SVILUPPO DI UNA APPLICAZIONE PER IL RICONOSCIMENTO E L'ANALISI DI UNA POSIZIONE DI UNA PARTITA DI SCACCHI

Sistemi Digitali M

Gabriele Corsi

Febbraio 2023

Indice

Introduzione

Capita spesso durante partite di scacchi "dal vivo" di chiedersi quale sia la migliore sequenza di mosse da seguire in determinate fasi della partita e per avere una risposta completa e precisa sarebbe necessario inserire manualmente la posizione che si vuole analizzare all'interno di motori scacchistici, software in grado di valutare una posizione e determinare chi sia in vantaggio e quale siano le mosse migliori da effettuare. Il progetto si pone l'obiettivo di implementare in un dispositivo embedded (in questo caso uno smartphone Android) un software capace, grazie ad una foto, di specificare la posizione corrente di una partita di scacchi, restituendo all'utilizzatore la prossima mossa migliore e la notazione FEN (vedi capitolo 1.3) da poter inserire comodamente in altre applicazioni per avere un'analisi più completa ed approfondita della scacchiera.

Per raggiungere tale scopo due sono le principali sfide da superare:

- essere in grado di determinare le coordinate all'interno di una immagine degli spigoli della scacchiera e delle rispettive righe e colonne (argomento trattato nel Capitolo 2)
- essere in grado di stabilire le coordinate dei pezzi sulla scacchiera e la rispettiva classe di appartenenza (es.: pedone bianco, cavallo nero, ecc.)(argomento trattato nel Capitolo 3).

Una volta in possesso di queste due informazioni è possibile creare una rappresentazione "digitale" della scacchiera. Prima di descrivere la soluzione a questi problemi, nel Capitolo 1 viene fatta una breve introduzione al gioco degli scacchi, ad alcune regole meno conosciute, alla notazione FEN e a come funzionano i motori scacchistici, in modo da capire alcune scelte fatte durante la realizzazione del progetto.
Il Capitolo 4 illustra invece come vengono implementate le soluzione dei capitoli precedenti all'interno di un'applicazione Android e ne illustra il funzionamento. I capitoli 5 e 6 concludono la relazione riassumendo i risultati ottenuti, mostrando alcune criticità riscontrare e i possibili sviluppi futuri.

Capitolo 1

Gli Scacchi

Il seguente capitolo non vuole descrivere in dettaglio tutte le regole degli scacchi ma pone l'attenzione su alcuni particolari del gioco che sono stati utilizzati per prendere alcune decisioni sullo sviluppo del software.

1.1 Glossario

Si introduce un piccolo glossario per definire il significato di alcune parole che verranno usate durante la relazione:

Analisi Lo studio di una partita o di una posizione, allo scopo di valutare la qualità delle mosse e vari altri aspetti.

Colonna Una linea verticale di case della scacchiera. Le colonne sono denominate, secondo la notazione algebrica, con le lettere a b c d e f g h.

Pezzo Una delle figure utilizzate per giocare a scacchi, ovvero un re, una regina, una torre, un alfiere, un cavallo o un pedone.

Posizione È la situazione dei pezzi in un dato momento della partita.

Traversa Una linea orizzontale di case della scacchiera. Viene denominata secondo la notazione algebrica con le cifre 1 2 3 4 5 6 7 8 partendo dal lato del Bianco.

Vantaggio Quando un giocatore ha una superiorità tattica o di pezzi durante il gioco, durante lo svolgimento di esso il vantaggio può essere ribaltato.

1.2 Arrocco/En Passant

L'arrocco è l'unica mossa che permette di muovere due pezzi contemporaneamente, ovvero il re e una delle due torri. Essa si può effettuare solamente nelle seguenti condizioni:

- Né il re né la torre coinvolta devono essere stati mossi in precedenza.
- Non ci sono pezzi tra il re e la torre coinvolta.
- Il re e la torre devono trovarsi sulla stessa traversa.
- Il re, durante il movimento dell'arrocco, non deve attraversare case in cui si troverebbe sotto scacco.

Un particolare da notare è il seguente: la prima regola impedisce di poter determinare se sia possibile arroccare o meno senza avere lo storico delle mosse della partita. Infatti entrambi i pezzi sarebbero potuti essere stati mossi in precedenza e poi aver fatto ritorno alla casa di partenza. Per questo motivo all'interno del progetto è data la facoltà all'utente di decidere se nella posizione rilevata sia regolare o meno l'esecuzione di questa mossa (vedi Sezione 4.6).

Stesso ragionamento è fatto per l'En Passant, una mossa che coinvolge due pedoni: quando un pedone, muovendosi di due case, finisce esattamente accanto (sulla stessa traversa e sulla colonna adiacente) ad un pedone avversario, solo alla mossa successiva quest'ultimo può catturarlo come se si fosse mosso di una casa soltanto. Come nel caso precedente una semplice foto non è in grado di stabilire se vi è la possibilità di compiere o meno questa mossa.

1.3 Notazione FEN

La notazione Forsyth-Edwards (FEN) è utilizzata per descrivere una particolare posizione sulla scacchiera nel corso di una partita ed è in grado di fornire tutte le informazioni necessarie per consentire la prosecuzione di una partita da una posizione data. Questa notazione prevede l'utilizzo di una singola riga di testo, contenente sei campi separati da uno spazio, con soli caratteri ASCII. I campi sono i seguenti:

1. Informazioni riguardanti il posizionamento dei pezzi: ogni traversa della scacchiera, iniziando dalla numero 8 fino alla numero 1, è descritta con i contenuti di ciascuna casa a partire dalla colonna "a" fino alla colonna "h". I pezzi del Bianco sono indicati con le seguenti lettere K = Re (King), Q = Regina (Queen), R = Torre (Rook), B = Alfiere (Bishop), N = Cavallo (Knight), P = Pedone (Pawn). I pezzi del Nero sono indicati con le stesse lettere ma in minuscolo. Se vi sono delle case vuote tra un pezzo e un altro vengono indicate con un numero dall'1 all'8. Il simbolo "/" indica la fine di una traversa e l'inizio della successiva. La figura 1.1 mostra un esempio.

Figura 1.1: Esempio costruzione primo campo della notazione FEN

2. Colore attivo: “w” indica che il Bianco deve muovere, “b” indica che è il Nero a muovere

3. Possibilità di arrocco: se nessuno dei due giocatori ha la possibilità di arroccare questo campo è riempito con il simbolo “-”, altrimenti contiene una o più lettere: “K” se il Bianco ha la facoltà di arroccare corto, “Q” se può arroccare lungo. Stessa cosa vale per il Nero ma con i simboli in carattere minuscolo.

4. Possibilità di catturare En Passant: se non è possibile effettuare alcuna cattura en passant, si indica ”-”. Se un pedone ha appena effettuato la sua prima mossa di due case e c’è la possibilità di catturarlo en passant, verrà indicata la casa ”alle spalle” del pedone stesso.

5. Numero di semimosse: indica il numero di semimosse dall’ultima cattura o dall’ultimo avanzamento di un pedone

6. Numero di mosse: il numero complessivo di mosse della partita

Un esempio completo è il seguente:

rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2

1.4 Limite sui Pezzi

Si noti inoltre come i pedoni, sia bianchi che neri, non possono trovarsi né sulla prima ne sull’ultima traversa, questo poichè essi possono solo avanzare e arrivati all’ultima traversa vengono promossi con un cavallo o un alfiere o una torre o una regina. Per quest’ultimo motivo, dal punto di vista teorico, il massimo numero

di i cavalli, alfieri, o torri è 10 (8 pedoni promossi + 2 pezzi iniziali) e 9 per la regina. Nonostante questo il riconoscimento dei pezzi all'interno del progetto limita a 2 la cardinalità di ogni classe di pezzi (escludendo i pedoni, massimo 8, e il re massimo 1) decidendo in base alla precisione rilevata dalla rete neurale (vedi Sezione 4.4). Questa scelta è stata fatta tenendo conto che tipicamente le scacchiere vengono vendute senza pezzi di riserva, per cui è improbabile che vi siano più di due pezzi uguali (esclusi pedoni e re, per la regina alcuni set vengono forniti con due regine dello stesso colore, essendo la scelta più probabile per una promozione)

1.5 Motori Scacchistici

Un motore scacchistico è un programma in grado di analizzare le posizioni durante una partita di scacchi restituendo le migliori mosse possibili secondo i propri algoritmi. La valutazione (o *eval*) di una posizione inoltre le assegna un numero in una unità di misura chiamata *centipawn*. Un pedone di vantaggio corrisponde approssimativamente a 100 centipawn. Se il numero è positivo significa che il Bianco è in vantaggio, se è negativo sarà il Nero in vantaggio. Ovviamente questo numero non considera solo la differenza di pezzi fra i due schieramenti ma anche vantaggi tattici e posizionali. Per il progetto è stato scelto di utilizzare Stockfish, motore Open-source tra i più forti al momento, disponibile per la maggior parte delle piattaforme, compreso Android. La comunicazione con questo motore avviene attraverso il protocollo Universal Chess Interface (UCI) [1]. L'implementazione del motore è spiegata più in dettaglio nella Sezione 4.5.

