

INDICE

<i>Ringraziamenti</i>	<i>1</i>
<i>Introduzione</i>	<i>1</i>
Motivazioni	1
Descrizione del documento	2
<i>Capitolo 1</i>	<i>4</i>
Nozioni di Base	<i>4</i>
1.1 Spartiti musicali	4
1.2 Intelligenza Artificiale	5
1.3 Reti neurali	5
1.4 Deep Learning	6
1.5 Computer Vision & Object Detection.....	7
<i>Capitolo 2</i>	<i>9</i>
Tecnologie e Strumenti	<i>9</i>
2.1 YOLO.....	9
2.2 Swift	12
2.3 Xcode	13
2.4 CoreML	13
2.4.1 Vision Framework.....	14
2.5 Core Data	14
2.6 Python	15
2.7 Jupyter Notebook	15
2.8 Google Colab	17
<i>Capitolo 3</i>	<i>19</i>
Modello Predittivo	<i>19</i>
3.1 Motivazioni	19
3.2 Creazione del Dataset.....	20
3.3 Setup.....	24
3.4 Addestramento	25
3.5 Risultati	27
3.6 Elementi riconosciuti	28
3.7 Conversione a CoreML	29
<i>Capitolo 4</i>	<i>30</i>
User Experience	<i>30</i>
4.1 Introduzione	30
4.2 Progettazione.....	30
4.3 Risultato	33
<i>Capitolo 5</i>	<i>36</i>
Implementazione di PentaKey	<i>36</i>
5.1 Introduzione	36

5.2	Beans	37
5.3	Acquisizione immagini	39
5.4	Model Provider.....	39
5.5	Parser.....	40
5.6	Text Recognition.....	45
5.7	Score Drawer.....	46
5.8	Editor.....	49
5.9	Persistenza dati.....	51
<i>Capitolo 6</i>	54
Lavori correlati	54
6.1	Optical Music Recognition	54
<i>Capitolo 7</i>	57
Conclusioni	57
7.1	Possibili estensioni future	57
<i>Bibliografia</i>	60

Introduzione

Per secoli la musica è stata condivisa e conservata solamente con due metodi: trasmissione uditiva e sotto forma di documenti scritti, normalmente chiamati spartiti musicali.

Molte di queste partiture esistono sotto forma di manoscritti inediti e quindi rischiano di perdersi a causa dei normali danni del tempo.

Inoltre, nel mondo della musica non c'è nessun modo di facilitare gli scrittori musicali nell'atto di modificare spartiti già scritti. Nella situazione attuale, dopo la stampa di un qualsiasi spartito, un artista nel caso in cui voglia apportare una modifica dovrebbe riscrivere completamente tutto lo spartito, ripetendo un lavoro già precedentemente svolto.

Per preservare la musica è necessaria una qualche forma di composizione o, idealmente, un sistema informatico in grado di decodificare automaticamente le immagini simboliche e ricostruire digitalmente le relative partiture.

Impiego dei sistemi informatici mobili, può dare un forte aiuto alla risoluzione di questo problema.

Con il loro avanzamento si è estesa la possibilità di scattare delle foto con un'ottima qualità a tutti. Inoltre, per definizione sono dei dispositivi facilmente trasportabili ed in alcuni casi tascabili, non trascurando una buona potenza di calcolo.

Motivazioni

Prima di cercare una soluzione al problema, ci siamo chiesti se il problema che stavamo per affrontare necessitasse o meno di una soluzione. Per poter rispondere al nostro quesito, abbiamo creato un questionario che verificasse la reale identità del problema ed esponeva la nostra possibile soluzione.

Quindi, chiedendo alle persone all'interno del mondo musicale, abbiamo cercato di capire se ci potesse essere un certo interesse per una soluzione digitale a questo tipo di problemi. Non sorprende che la nostra idea ha ricevuto un buon feedback.

A questo punto il nostro compito è trasformare questa idea in qualcosa di reale e lo stiamo facendo ponendoci alcune domande fondamentali per valutare una possibile soluzione:

- *Quanto può essere fastidioso maneggiare un foglio musicale scritto a mano?*
- *Come potrebbe essere migliorato il modo di gestire un foglio musicale scritto a mano?*

Così fino ad ora abbiamo cercato di formalizzare le risposte a quelle domande in una soluzione in cui dovrebbe essere prevista la possibilità di digitalizzare uno spartito musicale e quindi di permettere agli utenti di modificare il testo in uscita.

Per fare ciò verrà proposta un'applicazione che sarà in grado di acquisire e successivamente analizzare la fotografia di uno spartito musicale.

Verrà impiegato, un modello di intelligenza artificiale che dopo il suo relativo training sarà in grado di riconoscere i simboli che lo compongono.

Inoltre, verrà realizzata una interfaccia utente che possa permettere la corretta visualizzazione di uno spartito musicale e permetta un facile utilizzo delle funzionalità proposte.

Infine, verranno aggiunte diverse funzionalità al fine di fornire all'utente finale la possibilità di modificare, aggiungendo o editando gli elementi dei propri spartiti musicali.

Descrizione del documento

In questo documento verrà proposta e illustrata una applicazione per il mondo iOS che permette di scansionare, visualizzare ed editare gli spartiti.

Verranno utilizzate tecniche di deep learning per la risoluzione del problema della decodificazione dell'immagine, affiancati da approcci imperativi per la loro corretta interpretazione.

La tesi è articolata come segue:

- Nel primo capitolo vengono elencate e spiegate le nozioni base di cui si basa la tesi e il progetto.
- Nel secondo capitolo viene fatta una intera panoramica sulle tecnologie e gli strumenti utilizzati per lo sviluppo dell'intero progetto.

- Nel terzo capitolo vengono analizzate nel dettaglio tutte le fasi per la creazione il modello predittivo. Partendo dalla creazione del dataset fino ad arrivare all’addestramento e alla sua messa in funzione.
- Nel quarto capitolo vengono mostrate le scelte, con le dovute motivazioni, legate all’user experience.
- Nel quinto capitolo vengono illustrate nel dettaglio tutti gli aspetti implementati che hanno caratterizzato la realizzazione del progetto.
- Nel sesto capitolo vengono illustrati gli studi e i lavori correlati al progetto proposto.
- Nel settimo capitolo vengono racchiuse le conclusioni e vengono elencate le possibili estensioni della tesi affrontata.

Capitolo 1

Nozioni di Base

In questa sezione, verranno elencati e spiegati i concetti principali presenti in questa tesi. Parleremo, degli spartiti musicali, di cos'è il Machine Learning e di che differenza c'è tra questo e l'Intelligenza Artificiale, di reti neurali, di cos'è il Deep Learning e degli accenni sulla Computer Vision e del problema dell'Object Detection.

1.1 Spartiti musicali

Uno spartito musicale, è un insieme di righe musicali che sono formati a loro volta da un insieme di parti (in inglese “Measure”) dove al loro interno vengono posizionati i simboli musicali. Inoltre, in un rigo musicale possono essere definiti la chiave musicale e il tempo, attributi che vanno ad influenzare la sua lettura.

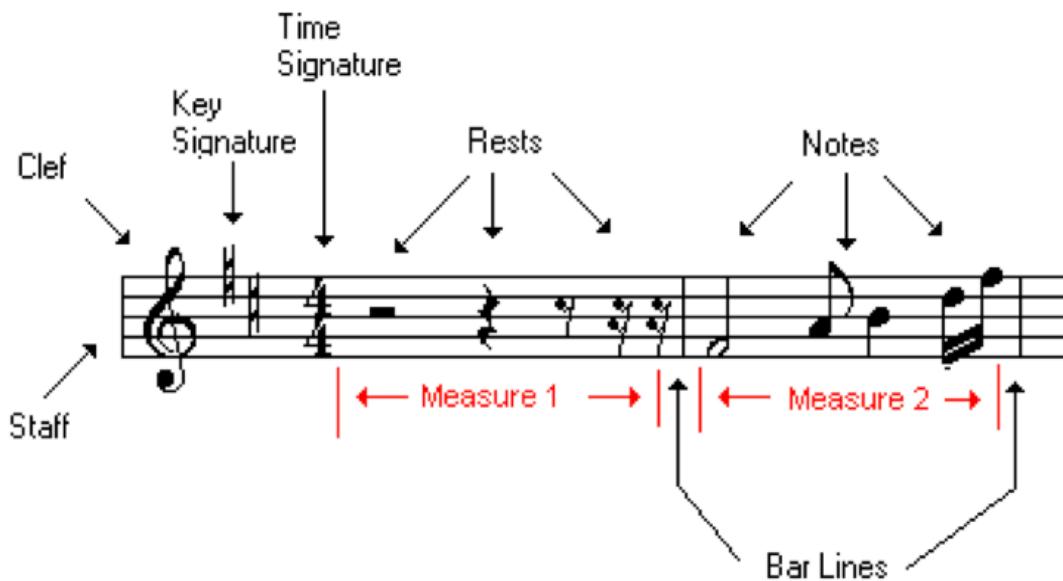


Figura 1 - Composizione Spartito

1.2 Intelligenza Artificiale

Spesso, viene fatta confusione tra Intelligenza Artificiale e Machine Learning in quanto, si crede che siano la stessa cosa ma, non è esattamente così.

L'Intelligenza Artificiale, è la scienza che mira a sviluppare macchine in grado di prendere decisioni autonomamente. È quella scienza, quindi, che sviluppa l'architettura necessaria affinché le macchine funzionino come il cervello umano.

Per sviluppare queste tipologie di macchine, l'intelligenza artificiale utilizza il Machine Learning.

Il Machine Learning, è una disciplina che consente ai robot e ai computer di compiere azioni senza il controllo umano, imparando dall'esperienza; inoltre, permette di migliorare il loro apprendimento ed il loro comportamento con l'esperienza in modo completamente autonomo, fornendogli dati ed informazioni sotto forma di osservazioni ed interazioni con il mondo reale.

Quando svolgiamo un'azione e questa ci riesce particolarmente bene, cerchiamo subito di capire i fattori che hanno permesso di svolgerla al meglio. Abbiamo dunque appreso qualcosa, e la nostra esperienza sarà utile per le nostre azioni future. Ovviamente c'è apprendimento anche quando un'azione non porta a risultati positivi; in questo caso, vale il principio “sbagliando si impara” [1].

1.3 Reti neurali

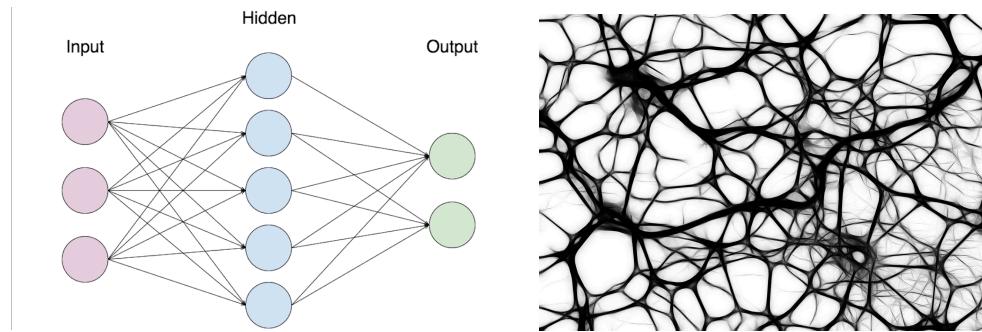
Le reti neurali sono usate nel Machine Learning.

Una rete neurale, è un modello computazionale composto da “neuroni” artificiali, chiamati così poiché sono ispirati ad una semplificazione di rete neurale biologica.

Più in dettaglio, i neuroni sono aggruppati in layer o livelli: il modello adottato è quello in cui sono presenti un certo numero di nodi di input (Input layer), un certo numero di nodi nascosti (Hidden layer) ed un certo numero di nodi di output (Output layer).

Quelli di primo livello, sono incaricati di tradurre il dato come input. Nel caso del riconoscimento di una immagine possiamo pensare al colore di un singolo pixel. Ogni neurone, è connesso ad un altro in base ad un indice che misura il peso della relazione. L'ultimo layer, è formato da neuroni che servono a riconoscere di che elemento si tratta

e contengono la definizione del singolo elemento. Attraverso analisi statistiche sulla base dei pesi delle singole connessioni, viene calcolata la probabilità che una data immagine corrisponda alle informazioni contenute.



1.4 Deep Learning

Le reti neurali “profonde” sfruttano un numero maggiore di strati intermedi (Hidden layer) per costruire più livelli di astrazione affinchè, molteplici livelli di astrazione possano dare alle reti neurali profonde un vantaggio enorme nell’imparare a risolvere complessi problemi di riconoscimento.

Per esempio, i neuroni del primo strato potrebbero imparare a riconoscere i bordi, i neuroni nel secondo strato potrebbero imparare a riconoscere forme più complesse, ad esempio triangoli o rettangoli, create dai bordi. Il terzo strato riconoscerebbe forme ancora più complesse, il quarto riconosce ulteriori dettagli e così via. Proprio perché ad ogni livello intermedio aggiungiamo informazioni ed analisi utili a fornire un output affidabili [2].

Il Deep Learning è una delle tecniche di apprendimento più usate al momento, grazie alla sua capacità di apprendimento e di precisione, vanno però precise alcune particolarità. La precisione del modello è condizionata dalla grandezza delle informazioni fornite al momento dell’addestramento. Più grande è la quantità di informazioni, migliore sarà la precisione del modello. Inoltre, il calcolo richiesto per l’addestramento non è banale, e deve essere preso in considerazione anche dal punto di vista economico, in quanto l’addestramento richiede le migliori GPU che hanno un costo elevato e l’addestramento potrebbe durare settimane o addirittura mesi.

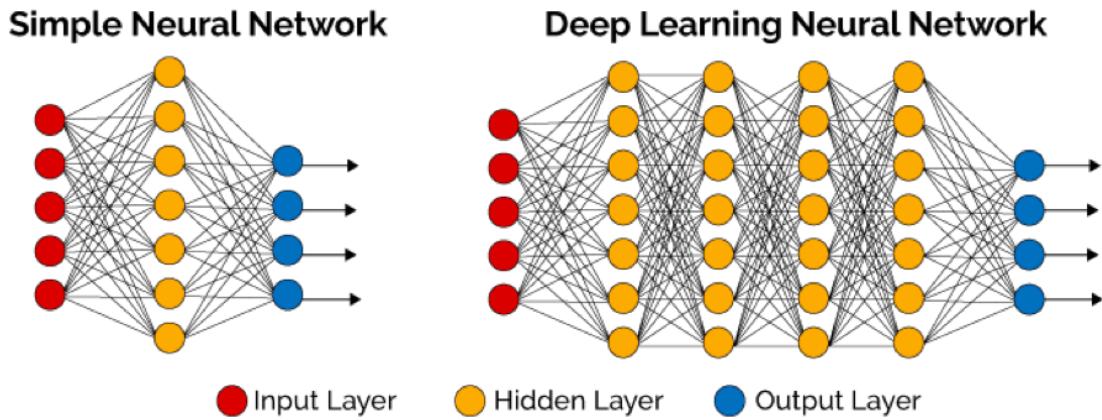


Figura 3 - Comparazione rete neurale semplice e rete profonda

1.5 Computer Vision & Object Detection

La Computer Vision è un campo dell'intelligenza artificiale che addestra i computer ad interpretare e comprendere il mondo visivo utilizzando immagini digitali da fotocamere, video e modelli di Deep Learning. Le macchine possono identificare e classificare con precisione gli oggetti e quindi reagire a ciò che "vedono" [3].

La computer vision, è uno dei campi in cui l'Intelligenza Artificiale è in maggiore espansione, basti pensare alle auto autonome.

Tra i problemi affrontati, vi è il riconoscimento e la categorizzazione di oggetti all'interno di un'immagine (Object Detection).

La classificazione delle immagini implica la previsione della classe di un oggetto in un'immagine. La localizzazione degli oggetti, si riferisce all'identificazione della posizione di uno o più oggetti in un'immagine.

L'Object Detection combina queste due attività, localizza e classifica uno o più oggetti in un'immagine.



Figura 4 - Esempio Object Detection

Capitolo 2

Tecnologie e Strumenti

In questa sezione, verranno elencate e analizzate le principali tecnologie che sono state utilizzate al fine di sviluppare l'intero progetto. Partiremo parlando di cos'è YOLO. Proseguiremo parlando di quali strumenti sono stati utilizzati e finiremo per osservare quelli che ci hanno permesso di effettuare il training del modello predittivo.

2.1 YOLO

Prima di poter parlare di YOLO (You Only Look Once), è doveroso introdurre la storia dell'Object Detection per comprendere al meglio le potenzialità e le migliorie introdotte. Senza scendere troppo nei dettagli, i primi algoritmi di Object Detection non erano in grado di lavorare su una immagine complessa ma, erano in grado di riconoscere solo una immagine con un singolo oggetto, e non erano in grado di generalizzare. In altre parole, potevano essere “configurati” solo su un genere di immagini (facce, cani, etc). Inoltre, avevano problemi di velocità e non erano abbastanza robusti. Da questo, ne conseguiva la difficoltà nel riconoscere immagini con una certa quantità di “rumore” o distrazioni in background.

Con l'avvento del Deep Learning, in particolare con lo sviluppo delle reti convolute, si ebbe un importante progresso. Un approccio comune a quasi tutti gli algoritmi è stato quello della “sliding window”, ovvero scansionare zona per zona tutta l'immagine, analizzandola una porzione alla volta.

Nel caso delle reti convolute (CNN), l'idea è di ripetere il processo con diverse dimensioni della finestra, ottenendo per ciascuna una predizione del contenuto, con grado di confidenza. Alla fine, vengono scartate le predizioni con grado di confidenza più basso.

Questo approccio produce ottimi risultati anche con scene complesse ma, dovendo eseguire la classificazione per migliaia di volte, si potrebbe arrivare ad aspettare minuti per avere il risultato di una sola immagine. Inoltre, le sliding window sono tutte quadrate, il che porta a generare bounding box, che spesso non corrispondono a quello atteso.

