Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

ANALISI ED IMPLEMENTAZIONE DI UN'APPLICAZIONE E DI UN CHATBOT PER LA CORREZIONE AUTOMATICA DI OPERAZIONI MATEMATICHE SCRITTE A MANO

Tesi di Laurea in:
ALGORITMI E STRUTTURE DATI

Relatore:
Prof. LUCIANO MARGARA

Presentata da:
PETRU POTRIMBA

Indice

Introduzione

Immaginate il seguente scenario realistico:

una maestra di matematica della scuola primaria di primo grado assegna ai propri bambini una serie di esercizi da fare a casa. Quando per la maestra arriva il momento della correzione, si ritrova circa 20/25 quaderni da correggere, ognuno contenente una decina o ventina di operazioni. Nonostante le operazioni da correggere possano risultare semplici, con un ammontare complessivo di 300/400 operazioni, la procedura di correzzione può diventare un compito tedioso e time-consuming.

L'obiettivo del mio lavoro di tesi è stato rendere automatica la correzione di queste operazioni.
Per verificare che effettivamente questa cosa si potesse fare, ci si è concentrati su alcune operazioni di test. In particolare, ci si è concentrati su operazioni quali somme, sottrazioni, moltiplicazioni e divisioni sia piane che in colonna. Quindi ci interessava capire se, con un dispositivo mobile e con un quaderno di matematica scritto a mano, c'era la tecnologia adeguata tale per cui, scattando una foto sul quaderno, si riusciva a raggiungere l'obiettivo imposto.
Da notare che destinatario di questo lavoro non è solamente la maestra di matematica della scuola primaria, ma sono anche i genitori che devono correggere i compiti dei propri figli, i quali, per diversi motivi, potrebbero non avere il tempo per farlo.
Si è pensato per cui di sviluppare una applicazione mobile che permetta di realizzare questo comportamento in modo efficacie e veloce.

Ovviamente, è importante andare a vedere prima se questa mia idea era già presente il letteratura. Effettivamente qualcosa di simile si è

trovato. In particolare si sono trovate applicazioni in cui, scattando una foto ad *una singola operazione* scitta a mano, questa riusciva a riconoscere i vari caratteri e simboli dell'operazione. Nonostante l'obiettivo di queste applicazioni sembri essere molto simile al mio, in realtà non è così in quanto queste applicazioni hanno come unico obiettivo il riconoscimento di una singola operazione matematica, il nostro obiettivo invece è quello di riuscire a riconoscere *tutte le operazioni* che sono presenti in una foto e successivamente correggerle. Per cui sorge la problematica di riuscire a segmentare (ovvero riuscire a separare uno o più oggetti di interesse dallo sfondo) le diverse operazioni matematiche. Non solo: essendo che il nostro obiettivo è anche correggere queste operazioni , disegnando opportuni simboli che indicano la correttezza o meno della operazione, è necessario calcolare il loro posizionamento nella foto in termini di coordinate (x, y). Nonostante quindi queste applicazioni eseguano un'altro compito, si è pensato però che potessero tornarci comunque utili nel momento in cui fossi riuscito a segmentare tutte le diverse operazioni e per ognuna fossi riuscito a trovare le sue coordinate sull'immagine. Così facendo, sarei stato in grado di ritagliare le singole operazioni matematiche dall'immagine originale e successivamente utilizzare le API di quelle applicazioni per riconoscere l'operazione matematica segmentata. Si quindi è deciso di utilizzare questo flusso di lavoro.

Si è scelto di sviluppare l'applicazione su piattaforma iOS con linguaggio di programmazione Swift 3. Per eseguire operazioni di elaborazione di immagini si è fatto uso della libreria GPUImage2.
Come prima elaborazione, è stata fatta una trasformazione dell'immagine da RGB a Grayscale in quanto il colore non ci veniva in aiuto. Il primo problema che ho affrontato è stato quello di dover segmentare tutti i caratteri e simboli delle operazioni matematiche dallo sfondo. Una prima soluzione è stata quella di applicare una binarizzazione con soglia globale ricavata tramite l'istogramma dell'immagine stessa. Questa soluzione però è risultata essere poco soddisfacente in quanto poteva verificasi che, dipendentemente dalle condizioni di luce in cui la foto veniva scattata, l'algoritmo di binarizzazione o non riusciava ad eliminare completamente i quadretti che i fogli dei quaderni di matematica hanno, oppure poteva accadere che intere operazioni, o porzioni di esse, venissero rimosse.

Si è quindi deciso di adottare un'altro approccio utilizzando l'algoritmo Canny Edge Detection che, configurato con opportune soglie, riesce efficacemente ad trovare i bordi dei caratteri e simboli di interesse eliminando il rumore provocato dai quadretti.
Questa soluzione funziona correttamente solo nelle condizioni in cui si scatta una foto senza inquadrare i bordi del foglio. Infatti l'algoritmo Canny Edge Detection ha lo scopo di trovare i bordi in un'immagine e, nel caso in cui si scatta una foto inquadrando anche i bordi del foglio, l'algoritmo li trova e li etichetta come caratteri e simboli di una possibile operazione matematica.
Nonostante questo difetto, è stato deciso comunque di adoperare questo algoritmo e lasciando questo problema da risolvere nei lavori fututi.
Segmentati tutti i diversi caratteri, il problema ora sta nel capire, per ogni carattere, a quale operazione matematica appartiene. Per riuscire a risolvere questo problema è stato necessario analizzare i diversi quaderni di matematica di diversi bambini delle scuole primarie per vedere se questi avessero una struttura comune che mi avrebbe permesso di scrivere un algoritmo ad hoc. Si è notato che una operazione è distanziata da un'altra operazione per almeno un quadretto in tutte le direzioni. Grazie a questo pattern comune, sono riuscito a scrivere un'algoritmo che separasse le diverse operazioni matematiche utilizzando opportune soglie fisse precalcolate.
L'applicazione di questo algoritmo mi ha permesso non solo di separare le diverse operazioni, ma mi ha permesso anche di trovare le coordinate (x, y) per ogni operazione nell'immagine.
Avendo le coordinate di ogni operazione, sono riuscito a ritagliare ogni operazione dall'immagine per poi utilizzare le API di Mathpix (una piattaforma in grado di riconoscere caratteri e simboli della foto di una operazione matematica passatagli in ingresso) per avere in output un risultato che mi rappresentasse l'operazione matematica passatagli in input. Essendo che il software di riconoscimento è in esecuzione sui server di Mathpix, si è incorso in diverse problematiche. In particolare una delle quali è stata che la risposta con il risultato del riconoscimento può arrivare in un qualsiasi momento imprevedibile. Si è risolto il problema creando un buffer che implementasse il pattern Observer in modo tale che, quando il server di Mathpix riempiva il buffer con la sua risposta, il buffer notificava all'applicazione che il messaggio con la risposta era arrivato e quindi pronto per essere processato.

Un'altra problematica riscontrata è stata che i server di Mathpix introducono un ritardo non indifferente quando devono processare un numero di operazioni consistente. Si è deciso comunque di appoggiarsi a questi server esterni per il riconoscimento delle operazioni e risolvere il problema nei lavori futuri.
Mathpix, dopo l'elaborazione, restituisce il risultato in formato JSON. Il risultato consiste in una serie di informazioni tra cui il risultato del riconoscimento dell'operazione matematica. Questo risultato può variare a seconda della specifica operazione (operazioni piane o operazioni in colonna), quindi è stato necessario scrivere algoritmi che, dipendentemente dal tipo di operazione, fossero in grado di estrapolare il risultato coerente. Una volta reso automatico questo processo di estrapolazione del risultato, le operazioni matematiche vengono analizzate per capire se queste sono corrette o meno. Avendo le coordinate per ogni operazione e sapendo se queste sono corrette o meno, si prende l'immagine originale scattata dal dispositvo mobile e si disegna direttamente su di essa un simbolo che rappresenti la correttezza o meno dell'operazione.

L'intero algoritmo restituisce risultati soddifacente per la maggior parte de casi, tuttavia è affetto da problemi legato alle soglie fisse scelte per la segmentazione delle singole operazioni matematiche. In particolare, quando viene scattata una foto ad una distanza piuttosto ravvicinata al foglio, l'algoritmo può non essere più in grado di segmentare correttamente le operazioni. Era necessario per cui variare queste soglie in base alla distanza che intercorreva tra il dispositivo e il foglio al momento dello scatto della foto. Per risolvere questo problema, si è fatto un utilizzo indiretto della realtà aumentata, utilizzando il framework ARKit offerto da Apple.

Capitolo 1

Tecniche algoritmiche per il riconoscimento di immagini

In letteratura sono presenti diversi algoritmi di riconoscimento di immagini ma nessuno di questi, singolarmente, concretizzava completamente il mio obiettivo. Il primo problema affrontato è stato la segmentazione dei diversi caratteri e simboli delle operazioni matematiche presenti in un'immagine. Come operazione preliminare è stata applicata la trasformazione dell'immagine a colori in immagine a livelli di grigio. Successivamente viene illustrato un possibile approccio di segmentazione basato sull'algoritmo di binarizzazione. Questa soluzione però è risultata essere poco efficace e piuttosto fragile in quanto, dipendentemente dalle condizioni di luce in cui la foto veniva scattata, l'algoritmo poteva non restituire il risultato corretto e quindi si optato per un'altro approccio utilizzando l'algoritmo Canny Edge Detection. Una volta riusciti a segmentare con successo i caratteri e simboli di tutte le operazioni matematiche, il successivo problema affrontato è stato riuscire a segmentare le intere operazioni matematiche e quindi riuscire a capire, per ogni carattere e simbolo segmentato, a quale operazione matematica appartenesse.
Ilustreremo in questo capitolo i diversi algoritmi impiegati, le problematiche sorte e come sono state risolte.

1.1 Da RGB a Grayscale

La prima operazione effettuata, essendo che il colore non veniva in aiuto in nessun modo, è stata la trasformazione dell'immagine scattata dal dispositivo dallo spazio RGB a Grayscale.

1.1.1 Formato RGB

Il formato RGB è un modello di colori di tipo *additivo*: i colori sono definiti come somma dei tre colori Rosso (Red), Verde (Green) e Blu (Blue), da cui appunto l'acronimo RGB, da non confondere con i colori primari sottrattivi giallo, ciano e magenta.
Altri 5 colori notevoli di questo tipo di modello sono il giallo (Rosso + Verde), il magenta (Rosso + Blu) e il ciano (Verde + Blu), inoltre la somma dei tre colori costituisce il bianco, e la loro totale assenza il nero.
Per le sue caratteristiche, è un modello particolarmente adatto nella rappresentazione e visualizzazione di immagini in dispositivi elettronici

1.1.2 Formato Grayscale

Il formato Grayscale di una immagine rappresenta il fatto che il valore di ogni pixel è un singolo campione che descrive una determinata quantità di luce. Immagini di questo formato, anche note come immagini in bianco e nero oppure immagini di grigio monocromatico, sono composte esclusivamente da sfumature di grigio. Il contrasto varia dal nero, l'intesità più debole, al bianco, l'intensità più forte. Esiste una sottoparte di immagini appartenenti a questo formato che sono costituite da unicamente due colori: il bianco e il nero. Queste immagini vengono chiamate immagini *binarie*. L'intensità di ciascun pixel è espressa all'interno di un range che si distribuisce da un valore minimo ad un valore massimo, inclusi. Questo spazio di valori è rappresentato in maniera astratta come un range che parte da 0 (totale assenza di luce, nero), a 1 (totale presenza della luce, bianco), con valori frazionari in mezzo. Questa è una nozione utilizzata solo per scopi accademici, non definisce i colori in termini di colometria. A volte, infatti, la scala può essere rovesciata. Si consideri per esempio l'esempio di una stampante: nel suo caso il valore numerico di intensità rappresenta la quantià di inchiostro da impiegare, indicando con 0 nessuna quantità di inchiostro

(foglio bianco) e con 1 la massima quantità di inchiostro da apporre (foglio nero).

1.1.3 Conversione da RGB a Grayscale

In letteratura sono presenti diversi algoritmi per convertire una immagine RGB in un'immagine Grayscale. Tuttavia tutti questi algoritmi, di base, utilizzano gli stessi tre step[1]:

1. prendere l'intensità dei canali rosso, verde e blu di un pixel;
2. applicare un algoritmo che trasformi questi valori in un singolo valore che rappresenti l'intensità del livello di grigio corrispondente;
3. sostituire i valori originali del rosso, verde e blu con il valore ottenuto al punto precedente.

L'algoritmo che è stato scelto per la conversione da RGB a Grayscale è il seguente:

Algorithm 1 Convert from RGB to Grayscale

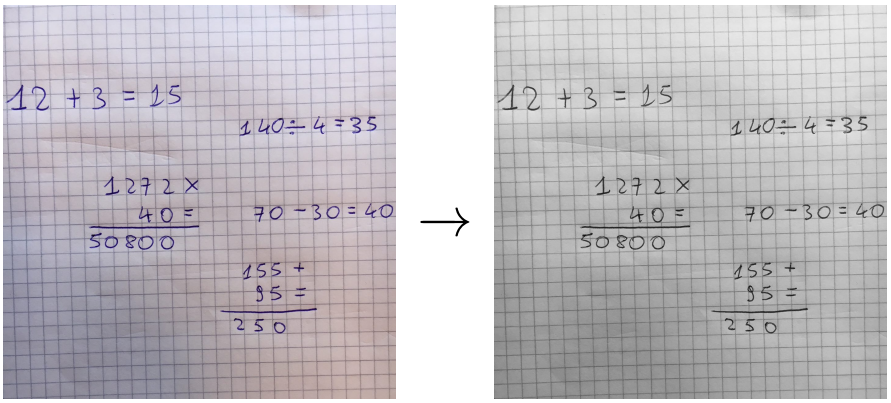
```
function CONVERT(image)
    for each pixel ∈ image do
        Red ← pixel.Red
        Green ← pixel.Green
        Blue ← pixel.Blue

        Gray ← (Red + Green + Blue)/3

        Pixel.Red ← Gray
        Pixel.Green ← Gray
        Pixel.Blue ← Gray
    end for
    return image
end function
```

Il criterio che decide come ogni pixel dell'immagine RGB debba essere transformato in un pixel Grayscale è mostrano nella linea 7

dell'algoritmo. In Figura 2.1 viene mostrato il risultato dell'algoritmo Convert.



