DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Università degli studi di Padova - Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Elettronica

Relazione per la prova finale:

Digitalizzazione di Scacchiere Bidimensionali: Studio delle Tecniche di Computer Vision per la Progettazione e Realizzazione di un'Applicazione Android

Relatore: Prof. Carlo Fantozzi

Laureando: Lavarini Angelo
Matricola n.:2066542

Padova, 13/3/2026

PANORAMICA

Lavarini Angelo

INTRODUZIONE

Lavarini Angelo

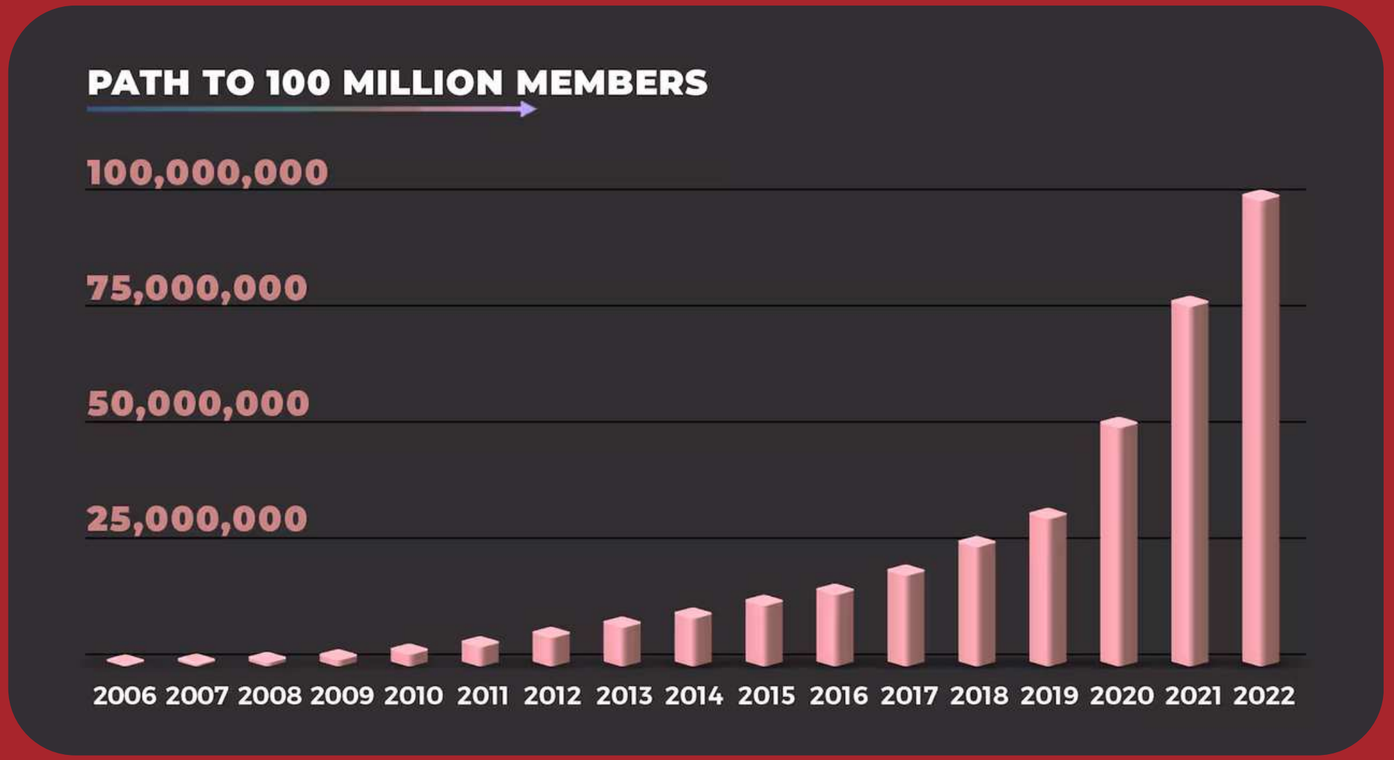


Figura 1: Crescita (in milioni) di membri sulla piattaforma di chess.com dal 2006 all 2022 (Report da Chess.com)

INTRODUZIONE

Lavarini Angelo

Problema:

necessità di analizzare le posizioni sulla scacchiera in tempo reale

Soluzione:

- Scattare una foto a uno schermo
- Digitalizzare la scacchiera
- Fornire l'analisi tramite un motore scacchistico open source

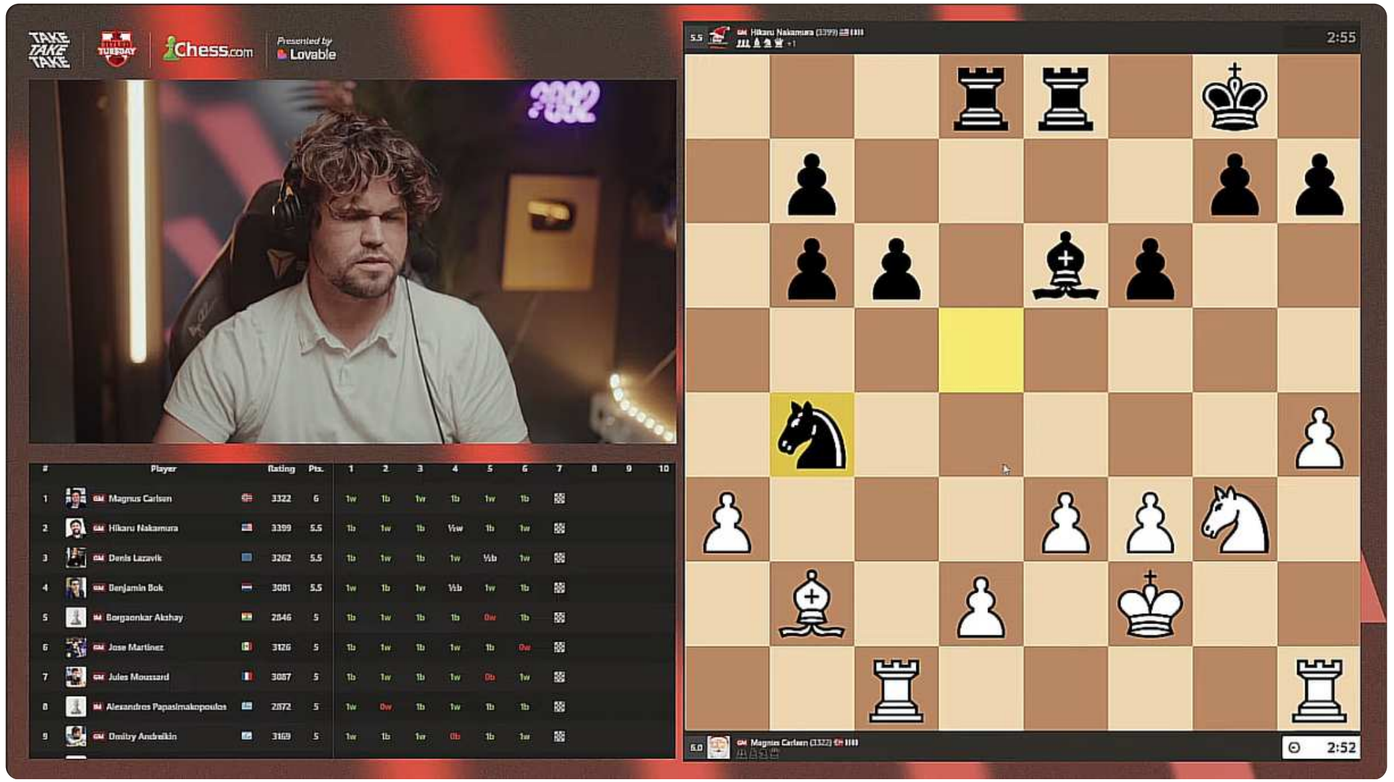


Figura 2: Streaming del canale YouTube “TakeTakeTake”

PREPARAZIONE

Lavarini Angelo

Ricerca sullo stato dell'arte

+

Image Acquisition

→

Preprocessing:
- Filtraggio
- Contrast Enhancement
- Interpolazione
- Correzione prospettica

↓

Image segmentation:

Feature Detection:
- Edge Detection

Feature Extraction:
- Trasformazione di Hough

←

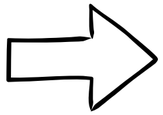
Object Detection + Classification
- Modelli TFLite
- Modelli Yolo

PROGETTAZIONE E REALIZZAZIONE

Lavarini Angelo

Struttura del progetto:

1.Localizzazione



2.Riconoscimento

Rilevamento degli spigoli della scacchiera + rettificazione prospettica

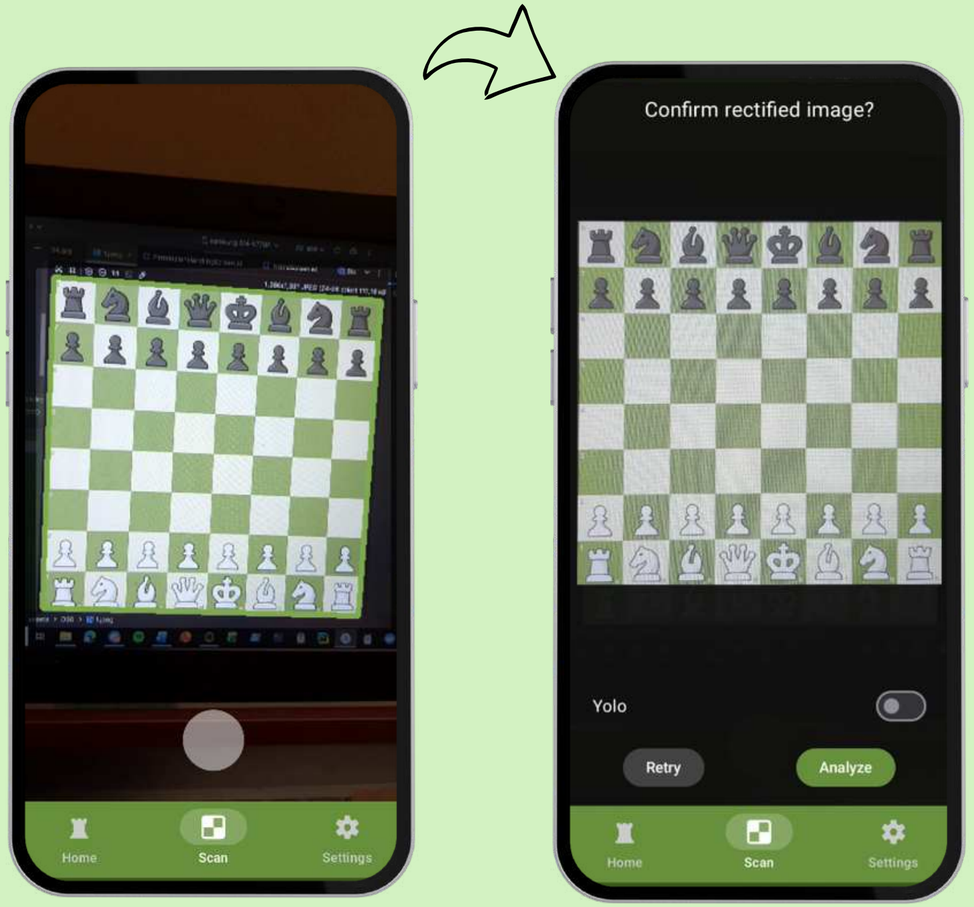
Rilevamento e riconoscimento della posizione, tipologia e colore dei pezzi

PROGETTAZIONE E REALIZZAZIONE

Lavarini Angelo

1.Localizzazione

1.1 Pre-processing
1.2 Identificazione bordi scacchiera
1.3 Rilevamento griglia 9x9
1.4 Estrazione posizione spigoli
1.5 Rettificazione prospettica



PROGETTAZIONE E REALIZZAZIONE

Lavarini Angelo

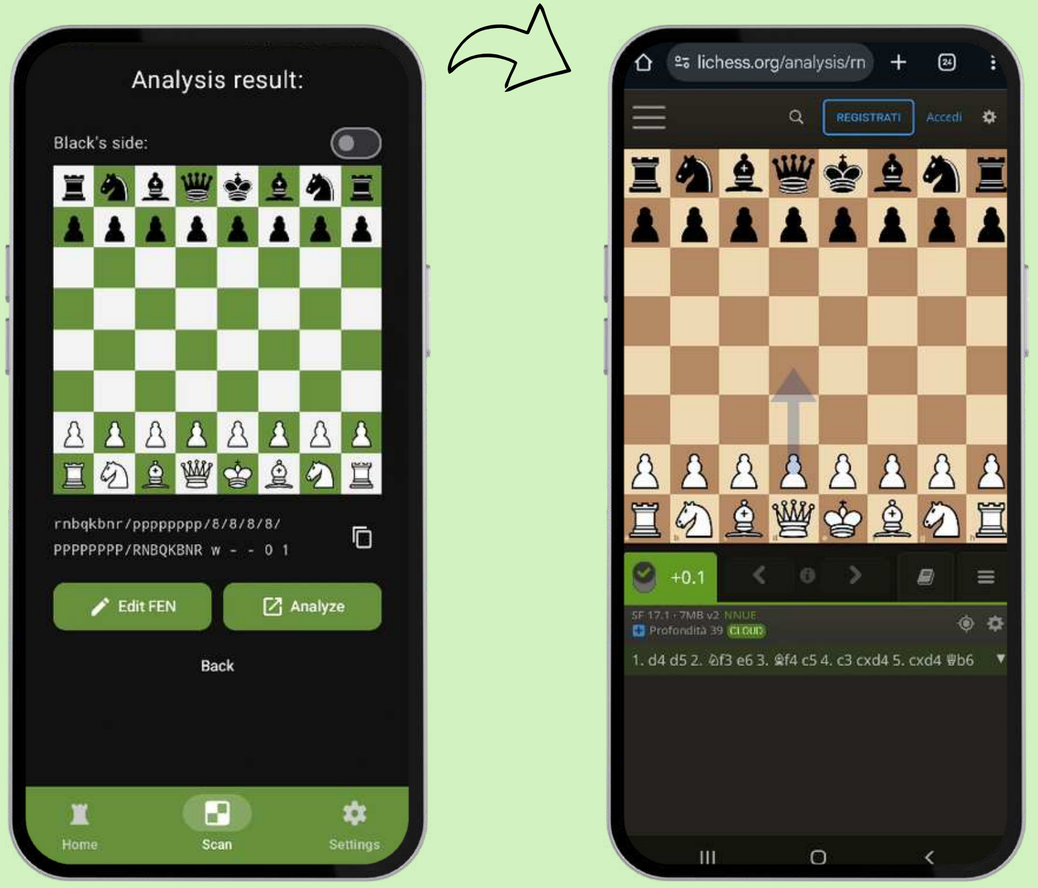
2.Riconoscimento

Due diversi approcci:

⇨ Riconoscitore A
(TFLite + algoritmo)

⇨ Riconoscitore B
(Yolo)

Vatsalparsaniya, Yolo
Model Github project

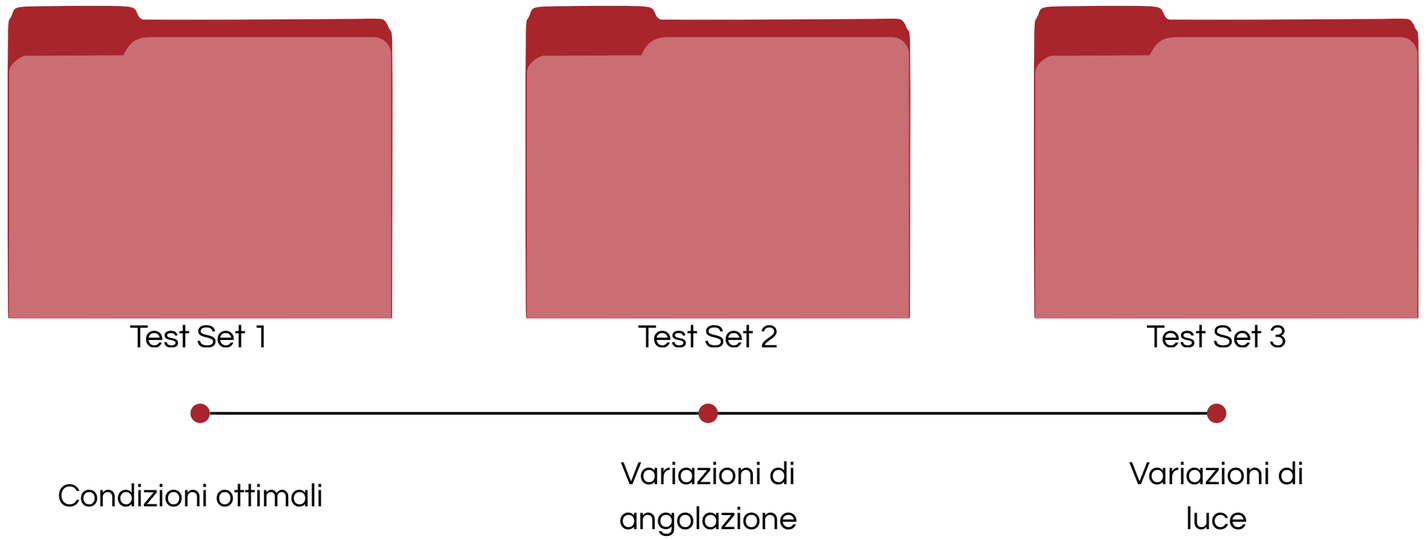


RISULTATI SPERIMENTALI

Lavarini Angelo

Metriche di valutazione:

- Accuratezza media per quadrato
- Exact Match Rate (EMR)
- Matrici di confusione



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

RISULTATI SPERIMENTALI (CLASSIFICATORE A)

UNIVERSITÀ DEGLI STUDI DI PADOVA

Lavarini Angelo

Set di Dati	Scacchiere	Accuratezza media (%)
Directory 1	100	99.89
Directory 2	100	99.66
Directory 3	102	99.88

Set	Test	Pezzi totali	Accuratezza pezzi neri (%)	Accuratezza pezzi bianchi (%)	Accuratezza globale (%)
Directory 1	Posizioni (Generale)	1795	99.36	100.0	99.72
Directory 2	Angolazione variabile	3200	97.9	99.38	98.19
Directory 3	Luce variabile	3264	96.81	99.39	98.1

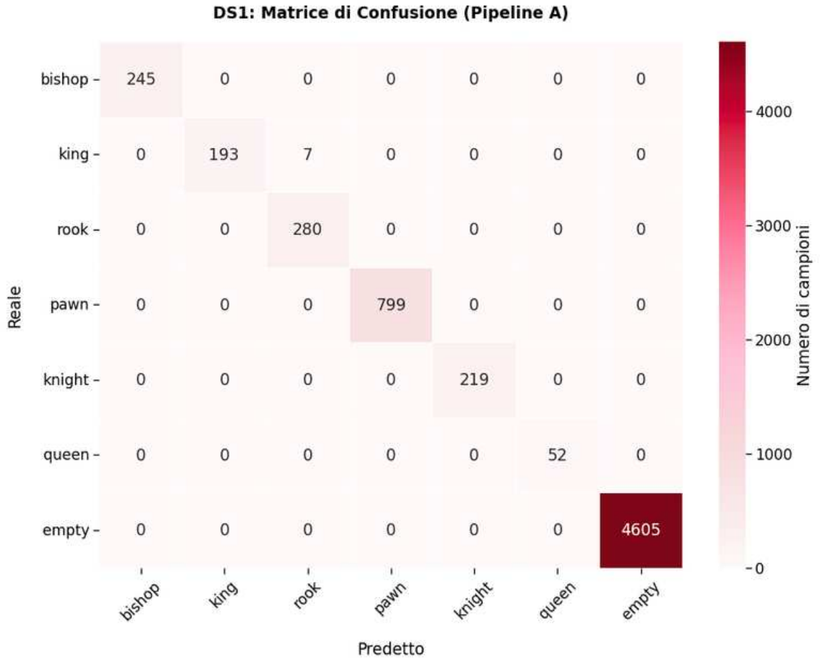
Set	EMR (%)
Directory 1	94
Directory 2	83
Directory 3	92.16

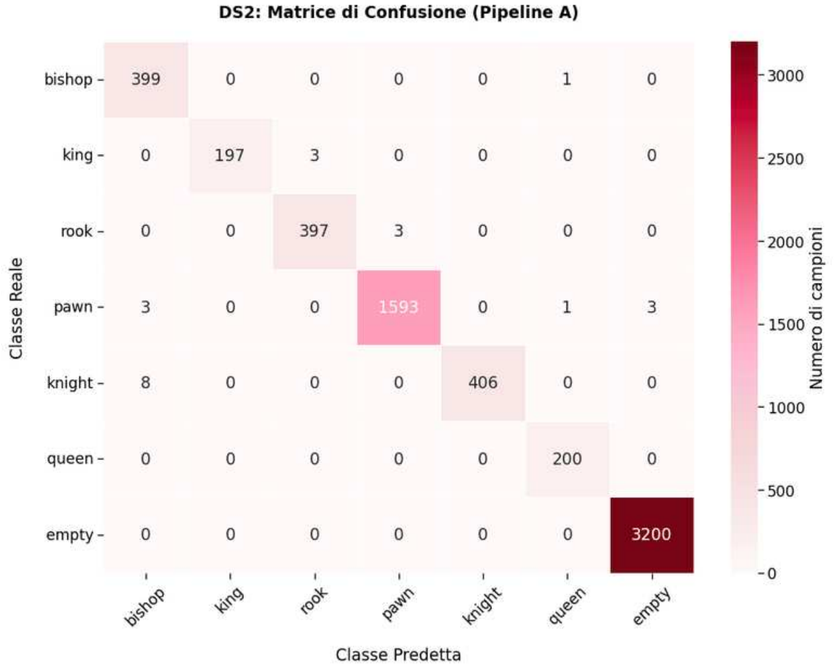
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

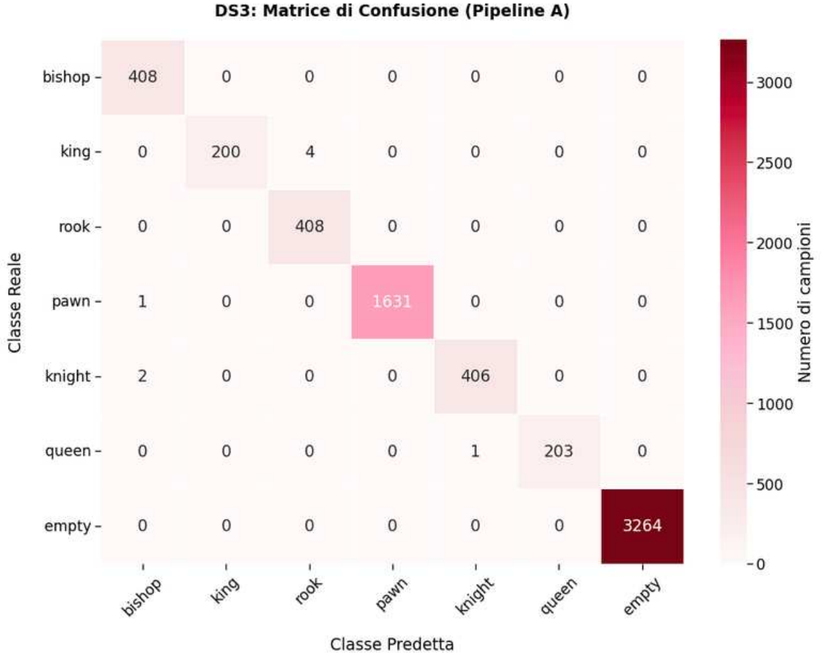
RISULTATI SPERIMENTALI (CLASSIFICATORE A)

UNIVERSITÀ DEGLI STUDI DI PADOVA

Lavarini Angelo







DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

RISULTATI SPERIMENTALI (CLASSIFICATORE B)

UNIVERSITÀ DEGLI STUDI DI PADOVA

Lavarini Angelo

Metrica	D1 (Generale)	D2 (Angolo)	D3 (Luce)
Immagini processate	100	100	102
Accuratezza media (%)	94.22	83.25	84.71
EMR (%)	10	0	0

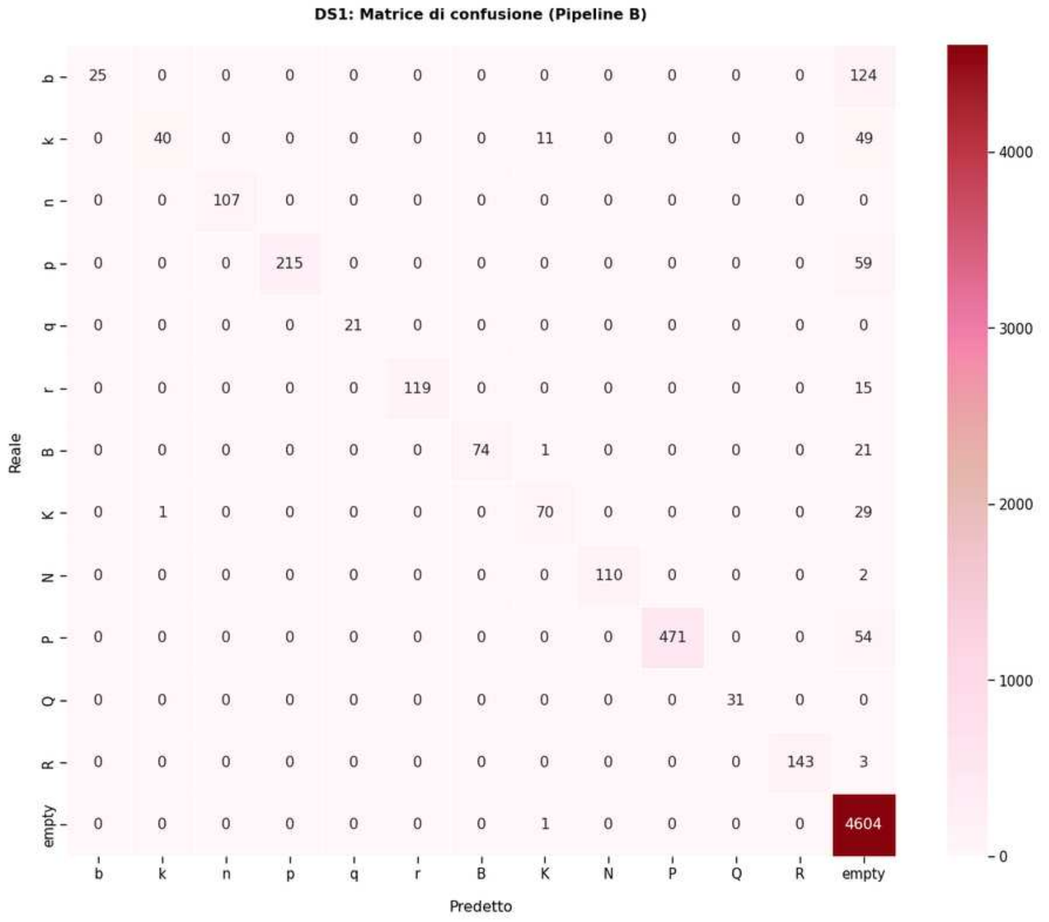


Figura 3: Partita su Chess.com

Grazie dell'attenzione

Link alla repository GitHub del mio progetto

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

LAUREA TRIENNALE IN INGEGNERIA ELETTRONICA

Digitalizzazione di Scacchiere Bidimensionali: Studio delle Tecniche di Computer Vision per la Progettazione e Realizzazione di un'Applicazione Android

Laureando

Lavarini Angelo

Matricola n. 2066542

Relatore

Fantozzi Carlo

ANNO ACCADEMICO 2025-2026

Data di laurea: 13/3/2026

Abstract

La rapida evoluzione dell'intelligenza artificiale negli ultimi anni, come per molti altri settori ha portato innovazione nel campo della Visione Artificiale: ne sono d'esempio le CNN (Convolutional Neural Networks).

Dal punto di vista ingegneristico, tuttavia, è fondamentale essere consapevoli e valutare attentamente le diverse strategie implementative disponibili prima di cimentarsi nello sviluppo di un progetto.

Questa tesi si propone di analizzare la pipeline classica di computer vision prendendola come caso di studio e, per ciascuno dei suoi blocchi funzionali, esamina le principali alternative implementative, discutendone vantaggi e limitazioni. Inoltre verranno analizzati due modelli di deep learning utili all'elaborazione del contesto dell'immagine da parte dei calcolatori: il primo basato su architettura Yolo, il secondo su MobileNet.

Le conoscenze acquisite vengono infine applicate a un problema reale: la digitalizzazione automatica di scacchiere 2D partendo da foto scattate da dispositivo mobile Android per permettere l'analisi delle posizioni sulla scacchiera. Verranno documentate e descritte le scelte progettuali adottate, illustrati i dettagli implementativi ed effettuati alcuni test volti a comparare e valutare i risultati delle varie decisioni implementative.

Indice:

Capitolo 1: Introduzione al problema e ai relativi strumenti risolutivi

Il seguente capitolo si pone l'obiettivo di introdurre il problema affrontato nella tesi e il contesto tecnologico in cui si inserisce.
Dopo aver definito il concetto di *computer vision* e aver motivato la scelta del dominio scacchistico come caso di studio, verrà presentato l'obiettivo del progetto.
Nell'ultima parte del capitolo, verrà elencata e motivata la scelta degli strumenti che sono stati utilizzati e mostrata l'interfaccia della pipeline classica della visione artificiale, ossia il caso di studio e riferimento metodologico di questa tesi.

1.1 Il Contesto: Digitalizzazione tramite Computer Vision

La visione artificiale può essere definita come *l'insieme dei processi che consentono a un calcolatore di ricavare un modello del mondo reale partendo da immagini bidimensionali.* [26]
L'obiettivo risulta quindi riuscire ad interpretare il contenuto di uno o più *frame* in modo da permettere ad un sistema di prendere decisioni automatiche.

Per raggiungere questo scopo, un sistema di visione integra componenti ottiche, elettroniche, algoritmi e/o modelli di intelligenza artificiale che collaborano per acquisire, digitalizzare ed elaborare immagini.

La parte relativa all'elaborazione di immagini è composta da una serie di algoritmi specifici, il cui compito è evidenziare e riconoscere determinate caratteristiche (come contorni, forme o strutture) al fine di eseguire controlli, classificazioni o selezioni: in base ai risultati dell'elaborazione, il sistema è quindi in grado di prendere decisioni basandosi sui dati estratti dall'immagine.

Nella maggior parte dei sistemi moderni inoltre viene introdotto un modello di Deep learning volto ad eseguire compiti di classificazione delle immagini di un certo tipo. Alcune delle applicazioni più comuni a cui si dedicano modelli di questo tipo sono [16]:

- **la classificazione delle immagini:** ossia la capacità di prevedere con accuratezza l'appartenenza di una determinata immagine a una determinata categoria
- **rilevamento degli oggetti:** combina classificazione e localizzazione per individuare la presenza di elementi di interesse all'interno di una scena

- **tracciamento degli oggetti (Motion capture):** segue o traccia un oggetto dopo che è stato rilevato all'interno di un video

1.2 Idea e Obiettivo del Progetto

Tra le molte applicazioni della computer vision, un caso particolarmente interessante è quello della digitalizzazione di contenuti grafici, come documenti o anche scacchiere digitali.

Il gioco degli scacchi, e in particolare gli scacchi online, ha avuto un'esplosione di popolarità negli ultimi 5 anni, in parte a causa della pandemia. Streamer di scacchi come "Gotham Chess", "Hikaru Nakamura" e molti altri giocatori professionisti di scacchi attirano decine di migliaia di spettatori ad ogni streaming su piattaforme come Twitch e Youtube.

A causa di tutte queste risorse online, nasce quindi la necessità da parte degli spettatori di poter utilizzare strumenti come l'analisi automatica dei computer per poter analizzare, studiare e rivedere le posizioni che vengono raggiunte dai loro content creator preferiti sulla scacchiera. Aprire manualmente software online per l'analisi e ricreare le posizioni della scacchiera a mano tuttavia è un processo tedioso e molto lento. [21] [22]

L'idea di questo progetto nasce proprio da questo problema. Utilizzando una serie di algoritmi di Computer Vision l'obiettivo è quello di permettere la seguente User Experience:

1. Scattare una foto inquadrando una scacchiera digitale visualizzata su uno schermo (tipica di applicazioni di scacchi online come Chess.com o Lichess)
2. Ricevere dall'immagine una rappresentazione della scacchiera in formato digitale sulla propria applicazione
3. Poter analizzare la posizione della scacchiera estratta dalla pipeline di computer vision molto velocemente attraverso un motore scacchistico Open Source online

Da un punto di vista accademico, questa funzionalità rappresenta un caso di studio ideale per studiare ed implementare algoritmi di computer vision. Il riconoscimento di una scacchiera a partire da un'immagine è stato studiato per molto tempo e ancora oggi ha offerto delle sfide ingegneristiche, in quanto un algoritmo infallibile in ogni contesto per questo compito ancora non esiste. [1]

1.3 Architettura della Pipeline Proposta

Date le premesse dei punti precedenti, per realizzare questo progetto è necessario definire e motivare gli strumenti e le metodologie che verranno adottati.
A questo scopo, la Sezione 1.3 è organizzata in tre parti:

- la sezione 1.3.1 presenta la pipeline classica della computer vision, ossia il modello teorico di riferimento composto dai blocchi funzionali comunemente utilizzati nei sistemi di elaborazione delle immagini
- la sezione 1.3.2 mostra invece come tale pipeline teorica venga adattata e declinata nel contesto specifico di questo progetto, illustrando la struttura logica della catena di elaborazione che verrà implementata nei capitoli successivi
- la sezione 1.3.3 organizza gli strumenti utilizzati per realizzare il progetto

1.3.1 La Pipeline classica di Computer Vision

Per implementare una pipeline efficace è necessario conoscere i blocchi fondamentali del processo, così da selezionare le tecniche più appropriate per il problema specifico.