Capitolo 2

Rilevamento Scacchiera

Questa parte del progetto prevedeva la creazione di un algoritmo di computer vision per la rilevazione di una scacchiera all'interno di una foto. In particolare l'obiettivo è quello di individuare le coordinate dei pixel dei vertici della scacchiera e le coordinate che identificano l'inizio e la fine delle colonne e delle traverse.

Il metodo utilizzato in questo progetto, descritto in seguito, fa utilizzo della peculiarità di una scacchiera di avere 18 linee rette, divise in due gruppi di 9 linee perpendicolari tra di loro. Il riconoscimento delle linee rette viene fatto in tempo reale, quindi l'utilizzatore avrà un feedback grafico nel momento in cui inquadrerà una scacchiera ed essa sarà riconosciuta dal programma. In questo modo l'utente finale sarà aiutato nel capire quando è possibile scattare la foto per ottenere il riconoscimento dei pezzi. Se infatti la scacchiera non fosse riconosciuta sarebbe inutile effettuare l'inferenza in quanto non avremmo le informazioni per stabilire la posizione dei pezzi relativamente alla scacchiera. Tra i requisiti si ha quindi una buona efficienza per garantire un adeguato framerate. Inoltre per semplicità, in questa versione dell'applicazione, il riconoscimento viene effettuato solo se l'inquadratura è perpendicolare rispetto ad un lato della scacchiera.
Inizialmente per risolvere questo problema erano state prese in considerazione anche ulteriori due soluzioni:

- l'utilizzo dell'algoritmo di Harris, capace di individuare gli angoli all'interno di una immagine. Benchè questo soluzione funzionasse bene con la scacchiera vuota, la sua applicazione diveniva complicata con la presenza di pezzi per cui è stato preferito l'utilizzo della trasformata di Hough.

- l'utlizzo di un rete neurale di object detection che fosse in grado di riconoscere una scacchiera. Questa soluzione è stata scartata poichè avrebbe imposto che il riconoscimento avvenisse solo se l'inquadratura fosse esattamente sopra alla scacchiera (perpendicolare al piano), in caso contrario infatti il rettangolo di selezione non sarebbe in grado di approssimare in modo preciso la prospettiva. La soluzione che utilizza la trasformata di

Hough è in grado di effettuare il riconoscimento da diverse angolature rispetto al piano come mostrato in Figura 2.1.

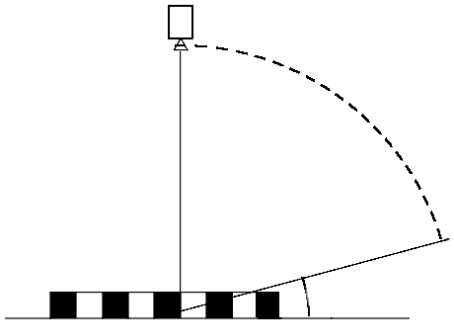

Figura 2.1: La soluzione trovata permette il riconoscimento inquadrando la scacchiera lungo tutta la linea tratteggiata

Questa parte del progetto è stata sviluppata usando python e le seguenti libreria:

- numpy 1.19.5
- opencv 4.5.1
- scikit-image 0.18.3
- scipy 1.4.1

2.1 Canny Edge Detection

La prima elaborazione che viene fatta all'immagine in input è il rilevamento dei contorni degli oggetti presenti nella foto anche detto "Edge Detection". Per ottenere questo risultato è stato impiegato l'algoritmo di Canny che utilizza la differenza di luminosità dei punti per determinare i contorni, ciò è molto efficace nello specifico caso del progetto poiché una scacchiera avendo case che si alternano con colori chiaro/scuro permette di avere nette differenze di luminosità dei pixel. In particolare è stata utilizzata l'implementazione presente nella libreria OpenCV:

```
cv.Canny(image, threshold1, threshold2)->edges
```

Questa funzione riceve in ingresso una immagine in scala di grigi, sotto forma di numpy array e due valori di threshold. Come output restituisce una immagine a singolo canale dove i pixel che rappresentano i bordi sono di colore bianco, mentre i restanti pixel sono neri.

L'algoritmo è suddiviso in più fasi:

1. Riduzione del rumore utilizzando un filtro Gaussiano 5x5
2. Viene calcolata la derivata prima della funzione di luminosità utilizzando una maschera di Sobel
3. Vengono soppressi i non-massimi: si rumuovono i pixel che non sono considerati parte di un bordo, facendo rimanere solo linee sottili
4. Attraverso l'uso di due valori soglia, MIN e MAX (con $MIN < MAX$), si stabilisce quali tra i punti trovati in precedenza è effettivamente un contorno e quali no. In particolare ogni punto il cui gradiente di intensità è maggiore della soglia MAX sono considerati bordi mentre quelli il cui gradiente di intensità è minore della soglia MIN sono scartati. I punti invece il cui gradiente di intensità è compresa tra MIN e MAX sono considerati bordi solo se sono "attaccati" a punti che sono sicuramente bordi.

E' importante quindi definire in maniera appropriata le due soglie in modo da ottenere un riconoscimento dei bordi soddisfacente, per questo motivo all'interno del progetto è stato scelto un assegnamento dinamico di queste soglie che è funzione dell'immagine in ingresso:

$$MIN = 0.26 * MEAN \tag{2.1}$$

$$MAX = 4 * MEAN \tag{2.2}$$

dove MEAN è la media dei pixel dell'immagine in scala di grigi. Queste due costanti sono state trovate mediante diversi test e grazie ad esse è possibile ottenere risultati apprezzabili in condizioni di luminosità differenti. In Figura 2.2 viene mostrato il risultato ottenuto partendo da una possibile immagine di input.

Figura 2.2: A sinistra una possibile immagine di input, a destra risultato dell'algoritmo di Canny

Successivamente,sia per migliorare le performance, sia per far capire all'utente finale il modo corretto per inquadrare la scacchiera, il programma applica il filtro mostrato in Figura 2.3 con il compito di trasformare tutti i pixel fuori dall'area bianca in pixel neri nell'immagine nella quale sono stati rilevati i bordi. Di questo filtro l'utente è avvisato mettendo l'immagine in Figura 2.3 sopra alla preview della fotocamera, in modo da spronarlo a inquadrare la scacchiera all'interno del quadrilatero bianco.