(a) Immagine RGB (b) Immagine Grayscale

Figura 1.1: Applicazione dell'algoritmo Convert all'immagine (a).

1.2 Binarizzazione

Per riuscire a segmentare tutti i caratteri e simboli delle operazioni matematiche, come primo approccio si è adottato l'algoritmo di binarizzazione con soglia globale ricavata tramite l'istogramma dell'immagine grayscale prodotta dall'algoritmo precedente Convert. L'algoritmo di binarizzazione consiste nel catalogare i singoli pixel dell'immagine come "pixel oggetto" se il loro valore è maggiore di una certa soglia e come "pixel di sfondo" se il valore è sotto la soglia. L'algoritmo di binarizzazione è il seguente:

Algorithm 2 Binarization of an image

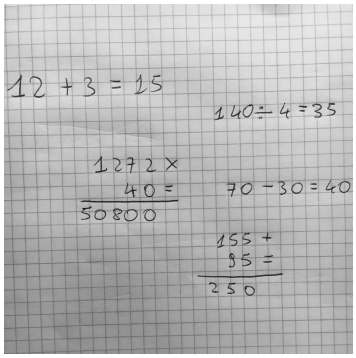
```
1: function BINARIZATION(image, threshold)
2:     for each pixel ∈ image do
3:         if pixel ≤ threshold then
4:             pixel ← 0
5:         else
6:             pixel ← 1
7:         end if
8:     end for
9:     return image
10: end function
```

1.2.1 Istogramma

Uno dei metodi principali per andare a ricavare una soglia adatta è analizzare l'istogramma dell'immagine. L'istogramma di un'immagine grayscale[2] indica il numero di pixel dell'immagine per ciascun livello di grigio. Dall'istogramma si possono estrarre informazioni interessanti, quali:

- se la maggior parte dei valori solo "condensati" in una zona, ciò significa che l'immagine ha uno scarso contrasto;
- se nell'istogramma sono predominanti le basse intensità, significa che l'immagine è molto scura e viceversa;
- se i diversi oggetti in un'immagine hanno livelli di grigio differenti, l'istogramma può fornire un primo semplice meccanismo di classificazione. Ad esempio un istogramma bimodale denota spesso la presenza di un oggetto abbastanza omogeneo su uno sfondo di luminosità pressoché costante.

L'istogramma dell'immagine grayscale ottenuta precedentemente è il seguente:



(a) Immagine Grayscale

(b) Istogramma dell'immagine

Figura 1.2: L'immagine (b) rappresentea l'istogramma dell'immagine (a).

Analizzando l'istogramma, si può notate come la maggior parte dei valori siano "condensati" in una determinata zona. Non si riesce per cui in modo diretto a trovare una soglia che permetta poi di essere utilizzata per la binarizzazione. Occorre per cui effettuare delle operazioni preliminari sull'istogramma. Le principali due operazioni che vengono applicate sono:

- la radice quadrata, ovvero ogni valore dell'istogramma viene sostituito con la sua radice quadrata (arrotondata all'intero più vicino);

- uno smooth ai valori dell'istogramma, ovvero ogni elemento viene ricalcolato come media locale su di un'intorno preliminarmente scelto.

Dopo l'applicazione di entrambe queste operazioni si ottiene l'istogramma seguente:

Figura 1.3: Istogramma dell'immagine (a) in Figura 2.2 a cui è stato applicato la radice quadrata e l'operazione di smooth.

Rispetto a prima, dove le tonalità basse erano quasi inesistenti, adesso l'istogramma risulta essere più distribuito su tutti i livelli di grigio. L'approccio più semplice e che a dispetto della sua facilità restituisce spesso dei risultati abbastanza soddisfacenti, è quello di prendere come soglia il valore medio dell'istogramma delle tonalità di grigio. La soglia viene quindi calcolata come la somma dell'intensità di tutti i pixel diviso il numero totale dei pixel. Applicando l'algoritmo di binarizzazione sull'immagine (a) in Figura 2.2 con la soglia appena ricavata, si ottiene il seguente risultato:

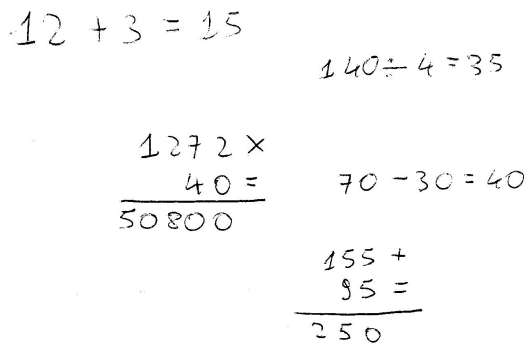


Figura 1.4: Risultato dell'algoritmo di binarizzazione applicato all'immagine (a) in Figura 2.2 con la soglia ricavata dall'istogramma in Figura 2.3.

Il risultato è abbastanza soddifancente in quanto si è riusciti a segmentare i vari caratteri e simboli delle operazioni matematiche eliminando i quadretti presenti sullo sfondo senza perdere troppi dettagli. Questo risultato è stato possibile in quanto la foto era in condizioni di lluminazioni pressochè ideali e ciò ha permesso, grazie al calcolo dell'istogramma e al suo aggiustamento tramite radice quadrata e smoothing, di poterci calcolare una soglia che eseguisse efficacemente questo compito.
Applicando lo stesso algoritmo su una foto con un livello di illuminazione differente si potrebbero però incorrere in diversi problemi. In particolare, quando l'immagine contiene una brusca variazione di luminosità, è possibile che l'algoritmo non segmenti correttamente i caratteri delle diverse operazioni matematiche. In Figura 2.5 mostriamo l'output dell'algoritmo di una immagine contenente una brusca variazione di luminosità:

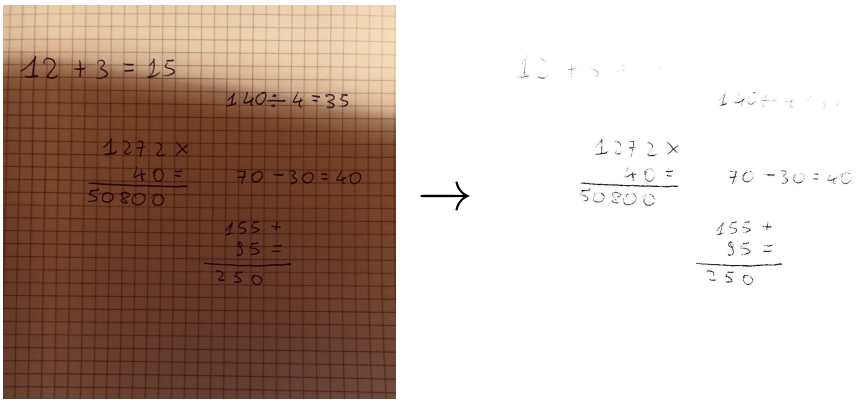


Figura 1.5: Risultato dell'algoritmo applicato ad un'immagine contenente una brusca variazione di luminosità.

Come si può chiaramente notare, in presenza della brusca variazione di luminosità, le operazioni vengono quasi del tutto cancellate e il resto delle operazioni risultano essere poco chiare.
L'algoritmo di binarizzazione è affetto da problemi anche quando l'immagine è molto luminosa. Mostriamo in Figura 2.6 un esempio:

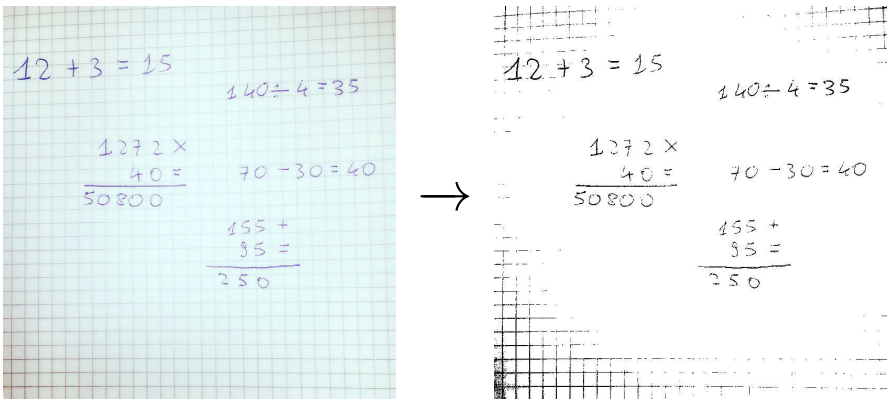


Figura 1.6: Risultato dell'algoritmo applicato ad un'immagine contenete una elevata luminosità.

Come si può notare, in presenza di immagini con un'elevata luminosità, l'algoritmo fatica a segmentare i caratteri e simboli delle operazioni matematiche. Il problema principale però consiste nel fatto che non riesce ad eliminare i quadretti, come si può ben notare degli angoli dell'immagine di output dell'algoritmo.
Questi problemi sono dovuti al fatto che l'algoritmo scelto per calcolare la soglia da utilizzare per la binarizzazione non tiene conto della luminosità dell'immagine.
In letteratura sono presenti diverse possibili soluzioni per calcolo della soglia da applicare, però si è preferito cambiare completamente approccio ragionando in altri termini e scartando questa prima soluzione che è risultata essere troppo fragile per il mio scopo. Si è optato di adoperare un'algoritmo che fosse in grado di riconoscere i contorni dei caratteri e simboli: l'algoritmo che svolge con successo questo compito è il Canny Edge Detection.

1.3 Canny Edge Detection

Canny Edge Detection[3] è un'algoritmo per il riconoscimento dei contorni; esso produce edge connessi che possono essere efficacemente utilizzati per successive fasi di elaborazione. Utilizza un metodo di calcolo multi-stadio per individuare contorni presenti nelle immagini e nel mentre è in grado di attenuare il più possibile il rumore. Sono presenti diversi algoritmi per il riconoscimento dei contorni, come ad

esempio algoritmi che adoperano gli operatori di Sobel o Prewitt. Lo scopo di Canny però era quello di trovare il miglior algoritmo possibile per riconoscere i contorni delle immagini. In questo contesto "migliore" può significare:

- *buon riconoscimento*: l'algoritmo deve individuare e "marcare" quanti più contorni possibile nell'immagine;
- *buona localizzazione*: i contorni marcati devono essere il più vicini possibile ai contorni reali dell'immagine;
- *risposta minima*: un dato contorno dell'immagine deve essere marcato una sola volta, e, se possibile, il rumore presente nell'immagine non deve provocare il riconoscimento di falsi contorni.

L'approccio prevede le seguenti fasi:

1. Smoothing Gaussiano;
2. Calcolo del gradiente;
3. Soppressione dei non-massimi in direzione ortogonale all'edge;
4. Selezione degli edge significativi mediante isteresi;

1.3.1 Smoothing Gaussiano

Nella prima fase l'immagine viene sottoposta ad un'operazione di convoluzione con un filtro pesato secondo una funzione gaussiana in modo da nascondere piccole imperfezioni e brusche variazioni di luminosità. Il risultato è un'immagine con una leggera "sfocatura" gaussiana, in cui nessun singolo pixel è affetto da disturbi di livello significativo.
Mostriamo nell'immagine seguente il risultato di questa operazione utilizzando un filtro di grandezza 5 x 5 applicato all'immagine (b) della Figura 2.1:

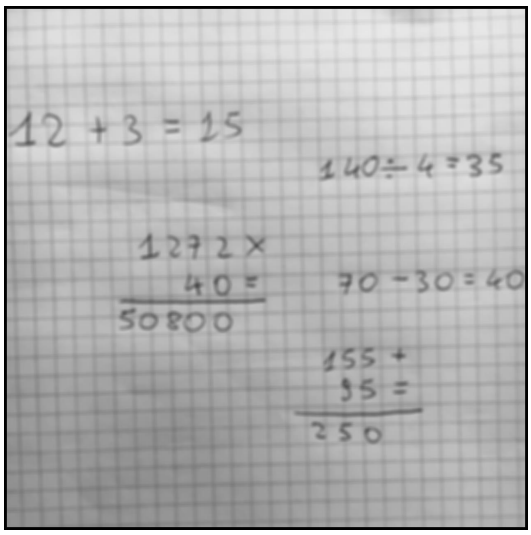


Figura 1.7: Risultato dell'operazione di smoothing Gaussiano applicato all'immagine (b) in Figura 2.1. Da notare la leggera sfocatura.