Gli step principali sono [18]:

1. **Image Acquisition:**
   Step fondamentale; la pipeline inizia con l'acquisizione dell'immagine o del frame video tramite dispositivi appositi (come telecamere industriali, smartphone, droni o sistemi di imaging specializzati). In questa fase fattori che influenzano direttamente la qualità del dato che verrà elaborato successivamente sono:
   a. illuminazione
   b. messa a fuoco
   c. risoluzione
   d. posizione dell'oggetto
2. **Preprocessing:**
   Una volta acquisita l'immagine, si procede con operazioni di preprocessing utili a migliorare la qualità e la coerenza dei dati (esempi: riduzione del rumore dell'immagine, resizing..).
3. **Image Segmentation [16]:**
   La segmentazione consente di isolare e delimitare regioni significative all'interno dell'immagine. Le tecniche utilizzate possono spaziare da approcci classici, come thresholding o edge detection, fino a metodi avanzati basati su deep learning.
   a. **Feature Detection:**
      Questa fase prevede l'identificazione dei punti o delle regioni di interesse all'interno dell'immagine, come angoli e bordi.
   b. **Feature extraction:**
      A partire dalle features identificate si calcolano dei descrittori matematici che rappresentano l'aspetto locale di quelle aree. Questi descrittori "estraggono" l'informazione essenziale dai pixel grezzi.
4. **Object Detection, Recognition e Classification:**
   L'obiettivo di questa fase è localizzare gli oggetti di interesse all'interno dell'immagine (Detection). Una volta individuati, gli oggetti vengono riconosciuti e classificati in categorie specifiche. (Recognition e Classification)
5. **Post-Processing:**
   Questa fase comprende le operazioni di filtraggio e consolidamento dei risultati. Un esempio è la *non-maximum suppression*, algoritmo utile per eliminare rilevamenti

ridondanti data la definizione di soglie di confidenza minime durante la rilevazione degli oggetti.

6. **Visualization e Interpretation:**
   Infine, i risultati possono venire visualizzati ed interpretati tramite rappresentazioni grafiche (bounding box, maschere o keypoint) per dare un feedback all'utente dei risultati della pipeline.

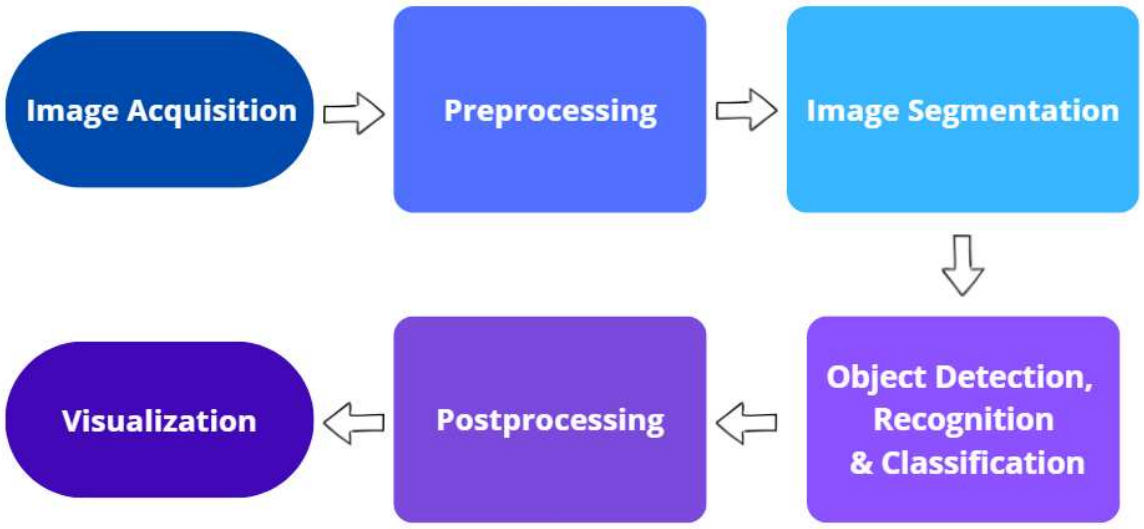


Figura 1: Semplice diagramma di flusso della pipeline classica di Computer Vision

1.3.2 Presentazione concettuale della catena di elaborazione delle immagini

Date le informazioni del punto precedente possiamo già riorganizzare una sequenza che descriverà gli aspetti principali della nostra pipeline utilizzando i blocchi funzionali della pipeline di visione artificiale classica vista al punto precedente.

Di seguito verrà presentata un'astrazione della pipeline funzionale che verrà poi effettivamente esplorata nei dettagli all'interno del capitolo 4. La pipeline esecutiva si divide in due macro-parti:

- *Localizzazione e rettifica:* Il sistema deve riconoscere la scacchiera all'interno dell'immagine, calcolare la geometria e correggere la prospettiva per ottenere una vista "dall'alto" (top-down), eliminando tutto ciò che è esterno al gioco.
- *Riconoscimento dei pezzi:* Sull'immagine rettificata, il sistema deve identificare la tipologia e la posizione di ogni pezzo per ricostruire lo stato del gioco.

Localizzazione e rettifica

a) Acquisizione e feedback in real-time**:** Tramite il dispositivo Android, l'utente inquadra la scacchiera. In questa fase, il sistema elabora il flusso video (preview) convertendo i frame in un formato idoneo per l'elaborazione.
b) Feature Detection (Rilevamento Bordi): Sul flusso video vengono applicati algoritmi per rilevare le linee che compongono la griglia della scacchiera.
c) Feature Extraction (Estrazione Coordinate): Un algoritmo dedicato analizza le linee rilevate per validare la presenza di una scacchiera completa ed estrarre le coordinate dei quattro angoli esterni.
d) Visualization: Se la scacchiera viene rilevata con successo, il sistema disegna un overlay grafico direttamente sulla preview, guidando l'utente verso uno scatto ottimale.
e) Rettifica Prospettica (Post-Scatto): Al momento dello scatto, l'immagine ad alta risoluzione viene acquisita. Utilizzando le coordinate degli angoli rilevati, si applica una trasformazione geometrica per "raddrizzare" la porzione di immagine compresa tra i 4 spigoli della scacchiera individuati, ottenendo così una vista ortogonale perfetta della sola scacchiera.

Riconoscimento dei pezzi

Object Detection, Recognition and Classification: L'immagine rettificata viene data in pasto a un modello di Deep Learning. Come verrà approfondito nel Capitolo 4, sono stati esplorati due approcci:

- Pipeline A: Un approccio ibrido con segmentazione delle caselle e classificazione tramite CNN (MobileNet).
- Pipeline B: Un approccio "end-to-end" basato su architettura YOLO (You Only Look Once) applicato all'intera scacchiera rettificata.

Una volta classificate le 64 caselle, il sistema traduce la matrice visiva in una stringa FEN (Forsyth-Edwards Notation); la stringa FEN viene infine utilizzata per generare un URL che apre la posizione scansionata direttamente su motori di analisi esterno.

Nei capitoli successivi tali blocchi verranno approfonditi. In particolare il capitolo 2 coprirà le tecniche di pre-processing, image segmentation ed extraction, mentre il capitolo 3 presenterà Object detection, recognition, classification e post processing dell'immagine attraverso l'analisi di 2 modelli di deep learning.

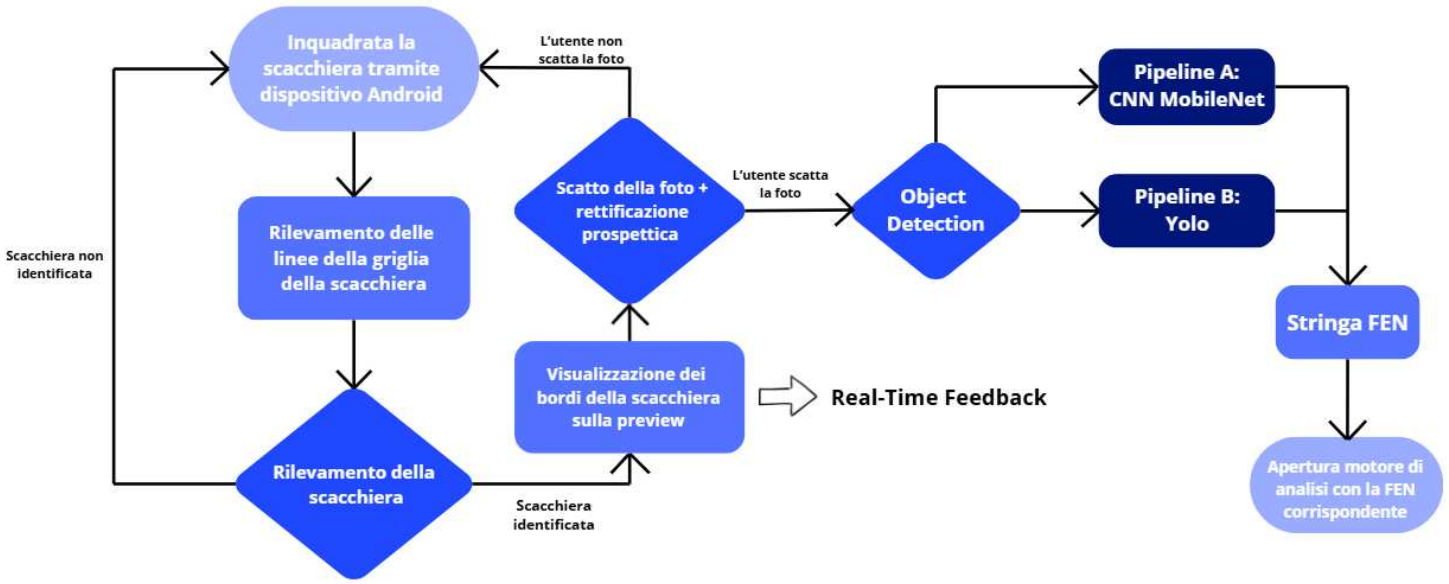


Figura 2: Diagramma a blocchi esemplificativo dell'applicazione

1.3.3 Piattaforma e Strumenti

- *Piattaforma di sviluppo:* Android. Al fine di rendere il software il più versatile e facilmente utilizzabile, lo sviluppo dell'applicazione in ambiente mobile risulta ideale.

- *Linguaggio utilizzato:* Kotlin + toolkit Jetpack Compose per costruire le UI. Rappresenta lo standard moderno consigliato per programmare in ambiente Android.

- *Gestione del codice:* GitHub, piattaforma online basata su Git che fornisce un servizio di hosting per progetti software.

- *Librerie utilizzate all'interno del progetto di maggior interesse:*
  - CameraX [11]: È stata scelta come API di gestione hardware al posto del modulo camera nativo di OpenCV per due ragioni critiche: la compatibilità e l'integrità del ciclo di vita. OpenCV presenta spesso limitazioni nel gestire correttamente i driver della fotocamera su diversi produttori Android. CameraX, essendo una libreria di sistema Google Jetpack, garantisce un comportamento uniforme e si integra nativamente con il toolkit Jetpack Compose, utilizzato per lo sviluppo dell'applicazione.
    La libreria inoltre (a differenza di *Camera* e *Camera2*) gestisce autonomamente l'apertura e la chiusura del sensore in base allo stato dell'applicazione, prevenendo sprechi di risorse e crash improvvisi (memory leak). Il vantaggio tecnico principale risiede però nell'utilizzo del caso d'uso ImageAnalysis: questo permette di intercettare il flusso video in tempo reale, estrarre i frame e inviarli alla pipeline di elaborazione senza interrompere la fluidità della preview mostrata all'utente.

  - OpenCV [3]: Utilizzata come motore primario per l'elaborazione d'immagine "classica". È stata scelta in quanto standard *de facto* nel settore della Computer Vision. Nel contesto di questa tesi, OpenCV è stata preferita per tre motivi principali:
    - Efficienza computazionale: La libreria è scritta in C/C++ ed espone interfacce Java/Kotlin tramite JNI (Java Native Interface), permettendo di eseguire operazioni matematiche complesse su matrici di pixel con prestazioni vicine al codice nativo.

- Modularità: Fornisce strumenti pronti all'uso per ogni fase della pipeline, dal filtraggio spaziale alla geometria proiettiva , essenziali per la rettifica della scacchiera.

L'unico svantaggio dell''integrazione di CameraX con OpenCV risiede nella gestione dei dati grezzi: la fotocamera di Android produce nativamente frame in formato YUV (Luma e Crominanza) mentre la libreria OpenCV lavora prevalentemente con matrici in formato RGBA. È stato quindi implementato un convertitore dedicato per trasformare i buffer ImageProxy (YUV) in oggetti Mat (OpenCV), operazione necessaria per poter applicare i metodi della libreria OpenCV.

TensorFlow Lite: libreria utile per utilizzare modelli CNN molto leggeri facilmente per permettere compiti di elaborazione più complessi sulle immagini.

❖ *Strumenti online:*

- ➢ Roboflow: interfaccia web sulla quale si possono condividere, scaricare ed utilizzare dataset per l'addestramento di modelli di deep learning.
- ➢ Teachable Machine: è un framework basato su web sviluppato da Google Creative Lab che semplifica radicalmente il processo di creazione di modelli di Machine Learning personalizzati. Lo strumento permette di addestrare classificatori di immagini, suoni o posture senza la necessità di scrivere codice complesso o gestire manualmente l'architettura della rete neurale.
  Nel contesto di questo progetto, Teachable Machine è stato utilizzato per generare modelli leggeri in formato TensorFlowLite.
- ➢ Chess.com/Lichess: piattaforme per giocare a scacchi online dalle quali ho estratto le immagini delle scacchiere per effettuare i test sulla mia implementazione.
- ➢ Canva: strumento online per la generazione di molte delle grafiche mostrate in questa tesi.
- ➢ Gemini: strumento di intelligenza artificiale utilizzato per aiutarmi a scrivere più velocemente il codice, fare debug e generare assets per l'applicazione. Questa IA inoltre è stata usata per aiutarmi a perfezionare il testo e per velocizzare la ricerca delle fonti utili da studiare.

Capitolo 2: Studio delle tecniche dal Pre-processing alla Feature Extraction

Il presente capitolo andrà ad analizzare i blocchi fondamentali della pipeline di visione artificiale dedicata alla pre-elaborazione dell'immagine, identificazione ed estrazione delle feature.

Per ciascuna fase verranno discusse le principali tecniche adottate in letteratura e nei sistemi reali, illustrandone gli aspetti teorici e i relativi casi d'uso.

2.1 Pre-processing dell'Immagine

Il pre-processing comprende l'insieme delle operazioni applicate all'immagine prima dell'elaborazione mediante algoritmi di visione artificiale più complessi.
Il suo scopo principale è migliorare la qualità e la coerenza del dato visivo, rendendo il sistema più robusto rispetto a variazioni di illuminazione, rumore, prospettiva o risoluzione.

In particolare le tecniche applicate mirano a [18] [17]:

- ridurre gli effetti del rumore;
- normalizzare il colore e migliorare il contrasto;
- evidenziare regioni di interesse;
- ridimensionare l'immagine preservando il più possibile l'informazione;
- correggere distorsioni geometriche e prospettiche.

Nei paragrafi successivi verranno analizzate le principali famiglie di tecniche.

2.2 Image denoising

Con il termine *image denoising* si indica l'insieme dei filtri e delle operazioni finalizzati alla riduzione del rumore nelle immagini. Per comprendere come questi filtri agiscano è necessario quindi conoscere almeno alcuni dei principali tipi di rumore che affliggono le immagini [2]:

1. Rumore Gaussiano
    - Rumore additivo che segue una distribuzione di probabilità normale.
    - Rende l'immagine "granulosa" , influenzando tutti i pixel in modo indipendente e uniforme.
    - Caratterizzato da media e deviazione standard, spesso originato da rumore elettronico dei sensori.

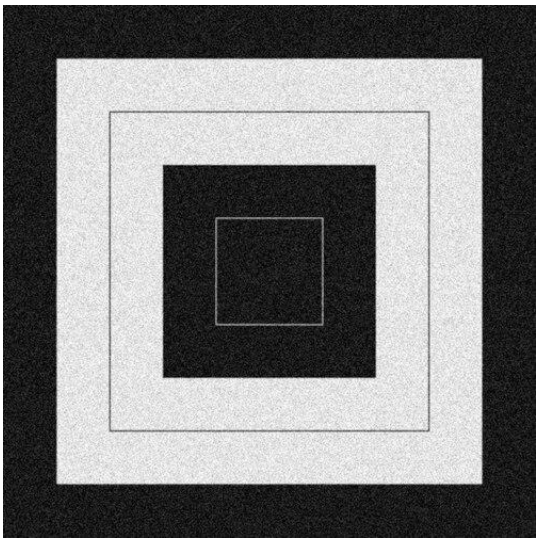


Figura 3: Rumore Gaussiano

2. Rumore Sale e Pepe
   - Rumore impulsivo che appare come pixel bianchi e neri sparsi casualmente.
   - Malfunzionamento degli elementi pixel, errori di memoria o errori del convertitore analogico-digitale.
   - Colpisce solo una piccola percentuale di pixel, lasciando gli altri inalterati.

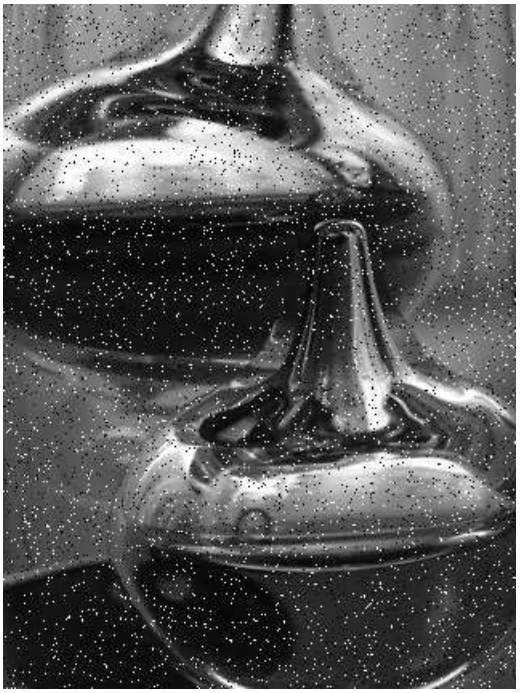


Figura 4: Rumore Salt and Pepper

3. Rumore Periodico

- Il rumore periodico è una tipologia di disturbo che si manifesta nelle immagini come un pattern ripetitivo sovrapposto al contenuto originale.
- Causa Comune: Deriva spesso da interferenze elettriche durante il processo di acquisizione dell'immagine

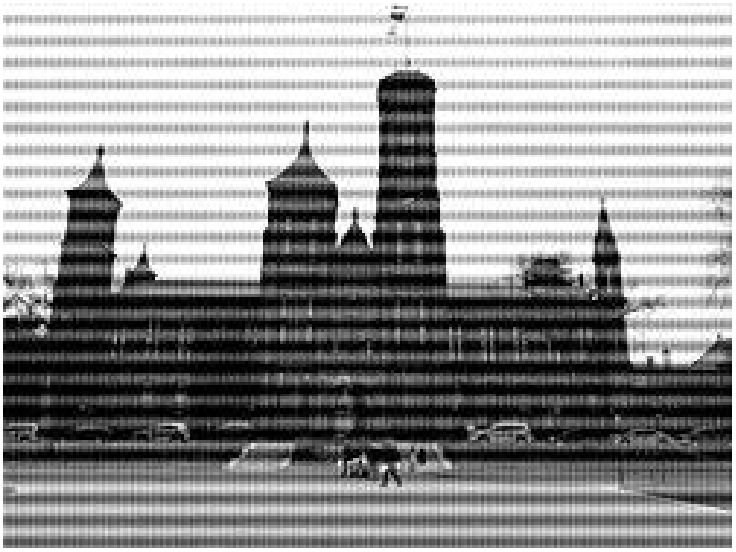


Figura 5: Rumore periodico

Per comprendere come funzionano le operazioni di pre-processing dobbiamo dare alcune definizioni [3]:

Un **Filtro** è un algoritmo che prende un'immagine di partenza $I(x, y)$ e calcola una nuova immagine $I'(x, y)$ modificando il valore di ogni pixel (x, y) in base a una funzione dei pixel circostanti.

Il **Kernel** è il modello, o *template*, che definisce due cose:

1. La forma e la dimensione dell'area circostante un pixel che deve essere considerata (es. $3\ x\ 3$, $5\ x\ 5$)
2. Come gli elementi di quell'area vengono combinati per produrre il nuovo valore del pixel

Molti kernel importanti sono kernel lineari. Un filtro è lineare se il valore assegnato al nuovo pixel (x, y) di I' può essere espresso come una somma pesata dei pixel vicini nell'immagine originale I.

L'**anchor point** indica come il kernel deve essere allineato con l'immagine sorgente.

Filtro Gaussiano

Il filtro gaussiano (o *Gaussian Blur*) è un filtro utilizzato per sfocare le immagini, e la sua efficacia deriva dal fatto che utilizza la funzione gaussiana per calcolare la trasformazione di ogni pixel.

Il Gaussian Blur è un filtro passa-basso (*low-pass filter*) perché l'immagine risultante è la Trasformata di Fourier di una Gaussiana (che è anch'essa una Gaussiana). Questo ha l'effetto di ridurre le “componenti ad alta frequenza dell'immagine” (ossia cambiamenti rapidi e improvvisi di intensità tra pixel vicini. Esempi sono i bordi netti e il rumore), ottenendo un effetto di sfocatura morbida.

Formula a due dimensioni utilizzata per la realizzazione del filtro:

$$G(x,\ y)\ =\ \frac{1}{2\ \pi\ \sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

dove:

- x e y sono le distanze dal centro del kernel
- σ è la deviazione standard, che definisce l'entità della sfocatura (maggiore σ = maggiore sfocatura)

I valori calcolati tramite la funzione G(x,y) per diverse coordinate (x,y) discrete vengono inseriti in una matrice quadrata. Il pixel originale riceve il peso maggiore, e i pesi diminuiscono simmetricamente all'aumentare della distanza. Di seguito viene fornito un esempio di kernel Gaussiano e una rappresentazione visiva più intuitiva di come questo scorra sull'immagine $I(x,\ y)$

$$\frac{1}{273} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & \underline{\mathbf{41}} & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figura 6: Kernel Gaussiano Normalizzato $5\ x\ 5$

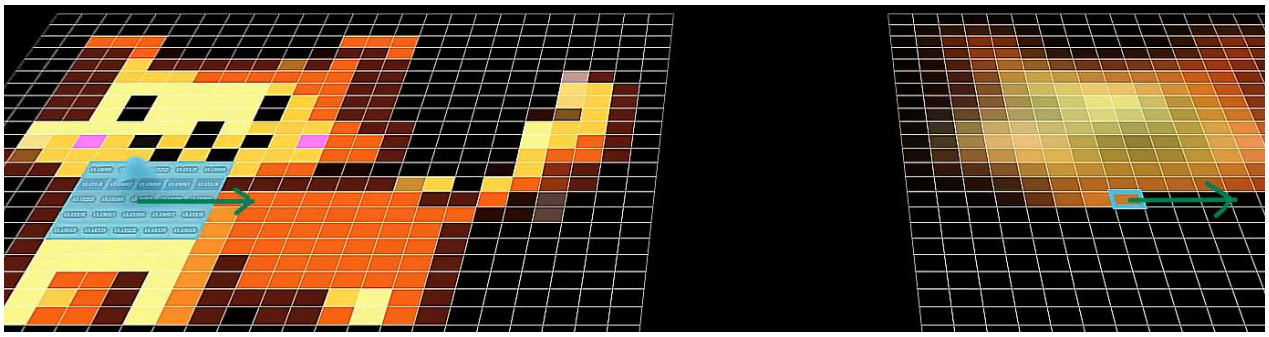


Figura 7: Visualizzazione più intuitiva di come il kernel, in questo caso gaussiano, scorra su un immagine

Figura 8: Immagine pre e post gaussian blur

Filtro Mediano

A differenza dei filtri lineari (come il Gaussian Blur), il filtro mediano sostituisce il valore di ciascun pixel con la mediana dei valori del pixel stesso e dei suoi pixel vicini. [2]
Il funzionamento risulta il seguente:

1. Si definisce la dimensione della finestra (es. $3\ x\ 3$).
2. Per ogni pixel $(x,\ y)$ nell'immagine:
    - Si raccolgono i valori dei pixel all'interno della finestra centrata su $(x,\ y)$
    - Si ordinano questi valori in ordine crescente.
    - Il pixel centrale $(x,\ y)$ viene sostituito con il valore mediano (il valore che si trova nella posizione centrale della lista ordinata).

Il filtro mediano è particolarmente efficace nel rimuovere il rumore sale e pepe e il rumore speckle.

A differenza del filtro gaussiano quindi, questo non è un filtro lineare.

Poiché la mediana seleziona un valore già esistente tra quelli vicini e ignora gli outlier (i picchi di rumore), il filtro mediano riesce a preservare i bordi (le discontinuità di intensità) meglio del filtro Gaussiano per livelli di rumore da piccoli a moderati. Questo lo rende una tipica fase di pre-elaborazione per gli algoritmi di *Edge Detection*.

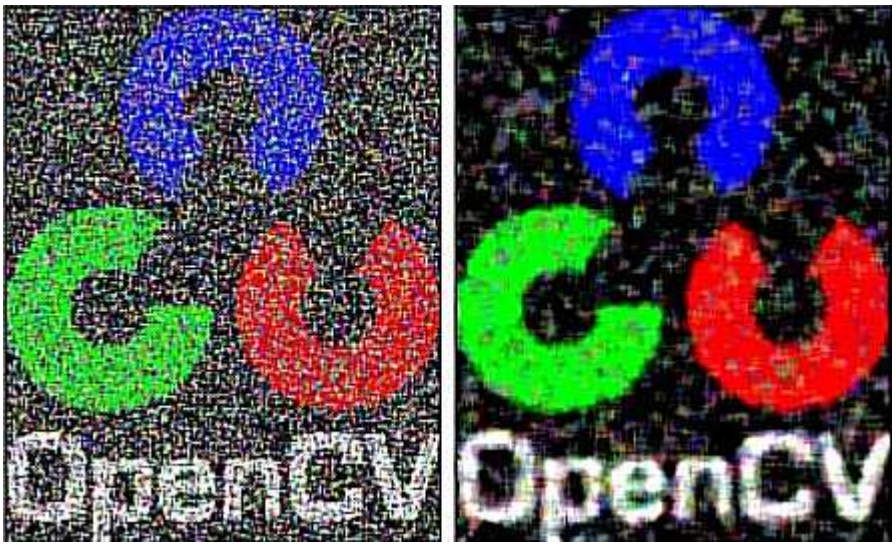


Figura 9: Immagine pre e post median blur

Filtro bilaterale

Il filtro bilaterale è un algoritmo di smussamento non lineare e con la cruciale proprietà di preservare i bordi, pur riducendo il rumore. [2]
Il filtro sostituisce l'intensità di ogni pixel con una media pesata dei pixel vicini. L'efficacia del filtro bilaterale deriva dal fatto che i pesi dipendono da una doppia condizione: la *vicinanza spaziale* e la *somiglianza radiometrica* (intensità o colore).

Da un punto di vista teorico il filtro bilaterale è definito come segue:

$$I_{filtered}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(|| I(x_i) - I(x)||) g_s(|| x_i - x||)$$

Dove W_p è il termine di normalizzazione:

$$W_p = \sum_{x_i \in \Omega} f_r(|| I(x_i) - I(x)||) g_s(|| x_i - x||)$$

- $I_{filtered}$ è l'immagine risultante
- I è l'immagine originale
- x sono le coordinate del pixel corrente
- Ω è la finestra (vicinato) centrata su x
- g_s è il **Kernel Spaziale**: Smussa in base alla distanza euclidea. Controlla quanto sono considerati i pixel lontani
- f_r è il **Kernel del Range**: Smussa in base alla differenza di intensità (*vicinanza radiometrica*). Controlla quanto sono considerati i pixel diversi per colore o luminosità

L'operazione è non lineare perché il peso totale dipende dal valore di intensità del dato $I(x)$ e non solo dalla posizione.

In termini implementativi, la distinzione cruciale tra un filtro gaussiano e uno bilaterale è che quest'ultimo non utilizza un kernel statico per tutta l'immagine.

Per ogni pixel infatti, per il calcolo dell'intensità del pixel il sistema prenderà in considerazione sia una matrice di pixel basati sulla distanza dal pixel centrale (anchor) e sia una matrice radiometrica calcolata dinamicamente basata sulla differenza di intensità tra il pixel centrale e i suoi vicini.

Questo significa che, mentre il kernel spaziale penalizza i pixel fisicamente lontani, il kernel del range penalizza i pixel 'numericamente' lontani nel dominio del colore. Il peso finale applicato a un pixel vicino è il prodotto di questi due fattori: se un pixel è vicino nello spazio ma appartiene all'altro lato di un bordo (quindi ha un colore molto diverso), il suo contributo alla media verrà annullato dal kernel del range, preservando la nitidezza del contorno.

Figura 10: Immagine risultante dall smoothing bilaterale

2.3 Contrast enhancement

Il Contrast Enhancement [2](Miglioramento del Contrasto) è una tecnica di elaborazione delle immagini utilizzata per:

- Aumentare la differenza di luminosità tra gli oggetti e lo sfondo
- Aumentare la visibilità degli oggetti nell'immagine.

Il suo scopo risulta quindi modificare e ridistribuire l'intensità dei pixel per migliorarne la visibilità. È un passaggio fondamentale in molti sistemi di visione artificiale, specialmente in condizioni di illuminazione sub-ottimali.

Gli algoritmi di miglioramento del contrasto si dividono in due categorie principali:

Algoritmi Globali: Assegnano lo stesso valore di intensità di output a tutti i pixel con lo stesso valore di input, indipendentemente dalla loro posizione nell'immagine

Algoritmi Locali: Regolano l'intensità in base alle caratteristiche dello spazio circostante di ciascun pixel. Forniscono risultati generalmente migliori

Di seguito verranno esposte due delle principali tecniche, rese disponibili anche dalla libreria OpenCV: *l'equalizzazione dell'istogramma* e *CLAHE.*

Histogram Equalization

L'Equalizzazione dell'Istogramma è un algoritmo globale che ha lo scopo di appiattire e allungare la distribuzione dei livelli di grigio in un'immagine, in modo da aumentare il contrasto complessivo. Può essere utilizzata sia su immagini a colori che in scala di grigi, tuttavia analizziamo il caso della scala di grigi come primo caso per comprenderne il funzionamento. [1] [2]

Si consideri un'immagine in scala di grigi X con L livelli di grigio (solitamente $L = 256$).

La probabilità $p_x(i)$ che un pixel scelto casualmente abbia un livello di grigio i è data dal numero di occorrenze n_i di quel livello, diviso per il numero totale di pixel n:

$$p_x(i) = \frac{n_i}{n}, \ 0 \leq i < L$$

$p_x(i)$ rappresenta il valore dell'istogramma normalizzato; l'area totale dell'istogramma è 1.

La funzione di distribuzione cumulativa è l'istogramma normalizzato accumulato ed è definita come la somma delle probabilità fino al livello i:

$$cdf_x(i) = \sum_{j=0}^{i} p_x(j)$$

L'equalizzazione mira a creare una nuova immagine Y con un *istogramma piatto* (grafico che mostra una distribuzione quasi uniforme di tutti i livelli di luminosità di un'immagine), ottenuta applicando una trasformazione T: $[0, \ L - 1] \rightarrow [0, \ L - 1]$ all'immagine originale:

La trasformazione che linearizza la CDF è definita come:

$$T(i) = cdf_x(i) = k$$

Per riportare i valori nell'intervallo di intensità originale $[0, \ L - 1]$, il valore trasformato $k = T(i)$ viene quindi riscalato in questo modo:

$$k' = ceil(L \cdot k) - 1$$

dove k' è il nuovo livello di grigio intero e $ceil$ arrotonda il numero all'intero successivo più grande.

