

# Applicazione Android per la conversione di schemi circuituali disegnati a mano in codice **LATEX** via CircuiTikZ

Progetto di Sistemi Digitali M

Luca Andreetti, Valentino Cavallotti, Mattia Moffa

<https://github.com/MAC-Projects/PhotoCircuit>

Aprile 2023

# Indice

<b>1 Introduzione</b>	<b>2</b>
1.1 T <sub>E</sub> X e L <sup>A</sup> T <sub>E</sub> X . . . . .	2
1.2 CircuiTikZ . . . . .	2
1.3 PhotoCircuit . . . . .	3
<b>2 Panoramica dell'applicazione e delle tecnologie utilizzate</b>	<b>4</b>
2.1 Comportamento dell'applicazione . . . . .	4
2.2 Organizzazione dell'implementazione . . . . .	4
2.3 Ipotesi sui circuiti . . . . .	5
<b>3 Implementazione dell'interfaccia utente</b>	<b>6</b>
3.1 MainActivity . . . . .	6
3.2 FirstFragment . . . . .	7
3.3 SecondFragment . . . . .	8
3.4 ThirdFragment . . . . .	10
<b>4 Localizzazione degli elementi circuituali e dei segmenti</b>	<b>11</b>
4.1 Localizzazione degli elementi circuituali . . . . .	11
4.2 Localizzazione dei segmenti . . . . .	17
<b>5 Classificazione degli elementi circuituali</b>	<b>18</b>
5.1 Creazione e allenamento del modello . . . . .	18
5.2 Utilizzo all'interno dell'applicazione . . . . .	20
<b>6 Rappresentazione logica del circuito e generazione dell'output</b>	<b>23</b>
6.1 Costruzione iniziale di nodi e segmenti . . . . .	24
6.2 Risoluzione delle irregolarità in presenza di diramazioni . . . . .	25
6.3 Rimozione dei segmenti duplicati . . . . .	26
6.4 Aggiunta degli elementi circuituali . . . . .	26
6.5 Raddrizzamento dei segmenti . . . . .	27
6.6 Generazione del codice CircuiTi <sub>Z</sub> . . . . .	28
<b>7 Collaudi</b>	<b>29</b>
<b>8 Conclusione</b>	<b>33</b>

# Capitolo 1

## Introduzione

### 1.1 $\text{\TeX}$ e $\text{\LaTeX}$

$\text{\TeX}$  è un sistema software estremamente diffuso per l’impaginazione di documenti, originalmente sviluppato nel 1978. Si tratta di un linguaggio di markup di tipo procedurale, che previa compilazione può produrre documenti in diversi formati, ad esempio DVI, PDF, HTML o testo semplice. Ne esistono diverse distribuzioni, come TeX Live e MiKTeX: queste permettono anche di installare *pacchetti* esterni, ovvero collezioni di macro che aggiungono funzionalità al sistema in modo di consentire una più veloce stesura di documenti.

$\text{\LaTeX}$  è un pacchetto, originalmente sviluppato nel 1984, che definisce una notevole quantità di macro  $\text{\TeX}$ . Nel tempo si è diffuso a un livello tale che al giorno d’oggi  $\text{\TeX}$  è raramente utilizzato senza: è incluso di default nelle maggiori distribuzioni ed è largamente utilizzato in ambito accademico per la pubblicazione di documenti scientifici nei campi più disparati.

Uno dei maggiori punti di forza di  $\text{\TeX}$  e  $\text{\LaTeX}$  è il loro notevole supporto all’impaginazione di formule matematiche complesse e di documenti scritti in sistemi di scrittura lontani da quello latino.

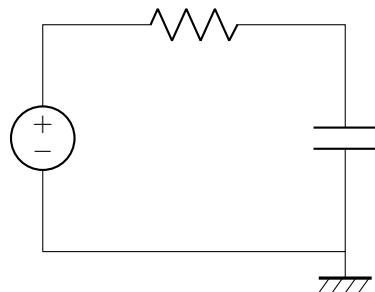
### 1.2 CircuiTikZ

PGF (*Portable Graphics Format*) è un pacchetto  $\text{\TeX}$  che offre macro per la generazione di elementi grafici in modo vettoriale. È controllabile attraverso un pacchetto di più alto livello, TikZ, che incapsula le macro offerte da PGF. Entrambi questi sistemi sono sviluppati da Till Tantau e Christian Feuersängge.

Il pacchetto CircuiTikZ è un add-on di TikZ sviluppato dal 2007 da Massimo Redaelli del Politecnico di Milano. Offre un insieme di macro per la rappresentazione di schemi circuituali ed è ampiamente usato in pubblicazioni di ambito accademico.

Con CircuiTikZ è possibile disegnare circuiti componendo *segmenti*, con o senza elementi circuituali, di cui si specificano le coordinate degli estremi. Si tratta di un pacchetto molto completo, che consente l’uso di numerosi elementi circuituali, di cui supporta anche le diverse convenzioni e varianti esistenti (europea, americana ecc.).

Un esempio di circuito disegnato attraverso CircuiTikZ è il seguente:



Il corrispondente codice è il seguente (può essere incluso direttamente all'interno di un documento L<sup>A</sup>T<sub>E</sub>X, a patto che nel preambolo sia presente il comando \usepackage{circuitikz}):

```
\begin{circuitikz}[american]
    \draw (0, 0) -- (4, 0) to [C] (4, 3) to [R] (0, 3) to [V] (0, 0);
    \draw (4, 0) node[eground] {};
\end{circuitikz}
```

Il codice sopra è composto da due istruzioni \draw, a cui vengono fornite liste di punti (espressi secondo la sintassi (x, y)), separati da designazioni dei segmenti che li connettono. Un'istruzione \draw molto semplice è la seguente:

```
\draw (0, 0) -- (1, 0)
```

Essa disegna un cortocircuito dalle coordinate (0, 0) alle coordinate (1, 0).

In realtà la sintassi -- è un'abbreviazione di to [short]. Ogni tipo di segmento è infatti espresso da un suo identificatore: short indica il corto circuito, R indica la resistenza, C indica il condensatore, L indica l'induttore e così via.

Dopo ogni coppia di coordinate è possibile esprimere l'aspetto dell'eventuale nodo che vi si trova. Tramite la seconda \draw del codice mostrato sopra, ad esempio, viene disegnato un nodo di tipo eground (*European ground*) alle coordinate (4, 0).

La prima \draw del codice disegna una serie di segmenti *seguendo un percorso*, ma è anche possibile esprimere separatamente, uno per volta:

```
\draw (0, 0) -- (4, 0);
\draw (4, 0) to [C] (4, 3);
\draw (4, 3) to [R] (0, 3);
\draw (0, 3) to [V] (0, 0);
```

In questo caso l'ordine delle direttive \draw può essere variato liberamente. L'uso di una sintassi di questo tipo è particolarmente utile quando il codice deve essere generato automaticamente da un algoritmo, come nel caso dell'applicazione PhotoCircuit.

### 1.3 PhotoCircuit

PhotoCircuit è l'applicazione Android oggetto di questa relazione. È in grado di produrre, in seguito all'acquisizione di una fotografia di un semplice schema circuitale disegnato a mano, il corrispondente codice CircuiTikZ. Questo viene sia mostrato a schermo che reso disponibile ad un determinato URL web pubblico, in modo che possa essere facilmente reperito e copiato su un altro dispositivo (tipicamente un PC con installata una distribuzione T<sub>E</sub>X).

Per funzionare, PhotoCircuit si avvale delle seguenti tecnologie:

- OpenCV: per l'elaborazione di immagini in forma algoritmica; in particolare è stata utilizzata per le elaborazioni preliminari, per la realizzazione dell'algoritmo di localizzazione degli elementi circuituali e per l'identificazione dei segmenti che li connettono, nonché svariate elaborazioni accessorie;
- Tensorflow Lite: per la classificazione degli elementi circuituali;
- Le varie API di Android, in particolare CameraX per l'acquisizione delle immagini;
- TermBin: per il caricamento del codice generato sul web in modo che sia disponibile ad un URL pubblico.

Il codice dell'applicazione può essere trovato su GitHub all'indirizzo <https://github.com/MAC-Projects/PhotoCircuit>.

## Capitolo 2

# Panoramica dell'applicazione e delle tecnologie utilizzate

### 2.1 Comportamento dell'applicazione

PhotoCircuit è strutturata in tre schermate:

- Schermata iniziale, che mostra l'anteprima della fotocamera e offre pulsanti per mettere a fuoco e per scattare la foto.
- Schermata intermedia, che mostra i risultati di vari algoritmi applicati all'immagine, e offre un pulsante per passare alla schermata finale.
- Schermata finale, contenente il codice L<sup>A</sup>T<sub>E</sub>X (CircuiTikZ) generato, e, non appena disponibile, anche l'URL TermBin al quale il codice è reperibile.

All'anteprima della fotocamera nella schermata iniziale sono sovrapposti rettangoli che mostrano all'utente le zone in cui l'applicazione ritiene di aver individuato elementi circuitali. I rettangoli vengono generati periodicamente in tempo reale: questo permette all'utente di verificare, prima di scattare la foto, se la qualità del disegno e/o le condizioni di luce sono sfavorevoli, e in tal caso di regolarle opportunamente.

Quando i rettangoli contengono esattamente gli elementi circuituali del disegno inquadrato, l'utente può procedere a scattare la foto. Questo provoca il passaggio alla schermata intermedia, nella quale sono mostrate:

- L'immagine in versione elaborata in forma binaria, in modo tale da mostrare i tratti che compongono il circuito in bianco e tutto il resto in nero.
- Le varie sotto-immagini contenenti ogni elemento circuitale trovato.
- Un'immagine che evidenzia con vari colori i segmenti rettilinei trovati.

Queste informazioni possono essere utili all'utente per verificare a colpo d'occhio se l'elaborazione dell'immagine è andata a buon fine.

Dalla seconda schermata l'utente può toccare un pulsante per passare alla terza. Questa contiene due aree di testo, una per l'URL TermBin e una per il codice L<sup>A</sup>T<sub>E</sub>X.

Premere il tasto «indietro» del cellulare quando si è nella terza schermata comporta il ritorno alla seconda e premerlo nuovamente comporta il ritorno alla prima, dalla quale è possibile scattare una nuova fotografia.

### 2.2 Organizzazione dell'implementazione

PhotoCircuit è costituita da un insieme di package Java:

- **ui**: si interfaccia con le API di Android al fine di implementare l'attività principale (`MainActivity`) e le tre schermate (`Fragment`) di cui è composta;
- **image\_processing**: implementa funzioni di utilità basate su OpenCV per l'elaborazione delle immagini, tra cui in particolare gli algoritmi di localizzazione degli elementi circuitali e dei segmenti rettilinei che li connettono;
- **ml\_handling**: incapsula il codice che utilizza il modello Tensorflow Lite atto alla classificazione degli elementi circuitali e alcune funzioni di utilità per l'interpretazione del suo output;
- **circuit\_model**: si occupa di trasformare gli output delle elaborazioni precedenti in una struttura dati che descrive il circuito in modo logico; in aggiunta definisce algoritmi di vario tipo che elaborano ulteriormente tale struttura dati (ad esempio per rendere i segmenti perfettamente orizzontali o verticali) e infine la trasformano in codice CircuiTikZ.
- **termbin\_handling**: gestisce la comunicazione TCP con TermBin che consente di effettuare l'upload del codice CircuiTikZ.