1.3.2 Calcolo del gradiente

Per il calcolo del gradiente, dato che lo smoothing Gaussiano dovrebbe aver rimosso la maggior parte del rumore, l'implementazione più efficiente si avvale degli operatori di Roberts. Risultano tuttavia di più semplice applicazione gli operatori di Sobel, in quanto non ci si deve preoccupare della rotazione degli assi di 45° che gli operatori di Roberts richiedono.
L'operatore calcola il gradiente della luminosità dell'immagine in ciascun punto, trovando la direzione lungo la quale si ha il massimo incremento possibile dal chiaro allo scuro, e la velocità con cui avviene il cambiamento lungo questa direzione. Il risultato ottenuto fornisce una misura di quanto "bruscamente" oppure "gradualmente" l'immagine cambia in quel punto, e quindi della probabilità che quella parte di immagine rappresenti un contorno; fornisce inoltre un'indicazione del probabile orientamento di quel contorno. Il gradiente di una funzione di due variabili è, in ciascun punto dell'immagine, un vettore bi-dimensionale le cui componenti sono le derivate del valore della luminosità in direzione orizzontale e verticale. In ciascun punto dell'immagine questo vettore gradiente punta nella direzione del massimo possibile aumento di luminosità e la lunghezza del vettore

corrisponde alla rapidità con cui la luminosità cambia spostandosi in quella direzione. Ciò significa che nelle zone dell'immagine in cui la luminosità è costante l'operatore di Sobel ha valore zero, mentre nei punti in prossimità dei contorni è un vettore orientato attraverso il contorno che punta nella direzione in cui si passa da valori di scuro a valori di chiaro.
Mostriamo nella seguente figura il risultato di questa fase:

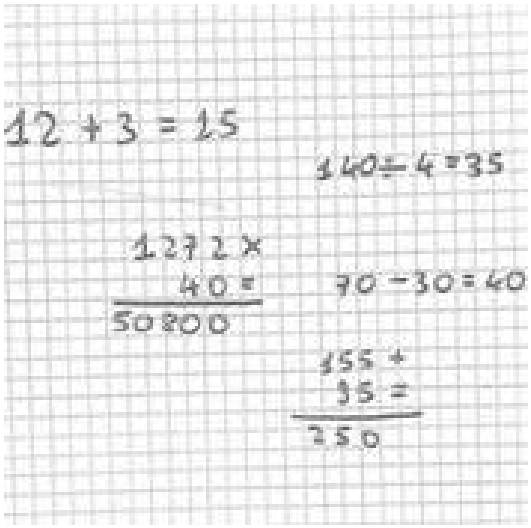


Figura 1.8: Modulo del gradiente calcolato utilizzando gli operatori di Sobel applicato all'immagine in Figura 2.7.

1.3.3 Soppressione dei non-massimi in direzione ortogonale all'edge

L'obiettivo di questa fase è eliminare dall'immagine modulo del gradiente i pixel che non sono massimi locali rispetto all'orientazione del gradiente.
La mappa dei gradienti fornisce il valore dell'intensità luminosa in ciascun punto dell'immagine. Una forte intensità indica una forte probabilità della presenza di un contorno. Tuttavia, questa indicazione non è sufficiente a decidere se un punto corrisponde oppure no ad un contorno. Solo i punti corrispondenti a dei massimi locali sono considerati come appartenenti ad un contorno e solo questi saranno presi in considerazione dai successivi step di elaborazione. Un massimo

locale si ha nei punti in cui la derivata del gradiente si annulla. Per eliminare i pixel che non sono massimi locali rispetto all'orientazione del gradiente, si stima il modulo del gradiente nei punti p1 e p2 mediante interpolazione lineare. La seguente figura mostra l'interpolazione nel caso l'orientazione del gradiente appartenga al primo ottante; gli altri casi sono analoghi.

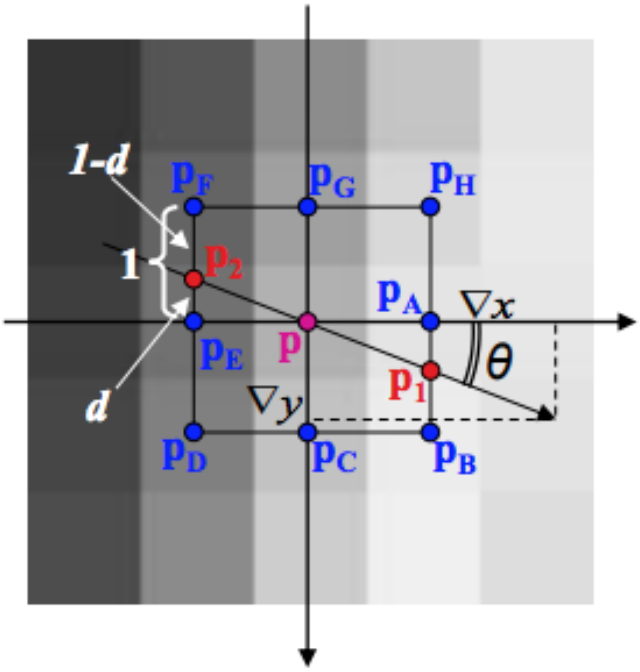


Figura 1.9: Interpolazione nel caso l'orientazione del gradiente appartenga al primo ottante.

Il modulo del gradiente nei punti *p1* e *p2* e la distanza d vengono calcolati nel seguente modo:

$$\|\nabla[\, p_1 \,]\| \cong d \cdot \|\nabla[\, p_{\mathrm{B}} \,]\| + (1 - d) \cdot \|\nabla[\, p_{\mathrm{A}} \,]\|$$

$$\|\nabla[\, p_2 \,]\| \cong d \cdot \|\nabla[\, p_{\mathrm{F}} \,]\| + (1 - d) \cdot \|\nabla[\, p_{\mathrm{E}} \,]\|$$

$$d = \frac{\nabla y[p]}{\nabla x[p]}$$

Il pixel p viene conservato solo è verificata la seguente condizione:

$$\|\nabla[\, p \,]\| \geq \|\nabla[\, p_1 \,]\| \wedge \|\nabla[\, p \,]\| \geq \|\nabla[\, p_2 \,]\|$$

Questo approccio non garantisce edge di spessore unitario (benché in genere lo siano). A tal fine può essere utilizzata una procedura di thinning al termine dell'intero algoritmo.
Mostriamo nella seguente figura il risultato di questa fase:

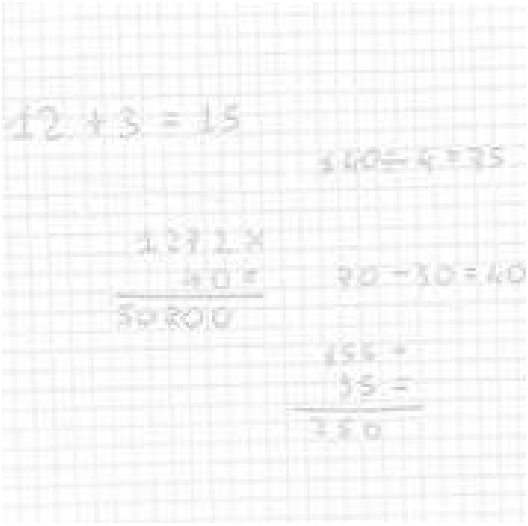


Figura 1.10: Risultato della soppressione dei non-massimi in direzione ortogonale all'edge.

1.3.4 Selezione degli edge significativi mediante isteresi

Al fine di selezionare solo gli edge significativi (tralasciando edge "spuri"), ma evitando allo stesso tempo la frammentazione, si utilizza il concetto di i*steresi*: vengono impiegate due soglie T1 e T2 (con T1>T2) per scremare ulteriormente i massimi locali ottenuti nella fase precedente:

- sono inizialmente considerati validi solo i pixel in cui il modulo del gradiente è superiore a T1;

- i pixel il cui modulo è inferiore a T1 ma superiore a T2 sono considerati validi *solo se adiacenti* a pixel validi.

Le soglie T1 e T2 sono tipicamente espresse come valori fra 0 e 1 (il modulo del gradiente va normalizzato nello stesso intervallo per permettere il confronto con le due soglie). Una corretta scelta di T1 e T2, così come un'adeguata scelta dell'ampiezza del filtro rappresentate una funzione aussiana nella prima fase, sono molto importanti per ottenere gli effetti desiderati. La scelta delle soglie comunque dipende solitamente dall'applicazione e sono tipicamente necessari vari esperimenti per giungere ai valori ottimali dei parametri. Facendo vari esperimenti, si è trovato un set-up soddisfacente scegliendo i seguenti valori per le soglie:

T1 = 0.1
T2 = 0.4

Mostriamo nella seguente figura il risultato finale dell'algoritmo:

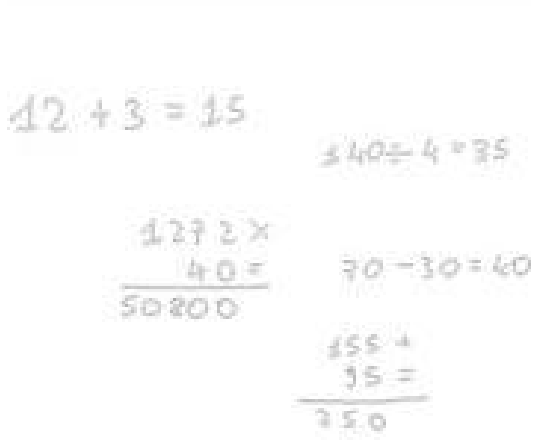


Figura 1.11: Risultato finale dell'algoritmo selezionando gli edge significativi mediante isteresi.

Come si può notare, grazie ad una appropriata scelta della grandezza del filtro gaussiano e delle soglie, siamo riusciti a selezionare con successo i bordi dei caratteri delle operazioni matematiche con una

percentuale molto bassa di rumore eliminando i quadretti. A differenza della binarizzazione, questo algoritmo funziona efficacemente anche con immagini che hanno brusche variazioni di luminosità, immagini troppo scure o immagini troppo chiare.
Si è deciso per cui di adottare questo algoritmo per la segmentazione dei caratteri e delle operazioni matematiche.

Per l'utilizzo dell'intero algoritmo Canny Edge Detection si è fatto uso della libreria GPUImage2. Di seguito si riporta il codice che implementa l'algoritmo:

```
public class CannyEdgeDetection: OperationGroup {
    public var blurRadiusInPixels:Float = 2.0 {
        didSet {
            gaussianBlur.blurRadiusInPixels =
                    blurRadiusInPixels
        }
    }
    public var upperThreshold:Float = 0.4 {
        didSet {
            directionalNonMaximumSuppression
            .uniformSettings["upperThreshold"] =
                    upperThreshold
        }
    }
    public var lowerThreshold:Float = 0.1 {
        didSet {
            directionalNonMaximumSuppression
            .uniformSettings["lowerThreshold"] =
                    lowerThreshold
        }
    }

    let luminance = Luminance()
    let gaussianBlur = SingleComponentGaussianBlur()
    let directionalSobel = TextureSamplingOperation(
    fragmentShader:
        DirectionalSobelEdgeDetectionFragmentShader)
    let directionalNonMaximumSuppression =
    TextureSamplingOperation(
          vertexShader:OneInputVertexShader,
            fragmentShader:
       DirectionalNonMaximumSuppressionFragmentShader)
    let weakPixelInclusion = TextureSamplingOperation(
    fragmentShader: WeakPixelInclusionFragmentShader)
```

```
35
36      public override init() {
37          super.init()
38
39          ({blurRadiusInPixels = 2.0})()
40          ({upperThreshold = 0.4})()
41          ({lowerThreshold = 0.1})()
42
43          self.configureGroup{input, output in
44          input --> self.luminance
45                  --> self.gaussianBlur
46                  --> self.directionalSobel
47                  --> self.directionalNonMaximumSuppression
48                  --> self.weakPixelInclusion --> output
49          }
50      }
51  }
```

Listing 1.1: Algoritmo Canny Edge Detection utilizzando la libreria GPUImage2 con linguaggio di programmazione Swift 3.

1.4 Segmentazione intera operazione matematica

Riusciti con successo a segmentare i diversi caratteri e simboli delle operazioni matematiche, il successivo problema sta nel capire, per ogni carattere e simbolo, a quale operazione matematica appartiene. Il problema principale stava nel capire quando una operazione terminava e quando un'altra iniziava per non rischiare di categorizzare più operazioni diverse come un'unica operazione. Per riuscire a risolvere questo problema, si sono analizzati diversi quaderni di matematica di diversi bambini della scuola primaria per vedere se questi avessero una struttura in comune che mi avrebbe permesso di scrivere un'algoritmo ad hoc. Si è notato che lo spazio che intercorreva tra la fine di una operazione e l'inizio di un'altra era abbastanza consistente, in particolare almeno un quadretto di 5 millimetri. Questa informazione mi ha permesso di scrivere un algoritmo in grado dividere prima orizzontalmente le diverse operazioni matematiche (basandosi sul fatto che una operazione è staccata da un'altra per almeno un determinato numero di pixel precalcoalto) e poi verticalmente. Prima di procedere con l'algoritmo

si è binarizzata l'immagine ottenuta dall'algoritmo di Canny Edge Detection in Figura 2.11.

1.4.1 Segmentazione orizzontale

Con la segmentazione orizzontale si intende riuscire ad arrivare a questo risultato mostrato in figura:

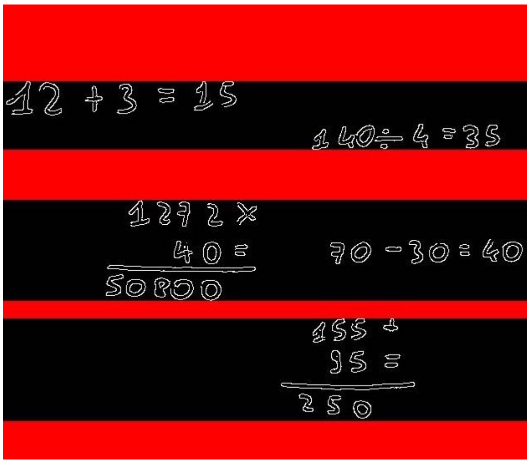


Figura 1.12: Risultato dell'algoritmo di segmentazione orizzontale.

Per riuscire ad arrivare a questo risultato si sono fatte *due assunzioni*:

1. un'operazione deve essere distante da un'altra operazione, a nord e sud di essa, per almeno una soglia precalcolata e la foto deve essere scattata ad una determinata distanza dal foglio (distanza che dipende dalla soglia precalcolata);

2. la foto deve essere scattata senza inquadrare i bordi del foglio.