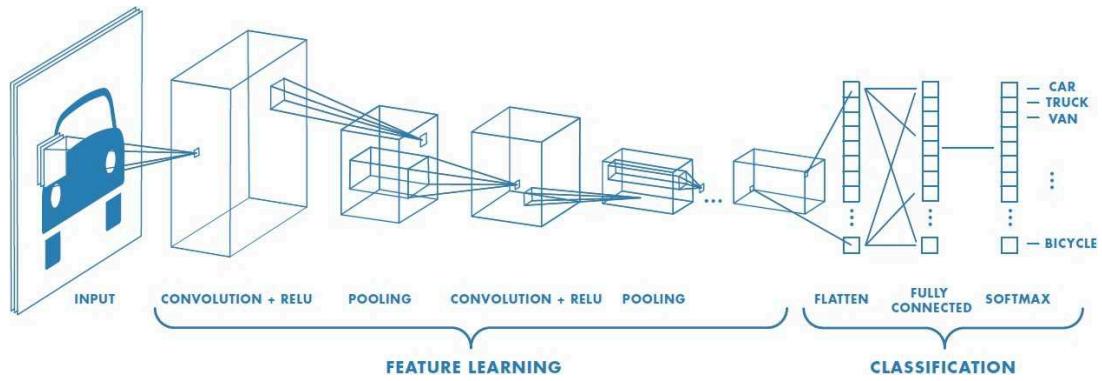


Figura 5 - Rete neurale CNN

La soluzione al problema è quella di utilizzare reti convolute a passata singola, ovvero che analizzano tutte le parti dell'immagine in parallelo, simultaneamente, consentendo di evitare l'uso della sliding window [4].

Il concetto, è quello di ridimensionare l'immagine in modo da ricavarne una griglia di quadrati. Nella terza versione, YOLO fa predizioni su 3 diverse scale, riducendo l'immagine rispettivamente di 32, 16 e 8, allo scopo di rimanere accurata anche su scale più piccole.

L'immagine in ingresso è divisa in una griglia di celle $S \times S$. Per ogni oggetto presente sull'immagine, si dice che una cella della griglia è "responsabile" della sua previsione. Quindi, se in input abbiamo un'immagine di dimensioni 416x416 questa, produrrà 3 diversi griglie di immagini di forma, 13 x 13 x 255, 26 x 26 x 255 e 52 x 52 x 255.

Fondamentalmente, una cella della griglia può rilevare solo un oggetto il cui punto medio dell'oggetto cade all'interno della cella. Per superare questa condizione, YOLOv3 utilizza 3 anchor box per ogni scala di rilevamento. Per ciascuna delle 3 scale, ogni cella è responsabile della previsione di 3 bounding box.

Gli anchor box sono un insieme di riquadri di delimitazione predefiniti di una certa altezza e larghezza che vengono utilizzati per acquisire la scala e le diverse proporzioni di classi di oggetti specifiche che vogliamo rilevare.

Dato che ci sono 3 anchor box previste su ogni scala, le caselle di ancoraggio totali sono 9: (10×13) , (16×30) , (33×23) per la prima scala, (30×61) , (62×45) , (59×119) per la seconda scala e (116×90) , (156×198) , (373×326) per la terza scala.

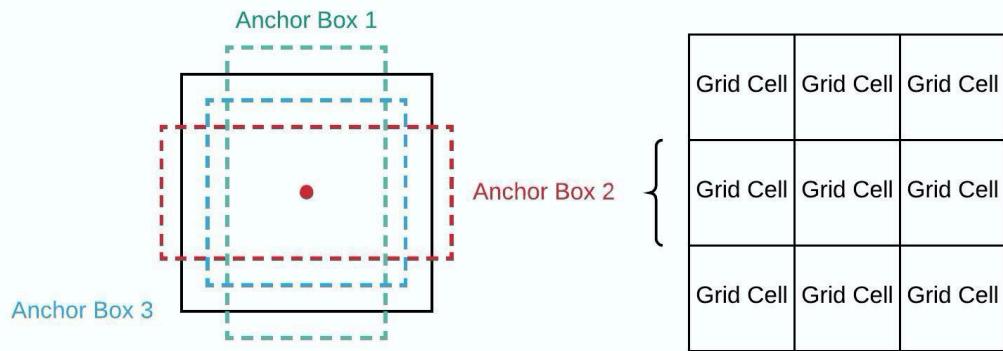


Figura 6 - Esempio anchor box

Ogni cella della griglia predice B riquadri di delimitazione (ROI) e C probabilità. La "B" è associata al numero di anchor box utilizzati. La "C" è il numero di classi che il modello può predire.

Ogni previsione ha 5 componenti: (x, y, w, h, confidenza). Le coordinate (x, y) rappresentano il centro del riquadro, relativo alla posizione della cella della griglia. Queste coordinate sono normalizzate (divise) rispettivamente per la larghezza e l'altezza dell'immagine per essere comprese tra 0 e 1. Anche le dimensioni del riquadro (w, h) sono normalizzate, rispetto alla dimensione dell'immagine.

Al termine dell'elaborazione, vengono mantenute solo le bounding box con la confidenza più elevata, scartando le altre.

Yolo v3 è in grado di lavorare con più di 80 classi differenti e risulta molto più preciso delle versioni precedenti e, pur essendo un po' più lento, rimane comunque uno degli algoritmi più veloci in circolazione. La terza versione usa come architettura una variante di Darknet, con 106 layer convoluti. Vi è anche Tiny YOLO, funzionante su Tiny Darknet che è in grado di girare su dispositivi limitati come gli smartphone.

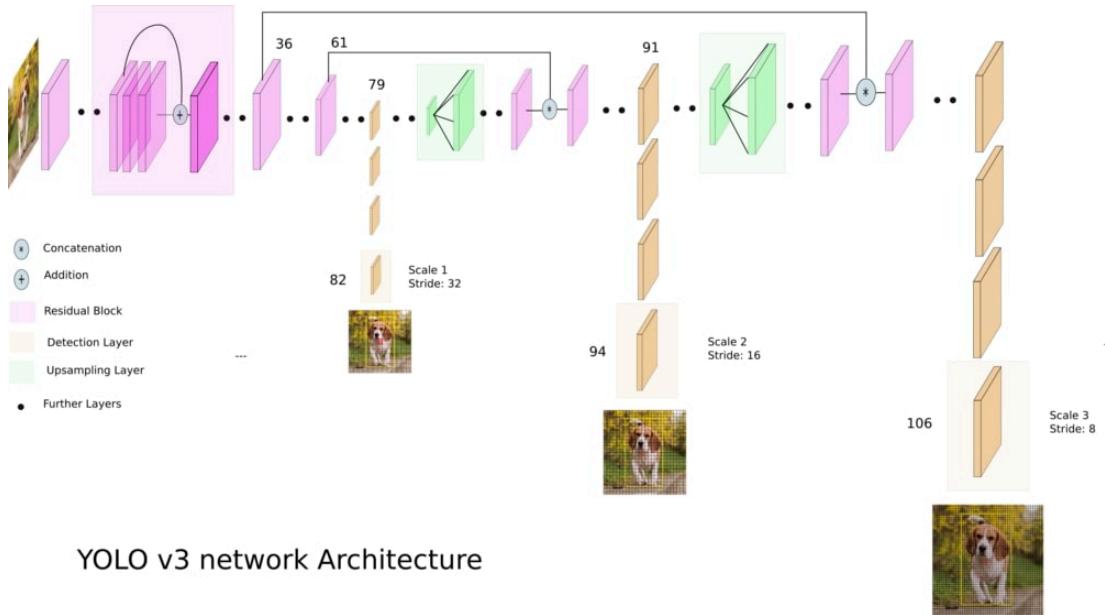


Figura 7 - Rete neurale YOLO

2.2 Swift

Swift è un linguaggio di programmazione open source sviluppato e gestito da Apple.

È un'evoluzione del linguaggio Objective-C che Apple utilizza da quando il co-fondatore Steve Jobs lo ha concesso in licenza decenni fa, ed è costruito per essere una versione semplificata ed altamente estensibile di Objective-C.

Oltre a Objective-C, Swift incorpora aspetti di Python, Rust, Ruby ed altri linguaggi. Molte delle sue funzionalità si concentrano sul rendere Swift più facilmente utilizzabile; questo include il supporto delle stringhe migliorato, tipi di opzioni e misure per la protezione da errori di programmazione come la dereferenziazione dei puntatori nulli o l'overflow di interi.

Swift è compatibile solo con i sistemi operativi di Apple, quindi non è possibile utilizzarlo per sviluppare software per dispositivi come Android o Windows. Puoi, tuttavia, usarlo per scrivere codice per qualsiasi prodotto Apple, poiché funziona su macOS, tvOS, iPadOS, watchOS e iOS. Linux è l'unica eccezione a questo, poiché Swift supporta il popolare kernel del sistema operativo open source [5].



Figura 8 - Swift logo

2.3 Xcode

Xcode è un ambiente di sviluppo integrato (IDE) prodotto e mantenuto da Apple per la creazione di applicativi software per le sue piattaforme. Contiene tutti gli strumenti necessari per la creazione di applicazioni native iPhone e iPad, fondendo numerose interfacce grafiche [6].

Supporta il linguaggio proprietario Swift e la compilazione incrementale, ovvero, Xcode è in grado di compilare il codice mentre viene scritto, in modo da ridurre il tempo di compilazione.

Infine, fornisce un ottimo ambiente di testing. Il software prodotto può essere testato sui vari dispositivi disponibili, attraverso l'utilizzo della virtualizzazione.



Figura 9 - Xcode logo

2.4 CoreML

CoreML, è il framework di machine learning utilizzato nei prodotti Apple per eseguire previsioni o inferenze rapide con una facile integrazione di modelli di machine learning pre-addestrati.

Grazie alla ottimizzazione delle prestazioni sul dispositivo sfruttando CPU, GPU e Neural Engine consente di eseguire previsioni in tempo reale sul dispositivo.



Figura 10 - CoreML logo

L'utilizzo di un modello predittivo dal dispositivo però presenta dei pro e dei contro da analizzare.

Pro:

- **Bassa latenza:** non è necessario effettuare una chiamata API in rete inviando i dati e quindi aspettando una risposta. Questo può essere fondamentale per applicazioni real-time.
- **Disponibilità (offline) e privacy:** l'applicazione viene eseguita senza connessione di rete, senza chiamate API e i dati non lasciano mai il dispositivo.

Contro:

- **Dimensioni:** aggiungendo il modello al dispositivo, aumentando le

dimensioni dell'app e alcuni modelli accurati possono essere piuttosto grandi.

- **Utilizzo delle risorse:** la previsione e l'inferenza sul dispositivo mobile richiedono molti calcoli, il che aumenta il consumo della batteria. I dispositivi più vecchi potrebbero avere difficoltà a fornire previsioni in tempo reale.

Infine, Apple oltre al rilascio di CoreML ha rilasciato un tool in python che permette di convertire i modelli pre-addestrati dai formati più noti al formato riconosciuto da CoreML. Migliorando la compatibilità tra i vari dispositivi e permettendo ai programmatore di poter sviluppare un modello predittivo con il framework più indicato.

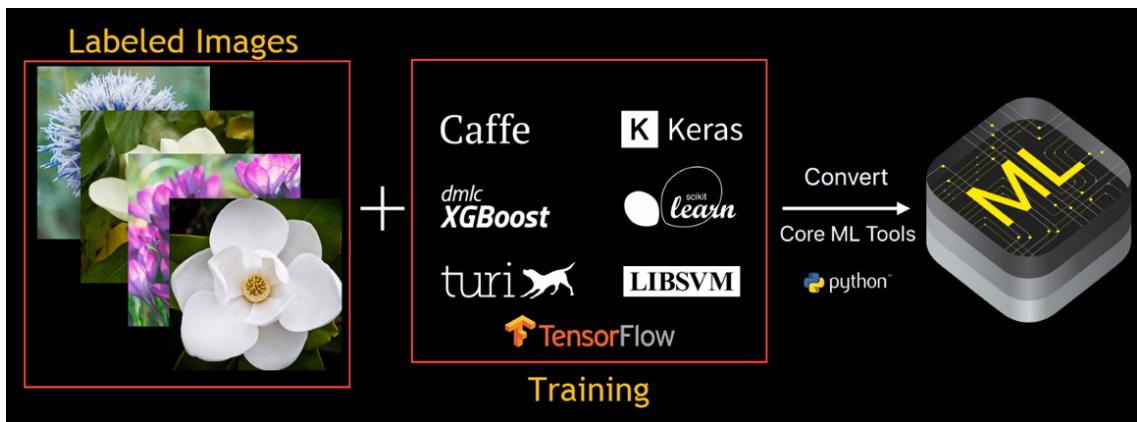


Figura 11 - CoreML tool pipeline

2.4.1 Vision Framework

Quando si utilizza Core ML per risolvere problemi di computer vision, come la classificazione ed il rilevamento di oggetti, il framework Vision ci può essere di aiuto. Sviluppato da Apple, Vision permette con un'interfaccia semplificata di compiere operazioni complesse come il rilevamento di volti e punti di riferimento, rilevamento di testo, riconoscimento di codici a barre e registrazione di immagini, in poche righe di codice.

2.5 Core Data

Core Data è il framework di persistenza dati distribuito da Apple per i suoi prodotti, utilizzato per gestire strutture dati complesse, dove una buona organizzazione delle

informazioni è essenziale.

Consente di serializzare i dati organizzati sul modello entità -relazione in archivi XML, binari o SQLite [7]. I dati possono essere manipolati utilizzando oggetti con una astrazione superiore che rappresentano le entità e le loro relazioni.

Core Data gestisce il ciclo di vita degli oggetti, inclusa la persistenza. Astrae l’interfaccia di SQLite, esponendo una interfaccia all’utente semplificata rendendo trasparente il suo utilizzo sottostante.

Infine, si occupa della gestione su disco, della gestione dei cambiamenti, della minimizzazione della memoria occupata e delle query su disco.

2.6 Python

Python, è un linguaggio di programmazione molto popolare in questo periodo, grazie alla sua sintassi molto simile alla lingua inglese che permette di scrivere programmi con meno righe di codice rispetto ad altri linguaggi di programmazione ed alla community che crea e fornisce liberamente pacchetti aggiuntivi [8].

È fortemente utilizzato nel mondo dei Big Data e nel Machine Learning ma, il suo dominio di applicazione è davvero molto ampio in quanto viene utilizzato per applicazioni web (lato server), sviluppo software e scrittura di script di sistema, per eseguire operazioni matematiche complesse, per la prototipazione rapida o per sviluppare software pronto per la produzione. Infine, esso funziona su molteplici piattaforme quali, ad esempio, Windows, Mac, Linux o Raspberry Pi.



Figura 12- Python logo

2.7 Jupyter Notebook

Una Jupyter Notebook è un ambiente di elaborazione interattivo che consente agli utenti di creare documenti chiamati “notebook” che includono: live code, Widget interattivi, Grafici, Testo, Immagini, Video ed ecc.

Un notebook combina tre componenti:



Figura 13 - Jupyter logo

- **L'applicazione Web:** un'applicazione Web interattiva per la scrittura e l'esecuzione di codice in modo interattivo e la creazione di documenti.
- **Kernels:** processi separati avviati dall'applicazione web del notebook che esegue il codice degli utenti in un determinato linguaggio di programmazione e restituisce l'output all'applicazione web del notebook. Il kernel gestisce anche cose come calcoli per widget interattivi, completo con tabulazioni e introspezione.
- **Documenti del notebook:** documenti autonomi che contengono una rappresentazione di tutto il contenuto visibile nell'applicazione Web del notebook, inclusi input e output dei calcoli, testo narrativo, equazioni, immagini e rappresentazioni multimediali di oggetti. Ogni documento del notebook ha il proprio kernel.

Attraverso il kernel, il notebook consente l'esecuzione del codice in una gamma di linguaggi di programmazione diversi. Per ogni notebook che un utente apre, l'applicazione web avvia un kernel che esegue il codice per quel notebook. Ogni kernel è in grado di eseguire codice in un singolo linguaggio di programmazione. Sono disponibili una serie di linguaggi di programmazione ma, il predefinito è Python.

I documenti notebook contengono gli input e gli output di una sessione interattiva, nonché il testo che accompagna il codice.

Quando si esegue l'applicazione Web del notebook sul computer, i documenti del notebook sono salvati sul file system locale con un'estensione “.ipynb”.

I notebook sono costituiti da una sequenza lineare di celle. Esistono tre tipi di cellule di base:

- **Celle codice:** input e output del codice live eseguito nel kernel.
- **Celle di markdown:** testo.
- **Celle grezze:** testo non formattato incluso, senza modifiche.

Plotting data and linear model

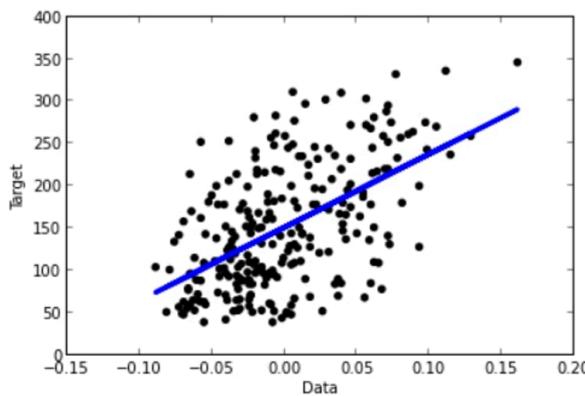
Now we want to plot the train data and teachers (marked as dots).

Text

With line we represents the data and predictions (linear model that we found):

```
In [14]: # Visualises dots, where each dot represent a data example and corresponding teacher
plt.scatter(x_train, y_train, color='black')
# Plots the linear model
plt.plot(x_train, regr.predict(x_train), color='blue', linewidth=3);
plt.xlabel('Data')
plt.ylabel('Target')
```

```
Out[14]: <matplotlib.text.Text at 0xb101b0cc>
```



Code

Plot

Figura 14 - Jupyter Notebook esempio

2.8 Google Colab

Colab è un servizio per notebook Jupyter che non richiede alcuna configurazione iniziale, fornendo al contempo accesso gratuito alle risorse di elaborazione, comprese le GPU.

Viene spesso utilizzato nel mondo del Machine Learning poichè, permette di utilizzare per un periodo di tempo limitato delle ottime schede video

(Nvidia Tesla K80 e Nvidia Tesla P100 nella versione Pro) e, pre-installa i componenti necessari per utilizzare framework come TensorFlow, TensorRT e Nvidia CUDA.

Fornisce la possibilità di lavorare sullo stesso notebook con altri collaboratori, visualizzando le relative modifiche o esecuzioni in real-time.

Inoltre, è possibile montare come partizione il proprio spazio di Google Drive, che risulta molto utile quando si vuole salvare i dati di un relativo training, per poter riprendere in un momento successivo.



Figura 15 - Google Colab logo

Colab però presenta anche diverse limitazioni:

- Una sessione può essere utilizzata fino ad un massimo di 12ore, dopo di ché viene deallocata.
- Dopo due ore di inattività la sessione viene deallocata.
- Dopo un uso eccessivo della GPU, vi può essere deallocata.

Queste limitazioni sono fin troppo limitanti, specialmente se si vuole addestrare un modello che richiede molto calcolo e tempo.

Colab fornisce anche una versione a pagamento (Colab Pro) che viene incontro a queste esigenze, fornendo anche una GPU più potente della versione classica.

