

Optical Music Recognition - Improving End-to-End and pipeline approaches

Francesco Magnani

15 maggio 2024

Indice

1	Introduzione	2
1.1	OMR pipeline	2
1.1.1	Image pre-processing	2
1.1.2	Riconoscimento dei simboli musicali	3
1.1.3	Ricostruzione della sintassi e rappresentazione finali	4
1.2	Approcci End-To-End	4
1.3	Limitazioni	4
1.4	Descrizione del progetto	4
2	Augmented End-To-End	5
2.1	Il modello	5
2.2	Obiettivo	5
2.3	Struttura della soluzione	5
2.4	IMG2DOC	6
2.4.1	Deformazione prospettica (warp)	7
2.4.2	Rilevazione dei quadrilateri	7
2.4.3	Pulizia dell'immagine (clean)	9
2.5	DOC2SEGMENTS	11
2.5.1	Bounding Box	11
2.5.2	Clustering dei righi musicali	11
2.5.3	Split segments	12
2.6	Utilizzo del modello	13
2.7	Risultati	13
2.7.1	Analisi delle prestazioni	14
2.7.2	Modello poco performante	14
2.7.3	Simboli non riconoscibili o malformati	15
2.7.4	Confusione degli spazi	15
2.7.5	<i>Bias</i> del modello	15
2.7.6	Soglie troppo intense sui simboli	15
2.7.7	Inclinazione dello spartito	16
2.7.8	Pre-processing non generalizzante	16
3	Staff Line removal	17
3.1	Possibili approcci	17
3.2	Preparazione del dataset	17
3.3	Training	18
3.4	Risultati	19
3.4.1	Applicazione e Post-Processing	19
4	Conclusione	22

1 Introduzione

Il problema dell'*Optical Music Recognition (OMR)* è un problema molto complesso che consiste nel leggere automaticamente spartiti scansionati/fotografati al fine di convertirli in un formato elettronico, come ad esempio il MIDI¹. L'OMR risulta essere molto collegato al problema dell'*Optical Character Recognition (OCR)*: entrambi hanno a che fare con la segmentazione e "traduzione" di simboli grafici, che vanno considerati in sequenza, anche se il secondo è più strettamente riferito al riconoscimento dei singoli caratteri. Le tecniche dell'OCR, tuttavia, non possono essere utilizzate nel contesto dell'OMR, in quanto gli la notazione musicale utilizzata negli spartiti presenta una struttura **bidimensionale** [1]. I simboli musicali, infatti, assumono diverso significato a seconda dell'ordine così come all'altezza rispetto alle linee del pentagramma. La complessità dell'OMR, però, non si riduce a quello: molte sono le problematiche collegate ad esso [2], per esempio:

- **Inter-conessione tra simboli:** i simboli musicali risultano essere spesso collegati tra loro, avendo le linee del pentagramma "incastrate" in mezzo ai simboli così come i simboli stessi collegati ad altri mediante legature o travature orizzontali², cambiando o no la semantica a seconda del contesto.
- **Variabilità dei simboli:** i simboli musicali possono cambiare di molto il proprio aspetto, non solo tra spartiti diversi, che possono appartenere ad editori diversi e quindi a notazioni grafiche differenti, ma anche *all'interno dello stesso spartito*[2]. Questo porta ovviamente ad ulteriore ambiguità, e maggiore difficoltà specialmente nella fase di riconoscimento dei simboli.
- **Ambiguità della notazione:** la notazione musicale è molto flessibile e cambia spesso anche in base al contesto storico dello spartito. Certi simboli riferiti a spartiti di una certa epoca storica possono assumere un significato diverso dai medesimi simboli applicati in contesto moderno. Non solo: alcuni simboli nello spartito assumono un significato diverso a seconda della presenza o meno di altri simboli presenti molto prima all'interno dello spartito, il che rende un'analisi locale molto più difficile.

Questi sono solo alcuni dei problemi riscontrabili durante l'analisi di spartiti musicali. Se si aggiunge che solo recentemente sono stati ottenute grosse collezioni di spartiti digitalizzati associati ad un formato elettronico per la memorizzazione della semantica musicale [3], tramite strumenti come **Sibelius** (<https://www.avid.com/en/sibelius>) e **Musescore** (<https://musescore.org>), appare chiaro quanto ancora l'OMR sia un problema aperto e di ricerca, pronto ad essere affrontato, per esempio, con l'utilizzo delle sempre più promettenti reti neurali.

1.1 OMR pipeline

Essendo un problema affrontato ormai da diversi anni, tante soluzioni sono state proposte in letteratura. Tradizionalmente, l'architettura con cui si affronta l'OMR segue una struttura a **pipeline** [4], dove diverse sotto-fasi, partendo dall'immagine di input, ottengono diversi modelli che insieme portano alla semantica musicale racchiusa nello spartito. Questa architettura è stata proposta in diverse versioni nel corso degli anni, ma una generale può essere riassunta nello schema in Figura 1.

Nei vari studi sull'OMR sono state adottate tecniche che vanno ad ampliare o modificare questa architettura. Per esempio, alcuni modelli [2] hanno utilizzato tecniche di apprendimento supervisionato dei simboli e modelli *fuzzy* per affrontare la variabilità dei simboli musicali, che quindi andassero a coinvolgere la sintassi musicale anche nel riconoscimento dei simboli.

1.1.1 Image pre-processing

L'analisi e modifica iniziali delle immagini si rivelano necessarie in modo tale da rendere più facili le fasi successive della pipeline. Un esempio è la correzione della rotazione delle immagini, dovuta per esempio ad una scansione imperfetta oppure ad una pagina piegata. Questa correzione è necessaria per

¹Protocollo standard per la comunicazione tra strumenti musicali elettronici, utilizzato anche come formato di file per la memorizzazione di tracce musicali

²Una legatura tra due note può avere diversi significati, come per esempio la prolungazione di un suono (*legatura di valore*). Le travature sono invece simboli grafici posti tra note per una più facile lettura grafica.

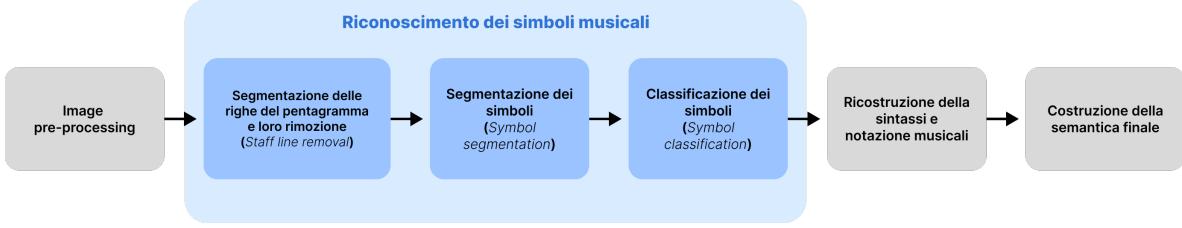


Figura 1: Tipica architettura utilizzata nei sistemi OMR, adattata da [4]

la fase di *Staff line removal*³, in quanto le tecniche più studiate (prima dell'avvento di tecniche in grado di generalizzare grazie all'utilizzo di reti neurali artificiali) per questo passaggio sfruttano algoritmi per la ricerca di righe orizzontali e verticali nell'immagine scansionata, per esempio la **trasformata di Hough** [2].

Altre trasformazioni si rivelano necessarie e/o utili per la segmentazione e classificazione dei simboli successive, per esempio binarizzazione e utilizzo di filtri, spesso per correggere problemi di stampa nei fogli digitalizzati.

1.1.2 Riconoscimento dei simboli musicali

Il riconoscimento dei simboli musicali è stato tradizionalmente suddiviso in tre sottofasi: **staff line removal**, **symbol segmentation** e **symbol classification**.

Il primo, lo *Staff line removal*, è un problema molto affrontato in letteratura, che può essere associato a quello della **semantic segmentation**, includendo però, ovviamente, non solo la segmentazione delle linee del pentagramma ma anche la loro successiva rimozione dell'immagine. Come già detto, tecniche come la trasformata di Hough possono essere applicate in questo caso, a patto che la fase di pre-processing sia sufficientemente robusta. In molti casi queste tecniche si rivelano, tuttavia, insufficienti perché richiedono, appunto, di eliminare prematuramente qualsiasi tipo di imperfezione dell'immagine, passaggio non così semplice e difficile da generalizzare. Per questo motivo sono stati provati, in tempi recenti, anche metodi che fanno uso di reti neurali come le **CNN** [5] o addirittura le **GAN** [6].

La rimozione delle linee del pentagramma dall'immagine è necessaria per rendere agile il successivo riconoscimento dei simboli, che appaiono a quel punto isolati in maniera più o meno netta (come si può notare in figura 2): esistono esempi [7][8] di sistemi OMR che saltano questa fase, sfruttando altre caratteristiche dell'immagine dello spartito per individuare i simboli, ma sono comunque una minoranza e mostrano risultati peggiori.

Nonostante la suddivisione tra i simboli ottenuta grazie a questa rimozione, tuttavia, molti simboli risultano ancora legati, come le travature orizzontali delle note da un ottavo, un sedicesimo ecc., motivo per cui non è possibile utilizzare tecniche banali, come ad esempio algoritmi per le componenti connesse, ma è necessaria un'ulteriore trasformazione che invece si basa sulle linee verticali per eliminare punti di connessione tra simboli diversi (ottenendo infine un'immagine simile alla figura 2c).



Figura 2: Spartito di J.S. Bach estratto dal dataset Rebelo [4]

La *Symbol segmentation* e *Symbol classification* sono riassumibili in un particolare caso di **Object Detection**. I primi approcci si basavano su tecniche come il *pattern matching* per ottenere la posizione dei simboli, seguite ovviamente da una fase di classificazione del singolo simbolo con classificatori già noti. Utilizzando reti neurali, invece, si può delegare a queste ultime una o entrambi le due fasi,

³Le *staff line* sono le linee presenti nel pentagramma, sia le 5 righe dove le note vengono collocate, sia per esempio divisorii verticali che spesso non hanno un valore semantico

utilizzando modelli già utilizzati per la Object detection o **Instance segmentation** (come UNet [9] per esempio).