Per riuscire ad arrivare al risultato in Figura 2.12, l'idea è stata prima di tutto di andare a scorrere ogni riga dell'immagine e per ognuna sommare la quantità di pixel di foreground: pixel bianchi, ovvero pixel che appartengono ad un operazione matematica.
L'output di questa operazione è mostrato di seguito:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 10, 24, 15, 23, 15, 32, 21, 26, 38, 23, 16, 18, 27, 31, 18, 18, 17, 30, 34, 25, 26, 35, 34, 36, 22, 23, 18, 23, 20, 20, 24, 37, 31, 16, 11, 22, 27, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 8, 26, 21, 15, 25, 38, 33, 31, 27, 34, 21, 28, 39, 58, 36, 38, 39, 50, 32, 39, 27, 26, 36, 25, 29, 22, 24, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 4, 16, 13, 20, 18, 17, 20, 25, 23, 20, 19, 19, 20, 17, 17, 16, 21, 17, 16, 17, 20, 25, 24, 21, 25, 13, 16, 22, 23, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 13, 20, 33, 31, 44, 42, 46, 53, 71, 48, 42, 42, 61, 50, 47, 44, 50, 48, 50, 39, 47, 27, 27, 28, 41, 27, 26, 22, 32, 94, 25, 12, 34, 59, 53, 0, 0, 7, 8, 4, 4, 12, 28, 31, 26, 21, 20, 35, 31, 29, 29, 22, 18, 19, 19, 20, 29, 24, 23, 26, 39, 31, 14, 9, 11, 14, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 19, 8, 7, 10, 16, 16, 12, 19, 16, 13, 16, 19, 25, 21, 15, 10, 7, 10, 14, 14, 15, 8, 11, 11, 12, 6, 4, 0, 0, 0, 0, 0, 0, 0, 0, 5, 11, 10, 6, 7, 22, 14, 7, 7, 15, 16, 8, 10, 9, 13, 17, 8, 6, 12, 19, 12, 11, 6, 8, 13, 7, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 40, 42, 60, 6, 22, 47, 29, 45, 0, 0, 0, 0, 8, 5, 15, 6, 8, 11, 14, 7, 16, 10, 12, 10, 15, 15, 14, 17, 13, 9, 15, 13, 16, 10, 19, 4, 4, 5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Gli zeri indicano che la riga i-esima non contiene pixel di foreground, i numeri diversi da zero invece indicano che nella riga i-esima sono presenti quel determinato numero di pixel di foreground. Essendo che il numero specifico della quantità di pixel di foreground non era rilevante in quanto solamente gli zeri ci interessano per la segmentazione orizzontale, si sono sostituiti tutti i numero diversi da zero con 1, con semantica che in quella riga sono presenti pixel di foreground. Il risultato è mostrato di seguito:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Una volta ottenuto questo output, l'idea per capire quali righe fossero candidate per essere colorate di rosso, e quindi per la segmentazione orizzontale, è stata trovare se in questo array fossero presenti un determinato numero di zeri contigui e, nel caso fossero presenti, allora i pixel delle righe rappresentate da questi zeri contigui venivano prese in cosiderazione per essere colorate di rosso.
Per semplicità visiva e per avere una struttura dati più contenuta, si è trasformato l'output precedente nel seguente:

[(82, 0), (39, 1), (13, 0), (29, 1), (54, 0), (31, 1), (15, 0), (36, 1), (2, 0), (30, 1), (20, 0), (28, 1), (8, 0), (27, 1), (9, 0), (9, 1), (4, 0), (28, 1), (34, 0)]

Questo array di punti (x, y) è stato ricavato in questo modo:
i primi elementi dell'array precedente sono 82 zeri contigui, allora al posto di scrivere tutti gli 82 zeri, scrivo che ci sono 82 elementi contigui e questi sono zeri $\longrightarrow$ (82, 0). Successivamente sono presenti 39 uni contigui $\longrightarrow$ (39, 1) e così via.
Questo array non è però ancora quello definitivo che mi serve per andare a disegnare le "strisce rosse" per la segmentazione orizzontale in quanto presenta del rumore. Si esamini il caso dell'operazione in colonna più a sud dell'immagine in Figura 2.12: essa è rappresentata dal seguente array:

$(20, 0)$, (28, 1), (8, 0), (27, 1), (9, 0), (9, 1), (4, 0), (28, 1), $(34, 0)$

I numeri di colore rosso rappresentano le due strisce a nord dell'operazione presa in considerazione ($(20, 0)$) e a sud di essa ($(34, 0)$).

E' evidente che tutti i punti in mezzo alle due strisce rosse rappresentano un'intera operazione matematica. È necessario per cui riuscire a capire, analizzando l'array, quali sono questi punti. L'idea è andare a trovare tutti quei valori di background che sono maggiori o uguali alla soglia calcolata precedentemente (punto 1 delle *assuzioni*). Una volta trovato nell'array un valore che rispetta questa condizione, allora si disegna la striscia orizzontale, mentre tutti gli altri valori minori di questa soglia vengono considerati come porzioni di una operazione matematica. Nell'esempio che stiamo considerando la soglia è di 16. Infatti, i valori rossi hanno un valore maggiore di 16, mentre tutti quelli in mezzo minore (questo per quanto riguarda i valori di background, ovvero i valori con il parametro y uguale a 0).
Essendo che i valori minori della soglia vengono etichettati come "pixel che appartengono ad una possibile operazione matematica", allora se ne è fatto un "merge" sommando tutti i punti x, eliminando i vecchi punti e creando un nuvo punto che rappresentasse la loro somma ed attribuendolo al foregorund semplicemente scrivendo 1 nel campo y del punto. Esempio di questo passaggio e mostrato di seguito:

(28, 1), (8, 0), (27, 1), (9, 0), (9, 1), (4, 0), (28, 1) $\longrightarrow$ (113, 1)

Si è proseguito applicando questo algoritmo a tutti i punti dell'array rappresentate l'immagine. Il risultato ottenuto è il seguente array di punti:

[(82, 0), (81, 1), (54, 0), (114, 1), (20, 0), (113, 1), (34, 0)]

Da notare che la sommatoria di tutti i campi x dei punti (x, y) dell'array rappresenta l'altezza dell'immagine. Si è strutturato volontariamente l'array in questo modo perchè così, quando si vanno a disegnare le strisce rosse per la segmentazione orizzontale, si è già in possesso delle coordinate di dove andare a disegnare sull'immagine ma soprattutto mi sarà utile quando andrò a ritagliare ogni operazione segmentata. Non solo, anche la posizione dei punti nell'array è importante. Si nota subito infatti che, guardando l'array, si riesce subito a capire che, per esempio, la seconda striscia rossa sull'immagine è rappresentata dal punto (54, 0).

1.4.2 Segmentazione verticale

Con segmentazione verticale si intende riuscire ad arrivare a questo risultato mostrato in figura:

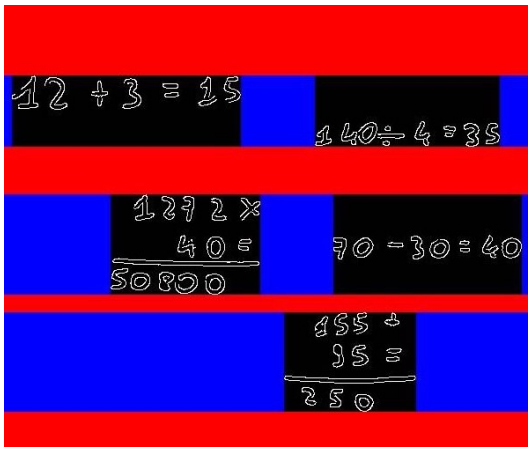


Figura 1.13: Risultato dell'algoritmo di segmentazione verticale.

Per riuscire ad ottenere il risultato mostrato in Figura 2.13, l'idea è stata di riapplicare l'algoritmo di segmentazione orizzontale per ciascuna zona di foreground ruotandola prima di 90°. Il risultato però non è sempre soddisfacente a causa della soglia cambalta per l'algoritmo di segmentazione orizzontale. In particolare, la soglia va aumentata in quanto la distanza che intercorre tra un numero e un simbolo ($+$, $-$, $=$, ...) può essere maggiore della distanza minima che può intercorrere tra una intera operazione matematica ed un'altra. In blu, nella FIgura 2.13, viene mostrato l'output di questo passaggio.
Una volta segmentate le intere operazioni matematiche, siamo pronti a ritagliare le operazioni dell'immagine originale ed utilizzare le API di Mathpix: piattaforma in grado di riconoscere caratteri e simboli dell'immagine di una operazione matematica passatagli in input. L'algoritmo di segmentazione delle intere operazioni matematiche funziona correttamente ma solamente sotto le due *assuzioni* fatte a pagina 21. Vedremo nel prossimo capitolo, grazie alla realtà aumentata, come riuscire a far funzionare l'algoritmo correttamente senza il primo vincolo imposto dall'assuzione (1) eliminando le soglie fisse e convertendole in soglie dinamiche. Per quanto riguarda il secondo

vincolo delle assuzioni, il problema è stato deciso di risolverlo nei lavori futuri.

1.5 Mathpix

Mathpix è un sistema di riconoscimento ottico dei caratteri, detto anche OCR (Optical Character Recognition): sono programmi dedicati al rilevamento dei caratteri contenuti in un documento o un'immagine e al loro trasferimento in testo digitale leggibile da una macchina. Segmentate le intere operazione matematiche e avendo le loro coordinate, si è riusciti a ritagliare dall'immagine originale tutte le diverse operazioni matematiche. In seguito ognuna di queste viene passata in input a Mathpix per il riconoscimento.
Il codice utilizzato per il riconoscimento dei caratteri e simboli dell'operazione matematica, usufruendo delle API di Mathpix, è il seguente:

```
1  MathpixClient.recognize(image: croppedImage!,
2                          outputFormats:
3                          [FormatLatex.simplified,
4                           FormatWolfram.on]) {
5                                (error, result) in
6                                self.observerOperation =
7                                String(result.debugDescription)
8                           }
```

Listing 1.2: Utilizzo dell'API *recognize* di Mathpix per il riconoscimento di caratteri e simboli presenti nell'immagine *croppedImage* passatagli in input.

- `image:` indica l'immagine in input;
- `outputFormats:` indica in che formato si vuole che il risultato sia ricevuto;
- `result.debugDescription:` è la risposta di Mathpix con il riconoscimento in formato JSON.

Considerando di ritagliare dall'immagine originale (ovvero quella scattata dal dispositivo mobile) la prima operazione in alto a sinsitra della Figura 2.13 ($12 + 3 = 15$), si ottiene il seguente output:

```
1  {
2      "detection_list" =      (
3      );
4      "detection_map" =      {
5          "contains_chart" = 0;
6          "contains_diagram" = 0;
7          "contains_graph" = 0;
8          "contains_table" = 0;
9          "is_blank" = "0.0001";
10         "is_inverted" = 0;
11         "is_not_math" = "0.0001";
12         "is_printed" = "0.0005";
13     };
14     error = "";
15     latex = "12 + 3 = 15";
16     "latex_confidence" = "0.995189821877";
17     "latex_confidence_rate" = "0.9994655251502";
18     "latex_list" =      (
19         "12 + 3 = 15"
20     );
21     position =      {
22         height = 40;
23         "top_left_x" = 0;
24         "top_left_y" = 0;
25         width = 259;
26     };
27     wolfram = "12 +3= 15";
28 }
```

Come si può notare, in linea `19` è presente il risultato in formato `latex` mentre in linea `27` è presente lo stesso risultato in formato `wolfram`. Un'altro dato interessante è il `latex_confidence` a linea `16` che indica la probabilità (espressa da 0 a 1) che il risultato da lui restituito sia corretto. Una problematica riscontrata è stata che questo risultato del riconoscimento poteva arrivare in qualsisi momento imprevedibile in maniera asincorina essendo che il sofware di Mathpix è in esecuzione su server esterni. Per risolvere questo problema, si è creato un buffer che implementasse il pattern Observer, in modo tale che, quando il server

di Mathpix riempiva questo buffer con la sua risposta, il buffer stesso notificava chi di dovere che la risposta da Mathpix è arrivata e quindi pronta per essere processata.
Un'altra problematica è che i server di Mathpix introducono un ritardo non indifferente per processare l'immagine e restituire il risultato. In particolare quando il numero di operazioni da processare inizia ad essere consitente, il server ci mette fino a qualche secondo per processarle tutte. Si è deciso comunque di utilizzare questo OCR per il riconoscimento delle operazioni e di risolvere questo problema nei lavori futuri (rendendo l'applicazione indipendete da OCR esterni).
Una volta riconosciuti i caratteri grazie a Mathpix, si sono scritti dei parser personalizzati in modo tale che estrapolassero dal risultato in formato JSON il risultato del riconoscimento dell'operazione in modo automatico. Nell'esempio precedente, essendo che l'operazione è semplice e piana, l'output è diretto: $12 + 3 = 15$. Quando si tratta invece di operazioni in colonna il parsing potrebbe non essere così diretto. Mostriamo per esempio il risultato `wolfram` del riconoscimento dell'operazione in colonna 1272 x 40 = 50800 nella Figura 2.13:

```
wolfram = "{ { 1272 x }, { 40 = } , { 50800} }";
```

Come si può notare il risultato contiene parentesi graffe e virgole che rendono la scrittura del parser più complicata. Una volta reso automatico il processo di parsing anche per le operazioni in colonna, avendo le coordiante di ogni operazione segmentata, queste vengono analizzate per capire se sono corrette o meno.
Analizzate le operazioni, avendo le coordinate per ogni operazione e sapendo se sono corrette o meno, si disegna sull'immagine originale, per ogni operazione, un simbolo che rappresenti la correttezza o meno di quest'ultima.
Mostriamo nella seguente immagine il risultato dell'intero algoritmo:

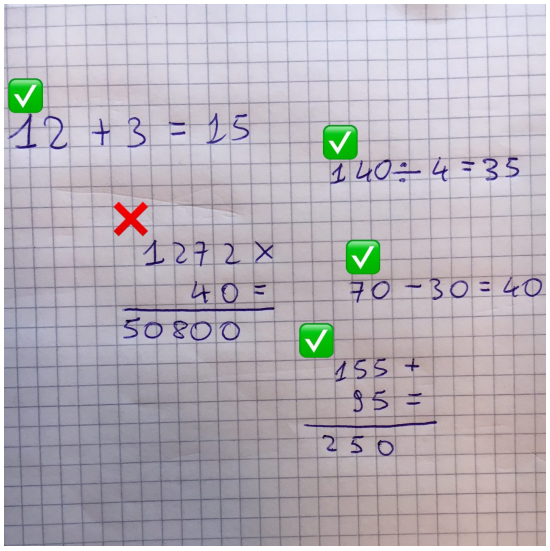


Figura 1.14: Risultato finale dell'algoritmo.