Capitolo 3

Modello Predittivo

In questa sezione verranno spiegate e motivate le operazioni effettuare per la creazione del modello predittivo. Partiremo dalla creazione del dataset fino ad arrivare all'addestramento e alla conversione del modello in un formato utilizzabile su piattaforme iOS.

3.1 Motivazioni

L'architettura scelta per questo modello predittivo è Yolov3.

YOLO offre, come già accennato nel capitolo 3, due tipologie di reti neurali: Tiny e Full. Tiny yolo è stata progettata per dispositivi con risorse limitate, presenta un numero inferiore di strati di nodi nascosti e di anchor box rispetto alla versione Full, permettendo un migliore throughput, riuscendo ad elaborare più di 30 frame al secondo anche su dispositivi limitati come uno smartphone. queste scelte di progettazione lo rende perfetto per applicazioni real-time, ma bisogna ricordare che, queste scelte di progettazione vanno a discapito della precisione e della robustezza del modello.

Nel nostro caso si è scelto di usare la versione Full, poiché non si è avuta la necessità di elaborare un numero alto di fotogrammi al secondo, ma bensì bisogna elaborare un solo fotogramma per volta con alta precisione andando a discapito della performance, cercando ovviamente di rimanere in un tempo di risposta accettabile (5-6 sec).



Figura 16 - Yolov3 full esempio

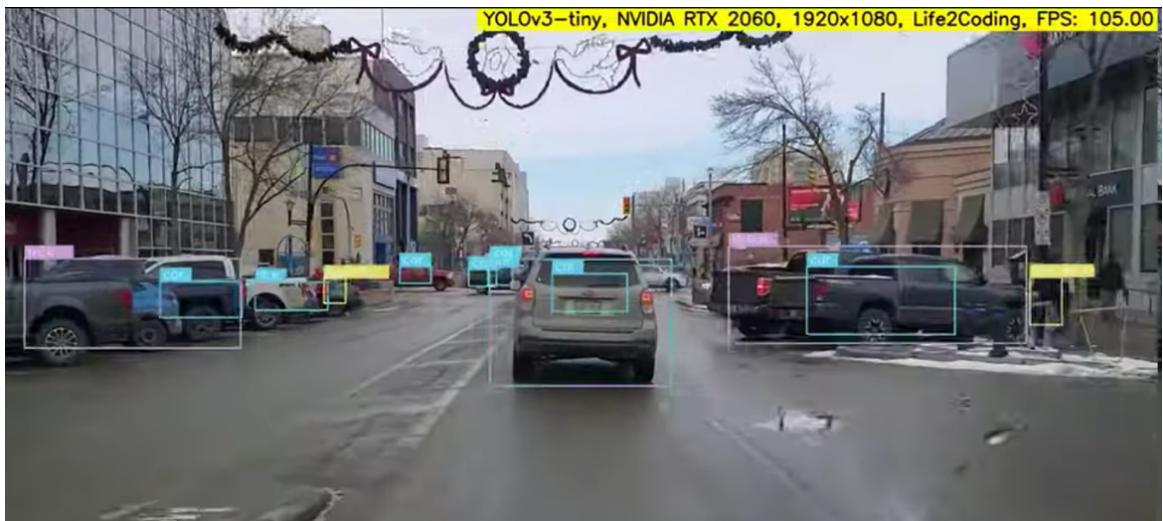


Figura 17 - Yolov3 tiny esempio

3.2 Creazione del Dataset

Un dataset, è una collezione di dati ben strutturata, fornisce in fase di training le informazioni necessarie del dominio di applicazione.

Il modello predittivo, che vogliamo creare deve poter individuare e riconoscere gli elementi che si trovano all'interno di uno spartito musicale.

Il primo passo da compiere è collezionare immagini che rappresentano degli spartiti musicali.

Queste immagini dovranno essere elaborate, ovvero andremo ad indicare all'interno di esse quali elementi vi sono e dove si trovano in un formato che potrà essere elaborato da un computer. Per far ciò utilizzeremo un tool di labeling, nel nostro caso HyperLabel,

che ci permette di inserire delle label all'interno delle immagini.

Le label non sono altro che dei bounding box a cui è associato una classe di oggetti.

La fase di labelling è molto costosa in termini di tempo, poiché l'inserimento di tutte le label all'interno di un'immagine mediamente complessa (più di 100 elementi da marcare al suo interno) può durare anche mezz'ora. Inoltre, in questa fase bisogna meticolosi, evitando di tralasciare elementi che il modello si aspetta di riconoscere ed evitando quando più possibili errori. La qualità del modello predittivo è dettata dalla qualità del dataset con cui è addestrato.

Il modello predittivo deve essere capace di predire 58 elementi differenti, quindi nel nostro dataset avremo 58 classi di elementi che dovranno essere marcati.

Aria Italiana

Mozart

The sheet music consists of eight staves of musical notation in G major (indicated by a G clef) and common time (indicated by a 'C'). The lyrics are written below each staff in a color-coded solfège system: red for 'sol', blue for 'fa', green for 'mi', orange for 'mi' (repeated), yellow for 'mi' (repeated), and black for 'fa' (repeated). The lyrics are as follows:

sol fa mi mi mi mi sol fa fa mi re re re re fa mi mi fa

sol sol sol sol fa mi fa sol la fa mi mi re re do mi do sol fa mi mi mi mi

sol fa fa mi re re re fa mi mi fa sol sol sol sol fa mi fa sol la fa

mi mi re re do mi do fa mi re re re re fa mi mi sol fa fa fa mi fa sol

fa mi mi fa sol sol sol fa mi fa sol la fa mi mi re re do mi do fa mi

re re re re fa mi mi sol fa fa fa mi fa sol fa mi mi fa sol sol sol sol

fa mi fa sol la fa mi mi re re do mi do

Figura 18 - Immagine non elaborata

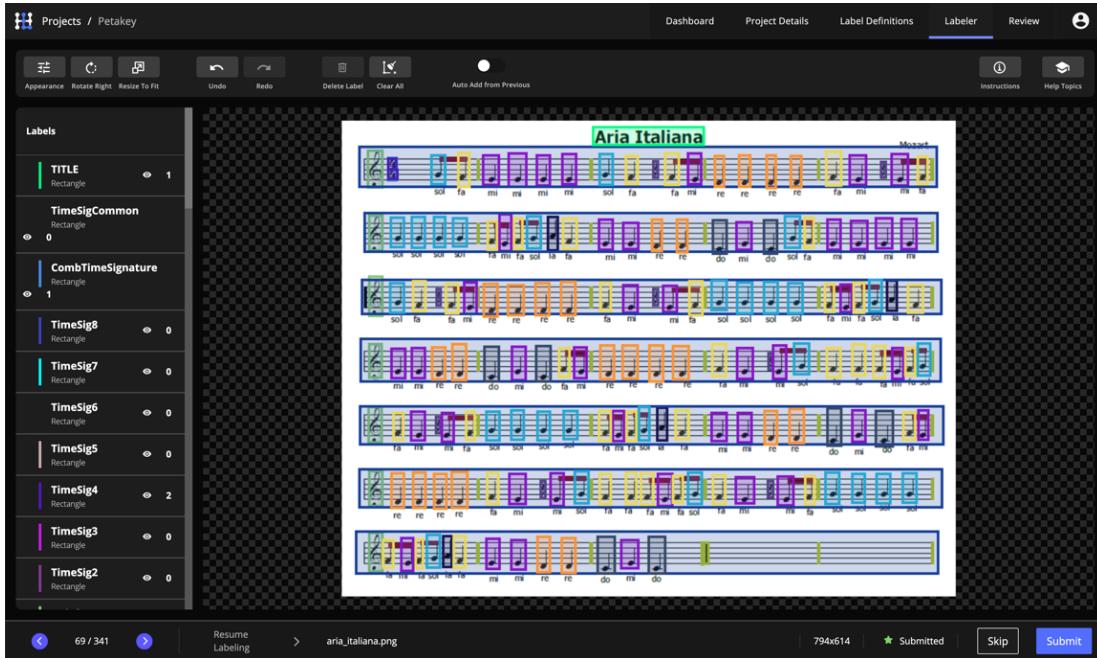


Figura 19 - Immagine elaborata

Il dataset da noi formato è composto da 341 immagini di media e alta complessità.

YOLO definisce un proprio standard per la rappresentazione dei dataset.

Nel file “obj.data” dobbiamo fornire informazioni e alcuni percorsi rilevanti:

- classes
- train
- valid
- names
- backup

Il parametro `classes` richiede il numero di classi. Nel nostro caso, è 58.

Inoltre, è necessario fornire i percorsi assoluti dei file di testo che contengono l'elenco dei file da utilizzare rispettivamente per l'addestramento (parametro `train`) e la convalida (parametro `valid`).

Il campo dei nomi rappresenta il percorso di un file che contiene i nomi di tutte le classi. Infine, per il parametro di `backup`, dobbiamo fornire il percorso a una directory esistente dove possiamo memorizzare i file dei checkpoint che man mano durante l'addestramento verranno creati.

Per ogni immagine si crea un file di testo che conserva le informazioni di ogni label. Ogni riga del file rappresenta una label dell'immagine a cui si riferisce, e viene

rappresentata in questo modo:

< object-class-id > <center-x > <center-y > <width> <height>

Il primo campo object-class-id è un numero intero che rappresenta la classe dell'oggetto. Va da 0 a (numero di classi - 1). Nel nostro caso attuale, abbiamo un range di interi che fa da 0 a 57 compreso.

La seconda e la terza voce, center-x e center-y sono rispettivamente le coordinate x e y del centro del riquadro di delimitazione, normalizzate.

La quarta e la quinta voce, larghezza e altezza sono rispettivamente la larghezza e l'altezza del riquadro di delimitazione, nuovamente normalizzata rispettivamente per la larghezza e l'altezza dell'immagine.

Le quattro voci precedenti sono tutte valori in virgola mobile compresi tra 0 e 1.

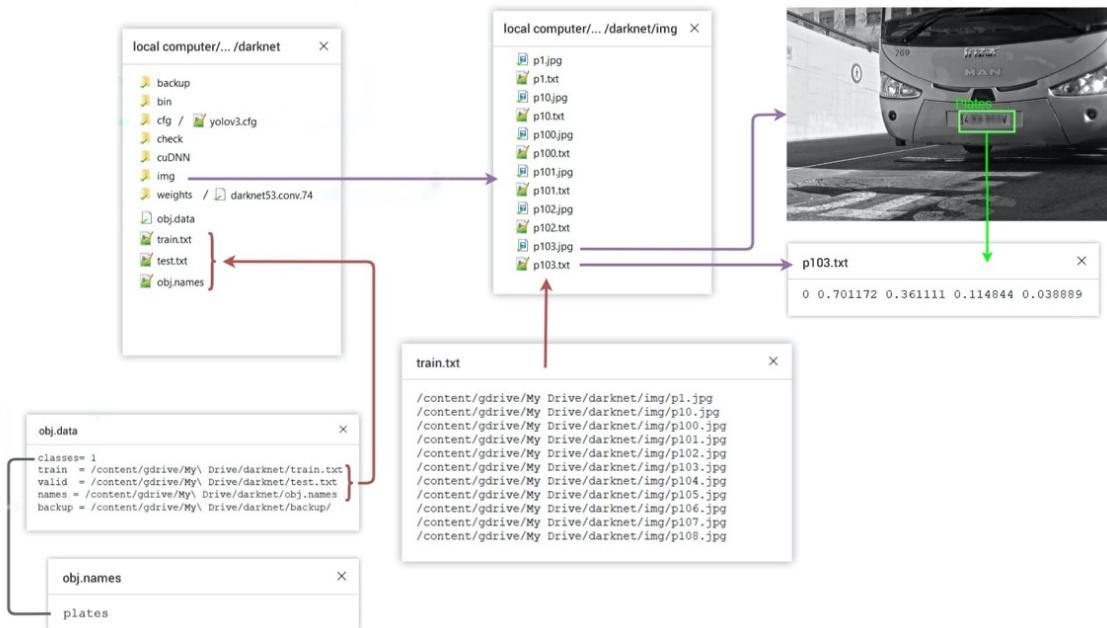


Figura 20 - Formato dataset YOLO

Possiamo tralasciare questa problematica del formato utilizzando l'interfaccia di HyperLabel che permette di esportare i dataset in diversi formati, tra cui YOLO.

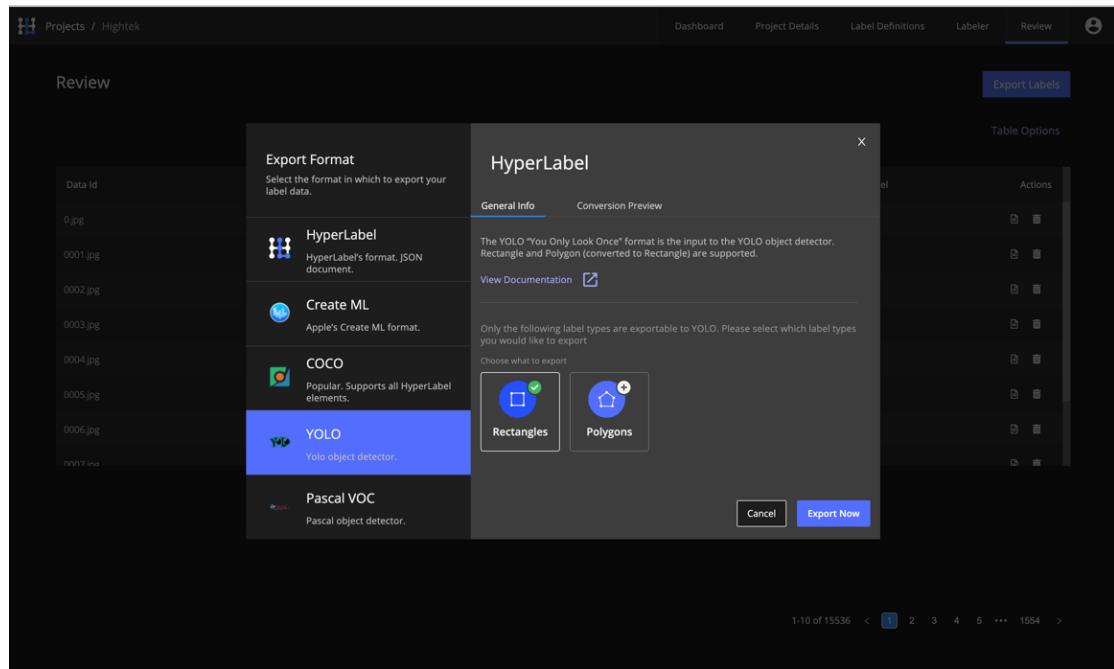


Figura 21 - HyperLabel interfaccia esportazione dataset

3.3 Setup

Prima di procedere con l'avvio dell'addestramento, abbiamo bisogno di generare un file di configurazione che YOLO utilizza per ricavare tutti i parametri che necessita per l'addestramento.

YOLO mette a disposizione molti esempi di file di configurazione, noi andremo a modificare il file “yolov3-spp.cfg” poiché possiede delle impostazioni aggiuntive per migliorare il riconoscimento di oggetti di piccole dimensioni all'interno di una immagine.

Sono da prendere in considerazione i seguenti paramenti:

- width
- height
- batch
- subdivisions
- max_batches

Width e height rappresentano la grandezza dell'immagine di input, possono assumere solo valori uguali e multipli di 32 per via dell'architettura di YOLO. Più grande è la grandezza dell'input migliore sarà la precisione del nostro modello. Purtroppo, la

quantità di calcolo richiesta e la dimensione dell'input sono direttamente proporzionali, andando ad aumentare la dimensione dell'input aumenteremo anche il tempo di addestramento e la quantità di memoria VRAM richiesta.

Il valore che assume “batch” chiamato anche “batch size” indica la quantità di immagini che deve essere elaborata ad ogni iterazione, quindi la quantità di immagini che dovranno essere spostate nella VRAM. Anche questo valore va ad influire sulla qualità del modello, infatti al suo aumentare, aumenta anche la qualità del modello prodotto. Il valore che assumerà è influenzato dalla capacità della VRAM, nel caso in cui la batch size fosse troppo alta avremo un errore, “CUDA OUT OF MEMORY”.

Il valore che assume “subdivisions” permette di dividere il batch principale in dei “mini-batch”. Per esempio, supponiamo di avere una batch size a 64, che eccedere il limite fisico della GPU. Settando la subdivisions a 16, YOLO dividerà il batch principale in 4 (64/16) mini-batch, così facendo non verranno più caricati 64 immagini alla volta ma bensì 16 in 4 step. Aumentando la grandezza di subdivisions, aumenteremo anche la quantità di tempo richiesta per l'addestramento.

Infine, il valore che assume max_batches indica il numero massimo di batch che dovrà elaborare. Per un rilevatore di oggetti di classi n, è consigliabile eseguire l'addestramento per almeno $2000 * n$ batch. Nel nostro caso con 58 1 classi, avremo 116000 batch.

Utilizzando Colab in versione Pro abbiamo a disposizione la scheda video Nvidia Tesla P100, che possiede 16GB di VRAM.

Si è deciso di settare l'input ad 800x800, batch a 64 e subdivisions a 32.

Il processo di addestramento per completare i 116000 batch, ha impiegato circa 600 ore.

3.4 Addestramento

Una volta decise le configurazioni da utilizzare e dopo aver creato il dataset, si può procedere con il vero e proprio addestramento.

L'addestramento, come già detto in precedenza verrà eseguito sulla piattaforma Colab che anche se in versione Pro non manca di limitazioni.

Non ci è consentito avviare la procedura di addestramento e lasciarla attiva per più di 12 ore, ma soprattutto non potevamo lasciare la procedura in background, ovvero per mantenere attiva la sessione bisogna avere almeno un dispositivo collegato. Nel caso

contrario dopo 20 minuti la sessione veniva distrutta per inattività.

La procedura di addestramento utilizza molti dati statici come per esempio, il dataset o eventuali elementi necessari per il training, per evitare di doverli caricare ogni volta su Colab, li caricheremo una sola volta nello spazio offerto dal servizio Google Drive. Una volta fatto, ci basterà montare lo spazio su Colab e spostare tutto il necessario al momento di esecuzione.