## 2.3 Ipotesi sui circuiti

PhotoCircuit supporta esclusivamente circuiti:

- Che contengano solo i seguenti elementi circuitali:
  - Resistenze
  - Capacitori
  - Induttori
  - Generatori di tensione (dipendenti o non)
  - Generatori di corrente continua (dipendenti o non)
  - Generatori di corrente alternata
  - Batterie
  - Diodi
  - Terra
- Che usino solo fili orientati (approssimativamente) in modo orizzontale o verticale
- Che non prevedano etichette o altri simboli (ad esempio frecce) vicino agli elementi circuitali
- I cui elementi circuitali siano posti ad una distanza adeguata tra loro e dai cambi di direzione dei fili

# Capitolo 3

## Implementazione dell'interfaccia utente

L'interfaccia utente dell'applicazione è stata inserita all'interno del package Java ui. Esso contiene le seguenti classi:

- **MainActivity**, che implementa l'unica *activity* dell'applicazione;
- **FirstFragment**, che implementa la prima schermata, ovvero quella di acquisizione foto;
- **SecondFragment**, che implementa la seconda schermata, ovvero quella che mostra graficamente i risultati intermedi delle elaborazioni;
- **ThirdFragment**, che implementa la terza schermata, ovvero quella che mostra il codice CircuiTikZ finale e il link a TermBin.

La prima classe discende da `AppCompatActivity`, le altre da `Fragment`.

- Un'attività è un possibile entry point di un'applicazione; dato che PhotoCircuit può essere avviata in un solo modo, possiede una sola attività;
- Un frammento è una porzione di UI riutilizzabile tramite incapsulamento all'interno di opportuni elementi.

Ognuna di queste classi possiede un corrispettivo file XML che ne descrive il layout.

### 3.1 MainActivity

Il layout della classe `MainActivity` contiene un singolo elemento: una `FragmentContainerView`, ovvero un contenitore di frammenti, che gestisce le transizioni dall'uno all'altro e mantiene automaticamente la storia delle transizioni (in modo da permettere di tornare indietro premendo il pulsante *indietro* fornito dal sistema Android). All'avvio deve mostrare il `FirstFragment`, per poi innescare le transizioni da un frammento all'altro quando necessario.

La `FragmentContainerView` è collegata ad un `NavGraph`, ovvero un grafo di navigazione che enumera i frammenti mostrabili e le transizioni (*azioni*) possibili tra essi. Nel caso di PhotoCircuit il `NavGraph` fa riferimento ai tre frammenti `FirstFragment`, `SecondFragment` e `ThirdFragment`, e consente la transizione dal primo al secondo e dal secondo al terzo.

La classe `MainActivity` non fa altro che attivare il proprio layout e richiedere i permessi necessari al funzionamento dell'applicazione. Questi, definiti nel file `AndroidManifest.xml`, sono tre:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

- Il permesso `CAMERA` è necessario per acquisire le fotografie attraverso CameraX;

- Il permesso INTERNET è necessario per poter aprire socket di rete;
- Il permesso ACCESS\_NETWORK\_STATE è necessario per controllare lo stato della connessione.

Di fatto solo il primo di questi tre permessi deve necessariamente essere richiesto all'utente a tempo di esecuzione, per cui è l'unico di cui la classe `MainActivity` si deve occupare.

Un altro ruolo della `MainActivity` è quello di inizializzare OpenCV:

```
if (OpenCVLoader.initDebug()) {
    Log.d("SUCCESS", "OpenCV loaded");
} else {
    Log.d("ERROR", "Unable to load OpenCV");
    throw new RuntimeException("Unable to load OpenCV");
}
```

## 3.2 FirstFragment

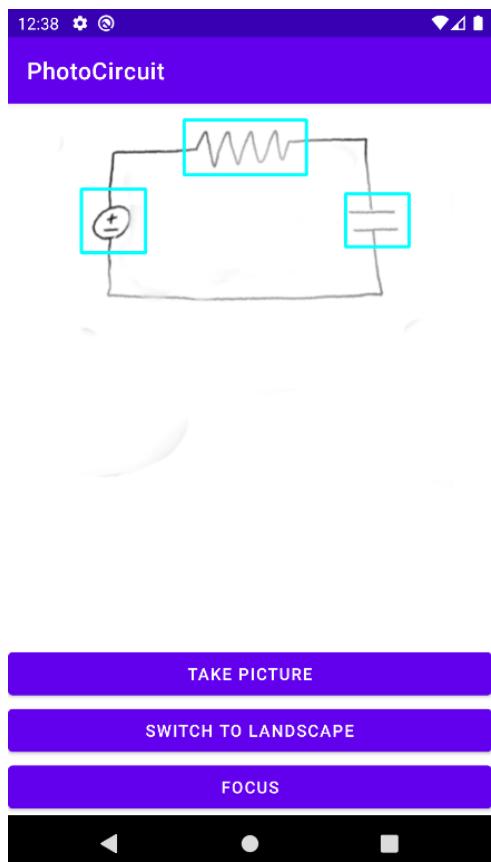


Figura 3.1: Schermata del FirstFragment

Il `FirstFragment` consente all'utente di acquisire le fotografie dei circuiti attraverso la fotocamera del dispositivo. Il suo layout è composto dai seguenti elementi:

- Un `LinearLayout` verticale contenente:
  - Una `ImageView` che mostra l'anteprima corrente della fotocamera
  - Un `LinearLayout` verticale con tre pulsanti:

- \* Uno per acquisire la fotografia (*Take picture*)
- \* Uno per cambiare la modalità dell'acquisizione tra verticale e orizzontale (*Switch to landscape, Switch to portrait*)
- \* Uno per forzare la messa a fuoco utilizzando il centro come punto focale (*Focus*)

La classe `FirstFragment` si occupa, una volta caricata all'interno della `FragmentContainerView`, di gestire l'acquisizione dalla fotocamera tramite CameraX utilizzando come *use case* sia `ImageAnalysis` che `ImageCapture`:

- `ImageAnalysis` viene utilizzato per elaborare l'immagine in tempo reale prima dell'acquisizione, al fine di mostrare a schermo rettangoli in corrispondenza delle parti dell'immagine che l'applicazione rileva come elementi circuituali. In particolare, ad ogni analisi dell'immagine (vale a dire ogni volta che CameraX chiama il metodo `analyze`), l'immagine viene ridimensionata e passata ad un opportuno thread (`ImageProcessingThread`) che, periodicamente, fornisce l'immagine attuale al metodo `processImage` della classe `ImageProcessor` (descritta nel capitolo 4), al fine di ottenere le coordinate dei rettangoli. A questo punto tali rettangoli vengono sovrapposti sull'immagine e il risultato viene mostrato a schermo reimpostando l'immagine mostrata dalla `ImageView`.
- `ImageCapture` viene utilizzato per l'acquisizione effettiva, che ha luogo quando viene premuto il pulsante *Take photo*. Anche in questo caso l'immagine viene ridimensionata, ma in seguito viene passata direttamente a `ImageProcessor.processImage`; in un secondo tempo viene caricato il `SecondFragment`, che riceve come argomento il risultato dell'elaborazione, che sarà la base per le elaborazioni successive.

### 3.3 SecondFragment

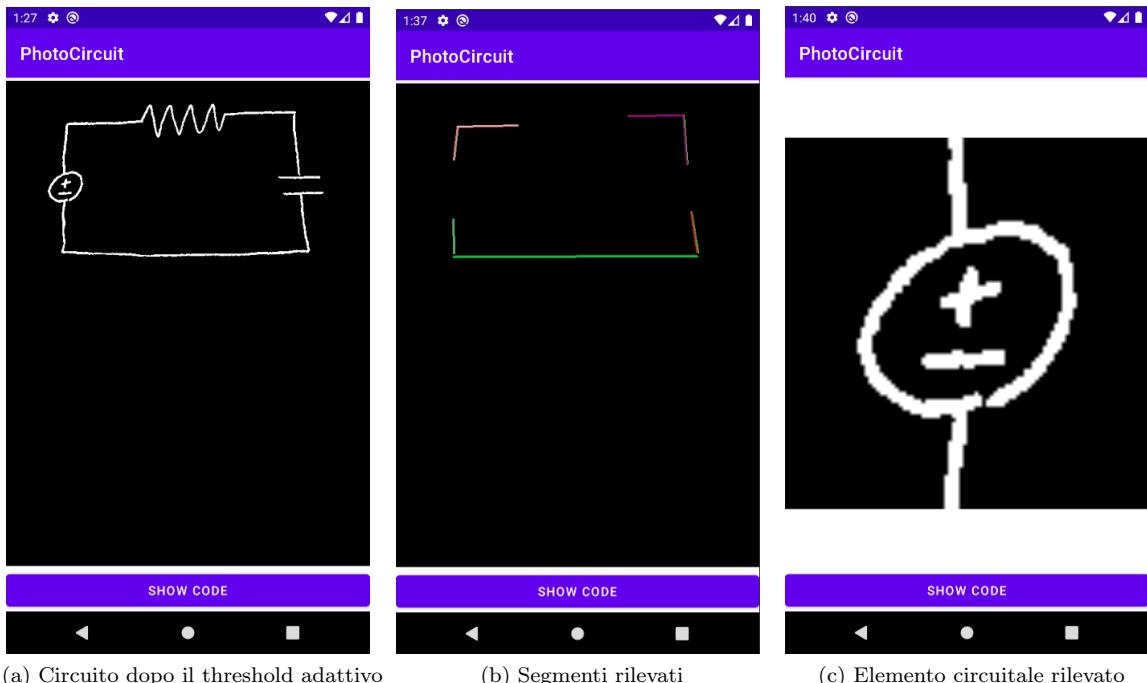


Figura 3.2: Schermate del `SecondFragment`

Lo scopo del `SecondFragment` è di svolgere il resto delle elaborazioni sul circuito e di mostrare all'utente alcune elaborazioni intermedie in forma grafica, così da permettergli di verificare direttamente se il riconoscimento è andato a buon fine, senza dover compilare il codice `CircuiTikZ`.

È costituito da un `LinearLayout` contenente:

- Una `ImageView` che mostra i risultati di elaborazione intermedi
- Un pulsante `Show code` che, quando premuto, procede con il caricamento del `ThirdFragment`, in modo da mostrare a schermo il codice finale generato e caricarlo su `TermBin`.