Nell'immagine che segue possiamo osservare concretamente l'effetto della trasformazione matematica appena descritta confrontando lo stato "prima" e "dopo" il processo:

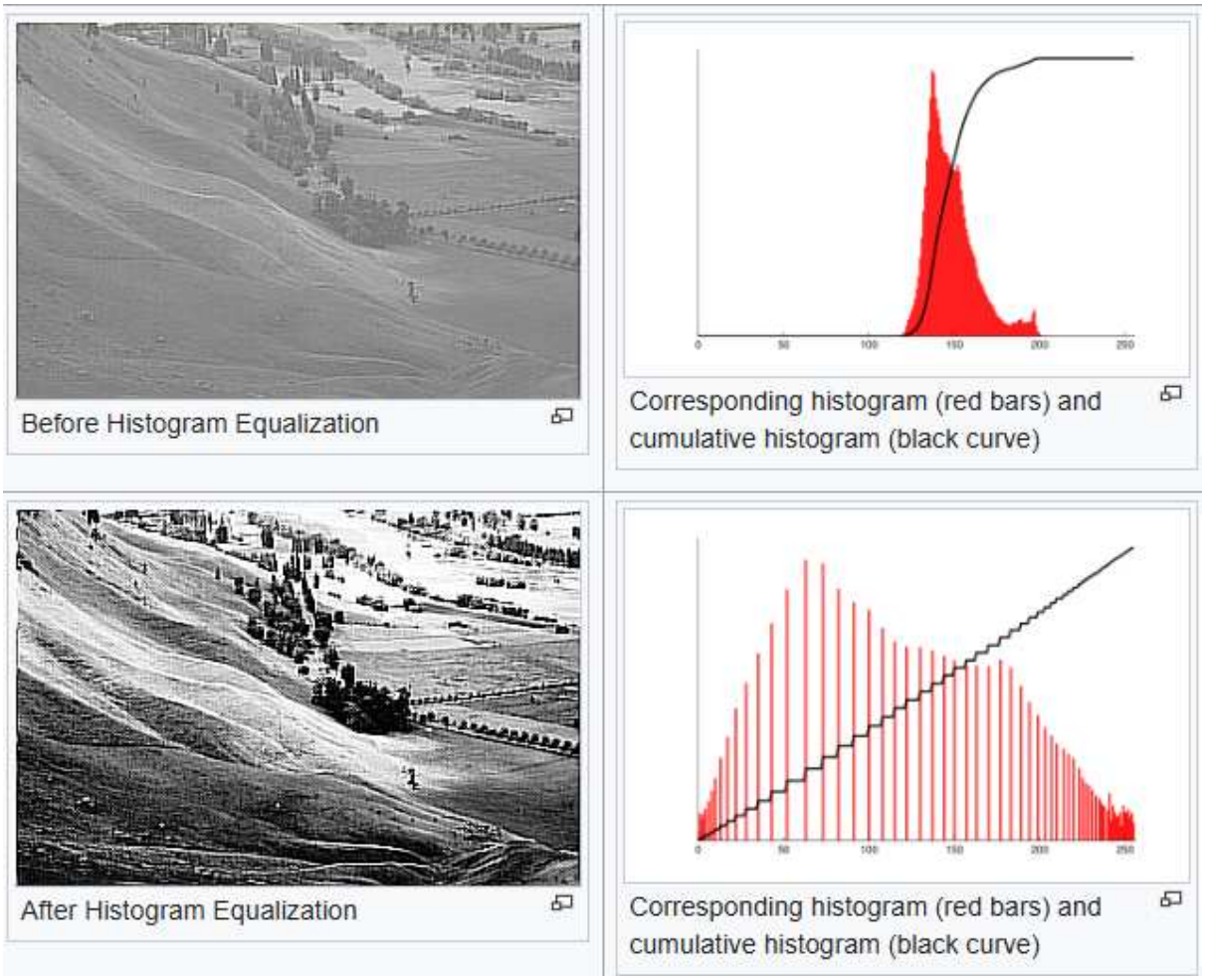


Figure 11 12 13 14: Equalizzazione dell'istogramma. N.B.: la scala dell'asse verticale dell'istogramma cumulativo è diversa da quella dell'istogramma

Prima dell'equalizzazione l'immagine originale appare "sbiadita" e povera di dettagli. Osservando il grafico corrispondente a destra, l'istogramma conferma questa percezione visiva: la quasi totalità dei pixel è concentrata in una stretta fascia di valori centrali. Di conseguenza, la gamma dinamica disponibile (da 0 a 255) è largamente inutilizzata. La curva cumulativa nera (CDF) riflette questa concentrazione, mostrando una crescita ripida solo in corrispondenza del picco dell'istogramma e rimanendo piatta altrove.

Dopo l'equalizzazione l'immagine trasformata mostra un contrasto decisamente più elevato, rendendo molto più visibili i dettagli del paesaggio e le texture del terreno. Il grafico a destra evidenzia come l'operazione abbia ridistribuito i livelli di grigio. Il risultato più significativo è visibile nella nuova curva cumulativa: ora approssima una retta diagonale costante. Questo andamento lineare indica che la probabilità cumulativa cresce in modo uniforme.

L'applicazione diretta dell'equalizzazione dell'istogramma a ciascun canale (Rosso, Verde, Blu) di un'immagine RGB è sconsigliata, poiché può causare cambiamenti drastici nell'equilibrio cromatico; è preferibile convertire l'immagine in uno spazio colore più appropriato (come YUV, approfondito in seguito nel capitolo 4) e poi applicare l'equalizzazione soltanto al *canale di luminosità*, lasciando inalterati i canali di colore.

CLAHE

Mentre l'equalizzazione dell'istogramma standard applica una trasformazione globale basata sull'intera distribuzione dei pixel, l'Adaptive Histogram Equalization adotta un approccio locale. Questo metodo è stato sviluppato per superare i limiti dell'equalizzazione globale, che spesso non riesce a migliorare sufficientemente il contrasto in immagini contenenti regioni con variazioni di luminosità più marcate.[2] [12]

Nell' AHE, l'immagine viene elaborata calcolando diversi istogrammi, ciascuno corrispondente a una distinta sezione locale dell'immagine. La trasformazione di ogni pixel deriva quindi dall'istogramma di una regione circostante (o *neighborhood*), come mostrato nella figura che illustra i riquadri di vicinato sovrapposti. Matematicamente, la funzione di trasformazione è proporzionale alla funzione di distribuzione cumulativa (CDF) dei valori dei pixel in quel vicinato, esattamente come per l'equalizzazione dell'istogramma.

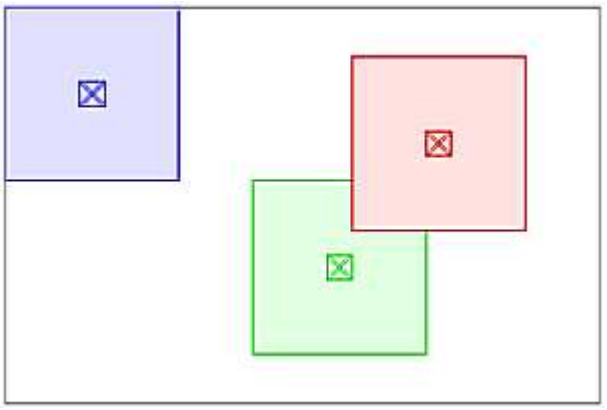


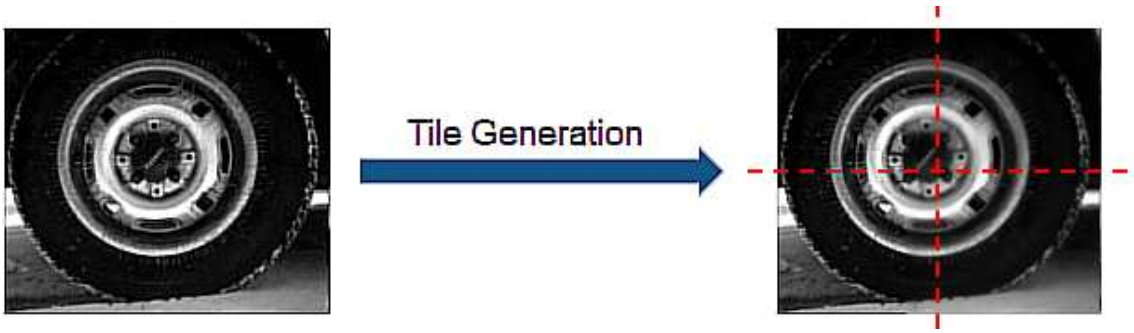
Figura 15: localizzazione di diversi istogrammi

L' AHE tuttavia presenta una criticità significativa: la tendenza a sovra-amplificare il rumore nelle regioni omogenee. In queste aree, l'istogramma è altamente concentrato in pochi livelli di grigio; di conseguenza, la pendenza della CDF diventa molto elevata, causando un'amplificazione eccessiva del contrasto che rende visibile il rumore di fondo.

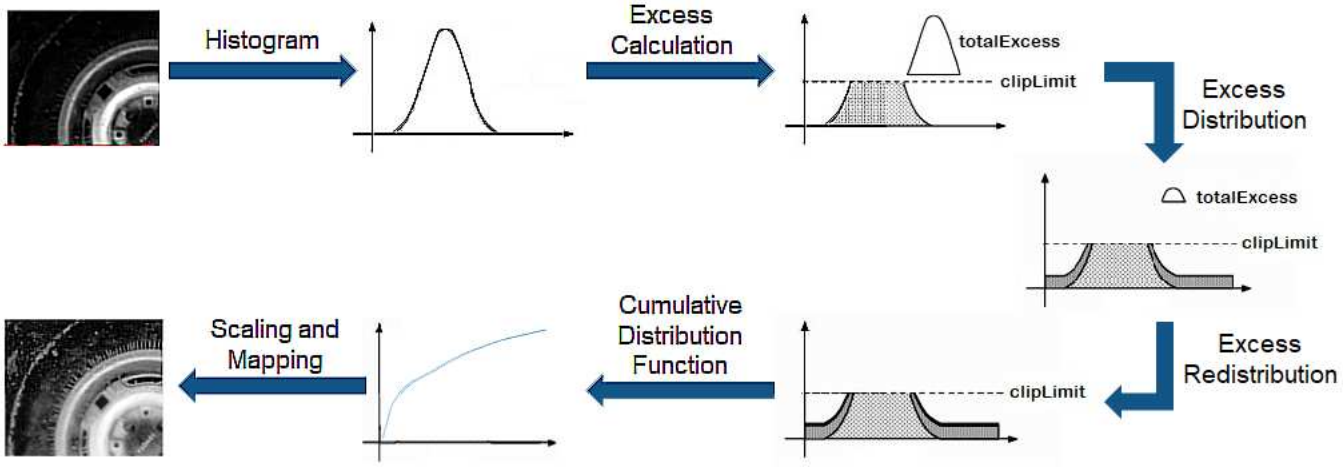
Per risolvere il problema dell'amplificazione del rumore, è stata introdotta una variante nota come Contrast Limited AHE. Questo algoritmo limita l'amplificazione del contrasto "clippando" (tagliando) l'istogramma a un valore predefinito prima di calcolare la CDF.

Questo algoritmo si articola in tre fasi principali, visibili nello schema di flusso della Figura 13 N.B. con il termine *bin* (che significa letteralmente "contenitore" o "scomparto") si intenderanno le singole barre dell'istogramma generato:

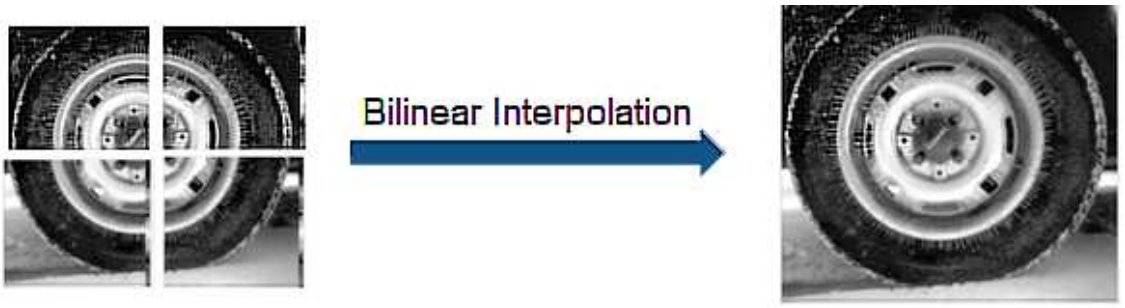
1. **[Figura 16]:** L'immagine viene suddivisa in sezioni rettangolari non sovrapposte chiamate *tiles* (tasselli)



2. **[Figura 17]:**
   - Per ogni tile viene calcolato l'istogramma.
   - Viene stabilito un *Clip Limit*
   - I valori dei *bin* dell'istogramma che superano questo limite vengono tagliati
   - La parte sopra la soglia di clip non viene scartata, ma ridistribuita uniformemente tra tutti i bin dell'istogramma
   - La seguente ridistribuzione alza il "livello del mare" dell'istogramma: se questa aggiunta porta alcuni bin a superare nuovamente il limite, la procedura può essere ripetuta ricorsivamente fino a quando l'eccesso è trascurabile
   - Una volta creata la nuova CDF, questa funge da funzione di trasformazione T: per ogni livello di grigio in ingresso i di ogni pixel dell'immagine, si consulta il valore della CDF a quel livello, ottenendo un valore k compreso tra 0 e 1; questo valore viene usato per mappare il vecchio livello di grigio i a una nuova posizione nello spettro di intensità.
   - il valore k è normalizzato (essendo una probabilità, è compreso tra 0 e 1); viene quindi riscalato per riportarlo al range dei livelli di grigio dell'immagine



3. **[Figura 18]:** Una volta calcolata la nuova mappatura per ogni tile, per evitare artefatti visivi a blocco (effetto scacchiera) ai confini tra un tile e l'altro, i pixel finali vengono generati combinando i risultati dei tile adiacenti tramite interpolazione bilineare (vedi sezione 2.1.4 Bilinear Interpolation).



Il risultato finale è un'immagine con un contrasto locale migliorato e bordi ben definiti, ma senza l'introduzione di artefatti di rumore nelle aree uniformi.

Anche per l'algoritmo CLAHE il suggerimento per l'applicazione su immagini a colori rimane lo stesso dell'histogram equalization: è preferibile applicare l'equalizzazione soltanto al *canale di luminosità*, lasciando inalterati i canali di colore.

2.4 Image segmentation

La segmentazione dell'immagine è un processo fondamentale nell'elaborazione digitale, mirato a suddividere l'immagine in regioni coerenti e significative per semplificarne l'analisi.

Le tecniche di segmentazione si dividono storicamente in due grandi classi:

1. Approcci classici di computer vision: Metodi algoritmici e matematici che si basano su proprietà locali e globali dei pixel
2. Tecniche basate sull'intelligenza artificiale: Metodi moderni che utilizzano reti neurali profonde per apprendere le feature e segmentare le immagini, dominando attualmente la ricerca

In questa sezione ci concentreremo sulle metodologie tradizionali. Queste si possono raggruppare in quattro categorie principali, ciascuna basata su un diverso criterio di partizionamento dei pixel. [16]

Segmentazione basata su una soglia (Thresholding)

Il *Thresholding* è il metodo più semplice ed efficiente, particolarmente adatto per segmentare oggetti con un netto contrasto rispetto allo sfondo. La tecnica opera dividendo i pixel in classi in base al confronto dei loro valori di intensità con un valore di soglia. Il risultato è un'immagine binaria, essenziale per algoritmi successivi come il rilevamento dei contorni. Ne esistono di due tipi:

Global Thresholding: Applica un unico valore di soglia fisso a tutta l'immagine. È rapido, ma fallisce in presenza di variazioni di illuminazione o contrasto non uniformi

Adaptive Thresholding

Il thresholding adattivo è una tecnica utilizzata nella segmentazione delle immagini per dividere un'immagine in regioni di primo piano e di sfondo regolando localmente il valore di soglia in base alle caratteristiche dell'immagine.

Il metodo prevede la selezione di un valore di soglia per ciascuna regione o blocco più piccolo, in base alle statistiche dei valori dei pixel all'interno di quel blocco. La soglia adattiva è utile per immagini con illuminazione non uniforme o contrasto variabile ed è comunemente utilizzata nella scansione di documenti, nella binarizzazione delle immagini e nella segmentazione delle immagini.

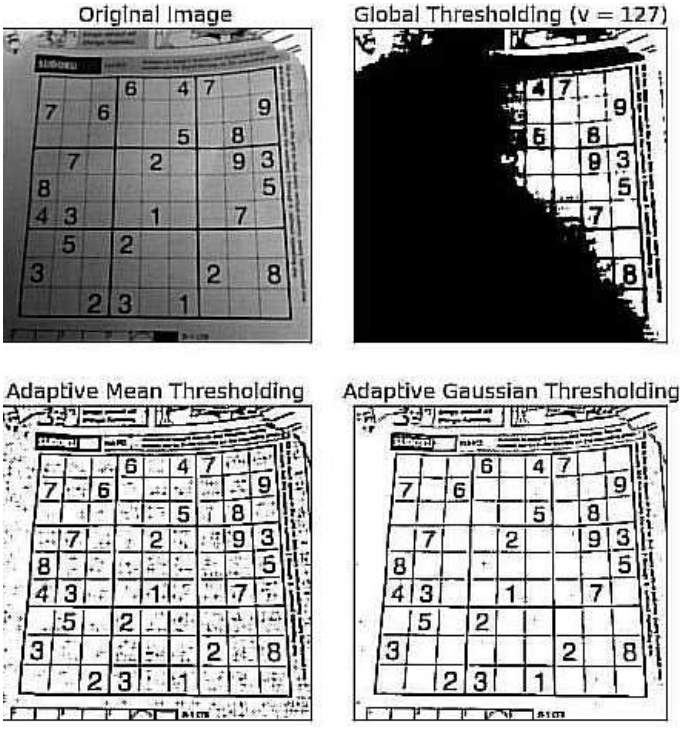


Figura 19: Global thresholding vs Adaptive thresholding

Segmentazione Basata sulla Regione (Region-based)

I metodi basati sulla regione raggruppano i pixel in segmenti omogenei in base a criteri di somiglianza (intensità, colore, *texture*).

Split and Merge [2]

È un approccio ricorsivo che inizia dividendo l'immagine in blocchi finché non soddisfano un criterio di uniformità, per poi unire le regioni adiacenti che presentano somiglianza.
L'algoritmo opera ciclicamente attraverso due fasi interdipendenti basate su un criterio di omogeneità predefinito.

1. Splitting: L'immagine, o una regione di essa, viene ricorsivamente divisa in quattro sotto-quadranti. Se non supera il test di omogeneità il nodo padre viene diviso in quattro nodi figli, e il processo continua finché ogni sotto-regione non soddisfa il criterio stabilito.

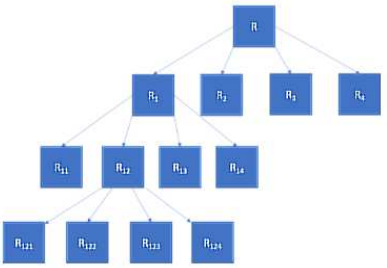


Figura 20: Quadtree split and merge

2. Merging: Una volta che tutte le regioni sono state suddivise al livello appropriato, il processo procede alla fase di fusione. Le regioni adiacenti che sono omogenee e condividono un confine vengono unite per formare un segmento più grande.

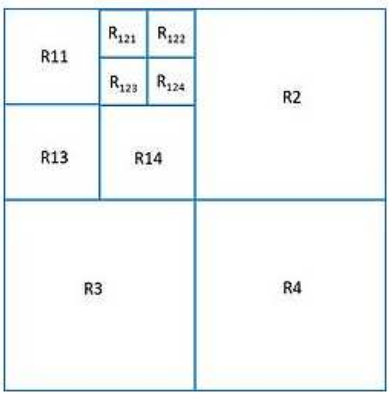


Figura 21: Immagine fusa risultante

Il ciclo continua finché non è possibile eseguire ulteriori divisioni o fusioni, lasciando l'immagine completamente partizionata in regioni omogenee e significative.

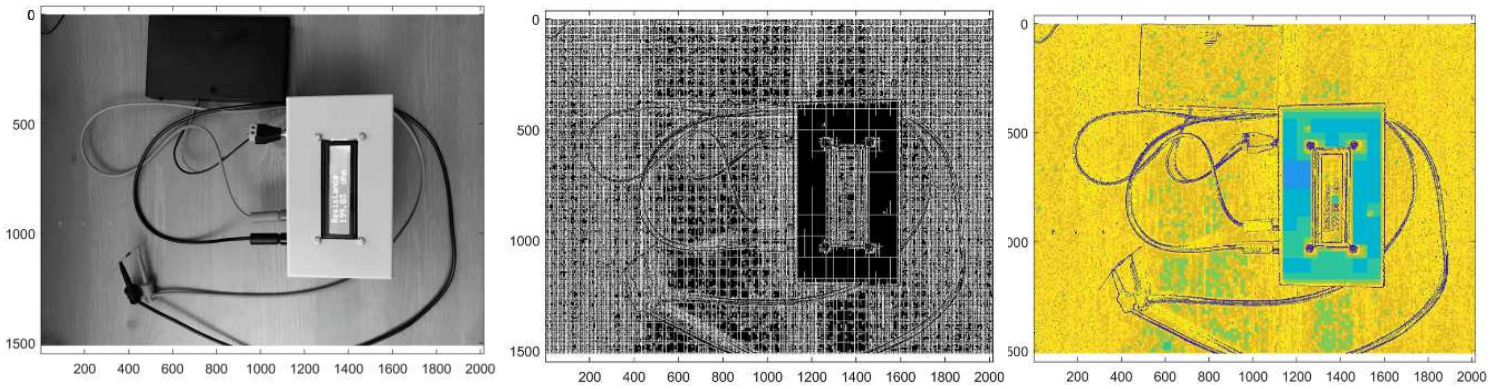


Figura 22, 23 e 24: esempio di segmentazione di un'immagine in scala di grigi il cui criterio di omogeneità è: max(regione)-min(regione) < 10. L'immagine centrale rappresenta i blocchi creati durante lo splitting dell'immagine, quella a destra il risultato finale merged.

Il criterio di omogeneità è flessibile e può basarsi su diverse proprietà. Alcune di queste sono:

Uniformità: La regione è omogenea se i livelli di grigio (o i valori di colore) al suo interno sono costanti o rientrano in un intervallo di soglia ristretto

Varianza: Un criterio comune è richiedere che la varianza del livello di grigio nella regione sia inferiore a un valore specificato

Media Locale o Globale: Si possono utilizzare relazioni tra la media di intensità della regione e la media dell'intera immagine

Graph-based Segmentation

Il GBS funziona costruendo una rappresentazione dell'immagine come un grafo non orientato: [16]

1. Ogni pixel dell'immagine è trattato come un nodo nel grafo.
2. Gli archi che connettono i nodi rappresentano la relazione di vicinanza tra i pixel.
3. Il peso di ogni arco definisce il grado di similarità o dissimilarità tra i due pixel connessi (ad esempio, la differenza nei valori di colore o intensità). Un peso basso indica alta somiglianza e continuità, mentre un peso alto indica un potenziale confine.

L'obiettivo primario del GBS è trasformare l'immagine in segmenti significativi applicando algoritmi di partizionamento. Questi algoritmi mirano a minimizzare il costo della separazione dei segmenti, ovvero minimizzare la somma dei pesi degli archi che vengono "tagliati".

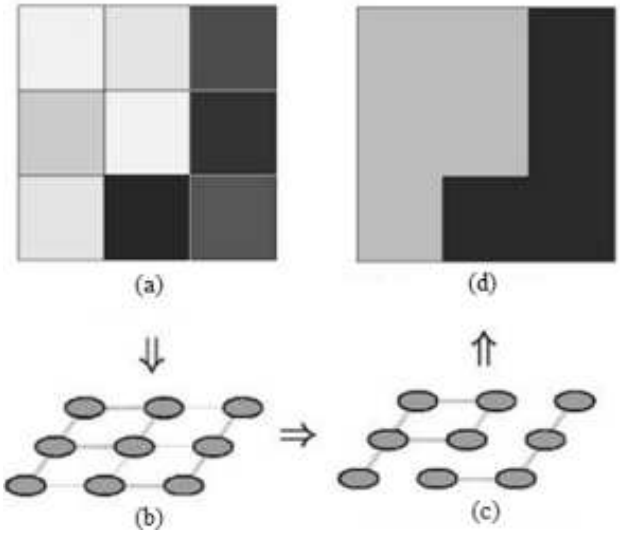


Figura 25: esempio di GBS in GrayScale

Il cuore della GBS risiede nella scelta dell'algoritmo di partizionamento, che garantisce che i segmenti risultanti siano accurati e significativi:

- Minimum Cut: È un approccio classico che mira a dividere il grafo in due segmenti rimuovendo l'insieme di archi con il peso totale minimo.

- Normalized Cut: Algoritmo più sofisticato che minimizza il rapporto tra il costo del taglio e la somma dei pesi degli archi all'interno dei segmenti stessi. Questo approccio tende a produrre partizioni più equilibrate e significative dal punto di vista visivo.
- Spectral Clustering: Si basa sull'analisi degli autovettori della matrice Laplaciana del grafo, la quale cattura le proprietà di connettività. Il *K-means clustering* viene poi applicato a questi autovettori per ottenere la segmentazione finale.

Il GBS offre una notevole flessibilità, potendo segmentare le immagini basandosi su vari criteri come colore, texture o luminosità. Tuttavia può essere computazionalmente costosa e complessa da implementare, specialmente per immagini di grandi dimensioni, richiedendo un'accurata taratura dei parametri per evitare l'*over-segmentation* (produzione di troppi segmenti piccoli).

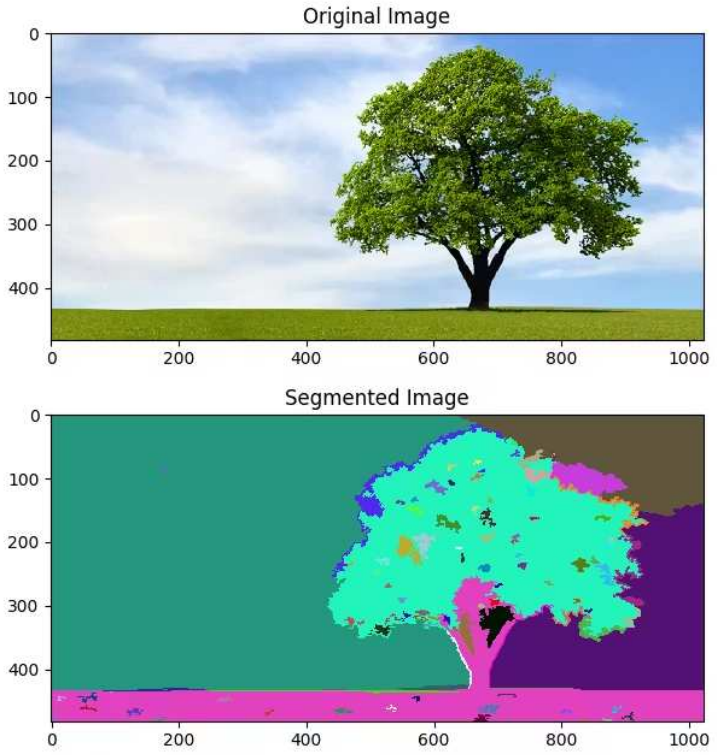


Figura 26: esempio di GBS su immagine a colori

Segmentazione Basata sul Clustering (Clustering)

A differenza degli approcci *region-based*, che impongono un vincolo di adiacenza spaziale, il Clustering raggruppa i pixel basandosi esclusivamente sulla loro distribuzione globale nel dominio dei colori o delle intensità. In questo senso, due pixel possono appartenere allo stesso segmento anche se si trovano ai lati opposti dell'immagine, purché siano "numericamente" simili.

K-means Clustering

Il clustering K-means è una tecnica di segmentazione non supervisionata che raggruppa i pixel in insiemi omogenei sulla base della loro somiglianza, trattando ciascun pixel come un punto in uno spazio delle caratteristiche (tipicamente colore, intensità o vettori multidimensionali). [2]

Questo metodo mira a suddividere un insieme di osservazioni in *K* cluster, in modo tale che ogni osservazione appartenga al cluster il cui *centroide* (= media aritmetica dei punti dati all'interno di ciascun cluster) è il più vicino.

La suddivisione dello spazio risultante forma una partizione in celle, dette di Voronoi, ognuna associata al centroide corrispondente. L'obiettivo principale dell'algoritmo è minimizzare la *within-cluster variance* o *WCSS* (Within-Cluster Sum of Squares), ovvero la somma delle distanze quadrate tra i punti e il loro centroide, rendendo i segmenti interni più compatti possibile.

Formalmente, dato un insieme di osservazioni $(x_1, x_2, \ldots, x_n)$, con ciascun $x_i \in \Re^d$, K-means suddivide questi punti in $k(\leq n)$ insiemi $S = \{S_1, S_2, \ldots, S_k\}$ minimizzando la funzione obiettivo:

$$arg\ min_S \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$$

- μ_i è il centroide del cluster S_i, definito come $\mu_i = \frac{1}{|S_i|} \sum_{x \in S_i} x$,
- $|S_i|$ è la dimensione di S_i
- $|| \cdot ||$ è la norma.

L'algoritmo K-means opera attraverso un processo iterativo composto da due fasi fondamentali:

1. Assignment Step

   Ogni osservazione viene assegnata al cluster con il centroide più vicino in termini di distanza euclidea al quadrato:

$$S_i^{(t)} = \{ x_p : || x_p - \mu_i^{(t)} ||^2 \leq || x_p - \mu_j^{(t)} ||^2 \ \forall j, 1 \leq j \leq k \}$$

   Questa operazione suddivide lo spazio secondo il *diagramma di Voronoi* generato dai centroidi correnti

2. Update Step

   Una volta assegnati i punti, si ricalcola il centroide di ciascun cluster:

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

3. Il processo viene reiterato fino alla convergenza, ovvero quando le assegnazioni dei punti ai cluster non cambiano più, o quando il valore della funzione obiettivo (WCSS) si stabilizza.

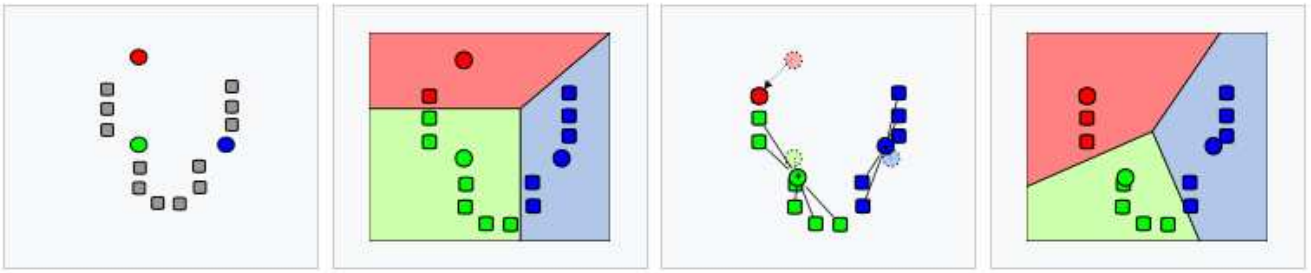


Figura 27 28 29 30: Visualizzazione del K-means Standard: 1) Inizializzazione di K centroidi nel dominio dei dati 2) Assignment + definizione dei cluster 3) Update centroidi 4) Reiterati assegnazione e update

La funzione WCSS diminuisce monotonicamente a ogni iterazione, generando una sequenza non negativa e monotona che garantisce la convergenza. Tuttavia, l'algoritmo non garantisce il raggiungimento del minimo globale; il risultato dipende fortemente dai centroidi iniziali. Per questo motivo si eseguono spesso più run con inizializzazioni diverse.

Alcune di queste strategie sono:

- Forgy method: sceglie casualmente k osservazioni come centroidi iniziali; tende a produrre centroidi distanziati.
- Random Partition: assegna casualmente ogni punto a un cluster e calcola i centroidi; tende a porre i centroidi iniziali vicino al centro dello spazio dei dati.
- Maximin, Bradley–Fayyad, K-means++: altri metodi più robusti che mirano a evitare convergenze premature.

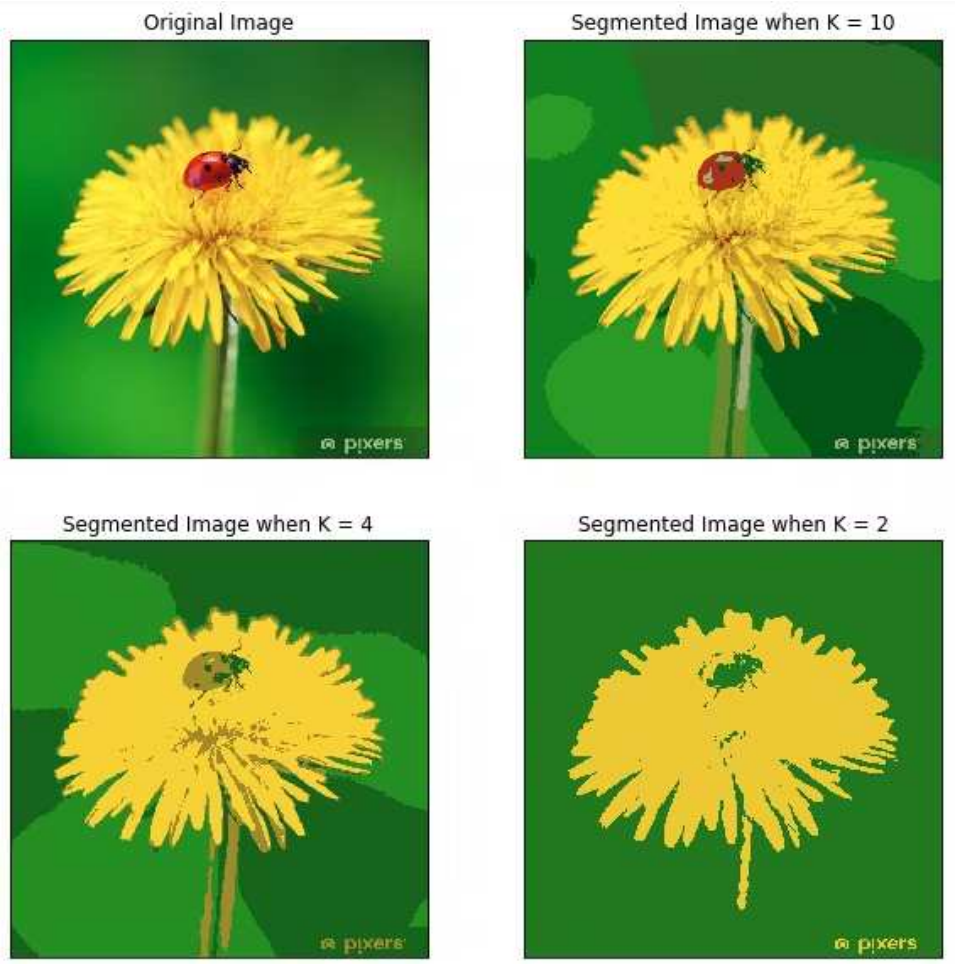


Figura 31: Risultato del clustering con numero di K differenti

2.5 Image Resizing

Il ridimensionamento delle immagini è un'operazione fondamentale nel pre-processing che mira a modificare le dimensioni di un'immagine, in aumento (*upsampling*) o in diminuzione (*downsampling*).