Capitolo 2

Realtà aumentata

2.1 Definizione

Con realtà aumentata, o dall'inglese Augmented Reality (AR), si vuole descrivere la tecnologia che permette la sovrapposizione di livelli informativi alla realtà direttamente o indirettamente visualizzata. In altre parole essa può essere descritta dal concetto di Mixed Reality nel quale realtà e oggetti virtuali vengono combinati in tempo reale all'interno di uno spazio tridimensionale.

2.2 Problema delle soglie fisse

L'algoritmo precedente funziona correttamente ma solamente sotto le assunzioni fatte, ovvero che un'operazione deve essere distante da un'altra operazione per almeno una soglia precalcolata e la foto deve essere scattata (senza inquadrare il bordi del foglio) ad una determinata distanza dal foglio, distanza che dipende dalla soglia precalcolata. Quindi ad esempio se la soglia è x, per il corretto funzionamento dell'algoritmo, la foto dovrà essere scattata ad una precisa distanza y dal foglio. Se questo vincolo non viene rispettato, si può incorrere nel seguente problema:

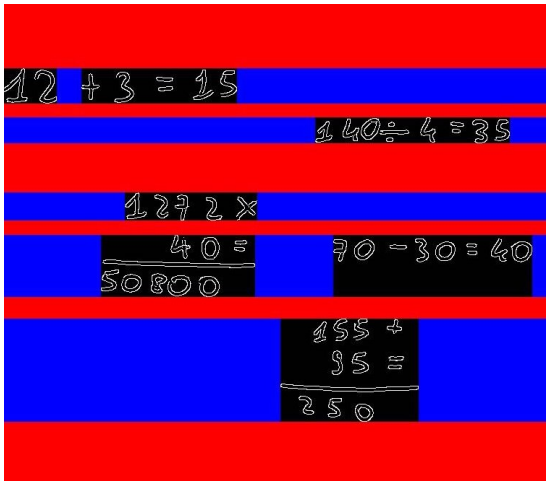


Figura 2.1: Risultato dell'algoritmo se si scatta la foto senza rispettare il vincolo imposto dalle soglie.

Come si può notare, le due operazioni più a sinistra in Figura 3.1 vengono segmentate in modo anomalo. In particolare l'operazione 12 + 3 = 15 viene interpretata come due operazioni separate: una operazione composta dal "12" ed un'altra operazione composta da "+3 = 15". Stessa situazione per l'operazione in colonna 1272 x 40 = 50800. E' evidente che il vincolo imposto dalle soglie fisse rende l'algoritmo troppo fragile. Si vuole ovviamente che l'algoritmo funzioni correttamente anche scattando foto a distanze differenti. Quindi è necessario un meccanismo che sia in grado di calcolare la distanza che il nostro dispositivo ha dal foglio al momento dello scatto per poi aggiustare di conseguenza la soglia. Per risolvere questo problema ci viene in aiuto la realtà aumentata.
Per poter fare utilizzo della realtà aumentata si è usufruito della libreria ARKit di Apple.

2.3 Rilevamento superficie orizzontale

Per riuscire a calcolare la distanza nel mondo reale che il dispositivo ha dal foglio instante per instante, è necessario prima di tutto riuscire a fare un rilevamento del piano su cui il foglio a cui scattare la foto è poggiato. Per riuscire a rilevare i piani orizzontali è necessario configurare opportunamente l'`ARSCNView`: una classe che permette di

creare, posizionare oggetti aumentati e catturare caratteristiche nel mondo reale. Riportiamo di seguito la configurazione utilizzata:

```
1     @IBOutlet weak var sceneView: ARSCNView!
2     private let configuration =
3                           ARWorldTrackingConfiguration()
4
5     override func viewDidLoad() {
6         super.viewDidLoad()
7         self.sceneView.debugOptions =
8         [ARSCNDebugOptions.showWorldOrigin,
9         ARSCNDebugOptions.showFeaturePoints]
10        self.configuration.planeDetection = .horizontal
11        self.sceneView.session.run(configuration)
12        self.sceneView.delegate = self
13    }
```

Listing 2.1: Configurazione per il rilevamento di superfici orizzontali nel mondo reale.

L'oggetto `sceneView` di tipo `ARSCNView` in linea `1` rappresenta la classe che permette di creare, posizionare oggetti aumentati e catturare caratteristiche nel mondo reale. In linea `9` viene specificato che si vogliono, tra le altre cose, catturare le informazioni relative alle superfici orizzontali. Se si volessero catturare le informazioni anche delle superfici verticali, basterebbe sostituire la riga `9` con la seguente riga mostrata di seguito:

```
1  self.configuration.planeDetection =
2                                   [.horizontal, .vertical]
```

Listing 2.2: Configurazione per il rilevamento di superfici orizzontali e verticali nel mondo reale.

2.4 Calcolo della distanza nel mondo reale

ARKit ci permette di calcolare la distanza (nel mondo reale) che intercorre fra due nodi oppure fra il dispositivo mobile stesso e un'altro nodo. Un nodo è un elemento strutturale di un grafo di scene, che rappresenta una posizione e si trasforma in uno spazio di coordinate 3D a cui è possibile collegare geometrie, luci, telecamere o altri contenuti visualizzabili. Sapendo che si può fare ciò allora l'idea è stata di posizionare sul foglio (dopo aver fatto il rilevamento del piano

orizzontale su cui il foglio poggia) un nodo invisibile all'utente e misurare istante per istante la distanza che intercorreva fra questo nodo e il dispositivo. Essendo che il posizionamento del nodo invisibile su foglio è stato demandato all'utente tramite un click sullo schermo, è stato necessario rendere lo schermo reattivo ai click. Per fare ciò bisogna aggiungere un `UITapGestureRecognizer` alla `sceneView`. Quindi nel codice precedente della `viewDidLoad` è necessario aggiungere anche il seguente codice:

```
1  override func viewDidLoad() {
2      ...
3      self.registerGestureRecognizerForPlaneAndDistance()
4      ...
5  }
6
7  private func registerGestureRecognizerForPlaneAndDistance
8                                                          () {
9      let tapGestureRecognizer =
10      UITapGestureRecognizer(
11        target: self,
12        action: #selector(tappedForPlaceAndDistance))
13        self.sceneView.addGestureRecognizer(
14                                        tapGestureRecognizer)
15 }
```

Listing 2.3: Codice per rendere reattivo lo schermo quando questo viene toccato.

Una volta reso lo schermo è reattivo, quando l'utente tocca lo schermo, viene invocato il metodo `tappedForPlaceAndDistance`. Questo metodo si occuperà di collocare il nodo invisibile sul foglio. Il codice del metodo `tappedForPlaceAndDistance` e dei campi che utilizza è riportato di seguito:

```
1
2      ...
3
4      private var startingPosition: SCNNode?
5      private var isStartingPositionPlaced = false
6
7      ...
8
9      @objc private func tappedForPlaceAndDistance(
10                         sender: UITapGestureRecognizer) {
11         if !self.isStartingPositionPlaced {
```

```
12              let sceneView = sender.view as! ARSCNView
13              let tapLocation = sender.location(
14                                              in: sceneView)
15              let hitTest =
16                  sceneView.hitTest(tapLocation, types:
17                    .existingPlaneUsingExtent)
18              if !hitTest.isEmpty {
19                  addItem(hitTestResult: hitTest.first!)
20                  self.isStartingPositionPlaced = true
21              }
22          }
23      }
24
25      private func addItem(hitTestResult: ARHitTestResult) {
26          let transform = hitTestResult.worldTransform
27          let thirdCol = transform.columns.3
28          let node = SCNNode()
29          self.startingPosition = node
30          node.position = SCNVector3(thirdCol.x, thirdCol.y,
31                                                  thirdCol.z)
32          self.sceneView.scene.rootNode.addChildNode(node)
33      }
```

Listing 2.4: Codice per l'aggiunta del nodo invisibile sulla superficie orizzontale rilevata.

Il campo `startingPosition` rappresenta il nostro nodo invisibile. `isStartingPositionPlaced` è un campo che tiene traccia del fatto che il nodo invisibile sia stato piazzato o meno. Quindi quando lo schermo viene toccato, il metodo `tappedForPlaceAndDistance` viene invocato: se il nodo non è stato ancora piazzato (riga `10`), allora viene presa la posizione nel mondo reale di dove l'utente ha cliccato (riga `12`) e a seguire si controlla se l'utente ha cliccato sulla superficie orizzontale rilevata precedentemente (e quindi sul foglio) (riga `13`). Se effettivamente l'utente ha cliccato sulla superficie orizzontale rilevata allora si aggiunge il nodo invisibile attraverso il metodo `addItem` (righe `16 e 17`). Altrimenti se l'utente non ha cliccato sulla superficie orizzontale rilevata, il click sullo schermo viene ignorato. Una volta piazzato il nodo sul foglio, la seguente procedura entra in funzione e misura la distanza insante per istante dal nodo piazzato al dispositivo:

```
1
2   private func renderer(_ renderer: SCNSceneRenderer,
3                         updateAtTime time: TimeInterval) {
4       guard let startingPosition = self.startingPosition
```

```
5                                                   else {return}
6           guard let pointOfView = self.sceneView.pointOfView
7                                                   else {return}
8           let transform = pointOfView.transform
9           let location =
10             SCNVector3(transform.m41, transform.m42,
11                                                   transform.m43)
12          let xDistance =
13                    location.x - startingPosition.position.x
14          let yDistance =
15                    location.y - startingPosition.position.y
16          let zDistance =
17                    location.z - startingPosition.position.z
18          DispatchQueue.main.async {
19              return self.distanceTravelled(
20              x: xDistance,
21              y: yDistance,
22              z: zDistance)
23          }
24
25    }
26
27    private func distanceTravelled(x: Float, y: Float,
28                                                 z: Float) -> Float {
29        return sqrtf(x*x + y*y + z*z)
30    }
```

Listing 2.5: Codice per la misura della distanza dal dispositivo mobile al nodo posizionato sul foglio.

Ora che siamo in grado di misurare la distanza che intercorre tra il nostro dispositivo e il foglio, non resta che fare un mapping tra la distanza rilevata e il valore che le soglie devono assumere a quella distanza per il corretto funzionamento dell'algoritmo. Per fare ciò bisogna prima di tutto associare un valore di partenza alle soglie e poi regolarsi su quello. Il valore di partenza è stato scelto tenendo il dispositivo ad una distanza di 26 centimetri in modulo dal foglio. I valori di partenza ricavati a quella distanza dal foglio sono 5 per la soglia orizzontale e 7 per la soglia verticale. Questo significa che, scattando una foto alla distanza di 26 centimetri dal foglio, per una corretta segmentazione delle operazioni è necessario usare come soglia orizzontale il valore 5 e come soglia verticale il valore 7, valori che rappresentano la minima distanza che può intercorrere tra la fine di un'intera operazione matematice e l'inzio di un'altra. È necessario poi calcolare anche i valori che le

soglie devono assumere alla distanza minima a cui il dispositivo può scattare la foto. La distanza minima scelta è `4` centimetri in modulo dal foglio. A questa distanza i valori per la soglia orizzontale e verticale sono, rispettivamente, `32.50` e `45.50`. Avendo questi dati riusciamo a calcolare, ad ogni precisa distanza, qual è il valore che le soglie devono assumere. Di seguito mostriamo una tabella che indica, per distanza in centimetro del dispositivo dal foglio, il valore che assumono le soglie:

Distanza [cm]	Soglia orizzontale (+1.25)	Soglia verticale (+1.75)
26	5.00	7.00
25	6.25	8.75
24	7.50	10.50
23	8.75	12.25
22	10.00	14.00
21	11.25	15.75
20	12.50	17.50
19	13.75	19.25
18	15.00	21.00
17	16.25	22.75
16	17.50	24.50
15	18.75	26.25
14	20.00	28.00
13	21.25	29.75
12	22.50	31.50
11	23.75	33.25
10	25.00	35.00
9	26.25	36.75
8	27.50	38.50
7	28.75	40.25
6	30.00	42.00
5	31.25	43.75
4	32.50	45.50

Essendo che le soglie rappresentano un numero di pixel, è stato necessario fare una operazione di approssimazione arrotondando i numeri in virgola mobile all'intero più vicino.
Mostriamo di seguito due esempi di funzionamento del seguente algoritmo applicato all'algoritmo di segmentazione delle intere operazioni matematiche: nel primo esempio è stata scattata una foto

ad una distanza piuttosto lontana dal foglio (20 centimetri) mentre nel secondo esempio è stata scattata una foto una distanza piuttoto ravvicinata (6 centimetri).
Illustriamo il risultato che si otteneva precedentemente senza l'applicazione dell'algoritmo appena descritto e il risultato ottenuto dopo con l'applicazione di questo algoritmo:

Prima:

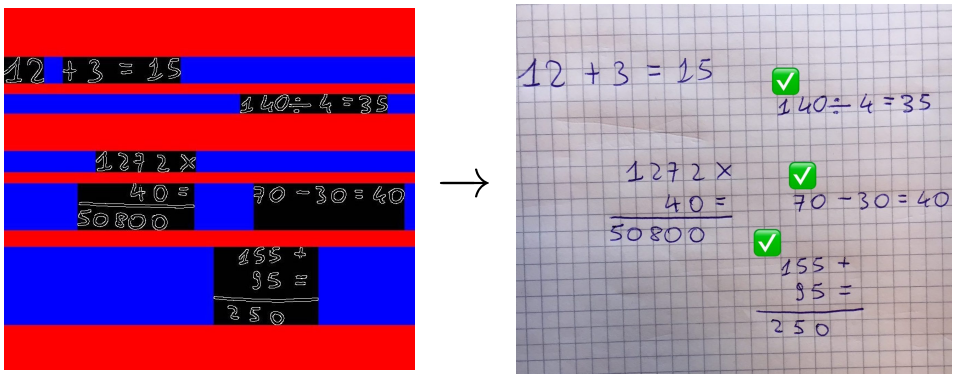


Figura 2.2: Applicazione dell'algoritmo con soglie *fisse* con il dispositivo mobile ad una distanza dal foglio di 20 cm.

