto. Il risultato è un tensore contenente le informazioni relative ai *bounding box*.

```
def RunInference(image,model):
  operations=model.get_operations()
  all_tensor_names=[]
  tensor_dict = {}
  for op in operations:
    for output in op.outputs:
      all_tensor_names.append(output.name)
  for tensor_name in ['detection_classes:0','detection_boxes:0','
    detection_scores:0']:
    if tensor_name in all_tensor_names:
      tensor_dict[tensor_name]=model.get_tensor_by_name(tensor_name)
  (width,height)=image.size
  image_array=np.array(image.getdata()).reshape((height,width,3)).
    astype(np.uint8)
  input_tensor={model.get_tensor_by_name('image_tensor:0'): np.
    expand_dims(image_array, 0)}
  sess=tf.Session(graph=model)
  output_tensor=sess.run(tensor_dict,feed_dict=input_tensor)
  sess.close()
  output_tensor=numpy.array(output_tensor['detection_boxes:0'][0],
    output_tensor['detection_scores:0'][0],['detection_classes:0'
    ][0])
  return output_tensor
```

La funzione **RunInference()** viene implementata secondo una logica diversa quando viene utilizzato il *framework* **Ultralitycs**, poiché i modelli sono basati sulla libreria di calcolo **PyTorch**. Per prima cosa l'immagine di ingresso, opportunamente ridimensionata, deve essere convertita in un tensore. A tal proposito si può procedere come fatto in precedenza con **TensorFlow**, ma è necessario riorganizzare i dati in modo tale da ottenere una matrice tridimensionale, contenente per ogni fetta i valori dei *pixel* associati a quel canale di colore.

L'*array* deve poi essere convertito in un tensore **PyTorch** e i valori devono essere normalizzati mappandoli nell'intervallo $[0, 1]$. La libreria **PyTorch** richiede che sia specificato il dispositivo su cui verrà allocato il tensore, cioè una *CPU* o una *GPU*. L'inferenza viene eseguita passando il tensore di ingresso al modello. Il risultato è un tensore contenente le informazioni relative ai **bounding box**. I riquadri ottenuti vanno filtrati, eseguendo la soppressione dei non massimi, e il tensore deve essere organizzato in modo che abbia una struttura simile a quella prodotta usando **TensorFlow**

```
def RunInference(image,model):
  device=torch_utils.select_device(device="")
  width,height=image.size
  image_array=np.array(image.getdata()).reshape((height,width,3)).
    astype(np.uint8)
  input_tensor=numpy.array((image_array[:,:,0],image_array[:,:,1],
    image_array[:,:,2]))
  input_tensor=torch.from_numpy(input_tensor).to(device).float()
  input_tensor/= 255.0
  if input_tensor.ndimension() == 3:
```

```
9       input_tensor=input_tensor.unsqueeze(0)
10    detection=model(input_tensor, augment=False)[0]
11    detection=non_max_suppression(detection)
12    detection=detection[0]
13    if len(detection):
14      output_tensor=torch.zeros(detection.shape,device=device)
15      output_tensor[:,[1,3]]=(detection[:,[0,2]])
16      output_tensor[:,[0,2]]=(detection[:,[1,3]])
17      output_tensor[:,4]=detection[:,4]
18      output_tensor[:,5]=detection[:,5]
19    return output_tensor
```

Capitolo 5

Conclusioni

5.1 Risultati

5.1.1 Analisi delle prestazioni dei modelli

Gli algoritmi di *object detection* basati sul *deep learning* sono estremamente adatti a lavorare in ambienti difficili come quelli agricoli, poiché sono caratterizzati da una buona immunità al rumore. Le reti neurali, infatti, sono in grado di lavorare anche quando le immagini di ingresso sono di bassa qualità a causa delle condizioni atmosferiche o dell'illuminazione. L'algoritmo in queste circostanze è caratterizzato da un forte degrado delle prestazioni, ma riesce comunque a lavorare senza bloccarsi. Inoltre, grazie alla tecnica del *transfer learning* è possibile riutilizzare modelli già addestrati adattandoli alle proprie esigenze in modo da semplificare la fase di progettazione.