1.1.3 Ricostruzione della sintassi e rappresentazione finali

Una volta implementata la precedente parte di questa architettura, è possibile utilizzare altre tecniche per sfruttare quanto appreso e costruire una **sintassi** dello spartito, in modo tale da verificare regole di base della notazione musicale e/o sfruttarle a nostro vantaggio per collocare i simboli in una sequenza possibile. Parte fondamentale di questa fase è anche **ricavare l'altezza di ogni nota**, analizzando per esempio la "testa" della nota⁴ e la sua posizione relativa rispetto alle linee del pentagramma precedentemente rilevate.

1.2 Approcci End-To-End

Recentemente, grazie ai recenti avanzamenti nel campo dell'intelligenza artificiale, nuovi approcci chiamati *End-To-End* sono possibili. Utilizzando certi tipi di reti neurali, come le **reti ricorrenti**, associate a particolari tipi di training [10] è possibile affidare alla rete **tutto il processo dell'OMR**, senza ricorrere ad una struttura a pipeline, quindi fornendo in input l'immagine dello spartito e ricevendo una rappresentazione (per esempio una lista di indici dei simboli presi da un vocabolario stabilito in partenza [3]) delle informazioni musicali contenute nello spartito in output.

1.3 Limitazioni

I sistemi studiati funzionano, fino a un limitato livello di accuratezza, su spartiti scansionati, senza particolari difetti nell'immagine, e soprattutto raggiungono una precisione accettabile solo su spartiti **monofonici**⁵, che rappresentano ovviamente una piccola parte della totalità di spartiti musicali presenti.

Oltre a questo, va fatta una distinzione tra sistemi **on-line** e **off-line**. I primi analizzano lo spartito direttamente producendo risultati quasi istantaneamente, focalizzandosi sull'aspetto *real-time* del riconoscimento, mentre i secondi analizzano delle scansioni complete di spartiti musicali, elaborando più a fondo e consumando più tempo e risorse. Le prestazioni di sistemi *on-line* risultano spesso scarse, in quanto si trovano a lavorare con spartiti spesso in più bassa risoluzione (per tenere il passo con la natura in tempo reale), senza contare la loro impossibilità di considerare aspetti "globali" dello spartito, dato che l'analisi è continua e si concentra solo su porzioni locali. Oltretutto, il problema dell'OMR richiede molte risorse computazionali, motivo per cui persino i sistemi *off-line* spesso ricorrono a scappatoie.

Infine, già da diversi anni si studiano sistemi OMR per l'analisi di partiture **manoscritte**, in cui la complessità, come si può immaginare, sale vertiginosamente rispetto agli spartiti impaginati tramite software per la composizione, vista la varietà ancora più grande di rappresentazioni dei simboli e l'imprecisione nella scrittura.

1.4 Descrizione del progetto

L'obiettivo di questo progetto è esplorare il campo dell'OMR, andando ad costruire **due parziali implementazioni dei processi precedentemente illustrati**, ovvero l'approccio a pipeline e quello End-To-End. Nello specifico, verranno effettuati due studi separati, uno chiamato **Augmented End-To-End** (Sezione 2) e un altro chiamato **Staff Line removal** (Sezione 3): il primo servirà a sfruttare modelli già addestrati per l'OMR End-To-End con l'aggiunta di una fase di pre-processing, il secondo, invece, consisterà nell'addestramento di reti neurali per effettuare la parte di Staff Line removal, introdotta nella Sezione 1.1.2.

Riassumendo le fasi che questo progetto andrà a toccare sono quelle evidenziate in Figura 3.

⁴La piccola parte ovale della note, una delle parti insieme a *corpo* e *coda*

⁵Spartiti senza note sovrapposte, dove esiste una sola "voce". La musica monofonica (o monodica) si contrappone a quella polifonica dove, invece, esistono due o più voci indipendenti sovrapposte.

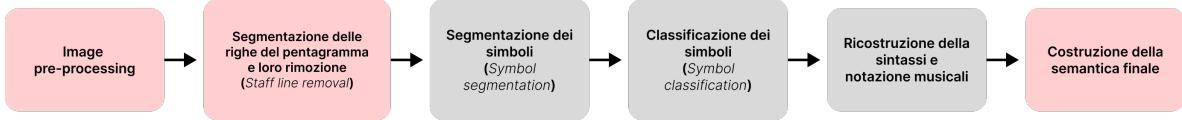


Figura 3: Fasi della pipeline esplorate dal progetto

2 Augmented End-To-End

In questa fase del progetto (d'ora in poi abbreviata AE2E), verranno esplorate reti neurali già disponibili e pre-addestrate per il riconoscimento ottico di spartiti musicali **monofonici** e **impaginati** (quindi non manoscritti). Nello specifico, verrà ripreso il lavoro svolto da Jorge Calvo-Zaragoza e David Rizo [3] in questa materia, portando l'utilizzo del loro modello in nuovi contesti, per esempio quello di **fotografie di spartiti**.

2.1 Il modello

La rete neurale creata da Zaragoza e Rizo è una CRNN (*Convolutional Recurrent Neural Network*), ovvero una rete composta prima da un blocco *convoluzionale* e successivamente uno *ricorrente*. La parte convoluzionale è ispirata alla rete VGG [12], mentre quella ricorrente è composta principalmente da unità BLSTM (ovvero *bi-directional long short-term memory*)⁶. Per guidare il training viene utilizzata la **Connectionist Temporal Classification** (CTC) [10], particolarmente adatta per questo task.

Il modello è pensato per convertire **piccole porzioni di spartito**, tipicamente un paio di battute e un numero massimo di simboli. È stato osservato che, aumentando la vicinanza e il numero dei simboli presenti nell'immagine, le prestazioni iniziano a calare⁷.

2.2 Obiettivo

Siccome il modello non è pensato per funzionare su immagini **catturate tramite foto**, l'obiettivo di questa parte di progetto è **espandere le sue capacità**, effettuando una fase di *pre-processing* sulle foto, in modo che queste diventino trattabili, e successivamente comparare le prestazioni sulla controparte digitale. Questa fase può essere riassunta nei seguenti punti:

- **Scannerizzare documenti:** dato l'immagine fotografata di uno spartito, effettuare una trasformazione che elimini difetti, condizioni di luce particolari ecc. che rendono la fotografia troppo diversa da una scannerizzazione
- **Trasformazione prospettica:** l'inclinazione e orientamento dell'immagine fotografata potrebbero renderla inutilizzabile. È necessario, quindi, un modo per rilevare i contorni del documento all'interno della foto e, tramite trasformazioni geometriche, ottenere una vista *Bird-Eye*, ovvero frontale al documento.
- **Segmentazione:** il modello lavora solo con frammenti di spartito, non con spartiti interi. Suddividere, quindi, lo spartito in piccoli frammenti trattabili dal modello è un ulteriore passo richiesto.

2.3 Struttura della soluzione

In Figura 4 viene riassunto il flusso dell'applicativo. Ogni singola fase verrà affrontata singolarmente.

Il codice è stato organizzato in una struttura simile al diagramma in Figura 5.

⁶Più altri livelli, maggiori dettagli sono ritrovabili nel relativo articolo [3]

⁷È opportuno specificare, inoltre, che il modello è stato addestrato su immagini del PrIMuS dataset [13], immagini generate artificialmente e spesso ad **alte risoluzioni**, motivo per cui il modello non è ancora adatto a casi d'uso reali

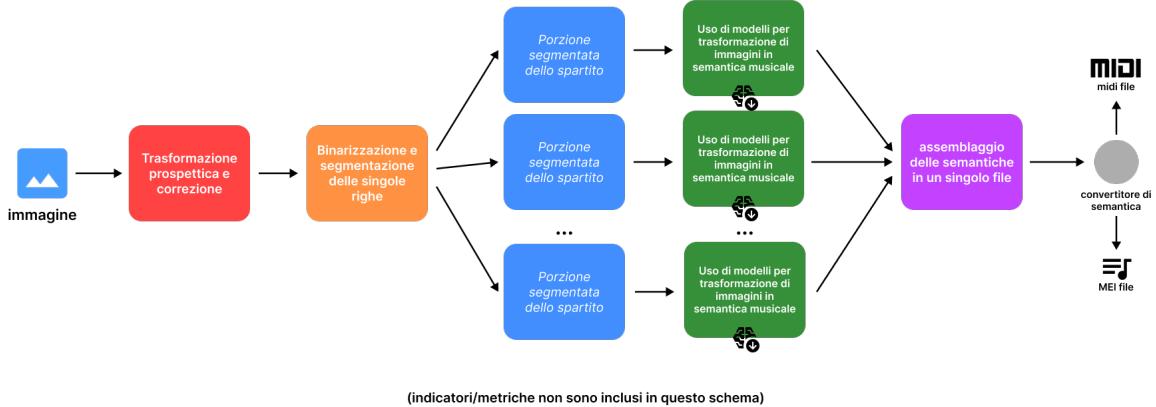


Figura 4: Visualizzazione del flusso AE2E

Il package `core` contiene le funzionalità che rappresentano l’obiettivo principale del progetto, mentre `semantic` ciò che ha più a che fare con l’utilizzo del modello e gestione della semantica musicale prodotta dal modello.