La sfida principale risiede nel mantenere la qualità dell'immagine e prevenire artefatti visivi. Quando la dimensione cambia, è necessario interpolare i valori dei nuovi pixel. I metodi più comuni includono:

- *Nearest Neighbor*
- *Bilinear interpolation*
- *Bicubic interpolation*

Di default la funzione resize della libreria OpenCV utilizza l'interpolazione bilineare, perché offre i migliori risultati (rispetto alla Nearest Neighbours) e costo computazionale minore rispetto alla Bicubic Interpolation.

Nearest neighbour Interpolation

L'Interpolazione del vicino più prossimo è l'approccio più semplice e rapido all'interpolazione. Questo metodo individua semplicemente il pixel "vicino" più prossimo nell'immagine sorgente e ne assume il valore di intensità per il pixel di destinazione. [19]

In un contesto unidimensionale (ad esempio, un grafico $y = f(x)$), se si vogliono inserire nuovi punti dati x_i tra i punti originali x_1 e x_2, a ciascun nuovo punto viene assegnato il valore $f(x_i)$ del punto originale x_1 o x_2 che è più vicino lungo l'asse orizzontale.

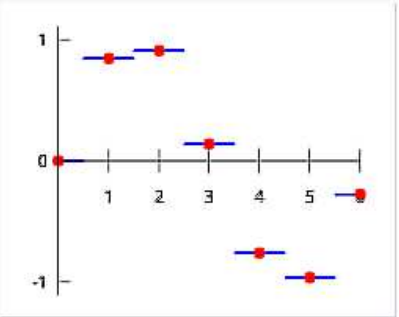


Figura 32: Nearest Neighbour Interpolation in una dimensione

Nelle immagini, si verifica una situazione analoga. Mettiamo caso che vogliamo riscalare un'immagine $m\,x\,n$ in una a $p\,x\,q$ pixels (assumendo $p > m$ e $q > n$). Per ogni pixel (j, k) dell'immagine di destinazione si calcolano le coordinate corrispondenti nell'immagine sorgente mediante la mappatura inversa, che utilizza i fattori di scala:

- Fattore di scala orizzontale: $s_1 = \frac{p}{m}$
- Fattore di scala verticale: $s_2 = \frac{q}{n}$

Questa mappatura permette di "risalire" al punto dell'immagine originale da cui deriva il pixel destinazione. Le coordinate sorgenti ottenute saranno generalmente frazionarie; per il nearest neighbour si effettua semplicemente un arrotondamento all'intero più vicino:

$$(i_{src}, j_{src}) = (round(j_{dest} \cdot s_1), round(k_{dest} \cdot s_2))$$

Il valore del pixel dell'immagine sorgente viene quindi copiato nell'immagine di destinazione:

$$I'(j_{dest}, k_{dest}) = I(i_{src}, j_{src})$$

Il risultato è un metodo estremamente rapido e computazionalmente leggero, ma che può produrre immagini dall'aspetto "a blocchi" o scalettato.

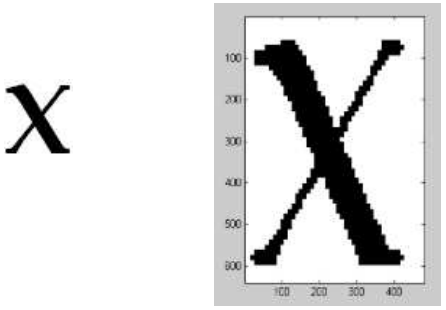


Figura 33: Nearest Neighbour in 2D; immagine piccola (sinistra) riscalata (destra)

Bilinear Interpolation

L'interpolazione bilineare è una tecnica utilizzata per stimare il valore di una funzione di due variabili, quando è nota solo su una griglia rettangolare di punti. L'idea è estendere la classica interpolazione lineare applicandola due volte (da qui il nome *bi*-lineare): prima lungo una direzione e poi lungo l'altra. [2]

Si supponga di voler stimare il valore della funzione f nel punto (x, y).
Sono noti i valori della funzione nei quattro vertici del rettangolo:

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

e i corrispondenti valori: $f(Q_{11})$, $f(Q_{12})$, $f(Q_{21})$, $f(Q_{22})$

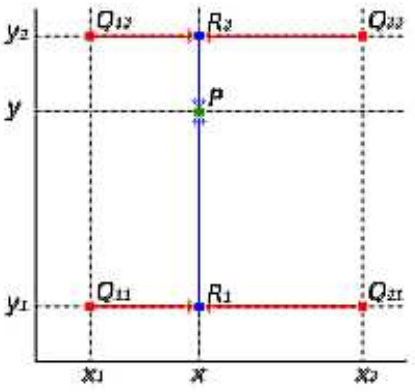


Figura 34: punti rossi: vertici del rettangolo, punto verde è il punto che vogliamo interpolare

Nel contesto del *resizing* delle immagini, tali punti rappresentano pixel dell'immagine sorgente, mentre i valori della funzione f corrispondono alla loro intensità luminosa (o, più in generale, al valore nel canale considerato: R, G, B o grayscale).

Supponiamo che vogliamo riscalare un'immagine $m\,x\,n$ in una a $p\,x\,q$ pixels (assumendo $p > m$ e $q > n$).

Per ogni pixel (i_{dest}, j_{dest}) della nuova immagine $p\,x\,q$, calcoliamo la posizione corrispondente nell'immagine $m\,x\,n$.

Si usa la seguente mappatura inversa:

$$x_{src} = j_{dest} \cdot \frac{m-1}{p-1}$$

$$y_{src} = i_{dest} \cdot \frac{n-1}{q-1}$$

Dato un punto sorgente, se le sue coordinate risultano punti interi prenderemo per la specifica coordinata i punti corrispondenti. Per le coordinate di numeri non interi invece:

- $x_1 = \lfloor x_{src} \rfloor$
- $y_1 = \lfloor y_{src} \rfloor$
- $x_2 = \lceil x_{src} \rceil$
- $y_2 = \lceil y_{src} \rceil$

Primo passo: interpolazione lungo l'asse x

Interpoliamo tra i due punti (x_1, y_1) e (x_2, y_1) , ottenendo il valore a (x , y_1) :

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

Analogamente, si interpola lungo la linea a $y = y_2$:

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

Secondo passo: interpolazione lungo l'asse y

A questo punto si interpolano i due valori ottenuti:

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

Sostituendo le formule precedenti, si ottiene l'espressione completa:

$$f(x,\ y)\ =\ \frac{y_2-y}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{11})\ +\frac{x\ -x_1}{x_2-x_1}f(Q_{21})\right)\ +\frac{y\ -y_1}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{12})\ +\frac{x\ -x_1}{x_2-x_1}f(Q_{22})\right)$$

Questa può essere riscritta anche nella forma compatta:

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

Bicubic Interpolation

L'interpolazione bicubica è una tecnica avanzata di upscaling delle immagini che produce risultati più morbidi e visivamente gradevoli rispetto a metodi più semplici, come il nearest neighbour o il bilinear interpolation. [2] [28]

Per comprendere l'interpolazione bicubica è utile partire dalla sua versione monodimensionale.
Supponiamo di voler interpolare una funzione $f(x)$ nell'intervallo $[0, 1]$. Se conosciamo:

- il valore di f agli estremi: $f(0)$, $f(1)$
- il valore della derivata agli stessi punti: $f'(0)$, $f'(1)$

allora possiamo costruire un polinomio cubico:

$$f(x) = ax^3 + bx^2 + cx + d$$

che soddisfi questi 4 vincoli ed interpoli i punti.
Determinare i valori dei coefficienti significa semplicemente risolvere il sistema:

$$\begin{aligned} f(0) &= d \\ f(1) &= a + b + c + d \\ f'(0) &= c \\ f'(1) &= 3a + 2b + c \end{aligned}$$

che porta alle note formule:

$$\begin{aligned} a &= 2f(0) - 2f(1) + f'(0) + f'(1) \\ b &= -3f(0) + 3f(1) - 2f'(0) - f'(1) \\ c &= f'(0) \\ d &= f(0) \end{aligned}$$

Ma da dove prendiamo le derivate, se stiamo interpolando dei semplici valori discreti?

Supponiamo di avere i valori $p_{0,}\, p_{1,}\, p_{2,}\, p_3$ rispettivamente in $x = -1,\ x = 0,\ x = 1\ e\ x = 2$ (quindi avere dei punti al di fuori dell'intervallo). Le derivate si stimano come pendenza media:

$$f(0) = p_1$$
$$f(1) = p_2$$
$$f'(0) = \frac{p_2 - p_0}{2}$$
$$f'(1) = \frac{p_3 - p_1}{2}$$

Combinando le formule appena scritte:

$$a = -\tfrac{1}{2}p_0 + \tfrac{3}{2}p_1 - \tfrac{3}{2}p_2 + \tfrac{1}{2}p_3$$
$$b = p_0 - \tfrac{5}{2}p_1 + 2p_2 - \tfrac{1}{2}p_3$$
$$c = -\tfrac{1}{2}p_0 + \tfrac{1}{2}p_2$$
$$d = p_1$$

Possiamo quindi estendere questa idea all'interpolazione bicubica, che come per la bilineare applica in due dimensioni (bi) l'interpolazione cubica.

Consideriamo una cella quadrata di 4 pixel adiacenti. L'interpolazione bicubica approssima l'intensità luminosa all'interno della cella con un *polinomio bicubico*:

$$p(x,\ y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j$$

dove i 16 coefficienti a_{ij} determinano altezza, pendenza, curvatura e torsione della superficie. Per determinare un polinomio bicubico occorrono 16 vincoli, e questi provengono da:

1. Valori della funzione nei 4 angoli della cella $f(0,\ 0),\ f(1,\ 0),\ f(0,\ 1),\ f(1,\ 1)$
2. Derivate lungo x nei 4 angoli $f_x(0,\ 0),\ f_x(1,\ 0),\ f_x(0,\ 1),\ f_x(1,\ 1)$
3. Derivate lungo y nei 4 angoli $f_y(0,\ 0),\ f_y(1,\ 0),\ f_y(0,\ 1),\ f_y(1,\ 1)$
4. Derivate miste nei 4 angoli $f_{xy}(0,\ 0),\ f_{xy}(1,\ 0),\ f_{xy}(0,\ 1),\ f_{xy}(1,\ 1)$

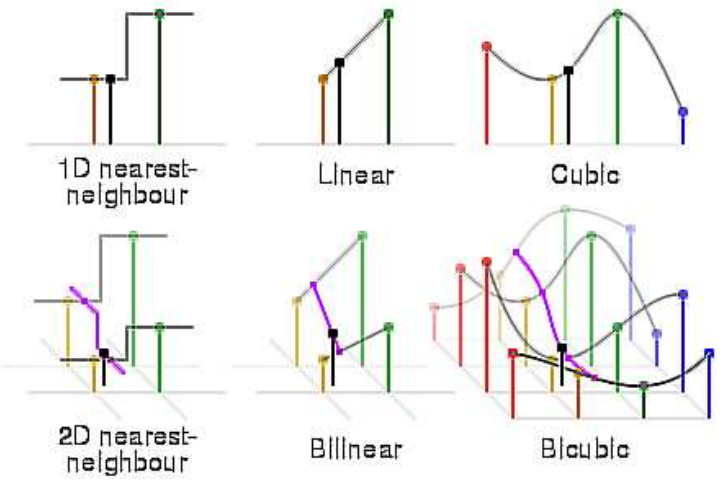


Figura 35: Le tre tecniche di interpolazione principali messe a confronto

Le derivate vengono stimate mediante differenze finite tra i pixel circostanti:

$$f_x(i,\ j)\ \simeq \frac{I(i+1,j)-I(i-1,j)}{2}$$

$$f_y(i,\ j)\ \simeq \frac{I(i\ ,j+1)-I(i\ ,j-1)}{2}$$

$$f_{xy}(i,\ j)\ \simeq \frac{I(i+1\ ,j+1)-I(i+1\ ,j-1)-I(i-1\ ,j+1)+I(i-1,j-1)}{4}$$

Per ogni cella, si costruisce un sistema lineare 16×16 che lega i valori dei pixel e le derivate ai coefficienti del polinomio. Raggruppiamo i parametri sconosciuti a_{ij} in:

$$\alpha = \left[\ a_{00}\ \ a_{10}\ \ a_{20}\ \ a_{30}\ \ a_{01}\ \ a_{11}\ \ a_{21}\ \ a_{31}\ \ a_{02}\ \ a_{12}\ \ a_{22}\ \ a_{32}\ \ a_{03}\ \ a_{13}\ \ a_{23}\ \ a_{33}\ \right]^T$$

e i valori dei punti e delle derivate calcolati in:

$$x = \left[\ f(0,0)\ \ f(1,0)\ \ f(0,1)\ \ f(1,1)\ \ f_x(0,0)\ \ f_x(1,0)\ \ f_x(0,1)\ \ f_x(1,1)\ \ f_y(0,0)\ \ f_y(1,0)\ \ f_y(0,1)\ \ f_y(1,1)\ \ f_{xy}(0,0)\ \ f_{xy}(1,0)\ \ f_{xy}(0,1)\ \ f_{xy}(1,1)\ \right]^T$$

A questo punto possiamo definire la relazione:

$$A\alpha\ =\ x$$

La matrice A è la matrice dei coefficienti del sistema, che ha dimensioni $16\ x\ 16$. Essa è costruita prendendo i coefficienti numerici che moltiplicano ogni a_{ij} nelle 16 equazioni di vincolo.

Ogni riga di A corrisponde a una delle 16 equazioni fornite:

- Le colonne di A corrispondono ai coefficienti delle incognite a_{ij}
- Le righe di A corrispondono ai vincoli (prima i valori f, poi f_x, poi f_y, poi f_{xy} per le combinazioni di punti rispettivamente $(0,\ 0)$, $(1,\ 0)$, $(0,\ 1)$, $(1,\ 1)$).

Invertendo la matrice A, otteniamo quindi:

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 \\ -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 \\ 9 & -9 & -9 & 9 & 6 & 3 & -6 & -3 & 6 & -6 & 3 & -3 & 4 & 2 & 2 & 1 \\ -6 & 6 & 6 & -6 & -3 & -3 & 3 & 3 & -4 & 4 & -2 & 2 & -2 & -2 & -1 & -1 \\ 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ -6 & 6 & 6 & -6 & -4 & -2 & 4 & 2 & -3 & 3 & -3 & 3 & -2 & -1 & -2 & -1 \\ 4 & -4 & -4 & 4 & 2 & 2 & -2 & -2 & 2 & -2 & 2 & -2 & 1 & 1 & 1 & 1 \end{bmatrix}$$

che possiamo utilizzare per calcolare le nostre incognite in α:

$$\alpha = A^{-1}x$$

La forma più concisa per rappresentare questo calcolo (omessi i passaggi intermedi per semplicità di lettura):

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Una volta determinati i 16 coefficienti a_{ij} che definiscono il polinomio bicubico, è possibile valutare l'interpolante in qualunque punto interno alla cella utilizzando semplicemente la formula $p(x,\ y)$.

Questo permette di stimare in modo preciso il valore del pixel richiesto durante il processo di upscaling. L'aspetto fondamentale è che il polinomio ottenuto non garantisce soltanto la continuità della funzione, ma assicura anche la continuità delle sue derivate prime: ciò significa che l'immagine risultante non presenta bruschi

cambiamenti di intensità tra una cella e l'altra, producendo quindi una superficie luminosa più regolare e visivamente più naturale.

Durante la fase di resize, ogni pixel dell'immagine di destinazione viene ricondotto alla corrispondente posizione frazionaria nell'immagine sorgente attraverso una trasformazione di mappatura inversa (come nei casi precedenti). Identificata la cella di appartenenza, si recuperano i valori necessari: i quattro valori ai vertici con le rispettive derivate. A questo punto, si sfrutta direttamente la forma matriciale di A^{-1} per ottenere i coefficienti del polinomio e si valuta $p(x, y)$ nella posizione desiderata.

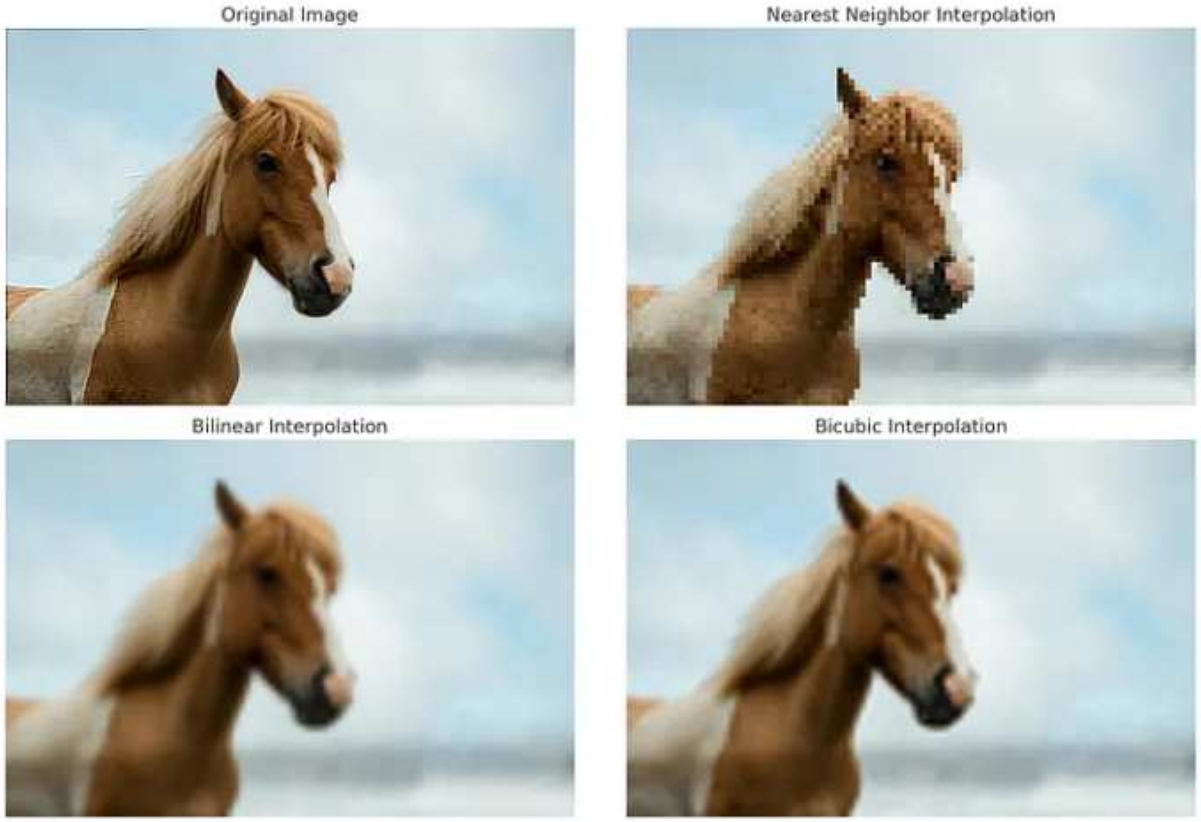


Figura 36: Tecniche di resizing a confronto

2.6 Perspective correction

Nel campo dell'elaborazione digitale delle immagini, le trasformazioni geometriche rivestono un ruolo fondamentale per modificare la posizione, la forma e la prospettiva dei contenuti visivi.

Le trasformazioni geometriche agiscono sulle coordinate dei pixel di un'immagine. Di seguito vengono descritte le due famiglie principali: trasformazioni affini e prospettiche (omografie), con le relative formule ed implicazioni.

Trasformazioni affini [27]

Una trasformazione affine T combina una trasformazione lineare A con una traslazione b. In $\Re^2$ si scrive:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

Gli esempi più comuni includono scala (dilation), traslazione, rotazione.

Scala (Dilation):

La scala corrisponde semplicemente al *Resize* dell'immagine; questo viene implementato però non specificando le dimensioni dell'immagine di destinazione, bensì attraverso dei fattori di scala. Per quanto riguarda i metodi di resize, si guardi il paragrafo precedente.

Traslazione:

La traslazione corrisponde allo spostamento della posizione di un oggetto. Se si conosce lo spostamento nella direzione (x, y) e si considera che sia (t_x, t_y), è possibile creare la matrice di trasformazione come segue:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Rotazione:

La rotazione di un'immagine per un angolo si ottiene tramite la matrice di trasformazione della forma:

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Ma OpenCV fornisce una rotazione scalata con centro di rotazione regolabile, in modo da poter ruotare in qualsiasi posizione si preferisca. La matrice di trasformazione modificata è data da:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$$

dove:

$$\alpha = scale \cdot \cos\theta$$
$$\beta = scale \cdot \sin\theta$$

Omografie [10] [20]

Una omografia H è una trasformazione proiettiva che mappa punti di un piano su un altro piano.

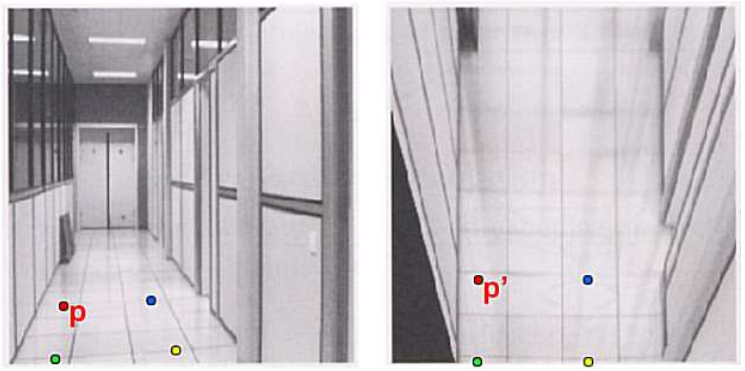


Figura 37: Esempio omografia

Questo tipo di trasformazione è fondamentale in visione artificiale, perché descrive come l'immagine di una stessa scena cambia quando una telecamera si sposta o varia il proprio orientamento.

In altre parole, due immagini della stessa superficie piana, acquisite da posizioni di ripresa differenti, risultano tra loro collegate da un'omografia. Tale relazione consente di modellare matematicamente il cambiamento di prospettiva tra le due viste.

Per esprimere correttamente tale relazione, si ricorre al formalismo delle *coordinate omogenee*, un sistema ampiamente utilizzato in geometria proiettiva

Al contrario delle coordinate cartesiane, le coordinate omogenee utilizzano 3 coordinate (x', y', z') per rappresentare i punti nello spazio. Convertire le coordinate omogenee in coordinate cartesiane inoltre è relativamente semplice:

$$(x', y', z') \rightarrow (x, y) : \begin{matrix} x = x'/z' \\ y = y'/z' \end{matrix}$$

La matrice di omografia viene spesso utilizzata per rappresentare la relazione di trasformazione omografica tra due immagini. La matrice di omografia H è definita come segue:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

In questa matrice:

- gli elementi $(h_{11}, h_{12}, h_{21}, h_{22})$ rappresentano la componente affine della trasformazione;
- i coefficienti (h_{13}, h_{23}) descrivono la traslazione;
- gli elementi (h_{31}, h_{32}) modellano la componente prospettica.

La relazione di trasformazione tra le coordinate omogenee di due punti corrispondenti nelle due immagini può essere quindi scritta come:

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}_d \\ \tilde{y}_d \\ \tilde{z}_d \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

dove $(x_s, y_s, 1)$ rappresenta le coordinate omogenee di un punto caratteristico nella prima immagine; $(x_d, y_d, 1)$ indica le coordinate omogenee del punto corrispondente nell'altra immagine. Possiamo espandere l'equazione:

$$x_d(h_{31}x_s + h_{32}y_s + h_{33}) = h_{11}x_s + h_{12}y_s + h_{13}$$

$$y_d(h_{31}x_s + h_{32}y_s + h_{33}) = h_{21}x_s + h_{22}y_s + h_{23}$$

Possiamo quindi estendere la forma:

$$x_d = \frac{\tilde{x}_d}{\tilde{z}_d} = \frac{h_{11}x_s + h_{12}y_s + h_{13}}{h_{31}x_s + h_{32}y_s + h_{33}}$$

$$y_d = \frac{\tilde{y}_d}{\tilde{z}_d} = \frac{h_{21}x_s + h_{22}y_s + h_{23}}{h_{31}x_s + h_{32}y_s + h_{33}}$$

$$\begin{bmatrix} x_s & y_s & 1 & 0 & 0 & 0 & -x_d x_s & -x_d y_s & -x_d \\ 0 & 0 & 0 & x_s & y_s & 1 & -y_d x_s & -y_d^{(i)} y_s & -y_d \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Essendo H definita a fattore di scala e avendo quindi 8 gradi di libertà, sono necessarie almeno quattro coppie di punti per determinarla. Convenzionalmente si impone $h_{33} = 1$ per fissare il fattore di scala.

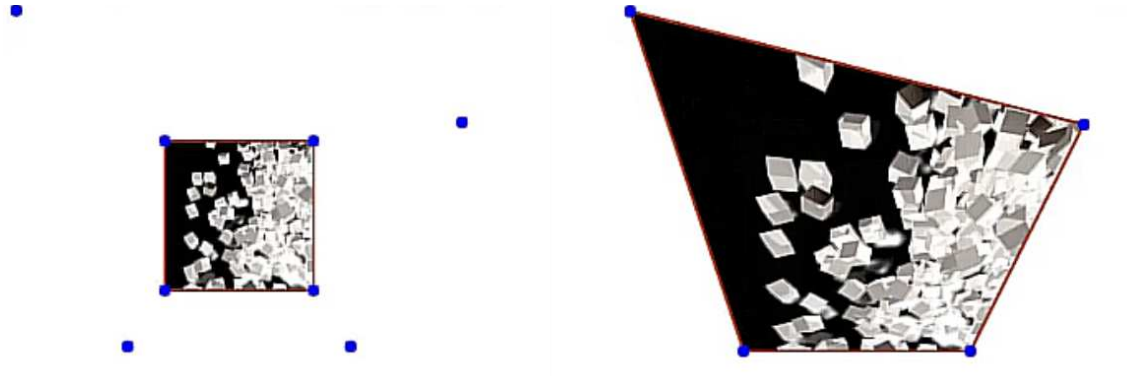


Figura 38: esempio di omografia date le 4 coppie di punti (partenza-destinazione)

OpenCV mette a disposizione due funzioni principali per lavorare con le omografie:

1. *getPerspectiveTransform()* calcola la matrice omografica H a partire da quattro punti sorgente e quattro punti destinazione.
2. *warpPerspective()*: applica la trasformazione usando H^{-1} per eseguire un *inverse mapping*: invece di prendere ogni pixel dell'immagine sorgente e proiettarlo nella destinazione (operazione che lascerebbe "buchi" nell'immagine risultante), si considera ogni pixel dell'immagine destinazione e si calcola da quale punto dell'immagine sorgente provenga. Ciò permette di usare tecniche di interpolazione, come quella bilineare, ottenendo un'immagine finale completa e continua.

2.7 Feature detection (edge detection algorithms)

La feature detection è il processo di identificazione di punti o pattern distintivi all'interno di un'immagine. Questi punti possono essere utilizzati in applicazioni di visione artificiale, come il riconoscimento di oggetti, il tracking o la ricostruzione 3D.

Possiamo classificare le caratteristiche in tre diverse tipologie, come mostrato di seguito:

Edges: sono i confini in cui la luminosità dell'immagine cambia bruscamente. Rappresentano i contorni degli oggetti all'interno di un'immagine.

Corners:punti in cui due o più bordi si incontrano. Si trovano spesso all'intersezione di forme o oggetti diversi.

Blobs: regioni dell'immagine che differiscono per proprietà, come luminosità o colore, rispetto alle regioni circostanti

Ai fini di questa tesi ci soffermeremo su alcuni algoritmi utili al riconoscimento dei bordi in un'immagine.

Prewitt Operator [26]

Nel contesto dell'estrazione delle feature geometriche, l'operatore di Prewitt [Prewitt, 1970] rappresenta una delle soluzioni più efficienti per l'approssimazione del gradiente di intensità. Dal punto di vista analitico, questo filtro opera una differenziazione discreta sui pixel dell'immagine, identificando le zone di discontinuità luminosa che definiscono i bordi.

A differenza di una semplice derivata, l'operatore associa a ogni coordinata spaziale un vettore gradiente caratterizzato da due parametri fondamentali:

1. *Magnitudine***:** che esprime la forza della transizione (maggiore è il salto di intensità, più elevata è la probabilità di trovarsi su un contorno reale).
2. *Orientamento:* che indica la direzione di massima variazione luminosa, puntando perpendicolarmente rispetto all'andamento del bordo.

In termini operativi, l'algoritmo impiega due maschere (kernel) di dimensioni 3 x 3 per analizzare le variazioni lungo le direzioni ortogonali. Se indichiamo con A l'immagine di input, le componenti del gradiente G_x e G_y vengono generate tramite il processo di convoluzione bidimensionale *:

$$\mathbf{G_y} = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_x} = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

(la convoluzione bidimensionale consiste nel moltiplicare elemento per elemento dell'operatore di prewitt con la sottomatrice 3 x 3 di A del centrata sul pixel che viene utilizzato correntemente e sommare; sui bordi si applicherebbero come per i filtri tecniche di padding: in questo testo non li ho trattati per semplicità).

La combinazione di queste due componenti permette di ottenere la mappa finale dei bordi attraverso il calcolo della norma euclidea:

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

Per determinare la direzione angolare Θ del gradiente, si ricorre alla funzione

$$\mathbf{\Theta} = \operatorname{atan2}(\mathbf{G}_y, \mathbf{G}_x)$$

L'operatore di Prewitt è relativamente semplice dal punto di vista computazionale rispetto ad operatori più sofisticati come Sobel o Canny, tuttavia è molto sensibile al rumore; quindi anche piccoli disturbi nell'immagine possono generare falsi bordi.

Figure 39 40: Rilevamento bordi tramite operatore di Prewitt

Sobel Edge Detection [5]

Il Sobel operator è un filtro discreto utilizzato per l'analisi dei bordi nelle immagini, che combina l'approssimazione del gradiente con un leggero effetto di smoothing. Rispetto al Prewitt operator, introduce un peso maggiore sui pixel centrali del kernel, migliorando la robustezza al rumore e fornendo gradienti più stabili in presenza di variazioni locali di intensità.

Il principio è simile a quello del Prewitt: l'operatore calcola il gradiente dell'intensità dell'immagine in due direzioni ortogonali, orizzontale e verticale. Se indichiamo con A l'immagine originale e con G_x e G_y le immagini contenenti le approssimazioni dei gradienti in direzione orizzontale e verticale, le convoluzioni con i kernel Sobel si scrivono come:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \qquad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

dove * indica la convoluzione bidimensionale.

Una volta calcolati G_x e G_y, allo stesso modo del Prewitt è possibile ottenere la magnitudine del gradiente e la direzione del bordo.

Grazie al peso maggiore sui pixel centrali, il Sobel operator è più resistente al rumore rispetto al Prewitt e fornisce stime più precise dei bordi, pur rimanendo computazionalmente leggero. Per questo motivo, rappresenta un passaggio naturale tra operatori semplici come il Prewitt e algoritmi più sofisticati come il Canny.

