Deep Learning Techniques for Music Generation – *A Survey*

Jean-Pierre Briot[1,2], Gaëtan Hadjeres[1,3] and François Pachet[4,1]

[1] Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, Paris, France
[2] PUC-Rio, Rio de Janeiro, Brazil
[3] École Polytechnique, Palaiseau, France
[4] Sony CSL, Paris, France

Preface

This book is a *survey* and an *analysis* of different ways of using *deep learning* (*deep artificial neural networks*) to *generate musical content*. First, we propose a *methodology* based on four *dimensions* for our analysis:

- *objective* – What musical content is to be generated? (e.g., melody, accompaniment...),
- *representation* – What are the information formats used for the corpus and for the expected generated output? (e.g., MIDI, piano roll, text...),
- *architecture* – What model of deep neural network is to be used? (e.g., recurrent network, autoencoder, generative adversarial networks...),
- *strategy* – How to model and control the process of generation (e.g., direct feedforward, sampling, unit selection...).

For each dimension, we conduct a comparative analysis of various models and techniques. For the *strategy* dimension, we propose some tentative *typology* of possible approaches and mechanisms. This classification is *bottom-up*, based on the analysis of many existing deep-learning based systems for music generation, which are described in this book. The last part of the book includes discussion and prospects.

Acknowledgements

This research has been conducted within the Flow Machines project which received funding from the European Research Council under the European Union Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 291156. We also would like thank Sony CSL for its support and research environment. Jean-Pierre Briot would like to thank CNRS for his permanent support, Laboratoire d'Informatique de Paris 6 for his research environment and additional support from CAPES (Brazil) through a PVE (visiting Professor) fellowship during his research stay at PUC-Rio University.

Acronyms

AI	Artificial intelligence
ANN	Artificial neural network
BOW	Bag-of-words
BPTT	Back-propagation through time
CAN	Creative adversarial network
CNN	Convolutional neural network
ConvNet	Convolutional neural network
C-RBM	Convolutional restricted Boltzmann machine
CS	Constrained sampling
GAN	Generative adversarial networks
GD	Gradient descent
GLO	Generative latent optimization
GLSR-VAE	Geodesic latent space regularization for the variational autoencoder
GRU	Gated recurrent unit
GS	Gibbs sampling
KL-divergence	Kullback-Leibler divergence
LSDB	Lead sheet data base
LSTM	Long short-term memory
MCMC	Markov chain Monte Carlo
MIDI	Musical instrument digital interface
MIR	Music information retrieval
ML	Machine learning
MLP	Multilayer perceptron
NN	Neural network
NTM	Neural Turing machine
PCA	Principal component analysis
RBM	Restricted Boltzmann machine
ReLU	Rectified linear unit
RHN	Recurrent highway network
RL	Reinforcement learning
RNN	Recurrent neural network
SGD	Stochastic gradient descent
SGS	Selective Gibbs sampling
SVM	Support vector machine
TTS	Text-to-speech
VAE	Variational autoencoder
VRAE	Variational recurrent autoencoder
VRASH	Variational recurrent autoencoder supported by history

Chapter 1
Introduction

Deep learning has recently become a fast growing domain and is now used routinely for classification and prediction tasks, such as image and voice recognition, as well as translation. It has emerged about 10 years ago, in 2006, when a deep learning architecture very significantly outperformed standard techniques using handcrafted features on an image classification task [40][1]. We may explain this success and reemergence of *neural networks* techniques by the combination of:

1. technical advances (notably *pre-training*[2] and *convolutions*);
2. availability of massive data;
3. dedicated computing power[3].

There is no consensual definition for *Deep learning*. It is a *repertoire* of *machine learning (ML)* techniques, based on *artificial neural networks*. The key aspect and common ground is the term *deep*, which means that there are *multiple layers* processing *multiple levels of abstractions*, as a way to express complex representations in terms of simpler representations. The technical foundation is mostly neural networks, as we will see in Section 5, with many variants (*convolutional networks*, *recurrent networks*, *autoencoders*, *restricted Boltzmann machines*...). For more information about the history and various facets of deep learning, see, e.g., a recent comprehensive book on the domain [29].

Important part of current effort in deep learning is applied to traditional machine learning *tasks*[4]: *classification* and *prediction* (also named *regression*), as a testimony of the initial DNA of neural networks: *linear regression* and *logistic regression* (see Section 5.1) and also translation. But a growing area of application of deep learning techniques is the *generation* of *content*. Content could be of various kinds: mostly *images*, *text* and *music*, the latter being the focus of our analysis. The motivation is in using the now widely available corpus to automatically learn musical styles and to generate new musical content based on this.

---

[1] Interestingly, the title of the article [40] is about "deep belief nets" and does not mention the term "neural nets", because, as Geoffrey Hinton (the father of deep learning) reminds it [58], at that time there was a strong belief that deep neural networks were no good and could never be trained and that ICML (International Conference on Machine Learning) should *not* accept papers about neural networks.

[2] Pre-training consists in prior training in *cascade* (one layer at a time, also named *greedy layer-wise unsupervised training*) of each hidden layer, [41] [29, page 528]. It turned out to be a significant improvement for making neural networks with several layers very efficient [23]. Pre-training is now rarely used (and has been replaced by other techniques), except for some specific objectives like *transfer learning*, which addresses the issue of *reusability* of what has been learnt.

[3] Graphic cards, initially designed for video games have now one of their biggest market in deep learning applications.

[4] Tasks in machine learning are types of problems and may also be described in terms of how the machine learning system should process an example [29, Chapter 5]. Examples are: classification, regression, translation, anomaly detection...

