Alma Mater Studiorum - Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Sviluppo e analisi di un algoritmo parallelo di riconoscimento di oggetti in immagini

Tesi di Laurea Magistrale

Relatore:
Dott.
Moreno Marzolla

Presentata da:
Antonello Antonacci

Sessione III
Anno Accademico 2014/2015

"Desidero innanzitutto ringraziare il professore Moreno Marzolla per la disponibilità e la gentilezza mostrate durante il tirocinio e la tesi. Inoltre vorrei dire grazie anche ai miei genitori ed a tutta la mia famiglia per avermi sempre sostenuto e aver creduto in me, rendendo possibile tutto questo. Un ringraziamento speciale lo dedico ad Antonella, sempre al mio fianco, che mi ha aiutato ad affrontare più di un momento difficile. Infine un abbraccio ai miei amici, il loro supporto è indispensabile nella vita di ogni giorno. Se ho raggiunto questo obbiettivo è anche merito di tutti voi.
Grazie di cuore."

Indice

Capitolo 1

Introduzione

L'object recognition [9] si occupa di trovare ed identificare gli oggetti all'interno di immagini o di sequenze video. È un campo vasto e giovane; per questo la ricerca in materia è molto attiva. Le applicazioni in campo scientifico possono ricoprire diversi settori: veicoli autonomi, video-sorveglianza, indicizzazione dei video, classificazione di immagini, robotica e tanti altri ancora.

Le difficoltà nel riconoscere gli oggetti in un'immagine crescono in relazione a diversi fattori, come ad esempio: condizioni di luce, posizione degli oggetti, rumore dell'immagine, distorsione e diverse caratteristiche dei vari oggetti da riconoscere.

Gli algoritmi di object recognition possono essere implementati seguendo diversi approcci; tra i più utilizzati ci sono gli algoritmi *Apparence-based* [15] e *Feature-based* [20]. Il primo approccio si basa sull'utilizzo di una serie di immagini per l'addestramento che rappresentino l'oggetto in diverse condizione di luce, rotazione e dimensione. Il secondo metodo invece partendo da un insieme di immagini estrae le feature, ossia le aree che descrivono meglio quel tipo di oggetto, a prescindere da dimensione, rotazione ed illuminazione.

L'Evolution COnstructed features [11] è il primo metodo che permette di estrarre delle feature[1] in modo automatico e senza l'intervento di un opera-

[1] rappresentazione delle informazioni relative ad un'immagine sotto forma di pixel

tore umano. Le prestazioni ottenute nel riconoscimento degli oggetti nelle immagini sono simili, ed in alcuni casi migliori, rispetto a quanto una persona esperta possa ottenere. Inoltre permette anche l'estrazione di feature non intuitive che possono risultare difficili da notare all'occhio umano e non si pone nessun tipo di limite sulle immagini di input: è quindi possibile utilizzare immagini ottenute da raggi X, risonanze magnetiche e foto ad infrarossi. Uno degli aspetti negativi dell'Eco feature, come affermano gli stessi autori dell'articolo [12], è il tempo di esecuzione elevato. L'obiettivo che si pone questa tesi è lo studio approfondito dell'algoritmo *Evolution COnstructed feature*, e l'implementazione di una versione sequenziale ed una parallela in modo da verificare quali possano essere gli effettivi guadagni in termini di tempo di esecuzione.
La parallelizzazione avviene non solo sfruttando i core della CPU, ma anche quelli della GPU attraverso la libreria OpenCV [3] che esegue una serie di funzioni utilizzate durante la fase di estrazione delle feature dalle immagini sfruttando potenzialità della scheda video. Questa libreria viene usata anche dagli autori dell'Eco Feature, ma in una versione datata che rispetto a quella attualmente disponibile non prevedeva ancora la completa implementazione delle funzioni di trasformazione delle immagini sfruttando la GPU. Difatti sono loro stessi ad affermare che con le future versioni di OpenCV si possono facilmente ottenere delle prestazioni migliori in termini di tempi di esecuzione.

1.1 Object recognition Feature-based

Una delle operazioni principali nell'ambito della computer vision [1] è la creazione di un insieme di simboli partendo da un determinato input; nell'object recognition feature-based questi simboli sono le feature. In generale le tecniche di machine learning [5] partendo da questi simboli estraggono dei pattern e li separano, attraverso l'utilizzo di modelli matematici, in varie classi. Questo processo evita all'utente di dover definire delle regole ed inol-

tre il risultato, in generale, può risultare più accurato di quanto non possa fare un essere umano esperto. Dunque si può facilmente dedurre che estrarre feature di buona qualità, partendo da un immagine, rappresenta un aspetto cruciale in questo ambito.
Esistono diverse fasi che portano all'estrazione delle feature:

- Feature Construction: genera l'insieme iniziale di feature, cerca le informazioni mancanti nelle relazioni tra di esse e aumenta lo spazio delle feature tramite inferenza o creando delle nuove feature;

- Feature Selection: è il processo che, in base a determinati criteri, seleziona un sottoinsieme di feature da quelle originali in modo da ridurre lo spazio di calcolo;

- Feature Extraction: genera delle nuove feature partendo dalle originali, in base a delle funzioni di mapping; in pratica unifica feature simili, in modo da renderle più significative ed allo stesso tempo elimina la ridondanza.

Figura 1.1: Step dell'object recognition
Fonte: *A feature construction method for general object recognition* [11]

La figura 1.1 mostra i vari passi del processo di object recognition: la fase di feature construction genera un insieme di simboli dall'immagine, quindi viene estratto un sottoinsieme dei simboli più significativi attraverso la fase

di feature selection oppure vengono creati dei nuovi simboli con delle funzioni di mapping tramite la fase di feature extraction, che si occupa anche di eliminare la ridondanza dall'insieme delle feature. Dopo queste tre fasi i simboli vengono analizzati e per ognuno di essi viene verificato se appartiene all'oggetto oppure no.
Analizzando i risultati ottenuti dai vari algoritmi di object recognition ed effettuando un confronto uomo - macchina si può affermare che la fase di feature construction è l'unica che svolta da un essere umano potrebbe offrire risultati migliori, mentre nel resto del processo le prestazioni di una macchina offrono sempre delle performance superiori. La figura 1.2 mostra una distribuzione dello svolgimento delle diverse fasi dell'object recognition tra un essere umano ed un computer. L'unica fase che nella maggior parte dei casi viene svolta ancora da un operatore umano è la fase di feature construction; mentre il resto delle operazioni vengono sempre svolte dai computer. L'algoritmo Evolution COnstructed feature implementa tutte le fasi a livello computer e non necessita dell'intervento di un operatore umano in nessuna fase.

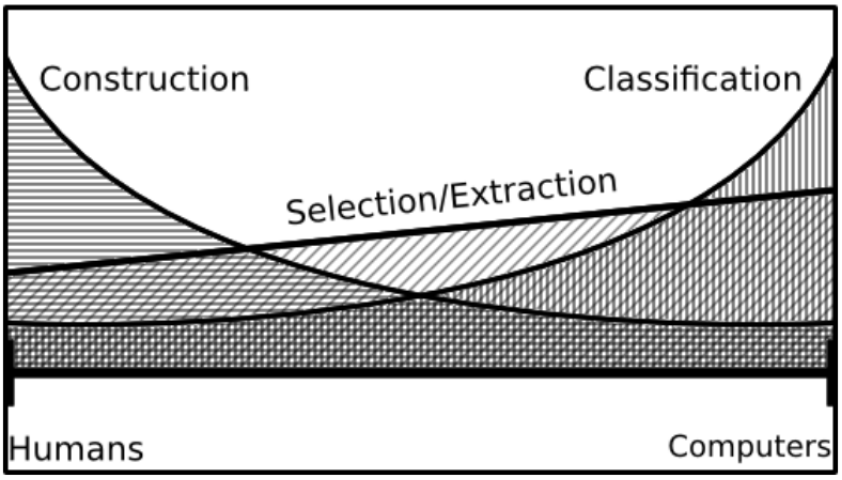

Figura 1.2: Distribuzione delle operazioni del processo di object recognition tra uomo e macchina

Fonte: *A feature construction method for general object recognition* [11]

Le feature di un'immagine sono una rappresentazione delle aree importanti sotto forma di pixel. Descrivere come gli esseri umani riescano ad individuare le feature in un'immagine è molto difficile poiché è un processo che si può definire come già programmato nel nostro cervello.
Per analizzare come viene implementato questo processo in una macchina consideriamo l'esempio mostrato in figura 1.3 ed in figura 1.4.

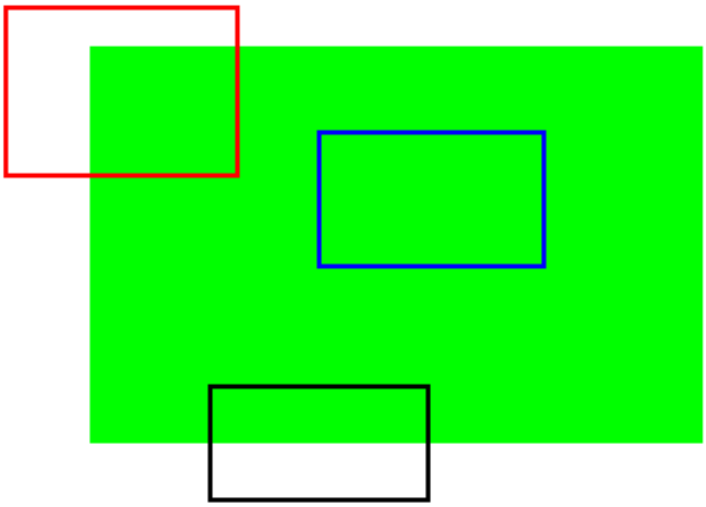

Figura 1.3: Esempio di feature in un'immagine

Nell'esempio abbiamo: un rettangolo grande di colore verde, che è l'oggetto di addestramento, e tre diverse feature identificate da rettangoli piccoli con bordi di colore diverso tra loro. Ogni feature rappresenta un'area dell'oggetto e lo scopo dell'esempio è capire quando una feature può essere considerata valida. Il rettangolo blu, posto al centro, è un'area piatta e non descrive una caratteristica dell'oggetto, per questo viene considerata una feature poco valida. Il rettangolo nero rappresenta un'area dell'immagine contenente un bordo dell'oggetto di addestramento, questo la rende una feature migliore rispetto alla precedente perché riesce a descrivere una particolarità dell'oggetto, ma non è ancora la migliore possibile. La feature migliore è quella rappresentata dal rettangolo rosso perché identifica un angolo dell'immagine; questa feature rappresenta un'area univoca dell'oggetto.

La figura 1.4 mostra l'utilizzo di queste tre feature per verificare se due oggetti sono rettangoli oppure no. La feature blu, sbagliando, classifica sia il triangolo che la circonferenza come rettangoli; in quanto trova in entrambi i casi delle aree uguali all'area estratta dal rettangolo. La feature nera, che rappresenta un bordo del rettangolo, può sbagliare nel caso del triangolo poiché trova un'area con un bordo del tutto identica a quella estratta dal rettangolo; con la circonferenza invece non sbaglia mai. Infine la feature rossa, che indica un angolo del rettangolo, non sbaglia in nessuno dei due casi dato che non trova mai un riscontro tra l'area estratta precedentemente e i due oggetti.
Dunque le feature sono dei veri e proprio pezzi dell'immagine, rappresentati tramite pixel, che vengono estratti dalle immagini di addestramento e successivamente utilizzate per cercare delle corrispondenze nelle immagini da testare.

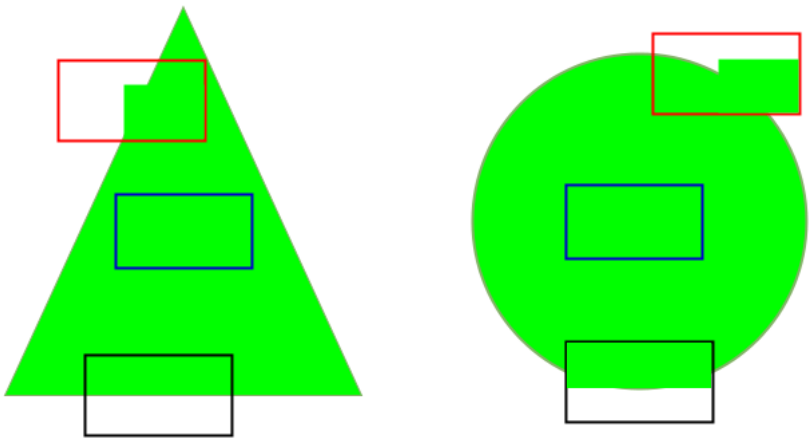

Figura 1.4: Esempio di utilizzo delle feature per il riconoscimento di triangoli

1.2 Algoritmi genetici

Un algoritmo genetico [13] è una tecnica di ottimizzazione che opera su una popolazione di possibili soluzioni. Le soluzioni subiscono lo stesso processo che avviene nella selezione naturale; ogni generazione è ottenuta applicando

una serie di funzioni di selezione, riproduzione e mutazione. L'obiettivo che questo tipo di algoritmo si pone è migliorare l'insieme delle soluzioni raffinando i risultati di generazione in generazione.
La selezione avviene tramite una funzione di fitness che consente di propagare solo gli elementi migliori per le generazioni successive.
La riproduzione è ottenuta attraverso il crossover che consiste nella combinazione di due elementi, scelti in modo casuale, della popolazione.
Infine vi è la mutazione che seleziona casualmente un elemento della popolazione e ne modifica uno o più dei suoi parametri. Rispettando l'andamento della selezione naturale, il crossover avviene con frequenza maggiore rispetto alla mutazione.
Questa tipologia di algoritmo segue uno schema preciso, riassunto della figura 1.5, diviso in quattro fasi: inizializzazione, selezione, applicazione degli operatori genetici e terminazione.
Nel caso dell'algoritmo Evolution COnstructed feature la popolazione iniziale viene generata in modo casuale e rappresenta un insieme di possibili soluzioni. Successivamente, ad ogni elemento della popolazione vengono applicate una serie di trasformazioni che portano all'estrazione delle feature per il riconoscimento degli oggetti. Tramite una funzione di fitness vengono selezionati, per le generazioni future, solo quegli elementi che più si avvicinano alla soluzione. Infine si applicano gli operatori genetici di crossover e mutazione in modo da creare nuovi elementi della popolazione. L'algoritmo termina al raggiungimento del numero massimo di generazioni. Un'analisi dettagliata dell'algoritmo è rimandata al capitolo successivo.