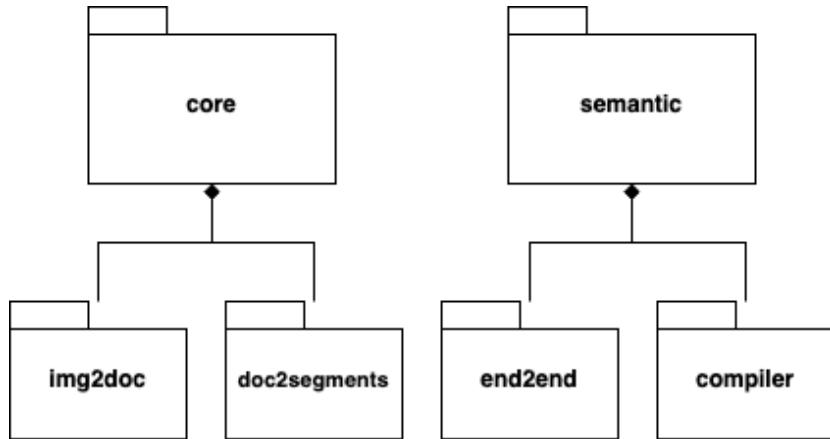


Figura 5: Struttura del progetto AE2E

2.4 IMG2DOC

Nel package `img2doc` sono contenute un insieme di funzioni il cui scopo è trasformare una fotografia dello spartito in un "documento" (`doc`). La principale funzione utilizzabile dall’esterno è la funzione `scan` contenuta in `scanner.py` che, data un’immagine e un insieme di parametri standard, ritorna l’immagine del documento scannerizzato. Quanto svolto dentro questo package riprende molto del lavoro svolto da Andrew Campbell [14] per creare uno scanner di documenti con la libreria OpenCV di Python, ma con l’obiettivo di migliorarlo ed estenderlo.

La funzione `scan` può essere riassunta nel seguente pseudo-codice:

```

1 def scan(image: Image): Image
2     rescale(image)
3     warp(image)
4     clean(image)
5     return image
  
```

Per poter lavorare con immagini di dimensioni variabili, viene prima eseguito un *rescaling* dell'immagine ad una data altezza, mantenendo il rapporto larghezza/altezza. Questo viene svolto da `rescale`. Successivamente avviene la fase **warp**.

2.4.1 Deformazione prospettica (warp)

Per poter modificare il documento come descritto è necessario effettuare una *Perspective Transform* utilizzando la relativa funzione della libreria OpenCV. In questo modo, una volta ottenuta la matrice della trasformazione da applicare, la funzione `warpPerspective` di OpenCV permette di ottenere l'immagine "deformata" e quindi con prospettiva frontale. Per utilizzare una *Perspective Transform*, tuttavia, è necessario conoscere le coordinate dei punti **prima e dopo la trasformazione** e, siccome vogliamo deformare l'immagine in modo che i vertici del foglio fotografato si trovino agli angoli dell'immagine, **è necessario ottenere le coordinate dei vertici del foglio all'interno della fotografia, che non sono note** (i punti di destinazione sono semplicemente gli angoli dell'immagine, quindi ottenere le loro coordinate risulta banale).

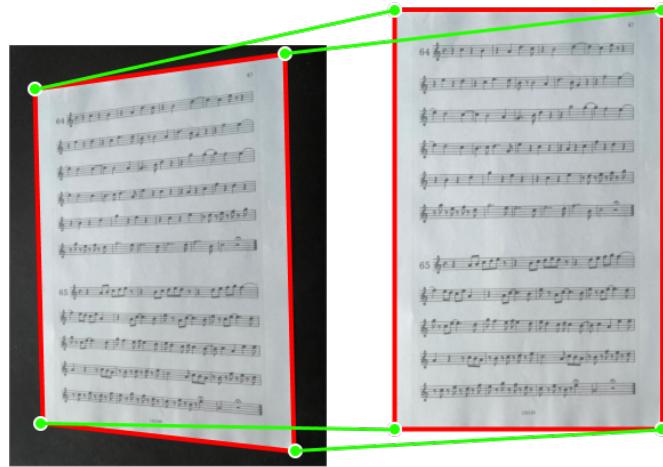


Figura 6: Trasformazione prospettica sulla fotografia dello spartito

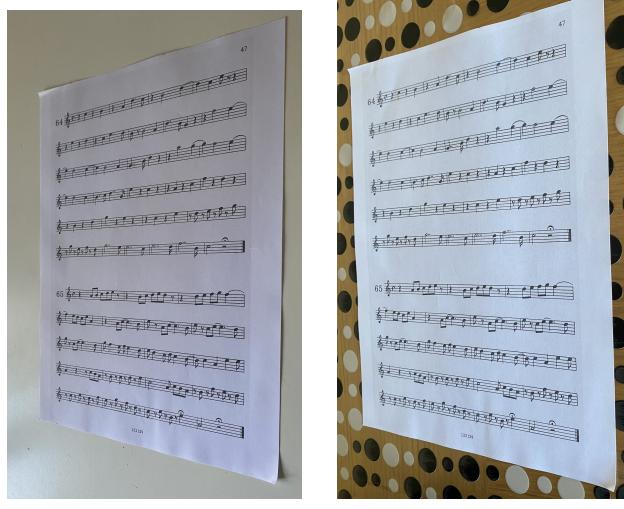
2.4.2 Rilevazione dei quadrilateri

Per ottenere le coordinate dei vertici del foglio nell'immagine fotografata, è necessario un metodo per **rilevare quadrilateri** nell'immagine. In Figura 6 si può notare un esempio di fotografia dov'è presente il foglio con il relativo quadrilatero. In questo particolare caso, può apparire semplice "segmentare" i contorni del foglio e quindi approssimarli ad un quadrilatero. È necessario specificare, tuttavia, che potrebbero esistere casi in cui la rilevazione non è altrettanto semplice, a causa per esempio di sfondi chiari o addirittura bianchi come il foglio, oppure in cui è necessario operare un'approssimazione dei contorni maggiore perché il foglio risulta più piegato in certi punti (esempi mostrati in Figura 7).

Diverse soluzioni possono essere adottate per risolvere questo problema (per esempio la già citata Trasformata di Hough per le linee), tuttavia le due studiate ed implementate in questo progetto sono state **l'approssimazione di contorni** e **l'Harris Corner Detection**.

Per ciascuno dei due metodi, viene effettuato un controllo sulla "qualità" del quadrilatero trovato, controllando l'**area** (escludendo quadrilateri troppo piccoli) e gli **angoli interni** (escludendo quadrilateri con angoli interni inappropriati). Se il quadrilatero così verificato non rispetta i parametri, viene scartato e la procedura fallisce (oppure sarebbe possibile in quel caso ritornare l'intera immagine non deformata).

L'approssimazione dei contorni (funzione `approx_contours` dentro il file `contours.py`) utilizza prima di tutto una *Canny Edge Detection* per ottenere un'immagine contenente gli *edge*, successivamente applica la funzione `findContours` di OpenCV per ottenerne i contorni. Una volta eseguita questa fase, viene applicata un'approssimazione poligonale dei contorni grazie alla funzione `approxPolyDP`, che ritorna i vertici della *polyline* così ottenuta (che specifichiamo essere chiusa, in quanto deve rappresentare un



(a) Sfondo chiaro

(b) Sfondo complesso

Figura 7: Esempi dove rivelare il quadrilatero risulta più complesso

quadrilatero). Il risultato ottenuto viene controllato per incontrare i requisiti spiegati precedentemente (in particolare, il numero dei vertici dev'essere di quattro, altrimenti la forma poligonale ottenuta non è un quadrilatero). Infine, la procedura viene ripetuta con diversi parametri finché non otteniamo un quadrilatero valido oppure abbiamo raggiunto un massimo numero di iterazioni. Il codice Python seguente riassume quanto descritto (versione semplificata del codice del progetto).

```

1 def approx_contours(rescaled_image):
2     image_h, image_w, _ = rescaled_image.shape
3     # ...
4
5     # Try different threshes
6     while not found and it < max_it:
7         edged = get_edged(rescaled_image, thresh[it])
8         approximated = []
9
10        (contours, hierarchy) = cv2.findContours(edged, cv2.RETR_EXTERNAL,
11                                         cv2.CHAIN_APPROX_SIMPLE)
12        contours = sorted(contours, key=cv2.contourArea, reverse=True)[:5]
13        for c in contours:
14            # approximate the contour
15            peri = cv2.arcLength(c, True)
16            for eps in EPS_SEARCH_SPACE:
17                approx_quad = cv2.approxPolyDP(c, eps * peri, True)
18                if is_valid_quadrilateral(approx_quad):
19                    found = True
20                    approximated.append(approx_quad)
21                    break
22                if found:
23                    break
24            final_contour, found = get_max_area_or_whole(approximated, image_w, image_h)
25            it += 1
26
return final_contour, found

```

Codice 1: La funzione `approx_contours`, semplificata ai fini di una più facile visualizzazione

Come si può notare a riga 25, la funzione `get_max_area_or_whole` viene applicata alla lista di contorni validi trovati, in modo da ottenere il contorno **con area maggiore**. Se non ci sono contorni validi, viene ritornata l'intera immagine come quadrilatero.

Il precedente metodo rileva in maniera piuttosto precisa i quadrilateri in varie situazioni, ma purtroppo **non risulta essere abbastanza flessibile rispetto alle condizioni della foto**. Casi come quelli mostrati in Figura 7, infatti, non vengono rilevati. Per questo motivo è stato implementato un secondo metodo, più flessibile ma purtroppo anche meno preciso, basato sulla *Harris Corner Detection*. Quest'ultimo viene applicato come secondo tentativo se l'approssimazione dei contorni fallisce.

