Music Generation using Recurrent Neural Networks

Kedarnath P

Advisor: **Prof. Dr. Subhasish Dhal**

Department of Computer Science and Engineering
Indian Institute of Information Technology Guwahati

This dissertation is submitted for the degree of
Bachelors of Technology

IIIT Guwahati May 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Kedarnath P
May 2019

Acknowledgements

I would like to thank my advisor Prof. Dr. Subhasish Dhal for his guidance and also for giving me freedom of opting the topic for my B.tech Project due to which I was able to make the most use of it and show my keen interest in the field of Deep learning, where I want my career to be in. I also thank few online resources which helped me to gain useful knowledge and complete my thesis.

Abstract

Our goal is to be able to build a generative model from a deep neural network architecture to try to create music that has both harmony and melody and is passable as music composed by humans. Many approaches are being used for music generation to this date. Many fields are revolutionized by the recent deep learning breakthroughs. Music generation is no exception. In this work we employ two different variants of Recurrent Neural Network (RNN) for music generation with both single instrument(Monophonic music) and Polyphonic Music(Multi-instrument music). And then we compare the human acceptance or the music generated by these two RNN- based models. RNNs are renowned for modeling sequential data and have become popular in the deep learning community for many applications. They are powerful models that have achieved excellent performance on difficult learning tasks having temporal dependencies.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Computer-Based Music Systems

The first music generated by computer appeared in 1957. It was a 17 seconds long melody named “The Silver Scale” by its author Newman Guttman and was generated by a software for sound synthesis named Music I, developed by Mathews at Bell Laboratories. The same year, “The Illiac Suite” was the first score composed by a computer[7]. It was named after the ILLIAC I computer at the University of Illinois at Urbana-Champaign (UIUC) in the United States. The human “meta-composers” were Hiller and Isaacson, both musicians and scientists. It was an early example of algorithmic composition, making use of stochastic models (Markov chains) for generation as well as rules to filter generated material according to desired properties. In the domain of sound synthesis, a landmark was the release in 1983 by Yamaha of the DX 7 synthesizer, building on groundwork by Chowning on a model of synthesis based on frequency modulation (FM). The same year, the MIDI6 interface was launched, as a way to interoperate various software and instruments (including the Yamaha DX 7 synthesizer). Another landmark was the development by Puckette at IRCAM of the Max/MSP real-time interactive processing environment, used for real-time synthesis and for interactive performances. Regarding algorithmic composition, in the early 1960s Iannis Xenakis explored the idea of stochastic composition[15], in his composition named “Atrees” in 1962. The idea involved using computer fast computations to calculate ´ various possibilities from a set of probabilities designed by the composer in order to generate samples of musical pieces to be selected. In another approach following the initial direction of “The Iliac Suite”, grammars and rules were used to specify the style of a given corpus or more generally tonal music theory. An example is the generation in the 1980s by Ebcioglu’s composition software named CHORAL of a four-part chorale in the style of Johann Sebastian ˘ Bach, according to over 350 handcrafted rules [2]. In the late 1980s David Cope’s system named Experiments in Musical Intelligence (EMI) extended that approach with the capacity to learn from a corpus of scores of a composer to create its own grammar and database of rules [1]. Since then, computer music has continued developing for the general public, if we consider, for instance, the GarageBand music composition and production application for Apple platforms (computers, tablets and cellphones), as an offspring of the initial Cubase sequencer software, released by Steinberg in 1989.

1.2 Deep Learning

The motivation for using deep learning (and more generally machine learning techniques) to generate musical content is its generality. As opposed to handcrafted models, such as grammar-based [13] or rule-based music generation systems [2], a machine learning-based generation system can be agnostic, as it learns a model from an arbitrary corpus of music. As a result, the same system may be used for various musical genres. Therefore, as more large scale musical datasets are made available, a machine learning-based generation system will be able to automatically learn a musical style from a corpus and to generate new musical content. As stated by Fiebrink and Caramiaux [4], some benefits are • it can make creation feasible when the desired application is too complex to be described by analytical formulations or manual brute force design, and • learning algorithms are often less brittle than manually designed rule sets and learned rules are more likely to generalize accurately to new contexts in which inputs may change. Moreover, as opposed to structured representations like rules and grammars, deep learning is good at processing raw unstructured data, from which its hierarchy of layers will extract higher level representations adapted to the task.

1.3 Present and Future

As we will see, the research domain in deep learning-based music generation has turned hot recently, building on initial work using artificial networks to generate music (e.g., the pioneering CONCERT system developed in 1994 [12]), while creating an active stream of new ideas and challenges made possible thanks to the progress of deep learning. Let us also note the growing interest by some private actors in the computer-aided generation of artistic content, with the creation by Google in June 2016 of the Magenta research project [3] and the creation by Spotify in September 2017 of the Creator Technology Research Lab (CTRL) [5].

1.4 Objective

Motivated by the recent success of Recurrent neural networks in generating new sequences and patterns, our objective is to generate music automatically using Recurrent Neural Network(RNN).We do not necessarily have to be a music expert in order to generate music. Even a non expert can generate a decent quality music using RNN.We all like to listen interesting music and if there is some way to generate music automatically, particularly decent quality music then it's a big leap in the world of music industry. Our task here is to take some existing music data then train a model using this existing data. The model has to learn the patterns in music that we humans enjoy. Once it learns this, the model should be able to generate new music for us. It cannot simply copy-paste from the training data. It has to understand the patterns of music to generate new music. We here are not expecting our model to generate new music which is of professional quality, but we want it to generate a decent quality music which should be melodious and good to hear. Now, what is music? In short music is nothing but a sequence of musical notes. Our input to the model is a sequence of musical events/notes. Our output will be new sequence of musical events/notes. But what type of music is to be generated? This can be:
• ***Monophonic music***, i.e. a sequence of notes, with a single voice (instrument or vocal), which can be monodic (at most one note at the same time).
• ***Polyphonic music***, with multiple voices (intended for more than one voice or instrument, i.e. more than one note at the same).

In our Experiment, first we generate simple monophonic music and later we will also generate Polyphonic Musical sequences.

Chapter 2

Representation of Music

Representation, is about the way the musical content is represented. The choice of representation and its encoding is tightly connected to the configuration of the input and the output of the architecture, i.e. the number of input and output variables as well as their corresponding types. We will see that, although a deep learning architecture can automatically extract significant features from the data, the choice of representation may be significant for the accuracy of the learning and for the quality of the generated content. For example, in the case of an audio representation, we could use a spectrum representation (computed by a Fourier transform) instead of a raw waveform representation. In the case of a symbolic representation, we could consider (as in most systems) enharmony.

2.1 Main Concepts

2.1.1 *Note*

In a symbolic representation, a note is represented through the following main features, and for each feature there are alternative ways of specifying its value:
• Pitch – specified by – frequency, in Hertz (Hz)
– vertical position (height) on a score; or
– pitch notation, which combines a musical note name and a number (usually notated in subscript) identifying the pitch class octave. An example is A4, which corresponds to A440 – with a frequency of 440 Hz – and serves as a general pitch tuning standard.
• Duration – specified by
– absolute value, in milliseconds (ms).

• Dynamics – specified by
– absolute and quantitative value, in decibels (dB)
– qualitative value, an annotation on a score about how to perform the note.

2.1.2 *Rest*

Rests are important in music as they represent intervals of silence allowing a pause for breath14. A rest can be considered as a special case of a note, with only one feature, its duration, and no pitch or dynamics. The duration of a rest may be specified by:
• absolute value, in milliseconds (ms); or
• relative value, notated as a division or a multiple of a reference rest duration, the full rest having the same duration as a full note.

2.1.3 *Rhythm*

Rhythm is fundamental to music. It conveys the pulsation as well as the stress on specific beats, indispensable for dance! Rhythm introduces pulsation, cycles and thus structure in what would otherwise remain a flat linear sequence of notes.

Beat and Meter

A beat is the unit of pulsation in music. Beats are grouped into measures, separated by bars. The number of beats in a measure as well as the duration between two successive beats constitute the rhythmic signature of a measure and consequently of a piece of music. This time signature is also often named meter. It is expressed as the fraction *number of Beats/BeatDuration*, where
• *number of Beats* is the number of beats within a measure; and
• *beatDuration* is the duration between two beats. As with the relative duration of a note or of a rest, it is expressed as a division of the duration of a full note .

More frequent meters are 2/4, 3/4 and 4/4. For instance, 3/4 means 3 beats per measure, each one with the duration of a quarter. . It is the rhythmic signature of a Waltz. The stress (or accentuation) on some beats or their subdivisions may form the actual style of a rhythm for music as well as for a dance, e.g., ternary Jazz versus binary rock.

Levels of Rhythm Information

We may consider three different levels in terms of the amount and granularity of information about rhythm to be included in a musical representation for a deep learning architecture:

• *None* – only notes and their durations are represented, without any explicit representation of measures. This is the case for most systems.
• *Measures* – measures are explicitly represented.
• *Beats* – information about meter, beats, etc. is included.

2.2 Format

The format is the language (i.e. grammar and syntax) in which a piece of music is expressed (specified) in order to be interpreted by a computer.

2.2.1 MIDI

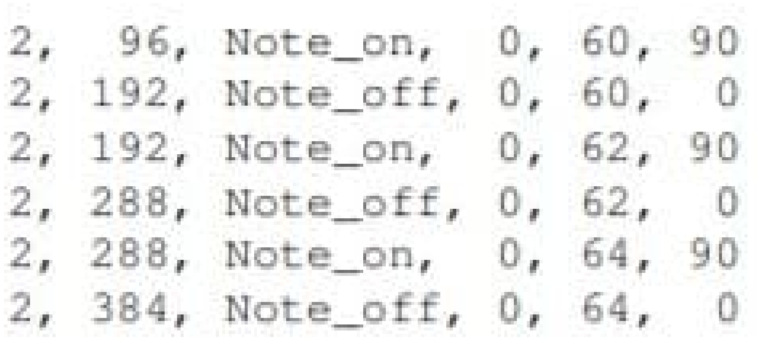
Musical Instrument Digital Interface (MIDI) is a technical standard that describes a protocol, a digital interface and connectors for interoperability between various electronic musical instruments, softwares and devices [11]. MIDI carries event messages that specify real-time note performance data as well as control data. We only consider here the two most important messages for our concerns:

• *Note on* – to indicate that a note is played. It contains
– a channel number, which indicates the instrument or track, specified by an integer within the set 0,1,... ,15;
– a MIDI *note number*, which indicates the note pitch, specified by an integer within the set 0,1,... ,127; and
– a *velocity*, which indicates how loud the note is played, specified by an integer within the set 0,1,... ,127.
An example is “Note on, 0, 60, 50” which means “On channel 1, start playing a middle C with velocity 50”;