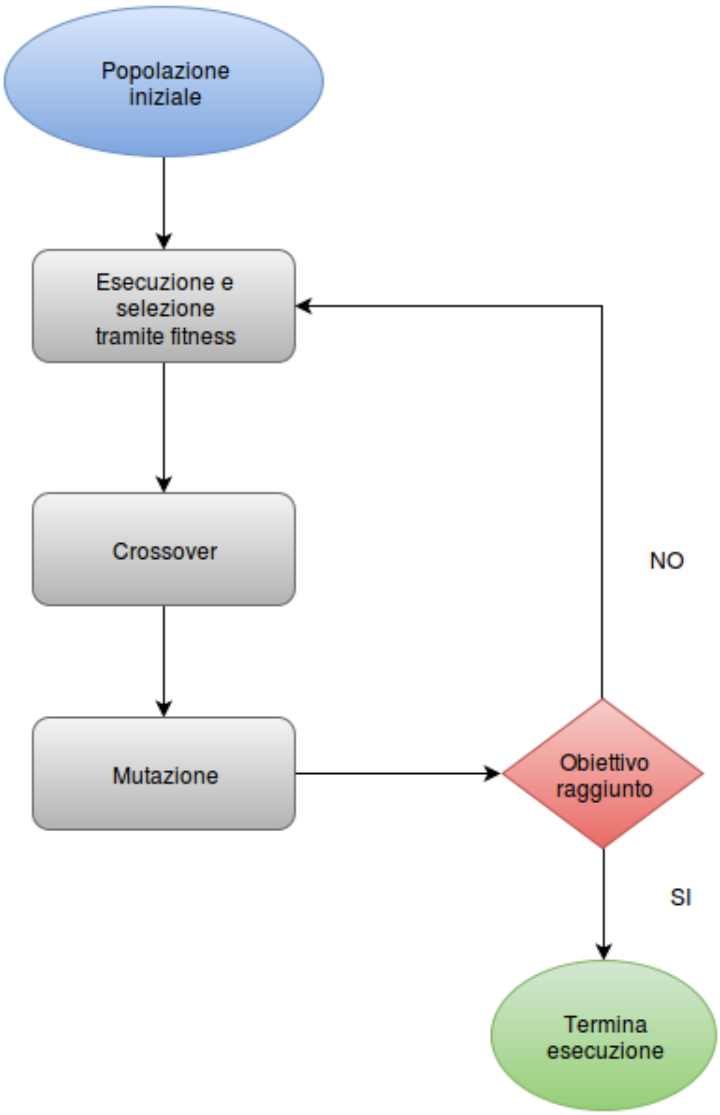

Figura 1.5: Schema di un algoritmo genetico

Capitolo 2

Eco Feature

Le Eco Feature sono il risultato di un algoritmo genetico che applica una serie di trasformazioni all'immagine di partenza in base a quanto specificato dall'equazione 2.1

$$\begin{aligned} &V_1 = T_1(I(x_1, y_1, x_2, y_2), \phi_1), \\ &... \\ &V_{n-1} = T_{n-1}(V_{n-2}, \phi_{n-1}), \\ &V = T_n(V_{n-1}, \phi_n) \end{aligned} \tag{2.1}$$

L'equazione 2.1 descrive in modo matematico il processo che porta all'estrazione delle feature da un immagine. Alla prima iterazione V_1 viene applicata una trasformazione T_1 ad una sotto-regione, scelta in modo casuale, $I(x_1, y_1, x_2, y_2)$ di un'immagine di partenza I; per i restanti n-1 passi la trasformazione T_i viene applicata all'output della trasformazione precedente T_{i-1}. Inoltre ogni iterazione V_i prende in input, oltre alla regione dell'immagine anche un vettore dati ϕ_i di dimensione variabile in base al tipo di operazione applicata al passo i. Dunque ϕ_i non è altro che un vettore di interi che rappresentano i valori di input della trasformazione T_i
La tabella 2.1 mostra l'insieme di tutte le possibili trasformazioni e il relativo numero di parametri di input.

Trasformazione	$\|\phi\|$		**Trasformazione**	$\|\phi\|$
Gabor Filter	6		Sobel Operator	4
Gradient	1		Diff. of Gaussians	2
Square Root	0		Morphological Erode	1
Gaussian Blur	1		Adaptive thresholding	3
Histograms	1		Hough Lines	2
Hough Circles	2		Fourier Transform	1
Normalize	3		Histogram Equalization	0
Convert	0		Laplacian Edge	1
Median Blur	1		Dinstance Transform	2
Integral Image	1		Morphological Dilate	1
Canny Edge	4		Harris Corner Strenght	3
Rank Transform	0		Census Transform	0
Resize	1		Pixel Statistics	2
Log	0			

Tabella 2.1: Trasformazioni e numero di parametri per ognuno di essi

Nella figura 2.1 viene rappresentata una schematizzazione di due Eco feature; ognuna di esse è composta da: una sotto-regione dell'immagine ed una serie di trasformazioni applicate in successione con relativi parametri.

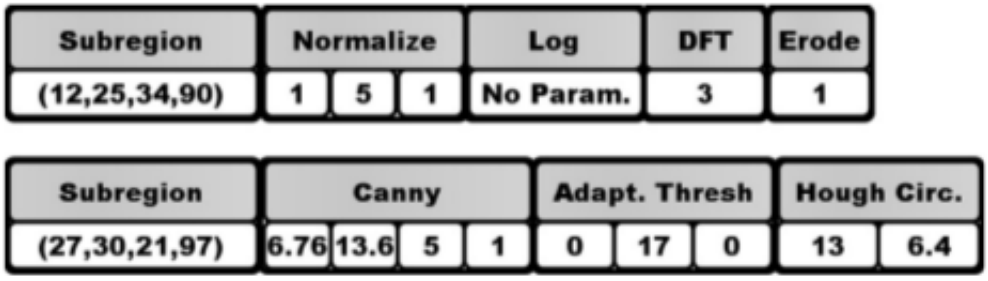

Figura 2.1: Esempio di due Eco Feature

Fonte: *A feature construction method for general object recognition* [11]

2.1 Costruire un Eco Feature

Come spiegato in precedenza ogni Eco Feature è costruita usando un algoritmo genetico standard [18]. Questa tipologia di algoritmo è molto efficiente nel risolvere problemi di ottimizzazione e di ricerca quando lo spazio dei dati di input è molto vasto.
Nel caso dell'Evolition COnstructed feature algorithm le Eco feature rappresentano gli elementi della popolazione ed i geni sono i parametri di input di ognuna di esse, quindi: la sotto-regione I(x_1, y_1, x_2, y_2), le trasformazioni $(T_1, T_2, ..., T_n)$ ed i relativi valori di input $(\phi_1, \phi_2, ..., \phi_n)$. Non esiste un numero fisso di geni per ogni elemento della popolazione, infatti il numero di trasformazioni può variare ad ogni passo della computazione così come il numero di parametri di ognuna di esse.
La fase iniziale dell'algoritmo consiste nel creare una popolazione casuale di Eco feature alle quali vengono assegnate un valore di fitness ed un perceptron. Un perceptron è un classificatore debole utilizzato per specificare quando un valore di input appartiene ad una classe oppure no; in pratica viene utilizzato per decidere se l'immagine contiene l'oggetto che stiamo cercando. Matematicamente un perceptron è un classificatore che mappa il vettore delle Eco feature V, ottenuto tramite l'equazione 2.1, in una classificazione binaria α, come illustrato dall'equazione 2.2. Il vettore di pesi W è il risultato dell'equazione 2.3, mentre il termine diagonale b è una costante ed è indipendente dai valori di input.

$$\alpha = \begin{cases} 1 & \rightarrow W \cdot V + b > 0 \\ 0 & \rightarrow altrimenti \end{cases} \tag{2.2}$$

Il vettore W è ottenuto dall'addestramento del perceptron tramite l'equazione 2.3, dove: V è il vettore delle Eco feature ottenuto applicando l'equazione 2.1; δ è l'errore calcolato come differenza tra l'output α del perceptron, calcolato

dall'equazione 2.2, e l'attuale classificazione dell'immagine; infine λ indica un valore di apprendimento.

$$\begin{aligned} &\delta = \beta - \alpha, \\ &W[i] = W[i] + \lambda \cdot \delta \cdot V[i] \end{aligned} \tag{2.3}$$

Il valore di fitness s è un intero che può assumere un valore nell'intervallo [0,1000]; viene calcolato tramite la formula 2.4 nella quale: p indica una penalità; t_p il numero di *true positive*; t_n i *true negative*, f_p i *false positive* ed f_n i *false negative*. t_p è il numero delle immagini classificate come contenenti l'oggetto da cercare che realmente contengono quell'oggetto. t_n è il numero delle immagini che non contengono l'oggetto ricercato classificate in modo corretto. f_p è il numero delle immagini classificate come contenenti l'oggetto da cercare ma che in realtà non lo contengono. Infine f_n è il numero delle immagini che contengono l'oggetto che stiamo cercando ma che sono state classificate come se non lo contenessero.

$$s = \frac{t_p \cdot 500}{f_n + t_p} + \frac{t_n \cdot 500}{p \cdot f_p + t_n} \tag{2.4}$$

Dopo aver calcolato il valore di fitness per tutti gli elementi, vengono eliminati dalla popolazione tutti coloro il cui valore di s risulti essere minore di un certo valore soglia. Successivamente vengono creati dei nuovi elementi della popolazione tramite il crossover e modificati altri attraverso la mutazione; entrambe queste operazioni sono riassunte nella figura 2.2.
Nell'esempio: il crossover estrae dal primo elemento la sotto-regione e la prima trasformazione e li unisce con le ultime due trasformazioni del secondo elemento; nel caso della mutazione, invece, non viene fatto altro che modificare il valore di un parametro della trasformazione *Erode*

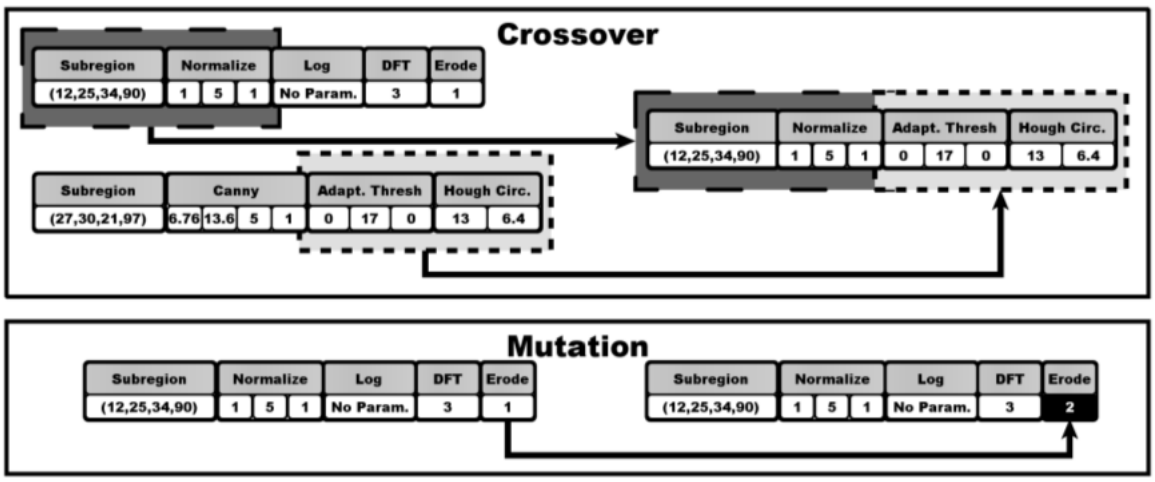

Figura 2.2: Esempio di Crossover e Mutazione
Fonte: *A feature construction method for general object recognition* [11]

L'immagine in figura 2.3 mostra come la selezione della sotto-regione (x_1, y_1, x_2, y_2) dell'immagine I, alla quale applicare le trasformazioni possono variare non solo nella posizione ma anche nella dimensione. Questo perché la scelta viene effettuata dall'algoritmo senza fare nessuna assunzione sulla validità di un'area rispetto ad un'altra, in questo modo l'importanza di una sotto-regione viene definita non da cosa rappresenta ma dalla sua capacità di classificare l'oggetto dopo aver subito le trasformazioni $(T_1, T_1, ..., T_n)$.