Nel momento in cui viene caricato, il `SecondFragment` svolge le seguenti operazioni:

- Estraie il risultato delle elaborazioni iniziali svolte dal `FirstFragment`, rappresentato da un oggetto della classe `ImageProcessingResult`.
- Dall'`ImageProcessingResult` ottiene un'immagine parzialmente elaborata (in particolare quella successiva all'operazione di threshold adattivo; si veda il capitolo 4 per maggiori dettagli) e da essa estrae le sotto-immagini contenenti gli elementi circuitali. Queste saranno poi passate al metodo `processImage` della classe `MLController`, che le classificherà tramite il modello Tensorflow Lite.
- Dall'`ImageProcessingResult` ottiene un'altra immagine ulteriormente elaborata (in particolare quella successiva alle operazioni di thinning e pruning leggero; si veda il capitolo 4 per maggiori dettagli); da questa cancella gli elementi circuitali, poi la fornisce al metodo `detectLines` della classe `ImageProcessor`, che si occupa di trovare i segmenti all'interno dell'immagine. Con questo meccanismo vengono rilevati appositamente solo i segmenti dei fili che *connettono* gli elementi circuitali, e non quelli che li *compongono*.
- Si basa sui dati ottenuti per mostrare a schermo le seguenti immagini, navigabili toccando su di esse (3.2):
  - Il circuito dopo l'operazione di threshold adattivo;
  - Le linee rilevate, ognuna di un colore diverso selezionato casualmente al fine di differenziarle;
  - Tutti gli elementi circuitali rilevati.
- A partire dagli stessi dati, costruisce un nuovo circuito chiamando il metodo `factory from` della classe `Circuit` (si veda il capitolo 6 per maggiori dettagli).
- Svolge ulteriori operazioni di elaborazione sul circuito ottenuto (in particolare lo «raddrizza» e inverte le coordinate della dimensione y).
- Converte il circuito in codice `CircuiTikZ` chiamando su di esso il metodo `toCircuiTikZ` (si veda il capitolo 6 per maggiori dettagli).
- Una volta che viene premuto il pulsante `Show code`, carica il `ThirdFragment` fornendogli il codice ottenuto.

### 3.4 ThirdFragment

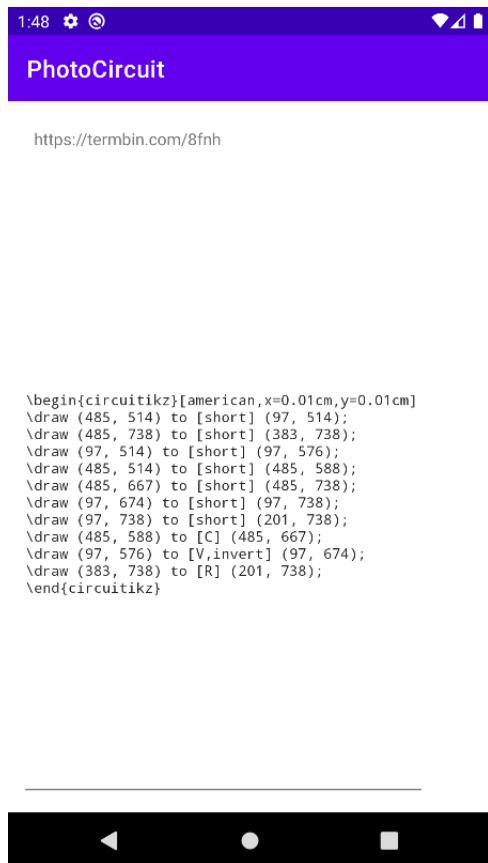


Figura 3.3: Schermata del `ThirdFragment`

Il `ThirdFragment` si occupa soltanto di mostrare a schermo il codice CircuiTikZ finale, di caricarlo su TermBin e di esporre l'URL così ottenuto. È costituito da un `LinearLayout` contenente:

- Una `TextView` che inizialmente mostra la dicitura «URL pending...» e poi, una volta caricato il codice su TermBin, l'URL;
- Un `EditText` non modificabile che mostra il codice CircuiTikZ. È stato utilizzato un `EditText` invece di una `TextView` al fine di consentire la selezione del testo, non supportata dalla `TextView`.

Una volta creato, il `ThirdFragment` aggiorna l'`EditText` con il codice ottenuto dal `SecondFragment`, dunque chiama (in un thread separato) il metodo `send` della classe `TermBinHandler` e successivamente aggiorna la `TextView` con l'URL risultante.

## Capitolo 4

# Localizzazione degli elementi circuituali e dei segmenti

L'individuazione degli elementi circuituali e dei segmenti rettilinei è implementata nel pacchetto `image_processing`, che contiene due classi: `ImageProcessor` e `ImageProcessingResult`.

`ImageProcessor` contiene la logica applicativa per l'elaborazione delle immagini. In particolare, ha due metodi importanti:

- `processImage`, che dirige l'applicazione di diversi algoritmi all'immagine e crea i rettangoli che contengono gli elementi circuituali.
- `detectLines`, che, data un'immagine, crea un Line Segment Detector OpenCV impostato con opportuni parametri e lo usa per individuare i segmenti nell'immagine, che poi restituisce.

La classe ha anche vari metodi di utilità, ad esempio per la conversione tra diversi formati di immagine, per il ridimensionamento e per la cancellazione degli elementi circuituali (da applicare prima della fase di individuazione dei segmenti, in modo che essa escluda i tratti che formano gli elementi).

La classe `ImageProcessingResult` è un semplice Java Bean, che incapsula i rettangoli prodotti dal metodo `processImage` insieme a due versioni dell'immagine originale sottoposte a certe elaborazioni parziali. Queste sono destinate rispettivamente al riconoscimento degli elementi circuituali con il modello Tensorflow Lite e alla rilevazione dei segmenti con il Line Segment Detector, dato che i due sistemi hanno diversi requisiti.

È stato scelto di implementare algoritmi appositi per svolgere questi compiti invece di utilizzare una CNN opportunamente allenata. Il motivo di questa scelta è dovuto a due fattori:

- Mancanza di un dataset di qualità adeguata (è stato trovato un unico dataset, ma le sue immagini erano troppo diverse da quelle che l'applicazione avrebbe dovuto interpretare, dunque i risultati non sono stati soddisfacenti)
- Insufficienza di risorse computazionali per un training adeguato

### 4.1 Localizzazione degli elementi circuituali

Il metodo `processImage` riceve in input l'immagine originale e la sottopone, in ordine, agli algoritmi descritti di seguito, implementati sulla base di diversi metodi di OpenCV definiti nella classe statica `Imgproc`.

**Conversione ad immagine in scala di grigi** Sfrutta il metodo `Imgproc.cvtColor`, fornendo come modalità di conversione `Imgproc.COLOR_RGB2GRAY`.

**Applicazione di un filtro bilaterale** Applica `Imgproc.bilateralFilter` (figura 4.2b). Questo algoritmo sfoca l'immagine, ma solo nelle zone lontane dai bordi degli oggetti nella foto [BilateralFilter]; data la foto di un circuito, ha l'effetto di ridurre il rumore solo nelle zone di «foglio», lasciando le linee del circuito nitide.

**Inversione di colore condizionale** Si tratta di un'inversione (*bitwise not*) dei pixel dell'immagine che viene applicata solo se il loro valore medio è sopra una certa soglia. Questo passaggio assicura di avere a disposizione un'immagine a sfondo scuro con tratti chiari, in modo da poter trattare tutte le immagini ugualmente da qui in poi.

**Applicazione di un threshold adattivo** Usa il metodo `Imgproc.adaptiveThreshold` utilizzando la modalità `Imgproc.THRESH_BINARY` [AdaptiveThreshold]. Questo risulta in un'immagine binaria, con i tratti del circuito in bianco e il resto in nero (figura 4.2c).

**Applicazione di una chiusura** Una *chiusura* è un'operazione che, data un'immagine binaria, corrisponde ad una dilatazione dell'area bianca di un certo numero di pixel seguita da un restringimento di egual misura; risulta in un'immagine analoga all'originale, ma con gli spazi vuoti piccoli colmati, in quanto sono riempiti dalla dilatazione iniziale ma non rigenerati dal restringimento [Closure]. È importante applicare tale operazione al circuito, in modo da eliminare eventuali interruzioni indesiderate nei fili, dovute ad esempio ad un disegno impreciso (figura 4.2d).

**Thinning («assottigliamento») dell'immagine** È realizzato da un algoritmo che riduce i tratti bianchi ad un sottile *scheletro* dello spessore di un pixel (figura 4.2e).

Per descriverne il funzionamento, bisogna prima definire la *trasformazione hit-miss*. Essa agisce su un'immagine binaria, partendo da un kernel in cui ogni cella può assumere i valori 0, 1 oppure indefinito. Ad ogni pixel dell'immagine viene sovrapposto il kernel; se ad ogni «1» (risp. «0») del kernel corrisponde un «1» (risp. «0») nell'immagine, allora l'immagine risultante avrà un «1» in tale posizione, altrimenti avrà uno «0» (i pixel indefiniti del kernel vengono ignorati). Lo scopo complessivo della trasformazione hit-miss è quindi di trovare tutti i pixel dell'immagine originale il cui intorno combacia con il kernel.

Data un'immagine binaria  $I$  e un kernel  $K$ , definiamo invece *hit-miss inverso* un'operazione del tipo:

$$\text{reverse-hit-miss}(I, K) = I - \text{hit-miss}(I, K)$$

Definiamo infine *hit-miss inverso iterato* l'esecuzione iterata di questa operazione un numero prestabilito di volte, oppure finché l'immagine dell'iterazione  $i$ -esima non risulta uguale a quella dell'iterazione  $(i - 1)$ -esima.

Il thinning [ThinningPruning] consiste nello svolgere un'operazione di hit-miss inverso iterato con tutte le rotazioni di 90° dei seguenti kernel (per un totale di otto kernel):

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & & \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 1 & 1 & 0 \\ 1 & & \end{bmatrix}$$

Una singola iterazione del thinning produce un'immagine binaria in cui sono bianchi soltanto i pixel il cui intorno *non* combacia con alcuno dei kernel dati. Questa immagine è come quella data, ma assottigliata di un pixel sui bordi (in particolare il primo kernel gestisce i bordi rettilinei mentre il secondo gestisce gli angoli). Le iterazioni terminano automaticamente quando l'immagine prodotta risulta uguale a quella data, il che indica che non è possibile assottigliare ulteriormente.

**Pruning («potatura»)** È un'operazione applicata allo scheletro ottenuto tramite il thinning che ne provoca l'*erosione* a partire dagli estremi (ovvero, ad esempio, dai pixel bianchi che hanno un

solo pixel bianco vicino) [ThinningPruning]. È anch'essa implementata mediante hit-miss inverso iterato, usando i seguenti kernel e tutte le loro rotazioni di 90°, per un totale di otto kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

L'algoritmo iterativo, in questo caso, va eseguito per un numero fissato di iterazioni, in modo da erodere il tratto solo fino a una particolare lunghezza.

L'applicazione crea due immagini sottoposte a pruning: una con pruning «leggero» (ovvero con poche iterazioni dell'algoritmo), usata in seguito per l'individuazione dei segmenti rettilinei, e una con pruning «pesante» (con più iterazioni), usata per l'individuazione degli elementi circuitali.

**Rilevazione delle linee non orizzontali o verticali** Si basa su un'applicazione dell'algoritmo di hit-miss inverso (non iterato), questa volta con i seguenti kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

L'obiettivo è di trovare parti dell'immagine originale che non rispettino i pattern di questi due kernel, ovvero di escludere le aree con pixel disposti in orizzontale/verticale. L'immagine risultante tenderà ad avere una maggiore densità di punti nelle zone con elementi circuitali composti da linee non prevalentemente orizzontali o verticali (figura 4.2f).