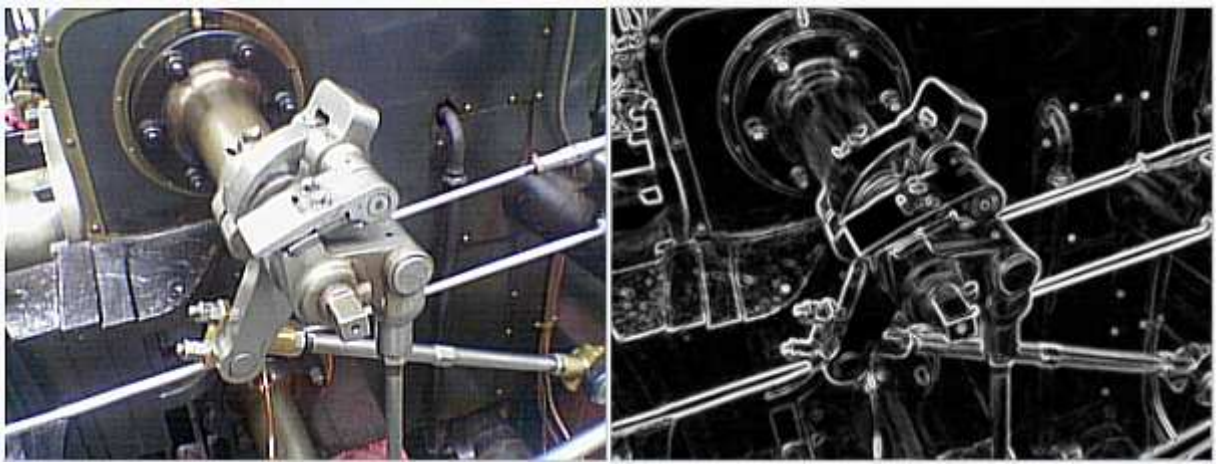


Figura 41 42: rilevamento dei bordi tramite Sobel

Canny Edge Detection [4]

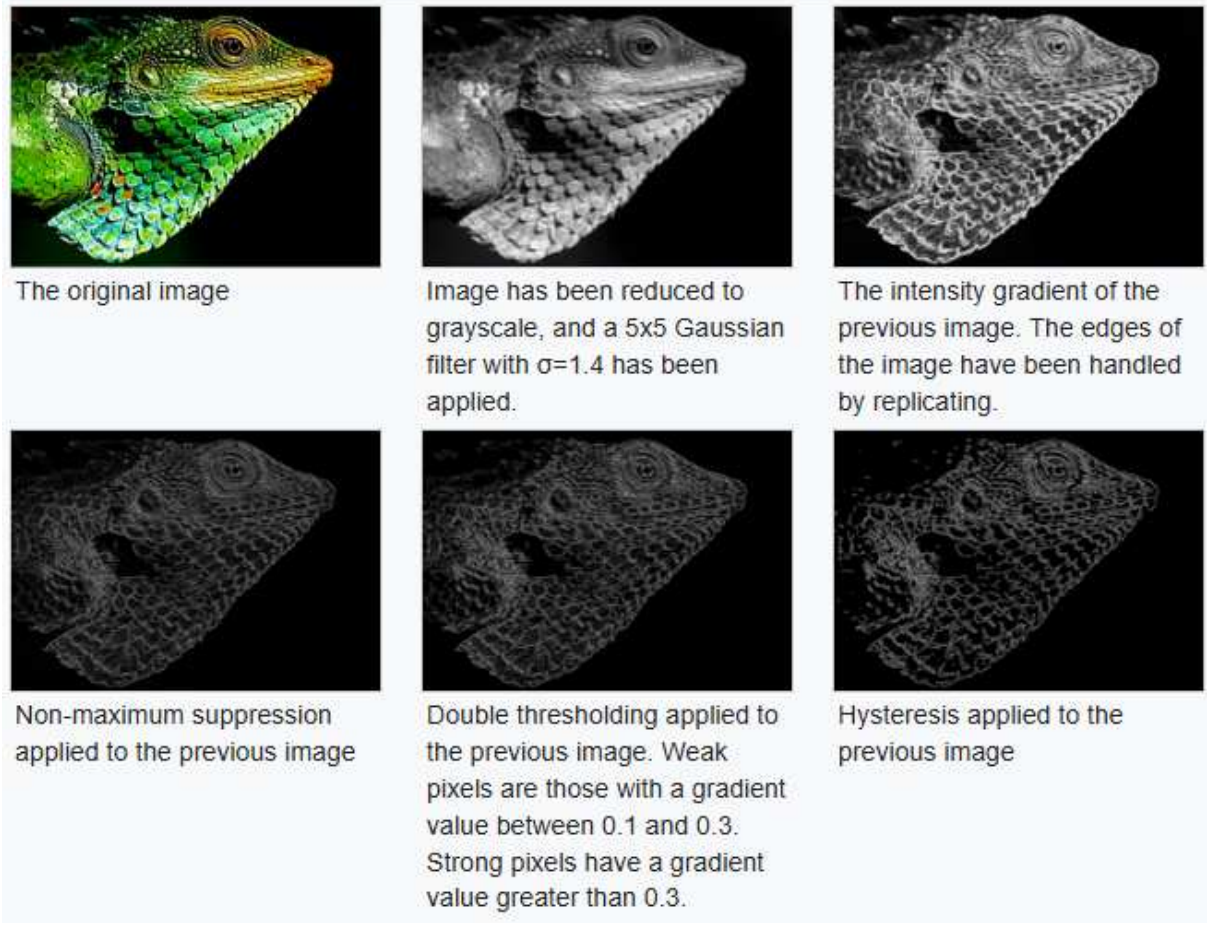
Il canny edge detector (John F. Canny 1986), rappresenta uno dei metodi più avanzati e utilizzati per l'individuazione dei bordi in un'immagine. A differenza degli operatori più semplici come Prewitt e Sobel, Canny nasce dalla volontà di definire un *modello matematicamente ottimale* per il rilevamento dei bordi. L'obiettivo era soddisfare tre requisiti fondamentali:

1. *Massimizzare la rilevazione* (bassa probabilità di errore).
2. *Massima localizzazione* del punto in cui avviene la variazione di intensità.
3. *Risposta singola per bordo* evitando bordi duplicati e sopprimendo falsi rilevamenti generati dal rumore.

Per garantire queste proprietà, Canny derivò una funzione ottimale composta da una combinazione di esponenziali, successivamente approssimata con la derivata del filtro gaussiano, dando origine alla pipeline a più stadi che caratterizza l'algoritmo.

Il metodo di Canny è articolato in cinque step consecutivi, ognuno dei quali contribuisce a garantire stabilità e precisione nella rilevazione dei bordi:

1. *Smoothing tramite filtro gaussiano*
   Per ridurre l'effetto del rumore, l'immagine viene convoluta con un kernel gaussiano.
2. *Calcolo del gradiente di intensità*
   Sull'immagine filtrata si applicano operatori di derivazione (tipicamente Sobel)
3. *Non-Maximum Suppression (NMS)*
   Per ottenere bordi sottili, ogni pixel viene confrontato con i due vicini lungo la direzione del gradiente: un pixel viene mantenuto solo se la sua magnitudine è la massima tra i tre; altrimenti viene annullato. (questo algoritmo verrà spiegato nel prossimo capitolo)
4. *Double Thresholding*
   Per separare bordi reali da risposte spurie, si utilizzano due soglie:
   - High threshold per definire i "bordi forti"
   - Low threshold per definire i "bordi deboli".
5. Edge Tracking by Hysteresis
   I *weak edges* (bordi deboli) vengono conservati solo se connessi a un strong edge.
   In questo modo il metodo elimina gran parte dei falsi bordi mantenendo continuità nelle strutture reali.



The original image

Image has been reduced to grayscale, and a 5x5 Gaussian filter with σ=1.4 has been applied.

The intensity gradient of the previous image. The edges of the image have been handled by replicating.

Non-maximum suppression applied to the previous image

Double thresholding applied to the previous image. Weak pixels are those with a gradient value between 0.1 and 0.3. Strong pixels have a gradient value greater than 0.3.

Hysteresis applied to the previous image

Figura 43 44 45 46 47 48: I 5 step del canny edge detector

2.8 Feature extraction [18]

Gli algoritmi di edge detection hanno lo scopo di produrre una mappa dei contorni dell'immagine, evidenziando punti in cui l'intensità varia bruscamente. Tuttavia, quando l'obiettivo finale è riconoscere una forma specifica questi metodi da soli non risolvono il problema. L'edge map contiene infatti tutti i contorni prodotti dall'immagine, non solo quelli appartenenti alla forma che vogliamo trovare.
Questa situazione porta a tre difficoltà fondamentali:

1. Nell'edge map compaiono decine o centinaia di contorni secondari: ombre, texture, rumore, parti di oggetti non rilevanti. Se ad esempio vuoi trovare le ruote (cerchi) di una bicicletta, l'edge map mostra sì i cerchi… ma anche i raggi, il telaio, lo sfondo e altre strutture. A quali punti quindi appartiene davvero la forma che stai cercando?
2. Gli edge detector non forniscono mai contorni perfetti. Parti della ruota possono essere mancanti o interrotte, sia per limiti dell'algoritmo, sia per condizioni reali: occlusioni, riflessi, saturazione, sfocature. L'oggetto che vuoi trovare non esiste mai completamente nell'immagine: devi essere in grado di riconoscerlo anche se è solo parzialmente presente.
3. Pixel che non appartengono realmente a nessun bordo significativo possono essere marcati come edges. Questi punti "falsi positivi" confondono il fitting, perché sembrano possibili candidati ma non appartengono alla geometria reale.

Serve quindi un metodo capace di:

- integrare informazione *parziale*
- ignorare gli outlier
- far emergere strutture semplici descritte da pochi parametri

È qui che entra in gioco la trasformazione di Hough, che risolve tutte e tre le difficoltà in un colpo solo grazie ad un elegante meccanismo di voto nello spazio dei parametri.

Trasformazione di Hough [17] [6]

La *Hough Transform* è una delle tecniche più consolidate per la rilevazione di forme geometriche semplici all'interno di immagini digitali. Il suo sviluppo risale agli anni 50 - 60: il primo metodo fu introdotto da Paul Hough nel 1959 per l'analisi automatica di fotografie di camere a bolle, e successivamente brevettato nel 1962.

La forma moderna e realmente applicabile in computer vision venne però definita più tardi, nel 1972, da Duda e Hart, che introdussero la parametrizzazione (r, Θ) per le rette, rendendo il metodo robusto ed efficiente. Negli anni successivi (Ballard, 1981) la trasformata venne generalizzata a forme più complesse, come cerchi o ellissi.

L'idea fondamentale è interpretare il riconoscimento di forme come un processo di voto:

- ogni pixel che fa parte di un bordo vota per tutte le curve che potrebbero passare per quel punto
- nello spazio dei parametri si costruisce un *accumulator*, cioè una matrice in cui ogni cella rappresenta una possibile geometria
- le celle che ricevono più voti corrispondono alle forme presenti nell'immagine

Il primo passo è scegliere una rappresentazione matematica che permetta di descrivere *tutte* le rette con un numero finito e stabile di parametri.
La forma comune $y = mx + b$ non è adatta, perché non può rappresentare le linee verticali (pendenza m infinita). Per fare ciò ci dobbiamo quindi spostare dalla rappresentazione della coordinate a quella dei parametri e tenere in considerazione la limitazione posta da m.

Duda e Hart per risolvere questo problema introdussero quindi la rappresentazione in *forma normale di Hesse*:

$$r = x\,cos(\Theta) + y\,sin(\Theta)$$

dove:

- r è la distanza della retta dall'origine,
- Θ è l'angolo del vettore normale alla retta.

Ogni retta corrisponde dunque a un punto nel piano (r, Θ), noto anche come Hough space.

Per un singolo punto dell'immagine (x_i, y_i), le infinite rette che lo attraversano possono essere espresse come:

$$r = x_i \cos(\Theta) + y_i \sin(\Theta)$$

Questa equazione descrive una sinusoide nello spazio (r, Θ) dove:

- Due punti allineati generano due sinusoidi che si intersecano nello stesso punto (r, Θ)
- Un insieme di pixel collineari produce quindi una concentrazione di intersezioni nello stesso punto

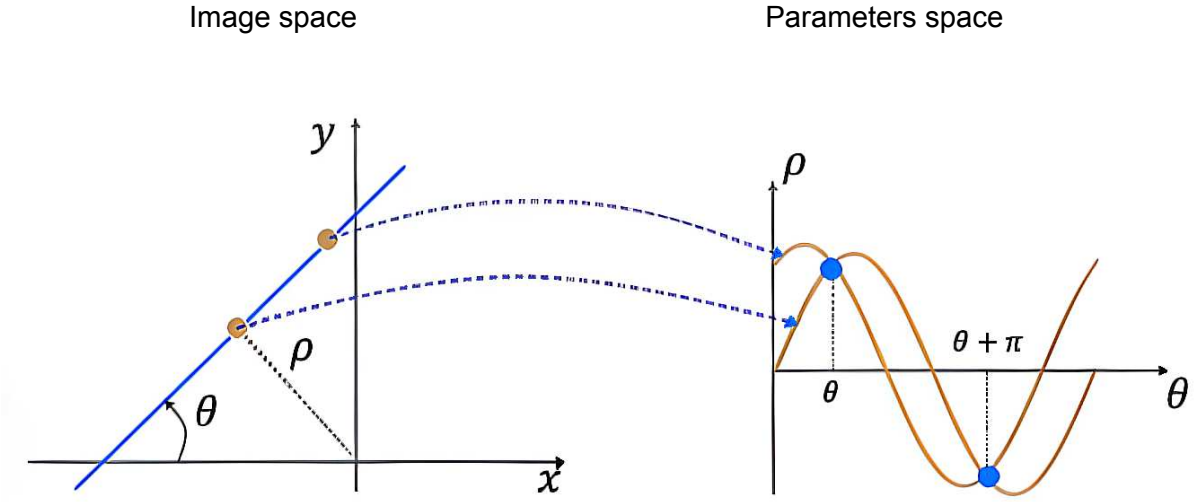


Figura 49: rappresentazione punti su una retta dallo spazio dell'immagine allo spazio dei parametri; nell'immagine viene usato ρ per descrivere r

L'algoritmo classico si basa sui seguenti passaggi:

1. *Edge detection*
   Si parte da un'immagine elaborata tramite Canny o altro operatore.
2. *Inizializzazione dell'accumulatore*
   Si discretizzano gli intervalli:
   $\Theta \in [0, \pi] \quad r \in [-r_{min}, r_{max}]$
   ottenendo una matrice bidimensionale $A(r, \Theta)$ inizialmente a zero
3. *Voto di ogni pixel di bordo*
   Per ogni pixel (x, y) appartenente ad un edge:
   - si cicla Θ sull'insieme discretizzato
   - si calcola $r = x \cos(\Theta) + y \sin(\Theta)$

- si incrementa:

$$A(r, \Theta) = A(r, \Theta) + 1$$

4. *Ricerca dei massimi locali*
   Le coppie (r, Θ) con alto valore nell'accumulatore rappresentano linee significative.
5. *Ricostruzione delle linee nell'immagine*
   Una volta trovata una coppia (r, Θ), la corrispondente retta viene tracciata nell'immagine.

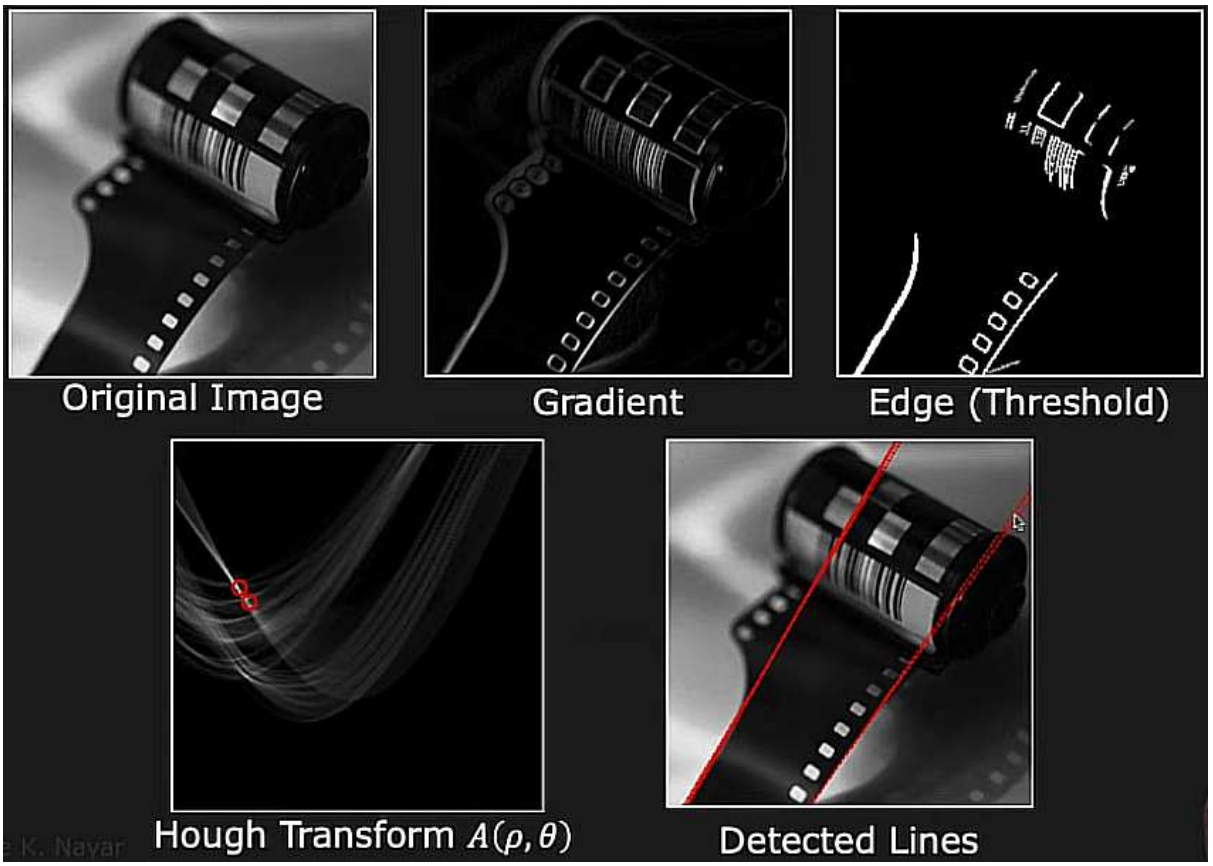


Figura 50: esempio di processo completo per il rilevamento delle linee

La trasformata di Hough tuttavia presenta un costo computazionale molto elevato: sono stati quindi affrontati diversi approcci per realizzare delle Trasformate di Hough di carattere Probabilistico. Queste, introducono ottimizzazioni basate sul campionamento casuale e sulla teoria della probabilità.

L'applicazione pratica più diffusa sfrutta il *random sampling*: votando solo su un sottoinsieme casuale dei pixel di bordo, la complessità si riduce drasticamente, mantenendo un'elevata accuratezza anche con solo il 2% dei dati originali.

Capitolo 3: Object Detection e Image Classification

L'obiettivo dell'object detection è localizzare e identificare specifiche istanze di oggetti noti all'interno di un'immagine. Nel tempo sono stati sviluppati approcci molto eterogenei per affrontare questo compito. [3]

Con l'avvento del deep learning l'object detection ha compiuto un salto di qualità: a differenza dei metodi classici, che facevano affidamento sull'estrazione manuale di caratteristiche locali (come i filtri di Haar, HOG o ORB), i modelli di deep learning integrano in un'unica architettura l'estrazione delle feature, la classificazione e la regressione delle bounding box.

In questo capitolo ci concentreremo proprio su queste architetture di deep learning, analizzando i modelli contemporanei per la rilevazione e classificazione degli oggetti nelle immagini.

Deep Learning e Reti Neurali Convoluzionali (CNN)

A differenza dei sistemi di machine learning tradizionali, il deep learning consente di estrarre automaticamente le caratteristiche rilevanti dai dati grezzi, costruendo rappresentazioni via via più astratte attraverso livelli successivi di trasformazioni non lineari.

Questi modelli apprendono a partire da un dataset costituito da dati etichettati (nel caso dell'apprendimento supervisionato) o non etichettati (per l'apprendimento non supervisionato), costruendo funzioni predittive capaci di mappare relazioni complesse tra input e output.

In particolare, il deep learning è efficace nell'identificazione di pattern significativi all'interno di grandi quantità di dati, potenziando i segnali utili e sopprimendo la variabilità irrilevante attraverso reti profonde caratterizzate da molteplici livelli nascosti.

Tra le varie architetture proposte una delle più influenti e utilizzate è la Convolutional Neural Network (CNN), introdotta nelle sue prime forme già negli anni '60 e oggi considerata uno dei modelli più rappresentativi nel campo della visione artificiale. Le CNN si sono affermate grazie alla loro efficacia in compiti complessi come classificazione e segmentazione di immagini, object detection, video processing, riconoscimento vocale e analisi del linguaggio naturale. [7]

Principi Fondamentali delle CNN [9]

Una CNN è tipicamente composta da quattro livelli fondamentali: convolutional layer, pooling layer, fully connected layer e non-linearity layer.

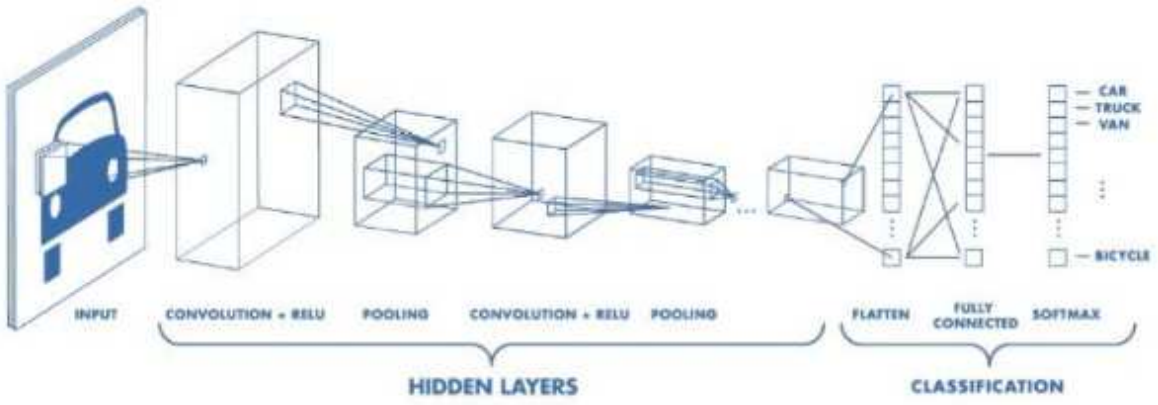


Figura 51: livelli fondamentali di una CNN

Il *convolutional layer* utilizza un insieme di kernel (o filtri) che vengono fatti "scorrere" sull'immagine per calcolarne la convoluzione ed estrarre le caratteristiche fondamentali; questi filtri funzionano esattamente come il filtro gaussiano o l'operatore di Prewitt e Sobel, tuttavia con una differenza fondamentale: i kernel della CNN non sono predefiniti, bensì appresi automaticamente dal modello per individuare bordi, texture e pattern visivi utili al compito.

La convoluzione può essere formalizzata come nell'equazione, dove ciascun valore della mappa di attivazione deriva da una somma pesata tra regione locale dell'immagine e kernel.

$$Activation\ map = Input * Filter$$
$$= \sum_{y=0}^{columns} \left(\sum_{x=0}^{rows} Input(x-p, y-q) Filter\ (x, y) \right)$$

Un livello convoluzionale è definito da tre parametri principali:

- *Kernel size*: la dimensione del filtro che viene fatto scorrere.
- *Stride*: il numero di pixel di cui il filtro si sposta a ogni passo.
- *Padding*: l'eventuale aggiunta di zeri attorno al bordo dell'immagine per controllare la dimensione in uscita.

Questi elementi determinano come la rete campiona l'immagine e quali caratteristiche visive può estrarre nelle fasi iniziali dell'elaborazione.

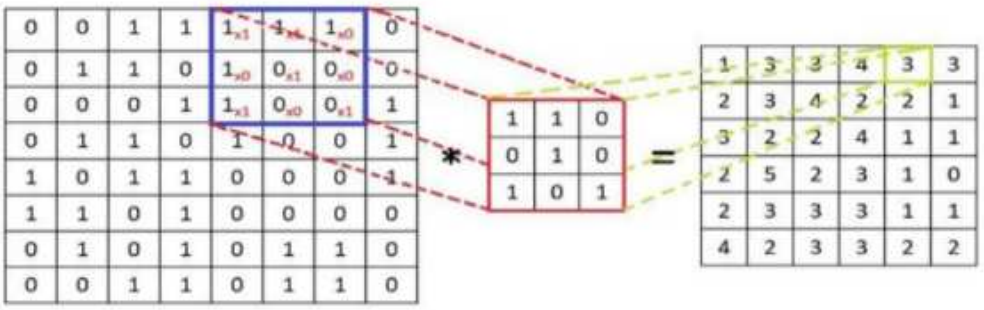


Figura 52: Strato convoluzionale

A valle dei livelli convoluzionali si trova il *pooling layer*, che riduce progressivamente la dimensionalità dell'attivazione mantenendo solo le informazioni più rilevanti. L'operazione più comune è il *max pooling*, che seleziona il valore massimo in un'area locale, riducendo la sensibilità ai piccoli spostamenti e al rumore.

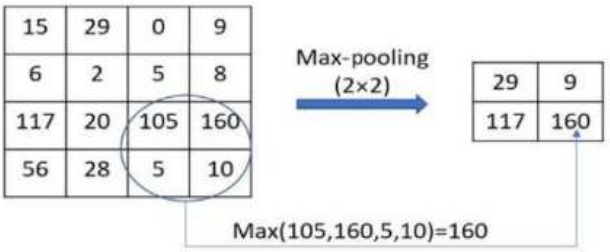


Figura 53: Max pooling

Il *fully connected layer* segue gli strati convoluzionali e di pooling e ha il compito di combinare le caratteristiche estratte nei livelli precedenti in modo non lineare per effettuare la classificazione finale. L'input tridimensionale viene "appiattito" in un vettore e passato a uno o più livelli completamente connessi.

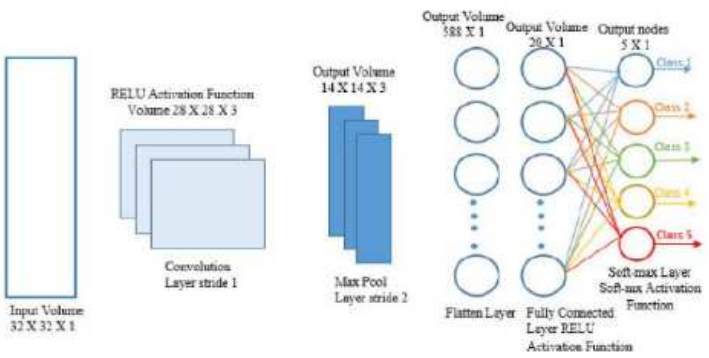


Figura 54: fully connected layer

Infine, ogni livello adotta una *funzione di attivazione*, che introduce non linearità nel modello. Tra le più utilizzate nelle CNN troviamo:

- *Sigmoid*, definita da

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

- *Tanh*, con intervallo di uscita $[-1, 1]$:

$$f(x)tanh = \frac{e^{x}+e^{-x}}{e^{x}+e^{-x}}$$

- *ReLU*, la più diffusa nelle CNN per efficienza computazionale:

$$f(x)ReLU = \max(0, x)$$

Il funzionamento di una CNN risiede nell'interazione sequenziale e gerarchica di tutti questi componenti. Il processo può essere descritto come una pipeline:

1. *Il convolutional layer* estrae caratteristiche locali dall'immagine tramite filtri appresi automaticamente. Nelle prime fasi questi filtri tendono a identificare bordi e orientamenti, mentre negli strati più profondi si specializzano in pattern sempre più astratti come texture, parti di oggetti o forme complesse. Il risultato di questa operazione è una activation map.
2. *La funzione di attivazione* viene applicata immediatamente dopo la convoluzione su tutti gli elementi della activation map singolarmente. Questo passaggio introduce la non linearità che permette alla rete di comporre le caratteristiche locali in strutture più complesse e discriminative. Senza attivazione, una sequenza di convoluzioni sarebbe matematicamente equivalente a una singola trasformazione lineare, e la rete sarebbe incapace di riconoscere oggetti complessi o separare classi visive.
3. *Il pooling layer* interviene per ridurre la dimensionalità delle mappe di attivazione, concentrando l'informazione ed eliminando variazioni dovute a piccoli spostamenti, rumore o deformazioni locali. In questo modo la rete mantiene la robustezza dell'informazione estratta, controllando al tempo stesso la complessità del modello.
4. *Il fully connected layer*, posto al termine della fase convoluzionale, combina tutte le caratteristiche apprese a livello globale. Qui la rete non analizza più regioni locali, ma utilizza un vettore di caratteristiche ad alto livello per effettuare la classificazione o la decisione finale.

In una rete neurale, l’input di ciascun livello può variare molto durante il training, rendendo più difficile l’apprendimento dei pesi.

Un'aggiunta cruciale nelle reti moderne è la Batch Normalization(o BN): poiché l'input di ciascun livello varia costantemente durante l'addestramento, la rete fatica a stabilizzare i pesi. La BN normalizza le attivazioni di ogni "batch" (gruppo di dati) affinché abbiano media 0 e varianza 1. Questo passaggio, inserito solitamente prima della funzione di attivazione, stabilizza il gradiente, velocizza la convergenza e risulta indispensabile per addestrare reti molto profonde.

MobileNet/EfficientNet [7]

MobileNet è un'architettura di rete neurale progettata per dispositivi mobili, sviluppata dal team di ricerca di Google e introdotta per la prima volta nel 2017.

Il suo obiettivo principale è garantire prestazioni elevate e bassa latenza per attività di classificazione e rilevamento di oggetti su smartphone, tablet e dispositivi con risorse computazionali limitate.

La caratteristica distintiva di MobileNet è l'uso di depthwise separable convolutions, una versione fattorizzata della convoluzione standard, che divide l'operazione in due fasi:

1. *Depthwise convolution*: applica un filtro separato per ciascun canale di input, filtrando le informazioni senza combinare i canali.
2. *Pointwise convolution (1×1)*: combina linearmente le uscite della depthwise convolution per produrre nuove feature map.

Questo approccio riduce drasticamente il numero di parametri e la complessità computazionale rispetto alle convoluzioni standard, mantenendo un'elevata accuratezza. Ad esempio, utilizzando convoluzioni 3×3 depthwise separable, MobileNet richiede 8-9 volte meno operazioni rispetto a una convoluzione tradizionale.

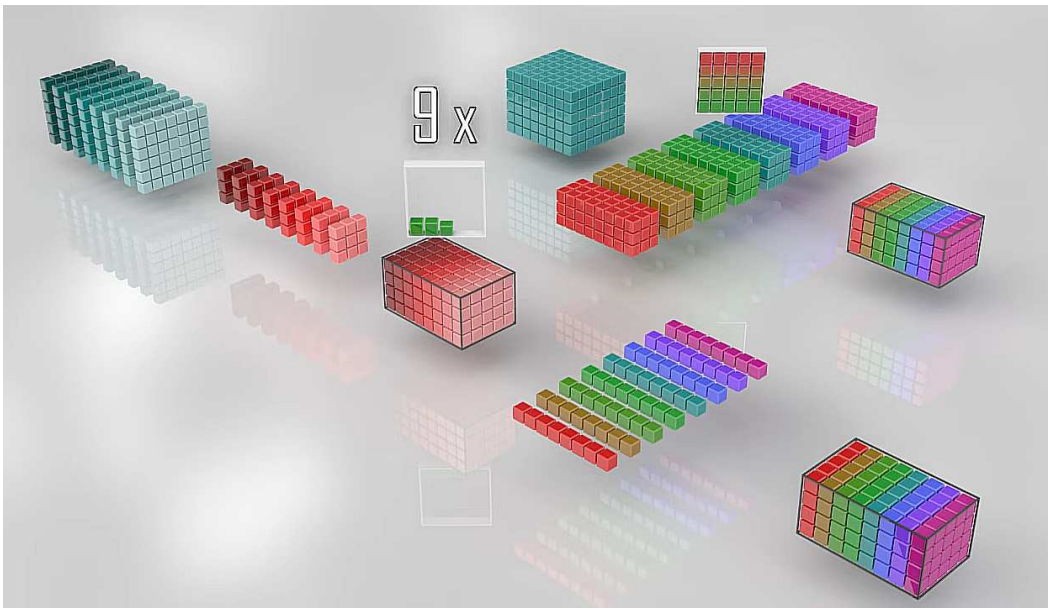


Figura 55: Rappresentazione visiva della convoluzione dw/s1 (a sinistra) contro la convoluzione classica

L'ultimo fully connected layer termina con softmax ossia una funzione che trasforma i valori numerici del fully connected layer in probabilità normalizzate associate a ciascuna classe. In questo modo, la rete può assegnare la previsione più probabile all'immagine in input.

La tabella seguente riassume la struttura completa del modello, mostrando dettagli su numero di strati, convoluzioni, funzioni di attivazione e pooling.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figura 56: architettura di MobileNet

Questa architettura compatta e flessibile rende MobileNet particolarmente adatta a scenari di transfer learning, come nel caso dell'utilizzo su piattaforme come Teachable Machine, dove reti pre-addestrate possono essere rapidamente adattate a nuovi compiti senza dover essere ricostruite da zero. [43]

YOLO: You Only Look Once [8]

YOLO, sviluppato da Joseph Redmon, ha introdotto per la prima volta un approccio end-to-end in tempo reale per la rilevazione di oggetti. A differenza di MobileNet, non serve solo a estrarre feature dall'immagine, ma a localizzare oggetti e predirne le classi.

Il nome "*You Only Look Once*" indica che la rete riesce a svolgere il compito di rilevazione con un'unica passata, a differenza dei metodi precedenti, che o usavano sliding windows ripetute centinaia di volte o dividevano l'operazione in più fasi.

YOLO unifica i passaggi di rilevazione rilevando simultaneamente tutti i bounding box. Per farlo, l'immagine di input viene divisa in una griglia S×S. Ogni cella della griglia predice B bounding box della stessa classe, con la propria confidence per C classi differenti.