Dopo:

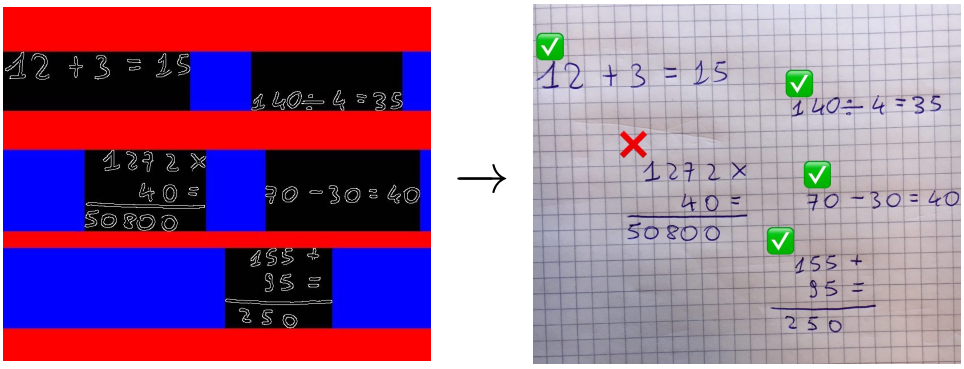


Figura 2.3: Applicazione dell'algoritmo con soglie *dinamiche* con il dispositivo mobile ad una distanza dal foglio di 20 cm.

Prima:

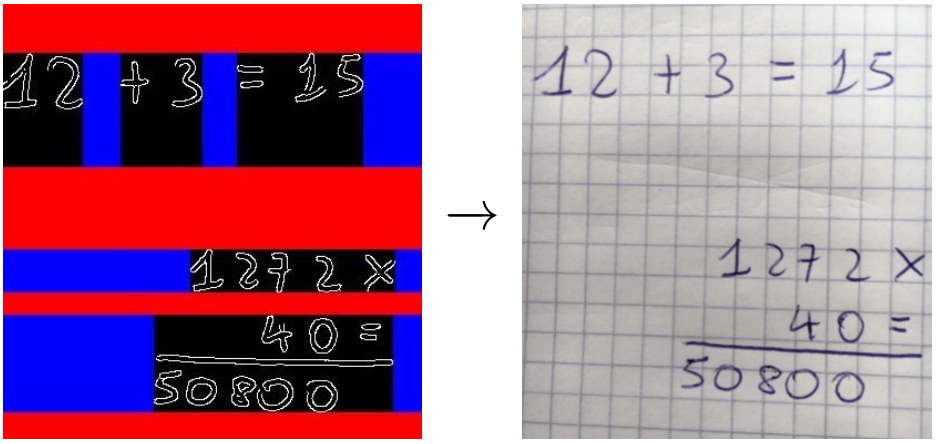


Figura 2.4: Applicazione dell'algoritmo con soglie *fisse* con il dispositivo mobile ad una distanza dal foglio di 6 cm.

Dopo:

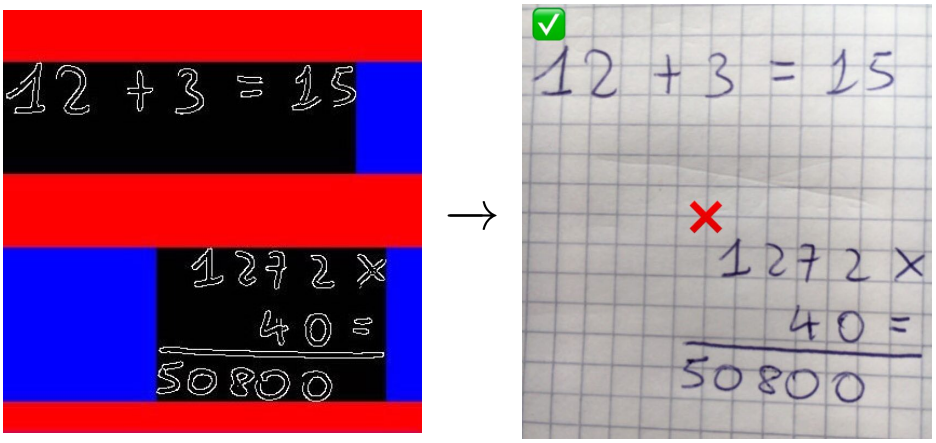


Figura 2.5: Applicazione dell'algoritmo con soglie *dinamiche* con il dispositivo mobile ad una distanza dal foglio di 6 cm.

Capitolo 3

Aspetti architetturali ed implementativi

In questo capitolo andremo a guardare nel dettaglio l'architettura scelta per l'applicazione, descrivendo i pattern utilizzati, il flusso di lavoro dell'applicazione con i relativi diagrammi, quali tecnologie sono state utilizzate e perchè e illustreremo il codice degli algoritmi più importanti utilizzati.

3.1 iOS

iOS è il sistema operativo creato e sviluppato da Apple esclusivamente per il suo hardware. È il sistema operativo che attualmente alimenta molti dei dispositivi mobili dell'azienda, inclusi iPhone, iPad e iPod Touch. E' attualmente il secondo sistema operativo per mobile più popolare in tutto il mondo dopo Android. Nonostante non sia il più popolare sistema operativo per mobile, per questo progetto è stato deciso di adottare questa piattaforma prima di tutto perchè non si conosceva il linguaggio Swift e quindi è stato un buon pretesto per impararlo, poi perchè iOS offriva librerie quali GPUImage2 e ARKit che svolgevao un buon lavoro per il mio scopo.

3.1.1 Swift

Swift[4] è un linguaggio di programmazione ad oggetti creato da Apple nel 2014. Sostenuto da una delle società tecnologiche più influenti al

mondo, Swift è destinato a diventare il linguaggio dominante per lo sviluppo di iOS e oltre.

Swift è:

- `open source:` i creatori di Swift hanno riconosciuto il fatto che per costruire un linguaggio di programmazione definitivo, la tecnologia deve essere aperta a tutti. Così, nei suoi tre anni di esistenza, Swift ha acquisito una grande comunità di supporto e un'abbondanza di strumenti di terze parti;

- `sicuro:` la sua sintassi ti incoraggia a scrivere codice pulito e coerente, anche se a volte può sembrare tedioso. Swift fornisce salvaguardie per prevenire errori e migliorare la leggibilità. Questo ha un forte impatto sulla sicurezza;

- `veloce:` Swift è stato costruito pensando alle prestazioni. Grazie alla sua semplice sintassi e tenuta manuale ti aiutano a sviluppare più velocemente;

- `popolare:` a partire da marzo 2018, è la 12a lingua più popolare, superando Objective-C, Go, Scala e R. Con oltre 40.000 stelle su GitHub e 187K StackOverflow, questa giovane lingua sta giustamente diventando una delle tecnologie dominanti nel settore.

3.1.2 Portabilità da iOS ad Android

La ragione per cui bisognerebbe portare l'applicazione da iOS ad Android è semplice: l'80 % dei dispositivi mobili (e non solo) in tutto il mondo hanno sistema operativo Android. Inoltre, la maggior parte dei dispositivi Android hanno un prezzo accessibile quindi molti dei potenziali clienti hanno un dispositivo con sistema operativo Android. Per portare una applicazione da iOS ad Android non si può semplicemente raccogliere e spostare elementi di design senza modifiche. Molti strumenti comuni forniscono modelli iOS per impostazione predefinita e si potrebbe pensare che questi si adattino anche alla piattaforma Android. In realtà non è così e qualsiasi tentativo diretto di trasferire l'app iOS su Android porta a un conflitto tra l'interfaccia dell'applicazione e del design della app iOS con l'interfaccia dell'applicazione Android. Bisogna ragionare su ogni

singolo componente della applicazione e trovare il modo duale per poterlo implemetare sulla applicazione Android. Quindi un porting diretto automatico non è possibile per applicazioni con un minimo livello di complessità. Per riuscire a portare la mia applicazione su Android le principali librerie da utilizzare sono la libreria per l'utilizzo della realtà aumentata e libreria che permetta di eseguire algoritmi di elaborazione delle immagini. Per la realtà aumentata, Android fornisce la piattaforma `ARCore` mentre per algoritmi di elaborazioni delle immagini, tra le varie librerie disponibili, si può utilizzare la libreria `openCV`.

3.2 Architettura MVC

L'architettura che è stata utilizzata per l'applicazione è l'MVC[5].
Il modello di progettazione Model-View-Controller (MVC) è piuttosto vecchio e ne esistono diverse varianti. Si tratta di uno schema di alto livello in quanto si occupa dell'architettura globale di un'applicazione e classifica gli oggetti in base ai ruoli generali che svolgono in un'applicazione. È anche un modello complesso in quanto comprende diversi modelli più elementari. I programmi orientati agli oggetti beneficiano in vari modi del modello di progettazione MVC adattandosi alle loro circostanze. I programmi in generale sono più adattabili ai mutevoli requisiti, in altre parole, sono più facilmente estendibili e, cosa più importante, manutenibili rispetto ai programmi che non sono basati su MVC.
Il modello di progettazione MVC considera tre tipi di oggetti: oggetti modello, oggetti view e oggetti controller. Il pattern MVC definisce i ruoli che questi tipi di oggetti giocano nell'applicazione e le loro linee di comunicazione. Quando si progetta un'applicazione, un passo importante è scegliere, per ogni calsse, a quale di queste tre categorie appartiene. Ognuno dei tre tipi di oggetti è separato dagli altri da confini astratti e comunica con gli oggetti degli altri tipi attraverso quei confini.

3.2.1 Model

Gli oggetti del *model* rappresentano conoscenze e competenze speciali. Conservano i dati di un'applicazione e definiscono la logica che manipola

tali dati. Un'applicazione MVC ben progettata ha tutti i dati incapsulati negli oggetti del modello. Qualsiasi dato che fa parte dello stato persistente dell'applicazione deve risiedere negli oggetti del modello una volta che i dati sono stati caricati nell'applicazione. Poiché rappresentano la conoscenza e le competenze relative a uno specifico dominio problematico, tendono a essere riutilizzabili.
Idealmente, un oggetto modello non ha alcuna connessione esplicita all'interfaccia utente utilizzata per presentarla e modificarla. Ad esempio, se si dispone di un oggetto modello che rappresenta una persona, è possibile che si desideri archiviare una data di nascita. Questa è una buona cosa da memorizzare nell'oggetto del modello `Persona`. In generale un oggetto modello non dovrebbe preoccuparsi dei problemi di interfaccia e presentazione.

3.2.2 View

Un oggetto *view* si concentra sull'interfaccia dell'applicazione e può consentire agli utenti di modificare i dati del modello dell'applicazione. La view non dovrebbe essere responsabile della memorizzazione dei dati che sta visualizzando. Un oggetto view può essere incaricato di visualizzare solo una parte di un oggetto modello o un intero oggetto modello o anche molti diversi oggetti modello. Gli oggetti view tendono ad essere riutilizzabili e configurabili e forniscono coerenza tra le applicazioni. In Cocoa (una API di iOS), sono definiti un gran numero di oggetti view e molte sono presenti nella libreria `Interface Builder`. Riutilizzando gli oggetti view, come gli oggetti `NSButton`, si garantisce che i pulsanti dell'applicazione si comportino esattamente come i pulsanti di qualsiasi altra applicazione Cocoa, garantendo un elevato livello di coerenza nell'aspetto e nel comportamento tra le applicazioni.

3.2.3 Controller

Un oggetto *controller* funge da intermediario tra gli oggetti della view dell'applicazione e i suoi oggetti modello. Gli oggetti controller possono anche eseguire attività di impostazione e coordinamento per un'applicazione e gestire i cicli di vita di altri oggetti. In un tipico progetto MVC, quando gli utenti immettono un valore tramite un oggetto della view, tale valore viene comunicata ad un oggetto controller.

L'oggetto controller può interpretare l'input dell'utente in un qualche modo specifico e quindi può dire a un oggetto modello cosa fare con questo input, ad esempio "aggiungi un nuovo valore" o "elimina il record corrente". Sulla base di questo stesso input dell'utente, alcuni oggetti controller potrebbero anche dire a un oggetto della view di modificarsi, come ad esempio dire ad un pulsante di disabilitare se stesso. Viceversa, quando un oggetto modello cambia, ad esempio si accede a una nuova origine dati, l'oggetto modello di solito comunica tale modifica ad un oggetto controller, che quindi richiede uno o più oggetti della view per aggiornarsi di conseguenza. Gli oggetti controller possono essere riutilizzabili o non riutilizzabili a seconda del loro tipo generale.