L'algoritmo 2.4 mostra lo pseudo codice relativo alla fase di estrazione delle feature dalle immagini. Nella prima fase vengono generate le nuove creature che sono inizializzate con una sotto-regione casuale (x_1, y_1, x_2, y_2) ed una serie di trasformazioni $(T_1, T_2, ..., T_n)$ con relativi parametri di input $(\phi_1, \phi_2, ..., \phi_n)$. Successivamente ogni creatura applica le trasformazioni alle immagini contenenti l'oggetto da riconoscere, seguendo quanto specificato dall'equazione 2.1; addestra i perceptron attraverso le equazioni 2.2 e 2.3; applica le trasformazioni alle immagini che non contengono l'oggetto e addestra i perceptron, allo stesso modo di quanto fatto nel passaggio precedente; calcola il valore di fitness tramite l'equazione 2.4 ed elimina quelle creature con un risultato minore di un dato valore soglia; infine applica gli operatori

Figura 2.3: Selezione di sotto-regioni dell'immagine
Fonte: *A feature construction method for general object recognition* [11]

genetici di crossover e mutazione per creare dei nuovi elementi della popolazione. Questo ciclo viene ripetuto fino al raggiungimento del numero di generazioni.
L'addestramento dei perceptron permette di avere dei classificatori in grado di riconoscere quando un immagine contiene oppure no l'oggetto che stiamo cercando. Durante questa fase se si utilizzano tante immagini contenenti l'oggetto e poche senza, si rischia di creare un sistema sbilanciato che generi molti falsi positivi, cioè avremo un modello capace di riconoscere l'oggetto, ma che non è in grado di capire quando quell'oggetto non è presente e quindi classifica tutto come positivo.

2.2 AdaBoost

Il boosting [4] è una famiglia di algoritmi basati su un approccio machine learning che partendo da regole deboli e poco accurate genera, attraverso un processo di addestramento, dei pattern di riconoscimento molto precisi.
L'AdaBoost di *Freund* e *Schapire* [16] è una delle implementazioni più utilizzate e studiate. Viene utilizzato in combinazione con altri algoritmi di

```
for dimensione della popolazione do
    genera nuove creature
    seleziona sotto-regione (x_1, y_1, x_2, y_2)
    seleziona trasformazioni (T_1(φ_1), ..., T_n(φ_n))
end for
for numero di generazioni do
    for creature do
        for immagini con oggetto do
            applica trasformazioni alle immagini, eq 2.1
            addestra i perceptron, eq 2.2 e 2.3
        end for
        for immagini senza oggetto do
            applica trasformazioni alle immagini, eq 2.1
            addestra perceptron, eq 2.2 e 2.3
        end for
        calcola valore di fitness, eq 2.4
        eliminare creature con fitness < valore soglia
    end for
    crea nuove creature tramite crossover
    applica mutazione
end for
```

Figura 2.4: Pseudo codice dell'algoritmo per l'estrazione delle feature

apprendimento, ed il lavoro che svolge è quello di effettuare una somma ponderata dei pattern di output ottenuti da tutti i classificatori in modo tale che partendo da una serie di classificatori deboli si riesca a definire un pattern molto più preciso. A differenza di altri algoritmi, dove bisogna specificare diversi parametri ogni volta che si cambia contesto di riconoscimento, nell'AdaBoost i pattern utilizzati vengono appresi durante il procedimento di addestramento; proprio per questo motivo viene spesso definito come il miglior classificatore *out of the box*.
L'AdaBoost in fase di addestramento utilizza un limite massimo di perceptron che può variare in base alle diverse esigenze del problema preso in analisi. Per ogni perceptron, calcolato durante la fase di estrazione delle feature attraverso l'equazione 2.2, viene calcolato l'indice di affidabilità tramite l'equazione 2.5. Il valore δ_w rappresenta l'errore del perceptron calcolato tramite l'equazione 2.3.

$$\rho = \frac{1}{2} \cdot ln \frac{1 - \delta_w}{\delta_w} \tag{2.5}$$

Durante la fase di classificazione ogni feature, con relativo perceptron, opera su una sotto-regione dell'immagine. Queste aree possono essere anche sovrapposte e possono variare di dimensione a seguito delle trasformazioni subite. Dopo che ogni perceptron ha verificato se l'immagine contiene oppure no l'oggetto che stiamo cercando, attraverso l'equazione 2.2, vengono unificati gli output di tutti i perceptron utilizzati, come indicato nell'equazione 2.6; dove X indica il numero massimo di perceptron utilizzati durante la fase di addestramento; ρ_x che viene ottenuto dall'equazione 2.5 è il coefficiente di affidabilità del perceptron x; α_x calcolata tramite l'equazione 2.2 indica l'output del perceptron x; infine τ rappresenta un valore soglia. Il risultato ottenuto c rappresenta la classificazione finale.

$$c = \begin{cases} 1 & \rightarrow \sum_{x=1}^{X} \rho_x \cdot \alpha_x > \tau \\ 0 & \rightarrow altrimenti \end{cases} \tag{2.6}$$

La figura 2.5 mostra uno schema del funzionamento dell'AdaBoost. Nell'esempio si parte da un immagine I da cui vengono estratte tre sotto-regioni. Ad ogni sotto-regione vengono applicate una serie di trasformazioni che portano alla generazione del vettore delle feature V, come indicato dall'equazione 2.1. Tramite i perceptron ogni vettore di feature V viene classificato come contenente l'oggetto oppure in base a quanto specificato dall'equazione 2.2, infine questi tre risultati vengono unificati, tramite l'equazione 2.6 e viene deciso se l'immagine contiene oppure no l'oggetto.

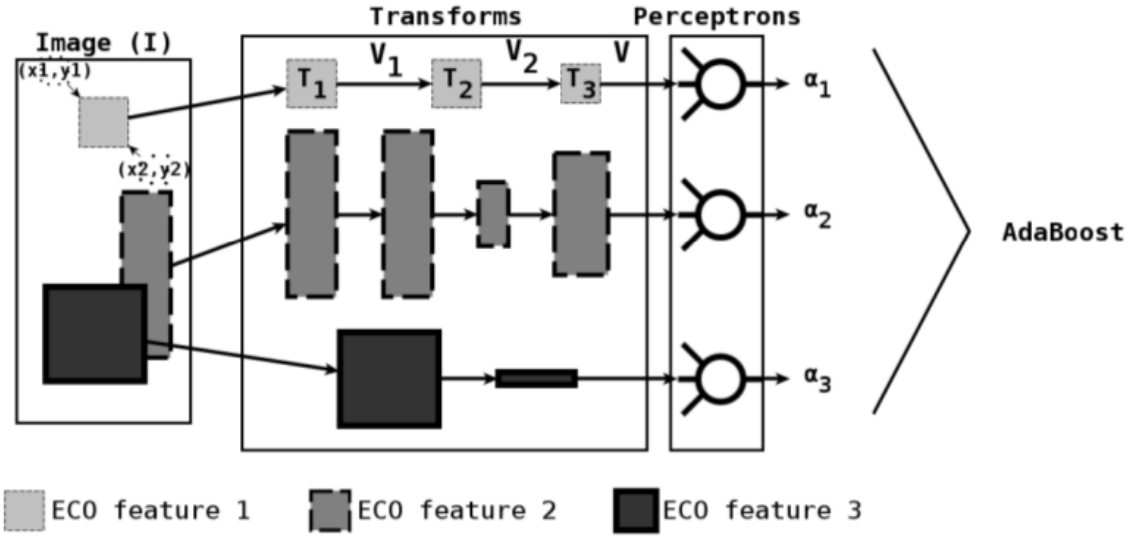

Figura 2.5: Esempio di applicazione dell'AdaBoost
Fonte: *A feature construction method for general object recognition*[11]

Capitolo 3

Programmazione Parallela

La programmazione parallela consiste nell'implementare algoritmi che permettano di sfruttare tutti i processori presenti su un determinato dispositivo, o su un insieme di dispositivi, in modo da rendere la computazione più veloce. Il problema di partenza viene quindi suddiviso in sotto-problemi indipendenti cosicché possano eseguire in modo autonomo sui diversi processori disponibili.
La figura 3.1 mostra concettualmente il confronto dell'esecuzione tra un applicazione sequenziale ed una parallela. Dopo la fase iniziale l'esecuzione della porzione A viene suddivisa tra quattro processori; in seguito vi è una fase B sequenziale ed un ultima fase C eseguita ancora in parallelo. L'esempio mostra anche come un algoritmo sequenziale non sempre sia completamente parallelizzabile: la fase B viene eseguita in sequenziale in entrambi i casi.

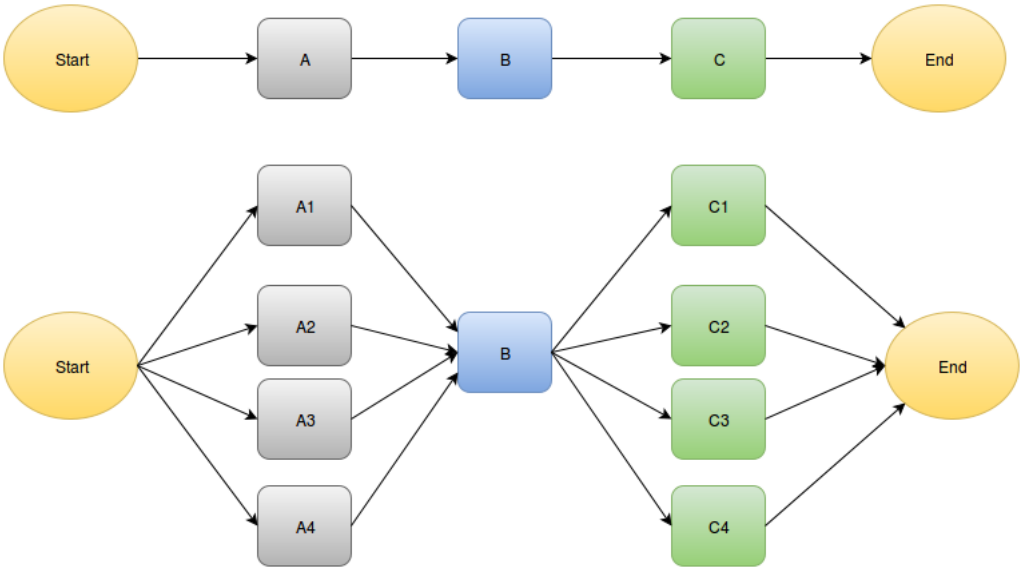

Figura 3.1: Confronto tra esecuzione sequenziale ed esecuzione parallela

Il parallelismo può essere applicato in diversi contesti, come ad esempio: un singolo computer multicore, un insieme di dispositivi collegati tra di loro, un particolare hardware specializzato oppure una combinazione di questi.

3.1 Condizioni di Bernstein

Nel progettare un algoritmo parallelo è importante valutare quali sono i sottoinsiemi indipendenti e quindi parallelizzabili. Le condizioni di Bernstein [2] descrivono quando due segmenti di un programma P_i e P_j sono indipendenti tra loro e quindi possono essere eseguiti in parallelo. Dati I_i, I_j e O_i, O_j, rispettivamente input ed output di P_i e P_j, questi sono indipendenti quando valgono le seguenti condizioni:

$$\begin{aligned} I_j \cap O_i &= \emptyset \\ I_i \cap O_j &= \emptyset \\ O_i \cap O_j &= \emptyset \end{aligned} \tag{3.1}$$

La prima condizione rappresenta una dipendenza a posteriori in cui l'output del primo segmento genera l'input del segmento successivo; la seconda è una

dipendenza a priori, quindi l'input del primo segmento dipende dall'output del secondo segmento; infine la terza è una condizione di output, in cui sia P_i che P_j salvano il risultato dell'esecuzione sulla stessa porzione di memoria, generando conflitti.
Nella programmazione parallela è molto importante gestire le interazioni tra i vari processori, in modo tale che questi possano organizzarsi sulla suddivisione dei compiti da eseguire per risolvere il problema principale e garantire che le condizioni di Bernstein vengano sempre rispettate.

3.2 Interazione tra i processori

L'interazione tra i vari processori può essere gestita in diversi modi, in base alle diverse architetture le comunicazioni avvengono tramite la memoria condivisa [7] oppure il message passing [6]

Memoria condivisa I processori accedono allo stesso spazio di memoria, ma possono disporre anche di una memoria cahce locale, come mostrato in figura 3.2. L'accesso viene effettuato in modo asincrono, così da evitare problemi di incoerenza che possano generare dei valori di output errati; questo problema è noto come race condition: il valore di output dipende dalla sequenza delle operazioni eseguite, a diverse sequenze corrispondono differenti risultati. L'accesso asincrono alla memoria può essere gestito in diversi modi, ad esempio: tramite l'utilizzo di un token che transita tra i vari processori e soltanto quando se ne è in possesso si può accedere alla memoria; attraverso semafori, lock oppure monitor.