Nonostante gli innumerevoli vantaggi, anche questi algoritmi hanno anche dei difetti. I modelli basati sul *deep learning* non possono essere descritti in linguaggio simbolico e risultano quindi incomprensibili per l'uomo. Per questo motivo spesso di parla di questi algoritmi come *black box*.

Un altro difetto è che non esistono metodi sistematici per scegliere la rete neurale perfetta per un determinato compito, come il conteggio delle olive. Per questo motivo è necessario testare diversi modelli valutando quale sia il migliore. Di seguito vengono mostrati e confrontati i risultati ottenuti utilizzando diversi algoritmi di *object detection*. I modelli sono stati realizzati utilizzando i *framework* **TensorFlow Model Garden** e **YOLO v3 Ultralytics**, descritti nella sezione 4.3.

Le prestazioni dei vari modelli vengono confrontate usano la metodologia descritta nella sezione 2.3, che prevede il calcolo del *Mean Average Precision*. Le prestazioni vengono valutate su due diversi insiemi di dati in modo da valutare la capacità della rete di generalizzare. In particolare, vengono utilizzati il dataset di addestramento, contenente 50 immagini, e un dataset di test, contenete 70 immagini ottenute attraverso tecniche di *data augmentation*. L'insieme di dati usato per l'addestramento risulta molto più complesso rispetto a quello di test, poiché ciascuna immagine contiene interi alberi di olivo con centinaia di frutti. Nel dataset di test ogni immagine contiene solamente poche olive.

Faster R-CNN Resnet 50						
	Mean Average Precision					
Durata (min)	0	20	40	60	80	100
Train set	0.001	0.719	0.813	0.815	0.817	0.816
Test set	0.002	0.327	0.384	0.381	0.390	0.437

Tabella 5.1: Risultati ottenuti durante l'addestramento di un modello basato su *Faster R-CNN* che impiega come *backbone* una rete residuale con 50 strati.

Faster R-CNN Resnet 101									
	Mean Average Precision								
Durata (min)	0	20	40	60	80	100	120	140	160
Train set	0.004	0.717	0.809	0.814	0.816	0.8117	0.818	0.817	0.817
Test set	0.054	0.357	0.416	0.415	0.427	0.413	0.460	0.470	0.450

Tabella 5.2: Risultati ottenuti durante l'addestramento di un modello basato su *Faster R-CNN* che impiega come *backbone* una rete residuale con 101 strati.

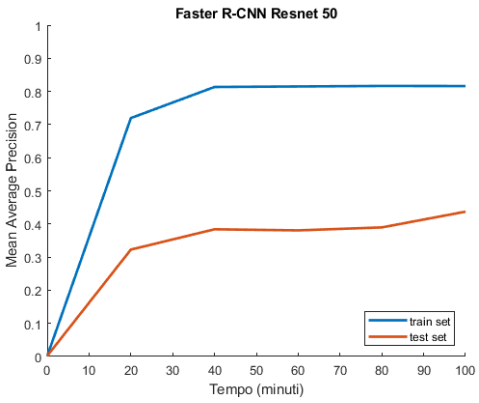


Figura 5.1: Andamento del *Mean Average Precision* durante l'addestramento di *Faster R-CNN ResNet 50*.

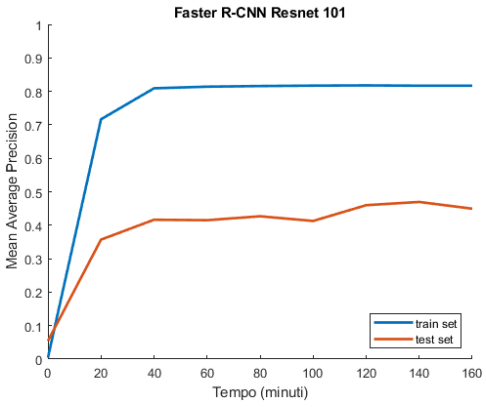


Figura 5.2: Andamento del *Mean Average Precision* durante l'addestramento di *Faster R-CNN ResNet 101*.