1.1 Related Work

To our knowledge, there are only a few partial attempts at analyzing the use of deep learning for generating music. Graves presented an interesting analysis, focusing on recurrent neural networks and on text (digital or handwritten) [31]. Humphrey, Bello and LeCun presented another interesting analysis, sharing with us some issues about music representation (see Section 4), but dedicated to music information retrieval (MIR) tasks, such as chord recognition, genre recognition and mood estimation [49].

One could also consult the proceedings of some recently created international workshops on the topic, e.g., Workshop on Constructive Machine Learning, at Conference on Neural Information Processing Systems (NIPS) in December 2016 [16] and Workshop on Deep learning for Music, at International Joint Conference on Neural Networks (IJCNN) in May 2017 [38].

There are also various models and techniques for using computers to generate music, e.g., rules, grammars, automata, Markov models, graphical models, where models are manually defined by experts or are learnt from examples. They will not be addressed here as we focus on deep learning architectures. Please refer to some general books about computer music, e.g. [88], and algorithmic modeling and generation of music, e.g., [80] and [15].

1.2 Requisites and Roadmap

This book does not need prior knowledge about deep learning and neural networks nor music.

Chapter 2 Method introduces the method of analysis and the four dimensions considered (objective, representation, architecture and strategy), that correspond to the next four chapters.

Chapter 3 Objective is mostly a reminder of the different types of musical content that we want to generate. Although a very short chapter, we think it is an useful reminder in order no to confuse between different objectives for generation (e.g., melody from scratch, counterpoint to an existing melody...) which usually lead to different architectures and strategies.

Chapter 4 Representation is a reminder and analysis of the different types of representation and techniques for encoding musical content (notes, durations, chords...). This chapter may be skipped by a reader already expert in computer music.

Chapter 5 Architecture is a reminder of the most common deep learning architectures (feedforward, recurrent, autoencoder...) used for generation of music. This includes a short reminder of the very basics of a simple neural network. This chapter may be skipped by a reader already expert in neural networks and deep learning architectures.

Chapter 6 Strategy is a tentative to classify various approaches (strategies) for using deep learning architectures to generate music. It is derived from our survey of numerous systems and experiments which are presented and discussed in next chapter.

Chapter 7 Systems is a survey of various systems and experiments in the literature. They are analyzed, compared and classified along the classifications proposed in previous chapters.

Chapter 8 Analysis summarizes the survey conducted in previous chapter through some tables, as a way to help at identifying the design decisions made for different systems.

Chapter 9 Other sources of inspiration goes outside the musical field to introduce some examples of systems and experiments, mostly in the field of image generation, that may be future sources of inspiration for music generation.

Chapter 10 Discussion revisits some issues that have been touched upon during the previous chapters.

1.3 Limits of this Book

This book does not intend to be a general introduction to deep learning. A recent and excellent book on this topic is [29]. Our survey and analysis of existing systems is obviously not exhaustive. We have tried to select the most representative ones and as the field of deep learning for music generation is currently very active, new systems are being presented at the time of our writing. Therefore, we encourage our readers and colleagues for any feedback and suggestions for improving this survey and analysis objetive and still ongoing project.

Chapter 2
Method

In our analysis, we propose to consider four main *dimensions* to *identify* and *classify* attempts at using deep learning to generate musical content[1]:

- *objective* – What type of musical content is to be generated[2]. This can be a *melody* (*temporal sequence* of *notes*[3]); a full *polyphony* (*chorale*) with multiple *voices*; an *accompaniment* to a given melody, e.g., through a sequence of *chords*, as an *harmony* associated to the initial melody; the association of a melody and chords (a *lead sheet*, common in Jazz), etc.
- *representation* – The nature of the information (data) used to *train* and to *generate* musical content. Possible representations are: *signal*, transformed signal (ex: *spectrum*, via Fourier transformation), *piano roll*, *MIDI*, *text*, etc. Note that in some cases, some *additional* information, (sometimes named *metadata*) may be provided to the system in order to improve the quality of the generation, ex: *pitch class*, *fermata*, etc. Also, *feature extraction* (*handcrafted* or *automated*) may be also applied to the data in order to improve generation.
- *architecture* – The architecture may be conventional, with various *layers* and *units*[4]. It may include *convolutions* (to enforce some *invariance*, or may be *recurrent*, to be able to learn *sequences*, *Autoencoder* architectures are also of interest, for their capacity to extract features. As we will see, some systems *combine architectural traits* (e.g., convolutions *and* recurrence) and, furthermore, some systems *combine architectures*.
- *strategy* – There are various *ways of using* deep learning architectures to generate musical content. For instance, a *direct use* means using a deep network architecture for prediction task (by feedforward computation) to produce music. Most of the systems actually make a *indirect* use of deep learning architectures and we will analyze various approaches: e.g., by *sampling* from a generated distribution, by *input manipulation*, by *querying* musical units from a generated description and *concatenating* them...

These 4 dimensions are not orthogonal, the type of architecture partially *constrains* the strategy of its use and vice-versa. The choice of representation is also partially determined by the objective and the architecture, and, as usual, selecting an appropriate representation and encoding are *key* decisions. Also, for most of these dimensions, solutions may be combined (e.g., as for *compound architectures*).

---

[1] In the following, we will call these various attempts (architectures and experiments, published in the litterature) *systems*.

[2] We could have used the term *task*. But, as its is a relatively well defined and used term in the machine learning community (see Section 1 and [29, Chapter 5]), we preferred using another term.

[3] Allowing or not simultaneous notes.

[4] also called *neurons*, because of the inspiration from *biological neural networks*.