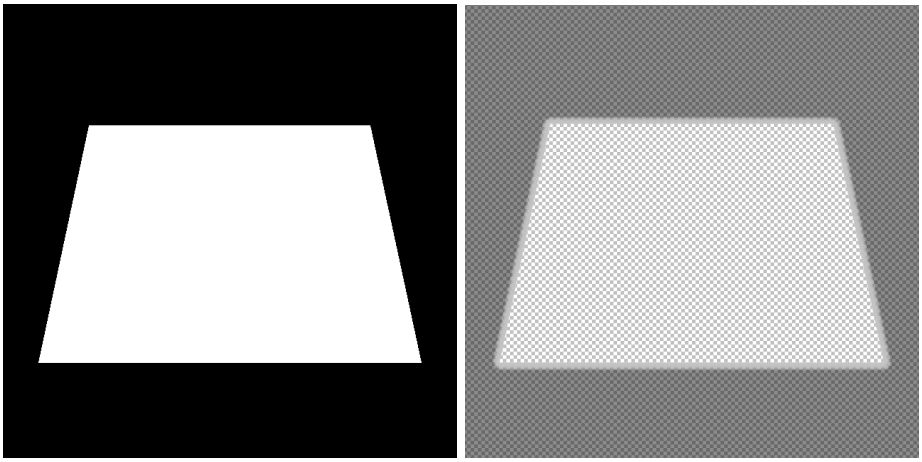

Figura 2.3: A sinistra filtro per eliminare bordi al di fuori dell'area selezionata, a destra immagine usata nella GUI

Ciò permette di avere performance più elevate poichè la trasformata di Hough del passaggio successivo verrà applicata per ogni pixel bianco dell'immagine. Il codice che filtra i pixel al di fuori dall'area di interesse è il seguente:

```
def filterCenter(img, input_image):
    mask = np.full((input_image.shape[0],input_image.shape[0]), 0).
                                        astype(np.uint8)
    points = np.array([[round(121 * input_image.shape[0] / 640),
                        round(171 * input_image.shape[0] / 640)],
                       [round(50 * input_image.shape[0] / 640),
                       round(505 * input_image.shape[0] / 640)],
                       [round(589 * input_image.shape[0] / 640),
                       round(505 * input_image.shape[0] / 640)],
                       [round(517 * input_image.shape[0] / 640),
                       round(171 * input_image.shape[0] / 640)]])
    cv2.fillPoly(mask, pts=[points], color=(255, 255, 255))
    img = np.where(mask.astype(bool), img, 0)
    return img
```

Successivamente viene effettuata una operazione di dilatazione per permettere anche alle linee più sottili di essere più rilevanti ottenendo il risultato finale mostrato in Figura 2.4.

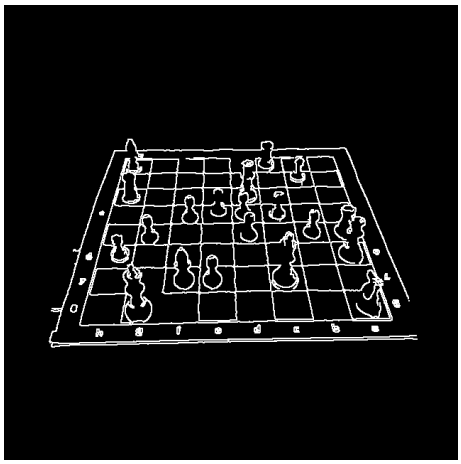

Figura 2.4: Risultato dopo l'applicazione del filtro e della dilatazione

2.2 Trasformata di Hough

Il passo successivo è stato applicare la trasformata di Hough. Questo procedimento permette di identificare linee rette all'interno di una immagine, in questo caso l'obiettivo è quello di individuare le 9 rette che separano le 8 colonne e le 9 rette che separano le 8 traverse.

L'idea alla base della trasformata di Hough è che, per ogni punto (x_i, y_i), classificato come bordo precedentemente, è possibile descrivere l'equazione della retta nel seguente modo: y_i=mx_i+c. Quest'ultima è riscrivibile come c=-mx_i+y_i, creando una equazione di una retta in cui le incognite sono m e c, essendo (x_i, y_i) noti. Ciò permette di guardare il problema sotto due domini differenti:

- il dominio dell'immagine, dove le incognite sono (x,y)
- il dominio dei parametri, dove le incognite sono (m,c)

Preso un punto nel dominio dell'immagine, esso diventerà una retta nel dominio dei parametri. In particolare ogni punto appartenente alla stessa retta nel dominio dell'immagini individua un fascio di rette nel dominio dei parametri.

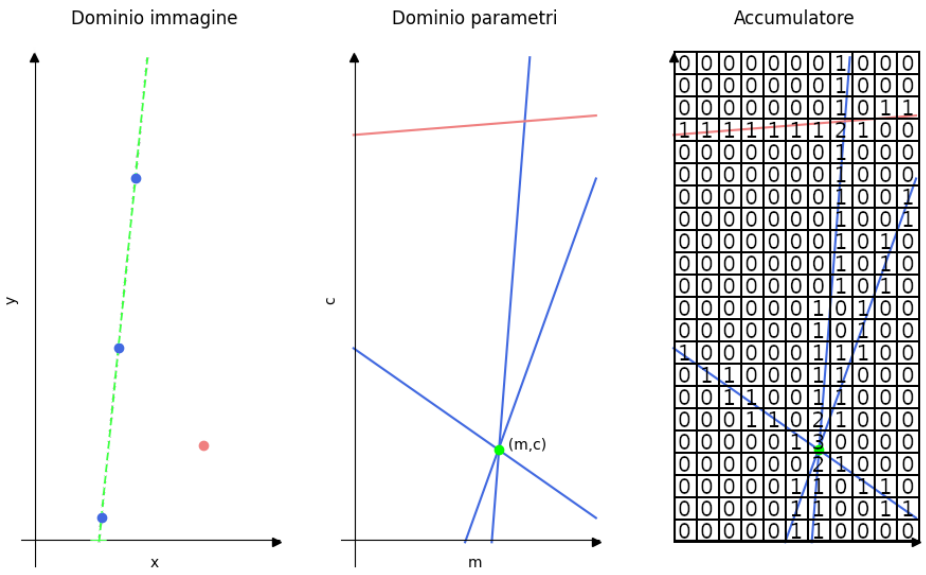

Figura 2.5: Esempio di correlazione tra punti e rette nei due dominio e di "schema di voto" dell'accumulatore

In particolare questo fascio di rette sarà individuato da un punto (m,c) che individua nel dominio delle immagini proprio la retta su cui giacciono i punti inizialmente presi, vedi Figura 2.5. Questa caratteristica viene implementata in un algoritmo attraverso la creazione di un "array di accumulazione" A(c,m) che rappresenta il dominio dei parametri inizializzato inizialmente a zero. Per ogni punto $(x_i,\ y_i)$, si aumenta di uno l'accumulatore dei punti che rappresentano la retta $(x_i,\ y_i)$ nel dominio dei parametri, implementando così uno schema di voto. Si noti come i punti nel dominio dei parametri nei quali si intersecano più rette avranno un valore maggiore rispetto agli altri punti. Infine trovando i massimi locali dell'accumulatore otterremo le linee rette nell'immagine iniziale.

Il problema di questa implementazione è che la variabile m può assumere valori tra -∞ e +∞ per cui sarebbe necessario l'utilizzo di accumulatori, e quindi memoria, altrettanto grandi. La soluzione è l'utilizzo della forma goniometrica delle equazioni lineari:

$$xsin(\theta) - ycos(\theta) + \rho = 0$$

In questo caso sia θ che ρ sono finiti. Il primo è compreso tra 0 (incluso) e π (escluso) ed indica l'angolo rispetto all'asse x del segmento che congiunge l'origine con la retta formando un angolo perpendicolare con essa. Il secondo

indica la distanza della linea retta dall'origine e nel nostro caso non può essere maggiore della diagonale dell'immagine in ingresso.

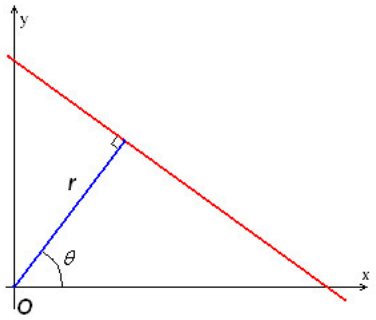

Figura 2.6: Rappresentazione di una retta in forma goniometrica

Utilizzando questa notazione un punto nel dominio dell'immagine rappresenta una sinusoide nel dominio dei parametri. Inizialmente era stata utilizzata la funzione:

```
cv.HoughLines(image, rho, theta, threshold) ->lines
```

Quest'ultima restituisce come output tutti i punti (quindi le rette nel dominio dell'immagine) che sono superiori alla threshold indicata, non permettendo però di compiere elaborazioni utili sull'accumulatore per cui questa implementazione è stata scartata. Successivamente era stata implementata una funzione python in grado di restituire l'output voluto ma essa risultava troppo lenta per permettere l'esecuzione in tempo reale. La soluzione finale è stata l'utilizzo della seguente funzione:

```
skimage.transform.hough_line(image, theta=None)->hspace, angles,
                                    distances
```

presente nella libreria scikit-image. La funzione prende in ingresso una immagine in cui i valori diversi zero identificano i bordi e un array che identifica gli angoli con i quali viene calcolata la trasformata, in radianti. Essa restituisce un array bidimensionale che rappresenta l'accumulatore, gli angoli e le distanze che definiscono le ordinate e le ascisse come mostrato in Figura 2.7. I valori dell'accumulatore vengono interpretati in scala di grigio, dove il nero rappresenta lo zero e i valori a salire corrispondono in un colore sempre più chiaro.