3.3 Progettazione

Essendo che l'algoritmo è diviso in diverse fasi ed utilizza piattaforme di terze parti, è stato necessario fare una fase di progettazione ben accurata. Di seguito viene mostrato il diagramma di attività che descrive come funziona ad alto livello l'applicazione:

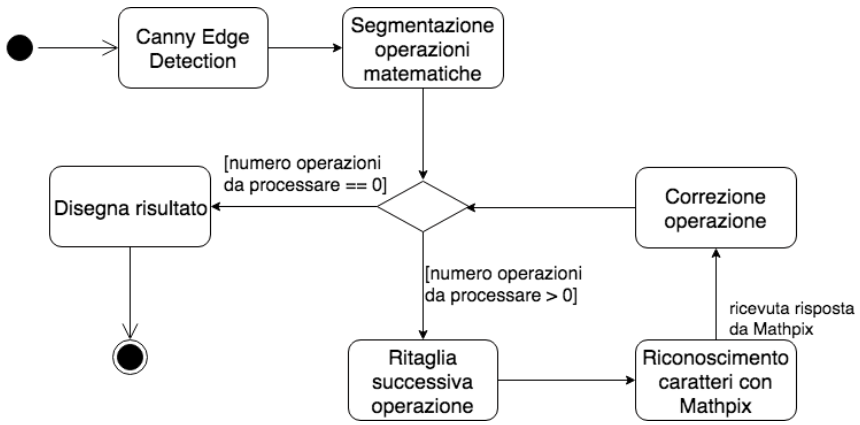


Figura 3.1: Diagramma di attività dell'intera applicazione.

Come primo passaggio viene applicato l'algoritmo `Canny Edge Detection` per riuscire a trovare i bordi dei simboli e caratteri delle operazioni matematiche e per riuscire, con opportune soglie, ad eliminare il rumore provocato dai quadretti presenti nei fogli. Successivamente viene applicato l'algoritmo per la segmentazione delle intere operazioni matematiche (`Segmentazione operazioni`

`matematiche`). Questo attività prende più passi che descriveremo più nel dettaglio nel paragrafo successivo. Segmentate tutte le operazioni matematiche, ovvero trovate le loro coordinare sull'immagine, si procede con un ciclo in cui, finchè ci sono ancora operazioni da processare, si ritaglia l'operazione dall'immagine originale (`Ritaglia successiva operazione`), l'immagine ritagliata viene passata a Mathpix per il riconoscimento dei simboli e caratteri (`Riconoscimento caratteri con Mathpix`) e infine, una volta che si riceve la risposta da Mathpix, si correggono le operazioni (`Correzione operazione`). Quando le operazioni da processare sono terminate, si esce dal ciclo e viene disegnato sull'immagine originale, per ogni operazione, un simbolo che indica se l'operazione è corretta o meno.

3.3.1 Progettazione algoritmo di segmentazione delle operazioni matematiche

L'attività `segmentazione operazioni matematiche` nel diagramma di attività in Figura 4.1 a sua volta è composto da sotto attività. Mostriamo le sotto attività nel seguente diagramma delle attività:

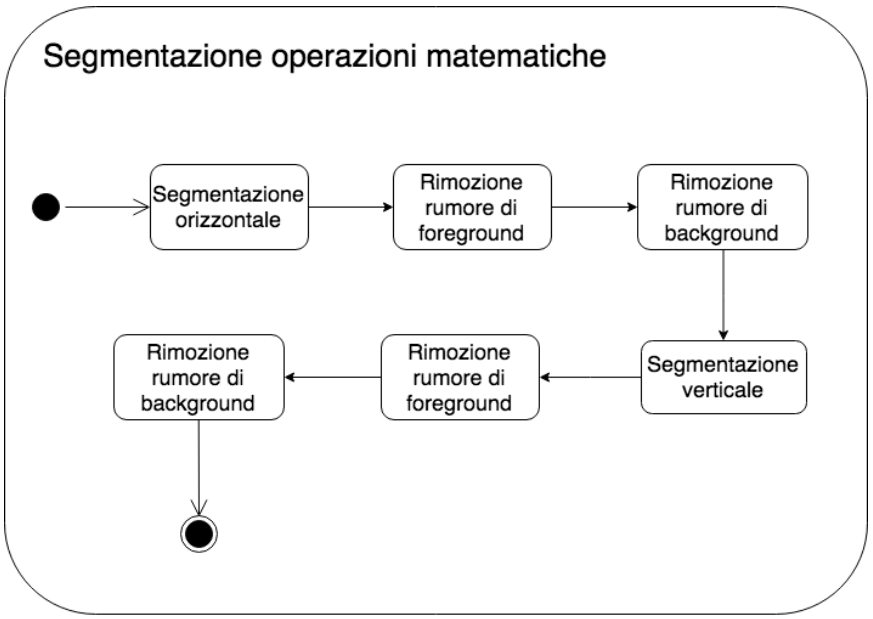


Figura 3.2: Sotto diagramma di attività dell'attività *segmentazione operazioni matematiche* del diagramma in Figura 4.1.

L'algoritmo inizia con l'attività di `Segmentazione orizzontale` che consiste nel separare orizzontalmente le diverse operazioni matematiche nell'immagine (come mostrato nell'esempio in Figura 2.12). In seguito al risultato dell'algoritmo `Canny Edge Detection`, l'immagine potrebbe essere affetta da rumore, in particolare l'algoritmo potrebbe aver etichettato come simboli di una operazione anche dei piccoli segni presenti sul foglio o piccoli scarabocchi.
E' necessario minimizzare chiaramente il rumore nell'immagine, quindi si procede con l'applicazione di algoritmi di rimozione del rumore di foreground (indicato nel diagramma come l'attività `Rimozione rumore di foreground`). Successivamente è necessario procedere alla rimozione del rumore di background (indicato nel diagramma come l'attività `Rimozione rumore di background`), ovvero la rimozione di quelle linee orizzontali che l'algoritmo di segmentazione orizzontale potrebbe etichettare come linee che separano due operazioni matematiche intere ma che in realtà è rumore. E' il caso delle operazioni in colonna: lo spazio che intercorre tra il primo numero e il secondo potrebbe essere etichettato come linea di separazione di due operazioni matematiche distinte quando in realtà è una unica. Si veda l'esempio in Figura 3.1.
L'algoritmo procede poi con la `Segmentazione verticale` applicando lo stesso processo della segmentazione orizzontale in modo ortogonale.

3.3.2 Progettazione interazione con Mathpix

Dovendo l'applicazione usufruire di servizi offerti da terze parti (Mathpix), è stato necessario fare una fase di progettazione per stabilire come la nostra applicazione dovesse interagire con Mathpix.
Di seguito viene mostrato un diagramma di sequenza che mostra come l'applicazione interagisce con Mathpix:

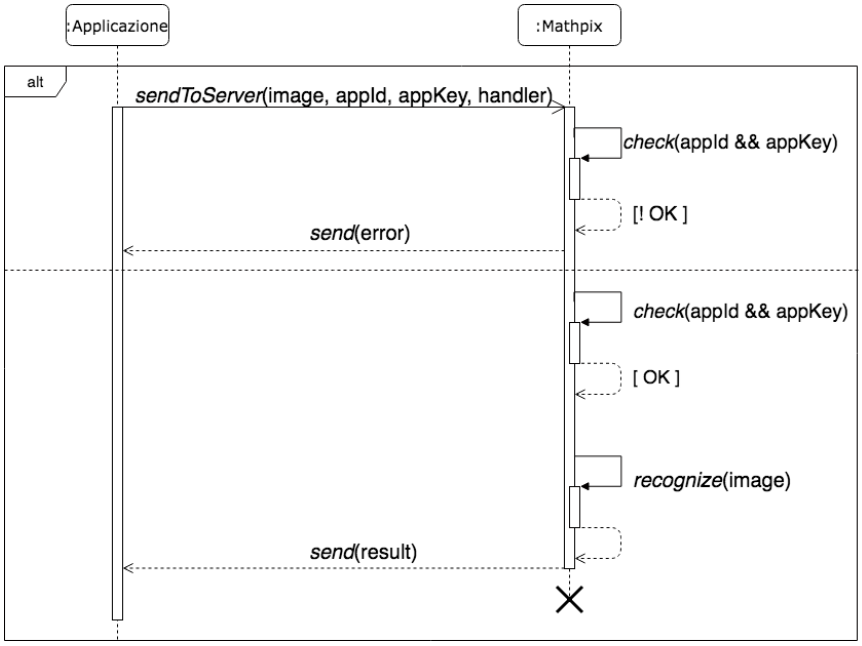


Figura 3.3: Diagramma di sequenza che descrive le interazioni dell'applicazione con Mathpix.

Questo flusso di lavoro è quello che succede ogni volta nell'attività `Riconoscimento caratteri con Mathpix` nel diagramma di attività in Figura 4.1. Ogni volta che si vuole mandare una immagine a Mathpix per il riconoscimento, viene inviata una richiesta *asincrona* dall'applicazione al server tramite il metodo indicato nel diagramma `sendToServer` specificando l'immagine di cui si vogliono riconoscere i caratteri e i parametri `appId`, `appKey` e `handler` che approfondiremo più nel dettaglio nella fase di implementazione. Una volta inviata la richiesta, Mathpix controlla se i parametri `appId` e `appKey` sono validi. Se non lo sono, allora viene spedito un messaggio di errore all'applicazione. Se invece sono validi, Mathpix procede con il riconoscimento dell'immagine e alla spedizione del risultato in formato JSON alla applicazione.

3.4 Implementazione

Di seguito mostreremo l'implementazione delle parti più importati dell'applicazione, in particolare mostreremo l'implementazione

dell'algoritmo di segmentazione delle operazioni matematiche e l'implementazione della connessione dell'applicazione con Mathpix.

3.4.1 Implementazione algoritmo di segmentazione delle operazioni matematiche

Come specificato nella fase di progettazione, l'algoritmo inizia con l'operazione di segmentazione orizzontale ovvero quella operazione che consiste nel separare orizzontalmente le diverse operazioni matematiche presenti nell'immagine. Per fare ciò, prima di tutto è stato necessario calcolare, per ogni riga dell'immagine, quanti pixel di foreground sono presenti e poi trasformare il valore le cui righe che hanno un valore > di `0` in `1`. Di seguito viene mostrata l'implementazione di questo passaggio:

```
1  private func calculateHorizontalForeground(
2      from pixelBuffer:
3           UnsafeMutablePointer<PhotoViewController.RGBA32>,
4      withWidth width: Int,
5      andHeight height: Int) -> Array<Int>? {
6          var foregrounds: Array<Int> = []
7          var foregroundAmount = 0
8          for row in 0 ..< Int(height) {
9              for column in 0 ..< Int(width) {
10                 let offset = row * width + column
11                 if pixelBuffer[offset] == .white {
12                     foregroundAmount += 1
13                 }
14             }
15             foregrounds.append(foregroundAmount)
16             foregroundAmount = 0
17         }
18         for index in foregrounds.indices {
19             if foregrounds[index] != 0 {
20                 foregrounds[index] = 1
21             }
22         }
23         return foregrounds
24 }
```

Listing 3.1: Calcolo dei pixel di foreground per ogni riga dell'immagine.

Successivamente a questo passaggio si procede con l'eliminazione del rumore di foreground e di background. Di seguito ne mostriamo l'implementazione utilizzata:

```
1  private func deleteNoise(
2       for arrayOfPoints: Array<CGPoint>,
3       withThreshold threshold: Int,
4       leftAndRightBlackValues leftAndRight: Int,
5       for noise: Int
6     ) -> Array<CGPoint>? {
7         var sums = arrayOfPoints
8         for index in 1..<sums.count - 1 {
9             if sums[index].y == noise {
10                if Int(sums[index].x) <= threshold {
11                    if Int(sums[index - 1].x)
12                    > leftAndRight
13                    || Int(sums[index + 1].x)
14                                    > leftAndRight {
15                        sums[index].y =
16                                    Swift.abs(noise - 1)
17                    }
18                }
19            }
20        }
21        if sums[0].y == 1.0, sums[0].x < 5 {
22            sums[0].y = 0.0
23        }
24        return sums
25    }
```

Listing 3.2: Algoritmo per la rimozione del rumore di foreground.

Il metodo `deleteNoise` se invocato con il parametro `noise` uguale a `0`, allora procede ad eliminare il rumore di background. Se invece viene invocato con il parametro `noise` uguale a `1`, allora il metodo procede con l'eliminazione del rumore di foregorund. Fatto ciò, occorre segmentare le operazioni verticalmente. Per effettuare questo passaggio, anche in questo caso è necessario calcolare, per ogni colonna dell'immagine, quanti pixel di foreground sono presenti e poi trasformare il valore le cui righe che hanno un valore $>$ di `0` in `1`. Di seguito viene mostrata l'implementazione di questo passaggio:

```
1  private func calculateVerticalForeground(
2         from pixelBuffer:
3         UnsafeMutablePointer<PhotoViewController.RGBA32>,
4         withWidth width: Int,
5         from start: Int,
6         to stop: Int
7     ) -> Array<Int>? {
8         var foregrounds2: Array<Int> = []
```

```
9           var foregroundAmount = 0
10          for col in 0 ..< Int(width) {
11              for row in (start ..< stop).reversed() {
12                  let offset = row * width + col
13                  if pixelBuffer[offset] == .white {
14                      foregroundAmount += 1
15                  }
16              }
17              foregrounds2.append(foregroundAmount)
18              foregroundAmount = 0
19          }
20          for index in foregrounds2.indices {
21              if foregrounds2[index] != 0 {
22                  foregrounds2[index] = 1
23              }
24          }
25          return foregrounds2
26      }
```

Listing 3.3: Calcolo dei pixel di foreground per ogni colonna dell'immagine.