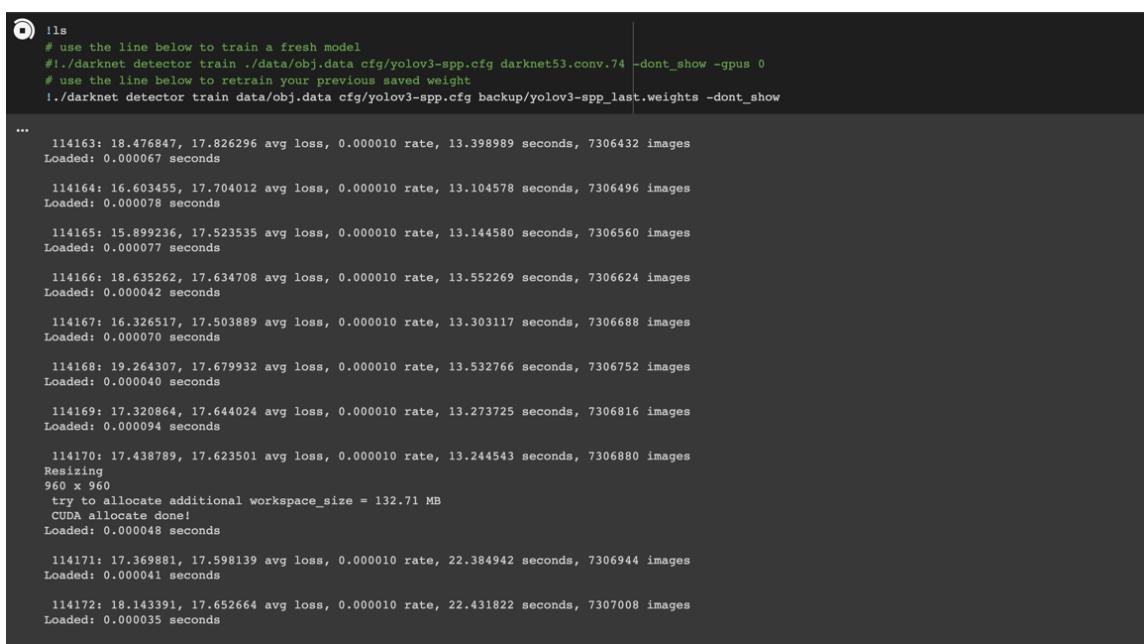
Per migliorare la velocità di addestramento utilizzeremo il processo del transfer learning. Nel transfer learning, la conoscenza di un modello già addestrato viene applicata a un problema diverso ma correlato, sostanzialmente cerchiamo di sfruttare ciò che è stato appreso in un'attività per migliorare la generalizzazione in un'altra.

Per un nuovo modello di object detection è una buona idea sfruttare i modelli esistenti addestrati su dataset molto grandi, anche se al loro interno potrebbero non contenere gli oggetti che si sta tentando di rilevare.

Nel nostro caso, utilizziamo “darknet53.conv.74” un modello fornito da YOLO, pre-addestrato su ImageNet, un dataset che contiene 14 milioni di immagini e 80 classi di oggetti.

Al primo avvio faremo partire il training dal modello pre-addestrato, dopodiché ad ogni 100 batch elaborate YOLO salverà un checkpoint.

Grazie ad essi, nel caso di arresto della procedura di addestramento potremo ripartire dall'ultimo checkpoint salvato in memoria.



```

ls
# use the line below to train a fresh model
# ./darknet detector train ./data/obj.data cfg/yolov3-spp.cfg darknet53.conv.74 -dont_show -gpus 0
# use the line below to retrain your previous saved weight
./darknet detector train data/obj.data cfg/yolov3-spp.cfg backup/yolov3-spp_last.weights -dont_show

...
114163: 18.476847, 17.826296 avg loss, 0.000010 rate, 13.398989 seconds, 7306432 images
Loaded: 0.000067 seconds

114164: 16.603455, 17.704012 avg loss, 0.000010 rate, 13.104578 seconds, 7306496 images
Loaded: 0.000078 seconds

114165: 15.899236, 17.523535 avg loss, 0.000010 rate, 13.144590 seconds, 7306560 images
Loaded: 0.000077 seconds

114166: 18.635262, 17.634708 avg loss, 0.000010 rate, 13.552269 seconds, 7306624 images
Loaded: 0.000042 seconds

114167: 16.326517, 17.503889 avg loss, 0.000010 rate, 13.303117 seconds, 7306688 images
Loaded: 0.000074 seconds

114168: 19.264307, 17.679932 avg loss, 0.000010 rate, 13.532766 seconds, 7306752 images
Loaded: 0.000040 seconds

114169: 17.320864, 17.644024 avg loss, 0.000010 rate, 13.273725 seconds, 7306816 images
Loaded: 0.000094 seconds

114170: 17.438789, 17.623501 avg loss, 0.000010 rate, 13.244543 seconds, 7306880 images
Resizing
960 x 960
try to allocate additional workspace_size = 132.71 MB
CUDA allocate done!
Loaded: 0.000048 seconds

114171: 17.369881, 17.598139 avg loss, 0.000010 rate, 22.384942 seconds, 7306944 images
Loaded: 0.000041 seconds

114172: 18.143391, 17.652664 avg loss, 0.000010 rate, 22.431822 seconds, 7307008 images
Loaded: 0.000035 seconds

```

Figura 22 - Esecuzione addestramento

3.5 Risultati

Nei capitoli e nei paragrafi precedenti abbiamo parlato delle tecnologie utilizzate e del procedimento che ci ha consentito di addestrare un modello predittivo per l'object detection. Però prima di proseguire mostrando i risultati del modello, mi sembra doveroso sottolineare che questa soluzione finale è stata frutto di altre sessioni di addestramento.

Prima di giungere a questa soluzione, vi sono stati diversi test che ci hanno permesso di capire il funzionamento della rete neurale e come potevamo migliorare il risultato atteso.

Inizialmente, abbiamo effettuato dei test con un dataset ridotto, intorno alle 70-80 immagini e con solo 7 classi di oggetti da riconoscere.

Una volta compresa la tecnologia, siamo passati a studiare quali classi di oggetti bisognava aggiungere, non solo per i fini del riconoscimento, ma soprattutto ai fini della creazione di un algoritmo che possa elaborare con correttezza le informazioni risultanti dal modello.

Solo grazie a tutto lo studio e ai test effettuati siamo arrivati a questi risultati.

Aggiungi un posto a tavola *Armando Trovajoli*

sol sol sol sol la, sol fa mi sol, DO DO DO RE, MI MI MI MI FA, MI RE DO DO, RE RE RE MI, RE, sol sol sol sol la, sol fa mi sol, MI FA MI RE, DO sol, la DO DO RE, MI sol sol sol, la DO DO RE, MI sol, la DO DO RE, MI sol sol sol, la DO DO.



Figura 23 - Esempio predizione immagine

Il modello ha riconosciuto il 100% degli oggetti che è capace di rilevare all'interno dell'immagine di input.

3.6 Elementi riconosciuti

Il modello nello stato attuale è capace di riconoscere 58 classi, suddividendole in categorie avremo:

- 21 note (le note comprese dalla 3 alla 5 ottava)
 - 34 simboli
 - 9 elementi per identificare il tempo dello spartito; Ovvero la frazione che si trova solitamente subito dopo la chiave musicale
 - 3 chiavi musicali (C, G, e F)
 - 7 classi di pause; Ogni classe identifica una pausa di tempo differente.
 - 6 code per le note musicali.
 - 7 attributi degli elementi (diesis, bemolle, beam, augmentation dot ed ecc.)
 - 2 attributi dello spartito (brace e barlineSingle)
 - 3 elementi per la corretta elaborazione delle informazioni
 - **Score**: come vedremo nel capitolo 6 verrà utilizzato per ordinare gli

elementi presenti nello sparito.

- **CombTimeSignature**: viene utilizzato per l'individuazione del tempo, al suo interno vi sono il denominatore e il nominatore della frazione di tempo.
- **Title**: individua la regione dov'è presente il titolo dello spartito.

3.7 Conversione a CoreML

YOLO fornisce il modello nel formato “.weight”, per utilizzarlo sui sistemi Apple abbiamo bisogno di convertirlo nel formato utilizzato da CoreML.

Anche se Apple fornisce un tool per la conversione, non è possibile convertire direttamente da YOLO a CoreML.

Per poter risolvere il problema, aggiungiamo uno step intermedio.

È possibile convertire un modello da YOLO a Keras, e da lì tramite il tool di conversione, potremmo ricavarci il modello per CoreML.

Capitolo 4

User Experience

In questo capitolo, verranno illustrate le scelte di design che ci hanno portato alla creazione dell’interfaccia grafica. Parleremo di come l’utente si interfaccia e comunica con essa. Inizieremo descrivendo cosa sono le humans interface guidelines di Apple ed il perché è importante seguirle. Continueremo mostrando i mockup dell’applicativo fino ad arrivare all’interfaccia grafica realizzata ed adoperata.

4.1 Introduzione

Apple utilizza una logica molto chiara per il design di una applicazione iOS, fornendo ai designer una ricca documentazione per migliorare l’user experience, le **Human Interface Guidelines (HIG)**.

Le HIG forniscono tutto il necessario per creare un design in pieno stile Apple, soprattutto se non si è un designer risultano utili come guida durante l’intero processo di sviluppo. Offre una visione completa e di alto livello degli elementi chiave dell’interfaccia utente e le “best practices” per aiutarti ad implementare le funzionalità della tua applicazione, fondendo anche il codice sorgente agli elementi che costituiscono il design.

Infine, creare un design seguendo le HIG permette di evitare che l’utente finale si trovi a dover imparare una nuova interfaccia e quindi, rendere inefficace l’esperienza utente poiché, essendo già abituato ad utilizzare applicativi di fabbrica un design simile rende più semplice e intuitiva.

4.2 Progettazione

Sostanzialmente, l’applicativo che vogliamo progettare non contiene molte viste utente in quanto, avremo una sezione dove visualizzare la libreria di spartiti e una sezione dove è possibile editarli e visualizzarli.

Per la creazione del design abbiamo deciso di mantenerci quanto più vicini alle HIG utilizzando scelte di design utilizzate in applicazioni di fabbrica.

Utilizzeremo una Collection View per la vista contenente la collezione di spartiti. Per la visualizzazione e la loro modifica, sarà doveroso creare una vista custom poiché, non vi è nessuna vista che ci permetterebbe di svolgere la funzionalità richiesta. Infine, sapendo che l'applicativo verrà utilizzato su piattaforme iPad sfrutteremo la loro dimensione utilizzando per i sottomenu dei popOver.

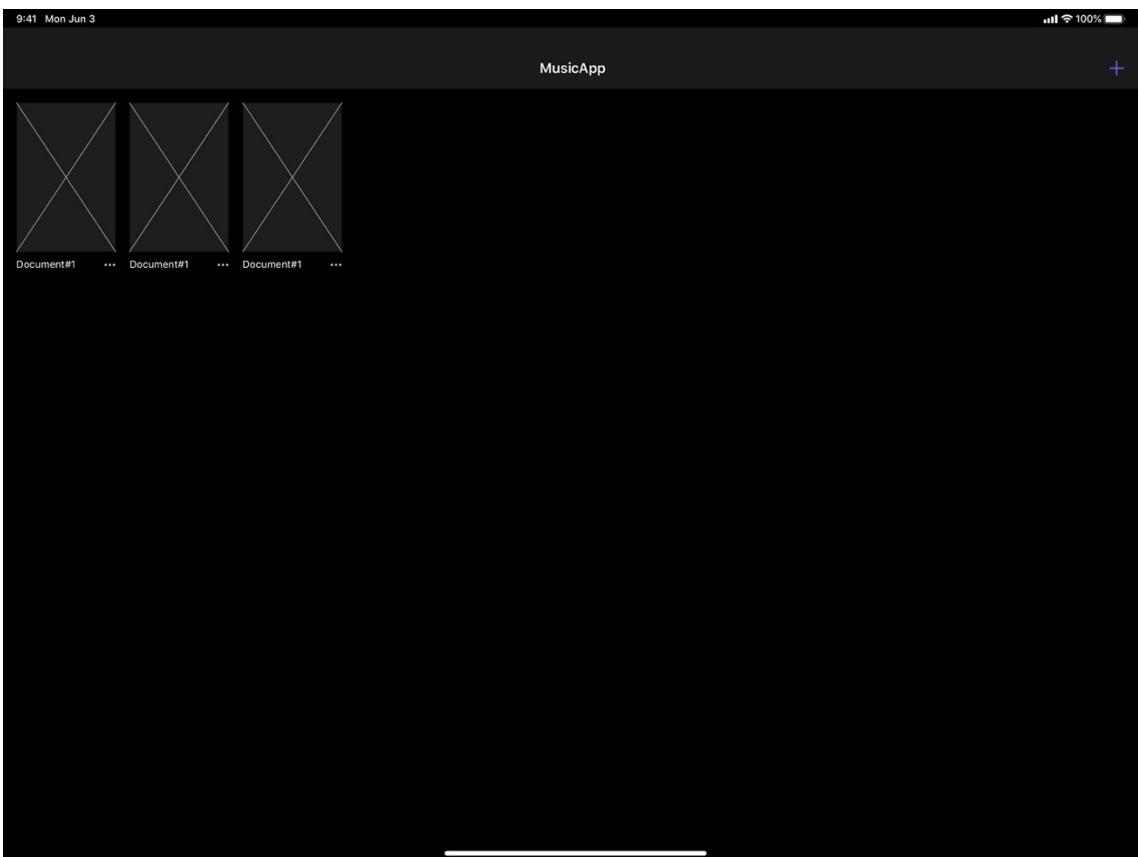


Figura 24 - Mockup collezione di spartiti

Attraverso questa interfaccia l'utente vedrà tutta la sua collezione di spartiti. Attraverso il bottone in alto a destra sarà possibile aprire un sottomenu che permetterà l'inserimento di un nuovo spartito. Cliccando su uno spartito l'utente verrà indirizzato alla vista di visualizzazione e modifica. I tre puntini a destra del nome permetteranno di gestire le varie operazioni possibili, come la rimozione o la condivisione.

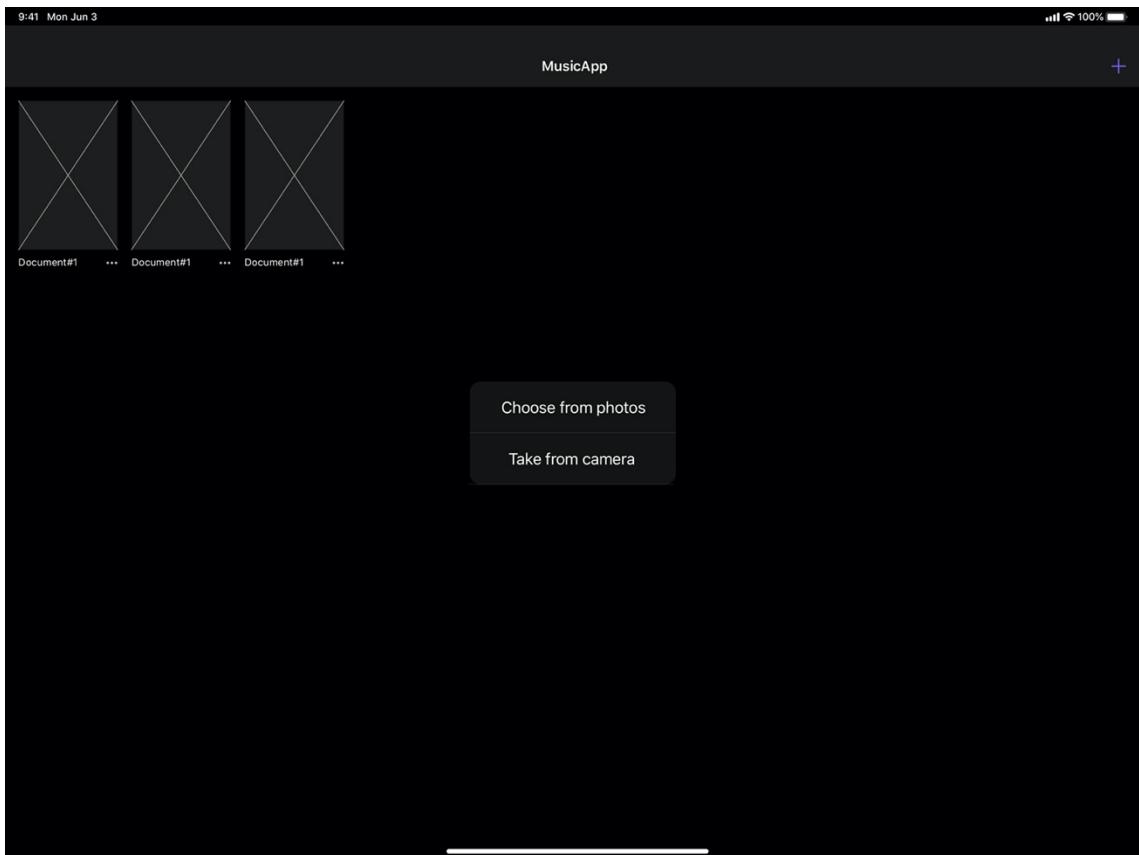


Figura 25 – Mockup popOver menu

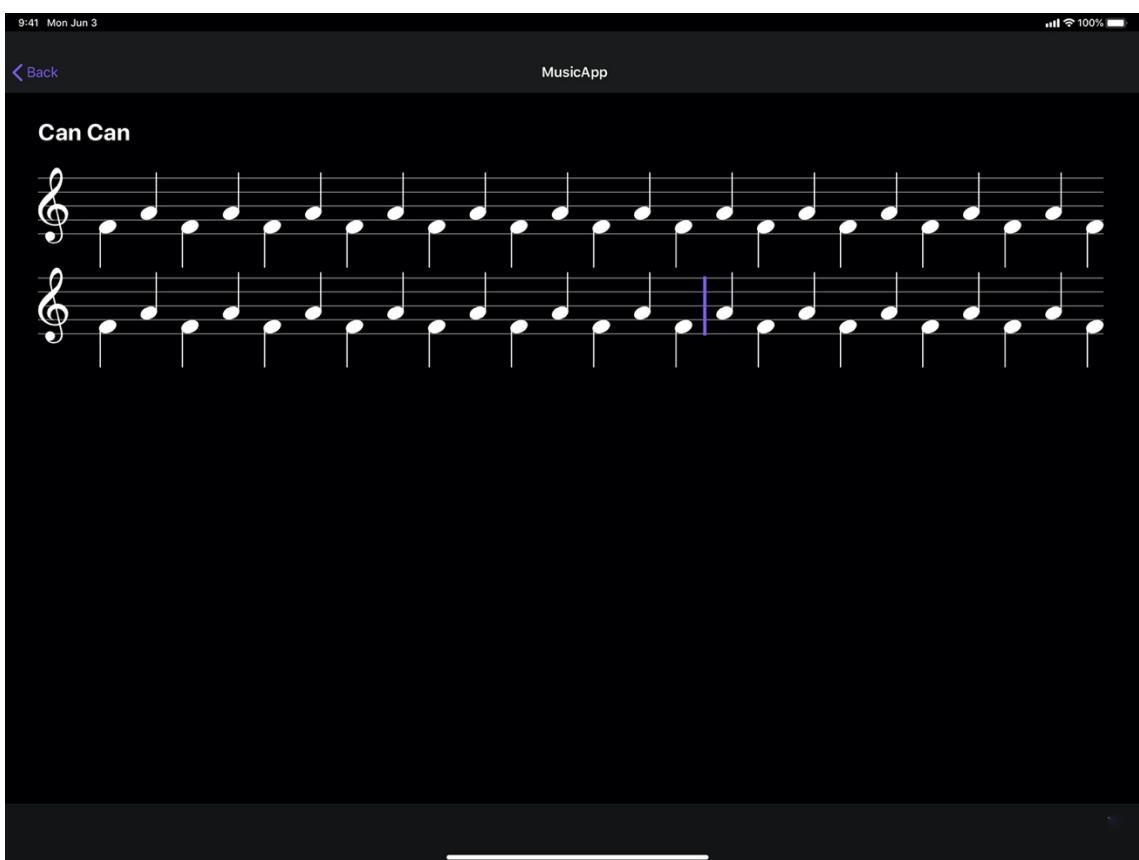


Figura 26 – Visualizzatore spartiti

4.3 Risultato

Il design adoperato non si allontana molto dai mockup iniziali però, sono state apportate delle modifiche durante l'evoluzione di alcune funzionalità. In particolare, la vista per la visualizzazione e la modifica dello spartito ha subito diversi cambiamenti, migliorando il controllo dei tool di editing. Infine, l'interfaccia è stata realizzata in modo da supportare sia la light mode che la dark mode utile non solo ai fini della durata della batteria ma anche nel caso si voglia utilizzare l'applicativo in ambienti con poca luce, ad esempio all'interno di un teatro.