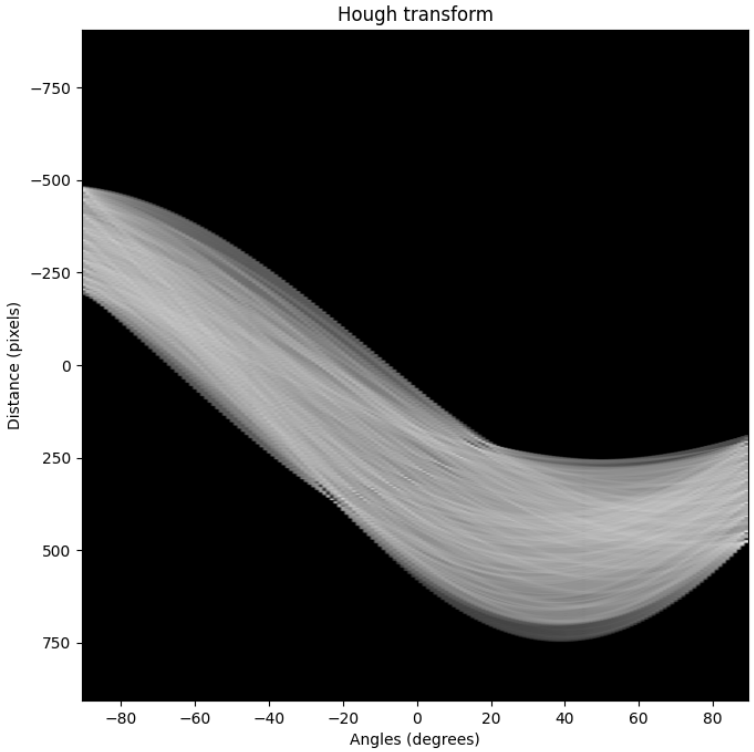

Figura 2.7: Accumulatore

2.3 Massimi e Eliminazione Outliers

A questo punto si può notare come nell'accumulatore siano facilmente visibili due gruppi di punti che giacciono su due rette. Un primo gruppo compreso tra i -20 e +20 gradi: ad ognuno di questi punti corrisponde una retta verticale della scacchiera, i loro angoli variano poichè a cause della prospettiva queste linee che nella realtà sono parallele non risultano tali invece in foto. L'altro gruppo di massimi è invece "diviso": esso si trova esattamente sulla linea dei 90 gradi e quindi anche dei -90. Ciò è dato dal fatto che le linee orizzontali della scacchiera rimangono perfettamente tali anche in foto ed essendo parallele all'asse x esse avranno $\theta = 90$. Si è deciso di dividere l'accumulatore in due zone più piccole in modo da semplificare successivamente l'estrazione dei massimi. Il primo accumulatore è la zona tra -45 e 45 gradi e racchiude tutti i punti che rappresentano le linee verticali, mentre per il secondo si è deciso di prendere la zona tra 45 e 90 gradi, invertirla rispetto all'asse x, e di unirla all'altra zona che rappresenta i -90 e -45 gradi.

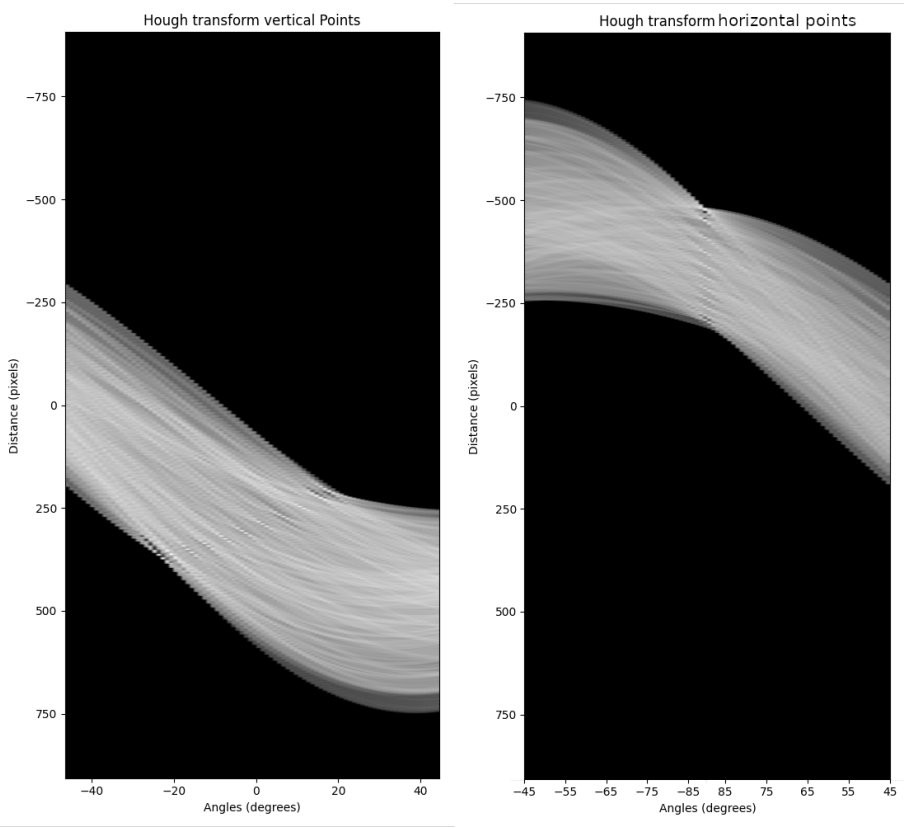

Figura 2.8: A sinistra l'area di interessa dei punti che individuano le righe verticali, a destra le linee orizzontali

Successivamente ad ognuno di questi due accumulatori è stata applicata la funzione:

```
skimage.transform.hough_line_peaks(hspace, angles, dists,
                                   min_distance=9, min_angle=10,
                                   threshold=None, num_peaks=inf) ->
                                    hspace, angles, dists
```

che è in grado di restituire i massimi locali sulla base di alcune variabili. Inizialmente in questa funzione era stato utilizzato per la variabile num_peaks il valore 9. Da alcune prove è emerso che non sempre le 9 rette individuate erano quelle corrette, per cui è stato deciso di aumentare a 12 il numero di massimi da trovare e successivamente rimuovere gli outliers, permettendo così una robustezza maggiore. Tra i falsi positivi più probabili vi sono i bordi della scacchiera che, essendo paralleli alle colonne e alle traverse, giacciono sulle stesse due rette in cui giacciono gli altri due gruppi di punti. Per eliminare gli outliers verticali viene utilizzato il seguente modo:

1. si ordinano per distanza dall'origine tutte le rette orizzontali trovate e si sceglie la retta al centro di questo insieme.

2. con la retta scelta si trovano i punti di intersezione con tutte le rette verticali.

3. è possibile utilizzare la distanza tra questi punti per eliminare gli outliers sapendo che le linee verticali in foto avranno distanze molto simili, al contrario i bordi della scacchiera non si troveranno a questa distanza.

Inizialmente si era deciso di utilizzare la distanza tra i punti dell'accumulatore o la loro ascissa come criterio per eliminare gli outliers ma ciò produceva scarsi risultati soprattutto per quanto riguardava l'individuazione degli outliers verticali. Viene utilizzato lo stesso sistema per filtrare le linee orizzontali con l'accortezza che le distanze in questo caso non saranno costanti perché subiscono l'effetto della prospettiva. Quindi più le linee si avvicineranno all'origine (angolo in alto a sinistra della foto) più la distanza tra una linea e l'altra si avvicinerà. In Figura 2.9 è possibile vedere i massimi selezionati.

