UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea triennale in INFORMATICA

Tesi Finale

MUSIC RNN

Relatore
Prof. Roberto Battiti

Laureando
Andrea Dellera

Anno accademico 2014-2015

1 Introduzione

Il mondo della musica è sempre stato qualcosa di astratto, di impalpabile. Regge ormai da decenni la figura del rocker che, grazie alla sua chitarra, anima folle sempre più grandi e si oppone alle scelte del governo corrente. Un po' più in disuso è la figura del musicista in voga negli anni d'oro della musica classica, da Mozart lo scrittore maltrattato dalla vita a Beethoven che, nonostante la sordità, riuscì a comporre un capolavoro come la *sonata al chiaro di luna*. Ma il processo creativo che sta dietro alla stesura di un brano può essere riprodotto da un qualcosa che non pensa come un computer? Può un algoritmo, una tecnica di programmazione, portare una macchina a produrre delle melodie? Lo scopo di questa tesi è di creare delle melodie utilizzando il machine learning per potare la macchina ad acquisire, ad imparare, brani già scritti per poi creare delle melodie originali e verrà perseguito facendo un confronto tra due tipologie di reti neurali: Feed Forward Netwoks (FFN) e Recurrent Neural Network (RNN).
I passi seguiti sono la definizione di una codifica per le note, la definizione delle reti e l'apprendimento di diverse sequenze di note, a partire da scale fino ad intere canzoni.

2 Codifica

Le note, estratte da partiture scritte in formato MusicXML, devono avere una codifica per essere interpretate dalla rete. Le features considerate sono cinque:

- nome della nota;
- alterazione;
- durata;
- punto di valore;
- ottava.

Di queste cinque il nome e l'alterazione sono unite in un'unica codifica visto che sono strettamente correlate tra loro così come la durata e punto di valore.

2.1 Codifica delle note

Le note nel sistema europeo sono chiamate *la, si, do, re, mi, fa, sol* che nel sistema americano corrispondono a *A, B, C, D, E, F, G*; in questo scritto verrà utilizzato il secondo, perchè più compatto e sintetico. Per codificare tutte le note servono almeno quattro bit, questo perchè oltre alle sette naturali riportate sopra abbiamo anche quelle alterate dai ♯ e dai ♭. Va ricordato però che introducendo entrambe le alterazioni nella codifica si hanno note ridondati dal punto di vista sonoro. Infatti se G ♯ e A ♭ indicano due note diverse, perchè cambia la tonalità in cui vengono usate, il suono che viene prodotto quando sono suonate è però lo stesso. Ecco perchè nella codifica utilizzeremo solo il diesis.

Codifica delle note

A	0000
A♯	0001
B	0010
C	0011
C♯	0100
D	0101
D♯	0110
E	0111
F	1000
F♯	1001
G	1010
G♯	1011

2.2 Codifica della durata

Per la durata della nota il ragionamento è analogo. La durata massima di una nota è $\frac{4}{4}$ (𝅝). Si trovano tutte nella forma $\frac{1}{2^n}$ dove $0 \leq n \leq +\infty$. Convenzionalmente però le prime tre della sequenza sono descritte come $\frac{4}{4}$ (𝅝), $\frac{2}{4}$ (𝅗𝅥), $\frac{1}{4}$ (♩). Le durate che verranno codificate arriveranno fino ad $\frac{1}{64}$. Come fatto precedentemente verrà assegnata una sequenza di bit ad ogni durata.

Codifica delle durate

$\frac{4}{4}$	000
$\frac{2}{4}$	001
$\frac{1}{4}$	010
$\frac{1}{8}$	011
$\frac{1}{16}$	100
$\frac{1}{32}$	101
$\frac{1}{64}$	110

Un'altra variabile che entra quando si parla di durata è il *punto di valore* (·). Questo strumento aumenta la durata della nota della sua metà e per codificarlo useremo un bit che sarà 0 se non c'è e 1 in caso contrario.