Tale risultato è poi soggetto ad una sfocatura gaussiana, un threshold e una dilatazione, in modo da generare aloni bianchi solo nelle suddette zone ad alta densità di punti (figure 4.2g e 4.2h).

Questa immagine sarà indicata come «immagine A» e sarà fondamentale in seguito.

**Rilevazione delle zone con punti terminali** L'immagine soggetta a pruning pesante è sottratta dall'immagine soggetta a thinning, in modo da ottenere un'immagine contenente le sole parti mancanti alla seconda (figura 4.2i), indicate in seguito come «punti terminali» del circuito.

L'immagine contenente i punti terminali è anch'essa soggetta a sfocatura gaussiana, threshold e dilatazione, in modo da generare un'immagine con aloni bianchi analoga all'immagine A (figure 4.2j e 4.2k). Questa immagine sarà indicata come «immagine B».

**Unione dei risultati parziali e generazione dei rettangoli** L'immagine A presenta aloni bianchi in corrispondenza di elementi circuitali ricchi di linee diagonali o curve, ad esempio resistenze e induttori. L'immagine B invece individua bene elementi formati da linee rette, ma con un'alta densità di punti terminali, ad esempio i condensatori, che ne hanno quattro.

Sovrapporre le immagini A e B risulta in un'immagine con aloni bianchi esattamente in corrispondenza degli elementi circuitali (figura 4.2l). Il metodo `ImgProc.findContours` di OpenCV, data tale immagine, restituisce i *contorni* dell'immagine, ovvero un array di array di punti; ognuno di questi array descrive il contorno di uno degli aloni bianchi.

Per ognuno degli insiemi di punti è possibile ottenere semplicemente il rettangolo che ne contiene tutti gli elementi usando il metodo `Imgproc.boundingRect`. I rettangoli ottenuti in questo modo sono quelli che delimitano gli elementi circuitali (figura 4.2m).

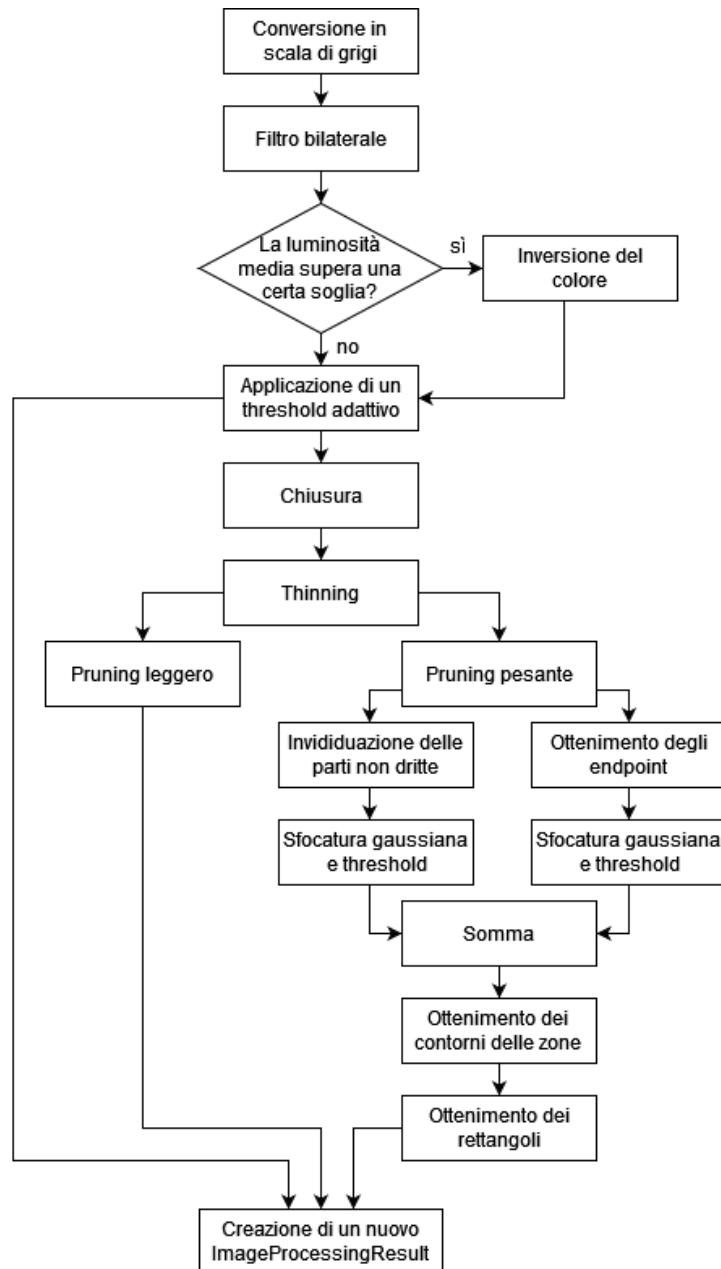


Figura 4.1: Diagramma di flusso che descrive l'algoritmo di localizzazione degli elementi circuituali

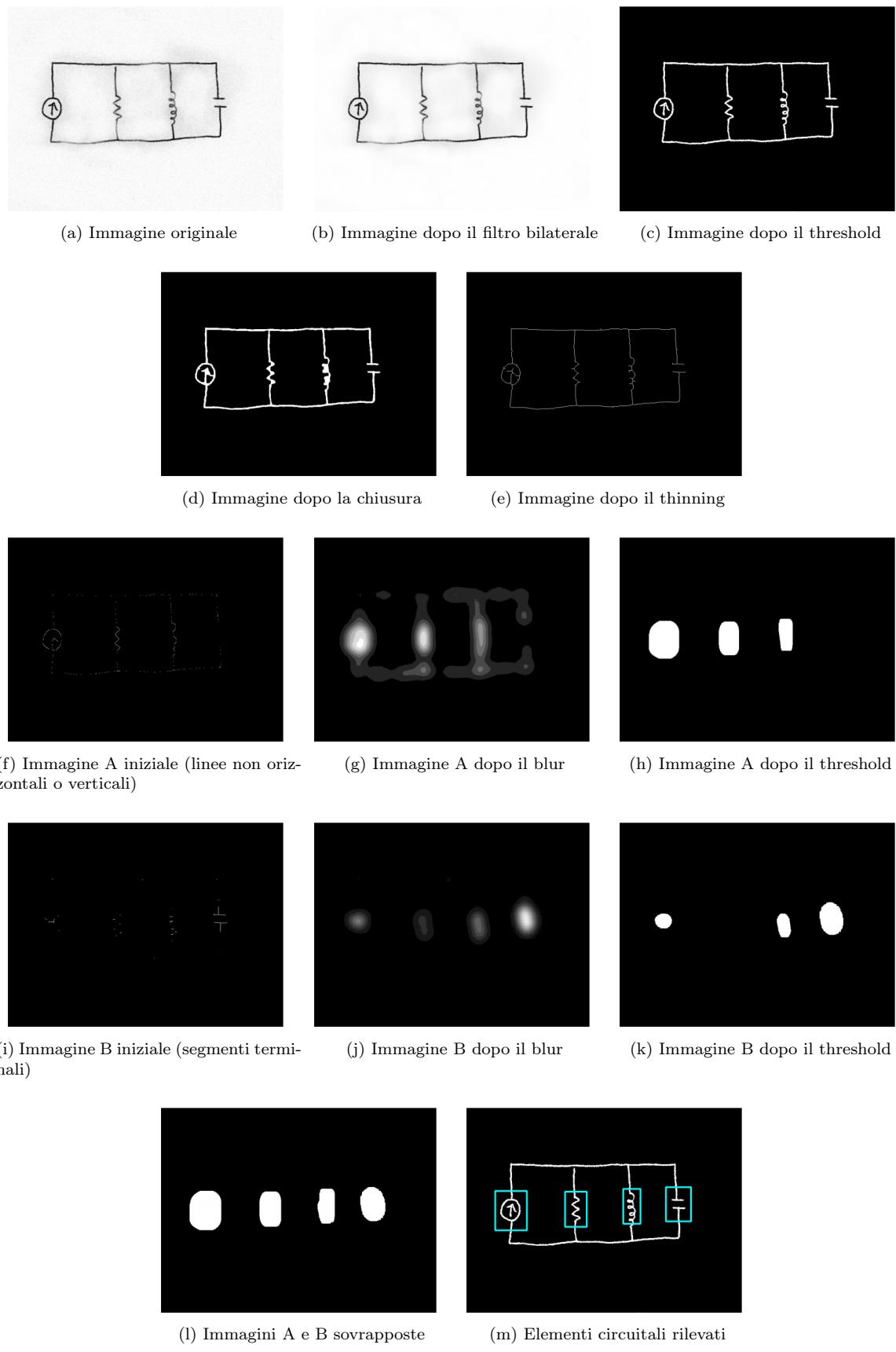


Figura 4.2: Risultati parziali delle elaborazioni descritte

L'implementazione in linguaggio Java del metodo `processImage` è mostrata di seguito.

```
public static ImageProcessingResult processImage(Mat imgIn) {
    Mat img = Mat.zeros(imgIn.size(), CvType.CV_8U);

    Imgproc.cvtColor(imgIn, img, Imgproc.COLOR_RGB2GRAY);

    Mat imgFiltered = new Mat();
    Mat imgThresh = new Mat();
    Mat imgClosed = new Mat();
    Mat imgThinned = null;
    Mat imgPruned = null;
    Mat outImgPruned = null;
    Mat preCurvedLines = null;
    Mat endpoints = new Mat();
    Mat elemPatches = new Mat();

    Imgproc.bilateralFilter(img, imgFiltered, 35, 10, 10);

    int avgColor = (int) (Core.mean(imgFiltered).val[0]);
    if (avgColor > 100) {
        Core.bitwise_not(imgFiltered, imgFiltered);
        avgColor = 255 - avgColor;
    }

    Imgproc.adaptiveThreshold(imgFiltered, imgThresh, 255,
        Imgproc.ADAPTIVE_THRESH_GAUSSIAN_C, Imgproc.THRESH_BINARY, 51, -20);

    // Closing
    Imgproc.morphologyEx(imgThresh, imgClosed, Imgproc.MORPH_CLOSE,
        Mat.ones(8, 8, CvType.CV_8U));

    // Thinning
    imgThinned = iterativeReverseHitmiss(imgClosed, kernelsThinning);

    // Light pruning (for output)
    outImgPruned = iterativeReverseHitmiss(imgThinned, kernelsPruning, 5);

    // Heavy pruning (for endpoints)
    imgPruned = iterativeReverseHitmiss(imgThinned, kernelsPruning, 22);

    // Get curved lines
    preCurvedLines = iterativeReverseHitmiss(imgPruned, kernelsStraight);
    Mat curvedLines = new Mat();
    Imgproc.GaussianBlur(preCurvedLines, curvedLines, new Size(101, 101), 0);
    Core.multiply(curvedLines, new Scalar(2), curvedLines);
    Imgproc.threshold(curvedLines, curvedLines, 6, 255, Imgproc.THRESH_BINARY);
    curvedLines = dilation(curvedLines, 12);

    // Get endpoints
    Core.subtract(imgThinned, imgPruned, endpoints);
    Imgproc.GaussianBlur(endpoints, endpoints, new Size(101, 101), 0);
    Core.multiply(endpoints, new Scalar(2), endpoints);
    Imgproc.threshold(endpoints, endpoints, 4, 255, Imgproc.THRESH_BINARY);
    curvedLines = dilation(curvedLines, 2);

    // Get patches
    Core.add(endpoints, curvedLines, elemPatches);
    elemPatches = dilation(elemPatches, 2);

    List<MatOfPoint> contours = new ArrayList<>();
    Mat hierarchy = new Mat();
    Imgproc.findContours(elemPatches, contours, hierarchy, Imgproc.RETR_EXTERNAL,
        Imgproc.CHAIN_APPROX_SIMPLE);

    List<Rect> rects = new ArrayList<>();
    List<Mat> mats = new ArrayList<>();

    int minArea = (int) (0.005 * imgIn.width() * imgIn.height());
    for (MatOfPoint contour : contours) {
        Rect rect = Imgproc.boundingRect(contour);
        if (rect.area() >= minArea) {
            rects.add(rect);
        }
    }
    return new ImageProcessingResult(rects, imgThresh, outImgPruned);
}
```

## 4.2 Localizzazione dei segmenti

Il metodo `detectLines` si occupa di trovare i segmenti che connettono tra loro gli elementi circuitali. Riceve in ingresso un’immagine da cui gli elementi circuitali sono stati già rimossi e restituisce una matrice in cui ogni riga contiene le due coppie di coordinate che caratterizzano un segmento.