• *Note off* – to indicate that a note ends. In this situation, the velocity indicates how fast the note is released.
An example is “Note off, 0, 60, 20” which means “On channel 1, stop playing a middle C with velocity 20”. Each note event is actually embedded into a track chunk, a data structure containing a *delta-time* value which specifies the timing information and the event itself. A delta-time value represents the time position of the event and could represent:

• a *relative metrical time* – the number of ticks from the beginning. A reference, named the

division and defined in the file header, specifies the number of ticks per quarter note or
• an *absolute time* – useful for real performances, not detailed here, see [11]. An example of an excerpt from a MIDI file (turned into readable ascii) and its corresponding score are shown in Figures 2.1 and 2.2. The division has been set to 384, i.e. 384 ticks per quarter note (which corresponds to 96 ticks for a sixteenth note .



```
2,  96, Note_on,  0, 60, 90
2, 192, Note_off, 0, 60,  0
2, 192, Note_on,  0, 62, 90
2, 288, Note_off, 0, 62,  0
2, 288, Note_on,  0, 64, 90
2, 384, Note_off, 0, 64,  0
```

Fig. 2.1 Excerpt from a MIDI file

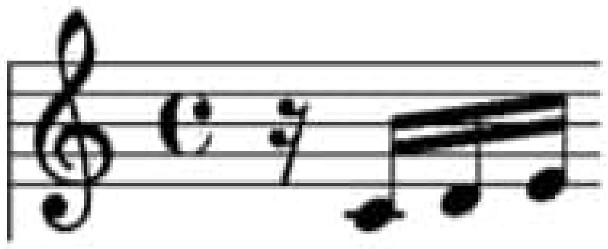


Fig. 2.2 Score corresponding to the MIDI excerpt

2.2.2 *Piano Roll*

The piano roll representation of a melody (monophonic or polyphonic) is inspired from automated pianos (see Figure 4.8). This was a continuous roll of paper with perforations (holes) punched into it. Each perforation represents a piece of note control information, to trigger a given note. The length of the perforation corresponds to the duration of a note. In the other dimension, the localization of a perforation corresponds to its pitch.

An example of a modern piano roll representation (for digital music systems) is shown in Figure 2.3. The x axis represents time and the y axis the pitch.

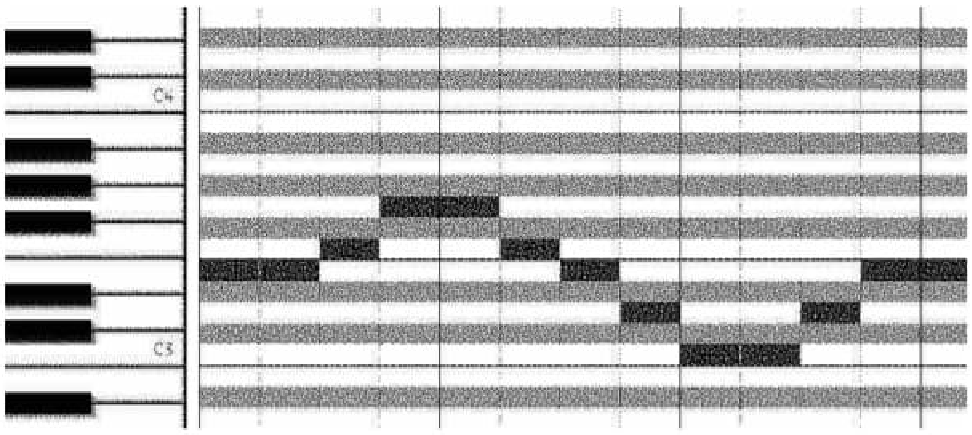


Fig. 2.3 Example of symbolic piano roll. Reproduced from [6] with the permission of Hao Staff Music Publishing (Hong Kong) Co Ltd.

There are several music environments using piano roll as a basic visual representation, in place of or in complement to a score, as it is more intuitive than the traditional score notation. An example is Hao Staff piano roll sheet music [6], shown in Figure 2.3 with the time axis being horizontal rightward and notes represented as green cells. Another example is tabs, where the melody is represented in a piano roll-like format [8], in complement to chords and lyrics. Tabs are used as an input by the MidiNet system.

The piano roll is one of the most commonly used representations, although it has some limitations. An important one, compared to MIDI representation, is that there is no note off information. As a result, there is no way to distinguish between a long note and a repeated short note. For a more detailed comparison between MIDI and piano roll, see [9] and [14].

2.2.3 *Text (ABC notation)*

Melody

A melody can be encoded in a textual representation and processed as a text. A significant example is the ABC notation [182], a de facto standard for folk and traditional music. Figures 2.4 and 2.5 show the original score and its associated ABC notation for a tune named “A Cup of Tea”, from the repository and discussion platform The Session [10].

The first six lines are the header and represent metadata: T is the title of the music, M is the meter, L is the default note length, K is the key, etc. The header is followed by the main text representing the melody. Some basic principles of the encoding rules of the ABC notation are as follows:

• the pitch class of a note is encoded as the letter corresponding to its English notation, e.g., A for A or La;
• its pitch is encoded as following: A corresponds to A_4, a to an A one octave up and a’ to an A two octaves up.

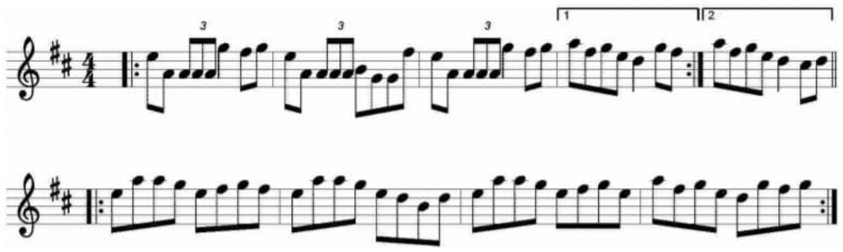


Fig. 2.4 Score of “A Cup of Tea” (Traditional). Reproduced from The Session [10] with permission of the manager

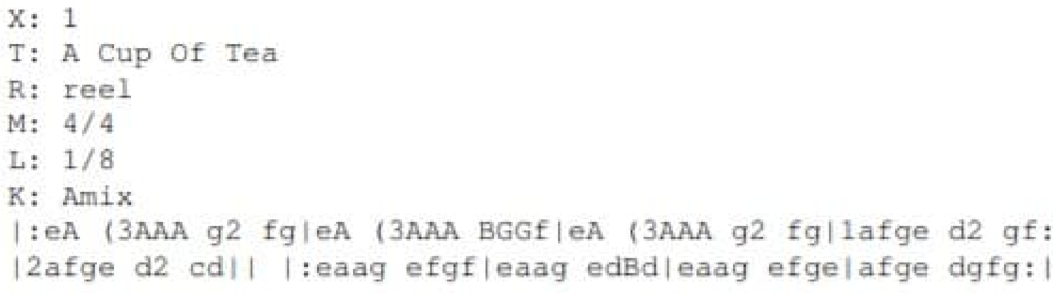
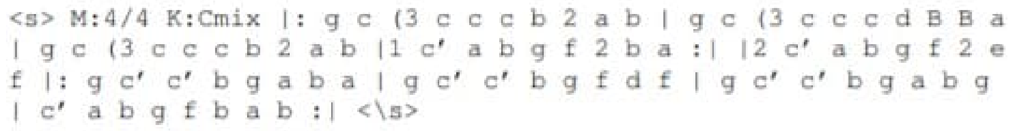

```
X: 1
T: A Cup Of Tea
R: reel
M: 4/4
L: 1/8
K: Amix
|:eA (3AAA g2 fg|eA (3AAA BGGf|eA (3AAA g2 fg|1afge d2 gf:
|2afge d2 cd|| |:eaag efgf|eaag edBd|eaag efge|afge dgfg:|
```

Fig. 2.5 ABC notation of “A Cup of Tea”. Reproduced from The Session [10] with permission of the manager

- the duration of a note is encoded as following: if the default length is marked as 1/8 (i.e. an eighth note – the case for the “A Cup of Tea” example), a corresponds to an eighth note , a/2 to a sixteenth note and a2 to a quarter note.; and
- measures are separated by “|” (bars).

ABC notation can only represent monophonic melodies. In order to be processed by a deep learning architecture, the ABC text is usually transformed from a character vocabulary text into a *token* vocabulary text in order to properly consider concepts which could be noted on more than one character, e.g., g2. Sturm *et al.*'s experiment, uses a token-based notation named the folk-rnn notation. A tune is enclosed within a "<s>" begin mark and an "<\s>" end mark. Last, all example melodies are transposed to the same C root base, resulting in the notation of the tune "A Cup of Tea" shown in Figure 2.6.



```
<s> M:4/4 K:Cmix |: g c (3 c c c b 2 a b | g c (3 c c c d B B a
| g c (3 c c c b 2 a b |1 c' a b g f 2 b a :| |2 c' a b g f 2 e
f |: g c' c' b g a b a | g c' c' b g f d f | g c' c' b g a b g
| c' a b g f b a b :| <\s>
```

Fig. 2.6 Folk-rnn notation of "A Cup of Tea". Reproduced from with the permission of the authors

Note: In our experiment we would be using the ABC and MIDI format for representing the music.

Chapter 3

Foundation and Related Work

This chapter contains a closer look on basic concepts and related published work that are essential for the work in this thesis. The first section introduces Recurrent neural Networks(RNN) and its variants. It also explains their relation to our models in computational neuroscience.

3.1 Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

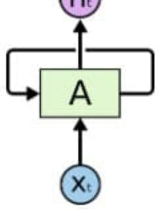


Fig. 3.1 Recurrent Neural Networks have loops

In the figure 3.1, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

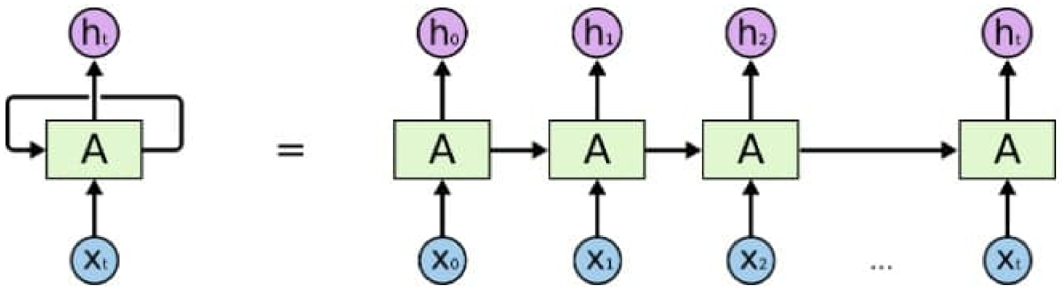


Fig. 3.2 An unrolled recurrent neural network

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning,etc. The Unreasonable Effectiveness of Recurrent Neural Networks. But they really are pretty amazing.

Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It's these LSTMs that this essay will explore.

