Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in informatica

RETI NEURALI RICORRENTI PER LA GENERAZIONE DI MUSICA SIMBOLICA

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
ANTONIO LOPEZ

Sessione I
2021/2022

Abstract

Il *Deep Learning* ha radicalmente trasformato il mondo del *Machine Learning* migliorando lo stato dell'arte in diversi campi che spaziano dalla *computer vision* al *natural language processing*. Non fermandosi a problemi di classificazione, negli ultimi anni, applicazioni di tipo generativo hanno portato alla creazione di immagini realistiche e documenti letterali. Il mondo della musica non è esente da una moltitudine di esperimenti nello stesso campo, con risultati ancora acerbi ma comunque potenzialmente interessanti. In questa tesi verrà discussa l'applicazione di un di modello appartenente alla famiglia del *Deep Learning* per la generazione di musica simbolica.

Indice

Elenco delle figure

Introduzione

La generazione musicale è uno dei vari campi in cui l'intelligenza artificiale (IA) viene messa alla prova in una situazione in cui viene richiesta creatività. Il problema della generazione è stato esplorato sia nel dominio audio con lavori come JukeBox [6] e WaveNet [8] sia nel dominio simbolico con MusicFrameworks [5] e MusicVAE [13]. I risultati ottenuti da queste reti mostrano le potenzialità dell'applicazione dei modelli di *Deep Learning* ad un problema sicuramente non banale. Nella generazione di musica simbolica c'è carenza di dati a disposizione. Avere accesso, ad una vasta gamma di brani in formato simbolico che spaziano tra generi e stili, risulta difficile, in quanto la codifica di un brano è un compito che richiede l'intervento di musicisti esperti. Questo ha portato alla nascita di community (Wikifonia[1], MidKar[2]) e di progetti come il Lakh midi dataset [11] e il LSDB [10] con lo scopo di creare delle collezioni di dati che fossero aperte e utilizzabili da tutti. Ciononostante, la qualità di tali dati non risulta essere eccelsa: molti dei brani codificati risultano essere scarni, presentando solamente il tema principale e tralasciando informazioni sulla struttura, sugli accordi e sulla tonalità. Inoltre, anche una volta codificati, questi dataset spesso vengono oscurati per questioni di copyright. Nelle diverse sperimentazioni, come quelle sopracitate, risulta spesso assente la generazione di una linea armonica associata alla melodia. Esse, infatti, si concentrano di più quest'ultima e mettono in secondo piano una parte fondamentale per la fruizione di un brano. L'obiettivo di questo lavoro, allora, è proprio quello di utilizzare una delle varie raccolte di brani per poter creare un dataset che possa essere utilizzato per generare, oltre ad una linea melodica principale, una seconda linea composta da accordi. Questi devono risultare coerenti sia tra di loro, sia con la melodia generata, per poter migliorare l'ascolto del brano. Il problema viene affrontato similmente ad uno di generazione di sequenze; un brano diventa l'unione di due sequenze parallele, una composta da note e pause e una da accordi. Il principale compito della rete diventa quello di completare tali sequenze in maniera coerente con la storia di quanto già generato.

[1] Attualmente il progetto è stato abbandonato, ma è ancora possibile accedere alla collezione di brani che vi era presente

[2] `https://midkar.com/`

Struttura della tesi

La tesi è strutturata nei seguenti capitoli:

- **Capitolo 1**: vengono brevemente introdotte le tecnologie utilizzate per la realizzazione della sperimentazione;
- **Capitolo 2**: vengono mostrate le varie fasi della creazione del dataset, partendo dalla scelta dei dati e arrivando alla codifica di questo.
- **Capitolo 3**: viene mostrata l'architettura scelta per la sperimentazione, il suo addestramento e i risultati ottenuti.

Capitolo 1

Tecnologie usate

Introduciamo quelle che sono le tecnologie utilizzate in questo progetto. La presentazione di queste, vuole fornire le principali idee che stanno alla base dei modelli utilizzati.

Deep Learning

Il *Deep Learning* fa parte dell' ampia famiglia dei metodi di *Machine Learning*. Il concetto alla base è molto semplice e si ispira al funzionamento della mente umana.
Una rete neurale artificiale simula la capacità di astrazione delle reti neurali biologiche, cercando di riconoscere ed estrarre informazioni utili per la risoluzione di problematiche complesse. Con apprendimento profondo intendiamo una serie di tecniche, organizzate

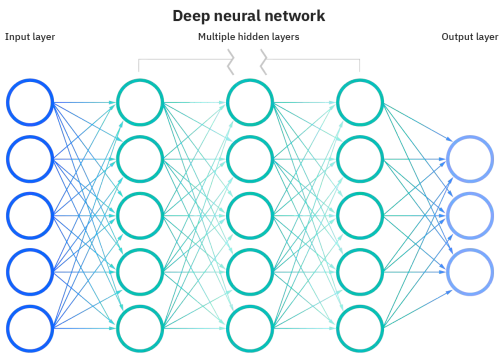


Figura 1.1: Esempio di rete neurale: possiamo osservare come ciascun livello utilizzi le informazioni ottenute dal livello precedente[1].

a livelli, dove ciascuno fornisce i propri risultati al successivo. Così facendo si viene a creare una struttura gerarchica e adattiva che permette ai diversi livelli di concentrarsi sulle diverse caratteristiche degli input.

Applicazioni del Deep Learning

L'utilizzo del *Deep Learning* è ormai ampiamente diffuso e i suoi campi di applicazione spaziano dalla medicina alla *cybersecurity*. Questo è prova di come tale approccio risulti efficace per affrontare problemi anche complessi, che spesso non riescono ad essere risolti dal normale approccio algoritmico; alcuni esempi sono:

- Il riconoscimento di immagini;
- La diagnostica medica;
- Il riconoscimento vocale;
- La traduzione linguistica;
- La generazione di immagini, testi, musica.

La scelta del *Deep Learnig*, per questo progetto, è quindi risultata naturale grazie al suo vasto campo di applicazione anche nella musica. In particolare alcuni lavori interessanti sono:

- *DeepSinger*: sintetizzatore di una linea vocale cantata [12];
- *Groove2Groove*: modifica lo stile di un brano [4];
- *Flow Machines*: in grado di analizzare l'idea primaria del compositore e aiutarlo, generando idee durante la composizione [9].

[1] `https://www.ibm.com/it-it/cloud/learn/neural-networks`

Reti neurali ricorrenti

Le reti neurali ricorrenti RNN, sono una classe di rete neurale artificiale. Tale tipologia permette di gestire dati di tipo sequenziale, ovvero successioni di input ordinati in maniera temporale. Questo tipo di architettura permette di affrontare problemi dove vi sono dipendenze tra le varie componenti di una sequenza. Per questa ragione, la risposta fornita dalla rete, in un certo momento, non dipende solamente dall'input attuale, ma anche dalla storia precedente. Alcuni esempi di applicazioni delle RNN sono:

- Problemi di text processing: traduzione, completamento di frasi...;
- *Time Series Forecasting*: data una serie di dati fino al momento **t**, si cerca di prevedere quelli che saranno i possibili valori futuri ai momenti **t+1**, **t+2** eccetera. Tali tipi di previsioni sono per esempio usate per anticipare eventi, ad esempio il possibile crollo della borsa;

In altre parole, tutti quei problemi dove il passato risulta essere una parte fondamentale, sia per poter prendere delle decisioni sia per poter prevedere, o meglio fare ipotesi, sul futuro.

Funzionamento

In una rete neurale ricorrente abbiamo la presenza di cicli. Infatti oltre all'input corrente si riceve anche il precedente stato del neurone. Tale stato viene detto *hidden state* e può essere visto come la memoria del nostro nodo. Questa, permette di mantenere le informazioni di contesto necessarie per poter meglio interpretare il prossimo input della serie.

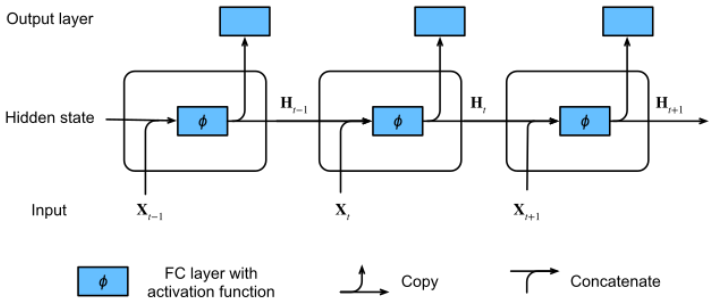


Figura 1.2: Esempio di rete neurale ricorrente [14]

Vanishing o Exploding Gradient

All'atto pratico, l'utilizzo di una RNN spesso porta al problema del *vanishing* o *exploding gradient*. L'addestramento di una rete prevede una fase detta *backpropagation*, nella quale vengono aggiornati i pesi della rete, tale fase prevede il calcolo del gradiente delle funzioni presenti nei neuroni. In una rete di tipo ricorrente, in base alla funzione di attivazione, il valore del gradiente può diminuire rapidamente (*vanishing*) oppure aumentare rapidamente (*exploding*), a causa della lunga catena di neuroni da dover attraversare. Questo rende, impossibile un addestramento efficace, per mitigare tale problema vengono utilizzati neuroni ricorrenti con una struttura più complessa.