2.3 Codifica dell'ottava

In ciò che va a costituire una nota una parte importante è l'*ottava*. L'ottava costituisce l'altezza della nota, un'offset rispetto a quella più bassa. Si può pensarla come una somma di $n * 12 semitoni$ rispetto alla nota più bassa. Anche qui la codifica che verrà seguita è binaria. Essendoci undici ottave, $C_{-1}; ...; C_9$, dovrebbero essere usati 4 bit per la rappresentazione ma, visto che le ottave C_{-1}, C_0 e C_9 non vengono praticamente mai utilizzate verranno usati tre bit per rappresentare le otto rimanenti.

Codifica delle ottave

C_1	000
C_2	001
C_3	010
C_4	011
C_5	100
C_6	101
C_7	110
C_8	111

2.4 Codifca delle pause

Le pause sono uno strumento musicale molto comune, vengono utilizzate per dire ad uno strumento di non suonare. L'unico dato che portano è quello della durata, visto che non hanno una ottava di riferimento. Verranno codificate con *1111* nel campo di codifica della nota, in quanto denotano una nota inesistente.

2.5 Esempi di codifica

Do di $\frac{4}{4}$, della terza ottava:	01100100000
Do di $\frac{1}{4}$, della terza ottava:	01100100100

2.6 Creazione degli esempi

Per creare un esempio bisogna decidere quante note in input avrà la rete ed, in base a quello, verrà costruito. Un esempio per una rete con cinque note in input è:

Esempio

$Nota1$			$Nota2$	$Nota3$	$Nota4$	$Nota5$	$Target$
ottava	nota	durata					

3 Rete neurale ricorrente

3.1 Numero di input

Teoricamente ad una RNN basterebbe una sola nota in input per imparare una canzone intera, visto che ha memoria di quello che ha visto ed in che ordine lo ha visto, della storia passata, ma si è reso necessario ampliare l'input ad N[1] note (come viene fatto in [4]) perchè una sola non risultava sufficiente per limiti di hardware. Con più note si usano cone input la probabilità che la rete indovini il target aumenta perchè rendiamo esplicita un pezzo di storia della canzone (le N note precedenti al target). Durante la fase di scrittura del codice è stata riscontrata una relazione tra il numero di note in input e la correttezza dell'output che conferma quanto detto; ovvero se come input si dava una sequenza composta solo da una sola nota questa riusciva a predirne correttamente solo un'altra come successiva. Se quindi ci si trovava nella situazione di una doppia scelta ecco che la rete non riusciva più a distinguere le due note. Sempre facendo riferimento allo scritto citato sopra [4] si legge che la rete creata era caratterizzata dall'avere otto note come input. Non c'è un modo per decidere arbitrariamente il numero di note in input ma bisogna trovare un compromesso tra la quantità di sequenze che si possono riconoscere e la complessità del sistema.

3.2 Tipo di rete utilizzata

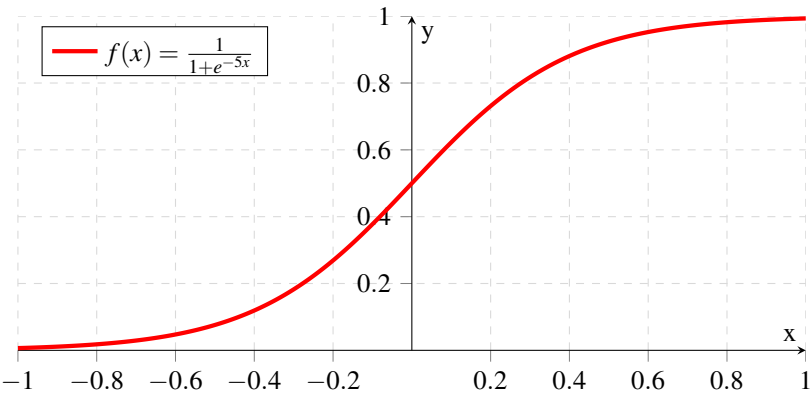
La rete neurale ricorrente è così composta:

- 1 layer di input con $Nnote * 11$ neuroni;
- 1 hidden layer con x neuroni[2];
- 1 layer di output con 11 neuroni.