3.1.1 The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

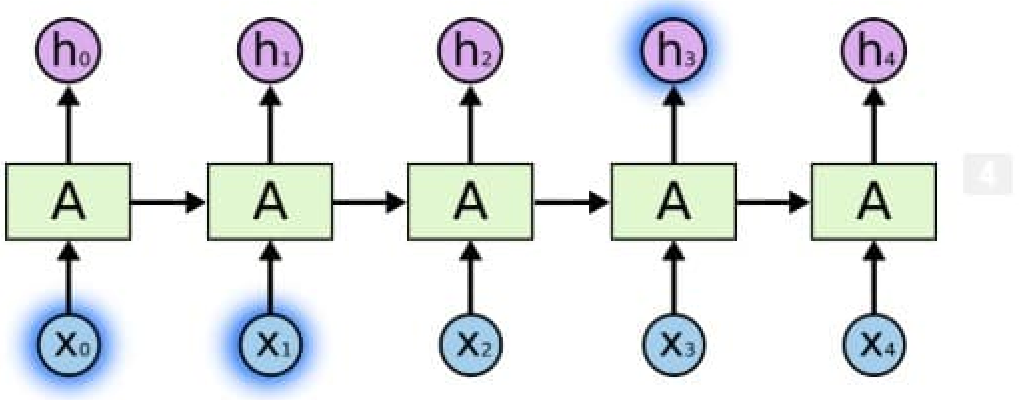


Fig. 3.3

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

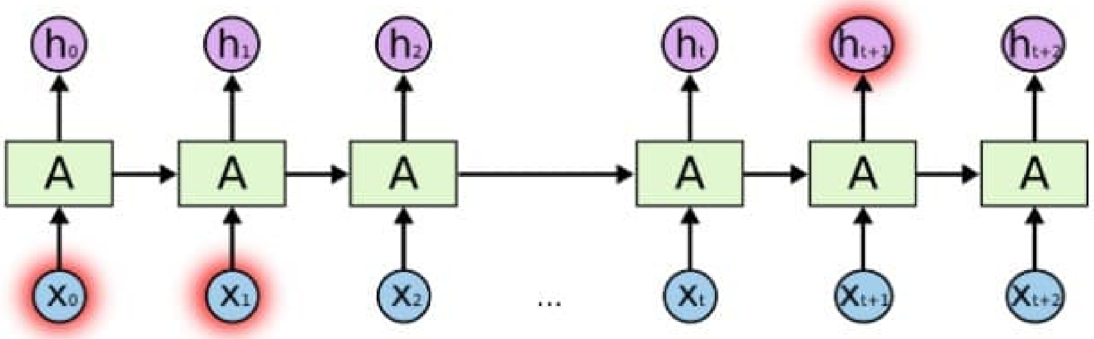


Fig. 3.4

In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by

Hochreiter (1991) [German] and *Bengio, et al.* (1994), who found some pretty fundamental reasons why it might be difficult. But, LSTMs don't have this problem!

3.2 LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by *Hochreiter* and *Schmidhuber* (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single *tanh* layer.

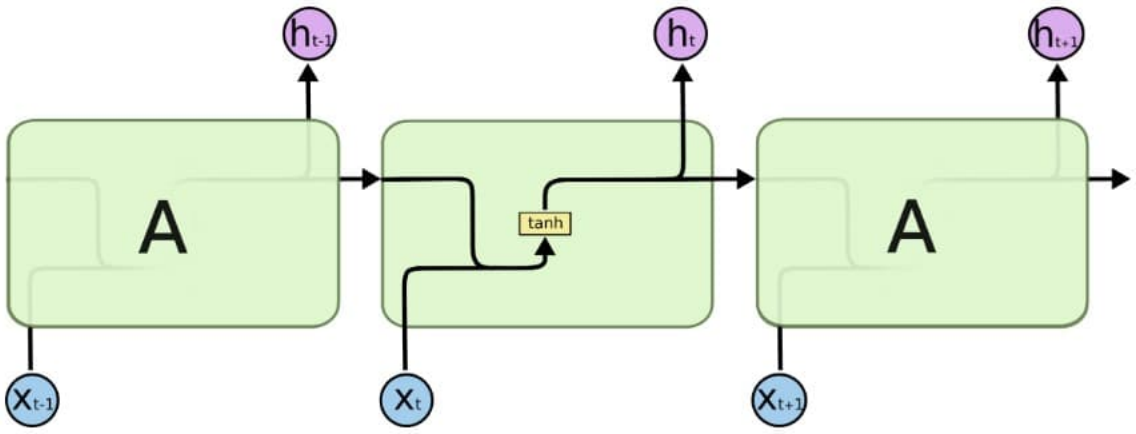


Fig. 3.5 The repeating module in a standard RNN contains a single layer

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

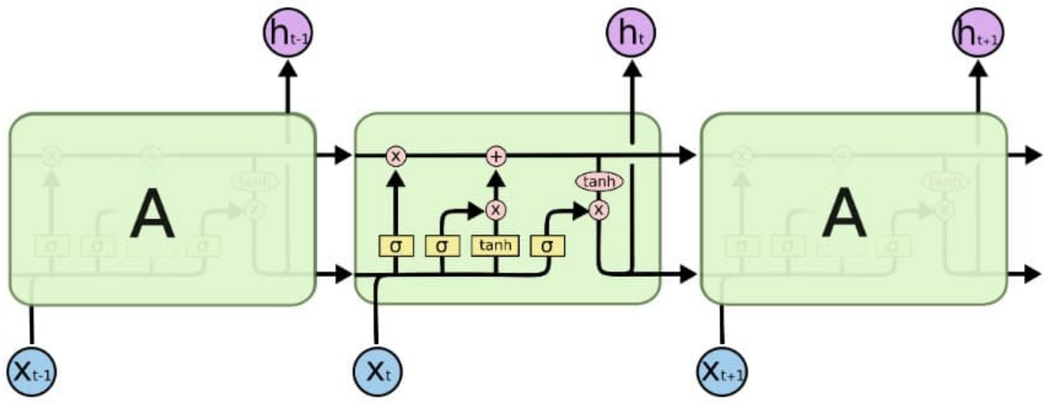


Fig. 3.6 The repeating module in an LSTM contains four interacting layers.

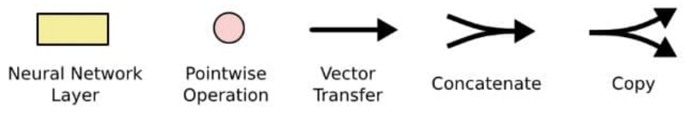


Fig. 3.7

In the figure 3.7, each line carries an entire vector, from the output of one node to the inputs of others. The circles represent pointwise operations, like vector addition, while the rectangular boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

3.2.1 The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

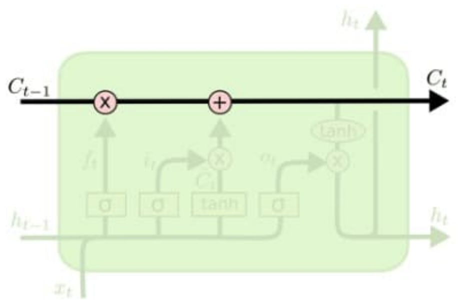


Fig. 3.8

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

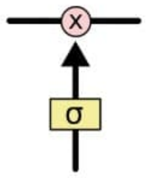


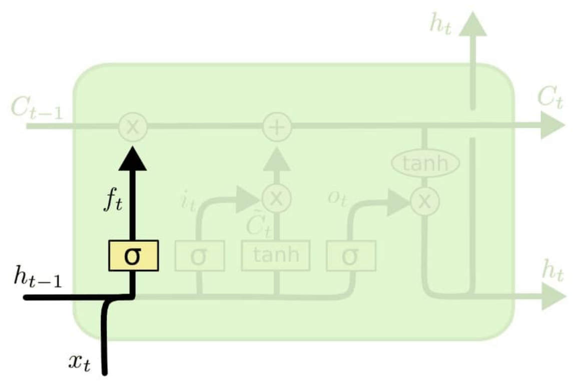
Fig. 3.9

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!” An LSTM has three of these gates, to protect and control the cell state.

3.2.2 Step-by-Step LSTM Walkthrough

The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at h_{t-1} and x_t and outputs a number between 0 and 1 for each number in the cell state C_{t1}. A 1 represents “completely keep this” while a 0 represents “completely get rid of this.”

Let’s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

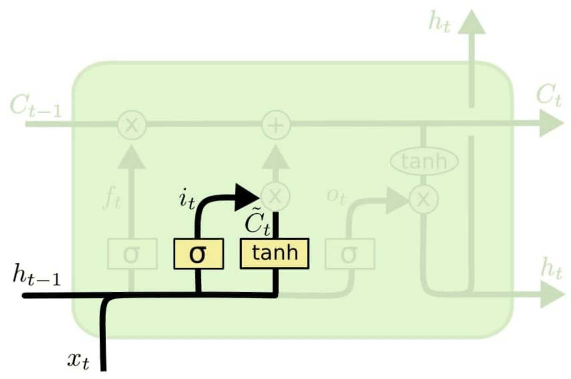


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Fig. 3.10

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

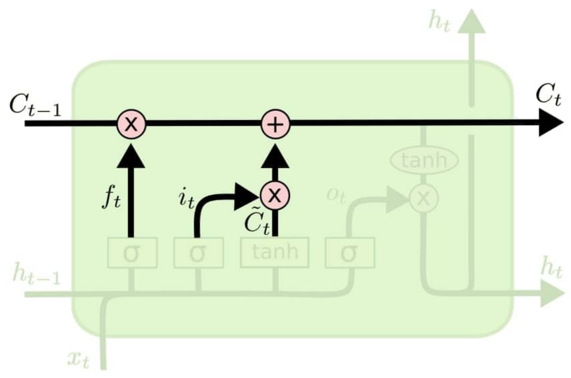
In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Fig. 3.11

It's now time to update the old cell state, C_{t1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add it $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

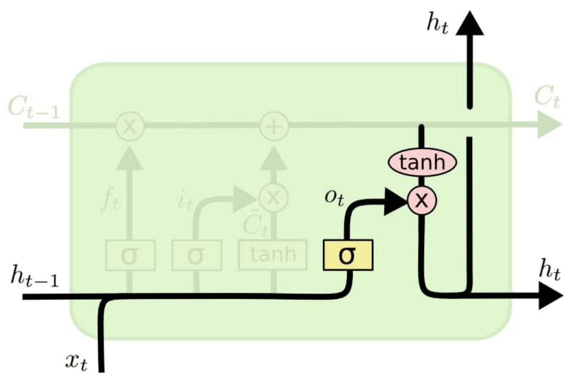


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Fig. 3.12

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between 1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Fig. 3.13

3.3 Char-RNN Architecture

Since our music is a sequence of characters therefore the obvious choice will be RNN or variations of RNN like LSTMs or GRUs which can process sequence information very well by understanding the patterns in the input.