Per raggiungere questo scopo, è stata utilizzata la classe `LineSegmentDetector` di OpenCV. Questa implementa l’algoritmo LSD per la rilevazione di segmenti all’interno di un’immagine [LineSegmentDetector].

Il metodo `createLineSegmentDetector` [LSDOpenCV] riceve in ingresso diversi parametri numerici, tra cui in particolare `ang_th`, la tolleranza a variazioni angolari delle linee. Questa è stata appositamente impostata ad un valore alto (70) rispetto a quello di default (22.5) al fine di ovviare alle imprecisioni nel disegno del circuito (infatti i segmenti disegnati a mano saranno tendenzialmente poco rettilinei).

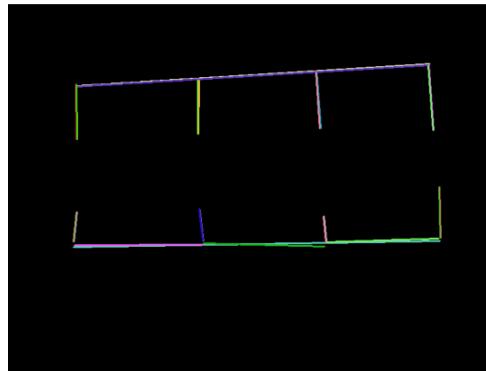


Figura 4.3: Segmenti rilevati nell’immagine in figura 4.2a

È evidente che i segmenti rilevati non sono ottimali per un utilizzo diretto da parte dell’applicazione, per i seguenti motivi:

- Tutte le linee sono duplicate: infatti l’algoritmo LSD fornisce i bordi rilevati e i segmenti che compongono il circuito ne hanno due;
- Nel caso di incroci di linee, LSD non si comporta in modo regolare: a volte i segmenti rilevati si interrompono ai punti di intersezione, altre volte no. Questa proprietà si può osservare nella figura 4.3, dove il segmento rettilineo inferiore è spezzato in più segmenti, mentre quello superiore no.
- Vertici che dovrebbero essere lo stesso sono vicini, ma non identici.

Tutti questi problemi sono risolti da componenti successivi dell’applicazione, in particolare dalla routine di generazione della rappresentazione logica del circuito, descritta nel capitolo 6.

# Capitolo 5

## Classificazione degli elementi circuituali

### 5.1 Creazione e allenamento del modello

Per la classificazione degli elementi circuituali è stata utilizzata una rete convoluzionale basata su Tensorflow e Keras.

Per l'allenamento è stato usato un dataset ottenuto da Kaggle, opportunamente modificato [Dataset]. È costituito da immagini  $120 \times 120$  di elementi circuituali disegnati a mano, in bianco su sfondo nero.

Le immagini hanno subito le seguenti trasformazioni:

- Sono state convertite da bitmap a PNG;
- Tramite uno script Python, ne sono state create varie versioni ruotate, dato che le immagini originali presentavano gli elementi solo con uno specifico orientamento. Questo è stato un passaggio fondamentale dato che l'applicazione deve riconoscere elementi circuituali orientati in vari modi, e deve dunque classificare diversamente i vari orientamenti. Ad esempio:
  - Il resistore ha due orientamenti possibili, dunque devono corrispondergli due classi, `resistor_r0` e `resistor_r1` (orizzontale e verticale rispettivamente).
  - Il generatore di corrente ha quattro orientamenti possibili, dato che oltre a poter essere orizzontale o verticale ha anche un verso. Ad esso corrispondono perciò quattro classi: `curr_src_r0`, `curr_src_r1`, `curr_src_r2` e `curr_src_r3` (destra, alto, sinistra, basso rispettivamente).
- Sono state eliminate delle irregolarità nel dataset iniziale, dato che esso conteneva rare immagini orientate nel verso opposto rispetto a tutte le altre dello stesso tipo.
- Come tecnica di data augmentation, tutte le immagini sono state duplicate, specchiandole: tutti gli elementi circuituali contemplati sono infatti simmetrici rispetto all'asse parallelo al filo.

La struttura del modello Tensorflow usato è la seguente:

```
model = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255., input_shape=(image_size[0], image_size[1], 1)),

    tf.keras.layers.Conv2D(16, 3, padding='same'),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Conv2D(32, 3, padding='same'),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.5),
```

```

    tf.keras.layers.Conv2D(64, 3, padding='same'),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Conv2D(128, 3, padding='same'),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Conv2D(256, 3, padding='same'),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4096),
    tf.keras.layers.ReLU(),
    tf.keras.layers.Dense(len(classes))
])

```

La rete è composta da diversi livelli:

- **Rescaling**: l'immagine di ingresso (già in scala di grigi) ha valori interi da 0 a 255; questo livello li converte in valori floating point da 0 a 1.
- Cinque livelli convoluzionali, che hanno lo scopo di effettuare una semplificazione delle immagini di input, composti a loro volta da sottolivelli:
  - **Conv2D**: svolge una convoluzione con un kernel  $3 \times 3$ , ovvero per ogni pixel svolge un prodotto scalare con i pixel vicini. Sono stati usati filtri di dimensione crescente (16, 32, 64, 128, 256) al ridursi delle dimensioni dei tensori nella rete.
  - **ReLU**: viene usata come funzione di attivazione; azzerà i valori negativi e lascia così come sono i valori positivi
  - **MaxPooling2D**: calcola una nuova matrice contenente i valori massimi di tutti i quadrati  $2 \times 2$  della matrice in ingresso; la matrice risultante avrà quindi un quarto della dimensione dell'originale (in questo modo viene ridotto il carico computazionale della rete)
  - **Dropout**: riduce l'overfitting azzerando parte delle celle della matrice in ingresso e normalizzando le altre celle in modo che la somma totale risulti uguale a prima; non è stato incluso nell'ultimo livello.
- **Flatten**: appiattisce la matrice in un array monodimensionale
- **Dense** con 4096 neuroni, che corrispondono alla dimensione del suo output
- **ReLU**
- **Dense** con numero di neuroni pari al numero di classi.

Al fine di fermare il modello prima dell'overfitting, è stato applicato un callback di *early stopping*:

```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10,
                                         restore_best_weights=True)
```

I grafici in figura 5.1 mostrano i risultati dell'allenamento. La validation accuracy ottenuta è stata di circa 0.94, un risultato più che accettabile considerando le caratteristiche del dataset (alcune classi

hanno elementi estremamente simili tra loro, ad esempio i generatori di tensione verso l'alto e verso il basso variano esclusivamente per la posizione dei simboli + e -).

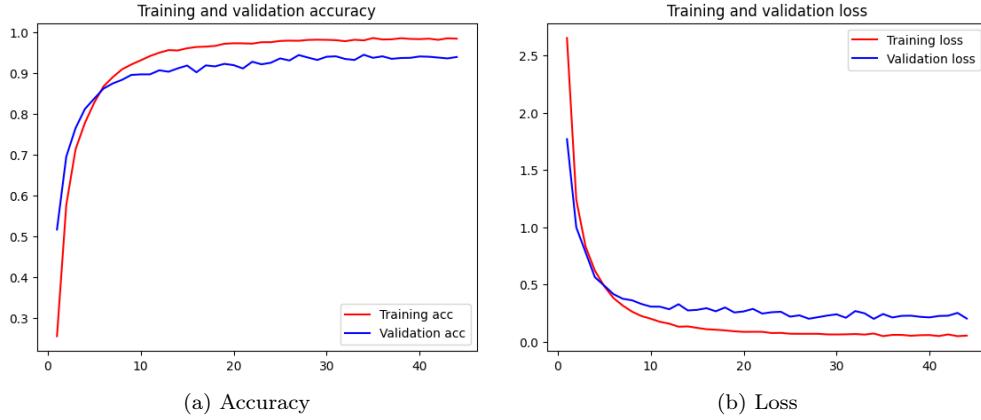


Figura 5.1: Risultati del training del modello

Una volta ottenuto il modello Tensorflow allenato, vi è stato aggiunto un livello **Softmax**, al fine di convertire i valori in output all'ultimo layer **Dense** (i *logit*) in opportuni valori di probabilità compresi tra 0 e 1:

```
model = tf.keras.models.load_model('classification_model_final.hdf5')
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])

probability_model.save('classification_model_final_prob.hdf5')
```

Infine è stata fatta una conversione in modello Tensorflow Lite, in modo da poter importare e usare il risultato nell'applicazione:

```
model = tf.keras.models.load_model('classification_model_final_prob.hdf5')
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
with open('classification_model_final_prob.tflite', 'wb') as f:
    f.write(tflite_model)
```

Il modello TensorFlow Lite aveva già dimensioni sufficientemente ridotte per un'applicazione mobile (circa 42MB), per cui non si è ritenuta necessaria un'ulteriore operazione di quantizzazione.

## 5.2 Utilizzo all'interno dell'applicazione

Nell'applicazione Android, il modello è gestito nel pacchetto **ml\_handling**, con le classi **MLController** e **CircuitElement**.

- **MLController** prevede metodi per adattare immagini ai requisiti del modello, sottoporre immagini al modello, ottenerne la corrispondente classificazione e per incapsulare i risultati in opportuni oggetti.
- **CircuitElement** incapsula un risultato di valutazione del modello, relativo ad un singolo elemento circuitale; ne verrà creata un'istanza per ogni elemento presente nel circuito fotografato.

Le immagini fornite al modello ritraggono gli elementi del circuito: sono sotto-immagini ritagliate dall'immagine *thresholded* vista in precedenza (figura 4.2c), in corrispondenza dei rettangoli individuati. Si

usa appositamente l'immagine thresholded dato che questa ha un aspetto molto simile a quello delle immagini del dataset.