Message passing Nel caso del message passing ogni processore dispone di una memoria propria e comunica con gli altri attraverso lo scambio di messaggi. La comunicazione può essere sincrona oppure asincrona: nel primo caso il mittente deve attendere fin quando il destinatario non è pronto per ricevere il messaggio, mentre nel secondo caso non vi è bisogno di attesa da

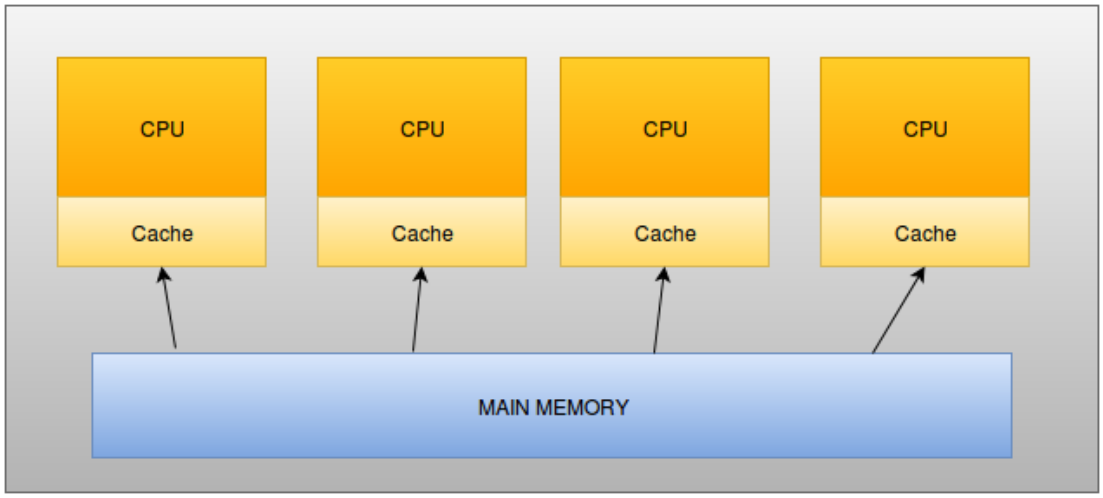

Figura 3.2: Architettura multicore con memoria condivisa

parte del mittente. La figura 3.3 mostra un esempio comunicazione attraverso il message passing tra i processi A e B, che operano su una variabile x.

3.3 Suddivisione del problema

La suddivisione del problema consiste nel creare dei sotto-problemi indipendenti tra loro, in modo tale che ogni processore possa eseguire il proprio lavoro in modo completamente autonomo. Questa suddivisione può avvenire a livello dei task oppure dei dati; la scelta del primo o del secondo caso è direttamente correlata al problema da risolvere.

Task Il parallelismo sui task consiste nella suddivisione del problema di partenza in operazioni indipendenti tra loro ed eseguirle in parallelo sui vari processori.

Dati Nel caso del parallelismo sui dati i processori eseguono le stesse funzioni, ognuno su un diverso sottoinsieme di dati.

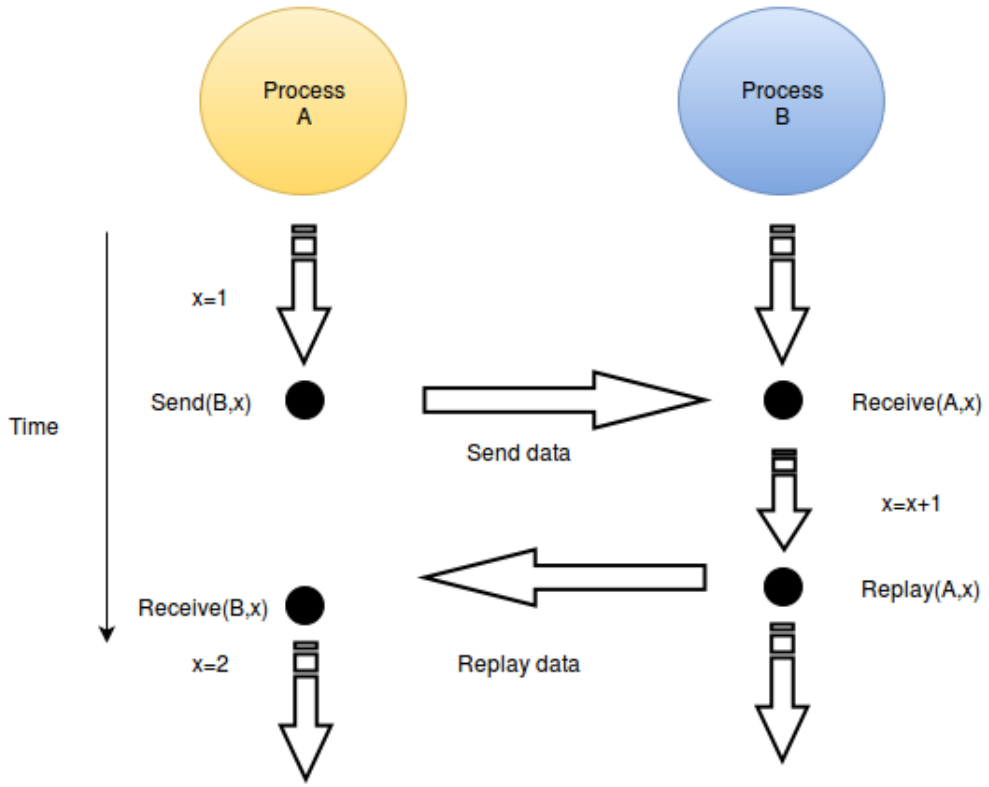

Figura 3.3: Esempio di comunicazione tramite message passing

3.4 Parallelizzazione implicita

La parallelizzazione implicita è una caratteristica di alcuni linguaggi di programmazione che permettono al compilatore oppure all'interprete di parallelizzare automaticamente il codice.
Un approccio simile alla programmazione parallela consente allo sviluppatore di non preoccuparsi di tutti i problemi visti precedentemente, come sincronizzazione tra processi e suddivisione del problema. Il problema però è che il programmatore non ha il pieno controllo del parallelismo, quindi si corre il rischio di avere algoritmi paralleli meno efficienti delle rispettive versioni sequenziali. Un altro aspetto negativo è la fase di debugging, che in questi casi può rivelarsi molto difficile proprio per via di questo livello di programmazione nascosto all'utente. La parallelizzazione implicita si adatta male ai linguaggi di programmazione imperativi, come python, con cui si ottengono dei risultati positivi solo in pochi casi; ad esempio su applicazioni che effettuano operazioni costose su vettori, o comunque strutture dati che potrebbero essere facilmente suddivise tra i vari processori. Nel caso di linguaggi di programmazione dichiarativi o logici, come il prolog, questo tipo di parallelismo si adatta meglio in quanto questi linguaggi sono caratterizzati da un alto livello di astrazione. Alcune caratteristiche che favoriscono questi linguaggi nella parallelizzazione implicita sono: variabili utilizzate come entità matematiche, il cui valore non può cambiare durante l'esecuzione; semantica basata su formule non deterministiche, come le clausole di selezione nei linguaggi logici o l'operatore apply nei linguaggi funzionali.

Capitolo 4

Implementazione

In questa tesi l'algoritmo Eco Feature viene implementato in due versioni: una sequenziale ed una parallela, in modo da poter confrontare i tempi di esecuzione e valutare gli effettivi guadagni ottenuti grazie alla parallelizzazione.
Entrambe le versioni sono state implementate tramite l'utilizzo del linguaggio python, sfruttando in particolare tre librerie: OpenCV nella versione 2.3.1 (2011) utilizzata dagli autori della versione descritta nell'articolo [12] per la versione sequenziale ed una versione aggiornata, la 2.4.11 (2015), per la versione parallela; le altre due importanti librerie utilizzate sono numpy [19] e multiprocessing [14]
OpenCV, che approfondiremo nella sezione successiva, nel caso del python non è altro che un wrapper della versione implementata per C++. Numpy è una libreria che implementa funzioni ottimizzate per operare su vettori multidimensionali. La libreria multiprocessing viene utilizzata nella versione parallela dell'algoritmo, questa libreria consente di creare un insieme di processi in base al numero di processori a disposizione così da distribuire il carico computazionale tra essi.

4.1 OpenCV

OpenCV è una libreria open source composta da più di 2500 algoritmi ottimizzati per la computer vision ed il machine learning [5]. Questi algoritmi possono essere utilizzati per il riconoscimento dei volti, l'identificazione degli oggetti, la tracciatura di movimento di una videocamera o degli oggetti stessi, l'estrazione di modelli 3D oppure la generazione di intere scene partendo da una serie di immagini.
La libreria è disponibile per i linguaggi c++, c, python, java e MATLAB e supporta i principali sistemi operativi, come: Linux, Android, Microsoft e Mac OS.
Nelle versioni più recenti OpenCV implementa un parziale supporto verso le GPU attraverso l'utilizzo di architetture di sviluppo come Cuda [8] ed OpenCL [17].
Nell'algoritmo Eco feature la libreria OpenCV viene utilizzata per applicare le trasformazioni elencate nella tabella 2.1, in modo tale da sfruttare anche i core messi a disposizione dalla GPU ed aumentare il livello di parallelismo.
OpenCV gestisce la memoria in modo automatico, dunque tutti gli spazi di memoria utilizzati per allocare le strutture dati necessarie all'esecuzione delle funzioni vengono de-allocati quando non ritenuti più necessari.
OpenCV lavora sui pixel delle immagini, codificati tramite una rappresentazione a 8 oppure 16 bit per canale. Alcune operazioni come la conversione dello spazio dei colori, la regolazione della luminosità o del contrasto e interpolazioni complesse possono produrre in output valori al di fuori di tale intervallo di tale codifica. Per risolvere questo problema si utilizza il metodo della saturazione aritmetica.
Ad esempio per salvare il risultato k, di un'operazione su un immagine ad 8 bit si cerca il valore più vicino nel range [0,255] tramite l'equazione 4.1.

$$I(x, y) = min(max(round(k), 0), 255) \tag{4.1}$$

4.2 Implementazione sequenziale

Il *main* dell'Eco feature è rappresentato dal file *eco_feature.py* dove viene creata una nuova istanza della popolazione tramite la classe Genetic

```
gen=genetic.Genetic(population,generation,
        fitness_penality,fitness_th,pop_list)
```

Successivamente vengono invocate le funzioni definite nella classe Genetic che permettono di caricare le immagini, inizializzare gli elementi della popolazione, eseguire la fase di estrazione delle feature ed il calcolo finale dei risultati.

```
gen.load_img(dir_train,dir_test)
gen.initialize()
gen.run()
gen.adaboost()
```

4.2.1 Classe Genetic

La classe Genetic rappresenta l'insieme di tutti gli elementi che fanno parte della popolazione; questa classe è la più importante poiché gestisce tutte le fasi dell'algoritmo.
I parametri sono:

- population: dimensione della popolazione;
- generation: numero di generazioni;
- fitness_penality: valore di penalità utilizzato nel calcolo della fitness;
- fitness_th: valore soglia della fitness;
- pop_list: lista degli elementi della popolazione.

Caricamento immagini

Il caricamento delle immagini avviene nelle funzione load_img tramite l'utilizzo della funzione imread della libreria OpenCV che supporta tutti i più comuni formati di immagini, come ad esempio: bmp, jpg, jpeg, png.

```
train_images=os.listdir(train_dir)

for x in train_images:
    train_img.append(cv2.imread(x))

test_images=os.listdir(test_dir)

for x in test_images:
    test_img.append(cv2.imread(x))
```

Le immagini vengono suddivise in due liste: la prima che rappresenta l'insieme delle immagini contenenti l'oggetto per il quale si vuole addestrare il riconoscitore, la seconda contenente altre immagini utilizzate per valutare le differenze tra l'oggetto da riconoscere e gli altri oggetti.

Inizializzazione

La fase successiva è rappresentata dall'inizializzazione della popolazione.

```
for i in range(0,population):
  new_creature=creature.Creature([],[],[],[],[],
                                 0,0,1,1,1,1)
  transfor_count=random(x)
  transforms_list=[]

  for j in range (0,transfor_count):
    parameters=[]
    new_transform=transform.Transform(random(x),parameters,[])

    for k in range (new_transform.param):
      parameters.append(random(x))
      new_transform.param=parameters

    transforms.append(new_transform)
```

In base al valore di population viene creato un insieme di elementi della popolazione attraverso la classe Creature, con parametri iniziali nulli. Per ogni elemento viene generato in modo casuale un set di trasformazioni, con relativi parametri, che successivamente verranno applicate alle immagini per le estrazioni delle feature.

```
      new_creature.transformation=transforms
      new_creature.initialize()
      pop.append(new_creature)
```

Dopo aver generato una lista di trasformazioni casuali, questa viene assegnata alla creature appena creata che viene inizializzata attraverso la funzione initialize della classe Creature ed infine aggiunta alla lista degli elementi della popolazione.

Esecuzione

La fase di esecuzione è gestita dalla funzione run impostata con un doppio ciclo in base al numero di generazioni ed al numero degli elementi per ogni generazione.

```
for i in range(generation):

    for j in range(len(pop)):

        for k in range(len(train_img)):
            pop[j].perform_transforms(train_img[k])
            pop[j].c.append(1)
            pop[j].train_perceptron()

        for k in range(len(test_img)):
            pop[j].perform_transforms(test_img[k])
            pop[j].c.append(0)
            pop[j].output_perceptron()

        if(pop[j].fitness<fitness_th):
            pop.remove(j)

        if(random(x) < mutation_rate):
            pop[j].mutate()

    crossover(pop[random(x)],pop[random(y)])
```

L'esecuzione si divide in due fasi. In un primo momento si applicano le trasformazioni, tramite la funzione definita nella classe Creature perform_transforms, all'insieme delle immagini contenenti l'oggetto che l'algoritmo deve riconosce. Successivamente le trasformazioni vengono applicate ad un insieme di immagini che contengono un altro oggetto. La lista c è un classificatore che indica quando le feature estratte rappresentano l'oggetto a cui siamo interessati e quando no, indicandoli rispettivamente con i valori 1 e 0.