There is a special type of RNN called char-RNN. Now our music is a sequence of characters. We will feed one after the other character of the sequence to RNN and the output will be the next character in the sequence. So, therefore, the number of output will be equal to the number of inputs. Hence, Many-to-Many RNNs are used here, where number of output is equal to the number of input.

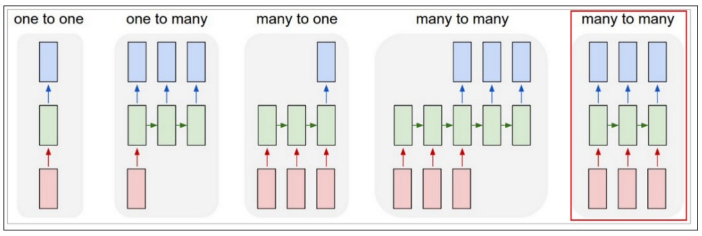


Fig. 3.14 Many-to-Many RNN

In the figure 3.14, many-to-many RNN with equal number of inputs and outputs are shown on right-side which is in the red box. Here, each green rectangle(middle) is RNN unit which is a repeating structure. A more detailed image of RNN is shown below.

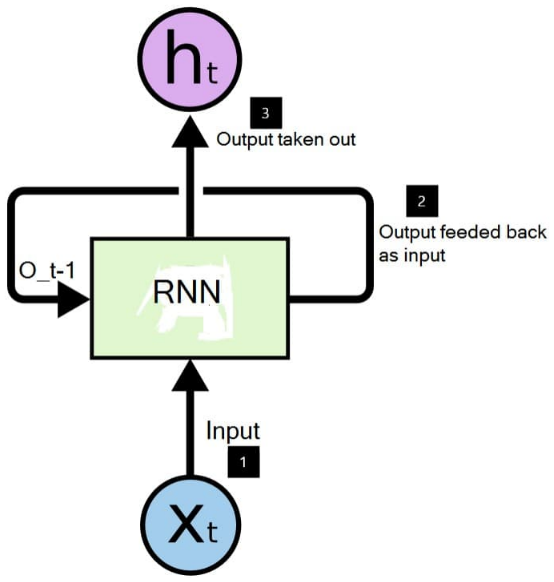


Fig. 3.15 Char-RNN

In the figure 3.15, X_t is a single character at time 't' which is given as an input to RNN unit. Here, O_{t1} is an output of the previous time 't-1' character which is also given as an input to RNN at time 't'. It then generates output 'h_t'. This output 'h_t' will again be given as input to RNN as 'O_{t1}' in next time step.

This output 'h_t' will be a next character in sequence. Let say our music is represented as [a, b, c, a, d, f, e,...]. Now, we will give first character 'a' as an input and expects RNN to generate 'b' as an output. Then in next time-step we will give 'b' as an input and expects 'c' as an output. Then in next time-step we will give 'c' as an input and expects 'a' as an output and so on. We will train our RNN such that it should output next character in the sequence. This is how it will learn whole sequence and generate new sequence on its own.

This will keep on going till we feed all of our inputs. Since, our music is a combination of many characters and our output is one of those characters so it can be thought of as multi-class classification problem. Here, we will use "Categorical Cross-Entropy" as a loss function which is also known as "Multi-Class log-loss". In the last layer we will keep "Softmax" activations. The number of "Softmax" activation units in last layer will be equal to the number of all unique characters in all of the music in train data. Each RNN can be a LSTM which contains 'tanh' activation unit at input-gate which is a differentiable function. Therefore, this RNN structure can be trained using back-propagation and we keep on iterating it using “Adam” optimizer till we converge. At the end, our RNN will be able to learn sequence and patterns of all the musical notes that are given to it as input during training.

3.4 Deep Biaxial-RNN Architecture

In the interest of theoretical continuations of our character-RNN success, we define a structure called a deep biaxial-RNN. The idea is that we have two axes (and one pseudo-axis): there is the time axis and the note axis (and the direction-of-computation pseudo-axis). Each recurrent layer transforms inputs to outputs, and also sends recurrent connections along one of these axes. But there is no reason why they all have to send connections along the same axis.

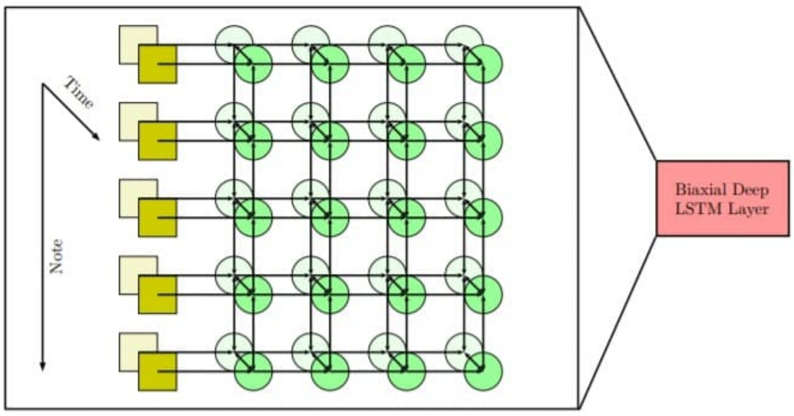


Fig. 3.16 deep Biaxial-LSTM architecture

Consider a MIDI file in the following manner: for each instrument, we have a matrix of notes by times, where at each time step and each note we have feature describing if the note is played and if it is newly articulated. After splitting up instruments, we consider this input as layer 0 in that instrument’s section of our computation graph. Then at each hidden layer, we have an affine transformation from the previous layer, and we use LSTM-RNN equations in each of these directions of the layer (i.e, note direction and time direction). At the end of this set of layers, the instruments have only learned independently (it’s entirely possible for the melody to optimize in one direction and the percussion to optimize in another), so we add a mixing layer that involves an affine transformation between the final hidden layers of all the instruments. We then use a sigmoid to calculate each note’s probability of being played at each time step, and then use these probabilities to evaluate loss and generate music.

Chapter 4

Problem Definition and Methodology

4.1 Problem definition

The main challenge would be testing the Originality of the Music, i-e, the generated musical notes or sequence should not just be the imitation of the notes from the dataset. The musical notes should be different and at the same time, it should be as satisfactory as music composed by humans. Since char-RNN takes one input character at a time 't' , the model can only be trained by data consisting of single instrument or monophonic music (atmost one note at same time).The problem with Char-RNN model is that in case of multi-instrument music or polyphonic music (more than one note at same time), the model failed to generate proper human-satisfactory music. Hence we define another powerful variant of RNN called deep Biaxial-RNN model for generating polyphonic music and verify that the model produced more human-satisfactory music compared to Char-RNN model.

4.2 Char-RNN Model Architecture

4.2.1 Many-to-Many RNN

In the figure 4.1, The box is a single RNN unit where 'C_i' is a first character input given to RNN unit. For first RNN we also have to provide zero input which is nothing but a dummy input because RNN always takes current input and previous output as input. Since, we don't have previous output for the first iteration so we input zero, only for first iteration.

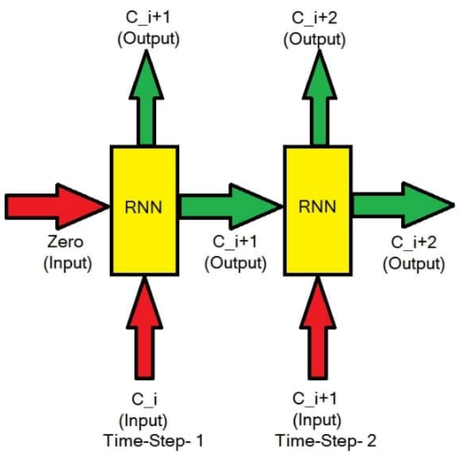


Fig. 4.1 Many-to-Many RNN

Now, we want our RNN to produce next character as output. So, the output of 'C_i' will be 'C_{i+1}' which is nothing but a next character in sequence. Now, our next input itself is a next character, and at the same time the output of previous input is also next character so we feed both of them to next RNN unit again and produces 'C_{i+2}' as output which is next character in the sequence. This is how Many-to-Many RNN works. Note, the above image shows time-unwrapping of RNN. It is only one RNN unit which repeats itself at every time-step.

4.2.2 Single Layer Many-to-Many RNN

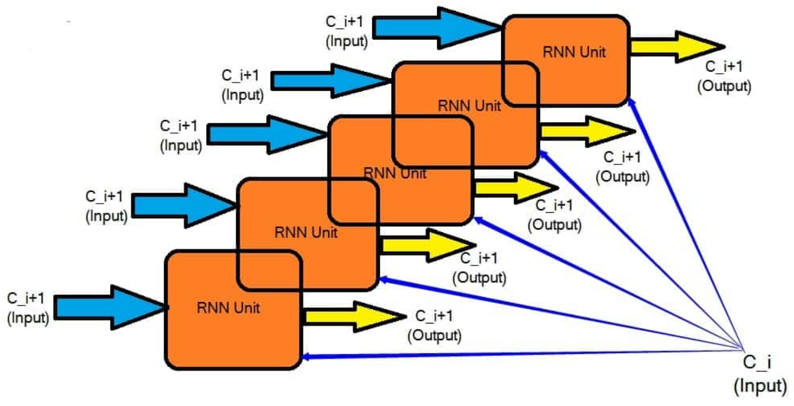


Fig. 4.2 Many-to-Many RNN with n Units in a layer

In figure 4.2, ’n’ RNN units in single RNN layer. We have constructed our single RNN layer exactly like this where we have 256 LSTM-RNN Units in one layer of RNN. At each time step all of the RNN units generate output which will be an input to the next layer and also the same output will again be any input to the same RNN unit. Here, in our project each RNN unit is an LSTM unit. In Keras library, in LSTM there is a parameter called ‘**return_sequences**’. It is False by default. But if it is True, then each RNN unit will generate output for each character means at each time step. This is what we want here. We want our RNN unit to generate output as the next character when given input as previous character in the sequence. We have stacked up many LSTM units here so that each unit will learn different aspect of the sequence and create a more robust model overall.

4.2.3 Model Architecture

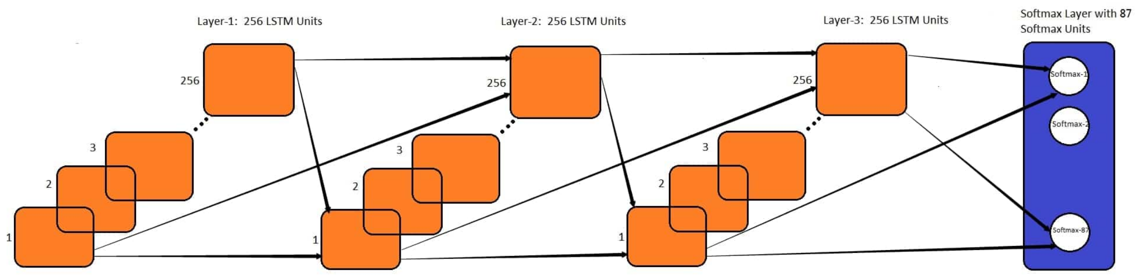


Fig. 4.3 Char RNN model Architecture

In the figure 4.2, one RNN layer with ’n’ units is shown. We have three such RNN layers each having 256 LSTM units in our model architecture in the figure 4.3. The output of each LSTM unit will be an input to all of the LSTM units in next layer and so on.