Le tabelle 5.1.1 e 5.1.1 mostrano i risultati ottenuti durante l'addestramento di due diversi modelli dell'algoritmo *Faster R-CNN*, attraverso il *framework* **TensorFlow Model Garden**. Ogni venti minuti sono state valutate le prestazioni dell'ultimo modello generato durante l'addestramento, calcolando il *Mean Average Precision* sul dataset di addestramento e su quello di test. Entrambi i modelli sono stati addestrati con un *batch size* pari a 1.

Le prestazioni dei modelli sui due dataset utilizzati sono pressoché identiche, ma il modello basato su *ResNet 50* risulta leggermente più veloce durante la fase di inferenza, come si può osservare dalle specifiche fornite dagli autori [41], per

questo motivo risulta preferibile tra i due.

SSD Mobilenet v1							
	Mean Average Precision						
Durata (min)	0	20	40	60	80	100	120
Train set	0.008	0.195	0.222	0.267	0.233	0.332	0.458
Test set	0.035	0.592	0.602	0.575	0.504	0.657	0.609

Tabella 5.3: Risultati ottenuti durante l'addestramento di un modello basato su *SSD* che impiega come *backbone* una rete neurale con architettura *Mobilenet v1*.

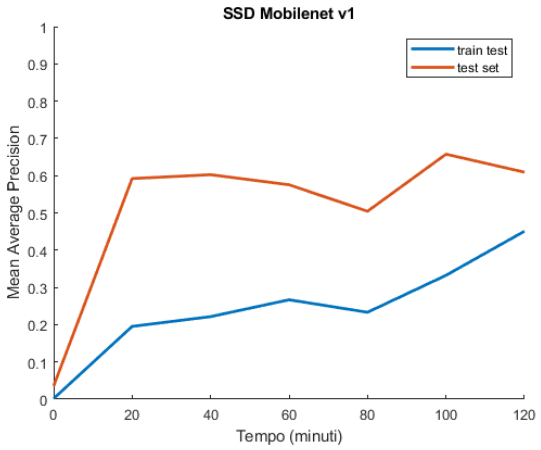


Figura 5.3: Andamento del *Mean Average Precision* durante l'addestramento di *SSD Mobilenet v1*.

La tabella 5.3 mostra i risultati ottenuti durante l'addestramento di un modello dell'algoritmo *SSD* basato su una rete neurale con architettura *Mobilenet v1*, attraverso il *framework* **TensorFlow Model Garden**. Il modello è stato addestrato con un *batch size* pari a 24. I risultati sono abbastanza singolari, poiché le prestazioni sul dataset di test sono migliori rispetto a quelle del dataset di addestramento. Si può ipotizzare che la causa di questo fenomeno sia dovuta al fatto che gli esempi contenuti nel *test set* sono più semplici rispetto a quelli di addestramento. Un algoritmo per il conteggio deve essere in grado di analizzare immagini estremamente complesse, contenente centinaia di olive, per questo motivo il modello deve essere scartato, nonostante risulti più veloce di quelli basati su *Faster R-CNN*.

La tabella 5.4 e 5.5 mostrano i risultati ottenuti durante l'addestramento di due diversi modelli dell'algoritmo *YOLO*, attraverso il *framework* **YOLO v3 Ultralytics**. In particolare, viene analizzato il modello originale di *YOLO v3* basato su *Darknet 53* e il modello *Tiny YOLO v3*, che utilizza un'architettura più semplice e leggera. Entrambi i modelli sono stati addestrati con un *batch size* pari a 8.

Tiny YOLO v3					
	Mean Average Precision				
Epoche	80	160	240	320	400
Train set	0.140	0.159	0.219	0.228	0.233
Test set	0.071	0.147	0.192	0.226	0.209

Tabella 5.4: Risultati ottenuti durante l'addestramento di un modello basato su *Tiny YOLO v3*.

YOLO v3					
	Mean Average Precision				
Epoche	80	160	240	320	400
Train set	0.673	0.699	0.785	0.790	0.793
Test set	0.319	0.394	0.403	0.409	0.411

Tabella 5.5: Risultati ottenuti durante l'addestramento di un modello basato su *YOLO v3*.