Long short-term memory

Una *long short-term memory* è una variante di una rete neurale ricorrente, ottenuta modificando la struttura interna di un neurone. All'interno di questo, vengono aggiunte diverse porte (o *gate*), oltre che un nuovo input detto *memory*, che consentono al neurone di decidere cosa è importante da ricordare e cosa va dimenticato. Vediamo quali sono queste porte:

- *Forget gate*: decide se l'informazione in ingresso deve essere memorizzata o meno. Riceve l'input corrente e il precedente stato del neurone;
- *Input gate*: decide il peso della memoria sul corrente input concatenato al precedente stato;
- *Output gate*: decide lo stato attuale del neurone.

Sicuramente l'addestramento di questa tipologia di RNN risulta più lento a causa del maggior numero di parametri da aggiornare. Nonostante questo, permette di limitare i problemi legati al calcolo del gradiente.

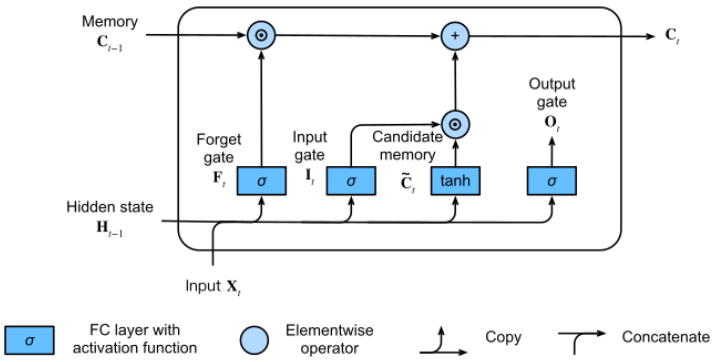


Figura 1.3: Esempio di neurone in una LSTM [14]

Python

Python é un linguaggio ad alto livello che, col tempo, é diventato uno tra i più popolari linguaggi per lo sviluppo di modelli e algoritmi di *Machine Learning*. Tra le varie librerie e *framework* che forniscono API per la progettazione e l'addestramento di reti neurali le principali sono:

- *Tensorflow + Keras*;
- *PyThorch*;
- *MXNet*.

Tensorflow

Tensorflow è una libreria open source che fornisce tutte le risorse necessarie alla creazione di modelli per l'apprendimento automatico. Rilasciata nel 2015 è diventata velocemente una delle più popolari soprattutto grazie anche alla presenza di numerosi progetti e risorse che la rendono un'ottima scelta per che si avvicina per la prima volta a tale mondo. La libreria permette anche di sfruttare le risorse di calcolo della GPU per migliorare i tempi di addestramento.

Keras

Keras è una libreria open source ad alto livello che fornisce un'astrazione superiore rispetto a quella fornita da *Tensorflow*. Questa non va in alcun modo a sostituirlo, bensì ne estende l'API semplificando l'interazione tra lo sviluppatore e la libreria.

Google Colab

L'addestramento di reti neurali è sicuramente un'operazione impegnativa per un calcolatore, soprattutto se questo non dispone di risorse elevate. Per questa ragione, sono nati diversi servizi che permettono l'esecuzione di codice su macchine remote, le quali presentano un hardware ad *hoc* per questo tipo di elaborazioni. *Colab* è una di queste piattaforme, messa a disposizione da *Google*, che permette l'esecuzione di codice (scritto sotto forma di Jupyter Notebook) in *cloud*. La piattaforma fornisce allo sviluppatore una discreta potenza di calcolo: è, infatti, possibile scegliere un ambiente di lavoro ottimizzato per l'addestramento di reti neurali.

Capitolo 2

Creazione e preprocessing del dataset

Il dataset scelto utilizza la collezione di brani che la community di Wikifonia ha codificato durante gli anni. Il dataset contiene circa 6,675 spartiti codificati in formato **MusicXML (MXL)** che spaziano dal classico (con brani di Bach e Mozart) al rock, passando per jazz e blues. Nonostante la mancanza di informazioni riguardanti la struttura del brano, come inizio e fine di strofa e ritornello oppure la presenza di bridge e special, il dataset codifica per la maggioranza dei brani anche la linea armonica diventando, quindi, la miglior scelta per tale progetto. Di seguito, andiamo ad esporre le operazioni di preprocessing effettuate sui dati per renderli utilizzabili dalla rete.

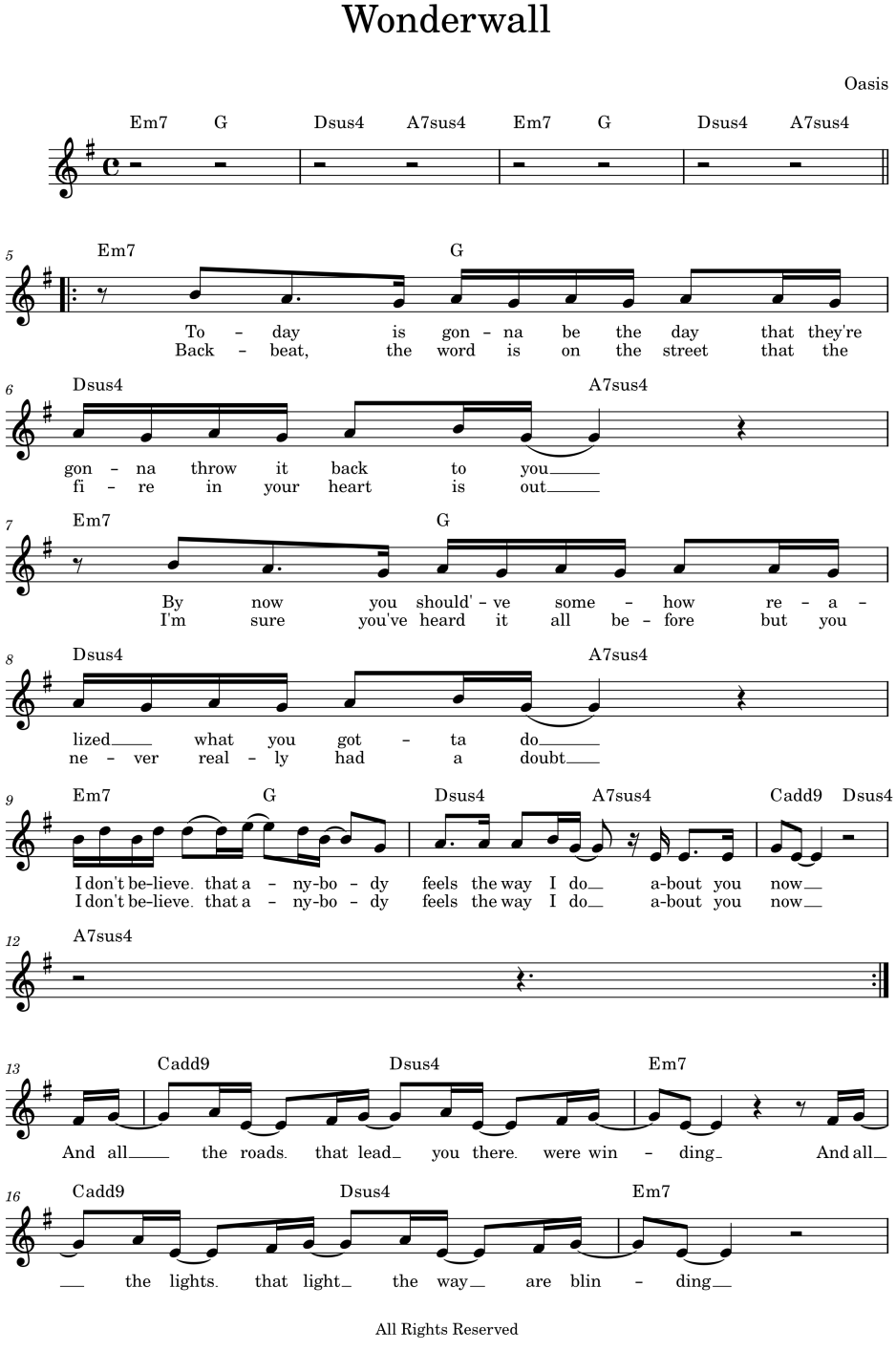


Figura 2.1: Esempio di brano presente nel dataset

Preprocessing

Con *preprocessing* dei dati si intende la manipolazione e la selezione di questi prima del loro utilizzo. Questa fase risulta fondamentale per ogni modello di *Machine Learning*. Non importa quanto una rete possa essere ben costruita, se i dati a disposizione non sono scelti in maniera appropriata i risultati ottenuti non saranno mai soddisfacenti, *garbage in, garbage out*. Nel nostro caso, la fase di preelaborazione avviene sui brani presenti nel dataset. In particolare, il nostro processo si compone di diverse fasi:

- Pulizia del dataset: vengono eliminati quei brani che non soddisfano determinate caratteristiche;
- Trasformazione dei dati: una volta selezionati i dati subiscono delle modifiche per poter essere meglio utilizzati;
- Selezione delle informazioni: da ciascun brano vengono mantenute solamente le informazioni necessarie, ad esempio titolo e compositore risultano essere elementi inutili per gli scopi della rete.
- Codifica in un formato intermedio: una volta aver estratto le informazioni necessarie, queste vengono salvate in un formato intermedio per evitare di dovere rieseguire il processo in futuro. Tale formato, seppur simile, non è lo stesso di quello fornito alla rete durante l'addestramento.