After three such layers of RNN, we have applied ‘**TimeDistributed**’ dense layers with “Softmax” activations in it. This wrapper applies a layer to every temporal slice of an input. Since the shape of each output after third LSTM layer is (16*64*256). We have 87 unique characters in our dataset and we want that the output at each time-stamp will be a next character in the sequence which is one of the 87 characters. So, time-distributed dense layer contains 87 “Softmax” activations and it creates a dense connection at each time-stamp. Finally, it will generate 87 dimensional output at each time-stamp which will be equivalent to 87 probability values. It helps us to maintain Many-to-Many relationship.

There is one more parameter in LSTM which is known as "**stateful**". Here, we have given "stateful = True". If True, then the last state for each sample at index 'i' in a batch will be used as initial state for the sample of index 'i' in the following batch. This is used because all of the batches contains rows in continuation. So, if we feed the last state of a batch as initial state in the next batch then our model will learn more longer sequences.

Once our char-RNN model is trained, we will then give any one random character rom the set of unique characters that we feed to our char RNN during training time to our trained char RNN, it will then generate characters automatically which will be based on the sequences and patterns that it has learnt during training phase.

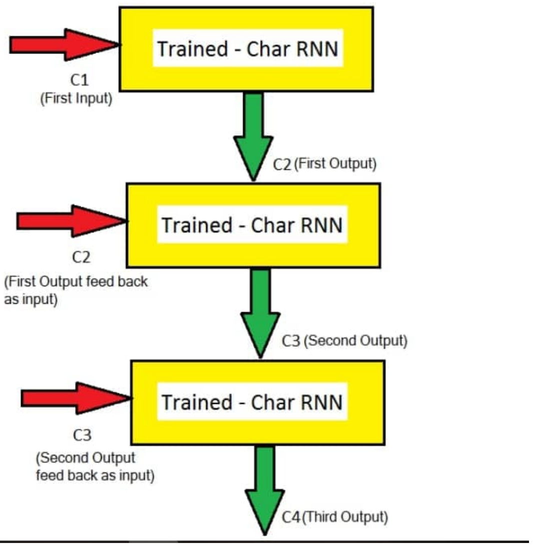


Fig. 4.4 Generate Output Sequences

In the figure 4.4, image we give "C1" as the an input to our trained RNN. Note, that "C1" is a character which should be present in the set of the characters that we feed to our char RNN during training time. Now our trained char RNN will generate output "C2". This "C2" output is feed back as input again to the trained char RNN. This will generate "C3" as an output. This "C3" output is feed back as input again to the trained char RNN and so on. Now we got new sequence of music [C1, C2, C3. . .]. This new characters of sequence is a new music generated by our trained char RNN which is based on the sequences and patterns that it has learnt during training phase.

4.3 Deep Biaxial Quad-Directional RNN Model

Our new neural network architecture shown in figure 4.5, is structured a bit differently than the state of the art biaxial RNN to generalize the generation task to any written song composition with a variety of instruments. To summarize, our neural network is a deep bidirectional, biaxial LSTM (d-BBLSTM) with an affine mixture layer that takes into account the relationships between the instruments.
We will now detail the d-BBLSTM structure and equations that make up $f^{-}1$.

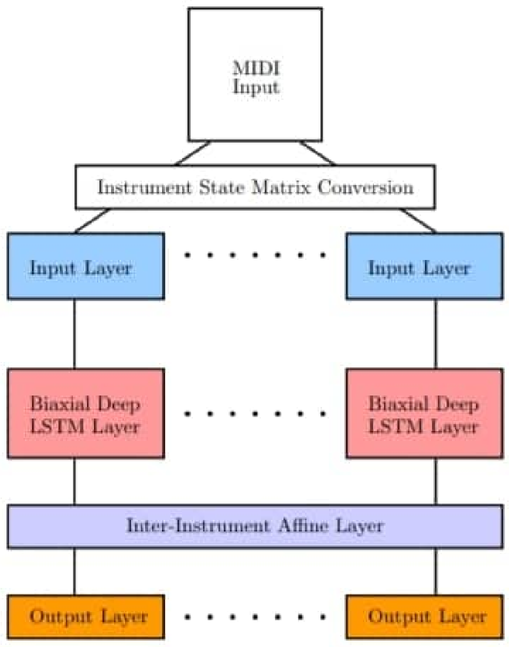


Fig. 4.5 deep Biaxial RNN

- **Input Layer:** The input layer has dimensions $\mathbb{R}^{B\times T\times N\times D_c\times M}$, where B is the size of the batch. Consider one of the input layers for instrument class c up until the instrument mixture layer. Let a single training example x $\in \mathbb{R}^{T\times N\times D_c}$ be the input to this layer .

- **d-BBLSTM layers:** The largest layers are the d-BBLSTM layers. The d-BBLSTM has a three-dimensional prismatic network structure of $T \times N \times L$ hidden nodes with bidirectional connections. We have a fixed dimension for the hidden node $h_{tn}^{(l)} \in \mathbb{R}^P$ with (t, n, l) [T] × [N] × [L]. The equations for the forward propagation are:

$$\overrightarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)} = \text{ReLu} \left(W_c\, {}^{c}x_{tn} + \overrightarrow{U_c}^{(l)}\, \overrightarrow{\overrightarrow{{}^{c}h}}{}^{(l)}_{t(n-1)} + \overrightarrow{V_c}^{(l)}\, \overrightarrow{\overrightarrow{{}^{c}h}}{}^{(l)}_{(t-1)n} + \overrightarrow{\overrightarrow{Z_c}}^{(l)}\, {}^{c}h^{(l-1)}_{tn} + \overrightarrow{\overrightarrow{b_c}}^{(l)} \right)$$

$$\overleftarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)} = \text{ReLu} \left(W_c\, {}^{c}x_{tn} + \overleftarrow{U_c}^{(l)}\, \overleftarrow{\overrightarrow{{}^{c}h}}{}^{(l)}_{t(n+1)} + \overrightarrow{V_c}^{(l)}\, \overleftarrow{\overrightarrow{{}^{c}h}}{}^{(l)}_{(t-1)n} + \overleftarrow{\overrightarrow{Z_c}}^{(l)}\, {}^{c}h^{(l-1)}_{tn} + \overleftarrow{\overrightarrow{b_c}}^{(l)} \right)$$

$$\overrightarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)} = \text{ReLu}\left(W_c\, {}^{c}x_{tn} + \overrightarrow{U_c}^{(l)}\, \overleftarrow{\overrightarrow{{}^{c}h}}{}^{(l)}_{t(n-1)} + \overleftarrow{V_c}^{(l)}\, \overrightarrow{\overleftarrow{{}^{c}h}}{}^{(l)}_{(t+1)n} + \overrightarrow{\overleftarrow{Z_c}}^{(l)}\, {}^{c}h^{(l-1)}_{tn} + \overrightarrow{\overleftarrow{b_c}}^{(l)} \right)$$

$$\overleftarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)} = \text{ReLu} \left(W_c\, {}^{c}x_{tn} + \overleftarrow{U_c}^{(l)}\, \overleftarrow{\overleftarrow{{}^{c}h}}{}^{(l)}_{t(n+1)} + \overleftarrow{V_c}^{(l)}\, \overleftarrow{\overleftarrow{{}^{c}h}}{}^{(l)}_{(t+1)n} + \overleftarrow{\overleftarrow{Z_c}}^{(l)}\, {}^{c}h^{(l-1)}_{tn} + \overleftarrow{\overleftarrow{b_c}}^{(l)} \right)$$

where $\overrightarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)}$, $\overleftarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)}$, $\overrightarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)}$, $\overleftarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)}$ all have dimension P and represent hidden layer activations trained in note-space and time-space for instrument class c, and the input x only feeds into the first layer (set this term to 0 for all other layers). We use the notation ${}^{c}h^{(l)}_{tn} = [\overrightarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)}; \overleftarrow{\overrightarrow{{}^{c}h_{tn}}}^{(l)}; \overrightarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)}; \overleftarrow{\overleftarrow{{}^{c}h_{tn}}}^{(l)}\,]$ to represent the concatenation of these hidden states for a given c, t, n, and ‘. We choose to use ReLU nonlinearities in order to combat the vanishing gradient problem. Using this formulation, we have all bias terms $b^{(l)}_c \in \mathbb{R}^{P}$ (for each direction) , $W_c \in \mathbb{R}^{P\times D_c}$, $Z^{(l)}_c \in \mathbb{R}^{P\times 4P}$ (for each direction), and all other weights $U^{(l)}_c$, $V^{(l)}_c \in \mathbb{R}^{P\times P}$ (for each direction). This gives us a total of ($\sum_c D_c P$) +4CP^2L+4CP^2L+4CPL = ($\sum_c D_c P$) + 4CP L(2P +1) parameters (depending on hyperparameters, this typically comes out to somewhere between 0.5 to 1.5 million parameters when we have two instrument classes: percussion and other instruments). We have the following initial states: $h^{(0)}_{tn} = h^{(l)}_{0n} = h^{(l)}_{(T+1)n} = h^{(l)}_{t0} = h^{(l)}_{t(N+1)} = 0$. The output of this layer is ${}^{c}h^{(L)}_{tn}$, which we will group into $H_c \in \mathbb{R}^{T\times N\times P}$. Note that each of 5 these nodes are actually LSTM nodes in order to improve the long-term dependencies of our model.

- **Mixing layer:** The mixing layer consists of a concatenation of the final hidden outputs from the previous layer along the hidden dimension to construct $H \in \mathbb{R}^{T\times N\times CP}$ followed by an affine transformation and sigmoid nonlinearity to the output layer. In the mixing layer, we do not mix along the time and note dimensions but rather along the concatenated hidden dimension. Ultimately, we'd like to output the tensor $\hat{Y} \in \mathbb{R}^{T\times N\times 2\times C}$ an output layer for each of the instruments. The equation for this transformation is as follows:
  $\hat{y}_{tn} = \sigma(\mathrm{W}_M\ h_{tn} + b_M)$
  where,h$_{tn} \in \mathbb{R}^{CP}$ lies along the hidden dimension of H. Here, we have $W_M \in \mathbb{R}^{2\times C\times CP}$ and $b_M \in \mathbb{R}^{2\times C}$. This layer has a total of 2C^2P + 2C parameters.