Il metodo basato sulla *Harris Corner Detection* (funzione `harris` dentro il file `contours.py`) applica a sua volta una funzione `harris_quad` sull'immagine, che ritorna un quadrilatero (lista dei vertici). La funzione `harris_quad`, prima di tutto, converte l'immagine in grayscale e successivamente applica la funzione di OpenCV `cornerHarris` per trovare i *corner* dell'immagine. Una volta fatto questo, il risultato della funzione viene convertito in una maschera applicando una soglia che dipende dal valore massimo ritrovato nei corner. La maschera conterrà quindi solo dei pixel con valore a `True` nelle posizioni in cui è abbastanza certo che sia presente un *corner*. Tra tutti i *corner* trovati in questo modo, si filtrano le quattro posizioni **più vicine ai quattro angoli dell'immagine**, e queste risultano essere le coordinate dei vertici del quadrilatero. Il codice Python seguente riassume quanto descritto.

```

1 def harris_quad(image):
2     img = image.copy()
3     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
4
5     gray = np.float32(gray)
6     dst = cv2.cornerHarris(gray, 2, 3, 0.04)
7     dst = cv2.dilate(dst, None)
8     mask = dst > 0.01 * dst.max()
9
10    h, w = gray.shape
11    true_indices = np.argwhere(mask)
12    # if we found less than 4 corners, we can't form a quadrilateral
13    if len(true_indices) < 4:
14        return [[h - 1, w - 1], [h - 1, 0], [0, 0], [0, w - 1]]
15
16    top_left = true_indices[np.argmin([math.dist(p, (0, 0)) for p in true_indices])]
17    top_right = true_indices[np.argmin([math.dist(p, (0, w - 1)) for p in true_indices])]
18    bottom_right = true_indices[np.argmin([math.dist(p, (h - 1, w - 1)) for p in
19                                           true_indices])]
20    bottom_left = true_indices[np.argmin([math.dist(p, (h - 1, 0)) for p in true_indices])]
21
22    corners = np.array([top_left, top_right, bottom_right, bottom_left])
23    return corners

```

Codice 2: La funzione `harris_quad`, chiamata internamente dalla funzione "pubblica" `harris`

Il risultato di questa funzione viene controllato a sua volta dalla funzione principale `harris`, che controlla, come per il metodo di approssimazione dei contorni, che il quadrilatero prodotto in questo modo sia valido (chiamando la funzione `is_valid_quadrilateral`).

Operando infine la trasformazione prospettica precedentemente descritta utilizzando come punti "sorgente" quelli trovati durante questa fase, si ottiene l'immagine con prospettiva frontale vista in Figura 6. È quindi possibile passare alla prossima fase.

2.4.3 Pulizia dell'immagine (clean)

In questa fase andremo ad ottenere un'immagine più chiara e con meno difetti, in modo da rendere la fotografia più simile ad una scansione. Per questo scopo, verranno applicate **soglie** e **operazioni morfologiche** più altre tecniche.

Per prima cosa, viene utilizzata una funzione chiamata `warped2sharpened`, che effettua diverse operazioni. Prima di tutto, viene applicata un'**ottimizzazione di luminosità e contrasto**, basandosi sui valori dell'istogramma del livello a scala di grigio. OpenCV mette a disposizione la funzione `convertScaleAbs` che permette di modificare luminosità e contrasto dati i valori di *gain* e *bias* (alfa e beta). Per determinare automaticamente questi due valori viene analizzato l'istogramma dell'immagine [15] e ne viene "tagliata" una parte, basandosi sulla distribuzione cumulativa per capire dove la

frequenza dei valori di grigi è minore di una certa soglia. Dopodiché, una volta determinato il range di grigi interessato, i valori della trasformazione sono determinati come segue:

```

1 alpha = 255 / (maximum_gray - minimum_gray)
2 beta = -minimum_gray * alpha

```

Dopo questo procedimento effettuiamo un *sharpening* dell'immagine utilizzando le funzioni di OpenCV e infine applichiamo una **soglia adattiva** per binarizzare l'immagine⁸.

Come operazione finale per pulire l'immagine, viene eseguito un filtraggio delle componenti connesse dell'immagine. Utilizzando la funzione di OpenCV `connectedComponentsWithStats`, vengono ottenute le componenti connesse e successivamente filtrate in base ad una soglia sulla loro area (ottenibili sul layer `CC_STAT_AREA`). Sono state testate anche operazioni morfologiche per eseguire questa fase di filtraggio, ma le condizioni sono risultate peggiori rispetto a quest'altro approccio.

Qui di seguito è riportato parte del codice che effettua la pulizia dell'immagine.

```

1 def clean_image(image):
2     img = automatic_brightness_and_contrast(image)[0]
3     # Convert the warped image to grayscale
4     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5
6     # sharpen image
7     sharpen = cv2.GaussianBlur(gray, (0, 0), 3)
8     sharpen = cv2.addWeighted(gray, 1.5, sharpen, -0.5, 0)
9
10    thresh = cv2.adaptiveThreshold(sharpen, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
11                                    cv2.THRESH_BINARY, thresh_block_size, thresh_c)
12
13    # ...
14
15    n_labels, labels, stats, _ = cv2.connectedComponentsWithStats(cv2.bitwise_not(thresh),
16                                                                None, None, None, 8, cv2.CV_32S)
17    areas = stats[1:, cv2.CC_STAT_AREA]
18    final_image = np.zeros(labels.shape, np.uint8)
19    component_pixel_thresh = ((h * w) - cv2.countNonZero(thresh)) *
20                            component_pixel_thresh_ratio
21    for i in range(0, n_labels - 1):
22        if areas[i] >= component_pixel_thresh:
23            final_image[labels == i + 1] = 255
24    final_image = cv2.bitwise_not(final_image)
25
26    return final_image

```

Codice 3: La funzione `clean_image`, ultima fase della scannerizzazione

Una volta ottenuta l'immagine con la giusta visione prospettica e migliorata dal punto di vista grafico, è possibile passare alla prossima fase.

In Figura 8 è possibile osservare gli effetti delle varie fasi sull'immagine di input.



Figura 8: Risultati intermedi durante la procedura *img2doc*

⁸Diverse ottimizzazioni sono state operate sui parametri di queste funzioni, specialmente la soglia adattiva. Come verrà spiegato nelle conclusioni, tuttavia, le limitazioni di questi tipi di approcci riguardano il non generalizzare abbastanza seguendo le diverse condizioni dell'immagine

2.5 DOC2SEGMENTS

Una volta ottenuta un'immagine binaria, è necessario estrarre i singoli frammenti da dare in input al modello. Per fare questo prima vengono estratti i *righi musicali* (ogni singolo pentagramma presente nello spartito, nello spartito dell'immagine precedente ne sono presenti 11, per esempio) e successivamente ciascuno di essi viene **suddiviso "a metà"**⁹, ottenendo infine i *segmenti*. Il motivo per cui è necessario questa ulteriore suddivisione è da ricercare nei requisiti del modello: si è notato, infatti, che sottponendo in input l'intero rigo le prestazioni calavano vistosamente, probabilmente a causa della eccessiva larghezza dei segmenti, internamente ridimensionati a tal punto da far risultare i simboli al suo interno incomprensibili.

Tutto il codice che opera la trasformazione del documento (*doc*) in segmenti si trova all'interno del package `doc2segments`. La funzione principale, l'unica pensata come chiamabile dall'esterno, è `segment_doc`, all'interno del file `segmentation.py`. Il blocco di Codice 4 contiene il funzionamento di questa funzione e verrà discusso in dettaglio.

```
1 def segment_doc(doc):
2     segments = [bbox.apply_on_image(doc) for bbox in cluster_bboxes(doc)]
3     return list(reversed(split_segments(segments)))
```

Codice 4: La funzione `segment_doc`: riassume tutta la fase di `doc2segments`

2.5.1 Bounding Box

Per rendere più dichiarative e leggibili alcune operazioni sulle *bounding box* è stata creata una classe apposita `BoundingBox` all'interno del file `bounding_box.py`. Nello specifico, contiene le coordinate del quattro vertici di una bounding box rettangolare e incapsula diverse operazioni, come l'**applicazione su un'immagine** (ottenere solo la porzione d'immagine catturata dalla bounding box), l'**espansione** (l'allargamento in tutte le direzioni), la **conversione da rettangoli nel formato di OpenCV** e la **fusione delle bounding box sovrapposte prese da un lista**.

Nel codice precedente è possibile notare l'applicazione di `apply_on_image`.

2.5.2 Clustering dei righi musicali

L'algoritmo applicato per il clustering sfrutta ancora una volta i **contorni** dell'immagine. Una volta invertita l'immagine (per l'algoritmo dei contorni è necessario farlo perché i simboli rappresentano la parte nera dell'immagine binarizzata, mentre i metodi di ricerca dei contorni rilevano i pixel bianchi) si applica di nuovo la funzione di OpenCV `findContours` e si **filtrano i contorni con un'area minima**, calcolata in base alla quantità di pixel dell'immagine.

Una volta trovati i contorni, viene sfruttata un'altra funzionalità di OpenCV, ovvero `boundingRect`, per trovare le *bounding box* rettangolari contenenti i vari contorni dell'immagine. Siccome potrebbero comparire bounding box **sovraposte**, a seconda di come sono disposti i contorni, viene utilizzata la funzionalità precedentemente implementata nella classe `BoundingBox` per **fondere le bounding box sovrapposte in una sola** (funzione `merge_overlapping_boxes`).