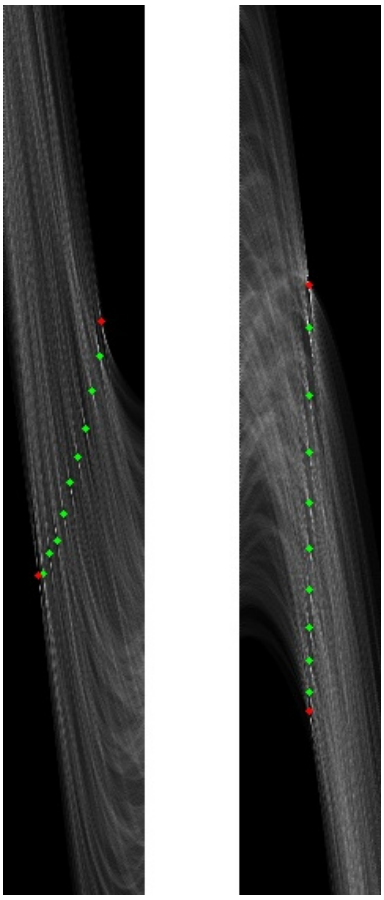

Figura 2.9: In verde i punti che rappresentano le linee delle traverse e delle colonne, in rosso i punti scartati

Concluso quest'ultimo passo è possibile sapere quali pixel dell'immagine appartengono a quale casa della scacchiera usando le 18 linee trovate. Il risultato finale è visibile in Figura 2.10.

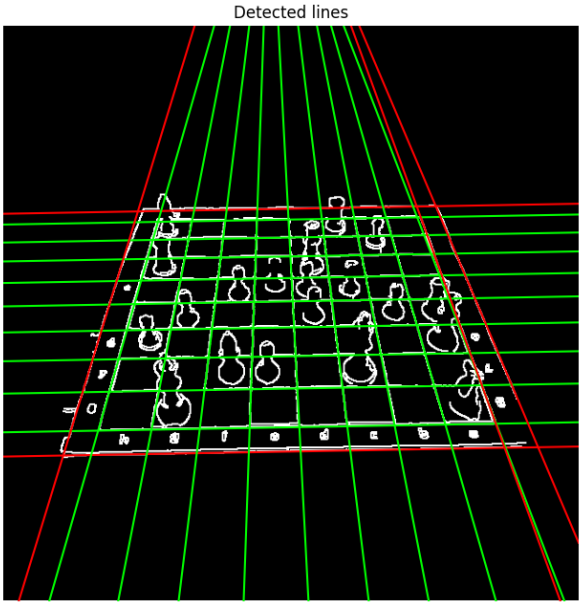

Figura 2.10: In verde vengono mostrate le rette utili, in rosso le rette scartate

Capitolo 3

Rilevamento Pezzi

Il secondo obiettivo da raggiungere consiste nell'individuazione dei pezzi, in particolare recuperare le coordinate che essi occupano all'interno dell'immagine e il "tipo" di pezzo (pedone bianco, pedone nero, cavallo bianco, ecc). Queste specifiche sono ottenute utilizzando una rete neurale convoluzionale (CNN) specializzata nel rilevamento degli oggetti. Si è deciso di utilizzare il metodo di transfer learning grazie al quale si utilizza un modello già allenato per svolgere un determinato riconoscimento come punto di partenza per lo sviluppo di un secondo modello specifico per un altro compito, permettendo di ottenere reti più precise con meno dati e in meno tempo. Per il progetto sono stati fatti vari tentativi con differenti modelli e framework tra i quali: YOLOv5 con Pytorch, MobileNet e EfficientNet con TensorFlow.

3.1 Dataset

Inizialmente il dataset usato era proveniente da un progetto trovato in rete. Esso era composto da 289 foto per un totale di 2870 annotazioni. Fin dai primi test però si è notato come i modelli allenati su questo dataset fossero poco precisi nei test reali, per questo motivo si è deciso di allargare il dataset con ulteriori 254 foto (circa 4000 annotazioni). L'obiettivo è stato quello di fornire al modello un dataset più variegato con situazioni di luce e inquadrature delle foto diverse. Inoltre è stata applicata una operazione di data augmentation in modo da ottenere un dataset più grande.

Infine per diminuire la probabilità di falsi positivi sono state introdotte anche diverse immagini di background, ovvero immagini in cui non vi sono oggetti del dominio da riconoscere. Il dataset è stato gestito usando la piattaforma *roboflow.com* che ha permesso l'etichettatura dei nuovi dati e l'esportazione in diversi formati.

3.2 YOLOv5

YOLO, acronimo di You Only Look Once, è un popolare modello di rilevamento di oggetti rilasciato nel 2015, la sua architettura prevede che i primi livelli abbiano il compito di estrarre le "features" dall'immagine in input che successivamente vengono passate a un sistema di previsione in grado disegnare rettangoli attorno agli oggetti individuati e identificarli con una classe. L'architettura YOLOv5 [2] viene rilasciata in reti con diverse dimensioni: nano, small, medium, large e extra large. Queste reti prendono in ingresso lo stesso input (640x640 pixels) ma il numero di livelli e la loro dimensione cambia, le reti più semplici permettono di avere inferenze più veloci con una precisione minore, al contrario reti più complesse permettono di avere una precisione migliore al costo di un tempo di inferenza maggiore. Dato che la rete dovrà essere distribuita su un sistema embedded questa duttilità ha permesso di poter sperimentare diverse reti e in base ai risultati scegliere quella più adatta dati i vincoli di potenza di calcolo e memoria. L'output della rete è un tensore delle seguenti dimensioni (1, 25200, num_classi+5). Nel caso di questo progetto le classi da individuare sono 12. In particolare la rete restituirà l'insieme di tutti i rettangoli di selezioni e per ognuno di esse saranno fornite le seguenti informazioni:

- la coordinata x e y del centro del rettangolo
- la larghezza w e l'altezza h del rettangolo
- un valore compreso tra 0 e 1 che rappresenta il grado di certezza del rettangolo
- infine per ogni classe, un valore compreso tra 0 e 1 che indica il grado di certezza relativo al tipo di oggetto individuato

Nel progetto si è deciso di identificare in maniera puntuale la posizione del pezzo utilizzando il pixel che si trova a metà della base del rettangolo di selezione traslato 10 pixel verso l'alto.

3.3 Allenamento YOLOv5 e Risultati Ottenuti

L'allenamento di questa rete è facilmente eseguibile attraverso lo script train_.py presente nella repository del progetto. E' sufficiente specificare:

- il path del file .yaml che specifica la posizione del dataset, il numero di classi e i loro nome
- il numero di epoche di cui sarà composto l'allenamento
- i pesi iniziali della rete, in questo caso si è scelto il file yolov5m.pb
- il file di configurazione della rete

- infine la dimensione di batch, se si specifica -1 viene calcolata in automatico la dimensione migliore

Inoltre vengono aggiunti anche il comando -device 0 per indicare che è possibile utilizzare la scheda grafica per compiere l'allenamento e il flag –cache che permette di caricare in memoria il dataset.

```
python train.py --data ./data.yaml --epochs 300
--weights "yolov5m.pb" --cfg yolov5m.yaml --batch-size -1
--device 0 --cache
```

Al termine dell'allenamento lo script salverà i pesi della rete dell'ultima epoca e anche i pesi che hanno restituito i valori migliori durante l'allenamento. Ora è possibile eseguire la validazione dei nuovi pesi per ottenere le metriche che determinano la precisione.

```
python val.py --weights best.pt --data data.yaml --img 640
```

Ecco i risultati per la rete YoloV5m:

```
YOLOv5m summary: 212 layers, 20897385 parameters, 0 gradients, 48.0 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95:
                   all         67        827      0.976      0.972      0.985      0.849
          black-bishop         67         58      0.942      0.948      0.962      0.838
            black-king         67         50          1      0.943      0.982      0.865
          black-knight         67         62      0.983      0.951      0.979      0.811
            black-pawn         67        143      0.986      0.993      0.989      0.856
           black-queen         67         39      0.925          1      0.987      0.865
            black-rook         67         58      0.982      0.938       0.99      0.866
          white-bishop         67         56       0.98      0.982      0.989      0.823
            white-king         67         47      0.975          1      0.991      0.857
          white-knight         67         68      0.985          1      0.995      0.857
            white-pawn         67        144      0.998      0.986      0.995      0.856
           white-queen         67         41      0.998      0.976       0.98      0.853
            white-rook         67         61      0.962      0.951      0.983      0.842
```

Figura 3.1: Risultati dell'allenamento

3.4 TensorFlow: Efficient-det e Mobilenet

Durante la realizzazione del progetto si è voluto inoltre provare il framework TensorFlow su due reti, la EfficientDet e la SSD MobileNet, usando due librerie diverse per l'allenamento dei modelli. Per la prima rete si è usata la libreria TensorFlow Lite Model Maker [3], essa semplifica il processo di allenamento e permette di ottenere un modello compatibile con la libreria Task [4]. Quest'ultima è una libreria di alto livello che consente la distribuzione di modelli su diverse piattaforme, tra le quali i dispositivi mobili, fornendo diverse interfacce che semplificano l'utilizzo di reti neurali.