Ogni predizione di bounding box comprende cinque valori:

- P_c: *confidence score*, indica quanto il modello è sicuro che il box contenga un oggetto e quanto sia preciso il box
- (b_x, b_y): coordinate del centro del box relative alla cella della griglia
- (b_w, b_h): larghezza e altezza del box relative all'immagine intera

L'output finale di YOLO è un tensor di dimensione S×S×(B×5 + C), a cui opzionalmente viene applicata la Non-Maximum Suppression (NMS) per rimuovere rilevazioni duplicate.

La Non-Maximum Suppression è una tecnica di post-processing utilizzata per ridurre il numero di bounding box sovrapposti, migliorando la qualità delle predizioni. Gli algoritmi di object detection spesso generano più box per lo stesso oggetto con punteggi di confidenza diversi. La NMS filtra i box ridondanti, mantenendo solo quelli più accurati.

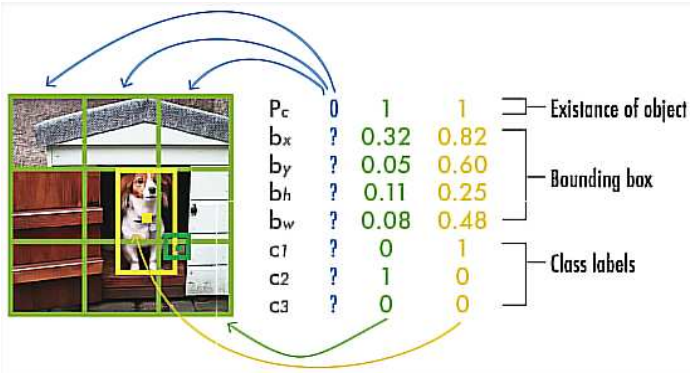


Figura 57: Yolo output prediction

La procedura generale del NMS è quella che segue:

Algorithm 1 Non-Maximum Suppression Algorithm

Require: Set of predicted bounding boxes B, confidence scores S, IoU threshold τ, confidence threshold T
Ensure: Set of filtered bounding boxes F

$F \leftarrow \emptyset$
Filter the boxes: $B \leftarrow \{b \in B \mid S(b) \geq T\}$
Sort the boxes B by their confidence scores in descending order
while $B \neq \emptyset$ **do**
  Select the box b with the highest confidence score
  Add b to the set of final boxes F: $F \leftarrow F \cup \{b\}$
  Remove b from the set of boxes B: $B \leftarrow B - \{b\}$
  for all remaining boxes r in B **do**
    Calculate the IoU between b and r: $iou \leftarrow IoU(b, r)$
    if $iou \geq \tau$ **then**
      Remove r from the set of boxes B: $B \leftarrow B - \{r\}$
    end if
  end for
end while

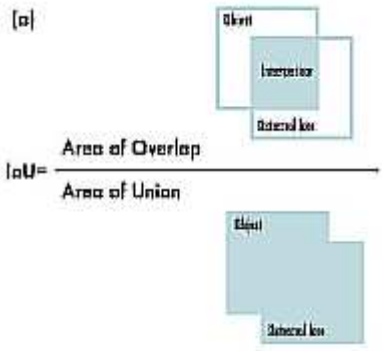


Figura 58: Intersection over Unit

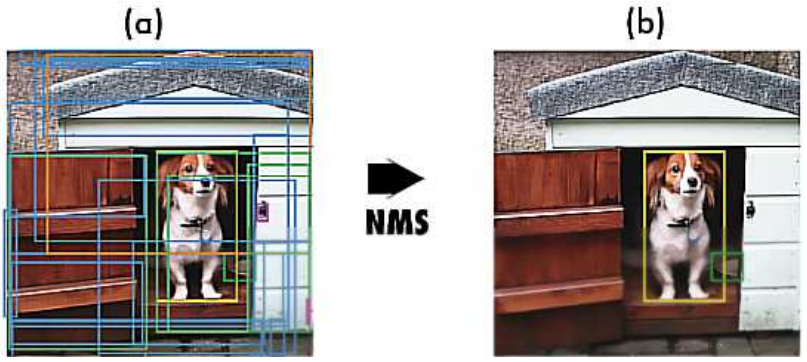


Figura 59: Output di Yolo filtrato post NMS

Questa tecnica è essenziale per evitare duplicazioni nelle predizioni e per migliorare la chiarezza dei risultati finali di object detection.

YOLO ha introdotto anche una versione più leggera, Fast YOLO, con soli 9 layer convoluzionali, ideale per applicazioni su dispositivi con limitate risorse computazionali.

	Type	Filters	Size/Stride	Output
	Conv	64	7 × 7/2	224 × 224
	Max Pool		2 × 2/2	112 × 112
	Conv	192	3 × 3/1	112 × 112
	Max Pool		2 × 2/2	56 × 56
1×	Conv	128	1 × 1/1	56 × 56
	Conv	256	3 × 3/1	56 × 56
	Conv	256	1 × 1/1	56 × 56
	Conv	512	3 × 3/1	56 × 56
	Max Pool		2 × 2/2	28 × 28
4×	Conv	256	1 × 1/1	28 × 28
	Conv	512	3 × 3/1	28 × 28
	Conv	512	1 × 1/1	28 × 28
	Conv	1024	3 × 3/1	28 × 28
	Max Pool		2 × 2/2	14 × 14
2×	Conv	512	1 × 1/1	14 × 14
	Conv	1024	3 × 3/1	14 × 14
	Conv	1024	3 × 3/1	14 × 14
	Conv	1024	3 × 3/2	7 × 7
	Conv	1024	3 × 3/1	7 × 7
	Conv	1024	3 × 3/1	7 × 7
	FC		4096	4096
	Dropout 0.5			4096
	FC		7 × 7 × 30	7 × 7 × 30

Figura 60: Architettura modello Yolov1 (FC = Fully connected, Dropout = “spegne” casualmente una percentuale di neuroni in un layer in ogni passo di aggiornamento dei pesi per evitare overfitting)

Dalla sua prima versione, negli ultimi anni Yolo ha subito diverse rielaborazioni e sono state realizzate nuove versioni più efficienti dello stesso modello partendo dalla stessa architettura.

Capitolo 4 – Implementazione della Pipeline, Risultati Sperimentali e Discussione

Il presente capitolo descrive nel dettaglio l'implementazione della pipeline di progetto, i cui fondamenti teorici sono stati introdotti nel primo capitolo.

La sezione 4.1 si articola in tre fasi principali:

- inizialmente verranno illustrate le metodologie di acquisizione dell'immagine, l'algoritmo di edge detection e la successiva rettificazione prospettica della scacchiera.
- Proseguendo, la terza e la quarta sottosezione esamineranno i due approcci metodologici adottati per il riconoscimento e la localizzazione dei pezzi sulla scacchiera.
- Infine, verranno analizzate le procedure di organizzazione dei dati necessarie per l'interfacciamento con il motore di analisi scacchistico.

La sezione 4.2 è invece dedicata alla fase di validazione:

- verranno dapprima definite le metriche di valutazione
- successivamente invece verranno discussi i risultati sperimentali ottenuti dai test condotti sulle due pipeline proposte.

4.1 Implementazione della Pipeline su Android

4.1.1 Acquisizione dell'Immagine e Pre-Processing

CameraX e il formato YUV: [26]

L'acquisizione delle immagini avviene tramite la libreria CameraX, che fornisce un flusso di frame in formato YUV_420_888.

Questo formato è composto da tre buffer separati:

- Y Luminanza
- U Crominanza blu
- V Crominanza rossa

Ogni buffer agisce come un'area di memoria temporanea dedicata che organizza i dati grezzi dei pixel in piani distinti, permettendo di accedere e processare le informazioni di luce e colore dei frame in modo indipendente.

Questa struttura è estremamente efficiente perché l'occhio umano è molto sensibile alla luminanza (rapporto tra l'intensità luminosa emessa da una sorgente nella direzione dell'osservatore e l'area apparente della superficie emittente) e molto meno alla crominanza (differenze tra le coordinate cromatiche di un corpo illuminato e quelle di un corpo di riferimento, di uguale luminanza).

Si può quindi ridurre la risoluzione dei canali colore senza perdere qualità percepita: mentre il buffer Y mantiene un rapporto 1:1 con la griglia dei pixel (piena risoluzione spaziale), i buffer U e V subiscono un sottocampionamento che dimezza sia la larghezza che l'altezza dell'informazione cromatica.

Esistono diversi tipi di sottocampionamento: in particolare il sottocampionamento 4:2:0 utilizza, data una matrice di riferimento di 4 x 2 pixel, un sistema di campionamento che ottimizza la distribuzione spaziale del colore.

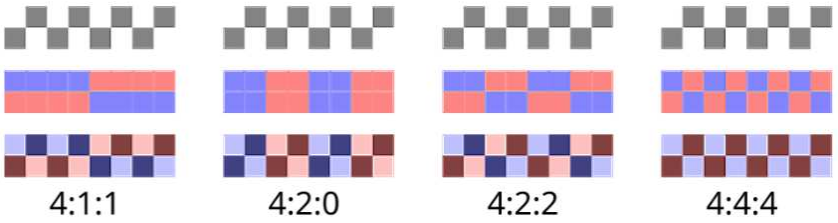


Figura 61: Esempi di sottocampionamento

Come mostrato nella figura 61, la notazione a tre cifre definisce il rapporto tra i canali per il formato 4:2:0 :

- Il primo numero rappresenta il riferimento di campionatura orizzontale della Luminanza (Y), indicando che in una riga di 4 pixel tutti i valori di definizione luminosa vengono mantenuti (piena risoluzione).
- Il secondo numero indica il fattore cromatico per la prima riga della matrice: in questo caso, vengono prelevati solo due campioni di crominanza (Cb, Cr) per ogni quattro pixel di luminanza.
- Il terzo numero indica che nella seconda riga della matrice non vengono estratti nuovi campioni cromatici rispetto alla prima.

Le tre cifre finali nella sigla YUV_420_888 indicano la profondità di ciascun buffer, che in questo caso è di 8 bit (avremo quindi 256 livelli per ogni canale).

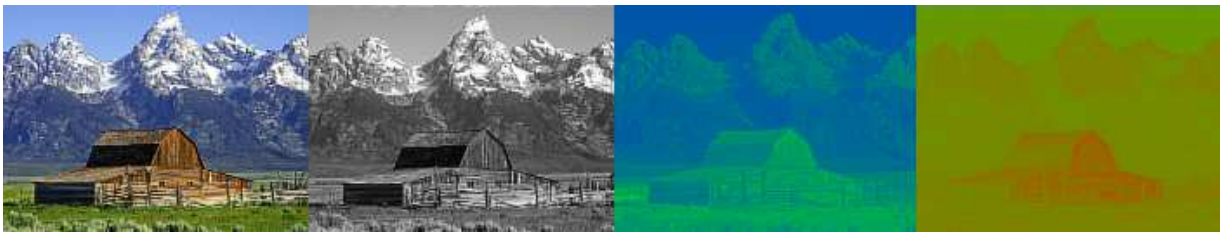


Figure 62 63 64 65: Partendo da sinistra: immagine originale, canale Y dell'immagine, canale U, canale V

Per utilizzare le immagini acquisite attraverso la libreria di CameraX si è quindi reso necessario convertire questo formato in una matrice. Questo è stato fatto attraverso l'extension function ad ImageProxy seguente:

```
fun ImageProxy.toMat(): Mat {
    // Verifica che il formato sia effettivamente YUV_420_888 (8 bit per canale, 3 piani separati).
    if (format != android.graphics.ImageFormat.YUV_420_888) {
        throw IllegalArgumentException("Image format not supported")
    }

    // Estrazione dei tre buffer (piani) di memoria: 0=Luminanza (Y), 1=U (Crominanza blu), 2=V (Crominanza rossa)
    val yBuffer: ByteBuffer = planes[0].buffer
    val uBuffer: ByteBuffer = planes[1].buffer
    val vBuffer: ByteBuffer = planes[2].buffer

    // Calcolo della quantità di byte disponibili in ogni buffer
    val ySize = yBuffer.remaining()
    val uSize = uBuffer.remaining()
    val vSize = vBuffer.remaining()

    // Creazione di un array unico (NV21) che conterrà tutti i dati.
    // La dimensione è la somma dei tre buffer (Y + U + V).
    val nv21 = ByteArray(ySize + uSize + vSize)

    // Copiamo l'intero buffer Y all'inizio dell'array nv21 (posizione 0).
    // Questo canale è a piena risoluzione (1:1 con i pixel del frame).
    yBuffer.get(nv21, 0, ySize)

    // Iniziamo a scrivere i dati del colore subito dopo la fine dei dati Y.
    var dstOffset = ySize
```

```
        // Poiché nel formato 4:2:0 il colore è dimezzato sia in larghezza che in altezza,
        // il ciclo itera su metà delle righe e metà delle colonne dell'immagine originale.
        val rows = height / 2
        val cols = width / 2

        // Parametri di memoria hardware:
        // rowStride: quanti byte saltare per passare alla riga successiva nel buffer.
        // pixelStride: quanti byte saltare tra un campione e l'altro (spesso 2 se i dati sono intrecciati).
        val rowStride = planes[1].rowStride
        val pixelStride = planes[1].pixelStride

        for (row in 0 ≤ until < rows) {
            for (col in 0 ≤ until < cols) {
                // Calcolo della posizione esatta del byte nel buffer originale,
                // tenendo conto degli spazi vuoti (padding) lasciati dall'hardware.
                val uIndex = row * rowStride + col * pixelStride
                val vIndex = row * rowStride + col * pixelStride

                if (dstOffset < nv21.size - 1) {
                    // Il formato NV21 vuole prima il valore V e poi il valore U.
                    // Prendiamo un campione di rosso (V) e uno di blu (U) e li mettiamo in fila.
                    nv21[dstOffset++] = vBuffer.get(vIndex)
                    nv21[dstOffset++] = uBuffer.get(uIndex)
                }
            }
        }

        // CREAZIONE MATRICE OPEN_CV:
        // Creiamo una matrice "grezza" alta Height (Y) + Height/2 (U+V).
        // Usiamo CV_8UC1: 8 bit (8U), Unsigned, 1 solo canale di byte grezzi.
        val yuvMat = Mat(height + height / 2, width, CvType.CV_8UC1)

        // Inseriamo l'array di byte ordinati (NV21) dentro la matrice.
        yuvMat.put(0, 0, nv21)

        // CONVERSIONE SPAZIO COLORE:
        // Trasformiamo i dati YUV ordinati in formato NV21 in una vera immagine a colori RGBA.
        // Qui OpenCV applica le formule matematiche per ogni pixel.
        val rgbaMat = Mat()
        Imgproc.cvtColor(yuvMat, rgbaMat, Imgproc.COLOR_YUV2RGBA_NV21)

        return rgbaMat
    }
```

1. *Estrazione dei piani Y, U e V:*

La funzione ottiene i tre buffer che compongono l’immagine:

- il piano Y, che ha la stessa risoluzione dell’immagine;
- i piani U e V, ciascuno con metà risoluzione in entrambe le direzioni.

Il piano Y può essere copiato direttamente perché contiene un byte per ogni pixel, mentre i piani U e V devono essere ricostruiti con più attenzione a causa di stride e pixel stride variabili.

2. *Ricostruzione manuale del formato NV21:*

La funzione ricostruisce un buffer nel formato NV21, che unisce:

- tutti i valori Y consecutivi,
- seguiti da valori di crominanza interleaved nel pattern V–U–V–U….

La ricostruzione dei piani UV richiede un doppio ciclo che scorre riga per riga, tenendo conto dello stride dinamico dei buffer forniti da CameraX (ossia la distanza reale tra pixel consecutivi in memoria).

3. *Creazione di un Mat YUV:*

Una volta costruito il buffer NV21, la funzione lo inserisce in un oggetto OpenCV Mat con un'unica componente per pixel.
La dimensione di questa matrice corrisponde all'altezza dell'immagine più un ulteriore 50%, dato che il formato 4:2:0 utilizza un solo byte per pixel per Y, ma un byte ogni due pixel per i componenti UV combinati.

4. *Conversione da NV21 a RGBA tramite OpenCV:*

L'ultimo passaggio utilizza OpenCV per convertire la matrice YUV in una vera immagine a colori:

- OpenCV interpreta il buffer come NV21,
- decodifica luminanza e crominanza,
- e produce una nuova matrice in formato RGBA, pienamente compatibile con tutta la pipeline di elaborazione successiva.

Il risultato finale è un oggetto Mat che rappresenta l'immagine esattamente come apparirebbe in RGB, pronto per essere filtrato, analizzato e utilizzato dagli algoritmi OpenCV.

ROI:

Per migliorare l'affidabilità del rilevamento della scacchiera e ridurre il rumore di fondo, è stato introdotto un sistema di Region of Interest (ROI) che opera su due livelli distinti ma complementari: uno grafico e uno computazionale.

La prima ROI è visiva ed è implementata nell'interfaccia ed oscura le aree periferiche dell'anteprima video lasciando libero soltanto un rettangolo centrale. Questa maschera non modifica in alcun modo i dati dell'immagine, ma serve a guidare l'utente a posizionare correttamente la scacchiera nel punto esatto in cui avviene il rilevamento. In questo modo si evita che la scacchiera venga rilevata in zone marginali, troppo vicine ai bordi dell'inquadratura o caratterizzate da distorsioni prospettiche più marcate.

Parallelamente, è stata introdotta una ROI computazionale, applicata direttamente alla matrice dell'immagine all'interno della pipeline OpenCV. Questa ROI è definita attraverso un ritaglio (submat) che seleziona solo la porzione centrale del frame, scartando i margini sinistro, destro, superiore e inferiore. Tutte le fasi successive vengono eseguite esclusivamente su questa porzione ridotta dell'immagine.

Ciò comporta numerosi vantaggi: riduce significativamente il rumore, limita la presenza di linee spurie provenienti dallo sfondo, migliora la stabilità del rilevamento e diminuisce il carico computazionale.

Figura 66: ROI visiva implementata nell'applicazione. Il rettangolo centrale rappresenta la sottomatrice su cui vengono svolte le operazioni di calcolo.

Filtering, smoothing, sharpening:

Per la fase di pre elaborazione delle immagini acquisite dalla preview della fotocamera sono stati scelti i seguenti passaggi:

1. Il primo consiste nella conversione dell'immagine in scala di grigi, necessaria per semplificare il contenuto visivo e ridurre la complessità dell'elaborazione successiva.
2. Successivamente viene applicato un *median blur*, un filtro robusto in grado di eliminare il rumore impulsivo senza degradare in maniera significativa i contorni. Questo è particolarmente utile nelle acquisizioni video da smartphone, dove la presenza di grana, artefatti di compressione o condizioni di luce variabile possono compromettere la pulizia dell'immagine.
3. Dopo la riduzione del rumore, viene applicato un *filtro di sharpening* tramite convoluzione con un kernel personalizzato. Questo kernel enfatizza le transizioni di intensità e aumenta la nitidezza dei bordi della scacchiera, rendendo più marcate le linee orizzontali e verticali del pattern. Lo sharpening serve quindi a compensare eventuali sfocature dovute al movimento della mano, alla distanza di messa a fuoco o alla risoluzione ridotta del frame. L'obiettivo è fornire al rilevatore di bordi una struttura più definita, evitando che linee troppo morbide o degradate vengano perse. [24]
4. Infine, il rilevamento dei bordi viene eseguito tramite un *Canny adattivo*. Invece di utilizzare soglie fisse, la soglia inferiore viene calcolata dinamicamente a partire dalla luminosità media della ROI, mentre quella superiore è impostata come multiplo della prima. Questa scelta è motivata dal fatto che l'illuminazione del pattern può variare sensibilmente da scena a scena: un ambiente molto luminoso produce bordi più marcati, mentre una scarsa illuminazione provoca contorni più deboli. Utilizzare soglie statiche renderebbe il rilevamento fragile e dipendente dalle condizioni ambientali, mentre un approccio adattivo permette a Canny di funzionare in modo coerente indipendentemente dalla luminosità del frame.

L'esito è un set di bordi più stabile e affidabile, che costituisce la base per il successivo rilevamento delle linee tramite Hough.

4.1.2 Localizzazione della Scacchiera e Rettifica Prospettica

Rilevamento linee con Hough

Per individuare le linee della scacchiera all'interno della ROI, ho scelto di utilizzare la *Probabilistic Hough Transform (HoughLinesP)*: a differenza della versione classica, la variante probabilistica restituisce direttamente segmenti finiti anziché parametri infiniti nel piano (ρ, θ), riducendo drasticamente il costo computazionale e consentendo l'elaborazione in tempo reale.

Una volta ottenuti i segmenti, ho fatto in modo di classificarli in linee orizzontali e linee verticali sulla base del loro angolo. Calcolando l'angolo tramite arctan2, è possibile discriminare in modo semplice e diretto l'orientazione del segmento: linee con un angolo prossimo a 0° vengono considerate orizzontali, mentre quelle vicine a ±90° vengono classificate come verticali.

Questa separazione è fondamentale perché semplifica notevolmente i passaggi successivi della pipeline, permettendo di ragionare su due famiglie indipendenti di linee, ciascuna con una direzione attesa e un ruolo preciso nel pattern della scacchiera.

Per ottenere un set finale ordinato, stabile e privo di duplicati, le linee vengono infine sottoposte a un processo di raggruppamento basato sulla distanza reciproca e sulla loro disposizione geometrica: ho implementato il metodo `groupLines()` per combinare linee che sono quasi sovrapposte tra loro (tipico quando Hough rileva più segmenti sulla stessa retta) e applicare un filtro compatto, eliminando duplicati, linee frammentate o segmenti troppo vicini per essere distinti.

L'obiettivo è produrre un insieme di linee pulito e ben strutturato, in cui ogni riga o colonna della scacchiera è rappresentata da una singola linea coerente.

Questo passaggio è cruciale perché dà stabilità all'intero processo di ricostruzione geometrica: un clustering corretto delle linee corrisponde a una corretta interpretazione del pattern, e permette alla pipeline di proseguire verso la ricostruzione dell'intersezione delle linee e la determinazione delle celle della scacchiera.

Calcolo delle intersezioni e stabilizzazione temporale

Una volta identificate e raggruppate le linee orizzontali e verticali principali, il passo successivo consiste nel determinare le intersezioni che definiscono i vertici della scacchiera.

Il codice calcola le quattro intersezioni estreme combinando le linee più esterne in ciascuna direzione: la prima e l'ultima linea orizzontale vengono incrociate con la prima e l'ultima linea verticale per ottenere rispettivamente le coordinate top-left, top-right, bottom-left e bottom-right. Questo approccio garantisce che i punti identificati rappresentino gli angoli della griglia, fornendo così una base coerente per ulteriori trasformazioni geometriche.

Per mantenere la stabilità della rilevazione nel tempo, ogni punto calcolato viene traslato rispetto alla ROI originale per ottenere coordinate assolute rispetto all'intera immagine.

Successivamente, viene definito un *bounding rectangle* che racchiude tutte e quattro le intersezioni. Se il movimento è entro una soglia il rettangolo corrente viene aggiunto ad uno storico dei rettangoli di cui viene calcolata una media mobile per ottenere un aggiornamento smussato e stabile della posizione della scacchiera.

Questo passaggio garantisce che la pipeline successiva riceva sempre punti coerenti e stabili, minimizzando fluttuazioni tra frame consecutivi e rendendo la rilevazione robusta anche in condizioni dinamiche o con rumore visivo.

Infine, all'interno dei frame della preview della fotocamera vengono disegnati quattro segmenti che collegano le coordinate delle intersezioni, fornendo un feedback visivo immediato all'utente e mostrando chiaramente quali angoli l'algoritmo ha rilevato.

Figura 67: Riconoscimento dei bordi della scacchiera e feedback visivo all'utente

Calcolo della trasformata prospettica e rettificazione della prospettiva

Una volta completata la stabilizzazione dei vertici della scacchiera, le coordinate individuate sulla matrice di anteprima (*preview*) vengono sottoposte a una procedura di riscalamento (*scaling*). Tale operazione è necessaria per proiettare i punti rilevati sul frame a bassa risoluzione del flusso video verso le dimensioni effettive dell'immagine acquisita ad alta risoluzione. Questo processo di mappatura garantisce che la geometria della scacchiera, calcolata in tempo reale, rimanga coerente e accurata anche sullo scatto finale destinato all'elaborazione.

Su quest'ultima immagine quindi, attraverso i nuovi vertici calcolati, si applica una trasformazione prospettica per ottenere una versione quadrata frontale della scacchiera, indipendente dall'angolo di ripresa della fotocamera.

Figura 68: Prospettiva della scacchiera rettificata

Di seguito è presente il codice utilizzato per il riconoscimento e rettifica della scacchiera:

```
fun findChessboardCorners(frameMat: Mat): ChessboardDetectionResult {
    val gray = Mat()
    val edges = Mat()
    val lines = Mat()

    var isDetected = false
    var orderedPoints: List<Point> = emptyList()
    var roiMat: Mat? = null

    try {

        val width = frameMat.cols()
        val height = frameMat.rows()

        // Definisce i margini: 15% sui lati, 23% sopra e sotto
        val marginX = (width * 0.15).toInt()
        val marginY = (height * 0.23).toInt()

        // Crea il rettangolo di ritaglio
        val roiRect = Rect(
            marginX,
            marginY,
            width - 2 * marginX,
            height - 2 * marginY
        )

        // Estrae la sottomatrice della Region Of Interest
        roiMat = frameMat.submat(roiRect)

        // Conversione in scala di grigi per ridurre la dimensionalità
        Imgproc.cvtColor(roiMat, gray, Imgproc.COLOR_RGBA2GRAY)

        // Filtro mediano con kernel 9x9
        Imgproc.medianBlur(gray, gray, 9)

        // Definizione del kernel di Sharpening
        val kernelArray = doubleArrayOf(
            -1.0, -1.0, -1.0,
            -1.0, 9.0, -1.0,
            -1.0, -1.0, -1.0
        )
        val kernelSharpening = Mat(3, 3, CvType.CV_64F).apply { put(0, 0, *kernelArray) }

        // Applicazione della convoluzione spaziale per rendere i bordi più netti
        Imgproc.filter2D(gray, gray, -1, kernelSharpening)
        kernelSharpening.release()

        // Calcola la luminosità media della ROI
        val mean = Core.mean(gray).`val`[0]

        // Algoritmo adattivo per le soglie di Canny
        val lower = max(45.0, mean * 0.8)
        val upper = lower * 4

        // Rilevamento dei bordi
        Imgproc.Canny(gray, edges, lower, upper)

        // Trasformata di Hough Probabilistica
        Imgproc.HoughLinesP(edges, lines, 1.0, Math.PI / 180, 100, 110.0, 40.0)

```

```
        val horizontals = mutableListOf<Line>()
        val verticals = mutableListOf<Line>()

        // --- Separazione ---
        for (i in 0 ≤ until < lines.rows()) {
            val l = lines.get(i, 0)
            val x1 = l[0]; val y1 = l[1]
            val x2 = l[2]; val y2 = l[3]

            // Calcolo dell'angolo del segmento tramite arcotangente
            val angle = atan2(y2 - y1, x2 - x1) * 180 / Math.PI

            // Classificazione basata sull'angolo
            if (abs(angle) < 10) horizontals.add(Line(x1, y1, x2, y2))
            else if (abs(angle - 90) < 15 || abs(angle + 90) < 15)
                verticals.add(Line(x1, y1, x2, y2))
        }

        // --- Raggruppamento e Filtro Compatto ---
        val groupedHorizontals = groupLines(horizontals, "horizontal")
        val groupedVerticals = groupLines(verticals, "vertical")

        if (groupedHorizontals.isNotEmpty() && groupedVerticals.isNotEmpty()) {

            // 1. ORDINAMENTO: Fondamentale per la sliding window.
            // Ordino le linee da sinistra a destra (verticali) e dall'alto in basso (orizzontali).
            val sortedHorizontal = groupedHorizontals.sortedBy { (it.y1 + it.y2) / 2.0 }
            val sortedVertical = groupedVerticals.sortedBy { (it.x1 + it.x2) / 2.0 }

            // 2. SELEZIONE DELLE "MIGLIORI" 9:
            // Se Hough ha rilevato 15 linee orizzontali,
            // findBestGroup sceglierà le 9 più vicine tra loro che probabilmente formano la scacchiera.
            val limitedHorizontal = findBestGroup(sortedHorizontal, 9, "horizontal")
            val limitedVertical = findBestGroup(sortedVertical, 9, "vertical")

            // 3. VALIDAZIONE MINIMA:
            // Per poter calcolare almeno i 4 angoli, servono almeno 2 linee per direzione.
            if (limitedHorizontal.size >= 2 && limitedVertical.size >= 2) {

                // Calcolo delle intersezioni tra le linee più esterne
                val topLeft = intersect(limitedHorizontal.first(), limitedVertical.first())
                val topRight = intersect(limitedHorizontal.first(), limitedVertical.last())
                val bottomLeft = intersect(limitedHorizontal.last(), limitedVertical.first())
                val bottomRight = intersect(limitedHorizontal.last(), limitedVertical.last())

                if (topLeft != null && topRight != null && bottomLeft != null && bottomRight != null) {

                    // Traslazione: riporta i punti dal sistema di coordinate della ROI a quello del Frame intero
                    val offset = Point(marginX.toDouble(), marginY.toDouble())

                    val absoluteTopLeft = Point(topLeft.x + offset.x, topLeft.y + offset.y)
                    val absoluteTopRight = Point(topRight.x + offset.x, topRight.y + offset.y)
                    val absoluteBottomLeft = Point(bottomLeft.x + offset.x, bottomLeft.y + offset.y)
                    val absoluteBottomRight = Point(bottomRight.x + offset.x, bottomRight.y + offset.y)

                    // 4 LOGICA DI STABILITÀ (Smoothing)
                    val xMin = minOf(absoluteTopLeft.x, absoluteTopRight.x, absoluteBottomLeft.x, absoluteBottomRight.x)
                    val yMin = minOf(absoluteTopLeft.y, absoluteTopRight.y, absoluteBottomLeft.y, absoluteBottomRight.y)
                    val xMax = maxOf(absoluteTopLeft.x, absoluteTopRight.x, absoluteBottomLeft.x, absoluteBottomRight.x)
                    val yMax = maxOf(absoluteTopLeft.y, absoluteTopRight.y, absoluteBottomLeft.y, absoluteBottomRight.y)

                    val width = (xMax - xMin).toInt()
                    val height = (yMax - yMin).toInt()

                    // Calcolo del rettangolo contenitore corrente
                    val currentBoundingRect = Rect(xMin.toInt(), yMin.toInt(), width, height)
```

```
                    // Misura lo spostamento rispetto alla posizione precedente
                    val dx = (currentBoundingRect.x - stableRect.x).absoluteValue.toDouble()
                    val dy = (currentBoundingRect.y - stableRect.y).absoluteValue.toDouble()

                    // Se il movimento è piccolo (stabile), aggiunge allo storico
                    val isMovementStable = rectHistory.isEmpty() || (dx < MIN_MOVEMENT_THRESHOLD && dy < MIN_MOVEMENT_THRESHOLD)
                    if (isMovementStable) {
                        rectHistory.add(currentBoundingRect)
                        if (rectHistory.size > MAX_HISTORY) rectHistory.removeAt(0)
                    } else {
                        Log.w("Detector", "Box jumped!")
                    }

                    // Media mobile: la posizione finale è la media degli ultimi N frame
                    if (rectHistory.isNotEmpty()) stableRect = calculateAverageRect(rectHistory)

                    // Output + salvataggio angoli
                    val currentCorners = listOf(absoluteTopLeft, absoluteTopRight, absoluteBottomLeft, absoluteBottomRight)
                    orderedPoints = orderCorners(currentCorners)

                    stableCorners = orderedPoints.toTypedArray()
                    isDetected = true
                }
            }
        }
        roiMat.release()
    } catch (e: Exception) {
        Log.e("Detector", "Error while looking for the chessboard", e)
    } finally {
        gray.release()
        edges.release()
        lines.release()
    }

    return ChessboardDetectionResult(stableCorners.toList(), isDetected)
}
```

4.1.3 Riconoscimento dei Pezzi

Per affrontare il problema dell'object detection dei pezzi degli scacchi, ho deciso di confrontare due approcci differenti ed ho quindi ideato ed implementato due diverse pipeline.