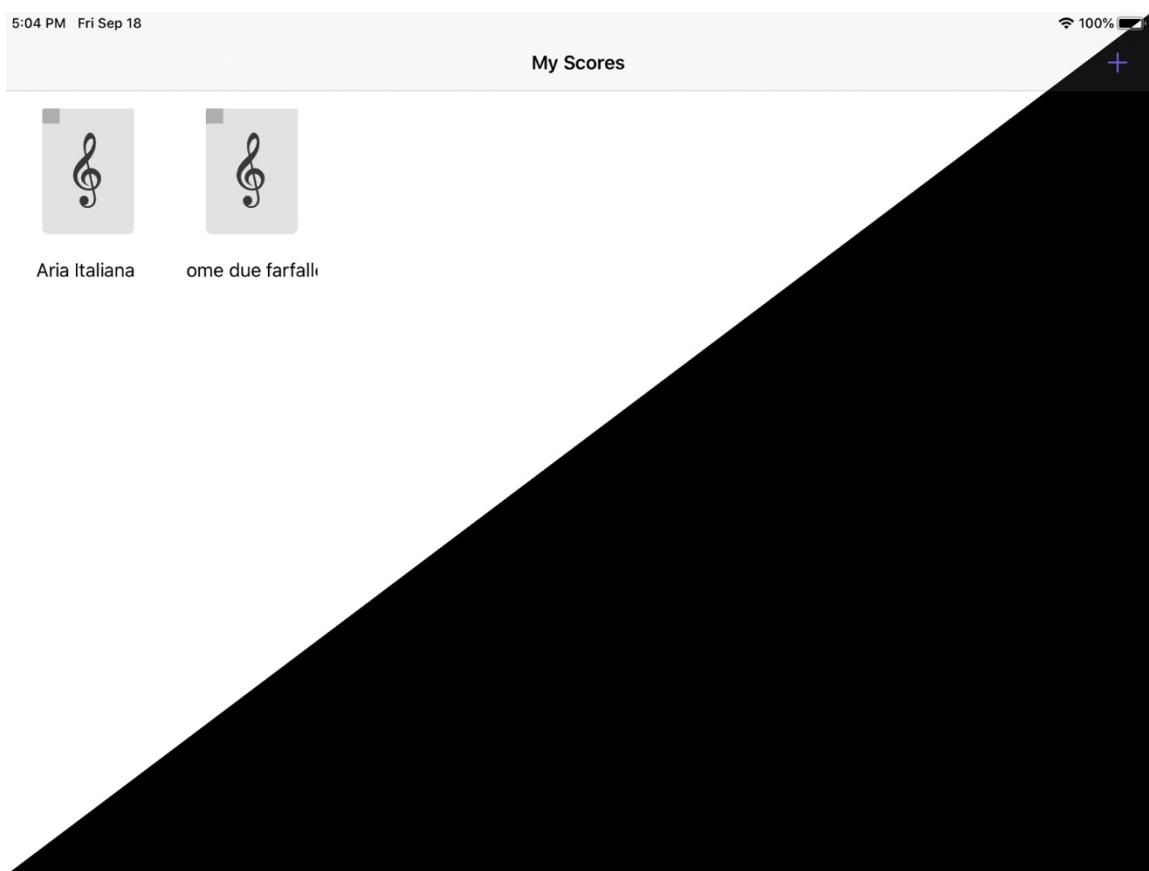


Figura 27 - Main Menu

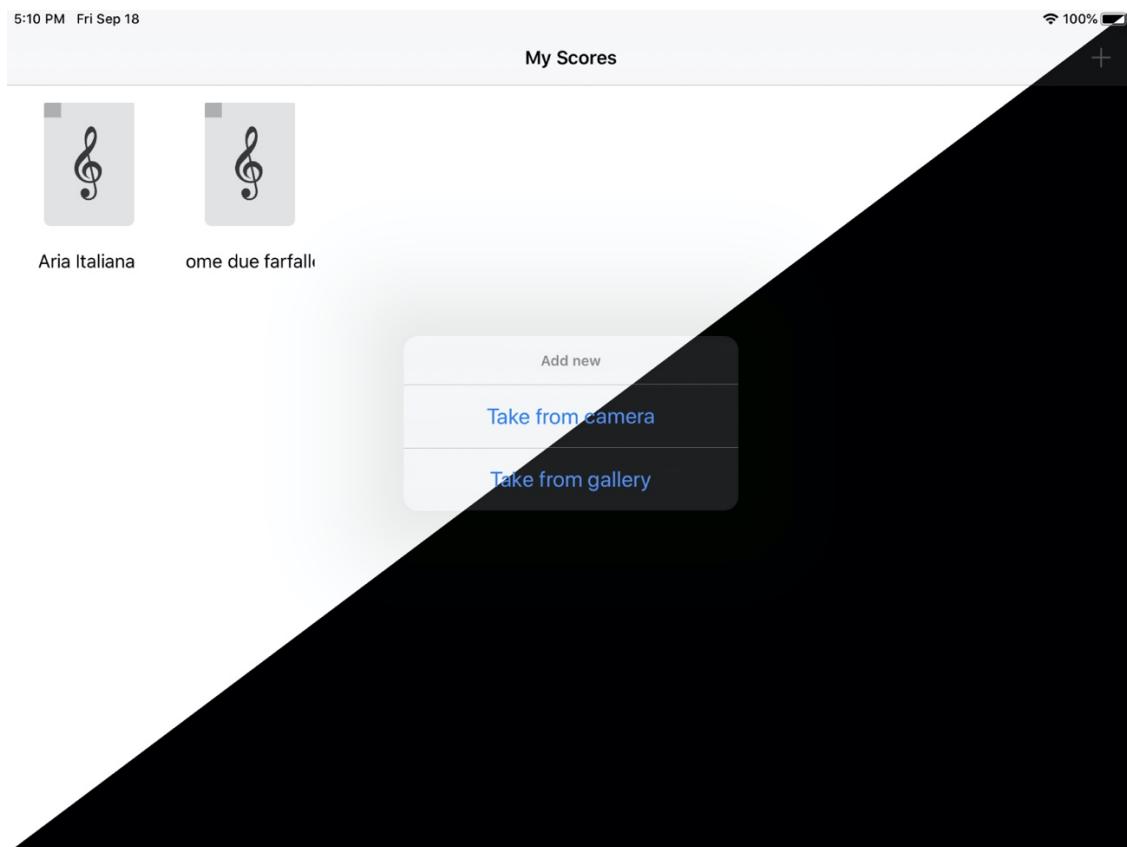


Figura 28 - Aggiungi nuovo spartito

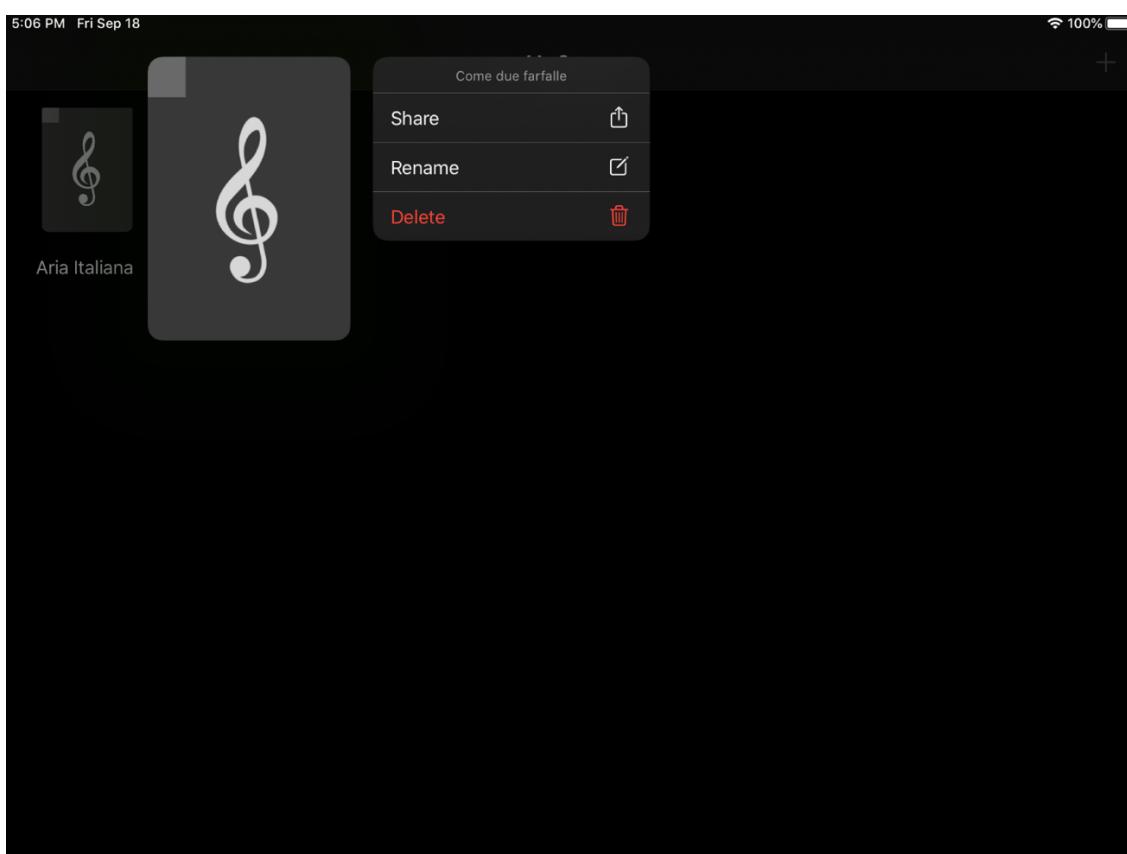


Figura 29 - Opzioni spartito

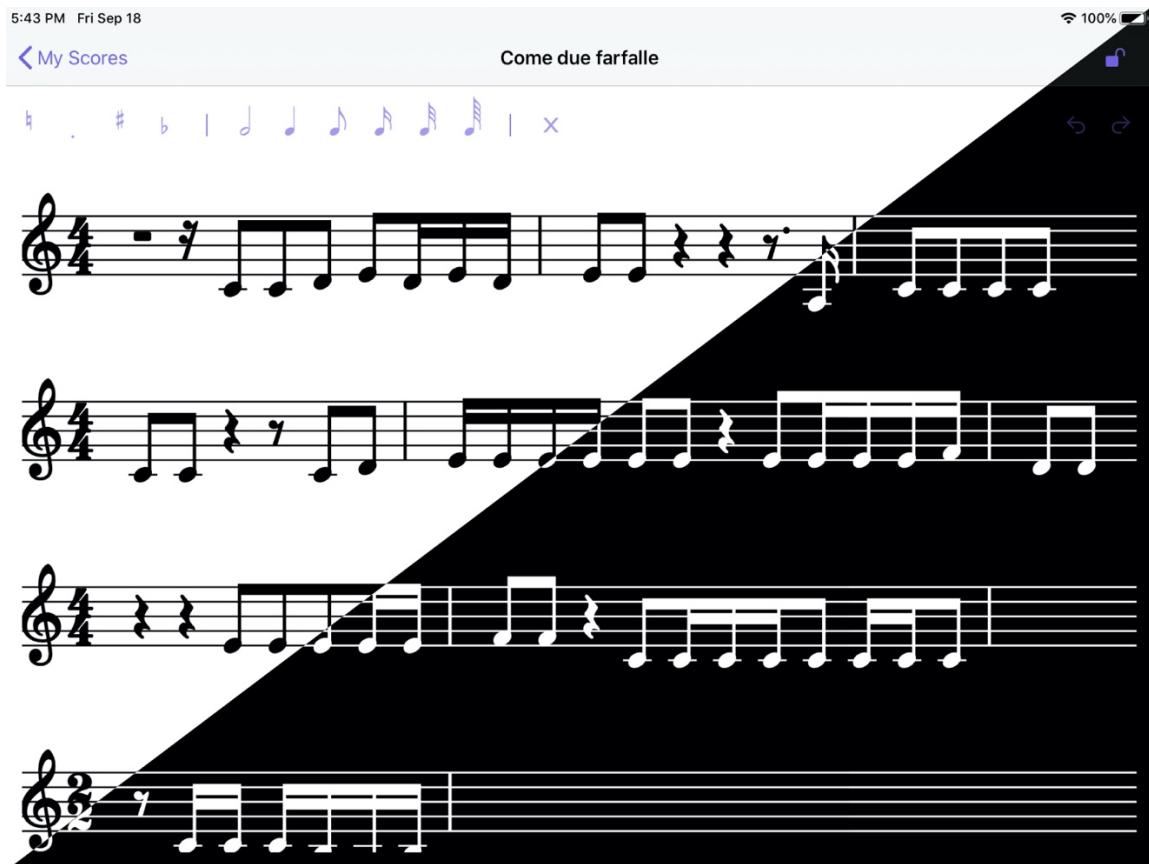


Figura 30 - Editor musicale

Capitolo 5

Implementazione di PentaKey

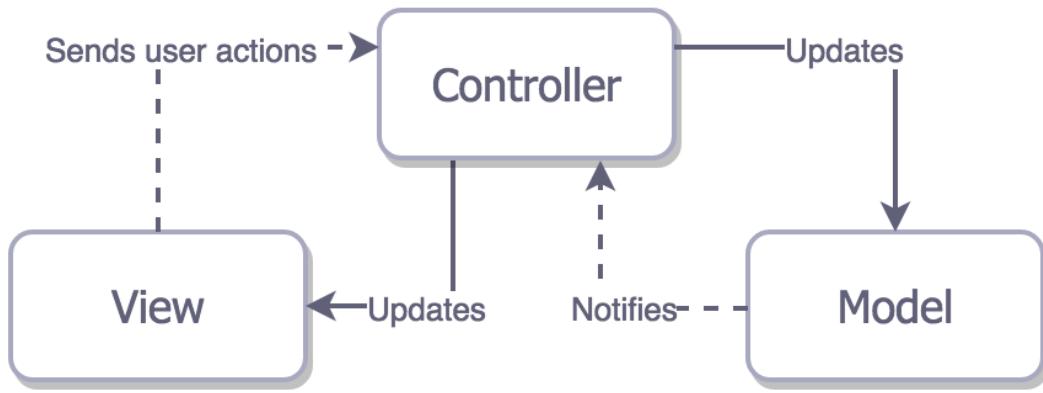
In questa sezione, verranno illustrate e motivate le scelte di progettazione che hanno contribuito alla realizzazione di questo progetto. Infatti, cominceremo illustrando come si è scelto di modellare gli oggetti che formano il nostro dominio del problema. Continueremo spiegando nel dettaglio l'algoritmo che elabora le informazioni risultanti del modello predittivo. Infine, concluderemo parlando delle funzionalità di editing, della persistenza dati e dell'iter necessario per la corretta elaborazione e visualizzazione dello spartito.

5.1 Introduzione

Prima di poter esporre gli aspetti implementativi e le varie componenti che lo compongono è doveroso introdurre il design pattern Model View Controller (MVC) e il suo utilizzo all'interno dell'applicazione.

Il Model View Controller è un design pattern strutturale, specifica che un'applicazione è costituita da un modello di dati, agli oggetti che si occupano della presentazione ed infine da oggetti di controllo.

- Il **Model** è dove risiedono tutti i dati del nostro problema. Vi possono essere anche oggetti che gestiscono la persistenza, oggetti Beans, parser e gestori.
- La **View** si occupa della presentazione dei dati del modello all'utente. Sa come accedere ai dati del modello, ma non sa cosa significano questi dati o cosa può fare l'utente per manipolarli.
- Il **Controller** si colloca tra la View e il Model. Contiene tutta la logica necessaria al fine di portare a termine una determinata operazione. Elabora e gestisce le iterazioni utente che vengono inviate dalla View.



Il pattern MVC è spesso utilizzato nel mondo iOS infatti, la stessa Apple lo utilizza e lo propone nei suoi framework.

Anche la nostra applicazione seguirà questo schema. Infatti, avremo il model layer che sarà composto dagli oggetti beans, gli oggetti che permettono di utilizzare il modello predittivo e l'oggetto che gestisce il text recognition.

Avremo un vasto insieme di view, che dovranno permettere la corretta visualizzazione dei simboli musicali su uno spartito.

Infine, avremo diversi controller che dovranno gestire tutte le iterazioni utente. Ognuno di essi, avrà il compito di gestire determinate views. Per esempio, avremo un solo controller che gestirà tutte le funzionalità di editing, un altro che gestirà le iterazioni del menu principale ed ecc.

5.2 Beans

I Beans sono classi di oggetti che modellano il dominio del problema. Nel nostro caso, modellano degli spartiti musicali.

Come già accennato nel capitolo 2, gli spartiti musicali sono formati da un determinato numero di righe musicali che a loro volta sono formati da un determinato insieme di misure. Nel nostro sistema, rappresenteremo uno spartito come se tutte le sue misure si trovasse solamente su un solo rigo poiché la quantità di righe musicali sono influenzate dalla grandezza del foglio su cui viene stampato lo spartito. Solamente l'oggetto che dovrà visualizzare lo spartito dovrà prendersi cura di questa problematica.

Ogni misura dovrà conservare oltre agli elementi musicali che lo compongono, anche la chiave musicale e il suo relativo tempo.