Chapter 3
Objective

The first dimension, the *objective* is the type of musical content to be generated. This can be, e.g.:

- a *melody*, 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) or *polyphonic* (more than one note can be played at the same time)[1]; monophony
- a *polyphony*, with multiple *voices* (intended for more than one voice or instrument), for instance a *chorale*;
- an *accompaniment* to a given melody. This can be:
  - a *counterpoint*, composed of one or more melodies (voices),
  - a sequence of *chords*, that provides some associated *harmony*;
- the association of a *melody* and *chords* – this is called a *lead sheet* and is common in Jazz – it may also include *lyrics*[2].

The *objective* implies the nature of the *output* to be generated, e.g., a melody. The *input* for the generation may have a similar nature or may differ. The input could be for instance: a melody (to generate an accompaniment), a single note (to begin a melody), or even nothing.

An important issue is the *destination* of the generated musical content. The destination could be the computer to play the musical content that has been generated, with the final output format being readable by the computer, e.g., audio or MIDI. Or the destination is a human user, then the final output format must be human readable, typically a music *score*. These formats may be interchangeable (e.g., MIDI to score and vice-versa), but with some possible loss of information.

Another important issue is about the *autonomy* of the generation of the musical content. It could be completely *autonomous* and *automated*, without any human intervention. Or it could be more or less *interactive* with a *control interface* for a human user to guide the process of generation. As deep learning for music generation is recent and basic neural networks techniques are non interactive, the majority of systems and experiments that we analyzed are not (yet) interactive[3]. But the most interesting goal appears to be the design of interactive support systems for the musicians (for composing, arranging and other activities), as, e.g., showed by the FlowComposer prototype [87].

---

[1] We will call this a *single-track* or *single-voice polyphony*, when this is intended for a single voice and instrument.

[2] Note that lyrics could be generated too. Although this target is out of scope for this book, we will see later in Section 4.3.3 that in some systems, music is encoded as a text. Thus, some of the techniques that we will describe are text based and could be applied to lyrics.

[3] There are some interactive systems, e.g., deepAudioController (see Section 7.1.3.2), and DeepBach (see Section 7.2.2.1).

Chapter 4
Representation

A second dimension is about the way musical content is represented.

4.1 Stages of representations

A least[1], three types and stages of data representations may be considered:

- *training input* – The dataset used for training the deep learning system;
- *generating input* – The eventual data that may be used as input for the generation, e.g., a melody for which the system will generate an accompaniment, or a note that will be the first note of a generated melody;
- *generated output* – The objective of the generation.

Depending on the starting material (*input*) and the objective (*output* to be generated), these three types of representations may be *equal* or *different*. For instance, in the case of the generation of monodic melodies, e.g., in [97], both training input and generated output are monodic melodies and generating input as well, in general restricted to a single starting note. In the case of generation of an accompaniment as a chorale polyphony, e.g., in [36], training input and generating input are monodic melodies and generated output is a set of monodic melodies.

4.2 Signal

The first type of representation of musical content is *audio signal*. It can be the raw audio signal (waveform), or its audio *spectrum*, processed via a *Fourier transform*. Figure 4.1 shows an example of *spectrogram* used as an input in the experiment described in [93]. The *x* axis represents time (in seconds), the *y* axis the frequency (in kHz) and the third axis, visualized in colors, represents the amplitude (in dBFS[2]).

[1] There may be more types and stages, depending on the complexity of the architecture which may include intermediary processing stages.

[2] Decibels relative to full scale, a unit of measurement for amplitude levels in digital systems.

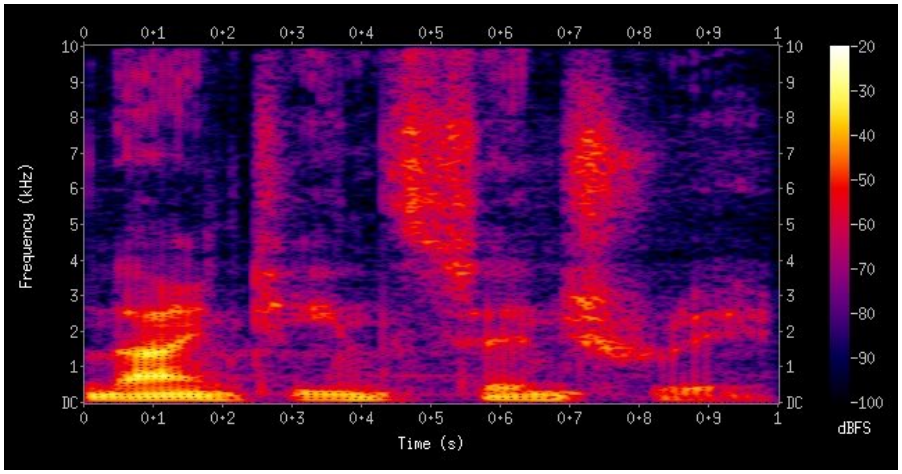

Fig. 4.1 Example of spectrogram.

4.3 Symbolic

Most of current systems and experiments on using deep learning for music generation focus on symbolic representations. Different forms of symbolic representations may be used and we summarize the most common ones in the following.