Dopo questa fase, per ogni elemento della popolazione, viene calcolato il valore di fitness tramite l'equazione 2.4; se il risultato è inferiore ad un certo valore soglia, indicato da fitness_th, allora quell'elemento viene rimosso dalla popolazione.
Gli ultimi compiti svolti dalla funzione run sono: la mutazione ed il crossover. Il crossover consiste nel selezionare due elementi a caso dalla popolazione ed unirli per crearne un terzo. La mutazione, che viene applicata in base ad un certo valore casuale, non fa altro che modificare un parametro di una delle trasformazioni.

Adaboost

L'ultima funzione implementata in questa classe è l'adaboost, il cui compito è di aggiornare i valori di ρ, α e τ in base alle feature estratte e successivamente classificare le immagini applicando l'equazione 2.6.

```
for i in range (feature.shape[1]):
    in_data=transpose(numpy.atleast_2d(feature[:,i]))
    reg.fit(in_data,c)
    tau=-1*reg.intercept_/reg.coef_
    ce=reg.predict(indata)
    Indicator=abs(ce-c)
    rho=dot(Indicator,W)

    if rho <= bestVal:
        bestVal=rho
        bestFeat=i

    J=rho
    alpha=1.0*J/(1-J)
    IndicatorList.append(Indicator)
    alphaList.append(alpha)
    tauList.append(tau)
```

Per ogni feature viene calcolata la fit, successivamente si calcolano i valori di predizione tramite la funzione predict della libreria sklearn [10] e viene calcolata la differenza in valore assoluto tra il vettore ottenuto e quello di classificazione generato precedentemente dalla funzione run. Infine vengono aggiornati i valori di ρ, α e τ ed aggiunti alle rispettive liste, tenendo traccia, ad ogni iterazione, dei migliori risultati ottenuti.

```
I=IndicatorList[bestFeat]
alpha=log(1.0/(alphaList[bestFeat]))
tau=tauList[bestFeat]
bestClassifier[]= array([bestFeat,tau,alpha])

for r in range(c.shape[0]):
    W[r]=W[r]*power(alpha,1-I[r])
    Wnorm=sum(W)
    W=W/Wnorm
```

Dopo aver aggiornato i valori di ρ, α e τ per ogni feature, vengono estratti i migliori risultati ottenuti e vengono aggiornati i valori dei pesi.
Quindi si può procedere alla classificazione di nuove immagini applicando i risultati ottenuti.

```
for u in range(test_data.shape[0]):
    data=test_data[u,:]
    sum=0

    if test_data[u,bestClassFeat[v]] > bestClassTh[v]:
                                sum=sum+bestClassAl[v]
```

Il risultato ottenuto viene confrontato con il valore di τ e classificato.

4.2.2 Classe Creature

La classe Creature rappresenta ogni elemento della popolazione; i parametri più importanti di questa classe sono:

- transformation: lista delle trasformazioni e relativi parametri;
- feature_list: lista delle feature estratte;
- c: classificatore che indica quali sono le feature estratte da immagini contenenti l'oggetto da riconoscere;
- perc: lista dei perceptron.

Applicazione delle trasformazioni

La funzione perform_transform si occupa di applicare le trasformazioni all'immagine data in input invocando la funzione apply_transform della classe Transform.

```
image=input_image[random(x1,x2,y1,y2)]

for i in range(len(transformation)):
    if(check_transformation_parameter_number):
        print "Error: parameter number"
    else:
        image=transformation[i].apply_transform(image)

feature_list.append(img)
```

Dall'immagine di input viene estratta una sotto-parte scelta in modo casuale ed a questa vengono applicate tutte le trasformazioni specificate da quell'elemento della popolazione. Per tutte le successive n-1 iterazioni l'input al passo i-esimo è l'output dell'iterazione (i-1)-esima. Dopo aver effettuato tutte le trasformazioni le feature estratte vengono salvate nella feature_list dell'elemento.

Aggiornamento dei perceptron

La train_perceptron e l'output_perceptron sono due funzioni che si occupano rispettivamente di implementare le equazioni 2.3 e 2.2 per ogni elemento della popolazione.

Fitness e mutazione

Le ultime due funzioni della classe Creature si occupano di calcolare il valore di fitness, in base all'equazione 2.4, e di applicare la mutazione, che consiste nella modifica di un parametro di una delle trasformazioni scelta in modo casuale.

4.2.3 Classe Transform

La gestione delle trasformazioni per ogni elemento della popolazione avviene tramite la classe Transform, i cui parametri sono:

- type: specifica il tipo di trasformazione;
- param: vettore dei parametri;
- trasform_img: immagine, o porzione di essa, alla quale applicare la trasformazione.

Inoltre vengono utilizzati due vettori, riportati di seguito, che indicano rispettivamente il tipo di trasformazioni ed il relativo numero di parametri.

```
transform_type=['NIL','GABOR_FILTER','MORPH_ERODE',...]
transform_value=[0,6,2,...]
```

Il compito di questa classe è di richiamare le giuste funzioni della libreria OpenCV per applicare le trasformazioni, come mostrato nell'esempio 4.2.3.

```
img=float32(img)
img/=255.0

if(p[1]<0): p[1]=0
if(p[1]>21): p[1]=21

t_type=int(p[0])
size=2*(int(p[1]))+1
point=int(p[1])

cv2.erode(tmp_img,cv2.getStructuringElement(t_type,
(size,size),(point,point)))
```

La prima operazione effettuata è una conversione dell'immagine in valori float a 32 bit, in un range che va da 0 a 255.
In seguito viene verificato se il valore dei parametri di input della trasformazione ricadono all'interno del range dei valori ammissibili, solo successivamente viene applicata la trasformazione all'immagine. L'output generato viene utilizzato come input per la trasformazione successiva.

4.2.4 Classe Perceptron

La classe dei Perceptron si occupa semplicemente di definire i valori utilizzati per il calcolo delle equazioni 2.2 e 2.3 tramite le funzioni train_perceptron e l'output_perceptron della classe Creature.

4.3 Implementazione Parallela

L'implementazione parallela segue la stessa struttura della versione sequenziale. La parallelizzazione avviene in parte sfruttando una versione aggiornata (2.4.11 - 2014) della libreria OpenCV rispetto a quella utilizzata precedentemente. Questa nuova versione della libreria implementa tutte le funzioni di trasformazioni delle immagini sfruttando i core della GPU, permettendo in

questo modo di parallelizzare quella parte dell'algoritmo relativa alla classe Transformation. Gli stessi autori dell'articolo [12] spiegano della possibilità dell'utilizzo della nuova versione di questa libreria per sfruttare le potenzialità della GPU.
Un ulteriore livello di parallelismo viene ottenuto sfruttando le CPU disponibili attraverso l'utilizzo della libreria multiprocessing del linguaggio python. Questa libreria consente di creare un insieme di processi che eseguono la stessa funzione, ognuno su di un sottoinsieme diverso di dati, dunque effettuando un parallelismo sui dati. Al termine dell'esecuzione ogni processore chiama una funzione di callback utilizzata per unificare i risultati parziali in un unico risultato finale.
Le modifiche al codice sono avvenute sostanzialmente nelle funzioni della classe Genetic dove si ha il maggior carico di lavoro. Inoltre vi sono sezioni del codice in cui non vi è la possibilità di parallelizzare poiché ogni iterazione utilizza come input l'output dell'iterazione precedente, come ad esempio nella funzione perform_transform della classe Creature, dove ogni trasformazione all'interno del ciclo viene applicata al risultato della trasformazione del passo precedente.

Inizializzazione

La funzione initialize nel caso sequenziale esegue un ciclo in base al numero degli elementi della popolazione; nella versione parallela questo ciclo viene diviso in relazione al numero dei processori disponibili ed eseguito tramite la funzione initialize_parallel. Gli indici di accesso al vettore contenente gli elementi della popolazione vengono calcolati come riportato dal codice 4.3

```
inf=(id_core * population) / core_number
sup=((id_core + 1) * population) / core_number
```

Successivamente ogni processore invoca la funzione parallela tramite la apply_async, utilizzando i propri indici di accesso alla memoria ed utilizzando come callback la funzione in_result

```
apply_async(initialize_parallel, args = (inf,sup, ),
callback = in_result)
```

La funzione initialize_parallel è identica alla versione sequenziale, con l'unica differenza che i risultati ottenuti da ogni core vengono memorizzati in una nuova struttura dati locale. Al termine dell'esecuzione ogni processore invoca la callback in_result che si occupa di raccogliere gli output generati da tutti i core ed unirli in un unico risultato finale.

Esecuzione

La funzione run nella versione sequenziale è composta da un doppio ciclo; il primo indica il numero delle generazioni, mentre il secondo rappresenta il numero degli elementi della popolazione. Il primo ciclo è impossibile da parallelizzare in quanto c'è una dipendenza tra le iterazioni; infatti ogni generazione è composta dalla popolazione generata, tramite una serie di operazioni, della generazione precedente. Il secondo ciclo invece viene parallelizzato seguendo lo stesso principio utilizzato nel caso dell'inizializzazione.

```
inf=(x*self.population)/core
sup=((x+1)*self.population)/core
apply_async(run_parallel, args =
(pop[inf:sup],fitness_penality,fitness_th, ),
callback = run_result)
```

Ogni core, in base al proprio indice, calcola i valori di inf e sup di accesso al vettore condiviso contenente gli elementi della popolazione; esegue la funzione run_parallel indicando come parametri: il sottoinsieme del vettore, i valori di fitness_peality e fitness_th e la funzione di callback result.
La funzione run_parallel, per ogni core, applica le trasformazioni alle immagini invocando la funzione perform_transform , aggiorna i valori dei perceptron ed infine calcola il valore di fitness, lo confronta con fitness_th e decide se tenere o scartare quell'elemento. Il risultato viene memorizzato in una struttura dati locale.

La funzione di callback run_result unisce i risultati parziali ottenuti da tutti i core, prendendo come input l'output della funzione run_parallel e salvando il risultato in una struttura dati globale.

AdaBoost

Infine seguendo lo stesso principio viene parallelizzata anche la funzione adaboost. Il ciclo effettuato in base al numero di classificatori deboli utilizzato dall'algoritmo per estrarre un unico classificatore forte viene diviso in relazione al numero dei core disponibili, ognuno dei quali effettua il calcolo dell'equazione 4.2:

$$\rho_x \cdot \alpha_x \tag{4.2}$$

La funzione di callback ada_result somma i risultati parziali ottenuti dai vari core.
In altri punti dell'algoritmo, come ad esempio il ciclo sulle generazione oppure quello sulle trasformazioni applicate alle immagini, non vi è la possibilità di parallelizzazione perché l'esecuzione dell'i-esima iterazione utilizza come input il risultato dell'iterazione (i-1)-esima.

Capitolo 5

Risultati

Lo scopo di questo lavoro è migliorare il tempo di esecuzione dell'algoritmo Eco feature, dunque l'analisi dei risultati si focalizza soprattutto sulla differenza dei tempi tra l'implementazione sequenziale e quella parallela.
Ovviamente è stata effettuata anche una valutazione delle prestazioni dell'algoritmo nel riconoscimento degli oggetti.

5.1 Precision e recall

Gli algoritmi di Object recognition valutano le proprie prestazioni attraverso il calcolo dei valori di Precision e Recall tramite le equazioni 5.1 e 5.2.

$$precision = \frac{t_p}{t_p + f_p} \tag{5.1}$$

$$recall = \frac{t_p}{t_p + f_n} \tag{5.2}$$

t_p rappresenta il numero dei *true positive*, f_p i *false positive* ed infine f_n i *false negative*. Con t_p si indica il numero delle immagini contenente l'oggetto che stiamo cercando classificate correttamente; f_p indica il numero delle immagini che non contengono l'oggetto ma che sono state classificate come se lo contenessero; f_n è il numero delle immagini che contengono l'oggetto ma che sono state classificate come se non lo contenessero; infine t_n indica il numero

delle immagini che non contengono l'oggetto classificate correttamente. La figura 5.1 mostra la rappresentazione sotto forma di insiemi di questi valori.

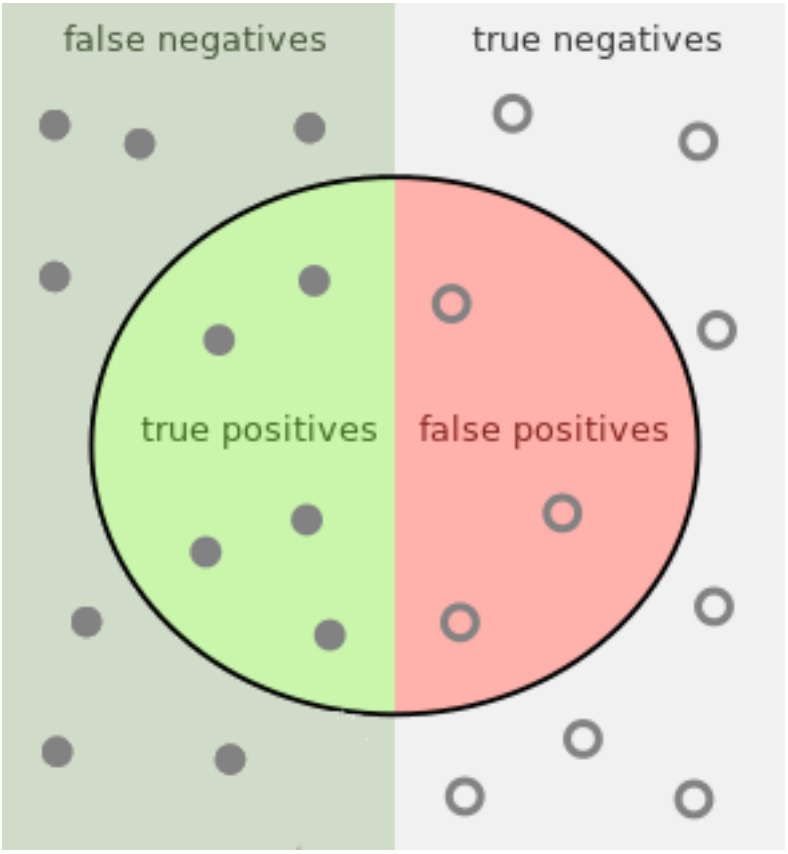

Figura 5.1: Rappresentazione dei valori t_p, f_p, f_n e t_n
Fonte: *Wikipedia*,
`https://en.wikipedia.org/wiki/Precision_and_recall`

Tramite la precision si calcola la qualità dei risultati ottenuti, cioè viene verificato la precisione dell'algoritmo nel riconoscere l'oggetto, il valore calcolato dall'equazione 5.1 è un valore compreso nel range [0,1], più questo risultato si avvicina ad 1 e più il sistema è considerato preciso.
La recall rappresenta invece una stima di completezza del sistema; anche in questo caso il risultato dell'equazione 5.2 è un valore compreso nel range [0,1], dove 1 indica la precisione massima per il sistema.