Figura 9: Risultati intermedi durante la procedura `doc2segments`

Siccome i frammenti così trovati hanno agli estremi dei pixel neri, che sono quindi troppo vicini ai bordi dell'immagine, viene utilizzata la funzione `expand` delle *bounding box* per aggiungere un bordo

⁹L'algoritmo che opera la suddivisione verrà spiegato più avanti: in realtà, il "taglio" effettuato di un rigo musicale segue un altro tipo di regola.

bianco chiamato nel codice "safety border", che rende i frammenti così più simile agli spartiti che il modello si aspetta in ingresso. In Figura 9 vengono mostrati i risultati delle varie fasi. Il blocco di Codice 5 seguente riassume il processo.

```

1 def cluster_bboxes(image):
2     # Function to filter out contours with low area
3     def filter_contours(cntr, min_area, min_height, max_area, max_height):
4         return [contour for contour in cntr if
5             min_area <= cv2.contourArea(contour) <= max_area and
6             min_height <= cv2.boundingRect(contour)[3] <= max_height]
7
8     inverted_image = cv2.bitwise_not(image)
9
10    h, w = inverted_image.shape
11
12    contours, _ = cv2.findContours(inverted_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
13
14    valid_contours = filter_contours(contours, min_cluster_area_ratio * h * w,
15        min_cluster_height_ratio * h, max_cluster_area_ratio * h * w,
16        max_cluster_height_ratio * h)
17
18    bounding_boxes = [cv2.boundingRect(contour) for contour in valid_contours]
19
20    bounding_boxes = BoundingBox.merge_overlapping_boxes(bounding_boxes)
21    bounding_boxes = [bbox.expand(safety_margin_ratio*h, safety_margin_ratio*h) for bbox in
22        bounding_boxes]
23
24    return bounding_boxes

```

Codice 5: La funzione `cluster_bboxes`: alcuni parametri sono stati rimossi per maggiore semplicità grafica

2.5.3 Split segments

Come mostrato nel Codice 4, prima di ritornare i segmenti viene applicata una seconda funzione su di essi: `split_segments`. Questa funzione suddivide ciascun segmento in due parti, effettuando un "taglio" circa al centro. Come già spiegato, infatti, è necessario operare questa ulteriore divisione per utilizzare il modello. Di seguito riassunto il codice di questa funzione.

```

1 def split_segments(segments):
2     split = []
3     for segment in segments:
4         shape = segment.shape
5         # Controlli sulla validita' del segmento
6         # ...
7         seg1, seg2 = optimal_half_split(segment)
8         if is_binary(segment):
9             seg1, seg2 = (
10                 small_blur(add_safety_border(smallest_boxed(seg1))),
11                 small_blur(add_safety_border(smallest_boxed(seg2))))
12
13             split.append(seg2)
14             split.append(seg1)
15     return split

```

Codice 6: La funzione `split_segments`

Se dividessimo il segmento semplicemente nella sua metà, tuttavia, rischieremmo di separare parti di alcuni simboli che si trovano in quel punto. Per questo è stata implementata la funzione `optimal_half_split` che, dato un segmento, trova la coordinata del "taglio" ottimale e opera la separazione nelle due parti. Questa coordinata viene trovata col seguente criterio: la coordinata x, più

vicina possibile al centro, e in modo che il **numero di pixel neri nell'immagine, lungo quella coordinata e nei pixel vicini sia inferiore ad una certa soglia**. In maniera più formale: l'algoritmo effettua una ricerca partendo dal centro dell'immagine controllando, per ogni coordinata x, che la sezione di pixel dell'immagine intorno a quella coordinata contenga un numero di pixel neri minore ad una certa soglia. Se viene trovata una coordinata x che rispetta questo criterio, viene utilizzata per dividere il segmento. Altrimenti il segmento viene scartato.

La funzione `optimal_half_split` è mostrata nel blocco di Codice 7.

```

1 def optimal_half_split(segment, threshold=0.1):
2     pixel_threshold = threshold * (segment.shape[0] * 10)
3     bin_segment = cv2.threshold(segment, 127, 255, cv2.THRESH_OTSU)[1]
4     bin_segment = cv2.bitwise_not(bin_segment)
5     width = segment.shape[1]
6     center_x = width // 2
7     optimal_x = -1
8
9     def is_optimal(x):
10         if x >= width or x < 0:
11             return False
12         num_pixels_left = cv2.countNonZero(bin_segment[:, x - 5:x + 5])
13         return num_pixels_left < pixel_threshold
14
15     for offset in range(center_x + 1):
16         left = center_x - offset
17         if is_optimal(left):
18             optimal_x = left
19             break
20         right = center_x + offset
21         if is_optimal(right):
22             optimal_x = right
23             break
24
25     left_part = segment[:, :optimal_x]
26     right_part = segment[:, optimal_x:]
27     return left_part, right_part

```

Codice 7: La funzione `optimal_half_split` che, dato un segmento, trova il punto ideale per suddividere il segmento

Una volta suddivisi i segmenti in due parti, questi vengono riportati in una lista ordinati in base all'apparizione nello spartito.

2.6 Utilizzo del modello

Svolte le precedenti fasi i segmenti sono ora pronti per essere analizzati dal modello. La classe `CTC` (nome dato per via dell'utilizzo dell'omonimo algoritmo per il training) è una versione modificata di quella originariamente sviluppata da Jorge Calvo-Zaragoza. Per crearla è sufficiente fornire il percorso del modello nel filesystem e successivamente è possibile chiamare il metodo `predict` su ciascun segmento. Quest'ultimo metodo ritorna un oggetto di un'altra classe, `EncodedSheet`, classe creata appositamente per contenere le *prediction* dell'algoritmo e renderle fruibili in maniera più comoda e flessibile, con molti metodi utili al suo interno, utilizzati anche per calcolare le metriche associate ai risultati.

2.7 Risultati

All'interno del file `metrics.py` si trova tutto il codice utilizzato per calcolare le metriche al fine di valutare i risultati ottenuti. Per valutare questi risultati, è stato creato un piccolo *test set* di fotografie di spartiti, catturate dalla fotocamera di un smartphone. Nello specifico: per 8 spartiti diversi sono state ottenute foto da **angolazioni diverse** (frontale, inclinata lateralmente, inclinata in basso e in alto) e, per ciascuna di esse, la foto è stata scattata in **quattro condizioni di luce differenti** (denominate *normal*, *shadow*, *dark* e *light*). E' stato inoltre cambiato lo **sfondo** (superficie chiare,

scure o con motivi) e anche la **qualità** dell'immagine. In totale, la dimensione è di circa 200 foto. Di seguito in Figura 10 è possibile notare alcuni esempi presi dal dataset (ciascuno in una delle diverse condizioni di luce).

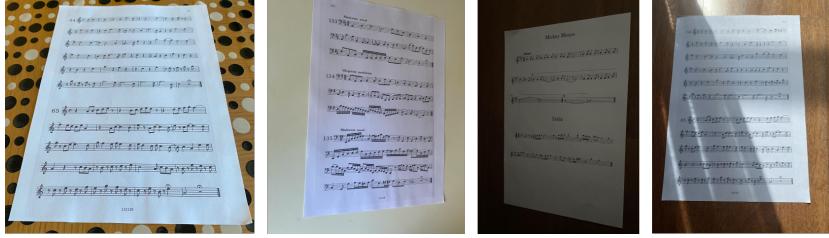


Figura 10: Campione di immagini prese dal *test set*

Le metriche utilizzate sono state l'**Indice di Jaccard** e il **Symbol Error Rate** (SER)¹⁰. Entrambe sono state calcolate sia rispetto all'output atteso (il riferimento) sia **rispetto all'output prodotto sull'immagine scannerizzata**: questo è stato fatto per evidenziare se effettivamente le prestazioni su fotografie sono comparabili a quelle ottenute sulle scannerizzazioni, che sono infatti previste dal modello in origine.

2.7.1 Analisi delle prestazioni

Di seguito in Tabella 1 vengono quindi riportate le prestazioni ottenute sul *test set* calcolate utilizzando le metriche descritte.

Prestazioni ottenute				
Condizioni di luce	Jaccard (ref)	Jaccard (scan)	SER (ref)	SER (scan)
normal	50.3%	61.4%	20%	19.5%
shadow	48.4%	44.6%	36.8%	38.1%
dark	32.3%	32.9%	45.2%	44.1%
light	11.9%	12.2%	78.4%	81.7%

Tabella 1: Prestazioni ottenute sul *test set*

Come si può notare, il metodo ha prodotto risultati in generale piuttosto insoddisfacenti, in alcuni casi addirittura molto scarsi, per esempio nelle foto in condizione *light*. Non solo: questo metodo sembra non portare a risultati accettabili **neanche se comparato alle immagini scannerizzate**, facendo effettivamente pensare che **non è il modello a produrre risultati pessimi, ma piuttosto tutta la fase di pre-processing**. Quali sono le cause? Nei prossimi paragrafi verranno identificate alcune delle varie problematiche responsabili delle prestazioni mostrate.

2.7.2 Modello poco performante

A fronte dell'analisi sulle performance, è stato notato come **il modello performasse scarsamente anche su alcune immagini scannerizzate**. Analizzando, infatti, i simboli predetti in quei casi, molto spesso gli errori commessi dal modello differivano completamente dagli errori commessi invece sulle fotografie con il pre-processing. In altre parole gli errori commessi dal modello sulle scannerizzazioni (tipo di input per cui è stato pensato in origine) **non erano in numero minore, erano semplicemente errori diversi**. Addirittura è stato osservato come in certi casi il modello applicato sulle immagini pre-processate **abbia ottenuto addirittura risultati migliori del modello sull'immagine scannerizzata**, segno che il modello è particolarmente sensibili alle condizioni grafiche dell'immagine.

¹⁰Metrica utilizzata anche nell'articolo originale [3]. Si calcola come il "numero medio di operazioni elementari (aggiunta, modifica, eliminazione) necessarie per produrre la sequenza di riferimento a partire da quella predetta", in rapporto alla lunghezza della sequenza. Un valore più alto equivale a prestazione peggiore

2.7.3 Simboli non riconoscibili o malformati

Alcuni dei simboli presenti nello spartito **non erano presenti nel "vocabolario" del modello**. Siccome il modello funziona ritornando un lista di indici riferiti ad un vocabolario di simboli, è stato possibile notare come questi simboli non fossero proprio presenti e quindi impossibili da rilevare. Alcuni, invece, sono risultati irriconoscibili dal modello a causa di imperfezioni nella stampa. Purtroppo la loro presenza è stata notata già a sviluppo avviato, ma comunque il loro numero è risultato esiguo.



Figura 11: Esempi di simboli non riconosciuti dal modello

2.7.4 Confusione degli spazi

Uno degli errori compiuti con maggior frequenza è risultato essere la **sbagliata collocazione verticale dei simboli**. Il modello ha spesso confuso gli spazi tra ogni riga del pentagramma, portando a valutare in modo erroneo l'**altezza della nota**. Questo potrebbe essere dovuto all'aggiunta di un eccessivo bordo bianco ai singoli segmenti, che quindi portano il modello a valutare peggio la dimensione verticale dell'immagine. Un modo per valutare meglio questo tipo di errore sarebbe stato l'impiego di metriche che tenessero conto della **distanza tra le note predette e quelle di riferimento**, invece di contare semplicemente ogni nota diversa come errore.