Il formato MIDI

Sebbene il formato scelto durante la fase di preprocessing non sia esattamente il MIDI (Musical Instrument Digital Interface), questo riprende alcune delle idee che ne stanno alla base. Risulta, quindi, importante capire quelli che sono alcuni dei concetti alla base di questa codifica.

Codifica delle note

Ciascuna nota viene codificata attraverso gli eventi **NOTE ON** e **NOTE OFF**: la differenza di tempo che intercorre tra le due rappresenta la durata del suono.
I numeri di nota trasmessi e riconosciuti vanno da 0 a 127 e sono equivalenti ad un'ipotetica tastiera di 128 tasti. Ad esempio, Do centrale è codificato con il numero 60

Octave	Notes											
Number	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127	–	–	–	–

Figura 2.2: Note e relative codifiche

Codifica degli accordi

Il MIDI non presenta una vera e propria codifica per gli accordi, i quali vengono semplicemente trattati come un caso di *polifonia*; di conseguenza la codifica risulta identica a quelle per le note, con gli eventi di accensione e spegnimento che risulteranno contemporanei per ciascuna nota dell'accordo. MIDI supporta una polifonia fino ad otto voci, superato tale valore una di queste viene eliminata.

music21

Interfacciarsi e manipolare direttamente il formato MIDI (o MXL nel nostro caso) risulterebbe particolarmente dispendioso in termini di tempo. Per questo motivo, si è deciso di ricorrere alla libreria *music21*, la quale fornisce un'ottima interfaccia per la manipolazione dei file MIDI e MXL. La libreria fornisce una rappresentazione gerarchica ad alto livello degli spartiti: ciascuna nota è contenuta in una battuta che a sua volta è contenuta in una linea che è contenuta in uno *score*. Ogni brano contiene, inoltre, dei metadati tra cui troviamo il titolo e il nome dell'autore.

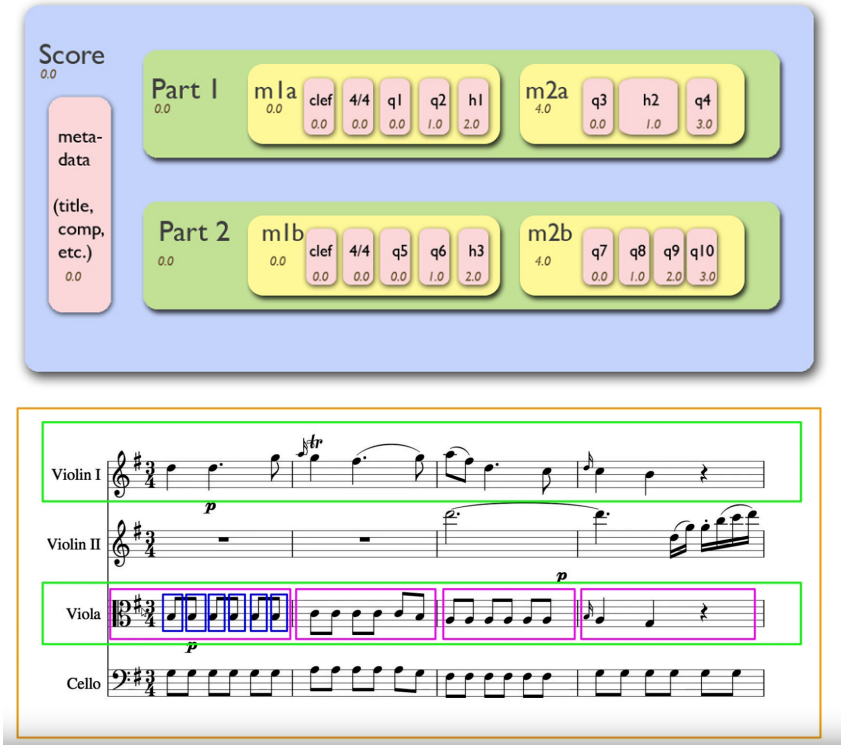


Figura 2.3: Rappresentazione di un brano in music21[1]

Una volta caricato un brano, la libreria permette la manipolazione di ogni singolo elemento all'interno dello *score*. É quindi possibile aggiungere e modificare accordi e note in qualsiasi punto dello spartito.

[1] `https://web.mit.edu/music21/doc/usersGuide/usersGuide_17_derivations.html`

Pulizia del dataset

La prima fase del *preprocessing* consiste nel convertire i file MXL in MIDI (o meglio nel formato fornitoci da *music21*) andando a scartare tutti quei file che risultano:

- Malformati: presentano errori nella codifica MXL;
- Avere un'indicazione di tempo non supportata: la codifica scelta per rappresentare i brani permette solamente l'utilizzo di un'unica indicazione di tempo.
  Quella scelta all'interno del lavoro è in 4/4 in quanto risulta essere la più diffusa, brani in 3/4, 5/4, 6/8 etc. non verranno presi in considerazione;
- Avere metriche di tempo non supportate: adottando una codifica basata su campionamento, ogni brano che presenta dei simboli che utilizzano un valore di tempo non multiplo di quello scelto per il campionamento risultano incodificabili, quindi vengono scartati. Per esempio, adottare un campionamento sugli ottavi esclude tutti quei brani che presentano i sedicesimi.

Controllo dei file malformati

Tale tipo di controllo è eseguito autonomamente dalla libreria durante il caricamento del file MXL. In caso di errore viene sollevata un'eccezione che porta a saltare il brano attuale e passare al succesivo.

Controllo sull'indicazione di tempo

Per eseguire tale tipo di analisi, ciascun brano viene prima caricato e successivamente viene verificata l'indicazione di tempo presente. La libreria salva tale informazione nei metadati dello spartito. È, però, possibile avere casi in cui l'indicazione di tempo risulta sconosciuta in quanto non è stata codifica dal creatore del brano: in queste situazioni il brano viene scartato.

Controllo sulle metriche di tempo

Per eseguire tale controllo viene esaminata la durata di ciascuna nota presente nel brano. La durata, memorizzata come un valore reale (ad esempio una semicroma ha un valore

pari a 0.25), viene confrontata con il valore di campionamento controllando che sia un multiplo di tale valore. Se così non è, il brano viene scartato e si passa al successivo.

Padding degli accordi

Per la linea melodica, non possiamo avere alcun "salto" dall'inizio alla fine del brano, in altre parole, per ciascuna battuta, e per tutta la durata di questa, deve essere presente una nota o una pausa. Il discorso è diverso per la linea armonica. Infatti, è pratica comune indicare l'assenza di accordi semplicemente non scrivendone nessuno, al posto di utilizzare il simbolo *no chord* (*N.C.*). Il dataset scelto non è esente da questa pratica, di conseguenza, è necessario andare a inserire manualmente tali accordi per evitare di avere una lunghezza di linea differente dalle altre che codificano il brano. Questa operazione non risulta comunque complicata, l'assenza di accordi può essere solamente presente o all'inizio del brano oppure per tutta la durata di questo (non è quindi presente una linea armonica). Si procede quindi ad aggiungere, in testa al brano, un numero di simboli *N.C.* pari alla lunghezza della melodia meno la lunghezza attuale della linea armonica.

Trasposizione

La trasposizione è la pratica musicale mediante la quale un brano viene riscritto in una tonalità differente da quella originaria. Trasporre i brani in un'unica tonalità è una tecnica particolarmente diffusa in lavori simili [2] [1]. Ci permette sia di avere l'intero dataset a disposizione, sia di poter ignorare tutte le problematiche relative alla definizione di una tonalità di riferimento durante la generazione. In questo caso, per semplicità, viene utilizzata come tonalità di riferimento il **Do maggiore** per i pezzi che presentano una tonalità maggiore e il **La minore** per i brani minori.