Valutazione precision e recall

I parametri utilizzati per la valutazione della precision e della recall dell'Eco feature sono i seguenti:

- popolazione: 40;
- generazioni: 100;
- immagini di addestramento con oggetto: 30;
- immagini di addestramento senza oggetto: 30;
- immagini per ogni test: 10.

Il grafico 5.2 mostra la media dei risultati di precision e recall ottenuti dopo aver eseguito una serie di 20 test.

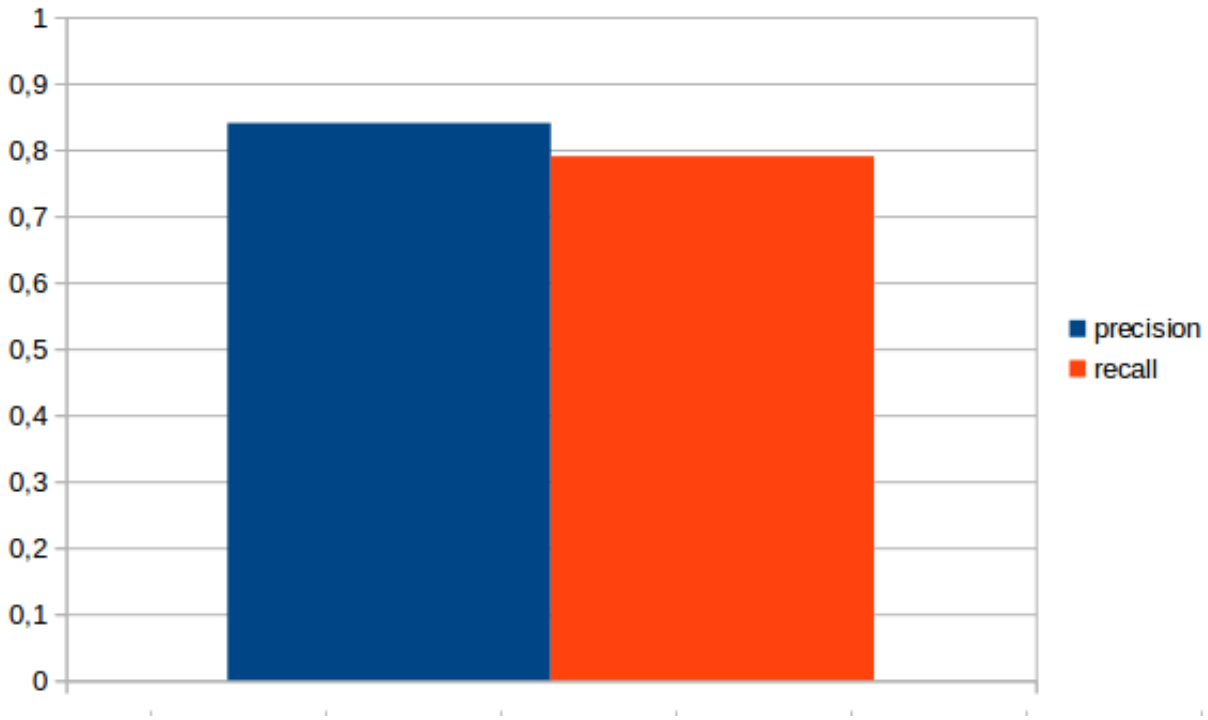

Figura 5.2: Precision e recall dell'algoritmo ECO feature

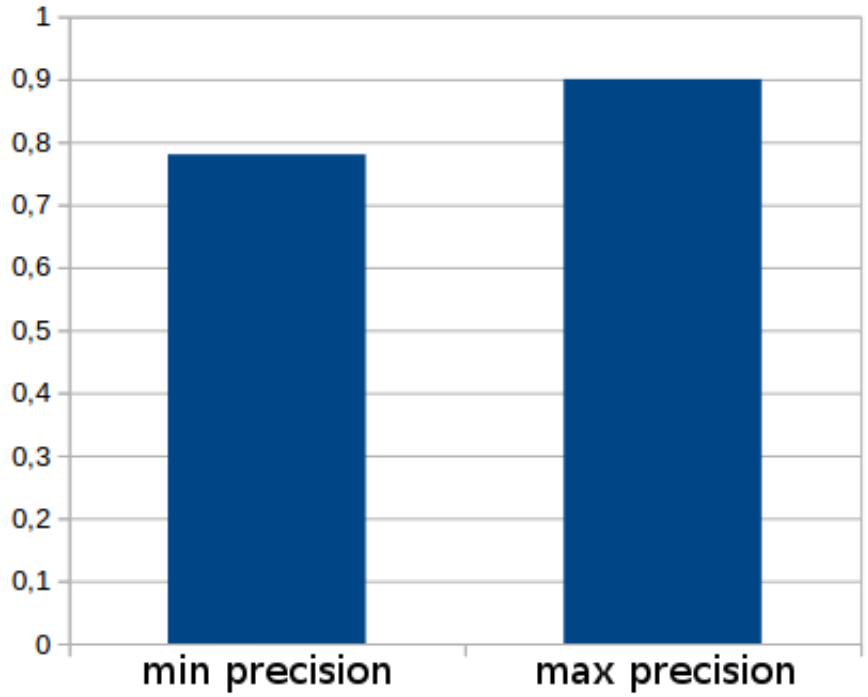

Figura 5.3: Valore minimo e massimo di precision ottenuti durante i test

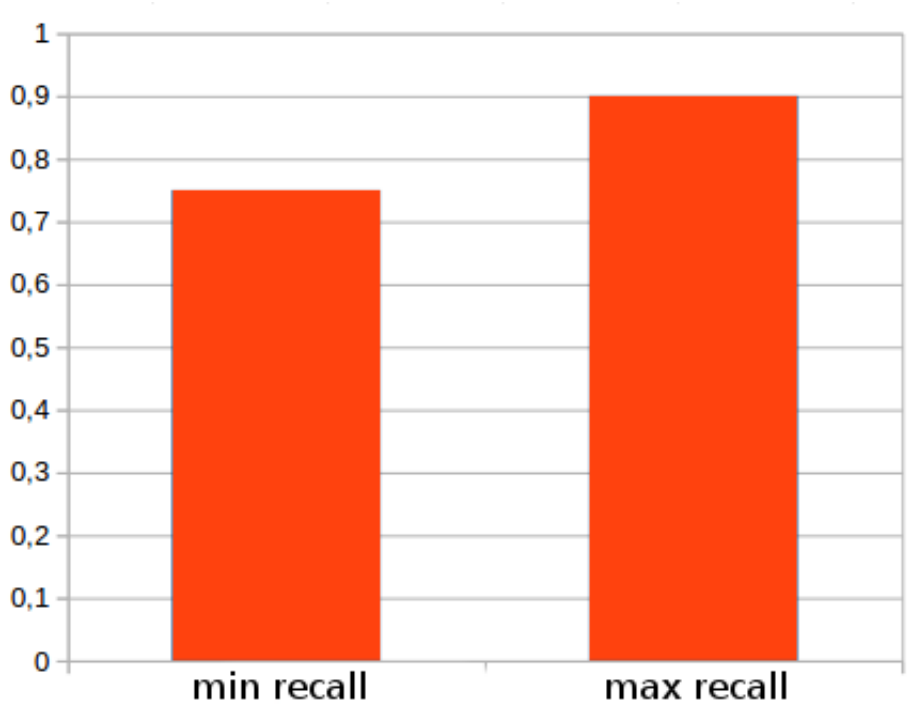

Figura 5.4: Valore minimo e massimo di recall ottenuti durante i test

I grafici 5.3 e 5.4 evidenziano rispettivamente i valori massimi e minimi di precision e recall ottenuti durante la fase di test
I risultati di precision e recall variano all'interno di un intervallo relativamente piccolo. Questa piccola differenza che si crea tra i risultati ottenuti è dovuta alla scelta casuale delle trasformazioni applicate alle immagini che in alcuni casi si rivelano essere più efficienti che in altri.

5.2 Tempi di esecuzione

L'analisi dei tempi di esecuzione si focalizza sugli andamenti dei tempi, sia per la versione sequenziale che per quella parallela, con l'aumentare della popolazione. Il numero di generazioni viene considerato meno rilevante poiché non essendo parallelizzabile non offre un guadagno diretto nell'algoritmo parallelo rispetto a quello sequenziale.

5.2.1 Valutazione dei tempi di esecuzione dell'algoritmo sequenziale

Per la valutazione dei tempi di esecuzione vengono effettuati diversi test in modo da analizzare l'andamento dei tempi con l'aumentare del carico computazionale.
La media del tempo di esecuzione dell'algoritmo sequenziale eseguito su insieme di 40 elementi della popolazione e 100 generazioni è all'incirca di 4700 secondi; durante i vari test dell'algoritmo sequenziale i tempi variano in un range di più o meno 100 secondi. La differenza nei tempi è probabilmente dovuta alle diverse trasformazioni applicate, dato che alcune richiedono un carico computazionale maggiore rispetto ad altre. Il successivo grafico, in figura 5.5, mostra l'andamento dei tempi di esecuzione in relazione all'aumentare degli elementi della popolazione. Ogni test è stato ripetuto dieci volte, il grafico riporta la media per ognuno di essi. Inoltre in questi test il numero di trasformazioni, che solitamente è un valore casuale tra 1 e 10,

viene mantenuto fisso a 5 in modo che i tempi di esecuzione non risentano di casi in cui abbiamo più o meno trasformazioni.

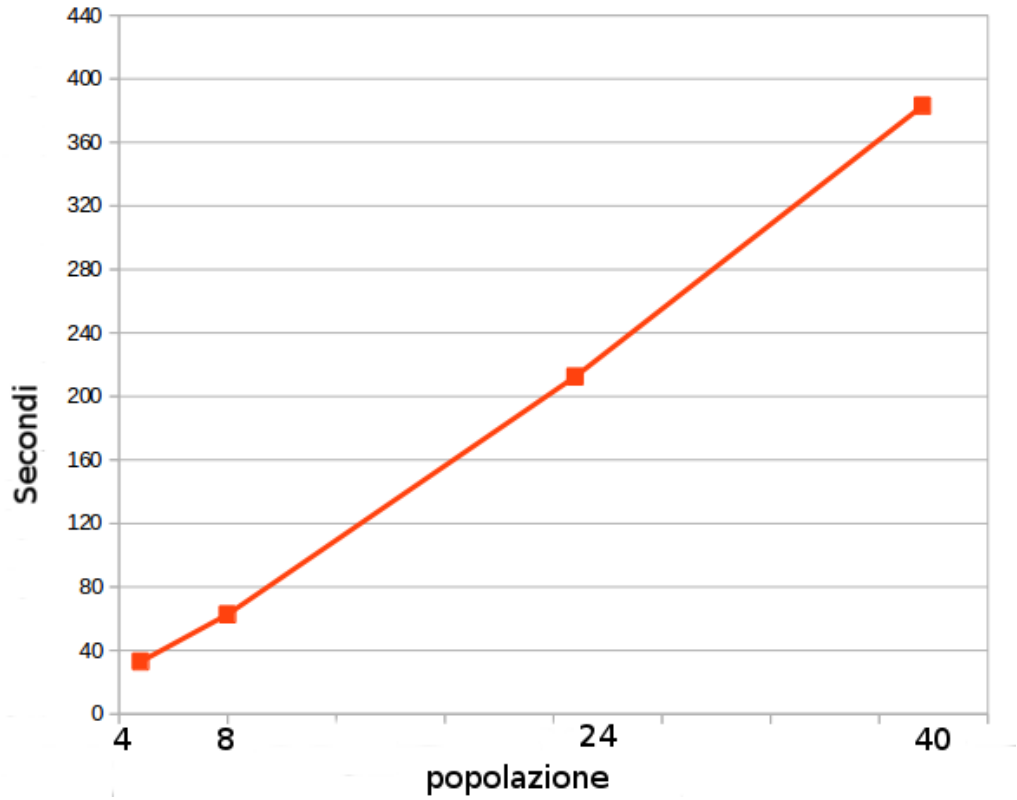

Figura 5.5: Andamento dei tempi di esecuzione all'aumentare della popolazione nell'algoritmo sequenziale

Il grafico precedente mostra i tempi di esecuzione che con 4 elementi della popolazione impiega in media circa 32 secondi. Raddoppiando il numero degli elementi il tempo segue lo stesso andamento con una media intorno ai 62 secondi, mentre con un numero di elementi 6 volte superiore a quello iniziale i tempi non seguono più quest'andamento ma iniziano ad essere più elevati. Nello specifico con 24 elementi abbiamo un tempo medio di poco superiore ai 210 secondi e con 40 elementi 383 secondi.