4.3.1 MIDI

MIDI (Musical Instrument Digital Interface) is a technical standard that describes a protocol, a digital interface and connectors for interoperability between various electronic musical instruments, software and devices [73]. MIDI carries *event messages* that specify note information (such as *pitch* and *velocity*) as well as *control signals* for *parameters* (such as *volume*, *vibrato* and *clock signals*). There are five types of messages and here we only consider the *Channel Voice* type, which transmits real-time performance data over a single channel. Two important (for our concerns) messages are:

- *Note on* – To indicate that a note is (or has to be) *played*. It contains a *status information* (what *channel number* is concerned, specified by an integer within `[0 15]` and two data values: a MIDI *note number* (the note's *pitch*, an integer within `[0 127]`) and a *velocity* (that indicates how loud the note is played[3], an integer within `[0 127]`). An example is `<Note on, 0, 60, 50>` which interprets as: "on channel 1, start playing a middle C with velocity 50".
- *Note off* – To indicate that a note ends. In that situation, velocity indicates how fast the note is released. An example is `<Note off, 0, 60, 20>` which interprets as: "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, as an *absolute value*, of the event and could represent:

- a *metrical time* – It represents the number of *ticks* from the beginning. A reference, named the *division* and defined in the file header, specifies how many ticks per quarter note ♩,

[3] For a keyboard, it means the speed of pressing down the key and therefore corresponds to the volume.

- or a *time-code-based time* – Not detailed here.

An example of extract from a MIDI file (turnt into readable ascii) and its corresponding score are shown at Figures 4.2 and 4.3. The division has been set to 384, i.e. 384 ticks per quarter note ♩, which corresponds to 96 ticks for an eighteenth note 𝅘𝅥𝅯.

```
2, 96, Note_on_c, 0, 60, 90
2, 192, Note_off_c, 0, 60, 0
2, 192, Note_on_c, 0, 62, 90
2, 288, Note_off_c, 0, 62, 0
2, 288, Note_on_c, 0, 64, 90
2, 384, Note_off_c, 0, 64, 0
2, 384, Note_on_c, 0, 65, 90
2, 480, Note_off_c, 0, 65, 0
2, 480, Note_on_c, 0, 62, 90
2, 576, Note_off_c, 0, 62, 0
```

Fig. 4.2 Extract from a MIDI file.

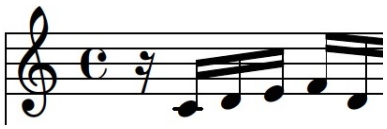

Fig. 4.3 Score corresponding to the MIDI extract.

In [47], Huang and Hu claim that one drawback of encoding MIDI messages directly is that it does not effectively preserve the notion of multiple notes being played at once through the use of multiple tracks. Since they concatenate tracks end-to-end, they posit that it will be difficult for such a model to learn that multiple notes in the same position across different tracks can really be played at the same time.

4.3.2 Piano roll

The *piano roll* representation of a melody (monodic or polyphonic) is inspired from *automated pianos* (see Figure 4.4). This was a continuous *roll of paper* with *perforations* (holes) punched into it. Each perforation represents a *note control information*, to trigger a given note. The *length* of the perforation corresponds to the *duration* of a note. On 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 at Figure 4.5. The x axis represents time and the y axis the pitch. In that example, two voices are encoded.

The piano roll is one of the most frequent representations used, although it has some limitations. An important one, comparing to MIDI representation, is that the information of the note off does not exist. As a result, there is no way to distinguish between a long note and two short notes (see Section 4.4.2). The experiment conducted by Huang and Hu [47] is interesting in that they compare using MIDI and piano roll formats. See also the publication by Walder entitled “Modelling Symbolic Music: Beyond the Piano Roll” [113].

Fig. 4.4 Automated piano and piano roll.

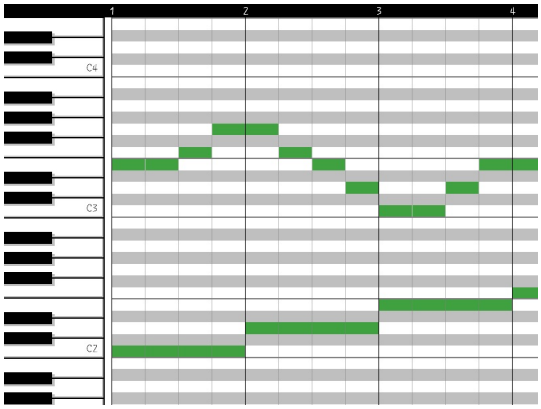

Fig. 4.5 Example of symbolic piano roll.

4.3.3 Text

A melody can be encoded in a textual representation and processed as a *text*. A significant example is the *ABC notation* [115], a *de facto* standard for folk and traditional music[4]. See at Figures 4.6 and 4.7 the original score and its associated ABC notation of a music named "A Cup of Tea", from the repository and discussion platform *The Session* [56].

The first 6 lines are the header and represent metadata (e.g., T: title of the music, M: meter (it is actually the time signature), L: default length, K: key…). It is followed by the main text representing the melody. We illustrate below some basic principles of the encoding rules (please refer to [115] for the details):

- the *pitch class*[5] 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 $A4$[6], `a` to an *A* one octave up and `a'` to an *A* two octaves up;

[4] Note that the ABC notation has been designed *independently* of machine learning concerns.

[5] A pitch class is modulo 12 and therefore independent of the octave position.

[6] Also named $A440$ and also known as the *Stuttgart pitch*, it has a frequency of 440 Hz and is the musical note *A* above $middleC$. It serves as a general tuning standard for musical pitch. Its corresponding MIDI note is `69`.