C'è una full connection tra il layer di input e l'hidden layer, tra l'hidden layer e il layer di output e tra l'hidden layer e se stesso. [3] La rete neurale non ricorrente è identica solo non presenta la connessione tra l'hidden layer e se stesso.
Le funzioni utilizzate all'interno dei neuroni sono di due tipi:

- Sigmoidale;
- Long Short Term Memory (LSTM).

Un neurone di tipo LSTM è un neurone che ricorda i valori per un determinato periodo mentre uno sigmoidale implementa una funzione simile a quella sottostante.



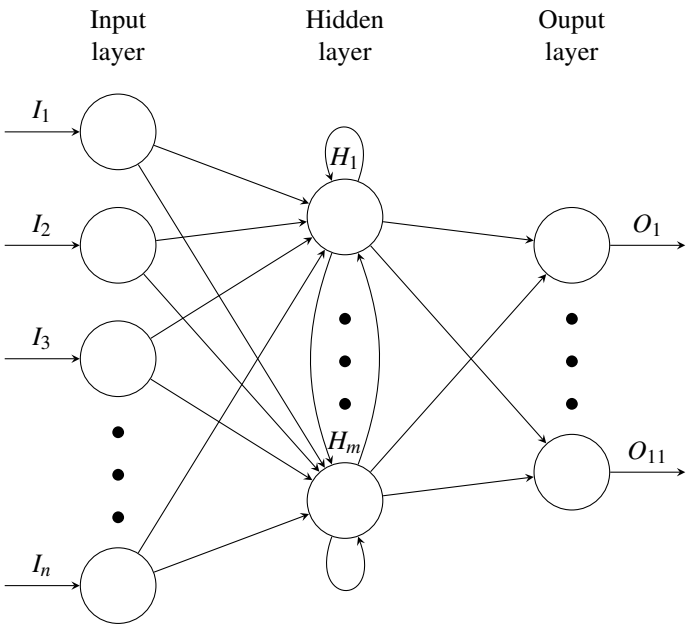
A seconda del tipo di rete utilizzata cambiano i tipi di neuroni (per esempio quelli LSTM possono essere utilizzati solo con RNN). Quelli utilizzati nel programma sono:

Tipi di neurone

	FFN	**RNN**
Hidden Neuron	Sigmoidal	LSTM
Output Neuron	Sigmoidal	Sigmoidal

Il seguente grafo rappresenta la RNN, con la connessione ricorrente trai i neuroni presenti nell'hidden layer.

[1] Il numero di note in input viene definito appropriatamente in ogni esperimento;
[2] Il numero esatto di neuroni presenti nell'hidden layer viene definito appropriatamente in ogni esperimento;
[3] I pesi che portano dal nodo A al nodo B possono essere diversi di quelli che portano dal nodo B al nodo A;


Input
layer
Hidden
layer
Ouput
layer
I_1
I_2
I_3
I_n
H_1
H_m
O_1
O_{11}

4 Train della rete

Per allenare la RNN è stato scelto l'algoritmo di Back Propagation, associato a Gradient Descent. L'aggiornamento dei pesi viene fatto in batchlearning, ovvero alla fine di tutte le epoch, per la rete neurale ricorrente e on-line, cioè alla fine di ogni epoch, per la rete neurale non ricorrente. Sono stati utilizzati diversi valori di learning rate e di momentum, due parametri che influiscono su come la rete è allenata; il primo è una costante che decide la quantità di cambiamento sui vecchi pesi all'interno della rete mentre il secondo comporta una ulteriore adattamento dei pesi in base ai cambiamenti precedenti e permette di avvicinarsi più velocemente al minimo.
Dopo diverse prove sono stati utilizzati valori i seguenti valori:

	FFN	**RNN**
Learning Rate	0.3	0.1
Momentum	0.9	0.3

Per il valore del learning rate è stata adottata la tecnica Bold Driver [1] che ne cambia la quantità a seconda di come procede l'apprendimento da parte della rete. La funzione di errore da minimizzare durante il la fase di train è quello dei minimi quadrati. Per allenare la rete è stata utilizzata la tecnica di k-fold cross validation [2], dividendo il database iniziale in dieci parti ed utilizzandone nove per il train e una per la validazione. Il codice riportato è la funzione di train utilizzata per allenare la rete; è stata modificata dal codice originale per adattarsi meglio al programma e per implementare la k-fold cross validation.

Listing 1: Train Until Convergence

```
trainingData = datasetTrain
validationData = datasetTest
self.ds = trainingData
bestweights = self.module.params.copy()
bestverr = self.testOnData(validationData)
trainingErrors = []
validationErrors = [bestverr]
while True:
        trainingErrors.append(self.train())
        validationErrors.append(self.testOnData(validationData))
        if validationErrors[-1] < bestverr:
                # one update is always done
                bestverr = validationErrors[-1]
                bestweights = self.module.params.copy()

        if maxEpochs is not None and epochs >= maxEpochs:
                self.module.params[:] = bestweights
                break
        epochs += 1

        if len(validationErrors) >= continueEpochs * 2:
                old = validationErrors[-continueEpochs * 2:-continueEpochs]
                new = validationErrors[-continueEpochs:]
                if min(new) > max(old):
                        self.module.params[:] = bestweights
                        break
return trainingErrors, validationErrors
```

La funzione salva lo stato dei pesi della rete quando, in fase di validazione, viene ottenuto un errore minore al minimo. Inoltre se l'errore comincia a salire nuovamente controlla di quanto lo fa e, in caso, blocca il train settando i pesi a quelli che davano errore di validazione minore e ritorna la progressione degli errori in fase di train e validazione. Per concludere questa sezione è da notare che è stata valutata l'ipotesi di usare l'algoritmo di Back Propagation Through Time [5], versione particolare di Back Propagation adattata alle RNN, ma non era presente nel pacchetto utilizzato per implementare le reti [3] e la complessità dell'implementazione l'ha resa infattibile.

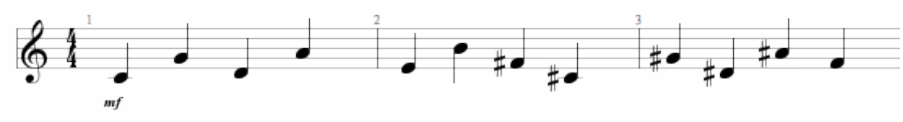
5 Esperimenti

5.1 Esperimento 1 - Imparare sequenze di note semplici

Prima di descrivere l'esperimento è necessario dare alcune informazioni sulla rete:

- sono state utilizzate cinque note per comporre l'input;
- il numero di neuroni nell'hidden layer è pari a venti.

La FNN impara sequenze più semplici di note più velocemente e più correttamente rispetto a quella ricorrente. Dalle prove fatte (arpeggi sulla tonalità di do maggio re e scala su due ottave di do maggiore) la FFN è stata capace di ricondursi più velocemente alle sequenze imparate. La RNN invece impara con errori le sequenze, risultato che contribuisce a non saper ricreare correttamente gli esempi.
Nel primo file le sequenze di note erano disposte in una forma a cerchio[4].

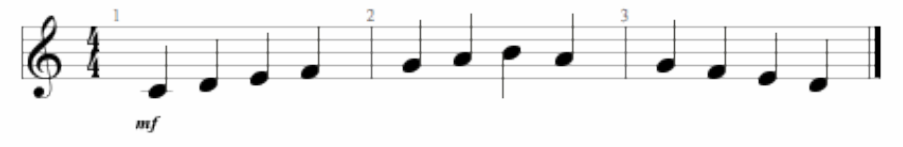


Gli errori medi ottenuti in fase di train e validazione sono:

Errori esempio a cerchio

	FFN	**RNN**
Train	0.000423	0.003124
Validazone	0.015625	0.027443

Gli errori della FFN sono più bassi rispetto alla RNN ma è accettabile visto che veniva chiesto di imparare una semplice progressione di note e non della musica più complessa. È possibile vedere in entrambi i casi che alla prima nota generata *sbagliata* l'errore viene poi esteso a tutte le note successive.
Nel secondo file invece le note erano disposte come a formare una figura ad otto.