- **Output layer:** The output layer consists of the final cross-entropy calculation. Since at this point we have calculated the binary target labels **Y** and the probabilities at the output , we end up with the element-wise sum:

  $CE(\mathrm{Y}, \hat{Y}) = \| = \mathrm{Y} \circ \log \hat{Y} \|_{1,1,1,1}$

  where $\| . \|_{1,1,1,1}$ indicates a sum over the absolute value of the 4-tensor (note that all entries in the tensor will be positive so the absolute value does not matter in this case).

Chapter 5

Experiment Setup and Implementation

5.1 Generating Monophonic music using Char RNN model

5.1.1 Using ABC-format data

Data Source : From ABC version of Nottingham Music Database, we have downloaded first two files:

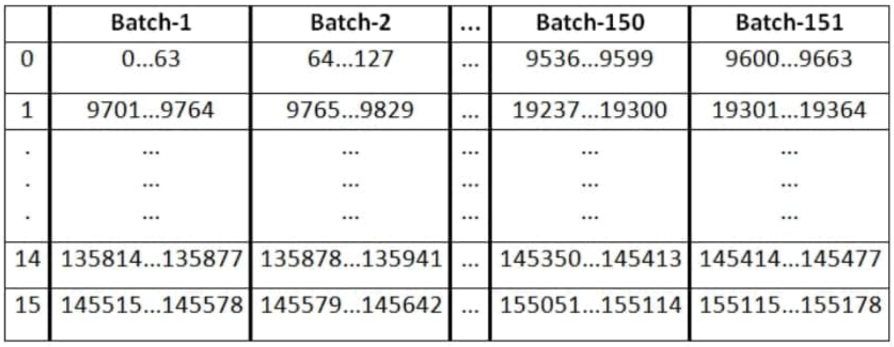
- **Jigs**: consisting of 340 tunes
- **Hornpipes** : consisting of 65 tunes

We will feed data into batches. We will feed batch of sequences at once into our RNN model. First we have to construct our batches. We have set following parameters:

Batch Size = 16

Sequence Length = 64

We have found out that there are total of **155179** characters in our data. Total number of unique characters are **87** We have assigned a numerical index to each unique character. We have created a dictionary where key belongs to a character and its value is it's index. We have also created an opposite of it, where key belongs to index and its value is it's character.



	Batch-1	Batch-2	...	Batch-150	Batch-151
0	0...63	64...127	...	9536...9599	9600...9663
1	9701...9764	9765...9829	...	19237...19300	19301...19364
.	...	...	...	...	...
.	...	...	...	...	...
.	...	...	...	...	...
14	135814...135877	135878...135941	...	145350...145413	145414...145477
15	145515...145578	145579...145642	...	155051...155114	155115...155178

Fig. 5.1 Batch structure

We have assigned index to each of the character. So in the figure 5.1, the numbers are not the exact numbers. In reality, the batches will contain index of the corresponding character.

```
def read_batches(all_chars, unique_chars):
    length = all_chars.shape[0]
    batch_chars = int(length / BATCH_SIZE)

    for start in range(0, batch_chars - SEQ_LENGTH, 64):
        X = np.zeros((BATCH_SIZE, SEQ_LENGTH))
        Y = np.zeros((BATCH_SIZE, SEQ_LENGTH, unique_chars))
        for batch_index in range(0, 16):
            for i in range(0, 64):
                X[batch_index, i] = all_chars[batch_index *
batch_chars + start + i]
                Y[batch_index, i, all_chars[batch_index *
batch_chars + start + i + 1]] = 1
        yield X, Y
```

Fig. 5.2 Function to create batches

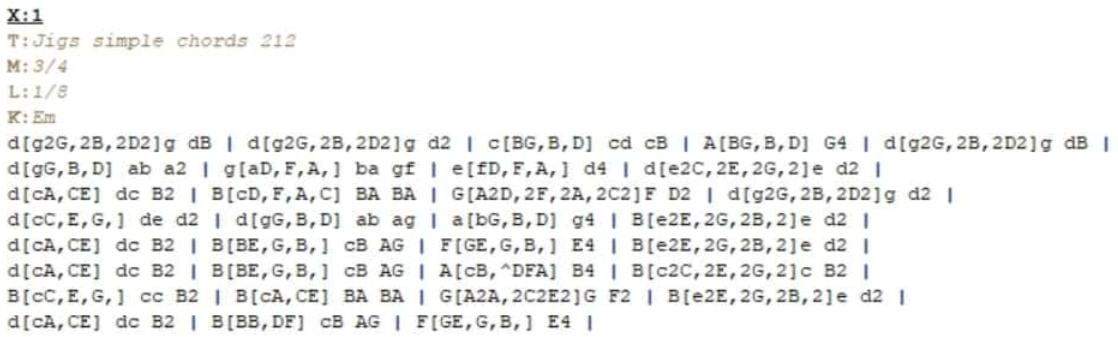
In the figure 5.2, a code snippet is shown as a function to create batches. Here, there are three nested loops. First loop denotes batch number. It runs every time when a new batch is created. Second loop denotes row in a batch and third loop denotes column in a batch.

We have trained our char RNN model through batches on Nvidia GeForce GTX 1060 GPU and total training time was about 3 hours. The implementation was done in Python3 using *Keras* library with Tensorflow Backend.

5.2 Generating Polyphonic music using Char RNN model

5.2.1 Using MIDI(Musical Instrument Digital Interface) Files

We have collected a bunch random midi tunes or files which are generally audio files from *Nottingham Polyphonic Music database*. The midi files are converted to ABC-format data for training our Char-RNN model using and open-source tool called EasyABC. Hence the Char-RNN model was trained on this ABC converted midi files.



```
X:1
T:Jigs simple chords 212
M:3/4
L:1/8
K:Em
d[g2G,2B,2D2]g dB | d[g2G,2B,2D2]g d2 | c[BG,B,D] cd cB | A[BG,B,D] G4 | d[g2G,2B,2D2]g dB |
d[gG,B,D] ab a2 | g[aD,F,A,] ba gf | e[fD,F,A,] d4 | d[e2C,2E,2G,2]e d2 |
d[cA,CE] dc B2 | B[cD,F,A,C] BA BA | G[A2D,2F,2A,2C2]F D2 | d[g2G,2B,2D2]g d2 |
d[cC,E,G,] de d2 | d[gG,B,D] ab ag | a[bG,B,D] g4 | B[e2E,2G,2B,2]e d2 |
d[cA,CE] dc B2 | B[BE,G,B,] cB AG | F[GE,G,B,] E4 | B[e2E,2G,2B,2]e d2 |
d[cA,CE] dc B2 | B[BE,G,B,] cB AG | A[cB,^DFA] B4 | B[c2C,2E,2G,2]c B2 |
B[cC,E,G,] cc B2 | B[cA,CE] BA BA | G[A2A,2C2E2]G F2 | B[e2E,2G,2B,2]e d2 |
d[cA,CE] dc B2 | B[BB,DF] cB AG | F[GE,G,B,] E4 |
```

Fig. 5.3 Polyphonic tune (ABC format)

In the figure 5.3, and example of Polyphonic ABC tune(file) is shown, which was converted from midi-format to ABC-format. The notes that are inside the squared brackets ('[ ]') indicate that they have been played at same time step.

5.3 Generating Polyphonic music using deep Biaxial RNN model

Our network is based on this architectural idea, but of course the actual implementation is a bit more complex. First, we have the input to the first time-axis layer at each time step: (the number in brackets is the number of elements in the input vector that correspond to each part)

- **Position** [1]: The MIDI note value of the current note. Used to get a vague idea of how high or low a given note is, to allow for differences (like the concept that lower notes are typically chords, upper notes are typically melody).
- **Pitchclass** [12]: Will be 1 at the position of the current note, starting at A for 0 and increasing by 1 per half-step, and 0 for all the others. Used to allow selection of more common chords (i.e. it's more common to have a C major chord than an E-flat major chord)
- **Previous Vicinity** [50]: Gives context for surrounding notes in the last timestep, one octave in each direction. The value at index 2(i+12) is 1 if the note at offset i from current note was played last timestep, and 0 if it was not. The value at 2(i+12) + 1 is 1 if that note was articulated last timestep, and 0 if it was not. (So if you play a note and hold it, first timestep has 1 in both, second has it only in first. If you repeat a note, second will have 1 both times.)
- **Previous Context** [12]: Value at index i will be the number of times any note x where (x-i-pitchclass) mod 12 was played last timestep. Thus if current note is C and there were 2 E's last timestep, the value at index 4 (since E is 4 half steps above C) would be 2.
- **Beat** [4]: Essentially a binary representation of position within the measure, assuming 4/4 time. With each row being one of the beat inputs, and each column being a time step, it basically just repeats the following pattern:
  0101010101010101
  0011001100110011
  0000111100001111
  0000000011111111

However, it is scaled to [-1, 1] instead of [0,1].

Then there is the first hidden LSTM stack, which consists of LSTMs that have recurrent connections along the time-axis. The last time-axis layer outputs some note state that represents any time patterns. The second LSTM stack, which is recurrent along the note axis, then scans up from low notes to high notes. At each note-step (equivalent of time-steps) it gets as input

- the corresponding note-state vector from the previous LSTM stack.
- a value (0 or 1) for whether the previous (half-step lower) note was chosen to be played (based on previous note-step, starts 0).
- a value (0 or 1) for whether the previous (half-step lower) note was chosen to be articulated (based on previous note-step, starts 0).

After the last LSTM, there is a simple, non-recurrent output layer that outputs 2 values:

- **Play Probability**, which is the probability that this note should be chosen to be played.
- **Articulate Probability**, which is the probability the note is articulated, given that it is played. (This is only used to determine rearticulation for held notes.)

The model is implemented in *Theano*, a Python library that makes it easy to generate fast neural networks by compiling the network to GPU-optimized code and by automatically calculating the gradients.

During training, we can feed in a randomly-selected batch of short music segments. We then take all of the output probabilities, and calculate the cross-entropy, which is a fancy way of saying we find the likelihood of generating the correct output given the output probabilities. After some manipulation using logarithms to make the probabilities not ridiculously tiny, followed by negating it so that it becomes a minimization problem, we plug that in as the cost into the AdaDelta optimizer and let it optimize our weights.

We can make training faster by taking advantage of the fact that we already know exactly which output we will choose at each time step. Basically, we can first batch all of the notes together and train the time-axis layers, and then we can reorder the output to batch all of the times together and train all the note-axis layers. This allows us to more effectively utilize the GPU, which is good at multiplying huge matrices.

To prevent our model from being overfit (which would mean learning specific parts of specific pieces instead of overall patterns and features), we can use something called dropout. Applying dropout essentially means randomly removing half of the hidden nodes from each layer during each training step. This prevents the nodes from gravitating toward fragile dependencies on each other and instead promotes specialization. (We can implement this by multiplying a mask with the outputs of each layer. Nodes are "removed" by zeroing their output in the given time step.)