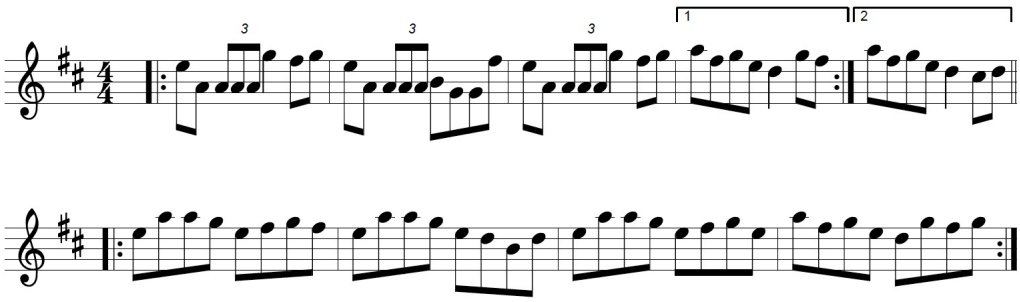

Fig. 4.6 Score of "A Cup of Tea".

```
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. 4.7 ABC notation of "A Cup of Tea".

- the *duration* of a note is encoded as following: if default length is marked as `1/8` (i.e. an eighth note[7] ♪ – the case for this example), `a` corresponds to an eighth note ♪, `a/2` to a sixteenth note 𝅘𝅥𝅯 and `a2` to a quarter note ♩;
- *measures* (also named bars)[8] are separated by `|` (bars).

Note that the ABC notation can only represent monodic melodies. This representation is for instance used by Sturm in [97] (see Section 7.3.1.2).

4.3.4 Chord

A representation of a *chord* could be *extensional*, enumerating the notes composing it, or *intensional*, specifying the *pitch class* of its root note (e.g., `C`, `A`...) and its type (e.g., *major*, *minor*, *dominant seventh*...), usually using an abbreviated jazz notation, e.g., `C`, `D-`, `E7`...[9]. In most cases, the abbreviated notation is chosen, as in Jazz and popular music.

In summary, a chord is usually represented by a pair $< pitchclass, type >$, where *pitch class* $\in$ {`A`, `A`$\sharp$[10], `B`, ..., `G`, `G`$\sharp$} and the set of possible types is predefined (e.g., {`+`, `-`, `7`, `-7`, `+`, `-7(`$\flat$`5)`, `11`, ``$\flat$`13(`$\flat$`9)`...}). Note that the way, standard in Jazz, to indicate a note other than the root (of the chord) as to be played by the bass (e.g., `A-7/E`[11]) is an additional issue most of time not considered in music generation.

---

[7] Named a *semi-quaver* in British english.

[8] The terms *measure* and *bar* are used interchangeably, although a bar is actually the graphical entity (line segment `|`) separating measures. Thus it seems as an abuse of language. We will stick to the term *measure* in this book.

[9] There are some synonyms, e.g., `C minor` = `Cmin` = `Cm` = `C-`; `C major 7th` = `CM7` = `Cmaj7` = `C`Δ...

[10] Actually, in the tempered system, `A`$\sharp$ is *enharmonically* equivalent (they have the same pitches elements) to `B`$\flat$, although semantically (considering tonality as well as composer's intention) they are different. See the related discussion in Section 4.4.6.

[11] Its meaning is an A minor seventh chord with an E, the fifth of the chord, played as the bass/lowest note.

An interesting alternative representation of chords, named *Chord2Vec* (inspired by the *Word2Vec* model for natural language [70]), has been recently proposed in [68][12]. Rather than thinking of chords (vertically) as vectors, they represent chords (horizontally) as sequences of constituent notes. More precisely: 1) a chord is represented as an (arbitrary length) ordered sequence of notes and 2) chords are separated by a special symbol (as for sentence markers in natural language processing). When using this representation for predicting neighboring chords, a specific compound architecture is used, named RNN Encoder-Decoder, offering a very accurate model (see Section 7.3.4.1).

Note that a somehow similar model has been used for polyphonic music generation in the BachBot system [66] (analyzed in Section 7.3.1.3). In this model, for each time step, the various notes are represented as a sequence and a special delimiter symbol (`|||`) indicates the next time frame (with constant time step of an eighth note ♪). Notes are ordered, in a descending pitch (soprano voice being the first one). Each note is encoded as its MIDI pitch value and a boolean indicating if it is tied to a note at the same pitch from previous frame. An example is shown at Figure 4.8, encoding two successive chords (the first having the duration of a quarter note) and the second one possessing a fermata (noted as `(.)`).