Trasposizione di un brano

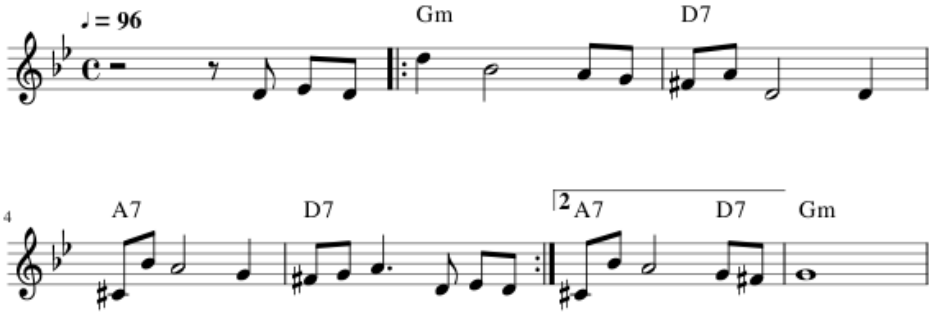
A questo punto della preelaborazione abbiamo tutti i brani che faranno parte del dataset, non verrà quindi eseguita nessuna ulteriore scrematura. La trasposizione di un brano si compone di tre fasi:

1. Individuazione della tonalità del brano: questa informazione può essere già presente all'interno dello spartito, se inserita in fase di codifica. Come per l'indicazione di

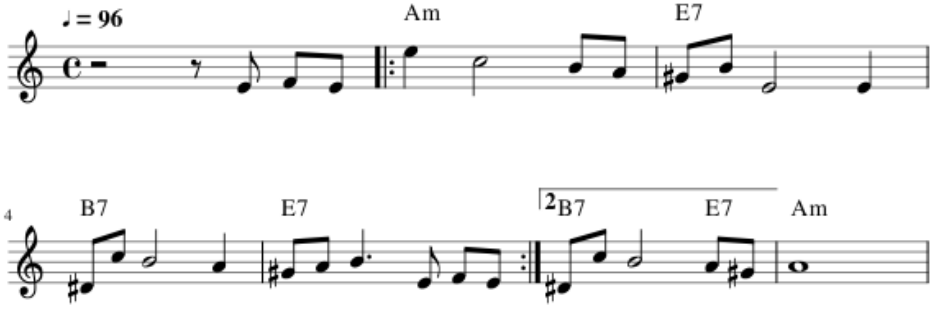
tempo tale dato è presente tra i metadati dello spartito. Se l'indicazione della tonalità non è presente, *music21* permette di inferirla. In entrambi i casi bisogna fare affidamento o nelle capacità di chi ha codificato il brano o nella bontà dell'analisi eseguita dalla libreria;

2. Calcolo dell'intervallo: evitando di proposito una definizione musicalmente precisa di intervallo lo definiamo come la distanza che separa due tonalità;

3. Trasposizione: per questo passo *music21* ci fornisce un metodo per trasporre un intero spartito dato un intervallo.



(a) Brano originale



(b) Brano trasposto

Figura 2.4: Esempio di trasposizione

Codifica del dataset

L'ultima fase del *preprocessing* consiste nel codificare in formato testuale il nostro dataset. Inoltre, per evitare di dover ripetere le operazioni di preprocessing e codifica, l'intero dataset viene salvato su file.

Formato utilizzato

La codifica utilizzata rappresenta un singolo brano attraverso 5 linee parallele di uguale lunghezza:

- Linea degli accordi: rappresenta in notazione anglosassone la qualità e il tipo di accordo;
- Linea ritmica degli accordi: tale linea indica se l'accordo viene suonato oppure se viene mantenuto;
- Linea melodica: utilizza la codifica MIDI del *pitch* per rappresentare la linea melodica principale del brano (in caso di più linee melodiche viene presa in considerazione solamente la prima);
- Linea ritmica melodica: indica se la nota corrispondente viene suonata o mantenuta;
- Linea delle battute: vengono codificate le battute indicando l'inizio di ciascuna. Tale linea è stata aggiunta successivamente come ulteriore task da far compiere alla rete per migliorare la qualità della generazione.

La lunghezza delle linee dipende dal valore di campionamento scelto: per esempio, scegliendo un campionamento pari ai sedicesimi, ciascuna battuta verrà codificata attraverso sedici token per ciascuna linea. La scelta di un valore di campionamento più alto permette la codifica di un numero maggiore di brani a discapito però dell'occupazione di memoria e della velocità di addestramento della rete.

Codifica delle linea armonica e melodica

La linea melodica viene codificata andando, per ciascuna nota, a dividere la sua durata per il valore di campionamento. Il risultato così ottenuto indica il numero di token che

codifica tale nota. Il valore di questi token sarà uguale al *pitch* corrispondente. Per la linea ritmica abbiamo un singolo uno (che rappresenta il momento in cui la nota viene suonata) seguito da un numero di zeri sufficienti a raggiungere il numero di token richiesti. Il procedimento è analogo per la codifica degli accordi.

Codifica delle battute

Ciascuna battuta è invece rappresentata da una stringa composta da un carattere che indica l'inizio di questa più una serie di caratteri uguali che rappresentano il perdurare della battuta. La lunghezza di tale stringa è pari a:

$$\frac{time\ signature}{time\ step}$$

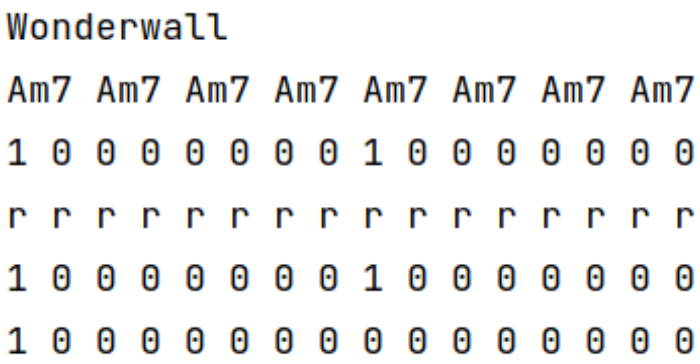


Figura 2.5: Codifica in sedicesimi di una battuta

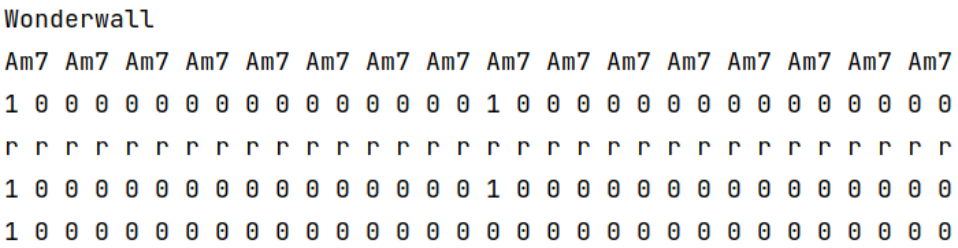


Figura 2.6: Codifica in trentaduesimi di una battuta

Limiti della codifica

La codifica scelta per la rappresentazione dei dati presenta delle limitazioni in termini di espressività:

- Non è possibile codificare brani con indicazioni di tempo differenti;
- Similmente, risulta impossibile codificare terzine, sestine e in generale tutte quelle metriche di tempo che non risultano essere un multiplo del valore di campionamento scelto;
- A causa della natura sequenziale della rappresentazione, non risulta essere possibile la codifica di note suonate all'unisono. Ciononostante, consapevoli di aumentare in maniera considerevole la grandezza del vocabolario, non è complicato estendere quest'ultimo includendo anche tali combinazioni. In questo caso, si è deciso di mantenere solamente una nota.

Divisione del dataset

Il dataset è stato suddiviso in sottoinsiemi di diversa grandezza per permettere una maggiore velocità di addestramento della rete durante le fasi di sviluppo e *testing*:

- XSmall: circa 40 brani, utilizzato principalmente per testare la codifica e il *preprocessing*;
- Small: raccolta di circa 797 brani per il *training* e 300 per il *testing*, risulta essere il dataset maggiormente utilizzato in quanto permette un buon bilanciamento tra tempo di codifica, addestramento e risultati ottenuti;
- Medium: circa 2400 brani per il *training* e 600 per il *testing*, usato sporadicamente a causa dell'elevato tempo necessario per l'addestramento.

Preprocessor

I compiti sopra elencati sono svolti dal *preprocessor*. Questo riceve in input:

- Il valore di campionamento: ottavi, sedicesimi, trentaduesimi, etc.

- Informazioni da codificare: allo stato attuale il *preprocessor* permette di decidere se aggiungere ulteriori linee alla codifica standard (accordi, ritmo accordi, melodia, ritmo melodia), come, ad esempio, quella che codifica le battute o quella che rappresenta il "battere" ed il "levare".