La prima pipeline segue un metodo basato su un modello tensor flow lite addestrato tramite Teachable Machine (basato quindi su architettura MobileNet), ossia una piattaforma online per l'addestramento di modelli tensor flow. [14]

L'addestramento è stato eseguito utilizzando circa 200 immagini per ciascuna delle 7 classi (una per ogni tipo di pezzo e una per la classe "empty") prelevate da un dataset disponibile su Roboflow, e usando anche foto scattate a scacchiere sul sito di Chess.com. [13] [21]

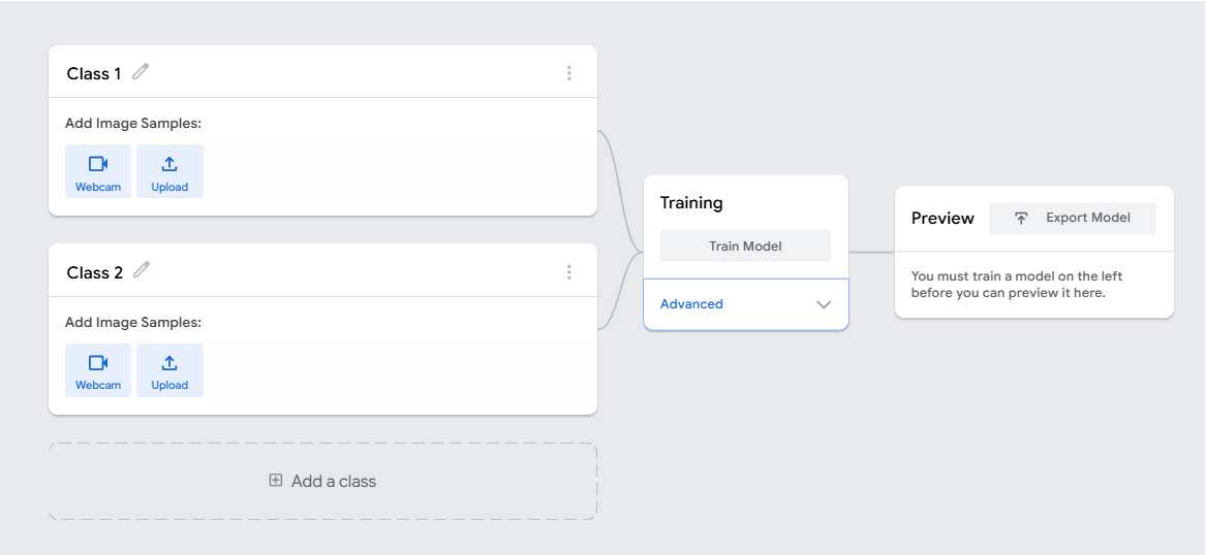


Figura 69: Interfaccia per l'addestramento di modelli TFLite su Teachable Machine

Per rendere il classificatore più robusto, sono state applicate tecniche di data augmentation, introducendo variazioni di rotazione, traslazioni, zoom, shear, flip orizzontale, luminosità e spostamenti di canale nel dataset di train, così da aumentare la capacità del modello di generalizzare su immagini diverse da quelle di training.

La seconda pipeline, invece, adotta un approccio più rapido e "tutto-in-uno", basato su un modello YOLO pre addestrato trovato online. Questo modello è in grado di riconoscere contemporaneamente tutti i tipi di pezzi e i loro colori, senza richiedere ulteriori elaborazioni sull'immagine, permettendo una classificazione immediata e semplificando notevolmente la pipeline di rilevamento. [15]

4.1.3.A Pipeline A – Classificatore MobileNet

La pipeline di analisi basata su TFLite opera attraverso diversi passaggi:

1. L'immagine viene inizialmente convertita in un oggetto Mat di OpenCV e trasformata nello spazio colore YCrCb, separando poi i tre canali e sul canale della luminanza Y viene applicata una CLAHE. I tre canali vengono poi ricombinati e riconvertiti in RGB, ottenendo un'immagine bilanciata pronta per la classificazione. Il motivo di queste conversioni è che il modello TFLite che ho utilizzato è stato addestrato su dati a cui ho applicato la stessa tecnica di contrast enhancement.
2. Successivamente, l'immagine viene suddivisa in 64 sotto-immagini, corrispondenti alle caselle della scacchiera.
3. Ogni sotto-immagine viene ridimensionata (upscaling) a 224×224 pixel e passata al classificatore TFLite addestrato per identificare i tipi di pezzi/caselle vuote.
4. Di seguito, per ogni casella non vuota, si utilizza un semplice algoritmo per determinare il colore del pezzo, distinguendo tra bianco o nero.
5. Il risultato finale per ogni casella viene quindi combinato in un'etichetta completa, come ad esempio white_pawn o empty, e salvato in una matrice 8x8 che rappresenta lo stato attuale della scacchiera.

Questa pipeline, pur richiedendo un'elaborazione casella per casella, garantisce un controllo molto fine sul processo di classificazione e permette di integrare tecniche di preprocessing avanzate (come CLAHE) per aumentare la robustezza e l'affidabilità del modello anche in condizioni di illuminazione variabile o con riflessi sulla scacchiera.

Di seguito il cuore del codice della classificazione tramite modello TFLite:

```
/**
 * Analizza l'immagine rettificata della scacchiera e restituisce una matrice 8x8
 * contenente le etichette dei pezzi (es. "white_pawn", "black_rook", "empty").
 */
fun analyzeWarpedImage(
    warpedBitmap: Bitmap,          // Immagine rettificata della scacchiera (solo la griglia)
    classifier: ImageClassifier    // Il modello TFLite caricato
): Array<Array<String>> {

    // Inizializza la matrice dei risultati 8x8 con valori di default
    val resultMatrix = Array(8) { Array(8) { "?" } }

    // Calcola la dimensione in pixel di una singola cella (assumendo una griglia quadrata)
    val squareSize = warpedBitmap.width / 8
```

```
// Allocazione matrici OpenCV per la gestione della memoria
val fullRgbaMat = Mat()
val fullRgbMat = Mat()
val ycrcbMat = Mat()
val resizedMat = Mat()

// Configurazione CLAHE: Clip Limit 3.0 per evitare rumore eccessivo, Grid 8x8
val clahe = Imgproc.createCLAHE(3.0, Size(8.0, 8.0))
val ycrcbChannels = ArrayList<Mat>(3)

// Bitmap riutilizzabile per l'input del modello TFLite (224x224 è standard per MobileNet)
val squareBitmap = createBitmap(224, 224)

try {
    // Conversione da Bitmap Android a Matrice OpenCV
    Utils.bitmapToMat(warpedBitmap, fullRgbaMat)

    // Rimozione del canale Alpha (Trasparenza) non necessario per l'elaborazione
    Imgproc.cvtColor(fullRgbaMat, fullRgbMat, Imgproc.COLOR_RGBA2RGB)

    // --- INIZIO PRE-ELABORAZIONE ILLUMINAZIONE (CLAHE) ---

    // 1. Conversione nello spazio colore YCrCb per separare la luminanza (Y) dal colore (Cr, Cb)
    Imgproc.cvtColor(fullRgbMat, ycrcbMat, Imgproc.COLOR_RGB2YCrCb)

    // 2. Divisione dei canali: [0]=Y (Luce), [1]=Cr, [2]=Cb
    Core.split(ycrcbMat, ycrcbChannels)

    // 3. Applicazione dell'equalizzazione adattiva (CLAHE) SOLO sul canale Y.
    //    Questo migliora il contrasto locale senza distorcere i colori dei pezzi.
    val claheYChannel = Mat()
    clahe.apply(ycrcbChannels[0], claheYChannel)

    // 4. Sostituzione del canale Y originale con quello equalizzato
    ycrcbChannels[0].release()
    ycrcbChannels[0] = claheYChannel

    // 5. Unione dei canali (Y migliorato + Cr originale + Cb originale)
    Core.merge(ycrcbChannels, ycrcbMat)

    // 6. Riconversione in RGB per la visualizzazione e l'inferenza
    Imgproc.cvtColor(ycrcbMat, fullRgbMat, Imgproc.COLOR_YCrCb2RGB)

    // --- INIZIO SCANSIONE GRIGLIA 8x8 ---

    for (row in 0 ≤ until < 8) {
        for (col in 0 ≤ until < 8) {

            // Definizione della Region of Interest (ROI) per la cella corrente
            val roi = Rect(col * squareSize, row * squareSize, squareSize, squareSize)

            // Estrazione della sottomatrice (senza copiare i dati, per efficienza)
            val squareMat = fullRgbMat.submat(roi)

            // Ridimensionamento a 224x224 px richiesto dal modello TFLite
            Imgproc.resize(squareMat, resizedMat, Size(224.0, 224.0))
```

```
                // Conversione finale in Bitmap per l'interprete TensorFlow
                Imgproc.cvtColor(resizedMat, resizedMat, Imgproc.COLOR_RGB2RGBA)
                Utils.matToBitmap(resizedMat, squareBitmap)

                // 7. INFERENZA TIPO PEZZO (Reti Neurali)
                // Il modello TFLite classifica la forma (es. "pawn", "rook", "empty")
                val pieceLabel = classifier.classify(squareBitmap)

                // 8. INFERENZA COLORE PEZZO (Euristica/Istogramma)
                // Un classificatore separato determina se il pezzo è bianco o nero
                // basandosi sulla distribuzione dei pixel nella cella ritagliata.
                val colorLabel = PieceColorClassifier.classifyPieceColor(resizedMat, pieceLabel)

                // 9. Costruzione dell'etichetta finale
                val finalLabel = if (colorLabel == "empty") {
                    "empty" // Se il colore dice vuoto, ignoriamo la forma
                } else {
                    "${colorLabel}_$pieceLabel"
                }

                // Log di debug e salvataggio in matrice
                Log.d("Classifier_Debug", "Square [$row,$col] = $label")
                resultMatrix[row][col] = finalLabel

                // Rilascio della sottomatrice per evitare memory leak nel loop
                squareMat.release()
            }
        }
    } catch (e: Exception) {
        Log.e("PipelineError", "Error during analysis", e)
    } finally {
        // Rilascio obbligatorio di tutte le risorse native OpenCV
        fullRgbaMat.release()
        fullRgbMat.release()
        ycrcbMat.release()
        resizedMat.release()
        ycrcbChannels.forEach { it.release() }
    }

    return resultMatrix
}
```

4.1.3.B Pipeline B – YOLO su immagine rettificata

La pipeline B, basata su YOLO, adotta un approccio più diretto per la rilevazione dei pezzi sulla scacchiera. L'immagine warping della scacchiera viene ridimensionata a 640×640 pixel, compatibile con l'input del modello YOLO, e passata attraverso la rete neurale per ottenere le predizioni dei pezzi presenti.

Il modello restituisce bounding box e classi per ogni pezzo, includendo anche il colore. Le coordinate dei rilevamenti vengono raccolte in un'unica lista, che rappresenta la posizione e l'identità di tutti i pezzi sulla scacchiera.

Grazie a questo approccio, tutte le informazioni sui pezzi (tipo e colore) vengono ottenute in un solo passaggio, senza necessità di ulteriori elaborazioni per ciascuna casella, rendendo la pipeline estremamente rapida e adatta a scenari real-time.

Di seguito il metodo principale che gestisce la pipeline di Yolo:

```
/**
 * Esegue l'inferenza YOLO utilizzando la tecnica del "Tiling" (o Slicing).
 * N:B: Per mantenere alta la risoluzione e migliorare le prestazioni del modello ho provato
 * a passare a Yolo non l'immagine intera, bensì delle griglie più piccole della scacchiera.
 */
fun runTiledYoloInference(
    warpedBmp: Bitmap,
    net: Net,
    classNames: List<String>,
    tileGridSize: Int = 2, // Griglia 2x2 (4 tiles totali)
    inputSize: Int = 640   // Dimensione input standard per YOLO
): List<YoloDetection> {

    val imageWidth = warpedBmp.width
    val imageHeight = warpedBmp.height

    // Calcolo delle dimensioni di ogni singola piastrella (Tile)
    val tileWidth = imageWidth / tileGridSize
    val tileHeight = imageHeight / tileGridSize

    val allDetections = mutableListOf<YoloDetection>()

    // --- 1. ITERAZIONE SULLA GRIGLIA (Sliding Window) ---
    for (rowTile in 0 ≤ until < tileGridSize) {
        for (colTile in 0 ≤ until < tileGridSize) {

            // Calcolo dell'offset (spostamento) per il ritaglio corrente
            val xOffset = colTile * tileWidth
            val yOffset = rowTile * tileHeight
```

```
            // Creazione della Tile: Ritaglio della porzione di immagine originale
            val tileBmp = Bitmap.createBitmap(warpedBmp, xOffset, yOffset, tileWidth, tileHeight)

            // Scaling: La tile viene ridimensionata a 640x640.
            // *NOTA*: Questo passaggio ingrandisce virtualmente i pezzi rispetto all'immagine intera,
            // rendendoli più facili da rilevare per la CNN.
            val resizedTile = tileBmp.scale(inputSize, inputSize)

            // --- 2. INFERENZA E NMS ---
            // Esegue la forward pass della rete neurale
            val outs = runYoloInference(resizedTile, net)

            // Post-Processing: Decodifica l'output grezzo di YOLO.
            // È QUI che viene applicata la Non-Maximum Suppression (NMS).
            // L'NMS filtra le centinaia di box sovrapposte mantenendo solo quella con
            // la "confidence" maggiore per ogni oggetto rilevato nella tile.
            val detections = postProcessDetections(outs, classNames, inputSize, inputSize)

            // --- 3. RIMAPPATURA COORDINATE (Locale -> Globale) ---
            for (det in detections) {

                // Le coordinate 'det.boxX' sono relative alla tile (0.0 - 1.0 rispetto alla tile).
                // Dobbiamo convertirle in coordinate assolute pixel rispetto all'immagine intera.
                val absX = det.boxX * tileWidth + xOffset
                val absY = det.boxY * tileHeight + yOffset

                // Normalizzazione finale rispetto alle dimensioni totali dell'immagine originale
                val normalizedX = absX / imageWidth
                val normalizedY = absY / imageHeight
                allDetections.add(det.copy(boxX = normalizedX, boxY = normalizedY))
            }

            // --- 4. GESTIONE MEMORIA ---
            // Fondamentale in Android/OpenCV per evitare memory leaks durante il loop
            outs.forEach { it.release() }
            resizedTile.recycle()
            tileBmp.recycle()
        }
    }

    // Ritorna la lista unificata di tutte le rilevazioni provenienti dalle varie tiles
    return allDetections
}
```

4.1.4 Generazione della Stringa FEN ed interfacciamento con Lichess

Nella fase finale, le informazioni ottenute dalle due pipeline di object detection vengono convertite in una rappresentazione strutturata della scacchiera. Ogni casella della griglia 8×8 viene analizzata e, se contiene un pezzo, viene creato un oggetto Piece corrispondente, con tipo, colore e posizione.

Questa operazione viene eseguita in modo leggermente diverso a seconda della pipeline utilizzata: per la pipeline TFLite si decodificano i label (classi disponibili che il modello è stato allenato a riconoscere, ossia pedone, torre, cavallo alfiere, regina e re) generati dal classificatore per ogni casella, mentre per la pipeline YOLO i label contengono direttamente informazioni su tipo e colore.

Una volta costruita la lista completa dei pezzi presenti, la posizione viene tradotta in una stringa FEN. Per una corretta ricostruzione della posizione scacchistica tuttavia non è sufficiente la sola disposizione dei pezzi sulla scacchiera; è necessario tracciare alcuni parametri dinamici che influenzano la legalità delle mosse successive:

- Cattura *En Passant*: Indica la casella "target" (es. *e3*) dietro un pedone che ha appena mosso di due case, rendendosi vulnerabile alla presa speciale da parte di un pedone avversario. In assenza di tale condizione, il campo viene impostato sul simbolo standard "-".
- Regola delle 50 mosse e Semi-mosse (*Half-move Clock*): Questo contatore registra il numero di "mezze mosse" (mosse singole di un giocatore) effettuate dall'ultima cattura o dall'ultima spinta di un pedone. Secondo il regolamento FIDE, se questo contatore raggiunge le 100 semi-mosse (50 mosse per lato), un giocatore può reclamare la patta. Il reset automatico del contatore a ogni cattura o mossa di pedone è fondamentale per la corretta valutazione della posizione da parte del motore di analisi.

La stringa FEN (Forsyth-Edwards Notation)[21] è uno standard testuale che permette di rappresentare in maniera completa e compatta lo stato di una scacchiera in un'unica riga di testo ASCII. Una stringa FEN è composta da sei campi, separati da spazi, e di solito racchiusi tra parentesi quadre:

1. Posizione dei pezzi: si descrive ogni traversa (*ranks*) della scacchiera a partire dall'ottava fino alla prima. All'interno di ogni traversa, si indicano i contenuti delle caselle dalla colonna "a" alla colonna "h". I pezzi bianchi sono rappresentati dalle iniziali inglesi maiuscole ("K" per il re, "Q" per la regina, "R" per la torre, "B" per l'alfiere, "N" per il cavallo, "P" per il pedone), mentre i pezzi neri con le stesse lettere in minuscolo. Le caselle vuote sono rappresentate da numeri da 1 a 8, corrispondenti al numero di caselle consecutive libere, e le traverse sono separate dal simbolo "/".
2. Giocatore di turno: indica chi deve muovere. "w" significa che è il bianco a muovere, "b" indica il nero.
3. Possibilità di arrocco: se nessuno dei due giocatori può arroccare si usa "-". Altrimenti, si specifica tramite lettere: "K" per arrocco corto bianco, "Q" per arrocco lungo bianco, "k" per arrocco corto nero, "q" per arrocco lungo nero.
4. Cattura en passant: se non è possibile catturare en passant si indica "-". Se un pedone ha appena mosso due caselle e può essere catturato en passant, la casella target viene indicata.
5. Numero di semimosse: conteggia le mezze mosse dall'ultima mossa di pedone o dall'ultima cattura, utile per la regola delle cinquanta mosse.
6. Numero totale delle mosse: indica quante mosse complete sono state effettuate nella partita, iniziando da 1 e incrementando di uno dopo ogni mossa del nero.

Ad esempio, considerando una posizione in cui non sono rilevanti arrocco, en passant e conteggio delle mosse, la FEN può ridursi ai soli campi pezzi e turno di gioco. Una possibile rappresentazione sarà: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1 per la posizione iniziale della scacchiera.

Questa notazione compatta permette di ricostruire esattamente la posizione di ogni pezzo, mantenendo informazioni sufficienti per riprodurre la partita in qualsiasi motore o editor di scacchi compatibile.

La FEN generata viene poi utilizzata per aprire direttamente l'editor di Lichess

val url = "https://lichess.org/editor/$encodedFen"

consentendo di visualizzare e verificare immediatamente la posizione rilevata dall'applicazione. Aprendo l'editor inoltre è quindi possibile modificare la configurazione della scacchiera in caso il modello di riconoscimento utilizzato sbagliasse la classificazione di alcune caselle.

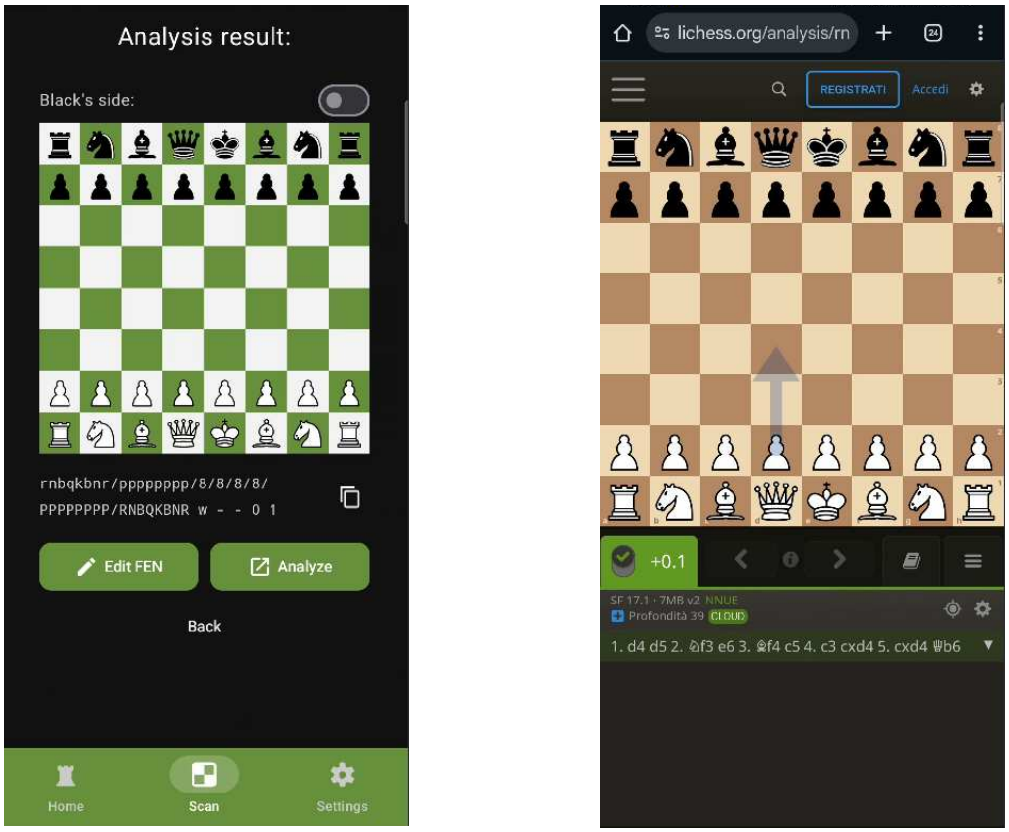


Figure 70 71: Risultato analisi della scacchiera e apertura FEN sul motore Lichess

4.2 Metodologia di Test

Per valutare le prestazioni delle due pipeline implementate per il riconoscimento dei pezzi sulla scacchiera, sono stati condotti test sistematici e strutturati.

La Pipeline A, basata su un modello TFLite, è stata analizzata mediante matrici di confusione per il riconoscimento dei tipi di pezzo e test di accuratezza per la classificazione del colore dei pezzi stessi.

La Pipeline B, basata su un modello YOLO pre-addestrato, è stata valutata considerando l'accuratezza della classificazione per ciascun quadrato della scacchiera, sfruttando il fatto che il modello contiene già un layer dedicato per ciascun tipo di pezzo e per il colore.

Per organizzare i test, sono state definite alcune domande chiave a cui ogni esperimento doveva rispondere, e sono state create directory di immagini di test contenenti circa un centinaio di immagini ciascuna. In ciascun test è stata isolata una singola variabile per poter misurare in modo accurato l'influenza dei diversi fattori sulle prestazioni dei modelli.

La directory DS1 è stata utilizzata per valutare le prestazioni del modello TFLite in condizioni ottimali, con immagini acquisite frontalmente dall'applicazione. Tutte le immagini contengono posizioni variabili sulla scacchiera, permettendo di analizzare l'accuratezza del sistema nel riconoscimento dei pezzi e dei colori in condizioni controllate.

La directory DS2 è stata utilizzata per verificare la robustezza della pipeline rispetto all'angolazione della fotocamera: le immagini sono state acquisite in maniera inclinata rispetto alla scacchiera, permettendo di valutare l'efficacia della rettificazione prospettica nell'adattare il modello a condizioni non frontali.

La directory DS3, infine, è stata creata per testare l'impatto delle condizioni di illuminazione: sono state raccolte immagini in ambienti con luce variabile, tra stanze buie, stanze illuminate artificialmente e stanze con luce naturale intensa. L'obiettivo era verificare la capacità del classificatore di colore e dei modelli TFLite e YOLO di mantenere prestazioni stabili in presenza di variazioni di luce.

Si segnala che le immagini DS2 e DS3 sono state acquisite sempre sullo stesso dispositivo e nella stessa stanza, per minimizzare variabili non controllate. Non sono stati effettuati test sul modello TFLite con immagini acquisite tramite screenshot, in quanto i quadrati della scacchiera utilizzati per l'addestramento erano esattamente quelli presenti in queste immagini.

4.2.1 Metriche

Di seguito un elenco delle metriche che sono state utilizzate per valutare i risultati dei test:

- *Numero di immagini processate*
  Indica il numero totale di immagini analizzate per ciascun set di test. Permette di verificare la dimensione del campione e la significatività dei risultati.

- *Tempo medio di inferenza (ms)*
  Misura il tempo medio necessario alla pipeline per analizzare un'immagine e produrre l'output. È utile per valutare l'efficienza e la fattibilità del modello in applicazioni in tempo reale.

- *Exact Match Rate (EMR)*
  Percentuale di scacchiere in cui tutti i pezzi sono stati identificati correttamente senza alcun errore. Fornisce una misura stringente della performance globale della pipeline.

- *Accuratezza media per quadrato (Square Accuracy)*
  Percentuale di caselle della scacchiera correttamente classificate indipendentemente dagli altri pezzi. Consente di valutare l'efficacia del modello su singole posizioni della scacchiera.

- *Matrice di confusione (Confusion Matrix)*
  Rappresenta il numero di volte che ogni classe reale (pezzo o casella vuota) è stata classificata come ciascuna delle possibili classi predette. Aiuta a individuare pattern di errore specifici tra pezzi simili (ad esempio cavallo versus alfiere).

- *Accuratezza del colore dei pezzi (Color Classification Accuracy)*
  Percentuale di pezzi correttamente identificati come bianchi o neri. Valuta la robustezza del modello nella distinzione del colore, essenziale per la ricostruzione dello stato della scacchiera.

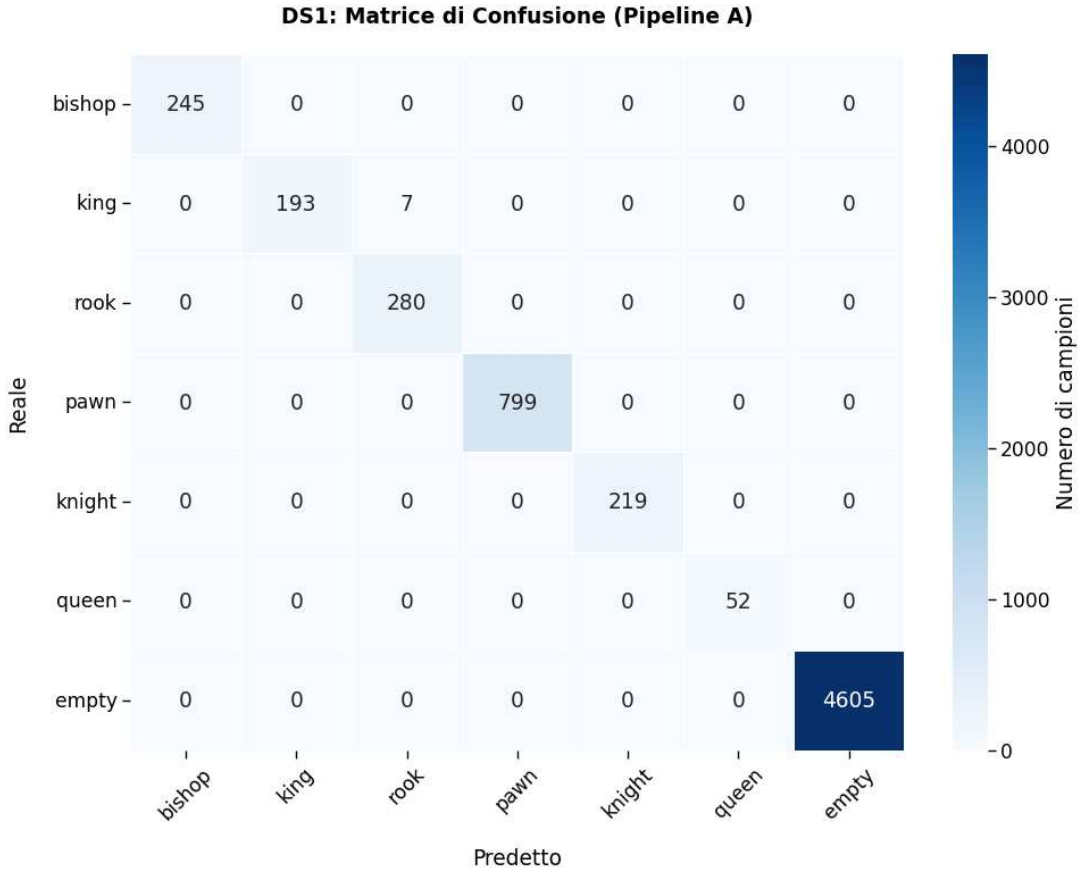
4.2.2 Risultati e Discussione

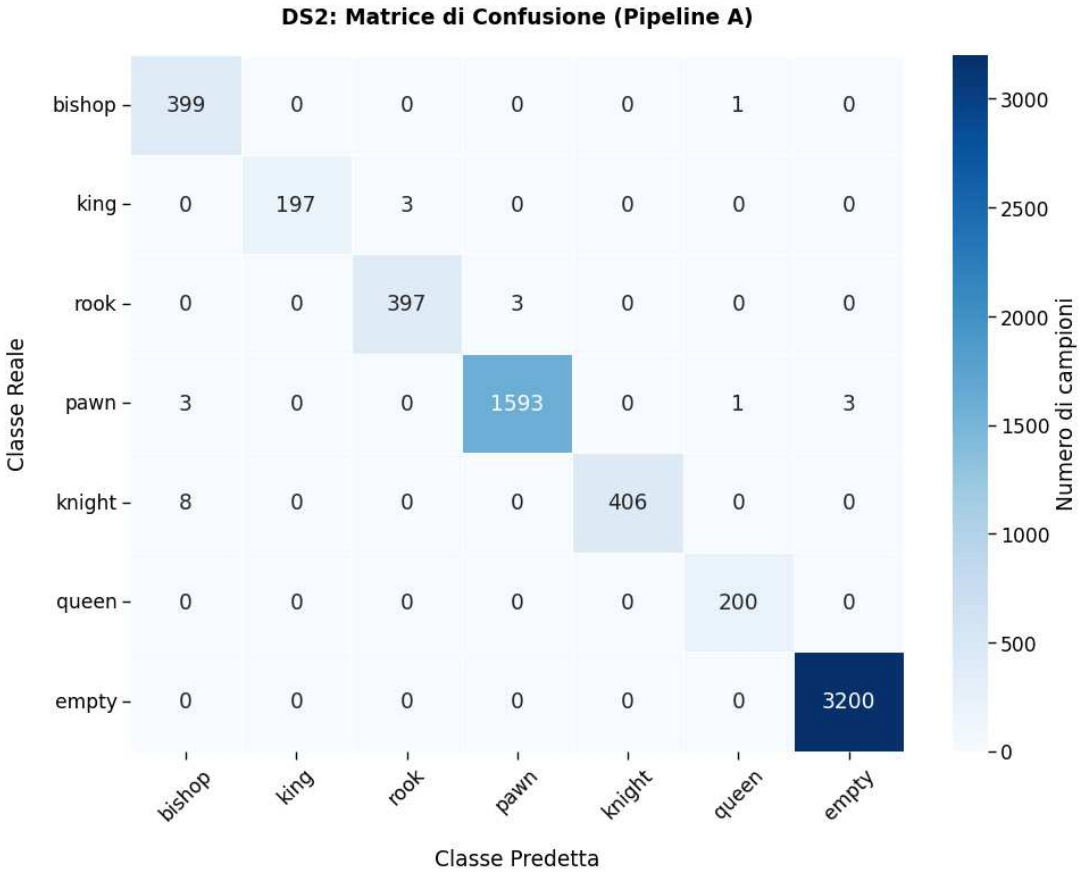
Riepilogo dei risultati della pipeline A:

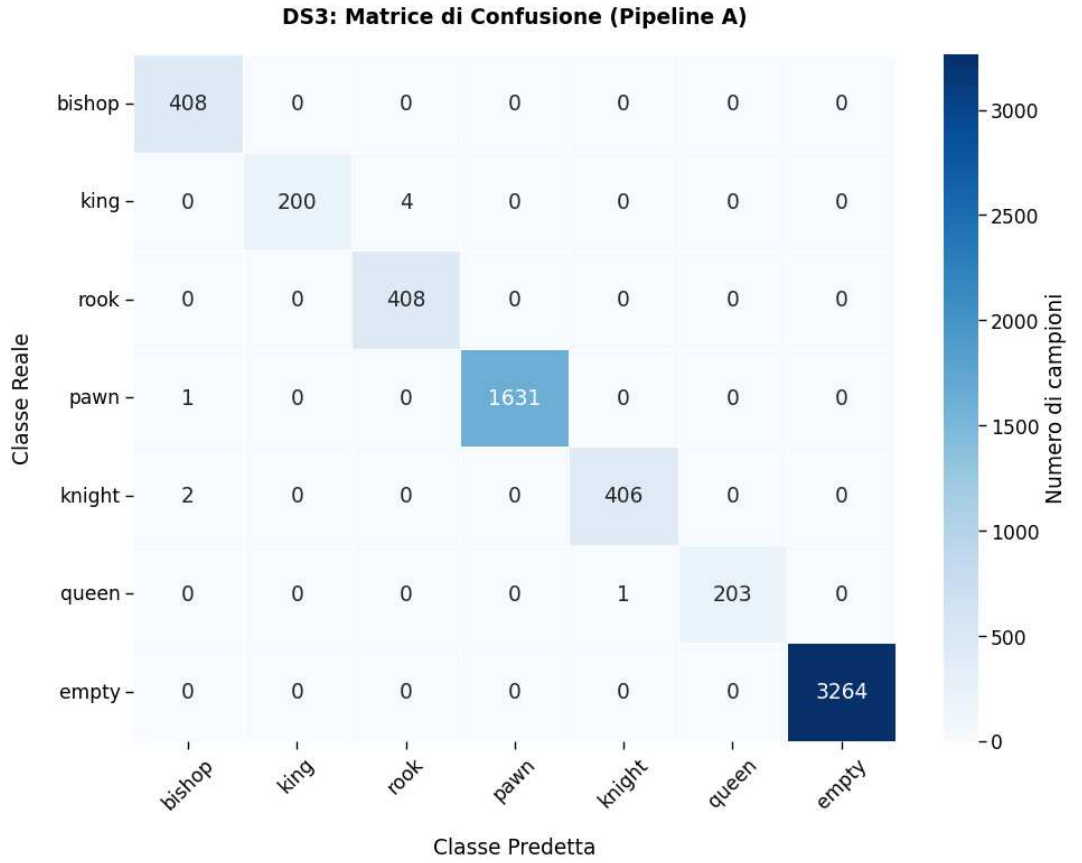
Set di Dati	Scacchiere	Accuratezza Media	EMR (%)
DS1	100	99.89%	94.00%
DS2	100	99.66%	83.00%
DS3	102	99.88%	92.16%

Risultati dell'accuratezza dell'algoritmo per il riconoscimento del colore dei pezzi:

Set	Scenario di Test	Pezzi Totali	Accuratezza Neri	Accuratezza Bianchi	Accuratezza Globale
DS1	Generale (Posizioni)	1795	99.36%	100.0%	99.72%
DS2	Angolazione Variabile	3200	97.0%	99.38%	98.19%
DS3	Luce Variabile	3264	96.81%	99.39%	98.1%




DS2: Matrice di Confusione (Pipeline A)
bishop 399 0 0 0 0 1 0
king 0 197 3 0 0 0 0
rook 0 0 397 3 0 0 0
pawn 3 0 0 1593 0 1 3
knight 8 0 0 0 406 0 0
queen 0 0 0 0 0 200 0
empty 0 0 0 0 0 0 3200
bishop king rook pawn knight queen empty
Classe Reale
Classe Predetta
Numero di campioni
3000 2500 2000 1500 1000 500 0


DS3: Matrice di Confusione (Pipeline A)
bishop 408 0 0 0 0 0 0
king 0 200 4 0 0 0 0
rook 0 0 408 0 0 0 0
pawn 1 0 0 1631 0 0 0
knight 2 0 0 0 406 0 0
queen 0 0 0 0 1 203 0
empty 0 0 0 0 0 0 3264
bishop king rook pawn knight queen empty
Classe Reale
Classe Predetta
Numero di campioni
3000 2500 2000 1500 1000 500 0

Dall’analisi delle tabelle e delle matrici di confusione, si evince che la Pipeline A ha ottenuto performance eccellenti. L’accuratezza media per singola directory si è mantenuta costantemente sopra il 99,5%; parallelamente, l’algoritmo per il riconoscimento del colore dei pezzi ha mostrato un’elevata affidabilità, contribuendo a un'accuratezza globale superiore al 98%.