During composition, we unfortunately cannot batch everything as effectively. At each time step, we have to first run the time-axis layers by one tick, and then run an entire recurrent sequence of the note-axis layers to determine what input to give to the time-axis layers at the next tick. This makes composition slower. In addition, we have to add a correction factor to account for the dropout during training. Practically, this means multiplying the output of each node by 0.5. This prevents the network from becoming overexcited due to the higher number of active nodes.

We have trained the model using a Nvidia GeForce GTX 1060 GPU. The model used two hidden time-axis layers, each with 200 nodes, and two note-axis layers, with 100 and 50 nodes, respectively. We trained it using a dump of *Classical Piano Midi* files from *Nottingham Polyphonic Music database* (same data which was used for training char-RNN model), in batches of ten randomly-chosen 8-measure chunks at a time.

Note: The total time for training was about 25 hours with 200 hidden nodes on time-axis and about 22 hours with 100 nodes on time-axis.

Chapter 6

Results and Conclusions

6.1 Evaluation of Char-RNN model generating monophonic music

In the graph (figure 6.1), the log-loss of our trained model and in (figure 6.2), the accuracy of our trained model are shown :

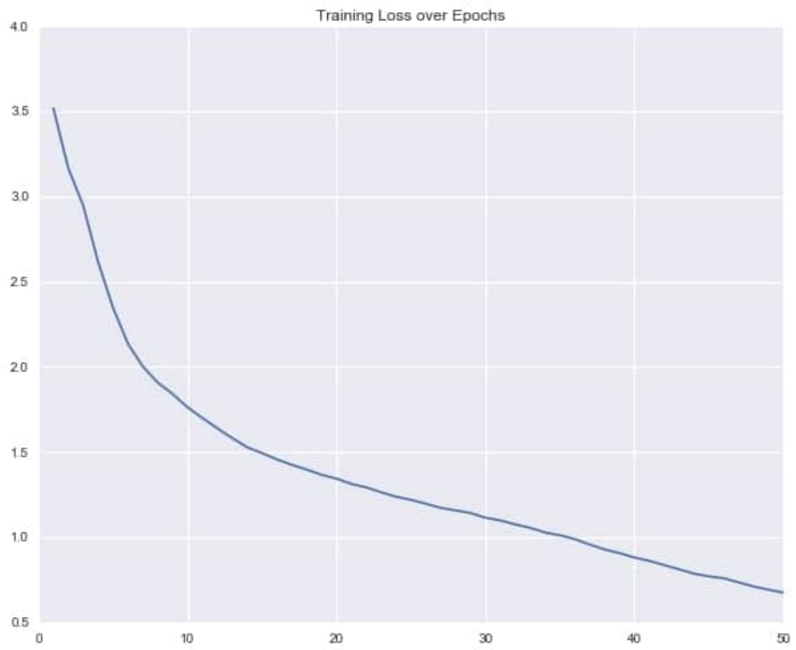


Fig. 6.1 Training Loss over Epochs

Epoch	Log-Loss	Accuracy
1	3.183	0.175
10	1.678	0.624
50	0.376	0.785
100	0.028	0.912

Table 6.1 Loss and Accuracy over Epochs

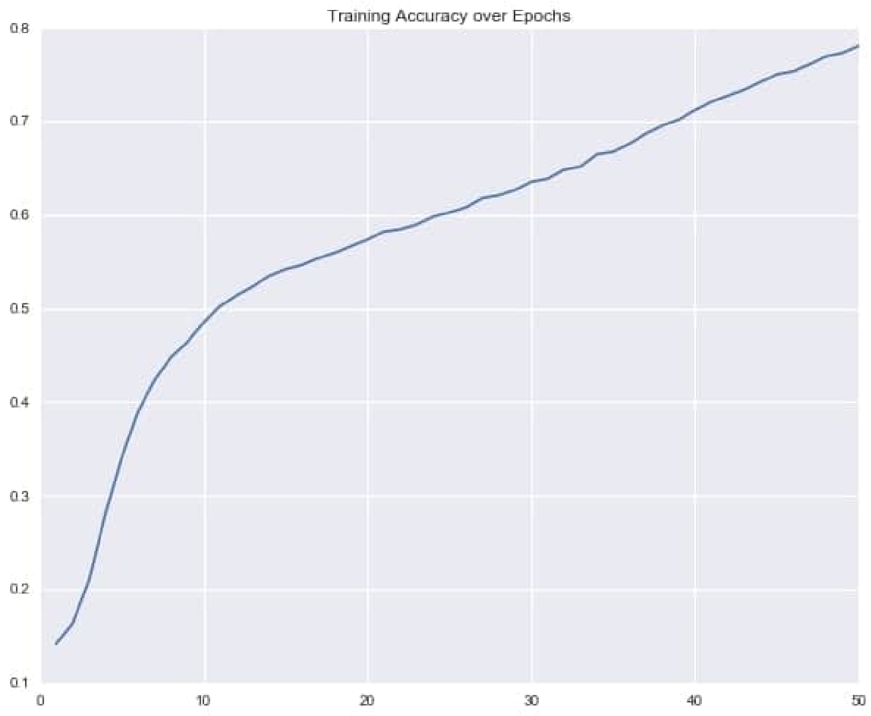


Fig. 6.2 Training Accuracy over Epochs

In the table 6.1, the numbers represent the loss and accuracy values after 1, 10, 50 and 100 epochs.The Accuracy at 100th Epoch was about 91% .

6.2 Evaluation of Char-RNN model generating polyphonic music

As we already discussed, we have converted the midi files(polyphonic music files) to ABC-format and trained the Char-RNN model to generate new polyphonic music. In the figure 6.3 and 6.4 , the training loss and accuracy over epochs are shown.

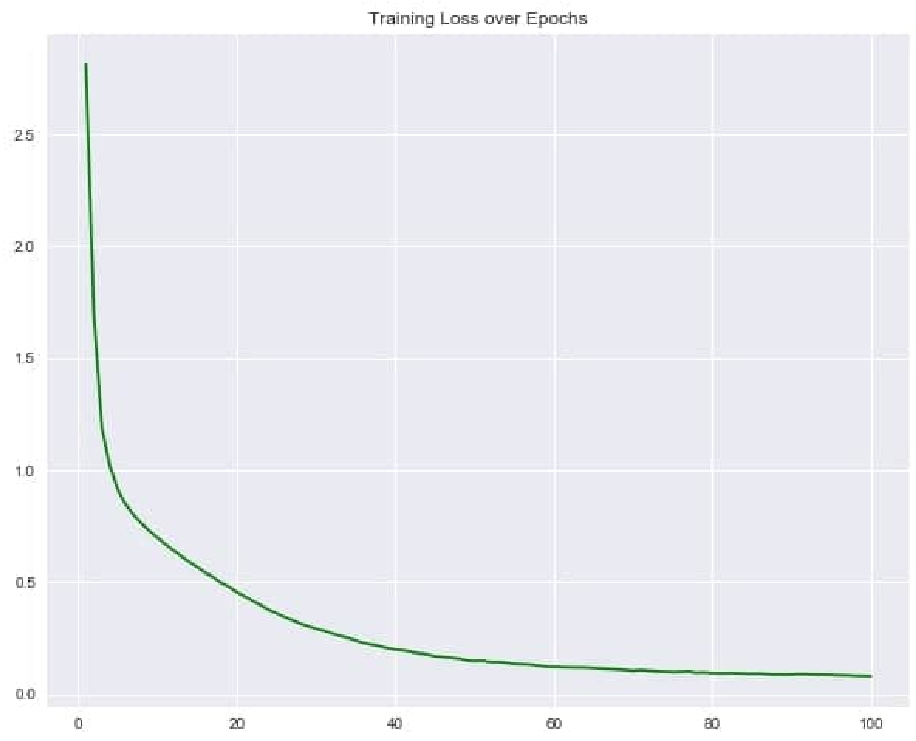


Fig. 6.3 Training Loss over Epochs

Epoch	Log-Loss	Accuracy
1	2.812	0.225
10	0.699	0.773
50	0.148	0.950
100	0.008	0.973

Table 6.2 Loss and Accuracy over Epochs

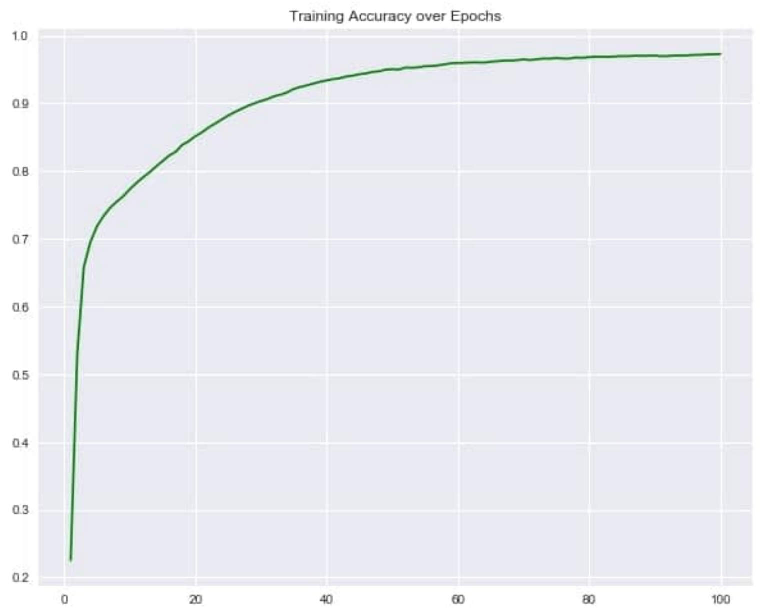


Fig. 6.4 Training Accuracy over Epochs

In the table 6.2, the numbers represent the loss and accuracy values after 1, 10, 50 and 100 epochs.The Accuracy at 100th Epoch was about 97% . But in this case, accuracy is not a right measure of testing our model. This is because in case of polyphonic music (abc-format), the set containing notes at same time step(within squared brackets) is smaller compared to full tune. Hence the model will tend to learn patterns from this smaller set compared to the full tune. Therefore, there is high chance that the model will produce repetitive notes which would also lead to repetitive tunes, if the accuracy is high. It is always recommended to perform human evaluation of music for testing the model rather than just checking the accuracy.

So, we cannot say that since the accuracy is high the model is very robust in generating polyphonic music. Hence, we have used Biaxial-RNN model for generating polyphonic music which has produced some pretty good tunes compared to Char-RNN model.