Infine, gli elementi musicali che si trovano all'interno di una misura saranno rappresentati come specializzazione di una classe astratta che modella solamente il concetto di un elemento musicale. Questa scelta impegnativa, ci permette di rappresentare con più semplicità una sequenza musicali e soprattutto, di modellare con più facilità gli insiemi di crome (beam).



Figura 31 - Sequenza di crome (Beam)

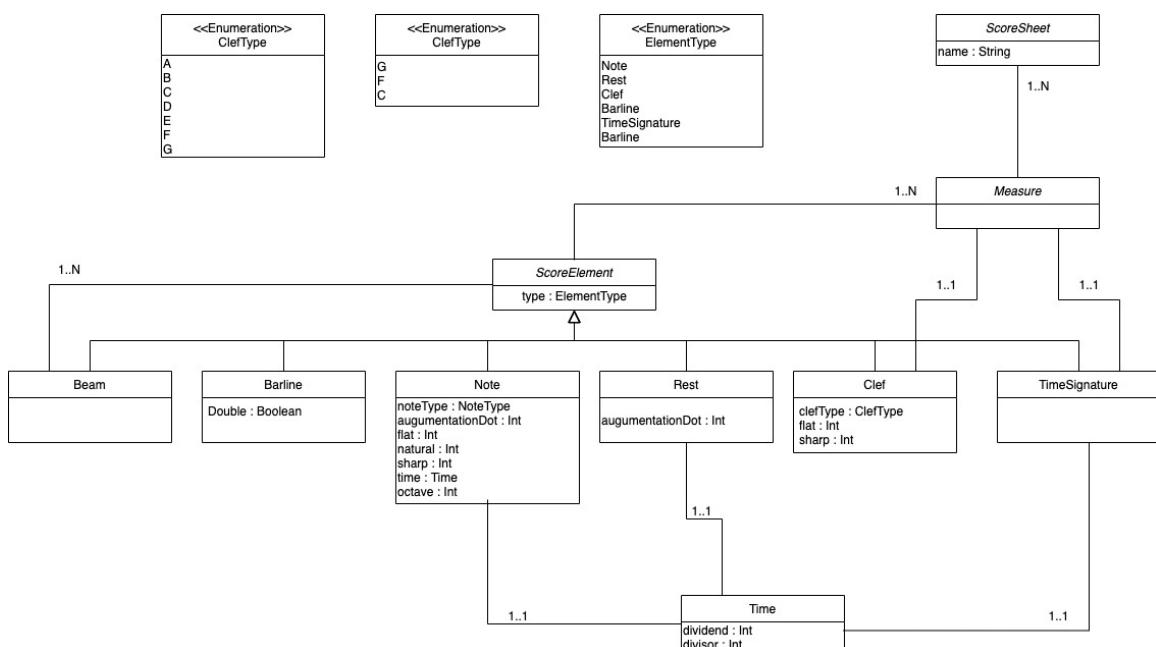


Figura 32 - Class Diagram

Come si può vedere dal diagramma, abbiamo utilizzato delle enumerazioni per rappresentare i diversi tipi di oggetti evitando di utilizzare delle stringhe per la relativa classificazione. Infine, è stata introdotta la classe “Time” che modella il tempo musicale ai fini di rappresentare con più facilità i tempi delle note, delle pause e della relativa misura.

Da notare la natura ricorsiva della classe “beam” in quanto, relazionato con la classe astratta “ScoreElement” oltre a poter contenere note o pause, potrebbe contenere anche un altro elemento della classe beam. Prendendo come esempio il beam rappresentato nella figura 25 ed utilizzando la nostra rappresentazione avremo che un oggetto beam che contiene un al suo interno 3 oggetti in sequenza: un beam che contiene una sola nota, una nota e un altro beam che contiene una sola nota.



Figura 33 - beam

5.3 Acquisizione immagini

L’acquisizione immagini è una parte sostanziale dell’applicativo in quanto, migliore è la qualità dell’immagine acquisita, migliore sarà la predizione del modello.

Per la sua migliore gestione, è stato utilizzato un modulo del framework Vision che ci permette di scannerizzare un documento fornendo una buona interfaccia grafica e migliorando la qualità dell’immagine.

Il modulo però è disponibile solo da iOS 13 e, nel caso in cui l’applicativo fosse utilizzato da un dispositivo con una versione precedente, verrà acquisita una semplice immagine senza impiegare il modulo di Vision.

Una volta acquisita l’immagine, verranno eseguite tutte le operazioni per estrazione dei dati e la loro visualizzazione.

5.4 Model Provider

La classe Model Provider è quella classe che si preoccupa di gestire le richieste al modello predittivo.

Permette con una interfaccia molto semplice di predire un’immagine in input ritornando le predizioni risultanti, nascondendo i dettagli implementativi per il corretto utilizzo del modello. Dettagli non di piccola natura poiché, utilizzando YOLO siamo impossibilitati nell’utilizzare il framework Vision in quanto non supportato.

Tutto l’iter dalla trasformazione dell’input in flusso di byte fino ad arrivare alla predizione sul modello, è stato codificato nella classe “YOLO”, che contiene la logica necessaria per predire e ricavare l’output evitando così di appesantire la classe “Model Provider” che manterrà un’interfaccia semplice.

Infine, essa è stata progettata per essere un singleton, permettendo di centralizzare le richieste al modello. Per ogni richiesta effettuata al modello, sarà restituita una sequenza di oggetti della classe Prediction.

<i>Prediction</i>
classIndex: Int score: Float rect: CGRect startPoint : CGPoint finishPoint : CGPoint labelName : String scoreIndex : Int?

Figura 34 - classe Prediction

Nella classe Prediction verranno inserite tutte le informazioni relative alla predizione come il bounding box (valore dell'attributo rect), l'indice della classe predetta (classIndex), la percentuale di confidenza (score) e il nome della classe predetta (labelName). Infine, i valori rimanenti sono stati aggiunti per la corretta elaborazione delle informazioni, “startPoint” e “finishPoint” conservano rispettivamente le coordinate del punto di inizio e di fine dell'elemento e, per concludere, scoreIndex rappresenta l'indice del rigo musicale di appartenenza (partendo dall'alto e da indice 0) inizialmente nullo poiché il valore verrà elaborato successivamente.

5.5 Parser

Le operazioni necessarie al fine di interpretare i dati forniti dal modello predittivo sono state centralizzate in una sola classe.

L'elaborazione può essere divisa in più fasi.

- I. In questa prima fase, verrà mostrata la tecnica utilizzata per ordinare gli elementi riconosciuti e verranno prese in considerazione delle limitazioni del modello predittivo.

Prima di poter elaborare gli elementi musicali, abbiamo bisogno di riconoscere la sequenza in cui si trovano sullo spartito.

Proprio per questo motivo, è stata aggiunta la classe di oggetti “SCORE” in quanto, questo ci permette di riconoscere prima la posizione del rigo musicale e

successivamente di analizzare tutti gli elementi che si trovano al suo interno, ordinarli per l'asse delle ascisse in modo da ricavare il corretto ordine di rappresentazione.

È da notare che il modello predittivo non sempre riesce ad individuare con correttezza l'intero rigo musicale in un'immagine complessa poiché, ne viene rilevata solo una parte.

Per migliorare la qualità dell'elaborazione prima di ricercare gli elementi musicali, ogni rigo verrà allungato fino al margine della foto se non interseca nessun altro rigo.

Una volta che un elemento è stato ritrovato, verrà settato il valore di "scoreIndex".

Nel caso in cui ci fossero elementi che non si trovino su nessun rigo riconosciuto verranno inseriti su uno score di default.

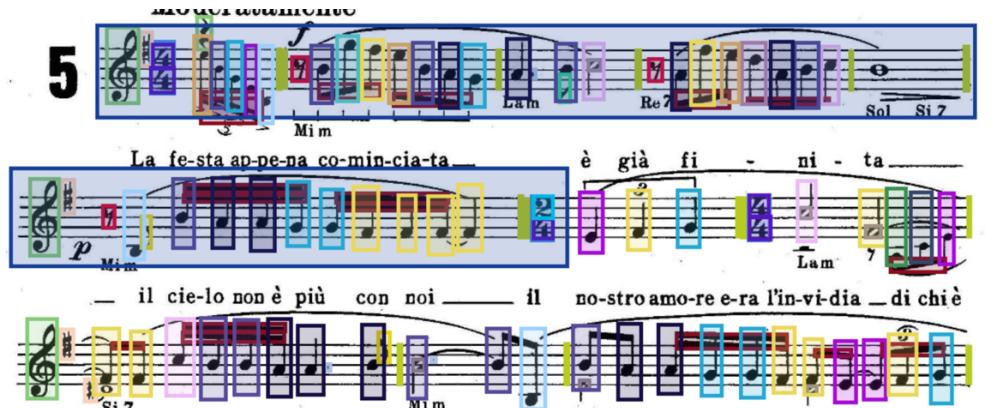


Figura 35

Prendendo come esempio di predizione la figura 27, l'algoritmo si comporterà in questo modo:

- Ricercherà prima gli elementi "SCORE" e ne troverà due.
- Allungherà fino ai margini dell'immagine **entrambi** gli elementi poiché nessuno dei due interseca un altro rigo.
- Verranno ricercati tutti gli elementi musicali per ogni rigo, ordinati secondo l'asse delle X e verrà settato il relativo valore di scoreIndex. Gli elementi che si trovano nel primo rigo avranno come scoreIndex il valore 0, invece nel secondo avranno valore 1 ed infine, gli elementi che si trovano nel terzo rigo avranno come valore -1 (valore di default) poiché non si trovano in nessun rigo musicale riconosciuto.

- II. In questa fase, verranno analizzati i problemi incontrati per la creazione di dell'algoritmo capace di rappresentare le informazioni prodotte dal modello e infine verranno esposte le considerazioni che ci hanno portato ad una soluzione accettabile.

Osservando la figura 27 notiamo che vi sono un determinato set di elementi che potrebbero essere composti da un determinato numero di sotto elementi. Per esempio, i beam possono avere al loro interno un determinato numero di note o addirittura altri beam. Anche le note possono contenere dei sotto elementi che ne modificano il tempo, come una coda oppure un cerchio vuoto. Infine, oltre alla possibilità di trovare degli elementi all'interno, non è trascurabile la possibilità di trovare degli elementi nell'intorno all'oggetto preso in considerazione che ne modificano l'interpretazione. Per esempio, elementi come bemolle e diesis, se presenti vicino ad una nota modificano la sua interpretazione.

Inizialmente, l'idea era di scannerizzare tutti gli elementi utilizzando solamente l'ascissa del punto di inizio.

Seguendo questa idea, avremmo potuto creare un algoritmo che avrebbe rappresentato con esattezza tutti gli elementi prendendo in considerazione anche gli elementi interni ed adiacenti. L'unica problematica sarebbe sorta quando avremmo incontrato un oggetto beam poiché, non conoscendo l'esatta fine dell'elemento, non sarebbe stato possibile dedurre quali sono i suoi sotto elementi. Inoltre, la problematica si ampliava per via della sua natura ricorsiva. Avremmo potuto risolvere la problematica verificando per ogni beam quali elementi andava ad intersecare in tutto il suo rigo musicale, però questa soluzione oltre alla sua complessità computazionale non è in grado di risolvere le problematiche legate alla natura ricorsiva dell'oggetto.

Prima di descrivere la soluzione, introdurremo una nuova classe di oggetti: “ParseElement” che aggiunge delle informazioni all'attuale classe di predizione (Prediction).

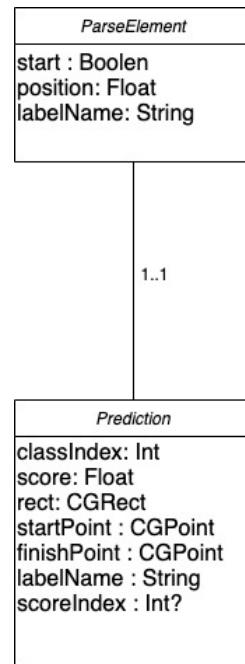


Figura 36 - Classe ParseElement

L’attributo *start* assume “true” se l’oggetto preso in considerazione indica l’inizio di un determinato elemento.

L’attributo *position* nel caso in cui *start* è vero, indicherà la il valore delle ascisse di *startPoint*, il valore delle ascisse di *finishPoint* altrimenti.

Utilizzando una sequenza di oggetti della classe *ParseElement* saremo capaci di conoscere sia l’inizio che la fine di un oggetto.

Creiamo per tutti gli oggetti almeno una occorrenza di *ParseElement* che indicherà l’inizio di un oggetto invece, per gli oggetti di cui abbiamo bisogno di conoscere la loro fine, creeremo una seconda occorrenza.

Tutte le occorrenze create verranno raggruppate in una sequenza ordinata in modo crescente secondo i parametri “*scoreIndex*” e “*position*”.

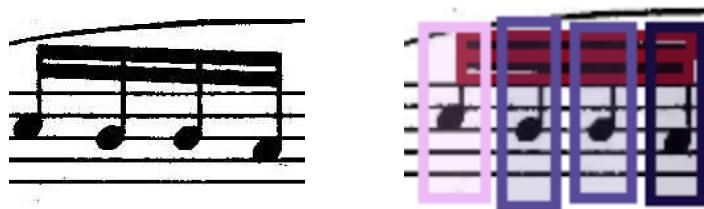


Figura 37

Considerando la figura 29 avremo la seguente sequenza:

“*nota1_inizio*, *beam1_inizio*, *beam2_inizio*, *nota1_fine*, *nota2_inizio*,

nota2_fine, nota3_inizio, nota3_fine, nota4_inizio, beam1_fine, beam2_fine, nota4_fine”.

Come possiamo vedere, attraverso questa sequenza siamo capaci di riconoscere gli elementi che si trovano dentro un altro elemento e tramite essa è possibile creare un algoritmo che scandisce una sola volta l’intera lista.

- III. In questa fase verrà analizzata la sequenza prodotta dalla fase due al fine di creare una sequenza ordinata di oggetti che estendono la classe astratta “ScoreElement”. Per migliorare la manutenibilità del codice si è scelto di utilizzare una logica decentralizzata in quanto, ogni elemento riconosciuto dal modello ha caratteristiche molto differenti dagli altri.

In particolare, avremo per ogni oggetto da rappresentare un handler che provvederà alla sua corretta rappresentazione. Non avremo un handler per ogni oggetto riconosciuto dal modello in quanto, non tutti gli elementi possiedono un’identità all’interno dello spartito. Per esempio, oggetti come diesis e bemolle presi singolarmente non hanno un significato preciso, cosa diversa se vengono considerati con una nota o ad una chiave. Questi elementi devono essere considerati come attributi di altri oggetti, e come tali saranno presi in considerazione dagli handler che gestiscono la rappresentazione di elementi che li possiedono.

Infine, avremo un solo dispatcher che provvederà a richiamare il coretto handler per la gestione dell’oggetto preso in considerazione.

Una volta che il dispatcher ha scandito tutta la sequenza della fase 2, avremo come risultato una sequenza di ScoreElement che rappresenta tutti gli oggetti riconosciuti dallo spartito.

- IV. In questa ultima fase, trasformeremo la sequenza di ScoreElement prodotta dalla fase 3 in un oggetto ScoreSheet.

Sapendo che lo spartito può essere visto come una sequenza di misure, ci basterà individuarle all’interno della sequenza di input.

Ogni misura, è delimitata da una barra nera che nel nostro sistema viene rappresentata da un oggetto della classe Barline. Ci basterà trovare un’occorrenza di quest’ultima nella sequenza di input per dedurre l’inizio della prossima misura. Trasformata la sequenza di ScoreElement in una sequenza di Measure, abbiamo

tutto il necessario per creare un oggetto di ScoreSheet.

Per concludere, passiamo ad analizzare per ogni fase la complessità computazionale dell'algoritmo.

Nella prima fase, ricerchiamo tutti gli elementi “SCORE” che impiega $O(N)$, dove N è la lunghezza della sequenza e, successivamente computiamo per ogni elemento il valore di `scoreIndex` che impiega $O(N \times S)$, dove S è il numero di righi musicali riconosciuti.

Nella seconda fase, creiamo per ogni elemento della sequenza di input al più due occorrenze della classe `ParseElement`, quindi avremo un tempo in $O(N)$.

Nella terza fase, creiamo la sequenza di `ScoreElement` che leggendo una sola volta la sequenza di `ParseElement` impiega $O(N)$.

Nella quarta fase, scomponiamo la sequenza di `ScoreElement` in misure impiegando $O(N)$.

Tirando le somme, per tutto l'algoritmo avremo un tempo in $O(N \times S)$ dove N è la lunghezza della sequenza e S il numero di righi musicali riconosciuti.

Ricordiamo che S è un numero relativamente basso in quanto la maggior parte degli spartiti non contiene più 10 righi musicali.

5.6 Text Recognition

Il modello predittivo, è addestrato non solo al riconoscimento dello spartito in sé ma, anche al riconoscimento del titolo dello spartito. Precisamente, il modello ci fornisce la regione dell'immagine dov'è presente il titolo dello spartito.

Utilizzando delle tecniche di text detection e text recognition fornite dal framework Vision siamo in grado di ricavare il relativo testo.

Per inviare una richiesta al modello di text recognition già implementato in Vision, abbiamo bisogno di creare una richiesta.

Una richiesta non è altro che un contenitore di parametri, tra questi vi sono:

- **recognitionLevel**: Può assumere solo due valori: `fast` e `accurate`; `Fast` permetterà di avere una risposta alla richiesta effettuata in tempi brevi, utile se si vuole creare un'applicazione real-time ma poco robusto nel caso si necessiti di precisione.

- **usesLanguageCorrection:** È un valore booleano. Se settato a falso non verrà applicato nessun controllo ortografico post-process, altrimenti verranno applicati utilizzando un dizionario di parole contenuto in Vision, andando a pesare sulle prestazioni.
- **regionOfInterest:** È un rettangolo che definisce la regione di interesse (ROI) dato che il text recognition non è una procedura con basso throughput, settare una regione di interesse eviterebbe di scansionare l'intera immagine migliorando le prestazioni.

Per le nostre esigenze, utilizzeremo la versione accurata e il controllo ortografico. Invece, per la regione di interesse utilizzeremo quando possibile la predizione che ci fornisce il nostro modello predittivo. Nel caso in cui non avessimo una predizione dal modello dato, utilizzeremo una ROI di default che comprende solamente il 25% della parte alta dell'immagine ed a quel punto Vision si occuperà di trovare il testo presente in quella porzione e riconoscerlo.

5.7 Score Drawer

In questo paragrafo parleremo di come abbiamo costruito il visualizzatore di spartiti. Inizieremo parlando della logica utilizzata per il posizionamento degli elementi e la tecnica utilizzata per la loro creazione. Infine, concluderemo spiegando nel dettaglio alcune delle view utilizzate per la visualizzazione degli elementi.