Le sotto-immagini ritagliate sono prima sottoposte al metodo `preprocessImage`, che le ridimensiona a  $120 \times 120$ , mantenendone le proporzioni e riempiendo di nero l'eventuale area aggiunta. Questo le rende analoghe a quelle del dataset e quindi riconoscibili.

Successivamente, ogni immagine preprocessata è sottoposta al modello con il metodo `processImage`, il quale restituisce un array di valori di probabilità, uno per ogni classe del modello. Tra questi, il valore di probabilità maggiore corrisponde alla presunta identità dell'elemento ritratto.

Per ogni risultato ottenuto si crea dunque un'istanza di `CircuitElement` contenente informazioni utili, prime tra queste la presunta identità dell'elemento e le coordinate del corrispondente rettangolo, entrambe informazioni utili per la creazione della rappresentazione logica del circuito (capitolo 6).

```
public static float[] processImage(Mat img, Context context) {
    Mat preprocessed = preprocessImage(img);
    float[] imgBuf = new float[(int) preprocessed.total() * preprocessed.channels()];
    preprocessed.get(0, 0, imgBuf);

    float[] results = null;
    try {
        ClassificationModelFinalProb model =
            ClassificationModelFinalProb.newInstance(context);

        TensorBuffer inputFeature0 = TensorBuffer.createFixedSize(new int[]{1, 120,
            120, 1}, DataType.FLOAT32);
        inputFeature0.loadArray(imgBuf);

        ClassificationModelFinalProb.Outputs outputs = model.process(inputFeature0);
        TensorBuffer outputFeature0 = outputs.getOutputFeature0AsTensorBuffer();

        results = outputFeature0.getFloatArray();

        model.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    return results;
}

public static List<CircuitElement> processImages(Mat image, List<Rect> rects, Context
    context) {
    List<CircuitElement> out = new ArrayList<>(rects.size());
    for (Rect rect : rects) {
        Mat cropped = new Mat(image, rect);
        float[] probabilities = processImage(cropped, context);
        int bestGuessIndex = -1;
        for (int i = 0; i < probabilities.length; i++) {
            if (bestGuessIndex == -1 || probabilities[i] >
                probabilities[bestGuessIndex]) {
                bestGuessIndex = i;
            }
        }
        String className = getClassNameFromIndex(bestGuessIndex);
        out.add(new CircuitElement(className, rect));
    }
}
```

```
        out.add(new CircuitElement(probabilities, bestGuessIndex, className, rect));
    }
    return out;
}
```

## Capitolo 6

# Rappresentazione logica del circuito e generazione dell'output

Una volta ottenuti i risultati degli algoritmi descritti ai capitoli 4 e 5, è necessario costruire una rappresentazione logica del circuito, basata su segmenti e vertici di segmenti piuttosto che su pixel all'interno di una matrice.

Le informazioni a disposizione sono:

- Gli elementi circuitali, in particolare la posizione e la classe di ognuno
- I segmenti rettilinei, individuati con il LSD, che connettono gli elementi

Come è stato descritto in 1.2, CircuiTikZ prevede la costruzione del circuito mediante composizione di segmenti logici, e tratta gli elementi circuitali come un particolare tipo di segmento. Per questo motivo è conveniente costruire la rappresentazione logica usando segmenti come elemento unitario fondamentale. In particolare, è stato definito un modello object-oriented composto dalle seguenti classi:

- **Node**: descrive un punto all'interno del circuito; incapsula una lista dei **Segment** di cui è vertice;
- **Segment**: descrive un segmento logico; incapsula i suoi vertici come due **Node**;
- **Element**, sottotipo di **Segment**: descrive un elemento circuitale; in questo caso i due **Node** sono i due vertici a cui l'elemento circuitale è collegato (nel caso dell'elemento circuitale «terra» sarà presente solo un vertice)
- **Circuit**: descrive l'intero circuito, incapsulando una lista di **Segment** e una lista di **Node**. Per il corretto funzionamento di alcuni degli algoritmi che vengono utilizzati sugli oggetti **Circuit**, è necessario che i nodi siano gestiti come riferimenti univoci: non si possono avere più oggetti **Node** uguali che non siano lo stesso oggetto (ovvero due **Segment** diversi con un vertice in comune conterranno lo stesso riferimento ad un particolare oggetto **Node**, e non due copie separate dello stesso).

La classe **Circuit** offre un metodo factory statico **from** che prende in ingresso i dati citati sopra:

- Una lista di **CircuitElement**
- La matrice ottenuta in uscita dall'applicazione dell'algoritmo LSD

e restituisce un nuovo **Circuit**, creato applicando i passaggi descritti nelle sezioni a seguire.

## 6.1 Costruzione iniziale di nodi e segmenti

La prima operazione da eseguire è trasformare i segmenti restituiti dal LSD in oggetti `Segment` e `Node` opportunamente strutturati: si itera sui segmenti in ingresso e si trasforma ognuno in un `Segment` con due `Node`, facendo attenzione a non creare `Node` duplicati; se due `Segment` hanno un vertice in comune bisogna rilevarlo e assicurarsi che il `Node` creato sia soltanto uno, condiviso tra i due.

Inoltre, come già indicato in 4.2, i segmenti ricevuti in input sono una rappresentazione approssimativa e ridondante dei fili, dunque segmenti che logicamente sarebbero adiacenti possono risultare leggermente sconnessi. Questo aspetto può essere gestito riconoscendo come lo stesso nodo non solo vertici identici, ma anche vertici vicini, entro una soglia prestabilita.

Il codice Java che svolge questa elaborazione iniziale è il seguente:

```
public static Circuit from(Mat lines, List<CircuitElement> circuitElements) {
    List<Node> nodes = new ArrayList<>();
    List<Segment> segments = new ArrayList<>();

    for (int i = 0; i < lines.rows(); i++) {
        double[] line = lines.get(i, 0);
        int x1 = (int)line[0];
        int y1 = (int)line[1];
        int x2 = (int)line[2];
        int y2 = (int)line[3];
        Node n1 = null;
        Node n2 = null;
        for (Node n : nodes) {
            if (pointsAreClose(x1, y1, n.getX(), n.getY())) {
                n1 = n;
            }
            if (pointsAreClose(x2, y2, n.getX(), n.getY())) {
                n2 = n;
            }
        }
        if (n1 == null) {
            n1 = new Node(x1, y1);
            nodes.add(n1);
        }
        if (n2 == null) {
            n2 = new Node(x2, y2);
            nodes.add(n2);
        }
        Segment newSegment = new Segment(n1, n2);
        n1.addSegment(newSegment);
        n2.addSegment(newSegment);
        segments.add(newSegment);
    }
    ...
}
```

Il metodo `pointsAreClose()` verifica se due punti sono sufficientemente vicini sulla base di una soglia.

## 6.2 Risoluzione delle irregolarità in presenza di diramazioni

Come è stato osservato in 4.2, l'algoritmo LSD non si comporta in modo regolare in presenza di diramazioni: a volte individua segmenti separati, a volte no. Questo si risolve suddividendo segmenti rettilinei lungo i quali si trovano diramazioni in più segmenti *minimi*, senza diramazioni.

Per raggiungere questo scopo, è sufficiente, per ogni segmento:

- Verificare se ospita diramazioni; questo viene fatto verificando la presenza di nodi in una sottile area intorno al segmento. In caso di esito negativo, il segmento è già minimo, quindi si procede con il successivo, altrimenti non è minimo e va spezzato, dunque si prosegue con i passi seguenti.
- Il segmento non minimo viene eliminato.
- Vengono creati  $n + 1$  nuovi segmenti, con  $n$  numero dei nodi di diramazioni trovati nei pressi del segmento originale. Ognuno di questi nuovi segmenti connette il nodo  $i$ -esimo con il nodo  $i + 1$ -esimo, con  $i = 0, \dots, n + 1$ , dove quelli di indice 0 e  $n + 1$  sono i due nodi estremi del segmento originale. I nuovi segmenti creati saranno dunque tutti sottosegmenti disgiunti dell'originale.

Segue il codice Java che implementa questo algoritmo.

```
private static void splitSegmentsAtBranches(List<Segment> segments, List<Node> nodes) {
    int nSegments = segments.size();
    for (int i = 0; i < nSegments; i++) {
        Segment s = segments.get(i);
        Node node1 = s.getNode1();
        Node node2 = s.getNode2();
        int x1 = Math.min(node1.getX(), node2.getX());
        int x2 = Math.max(node1.getX(), node2.getX());
        int y1 = Math.min(node1.getY(), node2.getY());
        int y2 = Math.max(node1.getY(), node2.getY());

        //...

        List<Node> collidingNodes = new ArrayList<>();

        for (Node n : nodes) {
            if (n == node1 || n == node2)
                continue;
            if (n.getX() > x1 && n.getX() < x2 && n.getY() > y1 && n.getY() < y2) {
                collidingNodes.add(n);
            }
        }

        Comparator<Node> comparator;
        if (s.isVertical()) {
            comparator = Comparator.comparingInt(Node::getY);
            if (node1.getY() > node2.getY()) {
                Node tmp = node1;
                node1 = node2;
                node2 = tmp;
            }
        } else {
            // ... (Analogo a sopra, con x al posto di y)
        }

        collidingNodes.sort(comparator);

        if (collidingNodes.size() > 0) {
            for (int j = 0; j <= collidingNodes.size(); j++) {
                Node first, second;
                if (j == 0) first = node1;
                else first = collidingNodes.get(j - 1);

                if (j == collidingNodes.size()) second = node2;
                else second = collidingNodes.get(j);
            }
        }
    }
}
```

```

        Segment newSegment = new Segment(first, second);

        if (j == 0) node1.removeSegment(s);
        if (j == collidingNodes.size()) node2.removeSegment(s);

        segments.add(newSegment);
        first.addSegment(newSegment);
        second.addSegment(newSegment);
    }
    segments.remove(i--);
    nSegments--;
}
}
}

```

### 6.3 Rimozione dei segmenti duplicati

Come è stato già descritto in 4.2, frequentemente i segmenti rilevati dall'algoritmo LSD risultano duplicati. Inoltre l'algoritmo descritto nella sezione precedente può risultare in segmenti minimi duplicati in caso il LSD avesse originalmente rilevato segmenti anche solo parzialmente sovrapposti. Per questo motivo è necessario svolgere un passo di eliminazione dei segmenti duplicati.

### 6.4 Aggiunta degli elementi circuituali

Gli elementi circuituali vanno aggiunti sotto forma di nuovi segmenti, da connettersi automaticamente ai nodi già creati dai passi precedenti. È sufficiente, per ogni elemento circuitale, determinare quali tra i nodi già trovati siano situati «intorno» al rettangolo che circonda l'elemento. Una volta trovati, è sufficiente creare un nuovo `Element` (sottotipo di `Segment`) e aggiungerlo alla lista di segmenti del circuito.