La libreria Model Maker, al contrario dell'esempio precedente, richiede un dataset con un formato diverso, il Pascal Visual Object Classes (Pascal VOC).

Roboflow è in grado di convertire il dataset anche in questo formato, per cui una volta esportato il nuovo dataset è possibile utilizzarlo come mostrato in seguito:

```
from tflite_model_maker.config import ExportFormat
from tflite_model_maker import model_spec
from tflite_model_maker import object_detector
import tensorflow as tf
assert tf.__version__.startswith('2')

label_map = ['black-bishop', 'black-king', 'black-knight', 'black-
                                    pawn', 'black-queen', 'black-rook
                                    ', 'white-bishop', 'white-king',
                                    'white-knight', 'white-pawn', '
                                    white-queen', 'white-rook']
train_data = object_detector.DataLoader.from_pascal_voc("./
                                    SistemiDigitaliM.v6-final-version
                                    .voc/train","./SistemiDigitaliM.
                                    v6-final-version.voc/train",
                                    label_map)
validation_data = object_detector.DataLoader.from_pascal_voc("./
                                    SistemiDigitaliM.v6-final-version
                                    .voc/valid","./SistemiDigitaliM.
                                    v6-final-version.voc/valid",
                                    label_map)
test_data = object_detector.DataLoader.from_pascal_voc("./
                                    SistemiDigitaliM.v6-final-version
                                    .voc/test","./SistemiDigitaliM.v6
                                    -final-version.voc/test",
                                    label_map)
```

Successivamente si caricano le specifiche del modello EfficientDet modificando però il numero di previsioni massime che la rete è in grado di fare (di default è 25), ovvero 32.

```
spec = model_spec.get('efficientdet_lite2')
spec.config.tflite_max_detections = 32
```

Infine si allena il modello specificando:

- il dataset di allenamento
- le specifiche del modello
- se allenare tutto il modello o solo una parte
- il dataset di validazione
- il numero di epoche

```
model = object_detector.create(train_data, model_spec=spec,
                                    batch_size=8, train_whole_model=
                                    True, validation_data=
                                    validation_data, epochs=150)
accuracy = model.evaluate(test_data)
model.export(export_dir='.', export_format=[ExportFormat.
                                    SAVED_MODEL, ExportFormat.LABEL,
                                    ExportFormat.TFLITE])
```

```
accuracy_tflite = model.evaluate_tflite('model.tflite', test_data)
```

Concluso l'allenamento è possibile valutare il modello creato e successivamente esportare questo modello in formato SavedModel o TFLite. Di default viene esportato un modello con applicata la Full Integer Quantization ma è possibile specificare il tipo di quantizzazione nel seguente modo:

```
config = QuantizationConfig.for_float16()
```

e passando questo valore nella funzione *create*.
I possibili tipi di quantizzazione sono i seguenti:

- for_dynamic(): utilizza la quantizzazione di tipo Dynamic Range Quantization
- for_float16(): tutti i pesi vengono quantizzati nel formato float16, riducendo le dimensioni del modello e dando la possibilità a sistemi in grado di operare con questo formato, per esempio le GPU, di svologere le operazioni più velocemente
- for_int8(): in questo caso il modello utilizzerà solo numeri nel formato int8, permetto la compatibilità con hardware capace solo di gestire questo formato, migliorando inoltre le performance. Per utilizzare questo tipo di quantizzazione è necessario fornire un dataset rappresentativo per calibrare il range min max di alcuni tensori del modello.

I risultati, visibili in figura 3.2, mostrano risultati meno precisi rispetto al modello YOLO

```
SavedModel accuracy:
{'AP50-95': 0.78498876,
'AP50': 0.9652525,
'AP50-95/black-bishop': 0.8075087,
'AP50-95/black-king': 0.80322194,
'AP50-95/black-knight': 0.76326823,
'AP50-95/black-pawn': 0.75185144,
'AP50-95/black-queen': 0.78310716,
'AP50-95/black-rook': 0.7854415,
'AP50-95/white-bishop': 0.7511376,
'AP50-95/white-king': 0.85616726,
'AP50-95/white-knight': 0.72359246,
'AP50-95/white-pawn': 0.780071,
'AP50-95/white-queen': 0.8192053,
'AP50-95/white-rook': 0.79529274}
```

Figura 3.2: Risultati dell'allenamento, EfficientDet

Per l'allenamento della rete SSD MobileNet sono stati utilizzati i file model_main_tf2.py e export_tflite_graph_tf2.py presenti nella repository Github di Tensorflow [5]. Una volta scaricato il modello dalla pagina TensorFlow 2 Detection Model Zoo [6], è necessario configurare il file pipeline.config che specifica i parametri dell'allenamento. Il parametro num_classes è stato modificato a 12,

batch_size a 8,e si sono specificati il path per il dataset e il path per il file che specifica la label map. In questo caso il formato del dataset è TFRecord. In aggiunta, dalle varie prove, è emerso che aumentare la variabile learning_rate_base a 0.11 ha permesso di ottenere dei risultati migliori. Successivamente è stato utilizzato lo script model_main_tf2.py specificando i seguenti parametri:

- il path per il file pipeling.config
- il path per la directory dove verrà salvato il nuovo modello
- un intero che specifica ogni quanto deve essere creato un checkpoint del modello

Concluso l'allenamento, verrà salvato un nuovo modello in formato "frozen graph" che, attraverso lo script export_tflite_graph_tf2.py sarà convertito nel formato SavedModel.

A questo punto è stato scritto il seguente script che permette, partendo da un modello SavedModel, di ottenere 3 modelli TFlite con 3 diversi tipi di quantizzazione.

```
import os
import tensorflow as tf
import numpy as np
import cv2
import random

def representative_dataset_gen():
    list_images = os.listdir("./train/images")
    for i in range(100):
        randomNum = random.randint(0, len(list_images))
        print("RandomNum: ", randomNum)
        pfad = "./train/images/"+list_images[randomNum]
        img = cv2.imread(pfad)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = (img - 127.5) / 127.5
        img = np.expand_dims(img, 0).astype(np.float32)
        yield [img]

PATH_SAVEDMOEL = "./saved_model"
# Convert the model
converterDRQ = tf.lite.TFLiteConverter.from_saved_model(
                                    PATH_SAVEDMOEL)
converterFIQ = tf.lite.TFLiteConverter.from_saved_model(
                                    PATH_SAVEDMOEL)
converterF16Q = tf.lite.TFLiteConverter.from_saved_model(
                                    PATH_SAVEDMOEL)

converterDRQ.optimizations = [tf.lite.Optimize.DEFAULT] # Dynamic
                                    range quantization
tflite_model = converterDRQ.convert()
with open('mobilenetFinal_drq.tflite', 'wb') as f:
    f.write(tflite_model)


converterFIQ.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
converterFIQ.experimental_new_quantizer = True
converterFIQ.target_spec.supported_ops = [tf.lite.OpsSet.
                                    TFLITE_BUILTINS_INT8, tf.lite.
                                    OpsSet.TFLITE_BUILTINS]
converterFIQ.representative_dataset=representative_dataset_gen #
                                    Full integer quantization
tflite_model_quant = converterFIQ.convert()
with open('mobilenetFinal_quant_fiq.tflite', 'wb') as f:
    f.write(tflite_model_quant)


converterF16Q.optimizations = [tf.lite.Optimize.DEFAULT] #Float16
                                    quantization
converterF16Q.target_spec.supported_types = [tf.float16]
tflite_fp16_model = converterF16Q.convert()
with open('mobilenetFinal_quant_F16Q.tflite', 'wb') as f:
    f.write(tflite_fp16_model)
```