Tutta la logica legata al posizionamento e la visualizzazione dei simboli musicali è centralizzata in una sola classe “ScoreDraw” che provvederà alla creazione dello spartito prendendo in considerazione i margini del dispositivo e lo spazio richiesto dai simboli.

La costruzione del pentagramma non è molto complessa, in quanto è formato da 5 linee parallele, facilmente disegnabili attraverso le funzioni fornite dal framework UIKit.

Una volta costruito il pentagramma lo divideremo in settori, ovvero la regione di spartito dove sarà possibile posizionale una view che visualizzerà un determinato elemento. Infine, utilizzeremo un set di spaziature per evitare che gli elementi si sovrappongano.

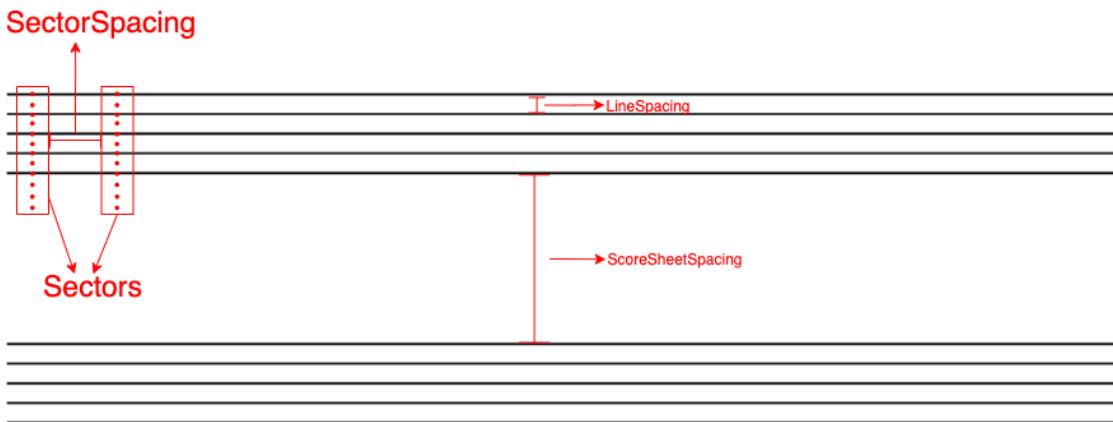


Figura 38 - Sistema di posizionamento

Prima di inserire un simbolo, viene calcolata la sua grandezza in modo tale che se nel caso non ci fosse abbastanza spazio, il visualizzatore creerà un nuovo rigo per posizionarlo.

La gestione e visualizzazione dello spartito non è una procedura leggera infatti, è importante limitare il più possibile il suo ricalcolo. Per migliorare le prestazioni viene elaborata in parallelo evitando di occupare per troppo tempo le risorse. Utilizzando l'interfaccia esposta dalla classe ScoreDraw è possibile richiedere il ricalcolo della procedura inviando con essa una callback che verrà richiamata a procedura terminata.

Per le view, abbiamo mantenuto la struttura gerarchica dei Beam mantenendo la classe astratta, in modo tale da poter gestire meglio una loro sequenza.

Quindi per ogni Beam avremo la sua relativa view che conterrà al suo interno tutta la logica necessaria per la sua corretta visualizzazione.

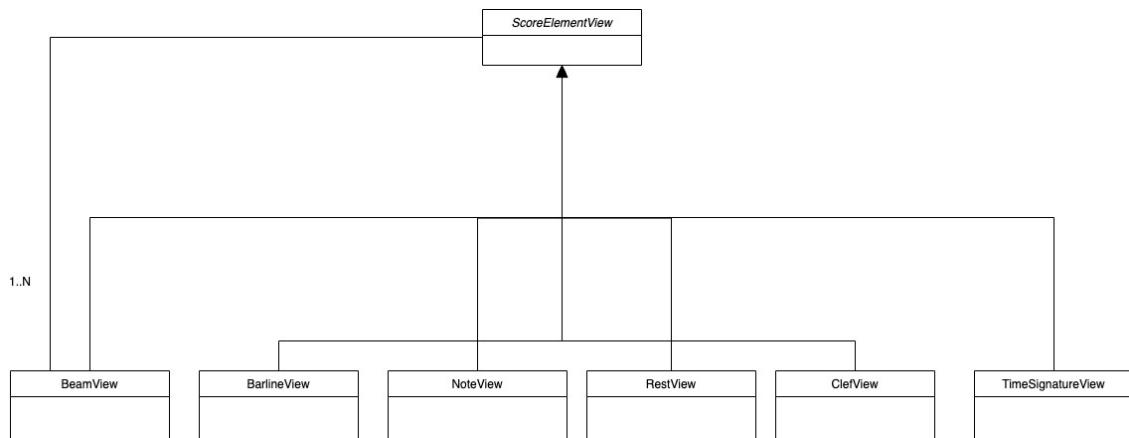


Figura 39 - Diagramma Views

Solitamente, per la scrittura di uno spartito musicale digitale vengono adoperati dei font creati ad-hoc che fanno parte della categoria dei SMuFL.

SMuFL è una specifica che fornisce un modo standard per mappare le migliaia di simboli musicali richiesti dalla notazione musicale convenzionale.

La maggior parte dei caratteri contenenti simboli musicali utilizza un layout glifo e sono presenti nello standard Unicode.

Utilizzeremo il font “Bravura” per la creazione delle view necessarie alla visualizzazione, evitando di dover disegnare da zero tutti i simboli necessari, risparmiando molto effort.

Per utilizzare i glifi forniti dal font, è possibile utilizzare delle funzioni fornite dal framework “UIKit” prodotto da Apple, dove vi è contenuto tutto il necessario per costruire una custom view, gestire le iterazioni con l’utente ed il supporto al disegno.

Una volta estratto il glifo, verrà collocato all’interno di un layer. Le view utilizzate per la rappresentazione dei Beans utilizzano uno o più layer in base ai valori degli attributi.

Le view create per le note e i beam sono tra le più complesse, in quanto gli oggetti in sé contengono attributi che alla loro variazione potrebbero modificare completamente la loro visualizzazione. Per esempio, al cambiare del tempo di una nota non altera solo la nota in sé ma, vi si potrebbero aggiungere anche degli altri oggetti. Queste caratteristiche rendono più complessa la parte di posizionamento dei layer. Inoltre, la view dedicata alle note presenta un’ulteriore logica che gli permette, in base alla nota presa in considerazione, di capire dove deve essere posizionata. Come si può vedere dalla figura 30 all’interno dei settori vi sono presenti dei punti, essi vengono calcolati moltiplicando il valore che assume “LineSpacing”, ogni punto calcolato rappresenta una possibile posizione della nota.

Infine, la view che gestisce gli oggetti beam risulta abbastanza complessa in quanto oltre a dover essere capace di gestire la sua natura ricorsiva, deve considerare tutti gli attributi dei suoi sotto elementi prima di poter essere disegnata.

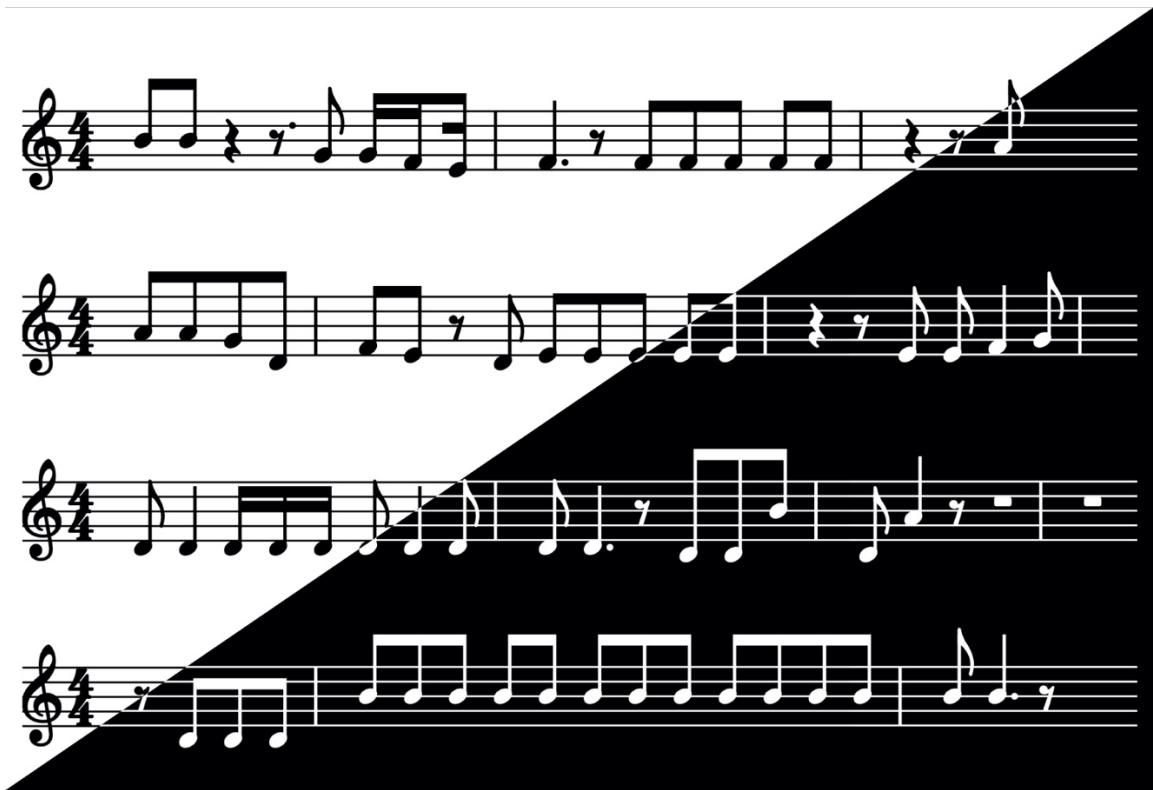


Figura 40 – Risultato visualizzazione spartiti

5.8 Editor

Le funzionalità di editing sono state centralizzate all'interno di un solo controller, che gestisce sia la visualizzazione degli spartiti che la loro modifica.

Ricordiamo che uno spartito musicale all'interno del nostro sistema non è altro che un oggetto che contiene una sequenza di misure. Per modificare uno spartito ci basterà modificare i valori che formano la sequenza. Sono state sviluppate 3 funzionalità principali: modifica, rimozione e aggiunta.

Le funzionalità sono state create implementando un protocollo da noi ideato che permette di costruire una nuova funzionalità senza dovere riscrivere nessun blocco di codice.

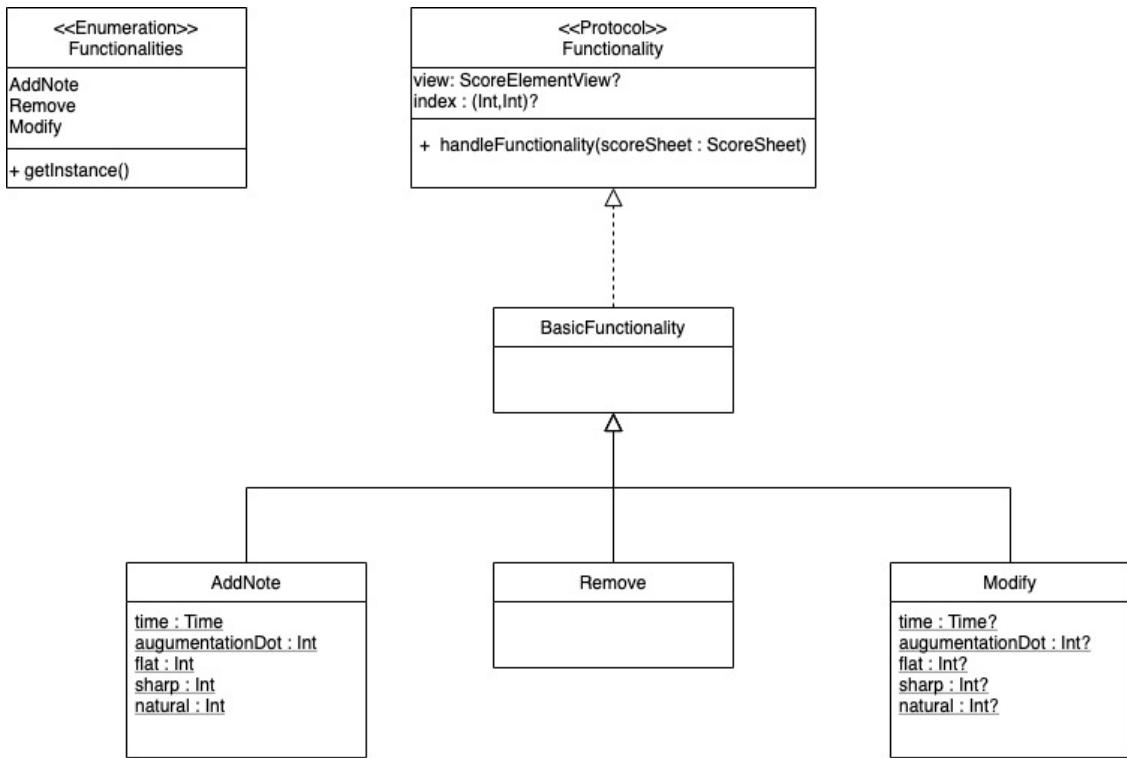


Figura 41 - Diagramma funzionalità

Avremo anche potuto evitare di creare la classe “BasicFunctionality” in quanto, la sua unica funzione è implementare il protocollo “Functionality”. Però, con questa scelta eviteremo di scrivere codice aggiuntivo durante la creazione di una funzionalità, semplicemente estendendo la classe.

Dato che le funzionalità non hanno attributi che necessitano di variare tra le varie istanze, verrà creato solo un riferimento per ogni funzionalità e solo tramite la funzione “getInstance” dell’enumerazione “Functionalities” sarà possibile referenziarla.

Inoltre, questo tipo di architettura permette di avere maggior libertà sull’aggiunta degli attributi necessari alla funzionalità poiché, non sempre funzionalità diverse necessitano degli stessi attributi.

Il controller avrà come attributo una variabile di tipo Functionalities che assumerà un determinato valore in base alla funzionalità attiva. Nel caso in cui non ci fosse nessuna funzionalità attiva verrà settato il valore nullo.

Ogni qualvolta che lo spartito viene toccato, il controller verificherà il valore della variabile di tipo Functionalities, solamente nel caso in cui non fosse nullo inserirà come valore in index la posizione dell’elemento toccato all’interno dello spartito ed in view inserirà la view toccata. Una volta fatto ciò, chiamerà la funzione “handleFunctionality” passando l’oggetto che contiene lo spartito musicale.

Oltre a queste 3 funzionalità è, stata implementata anche l’operazione indietro e avanti, che permettono all’utente di annullare un’operazione o di ripristinarla.

Queste operazioni sono state implementate con una struttura che contiene come attributi due variabili di tipo funzione. Una funzione per tornare indietro e una per ripristinare l’operazione. Per la loro corretta gestione gli oggetti della struttura saranno organizzati in una classe che li gestirà con uno stack con dimensione massima variabile (15 di default). Infine, verrà utilizzato un protocollo che sarà implementato dalla classe “BasicFunctionalità” provvedendo a fornire le funzioni di avanti ed indietro.

Ogni qualvolta che si utilizzerà una funzionalità, la classe che la implementa costruirà uno oggetto con una funzione che agisce sullo spartito e una che lo ripristina allo stato precedente. Dopodiché, il controller provvederà ad inserire l’oggetto creato nello stack ed eseguirà la funzione di avanti per eseguire l’operazione richiesta.

La soluzione utilizzata è sicuramente macchinosa e non immediata, però permette di evitare di salvare lo stato dello spartito ogni volta che esso viene modificato, evitando ulteriori consumi di memoria ed eliminando la necessità di copiare ogni volta lo spartito.

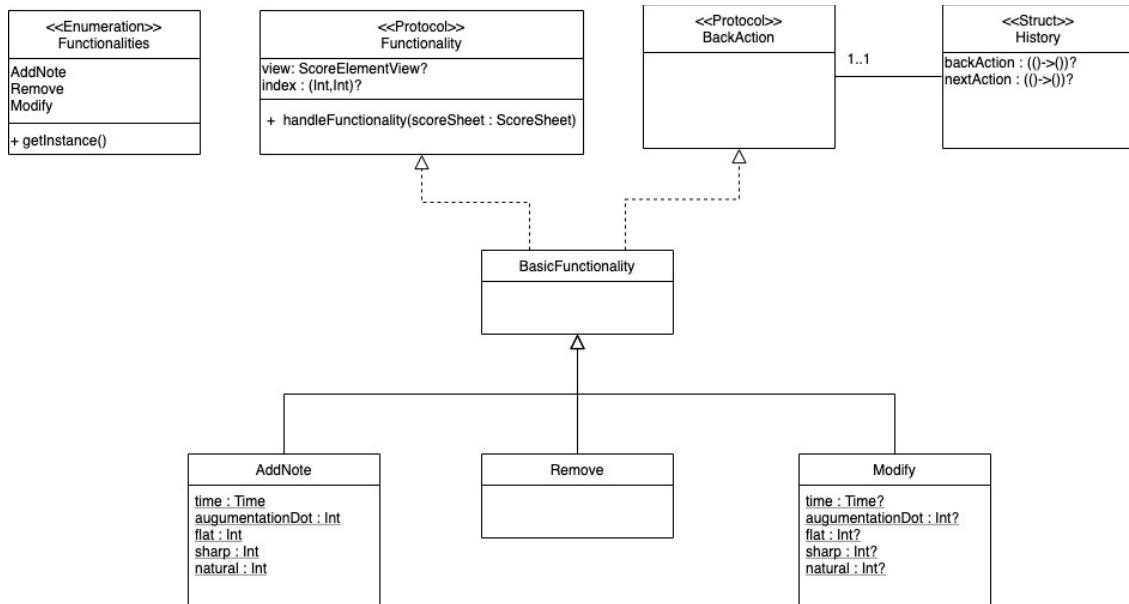


Figura 42 - Schema funzionalità completo

5.9 Persistenza dati

Per la gestione dei dati persistenti verrà utilizzato il framework Core Data.

Invece di mappare i nostri dati in uno schema entità-relazione utilizzeremo una tecnica

diversa che ci permetterà di implementare la persistenza con poche righe di codice. Prima di poter illustrare questa tecnica è importante capire come vengono definite le entità in Core Data. È possibile definirle utilizzando la GUI fornita da Xcode, ogni entità è composta da un elenco di attributi tipizzati. Tra i tipi che è possibile utilizzare in Core Data vi è “Trasformable”, che permette di conservare dei tipi non standard. Nel nostro caso l’entità che modella uno spartito musicale sarà composta da un attributo di tipo stringa che rappresenterà il nome e da un attributo di tipo trasformabile che rappresenterà la sequenza di misure che lo compongono.