Il codice che realizza questo algoritmo è il seguente:

```

private static void addCircuitElementsToSegments(List<Segment> segments, List<Node>
→ nodes, List<CircuitElement> circuitElements) {
    for (CircuitElement circuitElement : circuitElements) {
        Rect rect = circuitElement.getRect();
        Node n1 = null;
        Node n2 = null;
        for (Node n : nodes) {
            int direction = pointByRect(n.getX(), n.getY(), rect);
            if (direction != -1) {
                if (n1 == null) {
                    n1 = n;
                } else {
                    n2 = n;
                    break;
                }
            }
        }
        Element element = new Element(n1, n2, circuitElement.getBestGuessIndex());
        if (n1 != null)
            n1.addSegment(element);
        if (n2 != null)
            n2.addSegment(element);
        segments.add(element);
    }
}

```

Il metodo `pointByRect()` determina se un punto si trova approssimativamente sulla cornice di un rettangolo, sulla base di una soglia, e restituisce un valore nell'intervallo [0,3] che indica il lato del rettangolo in cui il punto si trova (0 destro, 1 superiore, 2 sinistro, 3 inferiore).

Eventuali elementi circuitali per cui è stato trovato *un solo nodo* verranno interpretati come «terra» dagli algoritmi successivi, ignorando la classificazione fornita dal modello di machine learning. Questa scelta causa l'aggiunta di elementi «terra» spuri in caso di errori precedenti, ma in compenso migliora la qualità dei risultati in caso di errori del modello (che sono frequenti a causa dell'irregolarità delle immagini degli elementi «terra» presenti nel dataset).

Dopo aver svolto questa operazione, il metodo `from` crea e restituisce il circuito.

## 6.5 Raddrizzamento dei segmenti

Fino ad ora non si è tenuto conto del fatto che i fili di un circuito disegnato a mano non sono quasi mai perfettamente orizzontali o verticali, ma un'applicazione che converte un tale circuito in un diagramma da mostrare in un documento dovrebbe indubbiamente eliminare queste imprecisioni. La classe `Circuit` offre un metodo `straighten` che svolge questa operazione.

Grazie alla scelta fatta in 6.1 di rappresentare nodi identici utilizzando lo stesso oggetto, questa operazione di raddrizzamento si riduce a pareggiare le coordinate di tutti i nodi che si trovano lungo ogni catena di segmenti adiacenti orientati approssimativamente allo stesso modo (orizzontalmente o verticalmente). In particolare le catene orizzontali possono essere raddrizzate in modo del tutto separato dalle catene verticali, per cui il metodo `straighten` in realtà non fa altro che chiamare due volte un altro metodo, `straightenSingleDim`, che applica il raddrizzamento ad un solo orientamento, fornитогli come argomento.

Assumendo di voler raddrizzare le catene verticali, l'algoritmo di `straightenSingleDim` consiste nelle seguenti operazioni:

- Ordinare i nodi in modo crescente secondo la loro ordinata; in questo modo, di ogni catena verrà sempre esaminato per primo il nodo che si trova al suo vertice superiore;
- Per ogni nodo *non già esplorato*:
  - Se ne esiste una, seguire la catena verso il basso iniziata da questo nodo e calcolare la media delle ascisse dei nodi incontrati;
  - Impostare l'ascissa di tutti i nodi incontrati lungo la catena alla media;
  - Marcare i nodi incontrati come *esplorati*: tutti i nodi della catena sono già allineati correttamente, e iterazioni successive non dovranno influire su di essi.

L'algoritmo funziona in modo analogo nel caso delle catene orizzontali. Segue il codice Java del metodo `straightenSingleDim`.

```
public void straightenSingleDim(int dimension) {
    List<Node> exploredNodes = new ArrayList<>();
    List<Node> sortedNodes = new ArrayList<>(this.nodes);
    sortedNodes.sort((n1, n2) -> dimension == 0 ?
        n1.getX() - n2.getX() : n1.getY() - n2.getY());

    int direction = dimension == 0 ? 0 : 3;

    for (Node n : sortedNodes) {
        if (exploredNodes.contains(n))
            continue;
        List<Node> connectedNodes = new ArrayList<>();

        Node nextNode = n;
```

```

int sum = 0;
do {
    sum += dimension == 0 ? nextNode.getY() : nextNode.getX();
    connectedNodes.add(nextNode);
    exploredNodes.add(nextNode);
} while ((nextNode = nextNode.getNodeOnDirection(direction)) != null);

int average = sum / connectedNodes.size();

for (Node nn : connectedNodes) {
    if (dimension == 0)
        nn.setY(average);
    else
        nn.setX(average);
}
}
}
}

```

## 6.6 Generazione del codice CircuiTikZ

Una volta ottenuta la rappresentazione logica del circuito con i segmenti raddrizzati, prima di poterne effettivamente generare il codice CircuiTikZ è necessario invertire le ordinate di tutti i nodi: infatti CircuiTikZ prevede un asse delle ordinate orientato dal basso verso l'alto, mentre finora è stato considerato dall'alto verso il basso. Per svolgere questa operazione la classe `Circuit` offre un metodo `invertY`, che riceve in ingresso l'altezza dell'immagine originale:

```

public void invertY(int height) {
    for (Node n : nodes) {
        n.setY(height - n.getY());
    }
}

```

La generazione del codice CircuiTikZ è svolta dal metodo `toCircuiTikZ`:

```

public String toCircuiTikZ() {
    StringBuilder strb = new
        StringBuilder("\\begin{circuitikz}[american,x=0.01cm,y=0.01cm]\\n");
    for (Segment segment : segments) {
        String draw = segment.getCircuiTikZDraw();
        if (draw != null)
            strb.append(draw);
    }
    strb.append("\\end{circuitikz}");
    return strb.toString();
}

```

Il metodo `getCircuiTikZDraw()` della classe `Segment` restituisce, sotto forma di stringa, la riga di codice contenente l'istruzione CircuiTikZ `\draw` necessaria per disegnare tale segmento (che sia un filo o un elemento circuitale).

# Capitolo 7

## Collaudi

L'applicazione è stata collaudata disegnando diversi circuiti (non necessariamente significativi in ambito elettrotecnico) che contenessero in modo variegato gli elementi circuitali previsti.

Nelle figure di seguito, per ogni test sono mostrati:

- L'immagine del circuito così come è stato fotografato;
- Il circuito dopo l'operazione di threshold adattivo;
- I segmenti rilevati dall'algoritmo LSD;
- Il circuito ottenuto compilando il codice CircuiTikZ restituito dall'applicazione.