Tiny YOLO v3 è stato progettato per essere estremamente veloce e leggero, in modo che possa essere essere eseguito sui dispositivi mobili. Nonostante impieghi solamente 5 millisecondi, se eseguito su *GPU*, per analizzare un'immagine, le prestazioni sono troppo basse per poter essere impiegato nel conteggio di olive. Al contrario l'algoritmo *YOLO v3* raggiunge quasi le stesse prestazioni delle reti basate su regioni nel dataset di test, impiegando però metà del tempo. Al contrario, sul dataset di addestramento ha risultati leggermente inferiori.

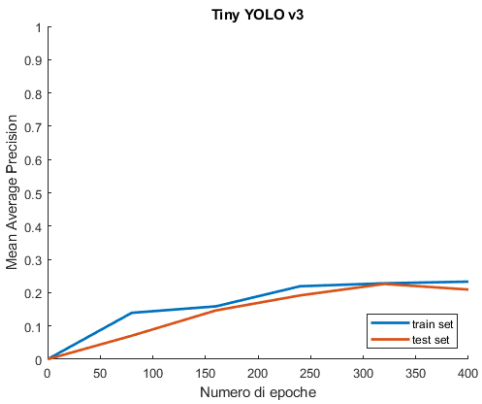


Figura 5.4: Andamento del *Mean Average Precision* durante l'addestramento di *Tiny YOLO v3*.

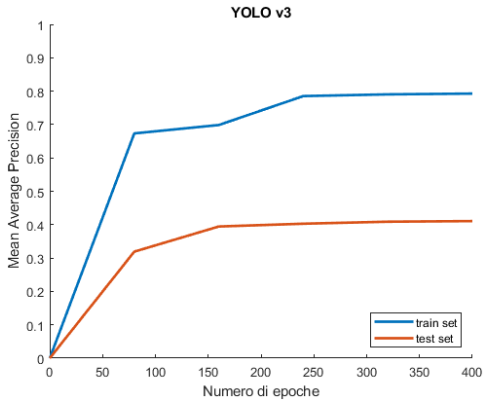


Figura 5.5: Andamento del *Mean Average Precision* durante l'addestramento di *Tiny YOLO*.

5.1.2 Limitazioni e sviluppi futuri

Tra i vari algoritmi analizzati si può concludere che quello che fornisce il miglior compromesso tra velocità e prestazioni è sicuramente *YOLO v3*. Risulta comunque opportuno testare i modelli anche su altri dataset per confermare i risultati ottenuti.

L'algoritmo descritto nel corso della tesi è sicuramente uno strumento interessante da poter applicare durante le attività di monitoraggio agricole, ma presenta comunque dei limiti. La capacità di localizzare e riconoscere le olive dipende molto dalle condizioni atmosferiche e da quelle di illuminazione. Il problema dell'illuminazione può essere risolto impiegando piccole luci sui sistemi di monitoraggio in modo da illuminare le olive e permettere un buon riconoscimento. La presenza di foglie può inoltre indurre il modello a riconoscere dei falsi positivi. A tale scopo si possono combinare i dati ottenuti dal rilevamento con letture di temperatura [3] effettuate mediante telecamere termiche.

In un futuro sviluppo dell'algoritmo l'operazione di conteggio potrebbe essere integrata con la stima del calcolo di maturazione dei frutti, in modo tale da ottenere informazioni più complete dal monitoraggio. Come già accennato, anche l'analisi del grado di maturazione richiede il riconoscimento e la localizzazione delle olive all'interno dell'albero. L'algoritmo descritto in questa tesi può dunque essere utilizzato come punto di partenza.

Bibliografia

[1] Wikipedia contributors. Precision agriculture. https://en.wikipedia.org/wiki/Precision_agriculture, 2002.

[2] MHarvey Koselka and Bret Wallach. Sensor system, method, and computer program product for plant phenotype measurement in agricultural environments, Dic 2004.

[3] Vijay Kumar, Gareth Benoit Cross, Chao Qu, Jnaneshwar Das, Anurag Makineni, and Yash Shailesh Mulgaonkar. Systems, devices, and methods for robotic remote sensing for precision agriculture, Aug 2019.

[4] Vinbot sito ufficiale.