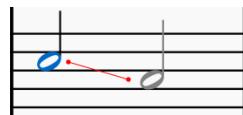


Figura 12: Esempio di confusione degli spazi del pentagramma. Il modello valuta la prima nota assegnandogli erroneamente l'altezza della seconda

2.7.5 *Bias* del modello

Il modo in cui è stato addestrato questo modello, purtroppo, rende difficile un'integrazione col metodo utilizzato nella parte di *doc2segments*, ovvero la **divisione dei segmenti in 2 parti**. Questa incompatibilità è dovuta al fatto che il modello è stato addestrato con segmenti di immagini di spartiti che **cominciano sempre all'inizio di un nuovo rigo musicale**. Questo porta il modello a predire certi simboli che **tipicamente sono presenti all'inizio di un rigo anche se effettivamente non sono presenti nel segmento analizzato**. Questo problema potrebbe essere risolto non dividendo i segmenti i due sotto-parti, ma purtroppo i segmenti risultano allora troppo "larghi", e in quel modo il *rescaling* interno causa un peggioramento ancora maggiore delle prestazioni. Questo rende il modello utilizzato e la soluzione trovata per i segmenti di fatto due approcci **incompatibili**.

Non solo: essendo che all'inizio di un rigo sono presenti i simboli che indicano la tonalità, cambiando conseguentemente la semantica di una quantità anche grande dei simboli presenti subito dopo, il metodo della divisione dei segmenti risulta ancora meno applicabile **perché l'informazione sulla tonalità non viene propagata alla seconda sotto-partita del segmento**, causando di fatto un potenziale errore medio molto alto.

In Figura 13 e 14 è possibile notare le discrepanze tra i simboli effettivi e predetti a causa delle problematiche appena descritte.

2.7.6 Soglie troppo intense sui simboli

Le soglie applicate riescono a binarizzare sufficientemente bene l'immagine ma, purtroppo, in certe particolari condizioni di luce e orientazioni dello spartito alcuni problemi potrebbero emergere. Uno di questi è il caso in cui **le soglie rovinino la qualità dei simboli musicali**, facendoli interpretare



Figura 13: Spartito effettivo

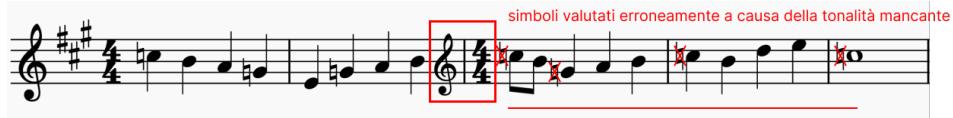


Figura 14: Spartito predetto: notare l'aggiunta di simboli non presenti nello spartito effettivo

erroneamente dal modello. Il caso più frequente è quello in cui la "testa" della nota viene leggermente sbiadita, risultando più vuota all'interno dell'ovale di quanto era in origine. Questo può non essere un problema se l'effetto ha intensità lieve, ma più questa aumenta e più il modello tende a sbilanciarsi verso un altro tipo di simbolo dove effettivamente la testa è vuota (come per esempio le note da 2 quarti o 4 quarti).

In Figura 15 viene mostrato meglio questo concetto.

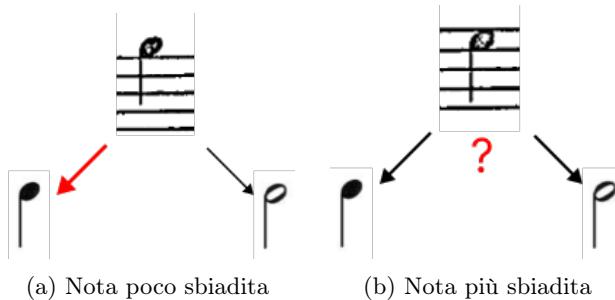


Figura 15: Esempio di casi in cui il modello si trova a distinguere i simboli sbiaditi

2.7.7 Inclinazione dello spartito

A causa della deformazione prospettica alcuni spartiti potrebbero essere stati modificati in modo da non aver le linee del pentagramma **perfettamente orizzontali**, bensì leggermente inclinate. Siccome, invece, il modello non è stato allenato su spartiti inclinati, questo potrebbe aver influito nel suo "giudizio". Nonostante questo, sono stati ritrovati ben pochi errori riconducibili a questa condizione.

2.7.8 Pre-processing non generalizzante

Per concludere la parte di Augmented End-To-End vengono ora spiegate le criticità di questa parte di progetto. Il principale difetto degli approcci proposti è stato l'**incapacità di generalizzare**: molte ottimizzazioni sono state applicate per permettere agli algoritmi trovati di funzionare in diverse condizioni di luce, orientazione, condizioni del foglio ecc. eppure il metodo perdeva sempre in prestazioni non appena ci si spostava da un caso ad un altro, facendo domandare, effettivamente, se fosse meglio concentrarsi piuttosto su un caso in particolare (per esempio, fotografie tutte raccolte in un'unica condizione di luce) ottimizzando gli algoritmi per quello.

Nonostante questo, è opportuno rimarcare che le prestazioni del modello sono comunque risultate insufficienti, concludendo che l'approccio end-to-end è ancora immaturo e necessita di molti ulteriori accorgimenti. Potenziali sviluppi futuri potrebbero includere tecniche di intelligenza artificiale più all'avanguardia rispetto alle reti ricorrenti.

3 Staff Line removal

In questa parte del progetto verrà implementata la parte di *Staff line removal* della pipeline OMR. Prima di procedere con l'implementazione vera e propria, verranno valutati velocemente diversi possibili approcci per poi arrivare alla preparazione del dataset e all'addestramento di una rete neurale.

3.1 Possibili approcci

Prima dell'avvento di tecniche basate sul Deep Learning, la maggior parte delle tecniche adottate per lo Staff line removal si basava su euristiche e algoritmi tradizionali di machine learning. Nella competizione ICDAR 2013 [16] (e prima ancora ICDAR 2011) sono stati valutati tanti diversi approcci proprio per eseguire la Staff line removal di immagini di spartiti anche manoscritti, con diversi livelli di rumore, deformazione e altre distorsioni dell'immagine che hanno evidenziato la complessa natura del problema.

Diverse soluzioni esistono per questo problema, alcune ormai additiate come poco performanti e altre che rappresentano lo stato dell'arte. Innanzitutto, abbiamo l'utilizzo di tecniche di elaborazione digitale delle immagini, come la già citata Trasformata di Hough, per individuare e rimuovere le linee. Questo metodo si basa sull'identificazione delle linee rette nelle immagini e sulla loro successiva eliminazione. Come già detto, tuttavia, è stato riscontrato che questa tecnica potrebbe compromettere la qualità della notazione musicale, in particolare nelle partiture più complesse o con righi musicali parzialmente visibili. Successivamente, è stato esplorato l'impiego di algoritmi di apprendimento automatico, come le reti neurali convoluzionali (CNN) [5], per rilevare e rimuovere i righi musicali. Questo approccio si è dimostrato più promettente, poiché le CNN sono in grado di imparare autonomamente i pattern associati alle *staff line* e di adattarsi a variazioni nelle partiture musicali. Infine, è stata valutata l'efficacia di tecniche ibride che combinano metodi di elaborazione digitale delle immagini con algoritmi di machine learning, coinvolgendo tecniche di feature extraction con potenzialmente anche reti neurali. Questo approccio ha mostrato risultati promettenti, consentendo una maggiore flessibilità nell'adattarsi a diversi tipi di partiture musicali e condizioni di scansione. Recentemente sono state impiegate persino reti come le GAN [6] per questo tipo di problema.

Per questo progetto è stato scelto un approccio ibrido che utilizza una rete neurale per effettuare la *semantic segmentation* dei righi musicali seguita da un insieme di banali tecniche di elaborazione delle immagini per rimuovere i righi dall'immagine a partire da una maschera di pixel. Come rete è stata scelta la *UNet* [9], modello valido utilizzato originariamente per la *semantic segmentation* di immagini biomediche.

3.2 Preparazione del dataset

Negli ultimi anni numerosi dataset in ambito OMR sono stati resi pubblicamente disponibili: tra questi, il dataset **Rebelo** e **Deepscores** [17] [18] sono stati analizzati più a fondo. Vista la ricchezza e qualità dei dati presenti, il secondo è stato scelto come materiale per l'addestramento della rete UNet. Nello specifico è stato utilizzato un sotto-insieme della seconda versione di Deepscores, chiamato **Deepscores V2 Dense**, che include diversi file in cui le diverse parti della partitura sono segmentate tramite una gradazione di colore. In Figura 16 è possibile vedere un esempio preso da tale dataset, sia il file che la versione segmentata.

Siccome stiamo cercando di segmentare solo i righi dello spartito, dei file segmentati ci serve in realtà solo una piccola parte. Gli autori di Deepscores mettono a disposizione un tool per navigare il dataset ed estrarre solo certe porzioni della parte segmentata, ma nel nostro caso è stato sufficiente estrarre la parte delle immagini che corrisponde ai righi selezionandone il relativo colore e binarizzare l'immagine in modo da ottenere una **maschera** dei righi musicali.

Estrarre le maschere in questo modo, tuttavia, non è risultato sufficiente. Si tratta, infatti, di immagini molto grandi (in media circa 2000x3000 pixel), il che rende molto dispendioso un addestramento diretto (in termini di risorse computazionali) e per di più le immagini sono di forma rettangolare (poco portatile e soprattutto diverso dai normali casi di training). Non è possibile, inoltre, eseguire un *downsampling* in quanto delle trasformazioni come il *rescaling* potrebbero peggiorare notevolmente la qualità dei già sottili righi nell'immagine della partitura, rendendo le maschere troppo sottili o addirittura vuote. Per questi motivi, l'approccio seguito è quello di lavorare, ancora una volta, con **sottoparti di immagine**, che questa volta chiameremo **chunks**.