5.2.2 Valutazione dei tempi di esecuzione dell'algoritmo parallelo

Anche nel caso dell'implementazione parallela la valutazione dei tempi di esecuzione avviene effettuando diversi test in modo da analizzare l'andamento dei tempi con l'aumentare del carico computazionale.
La media del tempo di esecuzione dell'algoritmo parallelo eseguito su insieme di 40 elementi della popolazione e 100 generazioni è all'incirca di 2250 secondi; come nella versione sequenziale anche in questo caso le differenze dei tempi dei vari test oscillano in un breve intervallo e sono probabilmente dovute alle diverse trasformazioni applicate bei vari casi.
Il successivo grafico, in figura 5.6, mostra l'andamento dei tempi di esecuzione in relazione all'aumento degli elementi della popolazione.

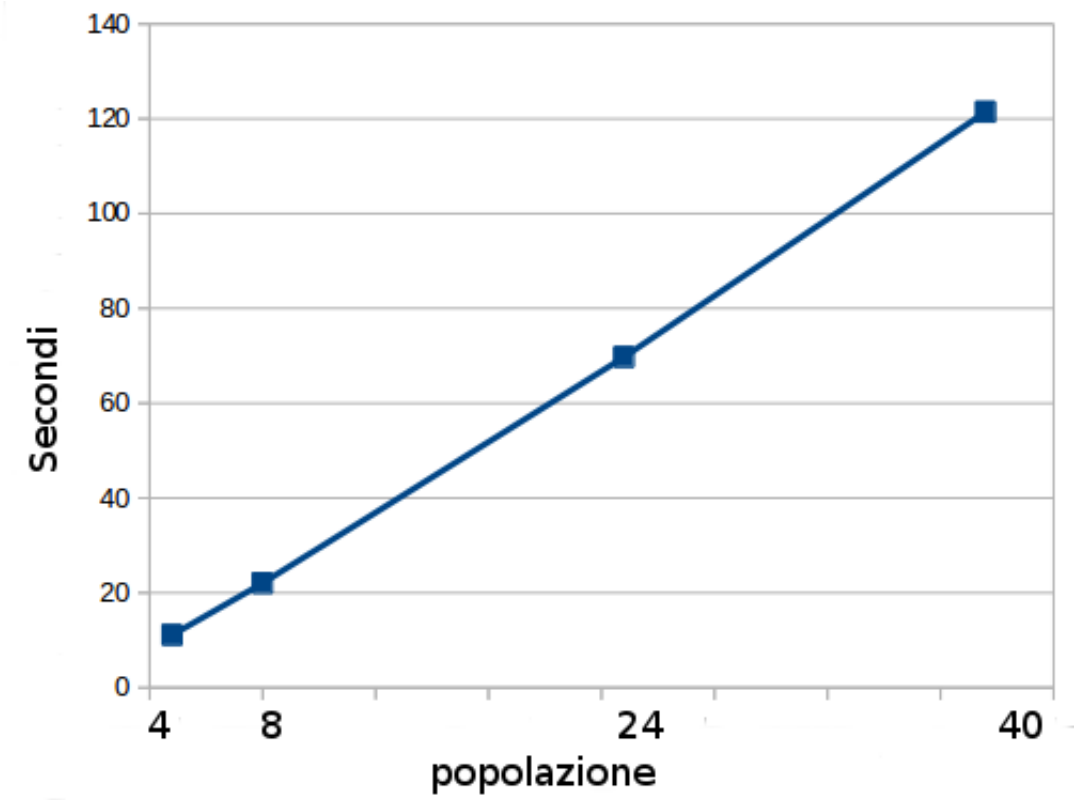

Figura 5.6: Andamento dei tempi di esecuzione all'aumentare della popolazione nell'algoritmo parallelo

Analizzando i tempi di esecuzione dell'algoritmo con l'aumentare della popolazione anche nel caso parallelo abbiamo un andamento simile alla versione sequenziale. Con 4 elementi della popolazione il tempo medio di esecuzione è di circa 11 secondi. Raddoppiando il numero degli elementi il tempo segue lo stesso andamento con una media di 22 secondi, mentre con un numero di elementi 6 volte superiore a quello iniziale i tempi non seguono più quest'andamento ma iniziano ad essere più elevati. Quindi con 24 elementi abbiamo un tempo medio di 71 secondi circa e con 40 elementi poco più di 120 secondi.

5.2.3 Confronto prestazioni algoritmo sequenziale e parallelo

Nell'analisi dei tempi viene dato maggiore risalto al numero degli elementi della popolazione piuttosto che al numero di generazioni perché, a differenza della dimensione della popolazione, le generazioni non possono essere parallelizzate come spiegato nel capitolo implementativo.

Confronto con diverse versioni di OpenCV Il primo confronto viene effettuato tra due implementazioni sequenziali, che però utilizzano due diverse versioni della libreria OpenCV. Vengono usate la versione 2.3.1, la stessa utilizzata dagli autori dell'articolo [12] e la versione aggiornata 2.4.11 che introduce un maggior numero di funzioni implementate su GPU.
Il test è stato effettuato impostando la popolazione a 40 elementi ed il numero di generazioni a 100.
Il grafico 5.7 mostra la differenza del tempo medio di esecuzione dell'Eco feature utilizzando le due diverse versioni della libreria. I guadagni in termini di tempo sono all'incirca del 19% e sono dovuti, oltre che dalla normale ottimizzazione delle funzioni della libreria, al porting di alcune di esse su GPU.

Confronto tra algoritmo sequenziale e parallelo Il confronto tra i tempi di esecuzione della versione sequenziale e quella parallela mostrano

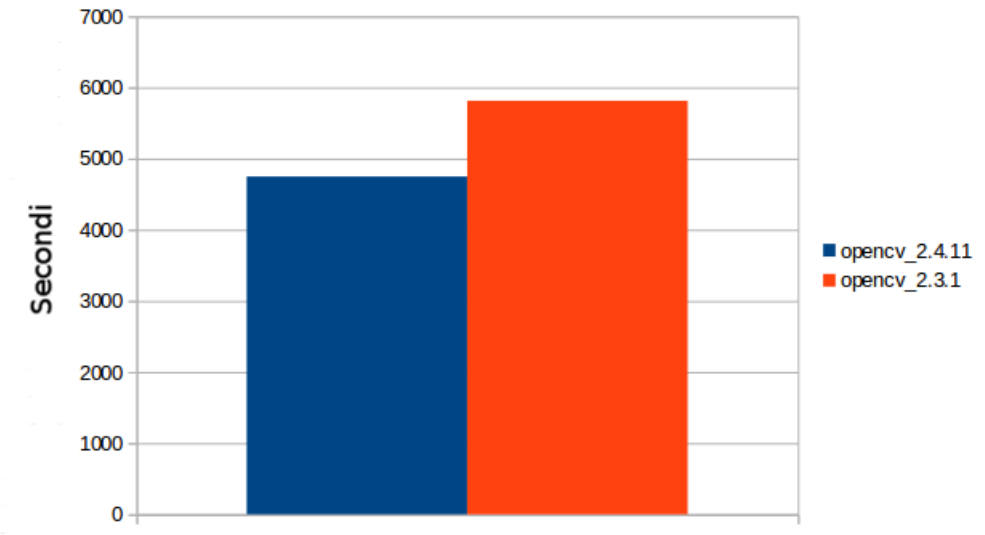

Figura 5.7: Differenza tempo di esecuzione con due diverse versioni della libreria OpenCV

un netto miglioramento, come era lecito aspettarsi. Tutti i test sono stati eseguiti su una macchina quad core.
Il primo confronto, nel grafico 5.8, mostra la differenza dei tempi medi tra l'algoritmo sequenziale e quello parallelo. Il test è stato effettuato su una popolazione di 40 elementi e 100 generazioni.
I risultati mostrano come la versione parallela impieghi meno della metà del tempo rispetto al tempo di esecuzione necessario all'algoritmo sequenziale.

Il grafico 5.9 mostra come la forbice dei tempi di esecuzione tenda ad allargarsi sempre di più con l'aumentare della popolazione. Questo si verifica perché all'aumentare dei dati il tempo di sincronizzazione necessario in fase di avvio dei processori e in fase di raccolta dei risultati parziali influisce sempre di meno sul tempo totale di esecuzione. Come detto i test sono stati effettuati su una macchina quad core, dunque ci si potrebbe aspettare che i tempi si riducano ad un quarto di quanto necessario in sequenziale; ovviamente questo non avviene sia per i già citati tempi di sincronizzazione necessari ai vari processori e nel caso specifico anche perché non tutto il codice è parallelizzabile. Inoltre ci possono essere dei casi in cui un implementazione sequenziale

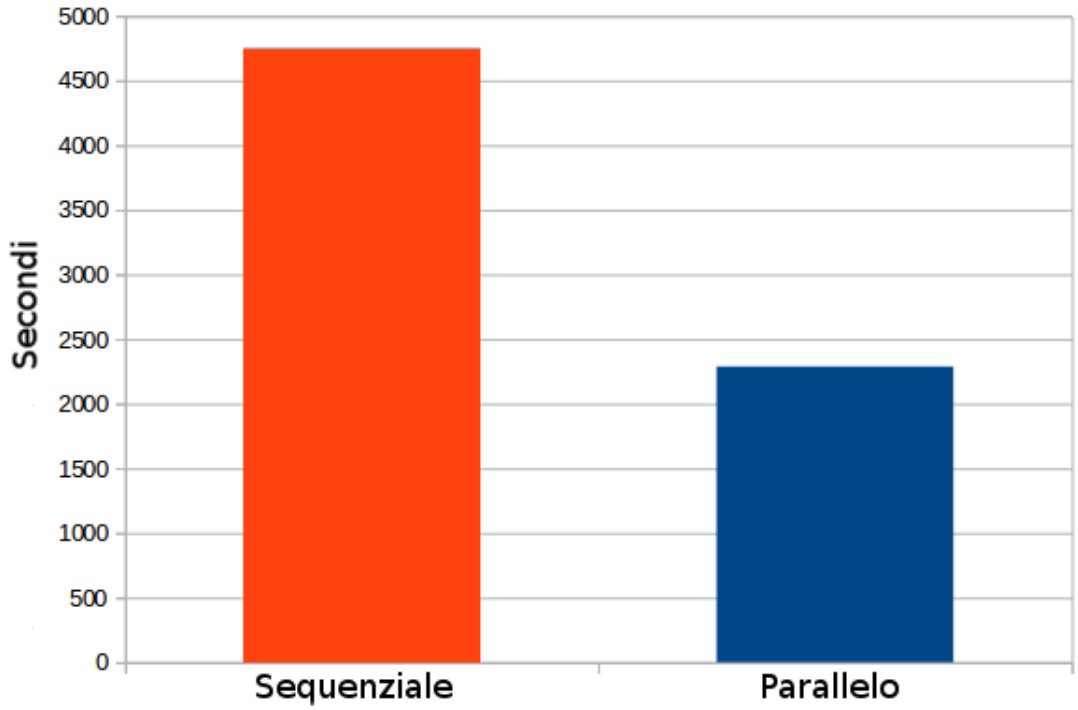

Figura 5.8: Media dei tempi di esecuzione per la versione sequenziale e quella parallela dell'Eco feature

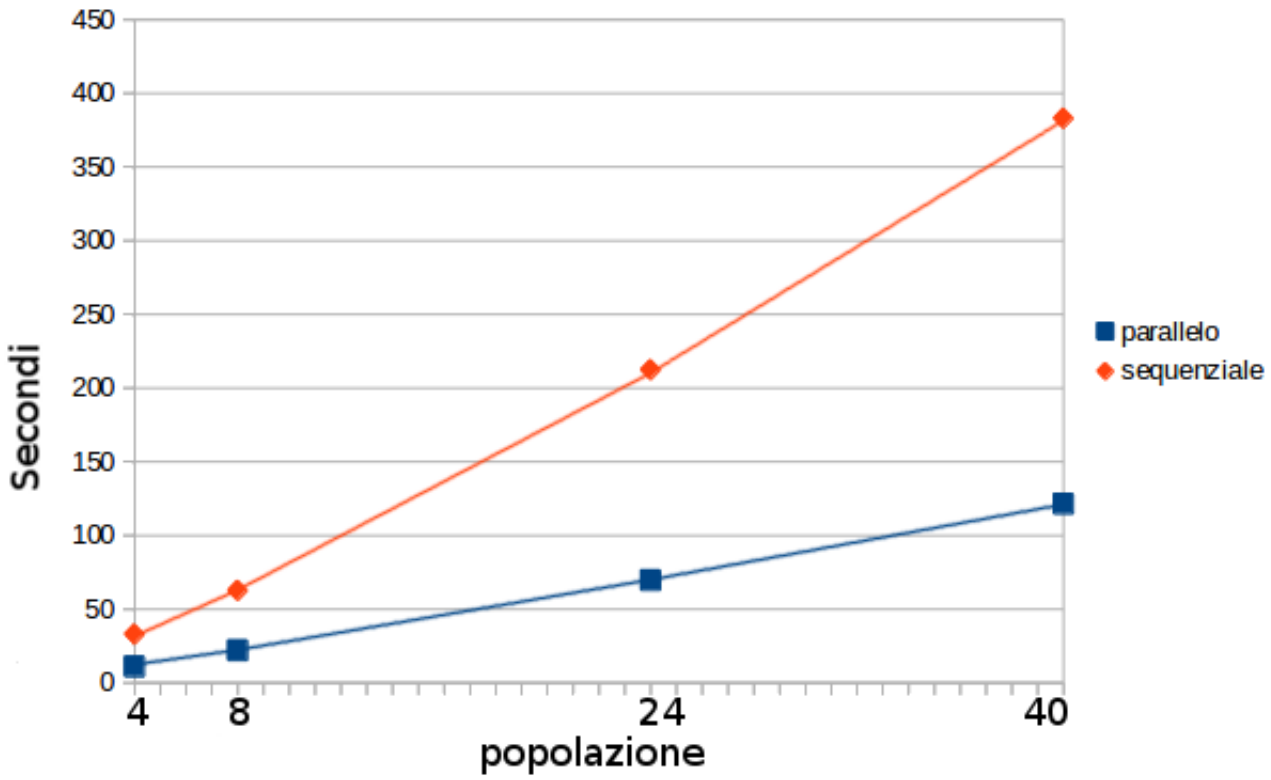

Figura 5.9: Confronto tra la versione sequenziale e quella parallela dell'Eco feature all'aumentare della popolazione

si rivela essere più efficiente rispetto al caso parallelo. Questo caso si verifica proprio nella fase di inizializzazione degli elementi della popolazione, in cui la versione sequenziale impiega meno tempo rispetto alla controparte parallela, come mostrato nel grafico 5.10.