Multithreading

Per migliorare la velocità di *preprocessing* si è utilizzato il supporto al *multithreading* fornito da *Python*. Per utilizzare questa funzionalità è necessario definire la funzione di *scan*, ovvero quella eseguita in parallelo dai *thread*. In questo caso, tale procedura, equivale a quella che racchiude le fasi di *preprocessing* viste in precedenza (tranne quella di salvataggio su file). Definita la cartella che contiene i brani da processare, ogni *thread* procede in maniera indipendente, fornendo il risultato della sua elaborazione, e richiedendo il prossimo file da elaborare. Solamente una volta finita la scansione della cartella il controllo ritorna al *preprocessor* che si occupa di salvare i risultati ottenuti.

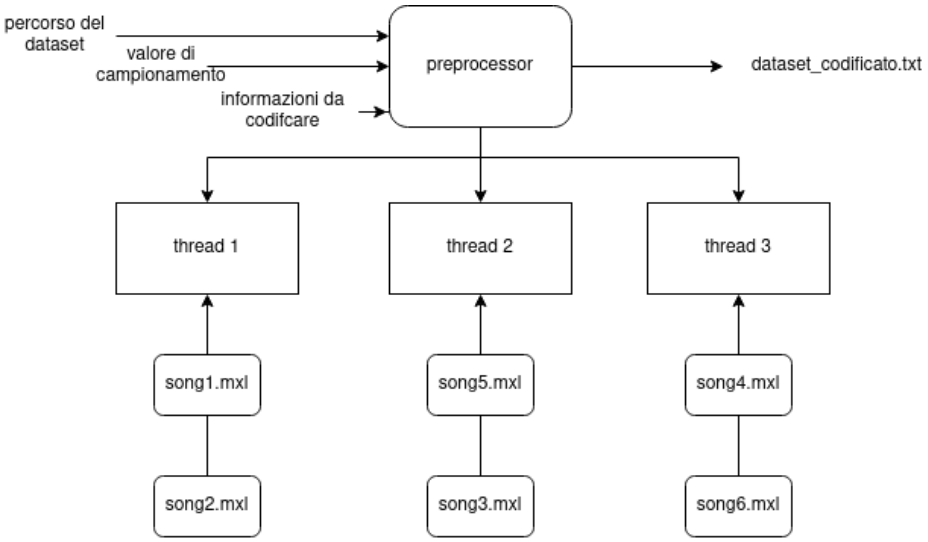


Figura 2.7: Il *preprocessor* una volta lanciati i *thread* aspetta la fine della scansione dei brani per poter proseguire.

Loader

Il *Loader* è un altra componente fondamentale che permette alla rete di astrarre da tutta la logica per la creazione delle sequenze necessarie all'addestramento ed al *testing*. Questo si occupa di:

- Caricare il dataset preprocessato e codificato;
- Creare i vocabolari necessari;
- Mappare ciascuna valore presente nel dataset con l'indice del relativo valore presente nel vocabolario di riferimento;
- Eseguire il *one hot encoding* di ciascun token;
- Creare le sequenze necessarie alla rete.

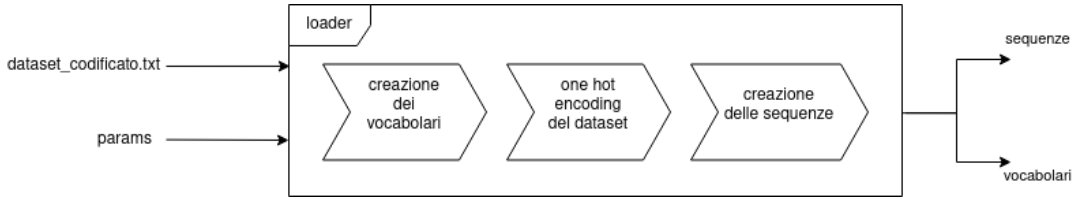


Figura 2.8: Struttura del *loader*

Caricamento del dataset

Il dataset viene salvato in memoria attraverso l'utilizzo di un dizionario $< chiave : valore >$ dove la chiave rappresenta il nome della linea e il valore corrisponde ad un *array* che contiene tutti i token relativi a tale linea per tutti i brani presenti nel dataset. Per questa ragione, oltre che il percorso del file creato dal *preprocessor*, è necessario avere il numero ed il nome delle linee con cui il dataset è stato codificato. Andiamo a definire *params* come la lista dei nomi di queste linee (nel nostro modello *params* sarà quasi sempre [*chords, chords play, melody, melody play, bars*]). La scansione del file avviene leggendo un numero di righe pari a $|params|$ e, successivamente, memorizzando ciascuna di esse in coda all'*array* corrispondente nel dizionario. Il numero di righe presenti nel file deve essere un multiplo di $|params|$.

Creazione dei vocabolari

Il prossimo passo è la creazione di un vocabolario per ogni elemento di *params* (nel nostro caso abbiamo cinque vocabolari). Ciascuno di questi, viene creato fornendo alla classe *Vocab* (presente all'interno di *Loader*) l'elenco dei token presenti; la classe assocerà a ciascun valore un indice in ordine decrescente di frequenza (in altre parole, al token più frequente viene associato l'indice zero, al secondo l'indice uno e così via).

accordo	indice	frequenza	%
C	0	128079	22.59
G7	1	57735	10.18
F	2	46996	8.29
Am	3	35190	6.21
G	4	31982	5.64
Dm7	5	18016	3.18
D7	6	17022	3.00
...	...	...	...

(a) Vocabolario e frequenze degli accordi

nota	indice	frequenza	%
r (pausa)	0	68539	12.09
64	1	54337	9.58
67	2	53889	9.50
60	3	49929	8.81
72	4	48982	8.64
69	5	39580	6.98
62	6	38292	6.75
...	...	...	...

(b) Vocabolario e frequenze delle note

nota suonata	indice	frequenza	%
0 (mantenuta)	0	422233	74.47
1 (suonata)	1	144765	25.53

(a) Vocabolario e frequenze della linea ritmica melodica

accordo suonato	indice	frequenza	%
0 (mantenuta)	0	528549	93.22
1 (suonata)	1	38449	6.78

(b) Vocabolario e frequenze della linea ritmica armonica

Mappatura del dataset

Una volta che i vocabolari sono stati creati è possibile eseguire la mappatura di ciascun token con il suo indice corrispondente, utilizzando il vocabolario di riferimento.

One hot encoding

L'ultimo passaggio prima della creazione delle sequenze è il *one hot encoding*. Tale tipo di codifica, utilizzata con variabili di tipo categorico, prevede la creazione di una lista di lunghezza pari al numero di categorie. I valori all'interno del vettore di *encoding* sono

tutti pari a zero ad esclusione di quello presente all'indice di valore pari alla categoria che si vuole codificare.

Creazione delle sequenze

La creazione delle sequenze consiste nel dividere il nostro dataset in blocchi (di dimensione fissa); ciascuno di questi blocchi verrà poi fornito alla rete in fase di addestramento. Ogni sequenza è composta da due parti:

- *Input*: rappresenta la successione da completare;
- *Label*: corretta continuazione dell'input.

La grandezza dei nostri blocchi sarà pari a *sequence_length + 1* $\times$ *params*. Generalmente *sequence_length* è scelto in modo tale da avere al suo interno una o più battute. Il suo valore è solitamente un multiplo di otto (discorso diverso è nel caso in cui, abbiamo indicazioni di tempo differenti come 3/4, in questo caso bisognerebbe utilizzare multipli di sei). Una volta a disposizione, le sequenze sono organizzate in *batch* e possono finalmente essere utilizzate dalla rete.

Capitolo 3

Architettura della rete

Di seguito andremo ad esporre l'architettura della rete, il suo addestramento ed i dettagli relativi alla generazione.

La rete è così strutturata:

- **Layer di input**: la rete si compone di un numero variabile di input, uno per ciascuna linea in cui è stato codificato il brano. Ciascuna linea di input è di tipo categorico e la sua dimensione dipende dalla *sequence_length* e dalla grandezza del vocabolario;
- **Layer di concatenamento**: concatena i vari input;
- **LSTM**: in generale è utilizzabile ciascuna tipologia di rete che possa lavorare su sequenze (RNN, GRU);
- **Layer di output**: per ciascuna linea di input abbiamo un corrispondente output rappresentato da una *feed foward* alla quale viene applicata come funzione di attivazione una *softmax*.

Le metriche e le *loss* utilizzate per tutte le linee in uscita:

- *Categorical crossentropy*;
- *Categorical accuracy*.

Alternativamente, eliminando l'embedding di tipo *one hot*, sono utilizzabili la *sparse categorical crossentropy* per la *loss* e la *sparse categorical accuracy* per *l'accuracy*. Similmente, per quelle linee che presentano un vocabolario composto da due soli elementi è possibile fare uso delle versioni binare delle funzioni sopracitate.