Il modello TFLite presenta inoltre un exact match rate molto alto per tutte e tre le directories dove sono stati effettuati i test (83% per la directory che ha riscontrato l’EMR più basso).

Dalle matrici di confusione emerge che le errate classificazioni riguardano principalmente lo scambio tra Re e Torri. Tale errore risulta morfologicamente comprensibile, data la somiglianza delle silhouette e delle proporzioni tra i due pezzi.

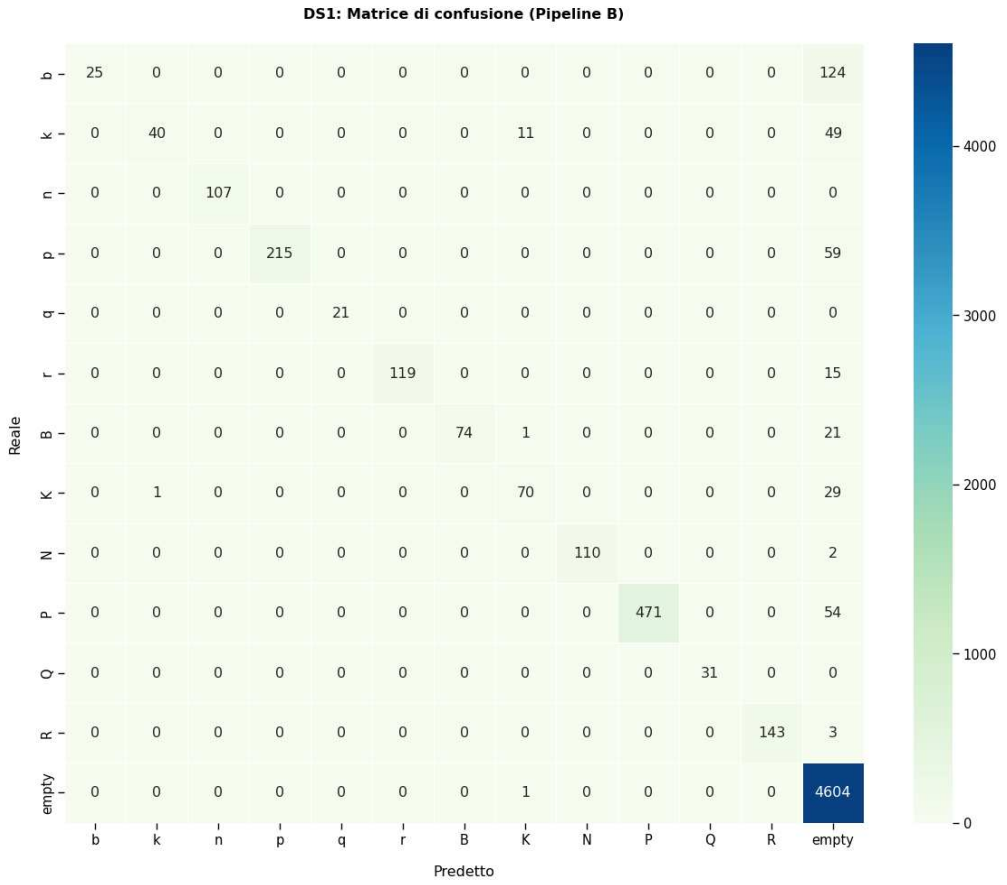
Per quanto riguarda la classificazione del colore, l'accuratezza è risultata più sensibile alle variazioni di angolazione e intensità luminosa. In particolare, l'accuratezza sui pezzi neri ha subito una flessione, scendendo al 97% (DS2) e al 96,81% (DS3). Questo risultato è riconducibile a due fattori principali:

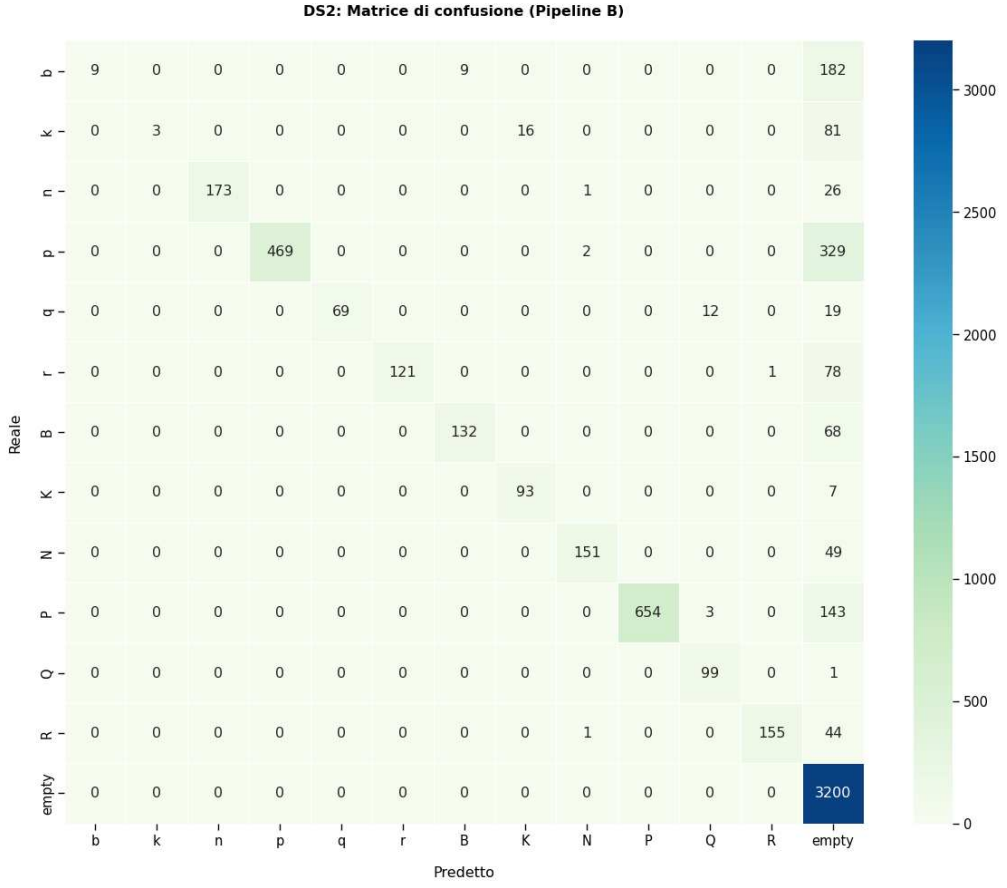
- In base a come viene scattata la foto, nelle acquisizioni effettuate da schermi digitali, l'interferenza tra la griglia dei pixel del monitor e il sensore della fotocamera può generare pattern di Moiré. Questi si manifestano come striature biancastre ondulate che alterano i valori cromatici dell'immagine. Poiché l'algoritmo analizza un patch centrale di ogni casella, la presenza di troppi pixel "chiari" dovuti al Moiré può portare il sistema a classificare erroneamente un pezzo nero come bianco.
- In condizioni di luce variabile, la superficie dello schermo o dei pezzi può presentare riflessi diretti che creano aree di saturazione bianca (clip del segnale). Se il patch analizzato ricade in una zona affetta da riflesso, la media dei pixel risulterà sbilanciata verso le alte luci, inducendo l'algoritmo in errore.

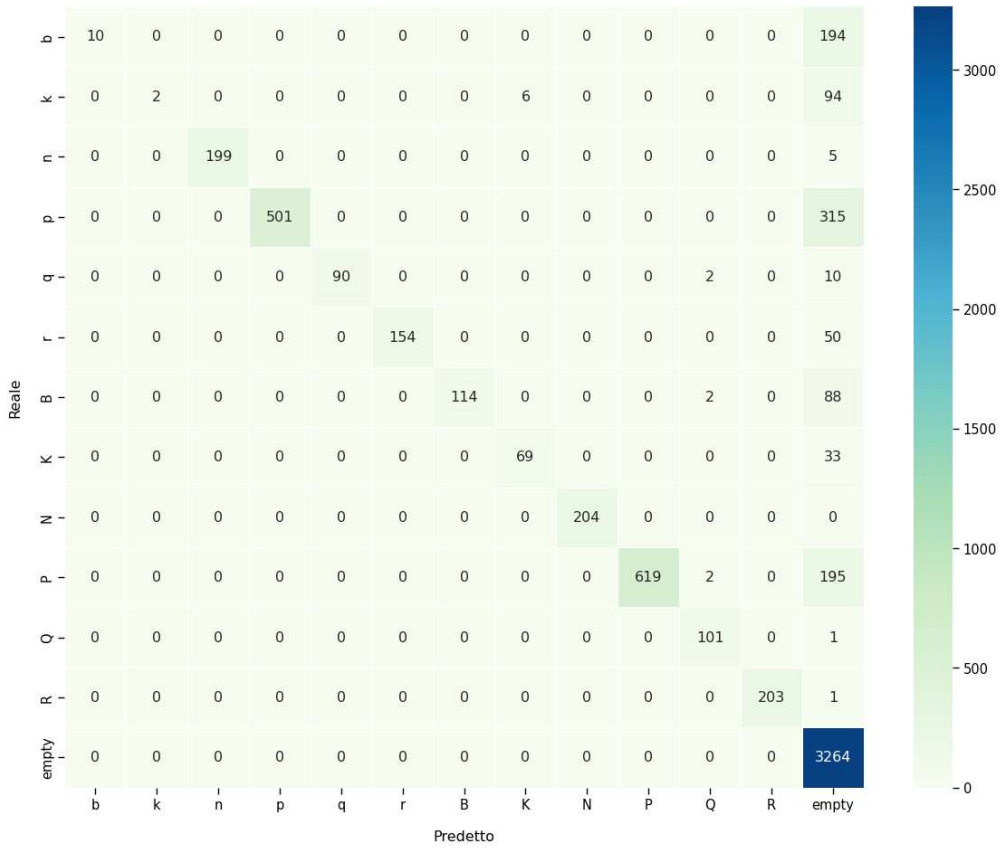
Queste osservazioni spiegano la discrepanza tra l'accuratezza dei pezzi bianchi (prossima al 100%) e quella dei pezzi neri, i quali risultano più vulnerabili ai disturbi luminosi. Una possibile soluzione per ottimizzare il sistema potrebbe risiedere nella calibrazione dinamica della soglia del valore medio (threshold) utilizzata per la discriminazione binaria del colore, o nell'implementazione di un algoritmo di pre-processing per la rimozione dei riflessi.

Report per quanto riguarda la pipeline B:

Metrica	DS1 (Foto)	DS2 (Angolo)	DS3 (Luce)
Immagini Processate	100	100	102
Tempo Medio Inferenza	558 ms	562 ms	559 ms
EMR	10.00%	0.00%	0.00%
Accuratezza Media /Quadrato	94.22%	83.25%	84.71%




DS2: Matrice di confusione (Pipeline B)
Reale
Predetto
b k n p q r B K N P Q R empty
b 9 0 0 0 0 0 9 0 0 0 0 0 182
k 0 3 0 0 0 0 0 16 0 0 0 0 81
n 0 0 173 0 0 0 0 0 1 0 0 0 26
p 0 0 0 469 0 0 0 0 2 0 0 0 329
q 0 0 0 0 69 0 0 0 0 0 12 0 19
r 0 0 0 0 0 121 0 0 0 0 0 1 78
B 0 0 0 0 0 0 132 0 0 0 0 0 68
K 0 0 0 0 0 0 0 93 0 0 0 0 7
N 0 0 0 0 0 0 0 0 151 0 0 0 49
P 0 0 0 0 0 0 0 0 0 654 3 0 143
Q 0 0 0 0 0 0 0 0 0 0 99 0 1
R 0 0 0 0 0 0 0 0 1 0 0 155 44
empty 0 0 0 0 0 0 0 0 0 0 0 0 3200
3000 2500 2000 1500 1000 500 0


DS3: Matrice di confusione (Pipeline B)
Reale
Predetto
b k n p q r B K N P Q R empty
b 10 0 0 0 0 0 0 0 0 0 0 0 194
k 0 2 0 0 0 0 0 6 0 0 0 0 94
n 0 0 199 0 0 0 0 0 0 0 0 0 5
p 0 0 0 501 0 0 0 0 0 0 0 0 315
q 0 0 0 0 90 0 0 0 0 0 2 0 10
r 0 0 0 0 0 154 0 0 0 0 0 0 50
B 0 0 0 0 0 0 114 0 0 0 2 0 88
K 0 0 0 0 0 0 0 69 0 0 0 0 33
N 0 0 0 0 0 0 0 0 204 0 0 0 0
P 0 0 0 0 0 0 0 0 0 619 2 0 195
Q 0 0 0 0 0 0 0 0 0 0 101 0 1
R 0 0 0 0 0 0 0 0 0 0 0 203 1
empty 0 0 0 0 0 0 0 0 0 0 0 0 3264
3000 2500 2000 1500 1000 500 0

Per quanto riguarda la Pipeline B, i risultati complessivi non hanno raggiunto i livelli di eccellenza della precedente sperimentazione. Sebbene il modello YOLOv3 Tiny presenti tempi di inferenza ridotti rispetto alla Pipeline A (la quale risulta intrinsecamente più onerosa a causa della sua struttura multi-stadio) l'incremento di velocità non è tale da giustificare il sensibile degrado della precisione.

È doveroso premettere che, in un'ottica di ottimizzazione dei tempi e per mantenere il focus della tesi sullo studio sistematico delle tecniche fondamentali di Computer Vision, si è scelto di integrare un modello YOLOv3 Tiny già addestrato, reperito da repository pubblico. Di conseguenza, le specifiche esatte del dataset di training e le relative iper-parametri non sono state oggetto di un controllo rigoroso, rappresentando una variabile esterna nel confronto tra le pipeline. [29]

Sebbene l'accuratezza media per singola casa possa apparire soddisfacente (94,22% per DS1, ~84% per DS2 e DS3), queste metriche sono ingannevoli se traslate sulla scacchiera completa. La natura moltiplicativa dell'errore ha infatti comportato un crollo drastico dell'Exact Match Rate (EMR), che è passato dal 10% del primo dataset allo 0% per i restanti. Questo dimostra come, in un compito che richiede un riconoscimento meticoloso di 64 elementi simultanei, anche un'accuratezza del 94% per cella sia insufficiente a garantire la correttezza della stringa FEN finale.

L'analisi delle matrici di confusione evidenzia un errore sistematico: la frequente classificazione di pezzi reali come "spazi vuoti". Si ritiene che la causa principale risieda nella bassa risoluzione di input richiesta da YOLO-tiny. Quando l'intera scacchiera viene compressa nelle dimensioni di input del modello, i dettagli morfologici dei singoli pezzi diventano troppo sfocati per una distinzione precisa.

La validità di questa analisi è supportata dal confronto tra le diverse modalità di input: l'adozione di una strategia a tile 2x2 ha prodotto risultati significativamente superiori rispetto all'elaborazione della scacchiera completa. Mentre nel secondo caso l'eccessiva compressione dell'immagine induceva il modello a confondere sistematicamente i pezzi con le caselle vuote, l'incremento della risoluzione relativa ottenuto tramite il tiling ha restituito matrici di confusione decisamente più bilanciate, confermando come la risoluzione sia il principale collo di bottiglia della Pipeline B.

5 Conclusioni e idee future

Lavorare a questo progetto si è rivelato un'esperienza estremamente stimolante, permettendomi di approcciarmi all'ambito della *Computer Vision* per me di grande interesse. Affrontare un lavoro di queste dimensioni ha rappresentato una palestra fondamentale per acquisire esperienza pratica nella gestione di sistemi complessi, dalla fase di ideazione fino alla scrittura del codice.

Il percorso non è stato privo di ostacoli. Alcuni errori di valutazione e imprevisti tecnici incontrati durante lo sviluppo hanno causato rallentamenti, ma si sono trasformati in importanti opportunità di apprendimento.

Forse la lezione che ho compreso è che la progettazione sistematica è la fase più critica della programmazione di sistemi complessi o lunghe pipeline: la capacità di prevedere e prevenire possibili problematiche a ogni step della pipeline è ciò che distingue uno sviluppo lineare da uno frammentato.

Guardando al futuro, vi sono numerosi spunti interessanti per estendere questo progetto. Tra questi, la classificazione di scacchiere tridimensionali rappresenta una direzione promettente, con l'obiettivo di riconoscere non solo la posizione e il tipo dei pezzi, ma anche eventuali caratteristiche spaziali aggiuntive (come inclinazioni o ostacoli sul piano di gioco), aprendo la strada a sistemi ancora più robusti e sofisticati per l'analisi automatica delle partite.

Bibliografia

Testi di Riferimento e Manuali

[1] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. Springer, 2022.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. Pearson, 2018.

[3] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

Articoli Scientifici

[4] J. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, 1986.

[5] I. Sobel and G. Feldman, "A 3x3 Isotropic Gradient Operator for Image Processing," *Stanford AI Project*, 1968.

[6] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Commun. ACM*, 1972.

[7] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, 2017.

[8] J. Terven *et al.*, "A Comprehensive Review of YOLO Architectures," *MLKE*, 2023.

[9] P. Purwono *et al.*, "Understanding of Convolutional Neural Network (CNN): A Review," *IJRCS*, 2022.

[10] S. Gupta and A. N. Malik, "A review on homography and its applications," *Electronics*, 2023.

Documentazione Tecnica, Dataset e Software

[11] Google Developers, "CameraX Overview," *Android Documentation*. [Online] https://developer.android.com/media/camera/camerax?hl=it

[12] MathWorks, "Contrast-Limited Adaptive Histogram Equalization (CLAHE)," *MATLAB Docs*. [Online]. https://it.mathworks.com/help/visionhdl/ug/contrast-adaptive-histogram-equalization.html#d126e9140

[13] A. Lavarini, "2D Chessboard and Chess Pieces Dataset," *Roboflow Universe*, 2024. [Online]. https://app.roboflow.com/angelo-fo38b/2d-chessboard-and-chess-pieces-udecm/browse?queryText=split%3Atrain&pageSize=50&startingIndex=0&browseQuery=true

[14] Google, "Teachable Machine," *Google AI experiments*. [Online]. https://teachablemachine.withgoogle.com/

[15] V. Parsaniya, "Yolo-Segmentation-Chess," *GitHub Repository*. [Online]. Disponibile: https://github.com/Vatsalparsaniya/Yolo-Segmentation-Chess

Risorse Web, Lezioni e Scacchi

[16] IBM, "What is Computer Vision?" e "Image Segmentation," *IBM Topics*. [Online]. https://www.ibm.com/it-it/think/topics/computer-vision https://www.ibm.com/think/topics/image-segmentation

[17] S. K. Nayar, "First Principles of Computer Vision," *Columbia University (YouTube)*. [Online]. https://www.youtube.com/watch?v=XRBc_xkZREg

[18] GeeksforGeeks, "Feature Extraction and Image Processing Algorithms," 2023. [Online]. https://www.tutorialspoint.com/computer-vision/computer-vision-feature-detection-extraction.htm https://www.geeksforgeeks.org/computer-vision/what-are-the-main-steps-in-a-typical-computer-vision-pipeline/ https://www.geeksforgeeks.org/computer-vision/image-processing-algorithms-in-computer-vision/

[19] S. Giassa, "Nearest Neighbour Interpolation," *Signal Processing and Computer Vision*, 2009. [Online]. https://www.giassa.net/?page_id=207

[20] University of Maryland, "Homography estimation and decomposition," *CMSC426 Course Notes*, 2019. [Online]. https://www.cs.umd.edu/class/fall2019/cmsc426-0201/files/16_Homography-estimation-and-decomposition.pdf

[21] Chess.com, "Chess popularity and FEN (Forsyth-Edwards Notation) guide," 2024. [Online]. https://www.digitec.ch/it/page/gli-scacchi-sono-pi-popolari-che-mai-perche-28274#:~:text=Gli%20scacchi%20sono%20più%20popolari%20che%20mai.,nella%20crescente%20popolarità%20degli%20scacchi

[22] L. Rozman (GothamChess) & H. Nakamura https://youtube.com/@gothamchess?si=hqf8gjcsr2nCXCQC https://youtube.com/@gmhikaru?si=Cs3VqhmFUGEhDuNl

[23] StackOverflow, "YUV_420_888 to RGB conversion for CameraX," 2019. [Online]. https://stackoverflow.com/questions/58102717/android-camerax-analyzer-image-with-format-yuv-420-888-to-opencv-mat

[24] Educative.io, "Sharpening and Smoothing images in Computer Vision," 2024. [Online]. https://www.educative.io/answers/sharpening-and-smoothing-images-in-computer-vision

[25] I. Macdonald, "Probabilistic Hough Transform," *University of Edinburgh*. [Online]. https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV1011/macdonald.pdf

[26] Wikipedia, "Voci consultate: Visione artificiale, Rumore d'immagine, Filtri (Mediano, Bilaterale), Equalizzazione dell'istogramma (AHE, CLAHE), Segmentazione (Split-Merge, K-Means), Interpolazione (Vicino più prossimo, Bilineare, Bicubica), Operatori di gradiente (Prewitt, Sobel), Canny Edge Detection, CNN, Transfer Learning, Formato YUV e Notazione FEN." [Online]. * Generali e Teoria: https://it.wikipedia.org/wiki/Visione_artificiale, https://it.wikipedia.org/wiki/Apprendimento_per_trasferimento

- Elaborazione Immagini: https://en.wikipedia.org/wiki/Image_noise, https://en.wikipedia.org/wiki/Median_filter, https://en.wikipedia.org/wiki/Bilateral_filter, https://en.wikipedia.org/wiki/Histogram_equalization, https://en.wikipedia.org/wiki/Adaptive_histogram_equalization
- Segmentazione e Interpolazione: https://en.wikipedia.org/wiki/Image_segmentation, https://en.wikipedia.org/wiki/Split_and_merge_segmentation, https://en.wikipedia.org/wiki/K-means_clustering, https://en.wikipedia.org/wiki/Bilinear_interpolation, https://en.wikipedia.org/wiki/Bicubic_interpolation
- Feature e Deep Learning: https://en.wikipedia.org/wiki/Prewitt_operator, https://it.wikipedia.org/wiki/Operatore_di_Sobel, https://en.wikipedia.org/wiki/Canny_edge_detector, https://en.wikipedia.org/wiki/Convolutional_neural_network
- Specifiche Progetto (YUV & Scacchi): https://it.wikipedia.org/wiki/YUV, https://it.wikipedia.org/wiki/Notazione_Forsyth-Edwards

[27] OpenCV Documentation, "OpenCV-Python Tutorials: Image Filtering & Thresholding,". [Online]. Disponibile: https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

[28] Paulinternet, Bicubic Interpolation [Online]. https://www.paulinternet.nl/?page=bicubic

[29] Vatsalparsaniya, Yolo Model Github project https://github.com/Vatsalparsaniya/Yolo-Segmentation-Chess/tree/main

Elenco figure:

Figura 13: Risultato dell'equalizzazione dell'istogramma su "Hawkes Bay NZ". (Fonte: P. Capper, Wikimedia Commons, Licenza CC BY 2.0)
https://commons.wikimedia.org/wiki/File:Equalized_Hawkes_Bay_NZ.jpg

Figura 14: Istogramma delle intensità dopo il processo di equalizzazione. (Fonte: Utente "Jarekt", Wikimedia Commons, Pubblico Dominio)
https://commons.wikimedia.org/wiki/File:Equalized_Histogram.svg

Figura 15: Finestre contestuali (Contextual Regions) per l'equalizzazione AHE. (Fonte: Utente "Vswitchs", Wikimedia Commons, Pubblico Dominio)
https://commons.wikimedia.org/wiki/File:AHE-neighbourhoods.svg

Figure 16 17 e 18: Le fasi del processo CLAHE: calcolo, ridistribuzione e mappatura. (Fonte: MathWorks Documentation © MathWorks, Inc.)
https://it.mathworks.com/help/visionhdl/ug/contrast-adaptive-histogram-equalization.html#d126e9140

Figura 19: Esempio di immagine pre e post global e adaptive thresholding. Presa dalla documentazione di OpenCV, licenza Apache 2.0.

Figura 20: Struttura dati Quadtree per la segmentazione. (Fonte: Utente "KoBEE515", Wikimedia Commons, Licenza CC BY-SA 4.0):
https://commons.wikimedia.org/wiki/File:Tree_structure.png

Figura 21: Partizione spaziale ricorsiva corrispondente al Quadtree. (Fonte: Utente "KoBEE515", Wikimedia Commons, Licenza CC BY-SA 4.0):
https://commons.wikimedia.org/wiki/File:Partition.jpg

Figure 22-24**:** Esempio di algoritmo Split and Merge: immagine originale, blocchi segmentati e risultato finale. (Fonte: Utente "KoBEE515", Wikimedia Commons, Licenza CC BY-SA 4.0)
(22:https://en.wikipedia.org/wiki/Split_and_merge_segmentation#/media/File:Original_grayscale.png,
23:https://en.wikipedia.org/wiki/Split_and_merge_segmentation#/media/File:Smblocks_seg.jpg,
24:https://en.wikipedia.org/wiki/Split_and_merge_segmentation#/media/File:Segmented_project_box.png)

Figura 25: Associazione tra segmentazione immagini e partizione di grafi. (Fonte: Camilus & Govindan, 2012):
https://www.researchgate.net/figure/Association-between-image-segmentation-and-graph-partitioning-a-Image-b-Graph-c_fig1_267987993

Figura 26: Segmentazione Graph-Based su immagine naturale. (Fonte: Rielaborazione da Encord su algoritmo di Felzenszwalb & Huttenlocher, 2004)
https://encord.com/blog/image-segmentation-for-computer-vision-best-practice-guide/

Figura 27-30: Passaggi dell'algoritmo K-Means (Step 1-4). (Fonte: W. Pace, Wikimedia Commons, Licenza CC BY-SA 3.0)
(27:https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_1.svg

28:https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_2.svg

29: https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_3.svg

30: https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_4.svg)

Figura 31: K-means clustering con diversi valori di K. (Fonte: C. Andaur, "Using K-Means Clustering for Image Segmentation", Medium 2021) https://cierra-andaur.medium.com/using-k-means-clustering-for-image-segmentation-fe86c3b39bf4

Figura 32: Esempio di funzione costante a tratti (Piecewise constant). (Fonte: Berland, Wikimedia Commons, Pubblico Dominio) https://commons.wikimedia.org/wiki/File:Piecewise_constant.svg

Figura 33: Interpolazione Nearest Neighbour. (Fonte: S. Giassa, "Signal Processing and Computer Vision", 2009) https://www.giassa.net/?page_id=207

Figura 34: Rappresentazione geometrica dell'interpolazione bilineare. (Fonte: Swienegel, Wikimedia Commons, Licenza CC BY-SA 4.0) https://commons.wikimedia.org/wiki/File:BilinearInterpolationV2.svg.

Figura 35: Confronto tra interpolazione 1D e 2D. (Fonte: Cmglee, Wikimedia Commons, Licenza CC BY-SA 4.0) https://commons.wikimedia.org/wiki/File:Comparison_of_1D_and_2D_interpolation.svg.

Figura 36: I pilastri dell'elaborazione immagini. (Fonte: J. Cardete, "The Art and Science of Interpolation", 2024) https://medium.com/thedeephub/the-art-and-science-of-interpolation-b12b99f2e053

Figura 37: Stima e decomposizione dell'omografia. (Fonte: University of Maryland, Corso CMSC426, 2019) https://www.cs.umd.edu/class/fall2019/cmsc426-0201/files/16_Homography-estimation-and-decomposition.pdf

Figura 38: Individuazione dei punti corrispondenti per la stima dell'omografia. (Fonte: Rielaborazione da F. Liu, Towards Data Science, 2021) https://towardsdatascience.com/understanding-transformations-in-computer-vision-b001f49a9e61/

Figura 39: Immagine originale "Bikesgray". (Fonte: D. Kennedy, Wikimedia Commons, Licenza CC BY-SA 3.0) https://commons.wikimedia.org/wiki/File:Bikesgray.jpg

Figura 40: Risultato dell'operatore di Prewitt su "Bikesgray". (Fonte: Walczak & Kennedy, Wikimedia Commons, Licenza CC BY-SA 3.0) https://commons.wikimedia.org/wiki/File:Bikesgray_prewitt.JPG

Figura 41: Immagine originale "Valve". (Fonte: Wikimedia Commons, Licenza CC BY-SA 3.0) https://commons.wikimedia.org/wiki/File:Valve_original_(1).PNG

Figura 42: Risultato dell'operatore di Sobel su "Valve". (Fonte: Wikimedia Commons, Licenza CC BY-SA 3.0) https://commons.wikimedia.org/wiki/File:Valve_sobel_(3).PNG

Figura 43: Immagine originale "Large Scaled Forest Lizard". (Fonte: Babujayan, Wikimedia Commons, Licenza CC BY 3.0)
https://commons.wikimedia.org/wiki/File:Large_Scaled_Forest_Lizard.jpg

Figura 44-48: Pipeline dell'algoritmo di Canny: Gaussian Blur, Gradiente, Non-maximum suppression, Double Threshold, Isteresi. (Fonte: Babujayan, Wikimedia Commons, Licenza CC BY 3.0)
44:https://commons.wikimedia.org/wiki/File:Canny_Walkthrough_1_Gaussian_Blur.png
45:https://commons.wikimedia.org/wiki/File:Canny_Walkthrough_2_Intensity_Gradient.png
46:https://commons.wikimedia.org/wiki/File:Canny_Walkthrough_3_Non-maximum_suppression.png
47:https://commons.wikimedia.org/wiki/File:Canny_Walkthrough_4_Double_Threshold.png
48:https://commons.wikimedia.org/wiki/File:Canny_Walkthrough_5_Hysteresis.png

Figura 49-50: Trasformata di Hough e rilevamento dei bordi. (Fonte: S. K. Nayar, First Principles of Computer Vision, Columbia University)
https://www.youtube.com/watch?v=XRBc_xkZREg.

Figure 51 52 53 54: Architettura delle Convolutional Neural Networks (CNN). (Fonte: Purwono et al., 2022)
https://www.researchgate.net/publication/367157330_Understanding_of_Convolutional_Neural_Network_CNN_A_Review#pf3

Figura 55:Differenza tra convoluzione standard e Depthwise Separable. (Fonte: Animated AI / Howard et al.) https://www.youtube.com/watch?v=vVaRhZXovbw

Figura 56: Architettura MobileNet. (Fonte: Howard et al., "MobileNets", 2017)
https://arxiv.org/pdf/1704.04861

Figure 57 58 59 60: Evoluzione delle architetture YOLO. (Fonte: Terven et al., "A Comprehensive Review of YOLO", 2023) https://www.mdpi.com/2504-4990/5/4/83

Figura 61: Schema dei rapporti di Chroma Subsampling. (Fonte: VIBBER & Mysid, Wikimedia Commons, Licenza CC BY-SA 3.0)
https://commons.wikimedia.org/wiki/File:Chroma_subsampling_ratios.svg

Figure 62 63 64 65: Esempio di scomposizione dello spazio colore YUV. (Fonte: B. Szymanski, Wikimedia Commons, Pubblico Dominio)
https://commons.wikimedia.org/wiki/File:Barn-yuv.png

Figura 66: Visualizzazione della ROI (Region of Interest) nell'interfaccia Android. Il rettangolo centrale indica l'area di elaborazione. (Fonte: Elaborazione dell'autore)

Figura 67: Feedback visivo per il rilevamento dei bordi della scacchiera. (Fonte: Elaborazione dell'autore)