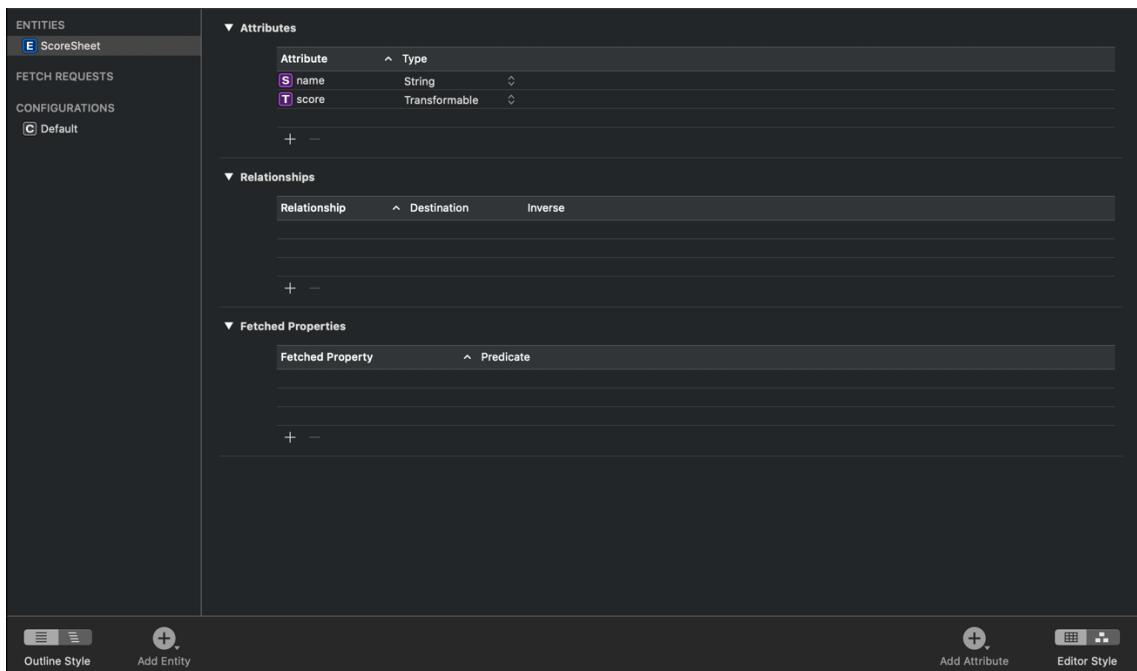


Figura 43 - Modello Core Data

Per completare la persistenza dati, tutti gli attributi della classe e la loro composizione che modellano una misura di uno spartito dovranno implementare il protocollo “NSSecureCoding” che provvederà a fornire a Core Data le funzioni di encode e decode per la corretta e sicura memorizzazione dei dati.

Infine, per la gestione del modello di dati persistente avremo bisogno di una classe che si prenderà cura di gestire il suo ciclo di vita, dal suo caricamento in fase di avvio dell’applicativo fino alla sua chiusura.

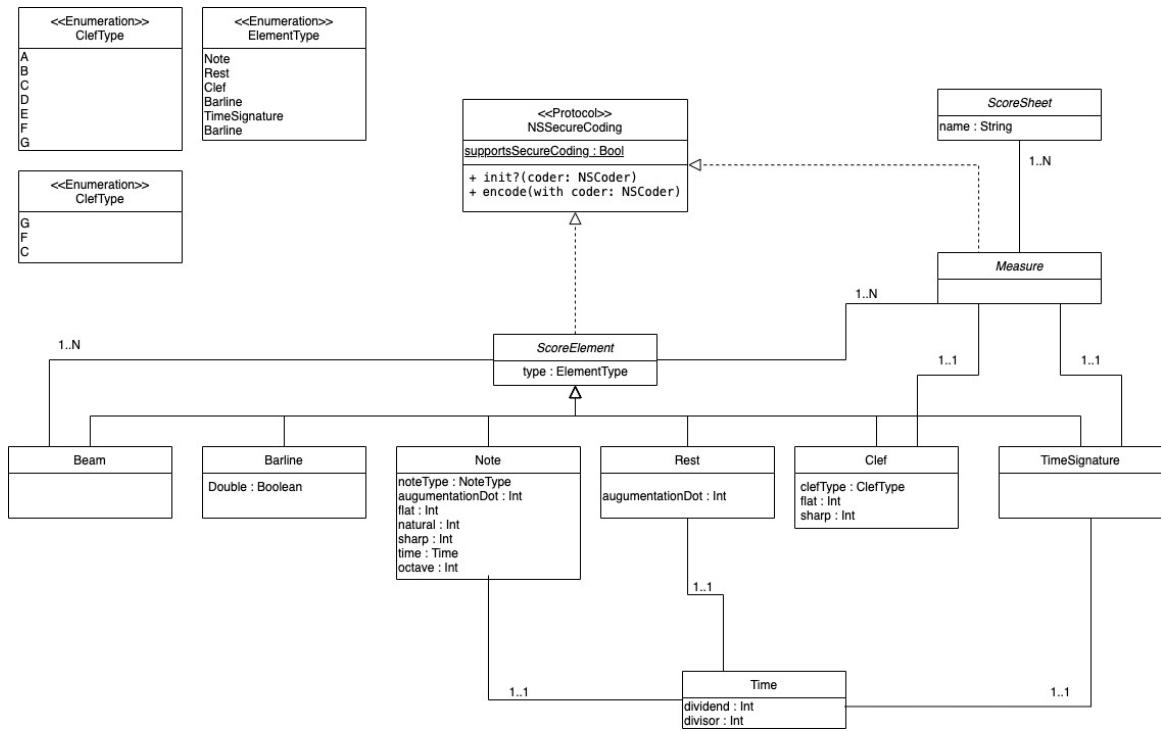


Figura 44 - Class Diagramm + Core Data

Capitolo 6

Lavori correlati

In questa sezione, verranno analizzati alcuni progetti correlati agli argomenti trattati all'interno di questa tesi. Partiremo introducendo l'optical music recognition (OMR), fino ad arrivare alla sua evoluzione e le sue varie implementazioni.

6.1 Optical Music Recognition

Optical Music Recognition è un campo di ricerca che indaga su come leggere computazionalmente la notazione musicale [10].

L'obiettivo di OMR è insegnare ad una macchina a leggere ed interpretare gli spartiti producendo una versione a lei comprensibile. Una volta acquisita digitalmente, lo spartito può essere salvato nei formati di file comunemente utilizzati.

Il processo di riconoscimento degli spartiti musicali è in genere suddiviso in passaggi più piccoli che vengono gestiti con algoritmi di riconoscimento di pattern specializzati.

Sono stati proposti molti approcci concorrenti con la maggior parte di essi che condivide un'architettura di pipeline, in cui ogni fase di questa pipeline esegue una determinata operazione, come rilevare e rimuovere le linee del personale prima di passare alla fase successiva. Un problema comune con questo approccio è che gli errori e gli artefatti che sono stati effettuati in una fase vengono propagati attraverso il sistema e possono influire pesantemente sulle prestazioni. Ad esempio, se la fase di rilevamento della linea del rigo non riesce a identificare correttamente l'esistenza del rigo musicale, i passaggi successivi probabilmente ignoreranno quella regione dell'immagine, portando a informazioni mancanti nell'output.

Un punto di partenza per qualsiasi ricerca OMR è la ricerca di Ana Rebelo [11], che contiene un completa introduzione ai sistemi OMR e una descrizione attuale dello stato del campo. Il documento descrive quattro principalifasi necessarie per qualsiasi pipeline OMR: pre-elaborazione, riconoscimento di simboli musicali, ricostruzione della composizione e rappresentazione finale. Il secondo componente, come suggerisce il nome, è dove viene svolto il lavoro di riconoscimento principale. Rilevamento e

rimuovendo le linee del rigo, segmentare i singoli simboli e classificare i simboli. [12] Non tutti i metodi seguono questo modello, infatti vi possono esistere delle implementazioni differenti. Per esempio, vi sono algoritmi che evitano la rimozione delle linee del rigo musicale, riconoscendo i simboli direttamente.

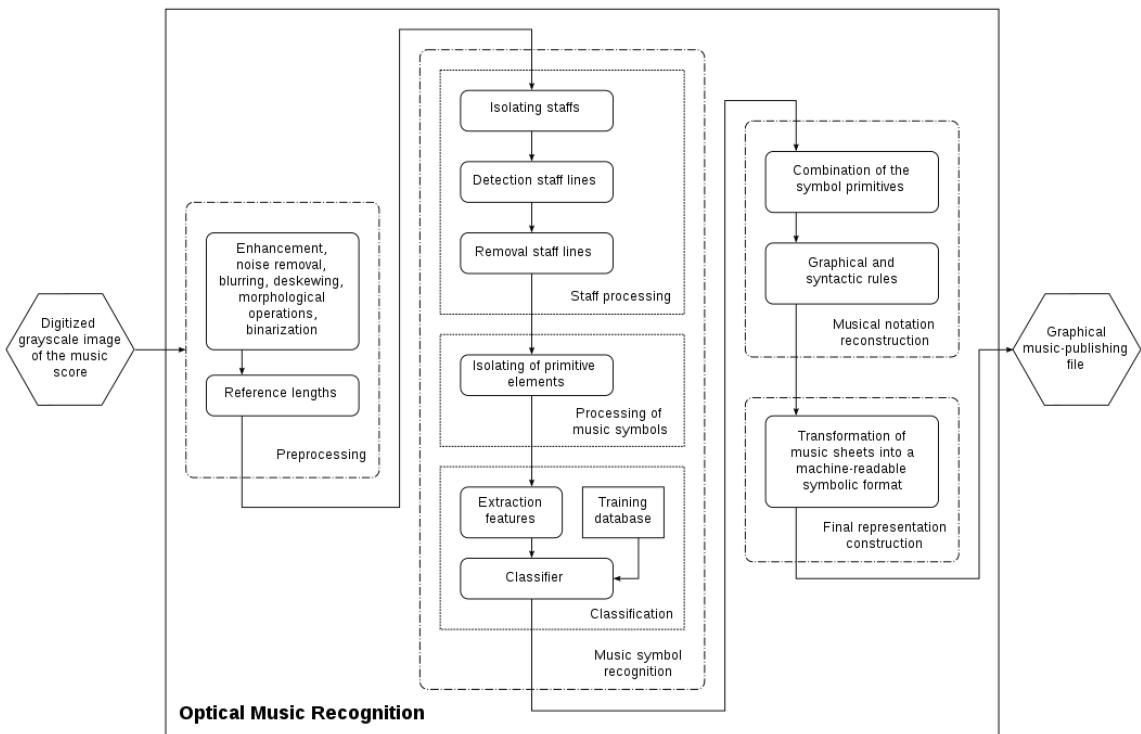


Figura 45 - Schema generale per l'OMR di Ana Rebelo

Con l'avvento del deep learning, molti problemi sono passati dalla programmazione imperativa con euristiche artigianali e ingegneria delle funzionalità all'apprendimento automatico. Nel riconoscimento ottico della musica, la fase di elaborazione del pentagramma [13], la fase di rilevamento di oggetti musicali [14], così come la fase di ricostruzione della notazione musicale [15] hanno visto tentativi riusciti con l'impiego di tecniche di deep learning.

Molti progetti open source per l'OMR sono stati realizzati, ma solo pochi di essi hanno raggiunto uno stato maturo e sono stati distribuiti con successo agli utenti.

La maggior parte delle applicazioni desktop commerciali sviluppate negli ultimi 20 anni sono state chiuse a causa della mancanza di successo commerciale, lasciando solo pochi fornitori che stanno ancora sviluppando, mantenendo e vendendo prodotti OMR. Oltre alle applicazioni desktop, è emersa anche una serie di applicazioni mobili sia per

dispositivi Android sia per dispositvi Apple che, purtroppo, anche per queste il loro sviluppo è stato interrotto. Vi sono ancora presenti alcune applicazioni che permettono di scannerizzare uno spartito musicale, salvarlo e riprodurlo.

Capitolo 7

Conclusioni

Essendo giunti al termine della stesura di questo documento, è doveroso dedicare qualche riga per riflettere sul lavoro svolto.

In questo lavoro di tesi è stato documentato ciò che è stato necessario realizzare al fine di estrapolare, visualizzare ed editare correttamente le informazioni di uno spartito, fornendo uno strumento portatile, versatile e completo per tutti gli scrittori musicali.

In particolare, è stata proposta una soluzione al problema descritto della interpretazione degli spartiti musicali, utilizzando delle tecniche di deep learning per il riconoscimento della notazione musicale, illustrando nel dettaglio tutto il processo di creazione fino al deployment, attraverso questa procedura siamo riusciti ad avere un ottimo risultato in quanto, abbiamo ottenuto ben il 95% di mAP (mean Average Precision, popolare metrica per la misurazione dell'accuratezza dei modelli di Object detection).

Inoltre, per facilitare il suo utilizzo è stata creata un'interfaccia utente che permettesse di interfacciarsi con le funzioni offerte con semplicità, attraverso l'utilizzo delle views fornite da Apple.

Infine, abbiamo completato il progetto aggiungendo delle funzionalità di editing, fornendo all'utente finale non solo la possibilità di raccogliere i suoi spartiti in un'unica locazione ma, anche la possibilità di modificare ed aggiungere simboli musicali.

7.1 Possibili estensioni future

La conclusione di questo documento non sta ad indicare che il progetto in questione sia giunto al termine oppure che non possa essere esteso in futuro.

In questa sezione, verranno analizzate alcune possibili estensioni che potranno essere aggiunte al progetto al fine di ampliare i casi d'uso e permettere al modello predittivo di aver una conoscenza maggiore.

Partendo dal modello predittivo, l'espansione del dataset con cui è addestrato gli fornirebbe una maggiore precisione e affidabilità. Inoltre, aggiungendo altri simboli

musicali renderebbe l'applicativo più completo.

Nello suo stato attuale l'applicazione riconosce solo sintatticamente i valori estrapolati dallo spartito, aggiungendo un modulo semantico migliorerebbe non solo la qualità dell'applicativo ma anche la facilità di scrittura e modifica di uno spartito.

Inoltre, dato che si parla di una applicazione nel campo musicale, le sue possibili funzionalità sono innumerevoli. Tra queste vi sono:

- Aggiungere la possibilità di riprodurre la musica composta, fornendo la possibilità di scegliere tra un set di strumenti.
- Comporre uno spartito utilizzando come input uno strumento virtuale.
- Implementare un algoritmo capace di comporre lo spartito da un audio musicale.
- Allargare il set di funzionalità presenti nell'editor musicale. Per esempio, fornire uno strumento che permette di dividere i beam evitando di dover cancellare l'elemento e riscrivere le note al suo interno. Oppure uno strumento capace di selezionare e spostare interne parti di uno spartito.

Infine, vi sono delle possibili estensioni che non riguardano principalmente il campo di utilizzo dell'applicazione ma in ogni caso rendono l'applicativo più completo. Tra questi vi sono:

- Codificare l'applicativo per il mondo Android.
- Permette la registrazione di un utente, fornendo uno spazio cloud riservato per la collezione degli spartiti.
- Facilitare la stesura di uno spartito permettendo la condivisione di esso tramite un link, modificando e visualizzando le modifiche in real-time tra i vari compositori.
- Rendere l'applicativo cross-platform. Fornendo una interfaccia web l'utente può facilmente continuare il lavo svolto dall'applicazione attraverso un qualsiasi browser.

Nonostante le proposte sopra riportate siano già valide idee da realizzare per ampliare i casi d'uso, il progetto come già detto in precedenza è legato al modo musicale, e le sue possibili estensioni non sono poche.

Prima di estendere, è doveroso analizzare attentamente la nuova funzionalità, soprattutto verificando la sua reale necessità utilizzando dei questionari indirizzati agli utenti finali.

Bibliografia

- [1] Studiarapido, Machine Learning: cos'è e quali sono le sue applicazioni. Available: <https://www.studiarapido.it/machine-learning-cose-e-quali-sono-le-sue-applicazioni/>
- [2] Nicoletta Boldrini, Deep Learning, cos'è l'apprendimento profondo, come funziona e quali sono i casi di applicazione.
Available: <https://www.ai4business.it/intelligenza-artificiale/deep-learning/deep-learning-cose/>
- [3] Internet4things, Sensori ottici cosa sono, come funzionano, tipologie e ambiti applicativi.
Available: <https://www.internet4things.it/open-innovation/sensore-ottico-cose-come-funziona-tipologie-e-ambiti-applicativi/>
- [4] Andrea Missinato, YOLO – un algoritmo di computer vision ultraveloce per una computer vision in tempo reale.
Available: <https://www.nontek.com/it/yolo-riconoscimento-oggetti-real-time/>
- [5] Wikipedia, Swift (programming language).
Available: [https://en.wikipedia.org/wiki/Swift_\(programming_language\)](https://en.wikipedia.org/wiki/Swift_(programming_language))
- [6] Wikipedia, Xcode. Available: <https://en.wikipedia.org/wiki/Xcode>
- [7] Wikipedia, Core Data. Available: https://en.wikipedia.org/wiki/Core_Data
- [8] Python, What is Python? Executive Summary.
Available: <https://www.python.org/doc/essays/blurb/>
- [9] Jupyter Notebook, The Jupyter Notebook.
Available: <https://jupyter-notebook.readthedocs.io/en/stable/notebook.html>

- [10] Pacha Alexander, Self-Learning Optical Music Recognition.
Available: https://www.researchgate.net/publication/334263279_Dissertation_-Self-Learning_Optical_Music_Recognition
- [11] Rebelo Ana; Fujinaga Ichiro; Paszkiewicz Filipe; Marcal Andre R.S.; Guedes Carlos; Cardoso Jamie dos Santos, Optical music recognition: state-of-the-art and open issues.
Available: <https://link.springer.com/article/10.1007/s13735-012-0004-6>
- [12] Van der Wel Eelco; Ullrich Karen, Optical Music Recognition with Convolutional Sequence-to-Sequence Models.
Available: <https://archives.ismir.net/ismir2017/paper/000069.pdf>
- [13] Gallego Antonio-Javier; Calvo-Zaragoza Jorge, Staff-line removal with selectional auto-encoders.
Available: <https://linkinghub.elsevier.com/retrieve/pii/S0957417417304712>
- [14] Pacha Alexander; Calvo-Zaragoza Jorge; Hajič, Jan jr., Learning Notation Graph Construction for Full-Pipeline Optical Music Recognition. 20th International Society for Music Information Retrieval Conference.