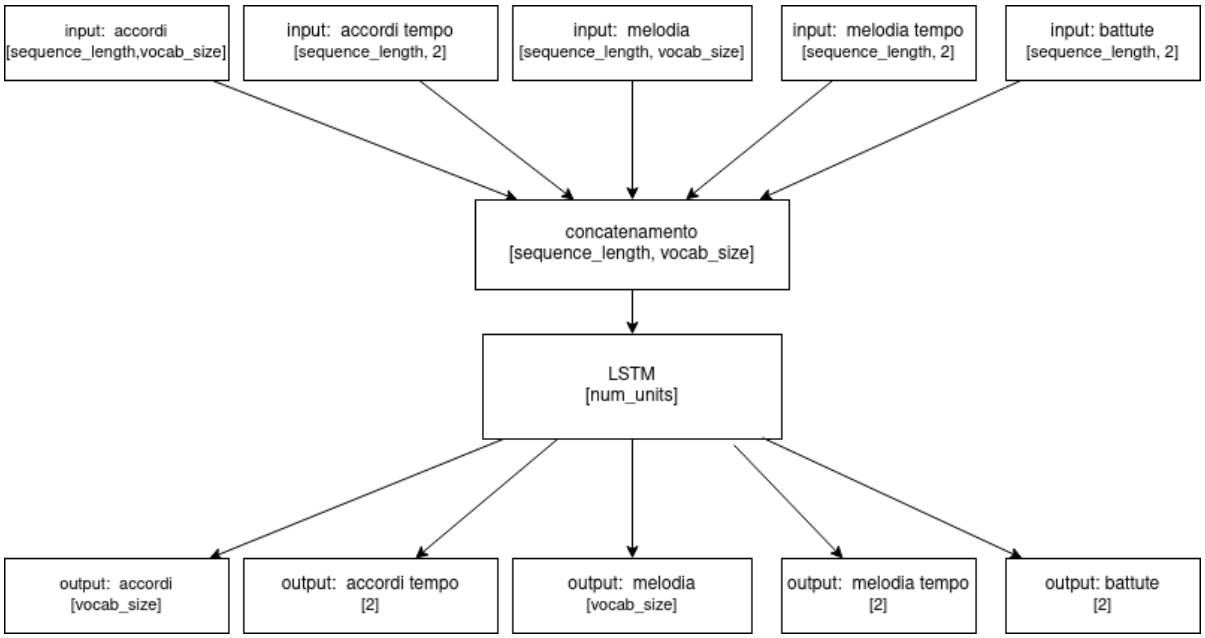


Figura 3.1: Architettura del modello

Vecchia architettura

Inizialmente, il primo modello creato prevedeva l'utilizzo di un unico input di dimensione pari a $sequence\_length \times params$. Inoltre era presente anche un unico layer di *output* e nessuna forma di *encoding*. Tale modello non forniva però risultati di rilievo, e risultava confusionario avere un unico vettore che rappresentasse linee differenti.

Addestramento

L'addestramento della rete è stato svolto seguendo diverse configurazioni, modificando i seguenti parametri:

- Valore di campionamento: i valori utilizzati sono gli ottavi, i sedicesimi ed i trentaduesimi;
- Codifica del brano: è stata modificata la codifica del brano aggiungendo una linea che codificasse le battute e, successivamente, una che codificasse il battere ed il levare all'interno di una singola battuta;
- Numero di *units* presenti nella LSTM: i valori utilizzati sono: 128, 256, 512;
- Dimensione del dataset: il dataset è stato suddiviso in diverse dimensioni, i test sono stati effettuati principalmente sul dataset di dimensione *small*.

output	accuracy	loss
accordi	0.92	0.2904
accordi tempo	0.91	0.1242
melodia	0.77	0.7232
melodia tempo	0.88	0.2881
battute	0.99	0.0057

(a) Accuracy e loss sul train set

output	accuracy	loss
accordi	0.84	0.8345
accordi tempo	0.91	0.2858
melodia	0.61	1.6049
melodia tempo	0.79	0.5695
battute	0.99	0.0249

(b) Accuracy e loss sul test set

Tabella 3.1: I valori qui presentati si riferiscono all'addestramento di una specifica configurazione. Ciononostante, le metriche degli altri modelli non si differenziano troppo da queste.

Sono invece rimasti invariati:

- Il numero di epoche: tutti gli addestramenti sono stati eseguiti con 10 epoche, aumentare tale valore non ha portato a nessun tipo di miglioramento;
- *Learning rate*: pari a 0.001;
- *Optimizer*: la scelta è ricaduta su *Adam*.

Una volta terminato l'addestramento i pesi del modello vengono salvati per poter essere successivamente utilizzati in fase generativa.

Generazione

La generazione avviene fornendo come input una sequenza di lunghezza pari alla *sequence_lenght* scelta durante l'addestramento. Dato tale input, la rete produce per ciascuna linea una distribuzione di probabilità, associando a ciascuna categoria un valore compreso tra 0 e 1. Una volta ottenute le probabilità si procede a scegliere la classe con valore maggiore fornendo in output gli indici relativi a tali classi.

Generazione di un brano

Per la generazione di un brano viene iterato il processo di sopra citato andando di volta in volta a modificare le sequenze in input nel seguente modo:

- La sequenza iniziale detta *seed* viene fornita in input. Nei test effettuati corrisponde sempre all'inizio di un brano presente nel dataset;
- Viene generata una possibile continuazione della sequenza, la quale viene concatenata al precedente input;
- Prima di ripetere la generazione, viene eliminata la testa della sequenza, ovvero la parte meno recente;
- Si ripetono tali punti finchè non si raggiunge la lunghezza voluta. Tutte le sequenze generate sono salvate in un nuovo vettore che rappresenta un nuovo brano.

Si procede, successivamente, al salvataggio del brano in formato testuale andando a sostituire agli indici delle classi i loro valori corrispondenti

Esempi

Di seguito andiamo a mostrare alcuni brani generati dalle diverse configurazioni di dataset, parametri della rete e *seed*. Per fare questo, introduciamo la seguente notazione per i titoli dei brani generati **nomebrano_[grandezza dataset][informazioni aggiuntive]_[num_units]** dove:

- Grandezza dataset: indica la partizione del dataset usata;

- Informazioni aggiuntive: indica la presenza di ulteriori linee per la codifica del brano, attualmente possiamo avere:
    1. **B**: indica la presenza di una linea che codifica le battute;
    2. **U**: indica la presenza di una linea che codifica il battere ed il levare in una battuta.
- Num units: indica il numero di unità all'interno presenti nella LSTM.

Oasis - Wonderwall

Possiamo notare una buona generazione per due delle tre configurazioni. Per quanto riguarda il brano ottenuto aggiungendo alla codifica del dataset l'indicazione del battere e del levare, notiamo un notevole peggioramento rispetto alla qualità del generato.

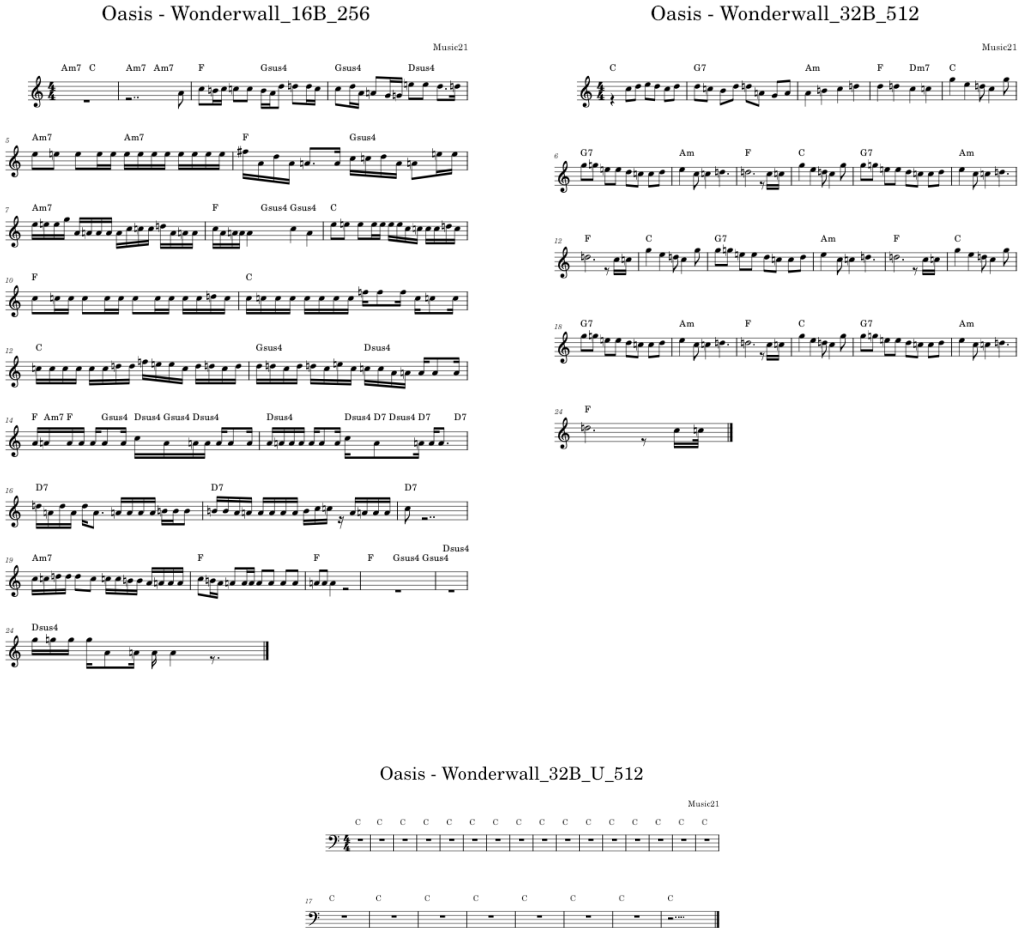


Figura 3.2: Esempi di generazione con seed iniziale il brano *Oasis - Wonderwall*.
Si può notare come la generazione cambi completamente andando a modificare il valore di campionamento. Tutti e tre gli esempi generati presentano sequenze ripetitive sia di singole note ma anche di intere frasi.