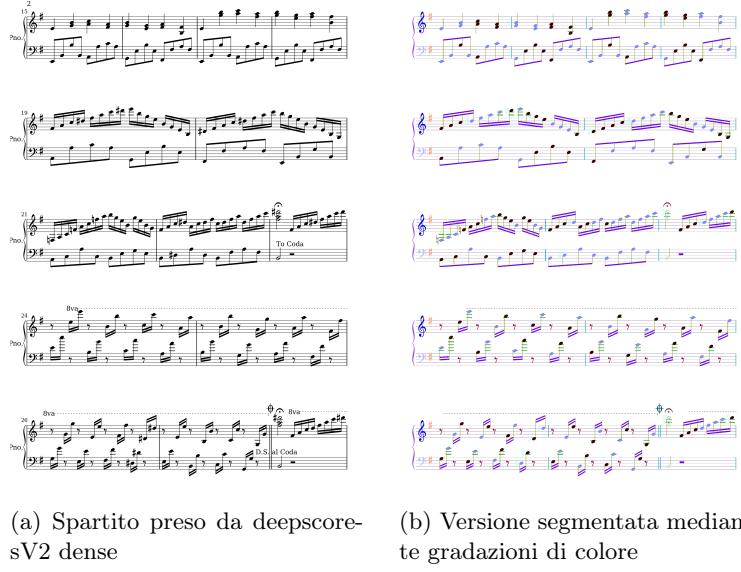


Figura 16: Esempio di file contenuto di deepscoresV2 dense

Seguendo questa filosofia, sono state estratti chunks di dimensione 512x512 pixel dalle immagini del dataset insieme alla relativa maschera ottenuta nel modo precedentemente descritto. In totale, sono state prodotte 640 immagini di training e 200 immagini di test. In Figura 17 è mostrato un esempio preso dal dataset processato, insieme alla relativa maschera. Come si può notare, anche l’immagine dello spartito è stata **invertita** per seguire l’approccio alla segmentazione di UNet (gli oggetti da segmentare sono bianchi su sfondo nero).

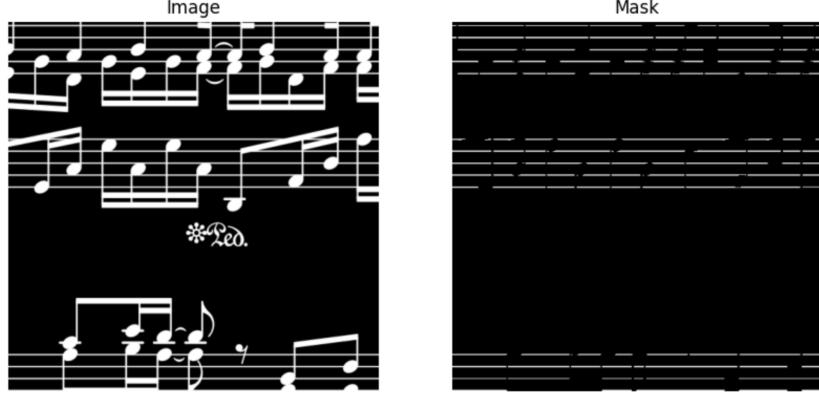


Figura 17: Esempio estratto dalla versione finale del dataset, dato in input per il training

3.3 Training

L’addestramento della rete è stato portato avanti con le librerie **Tensorflow** e **Keras** di Python. Le immagini sono state normalizzate per stabilizzare il processo di training. Utilizzando la funzione `train_test_split` della libreria **Scikit-learn** è stato ottenuto un validation set della dimensione di 64 immagini (un 10% del train set). Come metriche per il training sono state utilizzate l’**Accuratezza** e l’**Indice di Jaccard**. All’interno della UNet sono stati utilizzati un numero di filtri per i blocchi di *downsampling* e di *upsampling* pari a 64, 128, 256 e 512. Come ultimi iperparametri, è stato utilizzato l’ottimizzatore **Adam**, la *loss function* **Binary Cross-Entropy**, un *batch size* di 50 e infine un numero di epoche di 200 con *Early Stopping* impostato a 10 di *patience*.

L’addestramento è terminato dopo 102 epoche: l’andamento delle metriche utilizzate è mostrato in Figura 18.

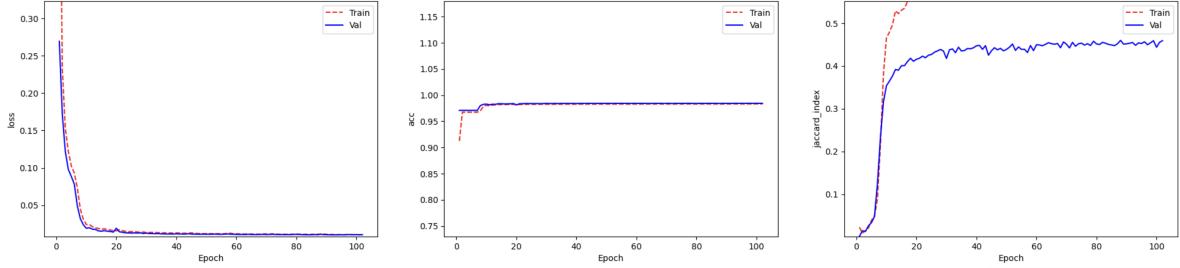


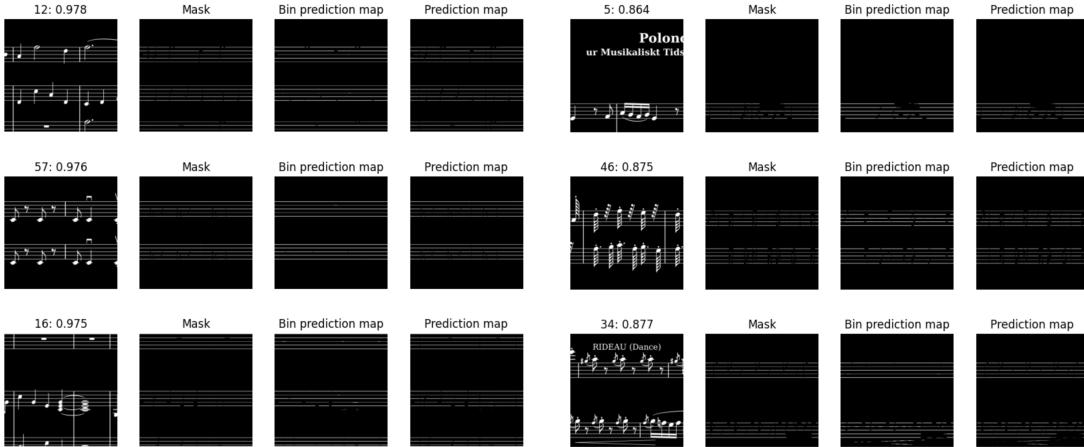
Figura 18: Andamento delle metriche durante le 102 epoche di addestramento

Una volta ottenuto il modello migliore, in realtà l'output del modello è una prediction map di dimensione 512x512 dove ogni pixel riporta la probabilità che quel pixel appartenga ad un rigo musicale. Per confrontare, quindi, questi output con il ground truth viene applicata una **soglia pari a 0.6**, calcolata osservando la variazione delle prestazioni a seconda della soglia selezionata.

3.4 Risultati

L'output finale è quindi una maschera binaria di pixel che contiene solo quelli appartenenti ai righi del pentagramma. Possiamo ora confrontare i risultati sul test set con il ground truth.

Il modello ottiene sul test set un'**Accuratezza media del 99.8%** e un **Indice di Jaccard medio pari a 93.29%**. Di seguito sono mostrati alcuni dei risultati migliori e peggiori.



(a) Risultati migliori sul Test Set

(b) Risultati peggiori sul Test Set

Figura 19: Risultati sul test set indicati con il valore di accuratezza

3.4.1 Applicazione e Post-Processing

Una volta aver ottenuto la maschera dei righi musicali, possiamo applicare questi risultati per rimuovere effettivamente le *staff lines* dall'immagine. Prima di tutto, per utilizzare questo modello sulle immagini di spartiti, anche quest'ultimi vanno suddivisi in *chunks* di dimensione 512x512: per far sì l'immagine sia completamente coperta da questi, viene applicato un bordo sul lato inferiore e destro dell'immagine in modo tale che la dimensione dell'immagine sia multiplo della dimensione dei *chunks*.

Una volta fatto questo, creiamo due funzioni che permettono da un'immagine di ottenere la lista dei suoi *chunks* e da una lista di *chunks* di riottenere l'immagine di partenza. Queste due funzioni sono `divide_in_chunks` e `reassemble_image` mostrate nel blocco di Codice 8. La scomposizione in *chunks* è invece visibile in Figura 20.

Una volta ottenuti i *chunks*, possiamo applicare il modello in sequenza su ciascuno di essi per ottenere la relativa maschera di pixel. Prima di applicare la maschera per rimuovere i righi del pentagramma, viene effettuata un'**operazione morfologica** sulla maschera, nello specifico una **dilatazione**