Di seguito vengono mostrati i risultati della rete con Full Integer Quantization.

```
Num evaluation runs: 63
Preprocessing latency: avg=8390.57(us), std_dev=0(us)
Inference latency: avg=81670.1(us), std_dev=3244(us)
Average Precision [IOU Threshold=0.5]: 0.940904
Average Precision [IOU Threshold=0.55]: 0.939779
Average Precision [IOU Threshold=0.6]: 0.939017
Average Precision [IOU Threshold=0.65]: 0.936923
Average Precision [IOU Threshold=0.7]: 0.921678
Average Precision [IOU Threshold=0.75]: 0.907057
Average Precision [IOU Threshold=0.8]: 0.864892
Average Precision [IOU Threshold=0.85]: 0.708282
Average Precision [IOU Threshold=0.9]: 0.41922
Average Precision [IOU Threshold=0.95]: 0.0362887
Overall mAP: 0.761404
```

Figura 3.3: Risultati dell'allenamento MobileNet Full Integer Quantization

Capitolo 4

Implementazione Android

Le soluzioni trovate sono state poi implementate all'interno di una applicazione Android. Quest'ultima risulta in grado di analizzare la scena inquadrata dalla fotocamera dello smartphone e quindi verificare in tempo reale il riconoscimento o meno di una scacchiera. A questo punto è possibile svolgere una inferenza per il riconoscimento dei pezzi e di conseguenza è possibile aprire una schermata di valutazione della posizione corrente della partita. Lo sviluppo e i test sono stati eseguiti sullo smartphone OnePlus 7 (GM1900).

4.1 Progetto Android

L'applicazione Android è stata creata usando l'ambiente di sviluppo Android Studio e il linguaggio di programmazione Kotlin (1.7.20). Per il progetto sono stati creati i seguenti file:

- `Piece.kt`, classe che rappresenta un pezzo della scacchiera, include la classe e la sua posizione nell'immagine
- `Chessboard.kt`, un singleton che rappresenta la scacchiera, include informazioni come la lista dei Piece presenti, se è stata effettuata una inferenza, la lista dei punti che determinano le 18 linee che formano la scacchiera e altre funzioni come, per esempio, la creazione della stringa FEN
- `BestMoveThread.kt`, una classe che estende la classe Thread, usata per eseguire la ricerca della mossa migliore utilizzando Stockfish data la notazione FEN
- `Utils.kt`, un object che raggruppa diverse funzioni di utilità
- `PrePostProcessor.kt`, un singleton utilizzato per processare i risultati ottenuti dalle inferenze del modello YOLOv5
- `chessboard_detection.py`, script Python per il riconoscimento della scacchiera

- infine `MainActivity.kt` e `EvaluationActivity.kt` che racchiudono la logica delle due schermate presenti nell'applicazione

L'elaborazione delle immagini è stata eseguita usando la libreria CameraX, in particolare utilizzando lo use case image analysis. All'interno del metodo analyze(), eseguito per ogni frame, è implementato il riconoscimento della scacchiera. Se il riconoscimento ha esito positivo vengono aggiornati i relativi campi all'interno del singleton Chessboard e si disegnano le linee trovate nel frame per poi mostrarlo nell'interfaccia.

4.2 Interfaccia Utente

L'interfaccia principale si presenta in questo modo: la metà superiore dello schermo mostra ciò che viene inquadrato dalla fotocamera dello smartphone, la parte inferiore mostra, una volta svolta l'inferenza utilizzando il bottone TAKE PHOTO, la rappresentazione digitale della scacchiera. In basso è possibile trovare il bottone SWITCH SIDE che permette di scambiare la posizione del nero e del bianco, e il pulsante EVAL BOARD che porta invece alla seconda schermata.

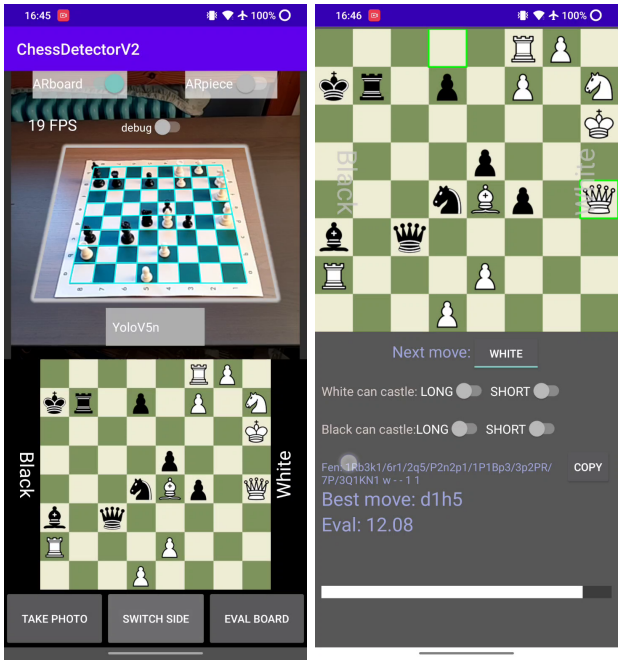

Figura 4.1: A sinitra l'interfaccia principale, a destra la schermata della valutazione

La seconda schermata mostra la posizione rilevata precedentemente con l'aggiunta di una grafica che indica la prossima migliore mossa. Sotto è presente

un bottone che consente di decidere se la prossima mossa è del Bianco o del Nero, segue poi una serie di switch che consente di specificare se è possibile l'arrocco da parte dei giocatori. Con queste informazioni è possibile mostrare la notazione Fen della configurazione ed è possibile copiarla nella clipboard attraverso il pulsante COPY. Infine viene mostrato per iscritto la mossa migliore e la valutazione in centipawn con una barra che rappresenta graficamente questo valore.

4.3 Chaquopy

Poichè il riconoscimento della scacchiera è stato implementato usando il linguaggio Python è stato necessario usare la libreria Chaquopy. In questo caso si invoca la funzione main presente nel file chessboard_detection.py passando il frame sotto forma di ByteArray, la sua larghezza e altezza attraverso due valori Int. Nello script Python è quindi necessario trasformare il ByteArray in un numpy array creando prima un oggetto ByteArray, trasformandolo successivamente in un numpy array monodimensionale e successivamente, avendo a disposizione la larghezza e l'altezza si invoca la funzione reshape come mostrato di seguito.

```
def main(image, width, height, debug):

    ba = bytearray(image)
    npArray = np.frombuffer(ba, dtype = np.uint8)
    npArray = npArray.reshape(height, width ,4)
    npArray = npArray[:,:,:3]
    inputImg = cv2.cvtColor(npArray, cv2.COLOR_BGR2RGB)
    ...
```

Lo script restituirà, sempre sotto forma di bytearray, nel caso di riconoscimento fallito il valore -1, nel caso di riconoscimento effettuato il valore 1 seguito dalla lista dei punti che individuano le linee trovate.

4.4 Conversione delle Reti per Sistemi Embedded

Il modello YOLOv5 addestrato precedentemente deve essere convertito in un formato adatto all'utilizzo su sistemi embedded per questo motivo il framework Pytorch possiede la funzione *torch.mobile_optimizer.optimize_for_mobile* capace di ottimizzare diverse operazioni all'interno del modello. Successivamente viene usata la funzione *_save_for_lite_interpreter* per esportare il modello nel formato ".ptl" compatibile con l'interprete disponibile per Android. Per quanto riguarda i modelli SSD MobileNet e l'Efficient-Det in versione TFLite, la libreria Tensorflow permette di scegliere quali risorse utilizzare per eseguire le inferenze, le varie opzioni sono:

- CPU, non specificando nulla ma dando la possibilità di scegliere il numero di thread da utilizzare attraverso la funzione setNumThread().

- GPU, invocando la funzione useGpu(), permettendo, alle operazioni che sono compatibili di essere eseguite nella Gpu del sistema mobile.
- NNAPI, invocando la funziona useNnapi(), permette l'utilizzo di Android Neural Networks API che fornisce accelerazione per le operazioni legate all'utilizzo di modelli TFLite.

Attraverso alcune prove si è verificato come alcuni modelli e alcuni tipi di quantizzazione abbiano performance migliori con l'utilizzo di determinate risorse.