```
(59, False)
(56, False)
(52, False)
(47, False)
|||
(59, True)
(56, True)
(52, True)
(47, True)
|||
(.)
(57, False)
(52, False)
(48, False)
(45, False)
|||
```

Fig. 4.8 Example of score encoding in BachBot.

4.3.5 Lead Sheet

Lead sheets are an important representation format for popular music (Jazz, pop...). A *lead sheet* conveys in a single or a few pages the score of a melody with annotations specifying the corresponding chord labels (harmony). Lyrics may also be added. Some important information for the performer, such as composer, author, style and tempo (ex: ballad, medium swing), etc. are usually also present. See an example of lead sheet at Figure 4.9.

Paradoxally, few systems and experiments use this rich and concise representation and most of time focus on melody. An example is Eck and Schmidhuber's Blues generation system (see Section 7.1.2.1) which outputs a combination of melody and chord sequences, although not as an *explicit* lead sheet. A notable contribution is the systematic encoding of lead sheets done in the Flow Machines project [81], resulting in the LSDB (Lead Sheet Data Base) repository [82], which includes more than 12 000 lead sheets.

[12] For information, there is another similar model, also with the name Chord2Vec model, from [48].

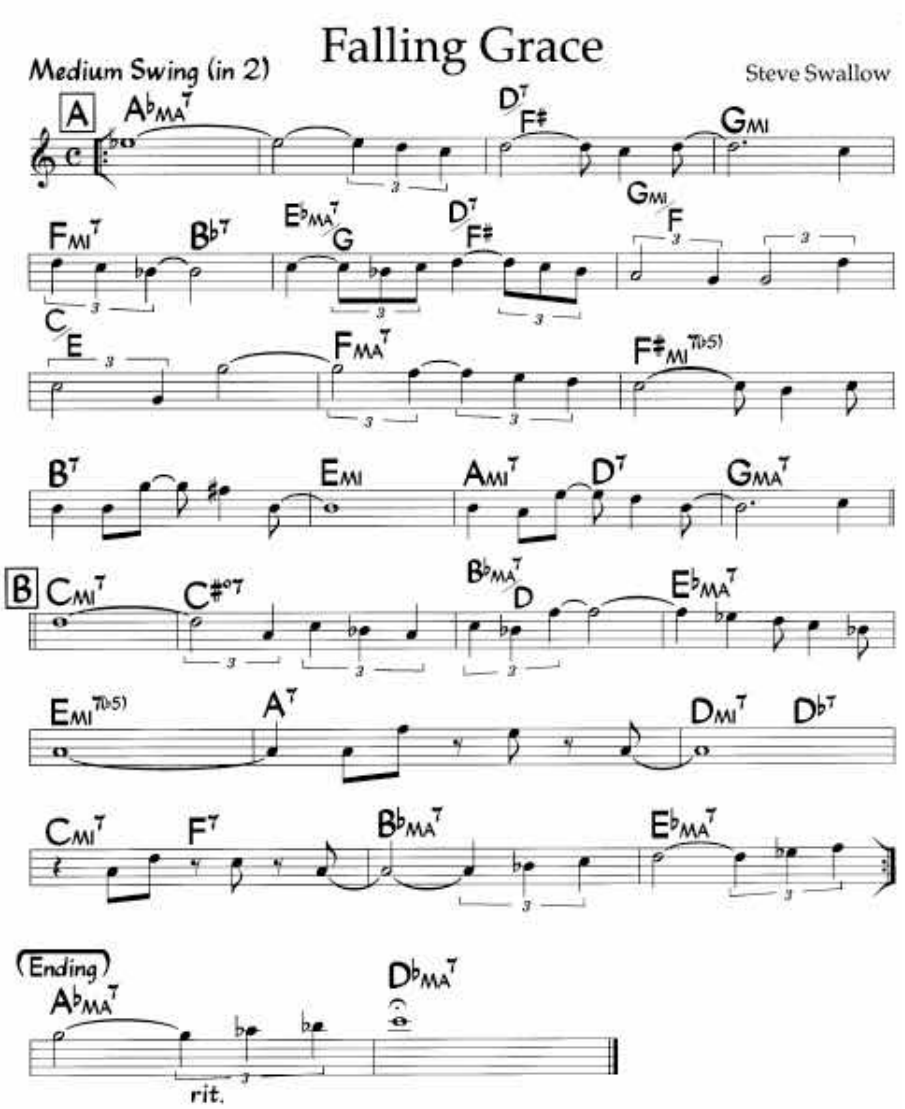

Fig. 4.9 Example of lead sheet.

4.3.6 Rhythm

Rhythm is implicit in the representation of melodies. But for drums and percussion, more dedicated representations of rhythm and of different drums and percussion parts may be considered. Different drums components, e.g., kick, snare, toms, hi-hats, cymbals... may be specified, in a way analog to polyphony by considering distinct simultaneous voices.

An example of system dedicated to rhythm generation is described in Section 7.4.1.1. In this system, each of the 5 components is represented through a binary value, specifying if there is or not a related event for current time step. Encoding is made in text, similar in spirit to Sturm *et al.*'s Celtic music system (see Section 7.3.1.2), more precisely following the approach proposed in [11]. Drums events are represented as a binary word of length 5 where each binary value corresponds to a drum component, e.g., `10010` represents simultaneous playing of kick and high-hat.

There is also condensed representation of the bass line part and some additional information represents the metrical structure (the beat structure). See more details in Sec-

tion 7.4.1.1. The authors [69] argue that this extra explicit information ensures that the network is aware of the beat structure at any given point.

4.4 Common Issues and Techniques

There are a number of issues and techniques which are common to most of representations.

4.4.1 Global vs Time Slice

The representation of *time* is fundamental for musical processes. There is a first decision about the *temporal scope* of the representation (and its relation to the *temporal nature* of the architecture used):

- *global* – In this first case, there is no notion of temporal sequence and no explicit notion of time. The granularity of processing by the deep network architecture is the representation as a whole. The architecture used is not recurrent (typically a feedforward architecture or an autoencoder). Examples are the MiniBach system (see Section 7.1.1.1) and the DeepHear system (see Section 7.1.3.1).
- *time step* (or *time slice*) – In this second case, the most frequent one, the atomic temporal granularity of the input (*training input* or *generation input*) is a *local* temporal *slice* of the musical content, corresponding to a specific temporal moment (time step). The granularity of processing by the deep network architecture is a time step. Note that the *time slice* is usually set to the *shortest note duration* (see Section 4.4.3), but it may be set larger, e.g., to a measure in the system discussed in [106].
- *note step* – This third case, rare, is proposed by Walder in [113]. In this approach, there is no fixed time step. The granularity of processing by the deep network architecture is a note. See [113] for more details.

A corollary of this design decision is that in the *global* case, the representation of a musical data used as a a training input and as a generation input needs to have a *fixed size* (the number of time steps), whereas in the *time step* and *note step* cases, the sequence size is *variable*: actual lengths of the training input, the generation input and the generated output may be different.

4.4.2 Note Ending

Another important issue is about the *note ending*, i.e. how is (*or* is not) represented the *end* of a note. In the MIDI representation format, the end of a note is explicitly stated (via a `Note Off` event[13]). In the piano roll[14] notation shown in Section 4.3.2, there is no explicit representation of the ending of a note and, as a result, one cannot distinguish between two repeating quarter notes ♩♩ and a half note 𝅗𝅥. In [21], Eck and Schmidhuber mention two strategies to address this limitation:

[13] Note that, in MIDI, a `Note on` message with a null (`0`) velocity is interpreted as a `Note off` message.

[14] Actually, in the mechanical paper piano roll, the distinction is made: two holes are different from a longer single hole. The end of the hole is the encoding of the end of the note.

- a first strategy (and the most common one) is to divide by 2 the size of the *time step* (*time slice*)[15] and always mark note endings with a special tag, e.g., `0`. The advantage is that one does not need to change input and target data structures;
- an alternative strategy is to have a *special computing unit(s)* in the network to indicate the beginning of a note. This method is employed by Todd in [106].

4.4.3 Time Quantization

Some global *time quantization*, i.e. the definition of the value of the *time step* is needed to temporally interpret the representation. Eck and Schmidhuber [21] mentions two alternative strategies:

- most commonly, the time step is set to the smallest duration of a note in the corpus (training examples/dataset), e.g., a sixteenth note . One immediate consequence of this "leveling down" is the number of processing steps necessary independently of the duration of actual notes;
- an alternative strategy, interesting to be noted, was devised by Mozer in the CONCERT system [77] (see Section 7.3.1.1), who used a distributed encoding of duration that allowed him to process a note of any duration in a single network processing time step. By representing in a single time step a note rather than a slice of time, the number of time steps to be bridged by the network in learning global structure is greatly reduced. The approach of Walder for a note granularity of processing (see Section 4.4.1) is analog. In this strategy, there is no uniform discretization of time (time slice) and no need for.

4.4.4 Feature Extraction

An existing data representation could be used as an input for deep networks or there can be a preliminary step of *feature extraction*, to represent the data in a more compact form. This could be interesting to gain efficiency and accuracy in the training and the generation. Also, a features-based representation may be more appropriate for indexing data, as for instance in the system based on querying musical units described in Section 7.11.1.1.

The set of *features* could be defined *manually* (*handcrafted*) or *automatically* (e.g., by an *Autoencoder*, see Section 5.3.4). In the case of *handcrafted features*, the *bag-of-words* (*BOW*) model is a common strategy for natural language text processing, but it may also be applied to other types of data, including musical data, as we will see in Section 7.11.1.1. It consists in transforming the original text (or arbitrary representation) into a "bag of words" (the vocabulary composed of all occurring words, more generally speaking, all possible tokens). Then, various measures can be used to characterize the text. The most common type is *term frequency*, i.e. the number of times a term appears in the text[16].

Sophisticated methods have been designed for neural networks architectures to autimatically compute a vector representation which preserves as much as possible the relations between the items. Vector representations for texts are named *word embeddings*[17]. A recent reference model for natural language is the Word2Vec model [70]. It has recently been

[15] usually set to the *smallest note duration*, see Section 4.4.3.

[16] Note that such a bag-of-words representation is a *lossy representation* (i.e. without effective means to perfectly reconstruct the original data representation).

[17] The term *embedding* comes from the analogy with *mathematical embedding* which is some injective and structure-preserving mapping. This term *embedding*, initially used for natural language processing, is now often used in deep learning as a general meaning for *encoding* a given representation into a vector representation.

transposed to the Chord2Vec model of vector encoding of chords, described in [68] (see Section 4.3.4).

4.4.5 Input Encoding

Once the format of the representation has been chosen, there is still the issue of how to *encode* this representation (a set of variables, e.g., pitch) into a set of *inputs* (also named *input nodes* or *input variables*) for the neural network[18]. Let us for instance consider the pitch of a note as a variable. There are two ways of encoding it:

- *value* encoding – The idea is to consider a single input node for a variable and directly encode its *value* by a numerical value. In the case of a pitch, various choices are then possible for representing it: the corresponding frequency (Hertz) value; the corresponding MIDI pitch (see Section 4.3.1), which is a common practice; or assign to each pitch a relative value within the interval `[0 1]` (this corresponds to a *normalization*); etc.
- *item* encoding, also named *one-hot*[19] encoding – The idea is to consider each possible note as a distinct element (*token*) of a vocabulary[20] and consider N input nodes, where N is the size of the vocabulary (e.g., the number of distinct notes). Then the *presence* and not the *value* of a note pitch, will be encoded: the occurrence of a specific note pitch will be encoded as value `1` for the corresponding input parameter and value `0` for all other input parameters. Intuitively, this corresponds to a *piano roll* representation, with as many lines (input nodes) as there are possible pitches (see at Figure 4.5).

Value encoding is rarely used except for audio and one-hot encoding is the most frequent strategy. It presents each distinct value (e.g., pitch of a note) as a distinct input and feature, all orthogonal, that can be therefore explicitly and separately handled. With value encoding, the risk is to loose some accuracy because of numerical operations (approximations). In other words, one-hot encoding is a *discretization*, thus more robust than the original *analogic* value. The disadvantage is that for a high cardinality the number of inputs can be large.

Actually, the choice of a one-hot encoding for the *output* (it is most of time chosen both for input and output), has an impact on the learning task, as it reformulates a prediction task (of an output value) into a classification task between the set of discrete possible values (this will be analyzed in Section 5.3.1.1).

4.4.6 Note Encoding

Most systems consider enharmony, i.e. in the tempered system, A♯ is equivalent to (having the same pitch as) B♭, although in the composer's intention (and tonality context) they may be different. An exception is the DeepBach system [36] (described in Section 7.2.2.1), which encodes notes using their real names (and not their MIDI pitches). The authors of DeepBach state that this additional information leads to a more accurate model and better results.