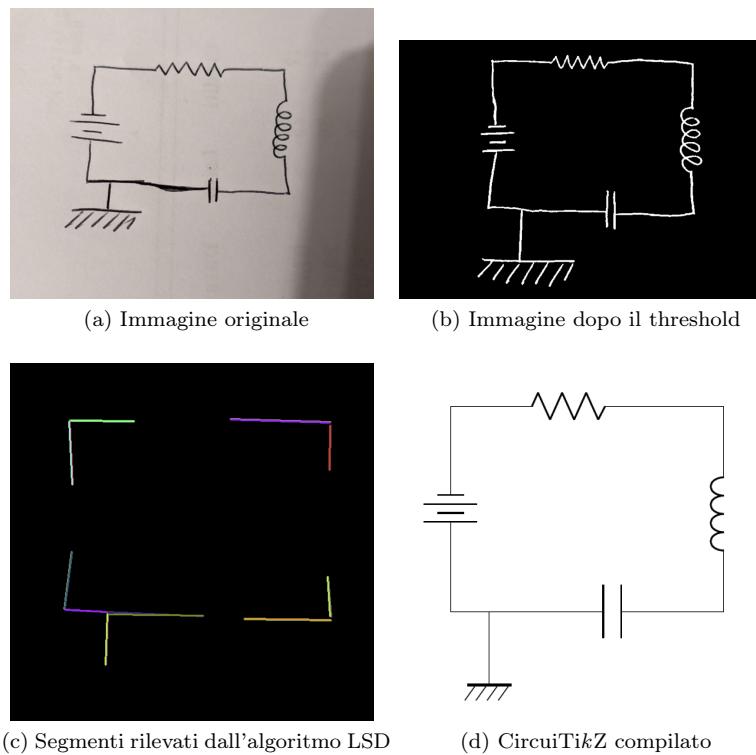


Figura 7.1: Primo test

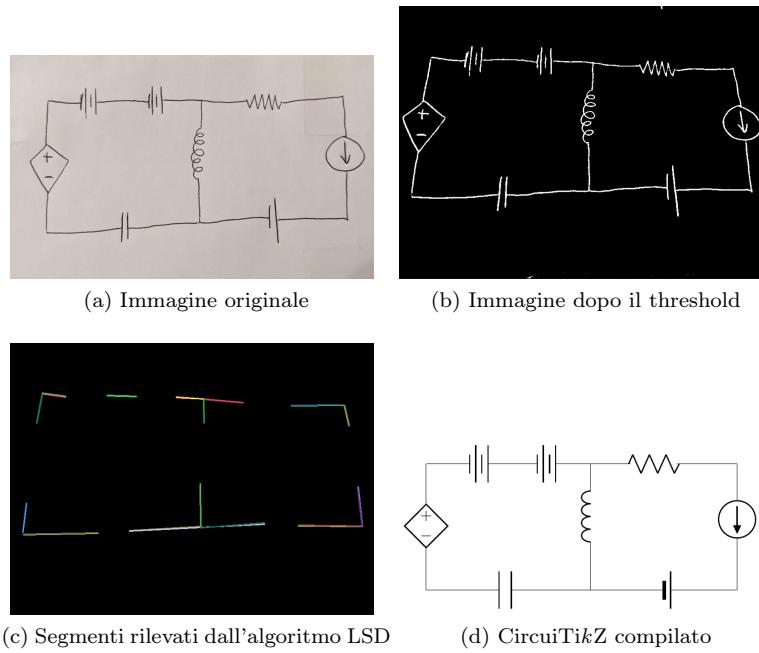


Figura 7.2: Secondo test

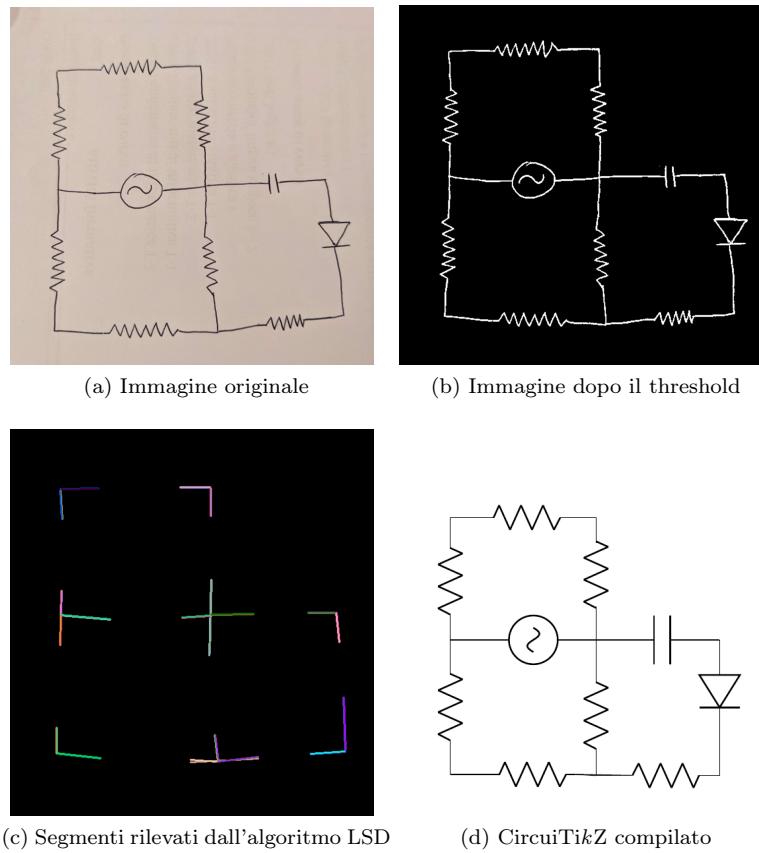


Figura 7.3: Terzo test

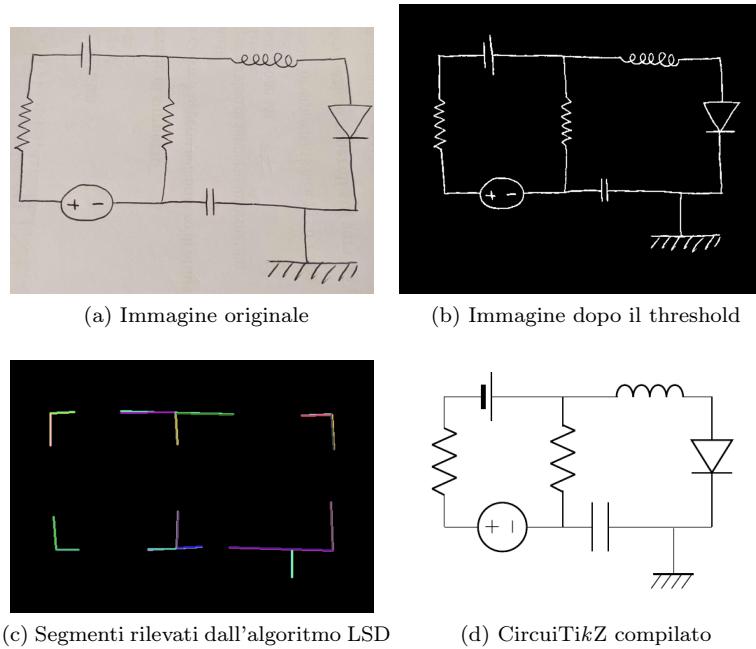


Figura 7.4: Quarto test

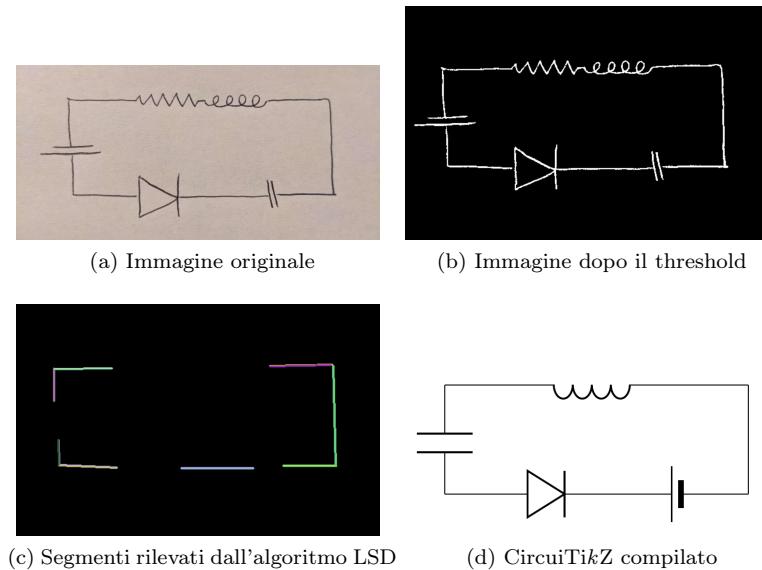


Figura 7.5: Quinto test. Si osservino i due errori del modello di machine learning (il generatore di tensione e il condensatore sono scambiati) e l'errore dell'algoritmo di localizzazione degli elementi circuitali, causato dal resistore e dall'induttore troppo vicini.

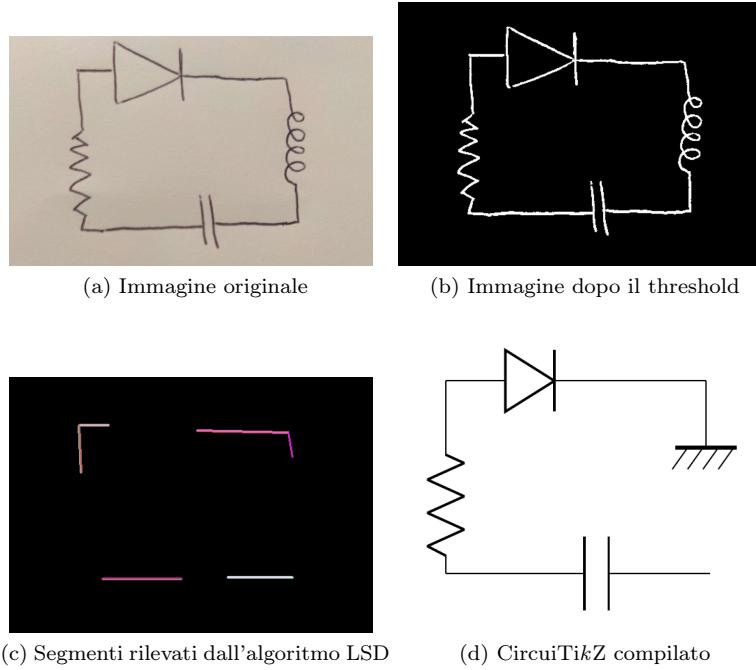


Figura 7.6: Sesto test. Si osservi l'errore dell'algoritmo di localizzazione degli elementi (che ha incluso nel rettangolo il cambio di direzione del filo), il conseguente errore di localizzazione dei segmenti e quindi l'inserimento di una terra spuria (come descritto in 6.4).

A titolo esemplificativo, segue il codice dell'esempio in figura 7.4.

```
\begin{circuitikz}[american,x=0.01cm,y=0.01cm]
\draw (361, 485) to [short] (531, 485);
\draw (106, 709) to [short] (70, 709);
\draw (465, 439) to [short] (465, 483);
\draw (368, 709) to [short] (182, 709);
\draw (70, 485) to [short] (113, 485);
\draw (531, 485) to [short] (531, 563);
\draw (227, 485) to [short] (316, 485);
\draw (274, 485) to [short] (227, 485);
\draw (70, 485) to [short] (70, 539);
\draw (316, 485) to [short] (274, 485);
\draw (274, 548) to [short] (274, 485);
\draw (531, 709) to [short] (480, 709);
\draw (531, 650) to [short] (531, 709);
\draw (70, 656) to [short] (70, 709);
\draw (274, 658) to [short] (274, 709);
\draw (182, 709) to [short] (274, 709);
\draw (274, 709) to [short] (368, 709);
\draw (465, 439) node[eground] {};
\draw (361, 485) to [C] (316, 485);
\draw (113, 485) to [V] (227, 485);
\draw (531, 563) to [diode,invert] (531, 650);
\draw (70, 539) to [R] (70, 656);
\draw (274, 548) to [R] (274, 658);
\draw (368, 709) to [L] (480, 709);
\draw (106, 709) to [battery2,invert] (182, 709);
\end{circuitikz}
```

# Capitolo 8

## Conclusione

Come si può appurare dal capitolo precedente, gli algoritmi utilizzati dall'applicazione si comportano abbastanza bene, a patto che si seguano gli assunti fatti in 2.3.

- Certi vincoli potrebbero essere alleviati modificando attentamente alcuni parametri degli algoritmi utilizzati dall'applicazione, in particolare del rilevatore delle posizioni degli elementi circuitali (ad esempio ridurre le dimensioni dei kernel dei blur gaussiani sulle immagini A e B ridurrebbe la distanza minima ammessa tra gli elementi circuitali, ma aumenterebbe il rischio che un singolo elemento venga riconosciuto come due elementi diversi).
- Altri non possono essere eliminati senza ideare algoritmi del tutto nuovi (ad esempio l'obbligo di utilizzare solo segmenti orizzontali o verticali).

La rete neurale che classifica gli elementi circuitali è stata allenata su un dataset le cui caratteristiche non combaciano sempre perfettamente con quelle degli elementi circuitali che le vengono forniti dall'applicazione durante l'esecuzione. La soluzione più ovvia al problema sarebbe di costruire un nuovo dataset basato effettivamente su circuiti disegnati a cui vengano applicate le stesse trasformazioni che l'applicazione utilizza per estrarre gli elementi circuitali.

Un altro sviluppo futuro potrebbe essere il miglioramento del codice TeX restituito in output traendo vantaggio della sintassi abbreviata offerta da CircuiTikZ descritta nell'introduzione.

Nonostante queste limitazioni, i risultati ottenuti sono piuttosto soddisfacenti. Grazie allo sviluppo di PhotoCircuit abbiamo avuto l'opportunità di cimentarci in ambiti che non avevamo mai toccato prima, come lo sviluppo di applicazioni *mobile*, la computer vision e il machine learning.

# Bibliografia

[AdaptiveThreshold]	adaptiveThreshold(), documentazione di OpenCV: <a href="https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#ga72b913f352e4a1b1b397736707afcde3">https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#ga72b913f352e4a1b1b397736707afcde3</a>
[BilateralFilter]	bilateralFilter(), documentazione di OpenCV: <a href="https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga9d7064d478c95d60003cf839430737ed">https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga9d7064d478c95d60003cf839430737ed</a>
[Closure]	R. Fisher, S. Perkins, A. Walker and E. Wolfart, <i>Closing</i> , <a href="https://homepages.inf.ed.ac.uk/rbf/HIPR2/close.htm">https://homepages.inf.ed.ac.uk/rbf/HIPR2/close.htm</a> , 2003, Edinburgh University
[Dataset]	Mahmoud Goudy, hossamalaa99, Muhammad Alaa, Hand-drawn Electric Circuit Schematic Components, <a href="https://www.kaggle.com/datasets/moodrammer/handdrawn-circuit-schematic-components">https://www.kaggle.com/datasets/moodrammer/handdrawn-circuit-schematic-components</a>
[LineSegmentDetector]	Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, Gregory Randall, <i>LSD: a Line Segment Detector</i> , <a href="https://www.ipol.im/pub/art/2012/gjmr-lsd/">https://www.ipol.im/pub/art/2012/gjmr-lsd/</a> , 2012, Image Processing Online
[LSDOpenCV]	createLineSegmentDetector(), documentazione di OpenCV: <a href="https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga26413521fde978c0c97f11149336f6e1">https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga26413521fde978c0c97f11149336f6e1</a>
[ThinningPruning]	R. Fisher, S. Perkins, A. Walker and E. Wolfart, <i>Thinning</i> , <a href="https://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm">https://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm</a> , 2003, Edinburgh University