Di seguito sono riportati gli errori in fase di train e validazione.

Errori esempio a otto

	FFN	**RNN**
Train	0.000759	0.002412
Validazone	0.001569	0.006690

Inoltre, in questo esempio, si è notato come un errore nella generazione delle note influenzi maggiormente la RNN visto che tiene in considerazione la storia passata. Anche in questo caso, come nel precedente, la FFN dà maggiore precisione quando si generano le note.

5.2 Esperimento 2 - Imparare una canzone

Quando si è trattato di fare imparare sequenze di note derivate da una canzone il processo di apprendimento è risultato molto più complesso ed infatti sono stati fatti degli accorgimenti alla rete:

- sono state aumentate le note in input, da cinque ad otto;
- è stato aumentato il numero di neuroni nell'hidden layer, da venti a cinquanta.

Questo perchè le sequenze di note erano maggiori in numero rispetto all'esperimento precedente ma anche perchè erano molto più varie. La RNN si è dimostrata più difficile da allenare, ha richiesto un numero maggiore di epochs, ma le note prodotte erano più esatte, proprio per la capacità di tenere in considerazione la storia delle note (cosa che la FFN non fa), più varie rispetto alla rete neurale non ricorrente che cercava di riprodurre il brano usato per il train ma nient'altro di più.
Il primo brano preso in esame è *Fly Me To The Moon*, il secondo invece è *Kathy's Song*.
Gli errori di train e validazione sono stati:

[4] può non sembra un cerchio ma rispetta il *ciclo delle quinte*, una progressione di note in cui ad ognuna viene fatta seguire la sua quinta naturale

Errori primo brano

	FFN	**RNN**
Train	0.580332	0.058844
Validazone	1.387929	0.892960

Errori secondo brano

	FFN	**RNN**
Train	1.745513	0.032295
Validazone	1.740951	0.985849

5.3 Esperimento 3 - Imparare più di una canzone

Quando si è trattato di far imparare alle reti più di una canzone sono sorti alcuni problemi perchè queste in qualche modo dimenticavano le sequenze già imparate a favore delle ultime viste; inoltre si riscontravano problemi a livello del codice utilizzato per implementare le reti [3] che non si è riusciti ad identificare ma evidentemente presenti visto l'andamento degli errori di train e validazione. DATI

6 Conclusioni

A conclusione del lavoro fatto si è giunti alla conclusione che le Feed Forward Networks possano imparare semplici sequenze di note come scale o arpeggi ma non riescano a dare un significato a ciò che generano, a creare della musica. Le Recurrent Neural Networks invece hanno la capacità di generare note in modo più complesso e vario tenendo conto della storia passata ma, arrivati ad un certo numero di note, perdono il contesto creando si note che si possono definire giuste in quanto appartenenti ad una certa tonalità o di una certa durata ma che all'orecchio umano risultano scollegate l'una dall'altra. Quindi neanche a questo tipo di rete si può chiedere di generare musica in modo completamente esatto.

7 Lavori futuri

Questo progetto è stato sviluppato come lavoro di tesi e tirocinio. Tuttavia verrà portato avanti, implementando le reti utilizzando un proprio codice e non quello di una libreria esterna ed il goal del progetto sarà creare diverse reti che creino musica di diversi generi.

References

[1] Roberto Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex systems*, 3(4):331–342, 1989.

[2] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[3] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. Pybrain. *The Journal of Machine Learning Research*, 11:743–746, 2010.

[4] Peter M Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, pages 27–43, 1989.

[5] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.