Durata Inferenza (media 50 inferenze in ms)			
	4 Thread	GPU	NNAPI
EfficientDet 448x448 (Dynamic Range Quantization)	77.88	107.07	81.50
MobileNet 640x640 (Full Integer Quantization)	83.15	137.57	113.43
MobileNet 640x640 (Float 16 Quantization)	136.13	186.98	255.92
MobileNet 640x640 (Dynamic Range Quantization)	193.44	178.57	307.99

Tabella 4.1: Test effettuati con OnePlus 7 GM1900

Da notare che quando si è scelto di utilizzare la GPU, la console di Android Studio ci avvisa di come alcune operazioni non sono supportate come mostrato in Figura 4.2. Ciò soprattutto per la rete MobileNet non permette un efficiente utilizzo delle risorse portando ad avere prestazioni non ottime con l'utilizzo di GPU in Android.

```
I/tflite: Created 1 GPU delegate kernels.
I/tflite: Replacing 33 node(s) with delegate (TfLiteXNNPackDelegate) node, yielding 7 partitions.
I/acceleration: ModifyGraphWithDelegate model namespace: unknown_namespace model id: unknown_model_id accelerator name: GPU
E/tflite: Following operations are not supported by GPU delegate:
    CUSTOM TFLite_Detection_PostProcess: TFLite_Detection_PostProcess
    356 operations will run on the GPU, and the remaining 1 operations will run on the CPU.
I/tflite: Replacing 356 node(s) with delegate (TfLiteGpuDelegateV2) node, yielding 2 partitions. <1 internal line>
```

Figura 4.2: La rete EfficientDet permette l'esecuzione di tutte le operazioni tranne 1 sulla GPU

```
I/tflite: Created TensorFlow Lite XNNPACK delegate for CPU.
I/tflite: Initialized TensorFlow Lite runtime.
E/tflite: Following operations are not supported by GPU delegate:
    CUSTOM TFLite_Detection_PostProcess: TFLite_Detection_PostProcess
    PACK: OP is supported, but tensor type/shape isn't compatible.
    RESHAPE: OP is supported, but tensor type/shape isn't compatible.
    112 operations will run on the GPU, and the remaining 48 operations will run on the CPU.
I/tflite: Replacing 112 node(s) with delegate (TfLiteGpuDelegateV2) node, yielding 2 partitions
```

Figura 4.3: La rete MobilenNet permette l'esecuzione di 112 operazioni sulla GPU, le restanti 48 saranno eseguite sulla CPU

Nell'implementazione finale si è deciso di utilizzare il modello YOLOv5 sia perchè risulta più preciso sia per il motivo spiegato nella sezione 4.5. Di seguito vengono riportate le performance della rete YOLOv5.

Durata Inferenza (media 50 inferenze in ms)	
YOLOv5n	176.48
YOLOv5s	411.21
YOLOv5m	867.45

Tabella 4.2: Test effettuati con OnePlus 7 GM1900

4.5 Inferenze

Le inferenze relative al modello YOLOv5 vengono svolte all'interno dello use case Image Capture, nel metodo takePicture(). Una volta invocato il metodo forward() utilizzando l'immagine fotografata, il tensore in output viene elaborato utilizzando la funzione outputsToNMSPredictions() presente nel file PrePostProcess.kt che permette di ottenere gli oggetti Piece. Una differenza sostanziale rispetto all'utilizzo della libreria di alto livello Object Detection API di Tensorflow è che in questo caso per ogni risultato ottenuto vengono salvati tutte le soglie di confidenza per ogni classe di ogni rilevamento. Questi dati vengono poi utilizzati per permettere la correzione di alcuni errori nell'inferenza: se sono stati rilevati per esempio 3 oggetti appartenenti alla classe white-knight, l'oggetto con la confidenza più bassa viene cambiato con una classe con confidenza più alta la cui cardinalità non ha raggiunto il massimo. Utilizzando la libreria Object Detection API di Tensorflow invece la funzione detect() restituirà semplicemente una lista di oggetti Detection con i seguenti campi:

- classe
- score
- rettangolo di selezione

4.6 Implementazione Stockfish

Per l'implementazione di Stockfish all'interno dell'applicazione si è proceduto nel seguente modo:

1. Si scarica dal sito ufficiale [7] il binario compatibile con il sistema operativo Android
2. All'interno del progetto Android nel file build.gradle(app) is aggiunge il seguente codice:

```
sourceSets {
    main {
        jniLibs.srcDirs = ['libs']
        jni {
            srcDirs 'src/main/jni', 'src/main/jniLibs'
        }
    }
}
```

3. Nello stesso file si aggiunge la seguente riga di codice:

```
implementation fileTree(dir: "libs", include: ["*.jar", "*.so"
                                               ])
```

4. Successivamente si rinomina il file binario in modo da avere come estensione ".so", e infine si crea la struttura mostrata in figura all'interno della nuova cartella jniLibs del progetto

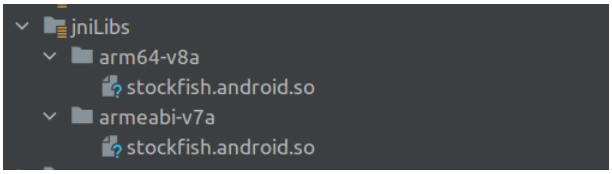

Figura 4.4:

A questo punto è possibile eseguire Stockfish all'interno della nostra applicazione lanciandolo come se fosse un normale programma con il seguente codice:

```
val path = applicationContext.applicationInfo.nativeLibraryDir + "/
                                    stockfish.android.so"
val file = File(path)
var process = Runtime.getRuntime().exec(file.path)
```

Quest'ultimo si comporterà come un programma a linea di comando per cui è necessario utilizzare degli oggetti di tipo BufferedWriter e BufferedReader per poter comunicare con esso. Nel progetto vengono usati i seguenti comandi:

- `isready`, comando che verifica se il motore è "alive", in caso affermativo risponde con "readyok"

- `position fen *fen string*`, imposta la posizione all'interno del motore, non c'è output
- `go movetime *mill_secs*`, inizia la ricerca della migliore mossa per i millisecondi specificati, successivamente a questo comando il motore inizierà a scrivere in output una serie di informazioni man mano che procede la sua ricerca in modo da poter modificare la GUI in maniera appropriata. Tra queste informazioni vengono mostrate a schermo la mossa migliore e la valutazione della posizione.
- `stop`, indica al motore di smettere di calcolare il prima possibile, come output fornisce la migliore mossa trovata fino a quel momento
- `quit`, termina il processo

Capitolo 5

Conclusione

In questo progetto è stata realizzata una applicazione per un sistema embedded in grado di fornire utili informazioni per l'analisi di una partita di scacchi giocata dal vivo. In particolare sono state implementate soluzioni riguardanti l'utilizzo della computer vision per il riconoscimento della scacchiera e l'utilizzo di reti neurali per il riconoscimento dei pezzi. L'applicazione ha dato risultati soddisfacenti, anche se vi sono presenti alcuni limiti, primo fra tutti il fatto che la rete neurale è stata addestrata solo su uno specifico set di pezzi che, pur essendo uno dei set più usati, limita l'utilizzo in maniera significativa. Un'ulteriore limitazione riguarda le situazioni in cui l'ambiente è scarsamente illuminato o alcuni pezzi occludono considerevolmente altri pezzi. In questo caso le predizioni perdono di precisione.

Capitolo 6

Sviluppi Futuri

Alcuni possibili sviluppi futuri sono:

- estendere il riconoscimento della scacchiera in modo che non sia più necessario essere perpendicolari ad essa ma in modo che sia riconosciuta da tutte le angolazioni
- inserire più set di pezzi in modo da rendere utilizzabile l'applicazione ad un insieme più ampio di persone
- aggiungere l'opzione di poter usare l'applicazione in modalità simil-Realtà Aumentata, dove la mossa migliore e la valutazione viene mostrata in tempo reale.
- effettuare il porting su sistema iOS
- sperimentare nuovi metodi per il riconoscimento della scacchiera

Bibliografia

[1] https://www.wbec-ridderkerk.nl/html/UCIProtocol.html

[2] https://github.com/ultralytics/yolov5

[3] https://www.tensorflow.org/lite/models/modify/model\
textunderscoremaker

[4] https://www.tensorflow.org/lite/inference\textunderscorewith\
textunderscoremetadata/task\textunderscorelibrary/overview

[5] https://github.com/tensorflow/models/tree/master/research/
object\textunderscoredetection

[6] https://github.com/tensorflow/models/blob/master/
research/object\textunderscoredetection/g3doc/tf2\
textunderscoredetection\textunderscorezoo.md

[7] https://stockfishchess.org/download/