Alan Menken, Howard Ashman - Under the Sea,
Carl Petter Opsahl - You

Il **numero di unità** influisce pesantemente sulla generazione quando si utilizzano i trentaduesimi e i sedicesimi come valore di campionamento del dataset.

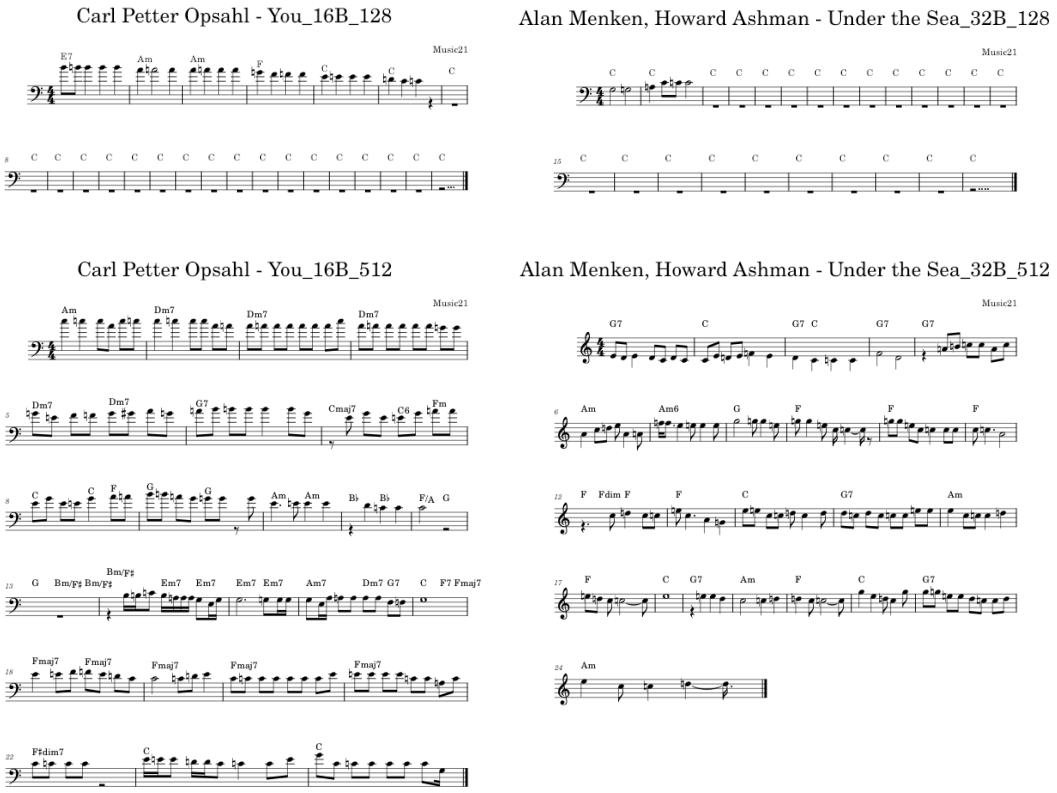


Figura 3.3: Esempi di generazione con diversi numeri di unità per la LSTM. Notiamo come la generazione sia meno ripetitiva con un numero maggiore di unità

Considerazioni

Dopo aver testato e confrontato il risultato delle diverse generazioni, ottenute modificando il *seed* iniziale e la struttura della rete, possiamo affermare che:

1. Per alti valori di campionamento è necessario avere un numero di unità elevate; questo è dovuto alla necessità, da parte della rete, di dover memorizzare sequenze di lunghezza maggiore;

2. L'utilizzo di un'ulteriore linea, per la codifica del battere e del levare, non migliora la generazione, andando in alcuni casi a peggiorare il risultato prodotto;

3. Al contrario, il task che prevede la predizione dell'inizio delle battute risulta utile;

4. La qualità delle sequenze generate diminuisce con la lunghezza di queste;

5. Per quanto riguarda la varietà delle frasi generate, si può notare come a fronte di alcune ripetizioni, si hanno anche sottosequenze originali e godibili.

6. La configurazione **32B_512**, tende a generare sempre la stessa linea melodica anche modificando il *seed* iniziale. Per ciascun seme infatti, dopo un certo numero di frasi tende a convergere sempre alla stessa sequenze. Tale sequenza è presente in uno dei brani del dataset.

Nonostante i risultati ottenuti non siano sicuramente comparabili a degli spartiti reali, è interessante come, anche un modello semplice possa fornire dei discreti risultati. Degna di nota è anche la mancanza di correlazione tra *l'accuracy* del modello è la qualità della generazione. Infatti, nonostante questa metrica si attesti su alti valori, non ci assicura in alcun modo una buona qualità delle sequenze in output. Viene quindi fuori la difficoltà nel valutare modelli che lavorino in campo generativo. Questi, a differenza dei modelli di classificazione, non hanno a disposizione una vera e propria *ground truth*, con cui andare effettivamente a testare l'efficacia del modello.

Decodifica

Il brano così generato risulta essere in una forma del tutto simile a quella dei brani presenti e codificati nel dataset. Per poter essere visualizzato e riprodotto c'è bisogno di produrre un file MXL (oppure MIDI), a partire dal brano salvato in modo testuale. Tale compito è svolto dal *Decoder*, il quale in maniera simmetrica a quello che avveniva per la codifica del brano, preso in input il valore di campionamento e le informazioni da codificare (nel nostro caso solamente le linee riguardanti accordi, melodia e le relative linee ritmiche), utilizza la libreria *music21* per la generazione di un file MXL. Il processo di decodifica è cosi svolto:

- Vengono letti, in maniera parallela, la linea melodica e la corrispondente ritmica (il procedimento è identico per la linea armonica);
- Se la nota letta è differente dall'ultima oppure se la linea ritmica presenta il carattere 1:
    1. Si aggiunge allo spartito la precedente nota con tempo uguale a *count* moltiplicato per il *time step*;
    2. Si reinizializza *count* a 1.
- Altrimenti si incrementa *count*. La variabile *count* tiene traccia del numero di token relativi ad una singola nota.

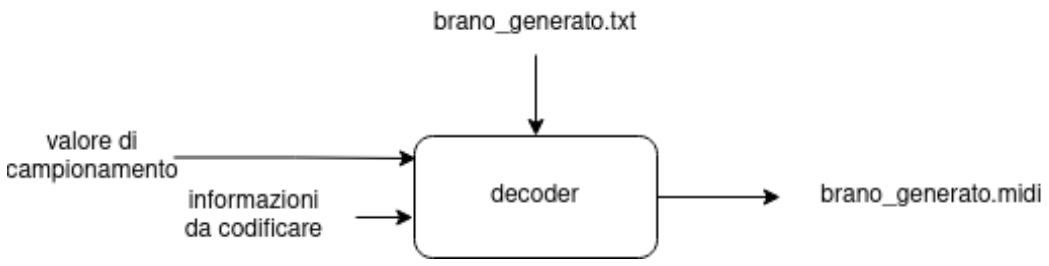


Figura 3.4: Decoder

Capitolo 4

Conclusioni

Il lavoro presentato è un semplice generatore di musica simbolica, che permette la creazione di una linea armonica e di una melodica partendo da un *seed* iniziale. Il dataset è stato creato ad *hoc* da una collezione di brani che presentassero le informazioni necessarie al raggiungimento dell'obiettivo del progetto. Sono state sperimentate diverse configurazioni e codifiche del dataset per cercare di individuare quale di queste fornisse i risultati migliori. In generale, si è potuto osservare la presenza di alcuni risultati interessanti rappresentati da frasi musicali, con relativi accordi di accompagnamento, godibili. Altri risultati sono apparsi ripetitivi e sicuramente poco orecchiabili.