3.4.2 Connessione a Mathpix

Per usufruire del meccanismo di riconoscimento dei caratteri, abbiamo fatto utilizzo delle API di Mathpix. Essendo che l'elaborazione avviene sui loro server, è necessario prima di tutto connettersi a loro. Per poterlo fare è necessario configurare un'account con un'ID e una chiave. Per configurare l'ID e la chiave si è usato il seguente codice:

```
1  MathpixClient.setApiKeys(
2   appId: "mathpix",
3   appKey: "139ee4b61be2e4abcfb1238d9eb99902")
```

Listing 3.4: Setup dell'account (ID + chiave) per poter usufruire dei servizi di Mathpix.

La chiave presente nel campo `appkey` è una chiave di prova che è possibile utilizzare soltanto per un periodo prestabilito. È possibile utilizzare le medesima oppure è possibile comprarne una permanente dal loro sito web.
Successivamente viene inviata una richiesta `http` al seguente dominio `https://api.mathpix.com/v3/latex`.
Il codice per l'invio della richiesta `http` ai server di Mathpix e per la gestione del risultato è riportato nel codice seguente:

```
@discardableResult func sendToServer(
                  image:UIImage,
                  appId: String?,
                  appKey: String?,
                  outputFormats formats: [MathpixFormat]?,
                  complitionHandler:
                  MathpixClient.RecognitionCallback?
              ) -> UUID {

      guard let appId = appId, let appKey = appKey else {
          fatalError("Set api keys before request")
      }
      let URL = Foundation.URL(
                           string: Constants.requestURL)!
      var request = URLRequest(url: URL)
      let imageData = UIImageJPEGRepresentation(
                           image, 0.5)
      let base64String = imageData!.base64EncodedString(
                             options: .init(rawValue: 0))
      let headers = [
          "content-type": "application/json",
          "app_id": appId,
          "app_key": appKey
      ]
      var formatsJson : [String : Any] = [:]
      formats?.forEach{
          formatsJson[$0.json.key] = $0.json.value
      }
      var parameters = [
          "url": Constants.bodyStart + base64String,
          "formats": formatsJson,
          "ocr": Constants.math
      ] as [String : Any]
      let postData = try! JSONSerialization.data(
         withJSONObject: parameters,
         options: [])
      request.httpMethod = "POST"
      request.allHTTPHeaderFields = headers
      request.httpBody = postData as Data
      request.cachePolicy = .reloadIgnoringLocalCacheData
      request.httpShouldHandleCookies = false
      request.timeoutInterval = 60
      let session = URLSession(
       configuration: self.currentSessionConfiguration())
      let taskId = UUID()
      if MathpixClient.debug {
```

```
47              parameters.updateValue("...", forKey: "url")
48              NSLog("Send request: %@",
49                                    request.debugDescription)
50              NSLog("parameters: %@", parameters)
51          }
52          let dataTask = session.dataTask(
53             with: request,
54             completionHandler: { [weak self]
55             (data, response, error) in
56              self?.tasks.removeValue(forKey: taskId)
57              var currentError : Error?
58              var result: RecognitionResult?
59              defer {
60                  DispatchQueue.main.async(execute: {
61                    if MathpixClient.debug {
62                      if let error = currentError {
63                        NSLog(error.localizedDescription)
64                      } else if let result = result?.parsed {
65                        NSLog("Success request: %@", result)
66                      }
67                  }
68                  complitionHandler?(currentError, result)
69                  MathpixClient.completion?(
70                                    currentError, result)
71              })
72            }
73              guard error == nil else {
74                  currentError = NetworkError(
75                                    error: error! as NSError)
76                  return
77              }
78              guard let data = data else {
79                  currentError = NSError(
80                     domain: "localhost",
81                     code: -1,
82                     userInfo:
83                        [NSLocalizedDescriptionKey : "error"])
84                  return
85              }
86              do {
87                  result = try RecognitionResult(data: data)
88              } catch {
89                  currentError = error
90              }
91          })
92          self.tasks[taskId] = dataTask
```

```
93          dataTask.resume()
94          return taskId
95      }
```

Listing 3.5: Invio della richiesta http ai server di Mathpix per usufruire del servizio di riconoscimento dei caratteri.

Descrizione de parametri che prende in input il metodo:

- `image:` è l'immagine che vogliamo inviare al server per il riconoscimento dei simboli e caratteri;
- `appId` e `appKey:` sono due strighe che rappresentano il nostro account preliminarmente configurato;
- `complitionHandler:` è una callback che, se vogliamo, possiamo fare eseguire a Mathpix dopo la sua elaborazione.

La risposta con il riconoscimento dal server può arrivare in un qualsiasi momento imprevedibile. Per risolvere questo problema, si è creato un buffer che implementasse il pattern Observer, in modo tale che, quando il server di Mathpix riempiva questo buffer con la sua risposta, il buffer stesso notificava chi di dovere che la risposta da Mathpix è arrivata e quindi pronta per essere processata.

3.5 Chatbot

Realizzata l'intera applicazione è sorta la seguente domanda: è stato proprio necessario costruire un'applicazione che fosse in grado di svolgere questo lavoro? L'interfaccia grafica dell'applicazione è sostanzialmente un software che permette di scattare fotografie. Se al posto di costruire un'applicazione a parte si costuisse un `chatbot` che esegua la nostra elaborazione? Così si è fatto: si è deciso di implementare un chatbot per Telegram che svolgesse lo stesso compito della nostra applicazione iOS.
Il chatbot inoltre ha diversi *vantaggi*, uno dei principali è sicuramente il fatto che l'utente per usare il servizio offerto dall'applicazione non deve per forza scaricarla, è probabile che abbia già Telegram sul suo dispositivo e quindi, per usufruire del servizio, è sufficiente che contatti il chatbot direttamente da Telegram. Un'altro vantaggio importante è per gli sviluppatori: non essendoci più un'applicazione con interfaccia

grafica, non c'è bisogno di lavorare sul design, sulla user experience e sull'implementazione di quest'ultima, il tutto è gestito da Telegram.
Uno *svantaggio* è che il software per scattare fotografie offerto da Telegram non consente di lavorare sulla realtà aumentata e quindi non permette di inserire il meccanismo di autoaggiustamento delle soglie discusso nel Capitolo 2. Tuttavia, per l'implementazione del chatbot, si è utilizzata la libreria `OpenCV` che, essendo molto più potente e robusta delle libreria `GPUImage2` di iOS, ha permesso di migliorare l'algoritmo di segmentazione delle intere operazioni matematiche rendendo il servizio offerto dall'applicazione utilizzabile anche senza l'utilizzo della realtà aumentata.
Per la realizzazione del chatbot si è utilizzato il linguaggio Python 3.

3.5.1 Creazione chatbot

Per poter creare il chatbot è necessario come operazione preliminare avere un meccanismo per poter interagire con il chatbot all'interno di Telegram. Per poterlo fare bisogna contattare dall'applicazione Telegram un bot che si chiama `BotFather` ed eseguire i passaggi da lui specificati. Una volta eseguiti correttamente questi passaggi, BotFather restituisce un `token` che rappresenta la chiave per accedere al nostro bot tramite le API offerte da Telegram.

3.5.2 Interazione con il chatbot

Per interagire con il bot si è utilizzata la libreria `telebot` che permette, con solamente una linea di codice, di connetterci al nostro chatbot e di avere un'istanza di esso con cui possiamo interagire scambiando messaggi.
Mostriamo il codice utilizzato di seguito:

```
1 import telebot
2
3 bot_token = 'token restituito da BotFather'
4 bot = telebot.TeleBot(token=bot_token)
```

Listing 3.6: Connessione al chatbot.

Per poter far intercettare i nostri messaggi al bot è necessario definire un `message_handler` apposito.
Mostriamo di seguito un'esempio di scambi di messaggi in cui bot

risponde all'utente con lo stesso messaggio che l'utente gli ha inviato (`echo`):

```
1 @bot.message_handler(content_types=['text'])
2 def echo(message):
3     bot.send_message(message.chat.id, message.text)
```

Listing 3.7: Esempio echo con il chatbot.

In questo esempio il chatbot risponde solamente se il messaggio inviato dall'utente è un messaggio testuale in quanto il campo `content_types` è di tipo `text`. È possibile specificare per il `content_types` anche le seguenti stringhe in modo tale che il chatbot possa essere reattivo anche quando inviamo, per esempio, documenti, audio, foto e altro:

```
text, audio, document, photo, sticker, video,
video_note, voice, location, contact, new_chat_members,
left_chat_member, new_chat_title, new_chat_photo,
delete_chat_photo, group_chat_created,
supergroup_chat_created, channel_chat_created,
migrate_to_chat_id, migrate_from_chat_id, pinned_message
```

Sapendo per cui come interagire con il bot, si è riscritto completamente l'intero algoritmo. Il bot accetta delle foto di operazioni matematiche in riga e in colonna e restituisce la stessa foto inviata dall'utente con le correzioni.

Capitolo 4

Conclusioni

Concludendo, in questa tesi è stato mostrato un'algoritmo in grado di corregge molteplici operazioni matematiche presenti in una foto. Ci si è concentrati su operazioni di testing quali somme, sottrazioni, moltiplicazioni e divisioni per verificare che questa cosa di potesse effettivamente fare.
Il primo problema affrontato è stato la segmentazione dei singoli simboli e caratteri delle operazioni. Essendo che i fogli di matematica contengono quadretti, è stato necessario applicare un'algoritmo che fosse in grado efficacemente di eliminarli e segmentare solamente i pixel di interesse. Si è utilizzato a tale scopo l'algoritmo Canny Edge Detection. Segmentati tutti i caratteri, il successivo problema è stato segmentare le intere operazioni matematiche, ovvero riuscire a capire, per ogni simbolo e carattere segmentato, a quale operazione matematica appartenesse. Questo è stato risolto con l'algoritmo dell'intera operazione matematica (Capitolo 1, paragrafo 2.4) completamente progettato ed implementato dal sottoscritto. Riusiciti a segmentare tutte le diverse operazioni matematiche, serviva un meccanismo che fosse in grado di riconoscere le operazioni matematiche per poi capire se queste fossero corrette o meno. Per fare ciò si sono usufruite delle API di Mathpix. L'algoritmo funziona correttamente ma solamente sotto determinate circostanze. In particolare l'algoritmo dipende da soglie fisse che lo fanno funzionare correttamente solo se la foto viene scattata con il dispositivo mobile ad una precisa distanza dal foglio. Era necessario per cui un meccanismo che riuscisse a capire a che distanza dal foglio si trovava il dispositivo nel momento dello scatto della foto in modo tale da aggiustare di conseguenza le soglie fisse e far funzionare l'algoritmo anche se la foto

veniva scattata a distanze dal foglio differenti. Per risolvere questo problema si è fatto un utilizzo indiretto della realtà aumentata.

4.1 Lavori futuri

Cos'è che si può ancora fare? Cos'ho ancora in mente?
L'algoritmo proposto è un buon punto di partenza. Tuttavia richiede diverse migliorie. Una delle prime migliorie che si vogliono applicare è rendere l'applicazione indipendente dalle API di Mathpix essendo che queste introducono un ritardo non indifferente stando in esecuzione su dei server. Per fare questo è necessario l'utilizzo di intelligenza artificale che sia in grado in maniera efficace ed efficiente di svolgere lo stesso lavoro di Mathpix. Si intende poi migliorare l'algoritmo di segmentazione delle intere operazioni matematiche cercando di renderlo più efficiente dal punto di vista computazionale. Questo è possibile farlo migliorando i sotto algoritmi che questo algoritmo utilizza.
Oltre alle operazioni testate in questa tesi si potrebbero riconoscere anche altre operazioni come la divisone in colonna, equivalenze ed operazioni con parentesi. Non è stato fatto perchè non è scopo della tesi.
Si potrebbe applicare questo algoritmo anche nel campo della geometria. In questo caso però sarà necessario construire una intelligenza artificiale in grado di riconscere i diversi solidi geometrici e come questi sono posizionati.
Si potrebbero riconoscere anche operazioni che in altri stati hanno layout differenti, come ad esempio la divisione in colonna che in Inghilterra presenta un layout differente rispetto a quello italiano.
Si possono ampliare le tecnologie di riconoscimento passando dal riconoscere, non solo operazioni matematiche delle elemantari, ma anche operazioni delle scuole medie.
Essendo che l'algoritmo non funziona correttamente se vengono inqudrati i bordi del foglio, è necessario trovare un'algoritmo che sia in grado di trovare i bordi del foglio ed eliminarli. Per questioni di tempo non sono riuscito a farlo.
La realtà aumentata è stato utilizzata solo indirettamente, ma in realtà potrebbe essere utilizzata dal punto di vista estetico con profitto al posto di utilizzare l'elaborazione di immagini per mostrare l'output.
Si intende, rendere automatica la ricerca del foglio sul piano orizzontale

e non demandarla più all'utente. Anche in questo caso sarà necessario l'utilizzo di intelligenza artificiale.

Bibliografia

[1] *From rgb to grayscale conversion*
http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6

[2] *Histogram*
http://bias.csr.unibo.it/fei/Dispense/2%20-%20FEI%20-%20Operazioni%20sulle%20Immagini.pdf

[3] *Canny Edge Detection*
http://bias.csr.unibo.it/fei/Dispense/3%20-%20FEI%20-%20Estrazione%20dei%20Bordi%20e%20Segmentazione.pdf

[4] *The Swift Language*
https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-swift-programming-language/

[5] *Model – View – Controller*
https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html