Questa differenza a vantaggio della versione sequenziale probabilmente è dovuta al basso carico computazionale necessario a svolgere questa fase e quindi i tempi della versione parallela dell'algoritmo vengono fortemente influenzati dalle sincronizzazioni necessarie ai vari processori in fase di callback per l'unificazione dei risultati parziali.
Inoltre i tempi, in entrambi i casi, crescono molto più lentamente rispetto all'andamento dell'algoritmo.

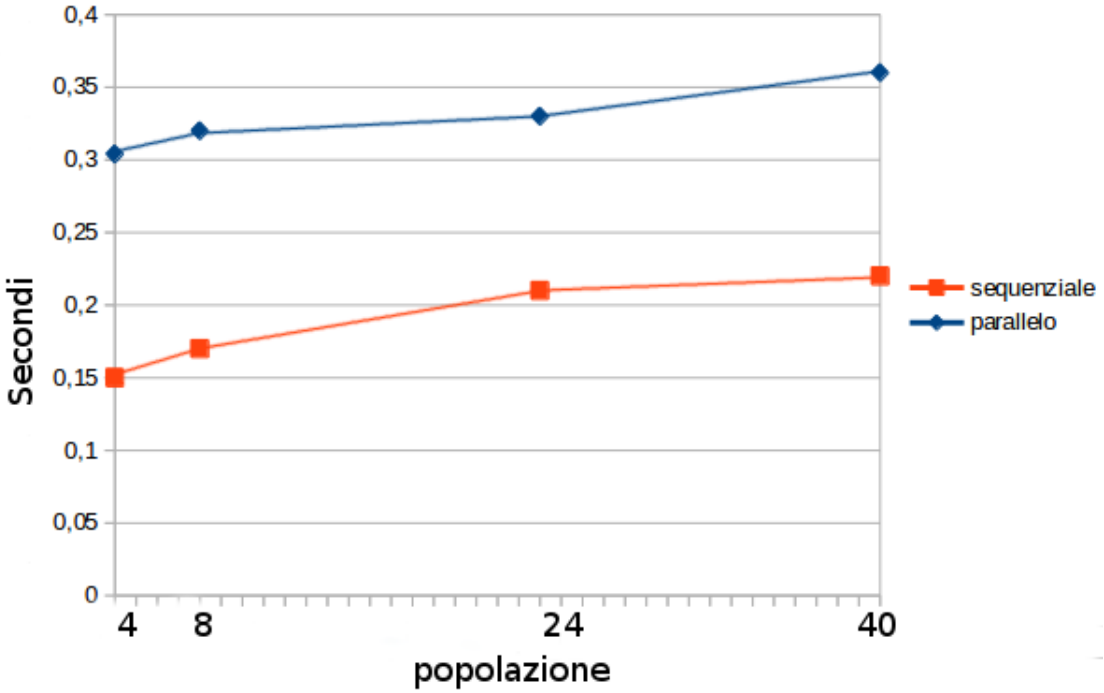

Figura 5.10: Confronto della fase di inizializzazione tra la versione sequenziale e quella parallela dell'Eco feature

Calcolo dello speedup Lo speedup è una metrica che valuta il miglioramento del tempo di computazione relativo all'esecuzione di un processo su due architetture con differenti risorse, in questo caso le risorse sono i processori utilizzati per la computazione.
Prendendo i tempi di esecuzione dei test precedenti, viene calcolato lo speedup tramite la seguente espressione:

$$S = \frac{t_s}{t_p} \tag{5.3}$$

S è il rapporto tra il tempo sequenziale e quello parallelo necessario per l'esecuzione dell'algoritmo. Per ottenere uno speedup lineare, che indica il risultato massimo ottenuto tramite la parallelizzazione, il rapporto tra S ed il numero dei processi deve essere pari ad 1.
Il grafico 5.3 mostra il valore dello speedup in base al numero di processori utilizzati per l'esecuzione. Come si può notare non viene mai raggiunto il valore ideale, rappresentato dallo speedup lineare, il motivo è dovuto non solo ai tempi di sincronizzazione dei processori, già citati precedentemente, ma anche perché diversi punti dell'algoritmo non possono essere parallelizzati e vengono quindi eseguiti in sequenziale.
Il risultato ottenuto mostra come passando da uno a due processori si ha un miglioramento delle prestazioni, ma è utilizzando quattro processori che si ottiene il miglioramento più ampio, questo per via dell'hardware del dispositivo utilizzato. Con 6 o 8 processori lo speedup migliora di poco perché in questo caso la macchina utilizzata per i test dispone di numero di processori fisici minore. Inoltre dal grafico si nota come lo speedup migliori senza mai raggiungere il valore di speedup lineare.

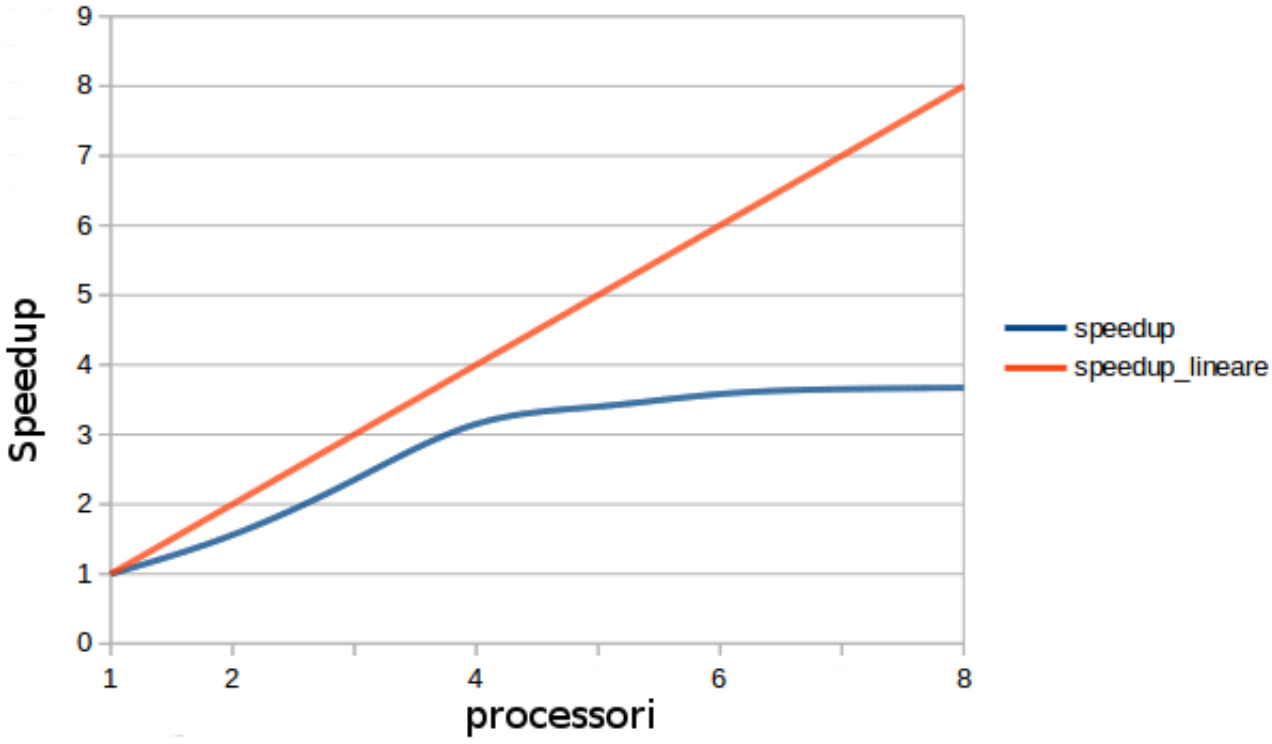

Figura 5.11: Calcolo dello speedup in relazione al numero di processori utilizzati

Capitolo 6

Conclusioni e Sviluppi Futuri

L'obiettivo finale di questa tesi è implementare e confrontare una versione sequenziale ed una parallela dell'algoritmo Eco feature ed analizzare quali sono gli effettivi guadagni in termini di tempo di esecuzione.
Innanzitutto dai risultati ottenuti si nota come la parallelizzazione non abbia in alcun modo inficiato sulle performance di precision e recall dell'algoritmo, ottenendo dei risultati pressoché identici in entrambi i casi.
Focalizzando l'attenzione sui tempi di esecuzione si nota come la parallelizzazione, come è lecito aspettarsi, offra dei concreti vantaggi nei tempi di esecuzione, riducendoli a meno della metà rispetto alla versione sequenziale. Inoltre con l'aumentare del carico computazionale questa forbice tende ad aumentare. Purtroppo non si riesce mai a raggiungere il valore ottimale di parallelizzazione rappresentato dallo speedup lineare, questo per i vari motivi discussi precedentemente come le sincronizzazioni dei processori e le porzioni di codice non parallelizzabile.
In conclusione si può affermare che, anche se non viene mai raggiunto il valore ottimo, i risultati ottenuti dalla versione parallela dell'algoritmo mostrano un notevole guadagno nei tempi di esecuzione riducendoli a meno della metà rispetto alla versione sequenziale, tutto questo senza influire sulla precision e la recall.

6.1 Sviluppi futuri

L'Eco feature è un algoritmo molto efficiente nel riconoscimento degli oggetti senza utilizzare nessun tipo di conoscenza fornita da un essere umano, dunque sotto questo punto di vista probabilmente è poco il lavoro da fare che potrebbe renderlo ancora migliore. Il problema principale, come per la maggior parte degli algoritmi di machine learning, è dato dall'elevata utilizzo di risorse dunque non solo nel tempo di esecuzione ma anche nell'utilizzo della memoria. In questa tesi viene effettuato un lavoro sul miglioramento dei tempi di esecuzione tramite la parallelizzazione dell'algoritmo, in futuro si potrebbe pensare di migliorare la gestione della memoria o di altre risorse per poter magari effettuare un porting su dispositivi come gli smartphone, sempre più potenti ma che dispongono di un insieme di risorse limitato rispetto ad un computer moderno.

Bibliografia

[1] Sven Bambach. A survey on recent advances of computer vision algorithms for egocentric video. *arXiv preprint arXiv:1501.02825*, 2015.

[2] Arthur J Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, (5):757–763, 1966.

[3] Gary Bradski et al. The opencv library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.

[4] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *ICML*, volume 96, pages 148–156, 1996.

[5] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[7] Richard M Karp. A survey of parallel algorithms for shared-memory machines. 1988.

[8] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[9] Mrs J Komala Lakshmi and M Punithavalli. A survey on performance evaluation of object detection techniques in digital image processing. *IJCSI*, page 86, 2010.

[10] Scikit learn. Scikit-learn, 2014.

[11] Kirt Lillywhite, Dah-Jye Lee, Beau Tippetts, and James Archibald. A feature construction method for general object recognition. *Pattern Recognition*, 46(12):3300–3314, 2013.

[12] Kirt Lillywhite, Beau Tippetts, and Dah-Jye Lee. Self-tuned evolution-constructed features for general object recognition. *Pattern Recognition*, 45(1):241–251, 2012.

[13] Mitchell Melanie. An introduction to genetic algorithms. *Cambridge, Massachusetts London, England, Fifth printing*, 3, 1999.

[14] Python. Parallel processing, 2014.

[15] Peter M Roth and Martin Winter. Survey of appearance-based methods for object recognition. *Inst. for Computer Graphics and Vision, Graz University of Technology, Austria, Technical Report ICGTR0108 (ICG-TR-01/08)*, 2008.

[16] Robert E Schapire. Explaining adaboost. In *Empirical inference*, pages 37–52. Springer, 2013.

[17] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[18] Daniel L Swets, Bill Punch, and John Weng. Genetic algorithms for object recognition in a complex scene. In *Image Processing, 1995. Proceedings., International Conference on*, volume 2, pages 595–598. IEEE, 1995.

[19] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[20] Paul Viola. Feature-based recognition of objects.