Limitazioni del progetto

Diversi sono i limiti, ed ampi i margini di miglioramento:

- Attualmente la rete non è conscia di quello che sta generando, non avendo la nozione di inizio e fine di un brano non sta realmente generando uno spartito. Per fare ciò dovrebbe, dopo un certo numero di battute, finire la generazione attraverso, ad esempio, un carattere speciale. Il tutto è stato ridotto ad un problema su sequenze che se da un lato risulta semplificare il lavoro, dall'altro è sicuramente limitante;
- Altro aspetto che pone dei limiti alle possibilità della rete è la codifica usata: come discusso in precedenza, non è possibile generare diverse unità di tempo, così come non è possibile generare brani polifonici e note suonate all'unisono;

- Le scarse risorse di calcolo a disposizione non hanno permesso di verificare la bontà del modello sull'intero dataset.

Possibili sviluppi futuri

Partendo da quanto presentato, è possibile sperimentare seguendo e combinando diversi approcci: ad esempio, applicando tecniche già utilizzate in altri lavori, ma adattate alla codifica e al dataset qui utilizzati. Di seguito, andiamo a presentare alcuni dei possibili sviluppi del lavoro presentato.

Autoencoder

Una prima sperimentazione possibile, è l'utilizzo di un autoencoder [7]. Questo si compone di due elementi:

- *Encoder*: mappa l'input ricevuto in uno spazio di dimensione ridotta detto *latent space*;
- *Decoder*: preso il *latent space* cerca di ricostruire l'input originale.

Si vuole, quindi, fornire, come ulteriore parametro di generazione, anche il *latent space* in modo tale da avere, a parità di *seed* iniziale, sequenze che differiscono al variare di questo nuovo input. Per realizzare tale tipo di esperimento è possibile basarsi su quanto presentato in MusicVAE [13]. Allo stato attuale del lavoro, è già presente una

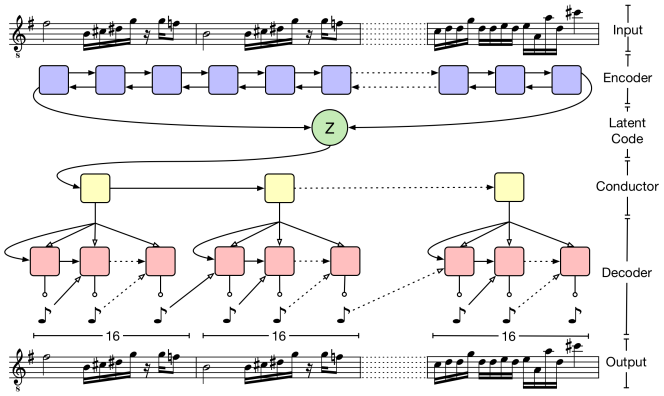


Figura 4.1: Architettura della rete

prima sperimentazione di tale approccio; infatti è stata realizzata sia l'architettura che l'addestramento della rete. Andiamo a vedere nel dettaglio le componenti di tale modello:

- *Latent space*: ottenuto come output dell'encoder, nella sperimentazione fatta viene utilizzato un valore costante;
- *Conductor*: implementato come una LSTM il cui stato iniziale risulta essere il *latent space*. Per ciascuna battuta si vuole avere un elemento del conductor, di conseguenza la lunghezza di questo dipende dal valore di campionamento e dalla *sequence_length*. Ad esempio, con un campionamento ad ottavi ed una lunghezza di sequenza pari a trentadue abbiamo un totale di quattro battute: il *conductor* sarà quindi composto da quattro elementi;
- Decoder: come per il *conductor*, viene utilizzata una LSTM. Questa riceve, oltre l'input visto in 3, un ulteriore parametro fornitogli dal *conductor*.

Il punto critico è rappresentato dal *latent space*, il quale dovrebbe essere creato in modo tale da codificare informazioni che possano essere utili, in fase generativa. Attualmente viene utilizzato come spazio latente un valore costante.

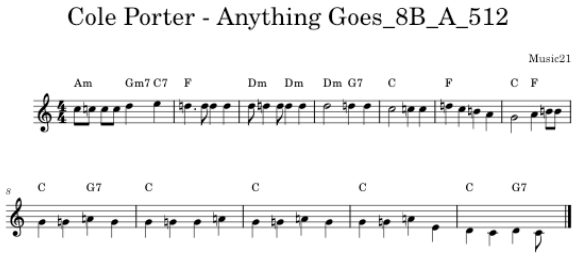


Figura 4.2: Esempio di generazione ottenuta con *latent space* costante

Mascheramento di input

La generazione attuale prevede la creazione parallela di due linee principali, quella melodica e quella armonica, partendo da altrettante linee di input. Possiamo, però, modificare tale approccio in due possibili modi:

- Generando gli accordi avendo a disposizione la melodia [3];
- Generando la melodia avendo a disposizione gli accordi.

In altre parole, si "maschera" una delle linee di input andando ad effettuare la generazione esclusivamente basandosi sulle altre a disposizione. Tale approccio, risulta interessante in quanto si riescono a creare degli accordi per brani che ne sono sprovvisti e, in maniera simmetrica, una linea melodica per degli accordi.

Temperatura

Altra tecnica molto utilizzata in ambito generativo è la temperatura. Tale meccanismo permette di aggiungere del non determinismo in fase generativa. Invece di scegliere semplicemente la classe con la più alta probabilità, viene utilizzato un *sample* di tale distribuzione, rimodellato da un valore detto appunto tempertatura. Si riescono, quindi, ad avere sequenze diverse anche utilizzando lo stesso *seed* iniziale. Inoltre, verrebbe anche mitigato il rischio di avere lunghe sequenze ripetitive.

Task

Fornire alla rete ulteriori *task* da portare a compimento è sicuramente una possibile opzione per migliorare i risultati. Come si è potuto vedere anche in questo progetto, l'aggiunta del conto delle battute ha portato ad un miglioramento della generazione. Alcuni possibili compiti da far svolgere alla rete potrebbero essere relativi alla struttura del brano, per esempio la predizione dell'inizio e della fine di strofa e ritornello. Trovare *task* aggiuntive che non risultino banali risulta essere complesso, sia per la mancanza di informazioni nei brani presenti nel dataset, sia per la conoscenza richiesta in ambito musicale.

Bibliografia

[1] Nicolas Boulanger-Lewandowski, Yoshua Bengio e Pascal Vincent. "Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription". In: *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012. URL: http://icml.cc/2012/papers/590.pdf.

[2] Jean-Pierre Briot, Gaëtan Hadjeres e François-David Pachet. *Deep Learning Techniques for Music Generation*. Springer, 2020. ISBN: 978-3-319-70162-2. DOI: 10.1007/978-3-319-70163-9. URL: https://doi.org/10.1007/978-3-319-70163-9.

[3] Kyoyun Choi et al. "Chord conditioned melody generation with transformer based decoders". In: *IEEE Access* 9 (2021), pp. 42071–42080.

[4] Ondřej Cífka, Umut Şimşekli e Gaël Richard. "Groove2Groove: One-Shot Music Style Transfer With Supervision From Synthetic Data". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 28 (2020), pp. 2638–2650. DOI: 10.1109/TASLP.2020.3019642.

[5] Shuqi Dai et al. "Controllable deep melody generation via hierarchical music structure representation". In: *Proceedings of the 22nd International Society for Music Information Retrieval Conference, ISMIR 2021, Online, November 7-12, 2021*. A cura di Jin Ha Lee et al. 2021, pp. 143–150. URL: https://archives.ismir.net/ismir2021/paper/000017.pdf.

[6] Prafulla Dhariwal et al. "Jukebox: A generative model for music". In: *arXiv preprint arXiv:2005.00341* (2020).

[7] Laurent Girin et al. “Dynamical Variational Autoencoders: A Comprehensive Review”. In: *Found. Trends Mach. Learn.* 15.1-2 (2021), pp. 1–175. DOI: 10.1561/2200000089. URL: https://doi.org/10.1561/2200000089.

[8] Aaron van den Oord et al. “Wavenet: A generative model for raw audio”. In: *arXiv preprint arXiv:1609.03499* (2016).

[9] François Pachet, Pierre Roy e Benoit Carré. “Assisted music creation with Flow Machines: towards new categories of new”. In: *Handbook of Artificial Intelligence for Music*. Springer, 2021, pp. 485–520.

[10] François Pachet, Jeff Suzda e Dani Martinez. “A Comprehensive Online Database of Machine-Readable Lead-Sheets for Jazz Standards.” In: *ISMIR*. 2013, pp. 275–280.

[11] Colin Raffel. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. Columbia University, 2016.

[12] Yi Ren et al. “Deepsinger: Singing voice synthesis with data mined from the web”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 1979–1989.

[13] Adam Roberts et al. “A hierarchical latent vector model for learning long-term structure in music”. In: *International conference on machine learning*. PMLR. 2018, pp. 4364–4373.

[14] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. 2020.