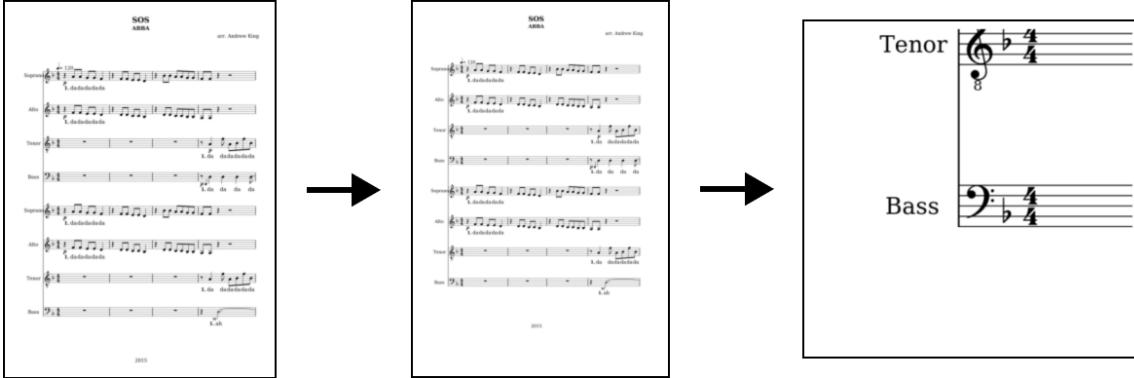


Figura 20: Prima parte di pre-processing: aggiunta del bordo e scomposizione in *chunks*

verticale: questo è stato fatto perché la maschera dei righi è risultata spesso essere troppo sottile per rimuovere pienamente i righi all'interno dell'immagine.

```

1 def divide_in_chunks(image, chunk_size=512):
2     img_chunks = []
3     for x in range(0, image.shape[1], chunk_size):
4         for y in range(0, image.shape[0], chunk_size):
5             y2 = min(y + chunk_size, image.shape[0])
6             x2 = min(x + chunk_size, image.shape[1])
7             img_chunk = image[y:y2, x:x2]
8             img_chunks.append(img_chunk)
9
10    return img_chunks
11
12 def reassemble_image(chunks, width, height, chunk_size):
13     rows = (height // chunk_size)
14     if height % chunk_size != 0:
15         rows += 1
16
17     assembled_image = np.empty((height, 0, 3), dtype=np.uint8)
18     i = 0
19     while i < len(chunks):
20         column = chunks[i]
21         j = i + 1
22         while j % rows != 0:
23             column = np.concatenate((column, chunks[j]), axis=0)
24             j += 1
25         assembled_image = np.hstack((assembled_image, column))
26         i += rows
27
28     return assembled_image

```

Codice 8: Le due funzioni utilizzate per lavorare con i *chunks*

Dopo tutte queste operazioni possiamo finalmente applicare la maschera, impostando a 0 i corrispondenti pixel del *chunk* corrispondente. Dopodiché, ritornando la lista dei *chunk* modificati, "ri-assembliamo" l'immagine originale. La maggior parte di queste operazioni è stata incapsulata all'interno della classe **StaffRemover**, il cui codice è stato riassunto in Codice 9.

```

1 class StaffRemover:
2     def __init__(self, model_path):
3         self.model = get_model(model_path)
4
5     # ...

```

```

7  def process_result(self, index):
8      output = np.squeeze(self.pred_bin[index], axis=2)
9      kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 3))
10
11     output_converted = output.astype(np.uint8)
12     dilated = cv2.morphologyEx(output_converted, cv2.MORPH_DILATE, kernel)
13     mask = dilated == 1
14
15     black_color = np.zeros(shape=(512, 512, 3))
16
17     # Apply the mask to retain BGR values where the mask is True
18     result = ((np.where(mask[..., None], black_color, [self.input_images[index]])[0]) *
19                255).astype(np.uint8)
20     _, result = cv2.threshold(result, 50, 255, cv2.THRESH_BINARY)
21     result = cv2.bitwise_not(result)
22     # ...
23
24
25
26     def remove_staffs(self, images, thresh=0.6, safety_border=0):
27         # ...
28         self.input_images, borders_h, borders_v = self.prepare_chunks()
29
30         preds = self.model.predict(self.input_images)
31         optimal_bin_thr = thresh
32         self.pred_bin = (preds >= optimal_bin_thr).astype(np.int64)
33
34         results = []
35         for i in range(self.pred_bin.shape[0]):
36             result = self.process_result(i)
37             h, w, _ = result.shape
38             results.append(result[max(borders_v[i]-1, 0):min(h-borders_v[i]+1, h),
39                                 max(borders_h[i]-1, 0):min(w-borders_h[i]+1, w)])
40
41         # ...
42         return results

```

Codice 9: Parte del codice di `StaffRemover`, la classe Python che incapsula quanto descritto per l'applicazione del modello

Chiamando, quindi, il metodo `remove_staffs` passando come input la lista dei *chunk* otteniamo infine la lista di *chunk* modificati, che compongono l'immagine finale.

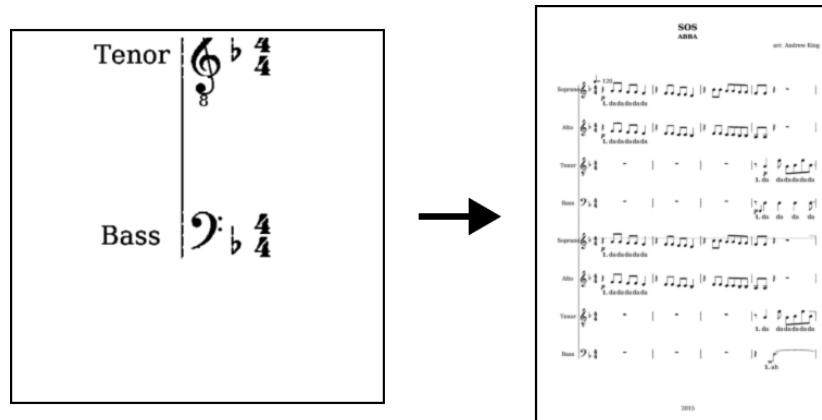


Figura 21: Dai *chunks* modificati riotteniamo l'immagine intera senza righi musicali

4 Conclusione

In questo progetto sono stati studiati diversi approcci al mondo dell'OMR, tra cui il più recente approccio End-To-End e una parte del più classico approccio pipeline. Questo studio ha dato la possibilità di esplorare approcci ibridi, che fondono l'utilizzo di tecniche tradizionali per l'elaborazione di immagini a più potenti reti neurali che permettono di generalizzare parte del problema. Essendo che il progetto in questione è piuttosto lontano dallo stato dell'arte dell'OMR, quanto implementato e studiato in questo contesto potrebbe essere riutilizzato in futuro per ampliare il lavoro svolto portandolo ad un livello ancora più elevato: uno di questi sviluppi futuri potrebbe essere, per esempio, l'addestramento di una propria rete neurale per il riconoscimento End-To-End, in modo da aver più controllo sul processo. Ciononostante, rimangono ancora molti dubbi sullo sviluppo di sistemi OMR: la disponibilità di dataset è si aumentata negli ultimi anni, ma in verità solo molto pochi di questi dispongono di materiale sufficientemente di qualità. Anche i sistemi che rappresentano lo stato dell'arte, inoltre, sono ben lontani dall'ottenere prestazioni ottime sul vasto mondo di spartiti pubblicamente disponibili, viste le problematiche spiegate e affrontate in questa sede. Utilizzare l'approccio a pipeline è sembrato più consone e robusto rispetto a quello End-To-End, che invece di generalizzare sembra talvolta introdurre variabili errate e bias: questo potrebbe dipendere ovviamente dal modello utilizzato ma sicuramente rappresenta in parte una proprietà del problema dell'OMR stesso e della notazione musicale in generale, ovvero il fatto di presentare ambiguità e interpretazioni lasciate all'esecutore/interprete dello spartito. Allo stesso tempo, però, lo sviluppo di questo parziale progetto ha dato modo anche di notare evidenti limitazioni negli approcci tradizionali per l'elaborazione delle immagini, che spesso si nascondono dietro l'utilizzo di parametri opachi, ottimizzati solo per un sotto-insieme dei casi effettivamente affrontabili.

L'OMR è senza dubbio un problema complesso, ma che presto o tardi raggiungerà la piena maturità, permettendo a migliaia di partiture che rischiano di venir perse, soprattutto manoscritte, di poter essere codificate e ripristinate in un formato nuovo e disponibile a tutti.

Riferimenti bibliografici

- [1] P. Bellini, I. Bruno, P. Nesi, "An Off-line Optical Music Sheet Recognition", 2004
- [2] Florence Rossant, Isabelle Bloch, "Robust and Adaptive OMR System Including Fuzzy Modeling, Fusion of Musical Rules, and Possible Error Detection", 2006
- [3] Jorge Calvo-Zaragoza, David Rizo, "End-to-End Neural Optical Music Recognition of Monophonic Scores", 2018
- [4] A. Rebelo, G. Capela, Jaime S. Cardoso, "Optical recognition of music symbols - A comparative study", 2009
- [5] Jorge Calvo-Zaragoza, Antonio Pertusa, Jose Oncina, "Staff-line detection and removal using a Convolutional Neural Network", 2017
- [6] Aishik Konwer, Ayan Kumar Bhunia, Abir Bhowmick, Ankan Kumar Bhunia, Prithaj Banerjee, Partha Pratim Roy, Umapada Pal, "Staff line Removal using Generative Adversarial Networks", 2018
- [7] Jorge Calvo-Zaragoza, Isabel Barbancho, Lorenzo J. Tardon, Ana M Barbancho, "Avoiding staff removal stage in Optical Music Recognition - Application to scores written in White Mensural notation", 2015
- [8] Quang Nhat Vo, Tam Nguyen, Soo Hyung Kim, Hyung-Jeong Yang, Guee-Sang Lee, "Distorted Music Score Recognition without Staffline Removal", 2014
- [9] Olaf Ronneberger, Philipp Fischer, Thomas Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation", 2015
- [10] Alex Graves, Santiago Fernández, Faustino Gomez, Jürgen Schmidhuber, "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks", 2006
- [11] Alexander Pacha, Jan Hajic Jr., Jorge Calvo-Zaragoza "A Baseline for General Music Object Detection with Deep Learning", 2018
- [12] Simonyan, K., Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2014
- [13] PriMuS dataset webpage <https://grfia.dlsi.ua.es/primus/>
- [14] Andrew Campbell OpenCV-Document-Scanner <https://github.com/andrewdcampbell/OpenCV-Document-Scanner>
- [15] Automatic brightness and contrast adjustment with OpenCV <https://shorturl.at/jknMR>
- [16] Muriel Visani, V.C Kieu, Alicia Fornés, Nicholas Journet, The ICDAR 2013 Music Scores Competition: Staff Removal, 2013
- [17] DeepScores dataset webpage <https://zenodo.org/records/4012193>
- [18] Lukas Tuggener, Yvan Putra Satyawan, Alexander Pacha, Jürgen Schmidhuber, Thilo Stadelmann, "The DeepScoresV2 dataset and benchmark for music object detection", 2021