6.3 Evaluation of deep Biaxial Quad-directional RNN model generating polyphonic music

In the graph (figure 6.5), the log-loss of our trained model with 100 hidden nodes on time-axis is shown:

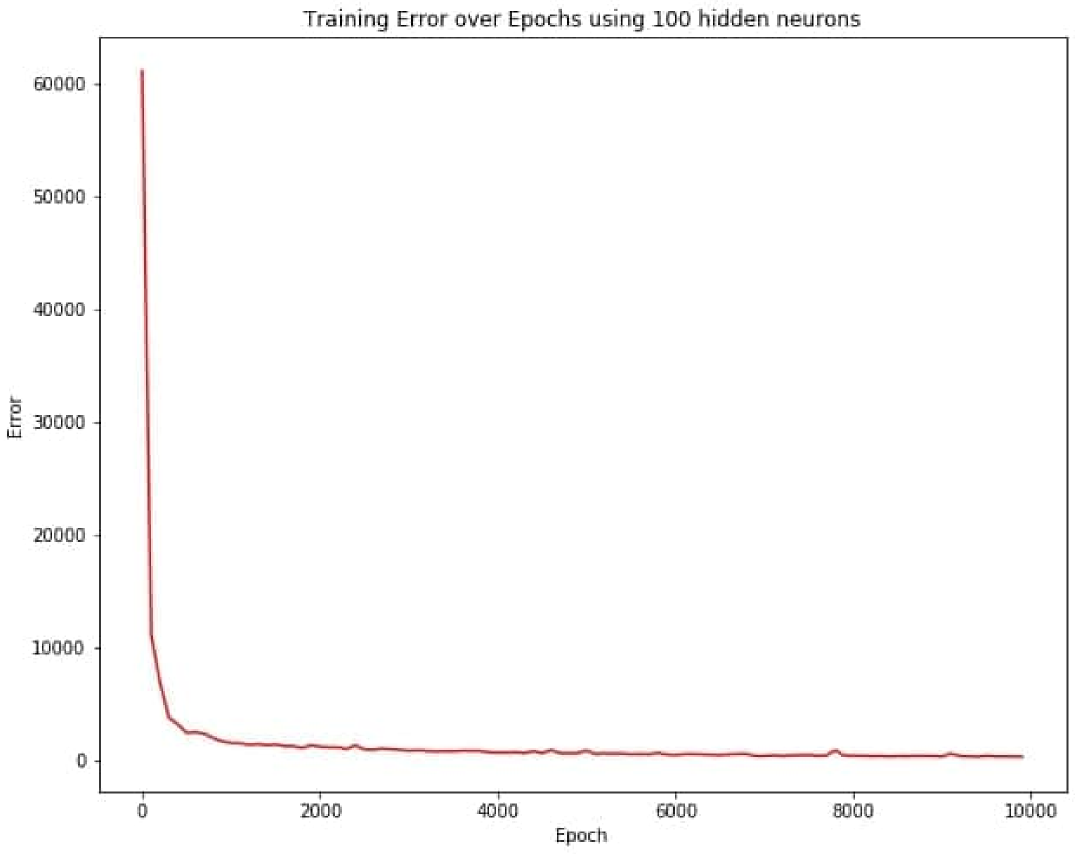


Fig. 6.5 Training Loss over Epochs with 100 hidden nodes

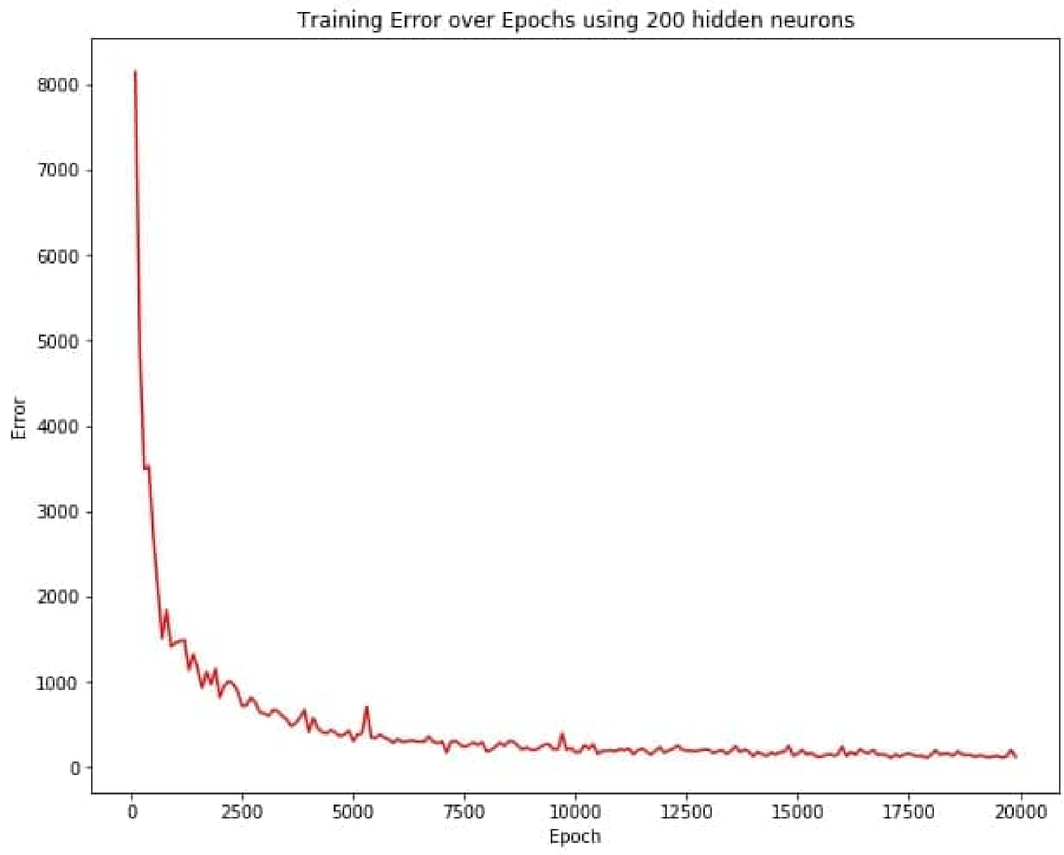


Fig. 6.6 Training Accuracy over Epochs with 200 hidden nodes

In the figure 6.6, we can see that the curve is not smooth compared to the graph(figure 6.5) with 100 hidden nodes. This is due to the residual or unwanted weights that are calculated which may lead to disturbance of error term.

Epoch	Log-Loss
100	11106.431
1000	1534.527
2500	963.903
5000	885.558
7500	495.111
9900	304.413

Table 6.3 For Network with 100 hidden nodes on time-axis

Epoch	Log-Loss
100	8146.665
1000	1458.269
2500	720.422
5000	308.203
7500	236.040
9900	228.437

Table 6.4 for Network with 200 hidden nodes on time-axis

In the table 6.3 and 6.4, the Cross-Entropy loss(Log-loss) over epochs is shown. We have trained our model till 10,000 epochs only. We need to train the model over more number of epochs to reduce the log-loss and increase the robustness of the model. But this would require a lot of patience to train on our GPU as it may take around several weeks of computation time. To reduce the computation time, we may have to switch to much powerful GPU for training the model.

Conclusion: After performing Human evaluation of music generated by our models, we can conclude that Char-RNN model has produced some human satisfying tunes in case of monophonic music but produced some repetitive tunes in case of polyphonic music. On the other hand, Biaxial-RNN model produced decent tunes in case of polyphonic music. this is because the Biaxial-RNN model learns patterns on both time and note axis where as Char-RNN model learns patterns only on time axis.

Human Evaluation of Music generated by our models can be found through this sound cloud link :

https://soundcloud.com/kedar-nath-997185105/tracks

References

[1] Cope., D. (1988). *The Algorithmic Composer.* A-R Editions.

[2] Ebcioglu., K. (1988). An expert system for harmonizing four-part chorales. *Computer Music Journal (CMJ)*, 12(3):43–51.

[3] et al., D. E. (2017). Magenta project, accessed on 20/06/2017. https://magenta.tensorflow.org.

[4] Fiebrink, R. and Caramiaux., B. (1984). The machine learning algorithm as creative musical tool,.

[5] for Artists., S. (2017). Innovating for writers and artists, accessed on 06/09/2017. https://artists.spotify.com/blog/innovating-for-writersand-artists.

[6] Hao., J. (2017). Hao staff piano roll sheet music, accessed on 19/03/2017. http://haostaff.com/store/index.php?main%20page=article.

[7] Hiller, L. A. and Isaacson.y, L. M. (1959). *A Course in Functional Analysis*.

[8] Hooktheory. (2017). Theorytabs, accessed on 26/07/2017. https://www.hooktheory.com/theorytab.

[9] Huang, A. and Wu., R. (2016). Deep learning for music.

[10] Keith., J. (2016). The session, accessed on 21/12/2016. https://thesession.org.

[11] (MMA)., M. M. A. Midi specifications, accessed on 14/04/2017.

[12] Mozer, M. C. (1994). Neural network composition by prediction: Exploring the benefits of psychophysical constraints and multiscale processing. *Connection Science*, 6(2-3):247–280.

[13] Steedman., M. (1984). A generative grammar for jazz chord sequences. *Music Perception*, 2(1):52–77.

[14] Walder., C. (June,2016). Modelling symbolic music: Beyond the piano roll.

[15] Xenakis., I. (1963). *Formalized Music: Thought and Mathematics in Composition.* Indiana University Press.

Appendix A

Installing Necessary packages and tools(Windows OS)

Installing Anaconda

1. Download the Anaconda Installer from https://www.anaconda.com/distribution/windows
2. Double click the installer to launch.
3. Click Next
4. Read the licensing terms and click “I Agree”
5. Select an install for “Just Me” unless you’re installing for all users (which requires Windows Administrator privileges) and click Next
6. Select a destination folder to install Anaconda and click the Next button
7. Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Prompt from the Start Menu
8. Choose whether to register Anaconda as your default Python. Unless you plan on installing and running multiple versions of Anaconda, or multiple versions of Python, accept the default and leave this box checked
9. Click the Install button. If you want to watch the packages Anaconda is installing, click Show Details.

10. Click the Next button.
11. After a successful installation you will see the “Thanks for installing Anaconda” dialog box.

Installing Tensorflow-gpu

1. Download CUDA toolkit 9.0 from
https://developer.nvidia.com/cuda-90-download-archive
2. Click on Windows
3. Click on Windows version 10
4. Click on Installer type as ’exe(local)’
5. Download the base installer and all the four patches.
6. After download, double click on .exe files and Express install all the files including the patches by choosing "Agree and continue"
7. Download cuDNN v7.0.5 for CUDA 9.0 from
https://developer.nvidia.com/rdp/cudnn-archive
8. Extract the downloaded cuDNN file and paste the cudnn64_7.dll into C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v9.0/bin folder .
9. Now open the command prompt and execute the following commands:

```
conda update conda
conda create -n tf10 python-3.6 pip
activate tf10

pip install --ignore-installed --upgrade tensorflow-gpu-1.10
```

Installing Theano

1. Create a new environment and install python2

    ```
    conda create --name py2 python=2.7
    ```

2. activate the new environment

    ```
    activate py2
    ```

3. Install theano in ’py2’ environment

```
conda install theano pygpu
```

Installing EasyABC

1. Download and install EasyABC for windows from https://www.nilsliberg.se/ksp/easyabc/
2. Double click on the downloaded .exe file and click "next" to complete the installation.