[5] Ladybird developer awarded researcher of the year.

[6] Wikipedia contributors. Normalized difference vegetation index. https://it.wikipedia.org/wiki/Normalized_Difference_Vegetation_Index, 2012.

[7] Wikipedia contributors. Segmentazione di immagini. https://it.wikipedia.org/wiki/Segmentazione_di_immagini, 2009.

[8] Wikipedia contributors. Sogliatura. https://it.wikipedia.org/wiki/Sogliatura, 2010.

[9] Mark W. Spicer, Arno Ruckelshausen, Timur M. Dzinaj, and Andreas Linz. Sensor system, method, and computer program product for plant phenotype measurement in agricultural environments, Mar 2007.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] Satya Mallick. Image recognition and object detection : Part 1. https://www.learnopencv.com/image-recognition-and-object-detection-part1/, 2016.

[12] Wikipedia contributors. Riconoscimento dei contorni. https://it.wikipedia.org/wiki/Riconoscimento_dei_contorni, 2008.

[13] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 1:886–893 vol. 1, 2005.

[14] Francesco Scala. *Machine learning, un'introduzione dettagliata*. MIT Press, 2016.

[15] Wikipedia contributors. Indice di jaccard. https://it.wikipedia.org/wiki/Indice_di_Jaccard, 2010.

[16] Renu Khandelwal. Evaluating performance of an object detection model. https://towardsdatascience.com/evaluating-performance-of-an-object-detection-model-137a349c517b, 2020.

[17] Wikipedia contributors. Matrice di confusione. https://it.wikipedia.org/wiki/Matrice_di_confusione, 2010.

[18] Amro Kamal. Yolo, yolov2 and yolov3: All you want to know. https://medium.com/@amrokamal_47691/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899, 2019.

[19] Alessandro mazzetti. *Reti Neurali Artificiali,Introduzione ai principali modelli e simulazione su persona/ computer*. Apogeo, 1991.

[20] Wikipedia contributors. Sistema nervoso. https://it.wikipedia.org/wiki/Sistema_nervoso, 2009.

[21] Wikipedia contributors. Percettrone. https://it.wikipedia.org/wiki/Percettrone, 2006.

[22] Wikipedia contributors. Linear separability. https://en.wikipedia.org/wiki/Linear_separability, 2004.

[23] Wikipedia contributors. Cognitrone. https://it.wikipedia.org/wiki/Cognitrone, 2019.

[24] Prabhu. Cnn architectures — lenet, alexnet, vgg, googlenet and resnet. https://medium.com/@RaghavPrabhu/cnn-architectures-lenet-alexnet-vgg-googlenet-and-resnet-7c81c017b848, 2018.

[25] Wikipedia contributors. Apprendimento profondo. https://it.wikipedia.org/wiki/Apprendimento_profondo, 2012.

[26] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[27] Rete neurale. https://www.okpedia.it/rete-neurale, 2014.

[28] Wikipedia contributors. Rete neurale feed-forward. https://it.wikipedia.org/wiki/Rete_neurale_feed-forward, 2016.

[29] Rohan Kapur. Rohan lenny 1: Neural networks the backpropagation algorithm, explained. https://ayearofai.com/rohan-lenny-1-neural-networks-the-backpropagation-algorithm-explained-abf4609d4f9d, 2016.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[31] Wikipedia contributors. Problema della scomparsa del gradiente. https://it.wikipedia.org/wiki/Problema_della_scomparsa_del_gradiente, 2019.

[32] Wikipedia contributors. Batch normalization. https://en.wikipedia.org/wiki/Batch_normalization, 2018.

[33] Jasper Uijlings, K. Sande, T. Gevers, and Arnold Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171, 09 2013.

[34] Pulkit Sharma. A step by-step introduction to the basic object detection algorithms. https://www.analyticsvidhya.com/blog/2018/10/a-step-by-step-introduction-to-the-basic-object-detection-algorithms-part-1/, 2018.

[35] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[36] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[37] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[38] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[39] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[40] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[41] Tensorflow 1 detection model zoo.

[42] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei Fei Li. Imagenet: a large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 06 2009.

Elenco delle figure

Elenco delle tabelle