---

[18] See Section 5 for more details about the input nodes of a network.

[19] The name comes from digital circuits, *one-hot* referring to a group of bits among which the only legal (possible) combinations of values are those with a single *high* (hot!) (`1`) bit, all the others being *low* (`0`).

[20] as in the *bag-of-words* model (see Section 4.4.4).

4.4.7 Meta-Data

In some systems, additional information from the score may also be also explicitly represented and used, as *meta-data*, such as note ties; fermata (e.g., the BachBot system, see Section 7.3.1.3); harmonics (e.g., the CONCERT system, see Section 7.3.1.1); key and time signature (e.g., see Section 7.4.1.1); instrument associated to a voice; etc. This extra information may lead to a more accurate learning and generation.

4.4.8 Transposition

A common technique in machine learning is to generate synthetic data as a way to artificially augment the size of the dataset (the number of training examples) in order to improve the learning. In the musical domain, a natural and easy way is *transposition*, i.e. to transpose all examples in all keys[21]. In addition to artificially augment the dataset, this provides a key (tonality) invariance of all examples and thus makes the examples more generic. This also reduces sparsity in the training data. This transposition technique is for instance used in [61], described in Section 7.7.1.1. An opposed approach is to transpose (align) all examples into a *single common key*. This has been advocated by [7] to facilitate learning (see Section 7.3.2.1).

4.4.9 Datasets

A practical issue is the availability of *datasets* for training systems and also for evaluating and comparing systems and approaches. In the image domain, there are some reference datasets (such as, e.g., the MNIST[22] dataset about handwritten digits [64]). There are not yet such reference datasets in the music domain. Let us mention, in the polyphonic music (chorales) domain, the JSB Chorales dataset, as introduced in [2][23].

Other examples are: the Symbolic music data by Walder [114], a huge set of cleaned and preprocessed MIDI files; and MusicNet [104], a collection of 330 freely-licensed classical music recordings, together with over 1 million annotated labels (indicating timing and instrumental informations). Concerning lead sheets, a reference dataset is the LSDB (Lead Sheet Data Base) repository [82], which includes more than 12,000 lead sheets (including from all Jazz song books), which has been developed within the Flow Machines project [81].

[21] Indeed in Jazz vocal music, it is common practice to transpose a lead sheet to a different key in order to match the vocal range of the singer.

[22] MNIST stands for Modified National Institute of Standards and Technology.

[23] Meanwhile, this dataset uses an eighth note quantization, whereas a smaller quantization at the level of a sixteenth note should be used, in order to capture the smallest note duration (eighth note), see Section 4.4.2.

Chapter 5
Architecture

Deep networks are a natural evolution of *neural networks*, themselves an evolution of the *Perceptron*, proposed by Rosenblatt in 1957 [89]. Historically speaking, the Perceptron was criticized by Minsky and Papert in 1969 [71], for its inability to classify *non linearly separable domains*. Their criticism also served in favoring an alternative symbolic approach of Artificial Intelligence. Neural networks reappeared in the 80's, thanks to the idea of *hidden layers* overpassing the linearity limits [91]. In the 90's, neural networks suffered some declining interest, with the competition from *support vector machines* (SVM) [112], efficiently designed to maximize the *separation margin* and with a solid formal background. In 2006, thanks to Geoffrey Hinton, neural networks resurfaced [40], thanks to an efficient *pre-training* technique [23], ending the prevalent opinion that neural networks with many hidden layers could not be efficiently trained.

The purpose of the following section is at first to remind or introduce the basics principles of *neural networks*. Our objective is to define the minimal *concepts* and *terminology* that we will use when analyzing various music generation systems. Then, we will introduce the concepts and basic principles of various derived architectures (autoencoders, recurrent networks, RBMs...) that are used in musical applications. We will not describe here extensively the techniques of neural networks and deep learning as, e.g., in the recent book [29].

5.1 Linear Regression as an Introduction

5.1.1 Linear Regression

Although bio-inspired, the foundation of neural networks and deep learning is *linear regression*. In statistics, linear regression is an approach for modeling the (assumed linear) relationship between a scalar variable y and a set of one[1] or more *explanatory variables*, x_1, ... x_n (jointly noted as x). A simple example is to predict the value of a house, depending on some factors (e.g., size, height, location...). Linear regression focuses on modeling the *conditional probability distribution* of y given x, noted $P(y|x)$.

Equation 5.1 is the general model of a (multiple) linear regression, where:

- h is the *model* (also named *hypothesis*, as the hypothetical best model to be discovered, i.e. learnt);
- b the bias (representing the *offset*, also sometimes noted θ_0);

[1] The case of one explanatory variable is called *simple linear regression*, otherwise it is named *multiple linear regression*.

- and $\theta_1, ... \theta_n$ the *parameters* of the model (the *weights*), corresponding to the explanatory variables $x_1, ... x_n$.

$$h = b + \theta_1 x_1 + ... + \theta_n x_n = b + \sum_{i=1}^{n} \theta_i x_i \tag{5.1}$$

5.1.2 Training

The purpose of training a linear regression model is to find values for the weights θ_i and the bias b that fit as well as possible with the actual data (various pairs of values $< x, y >$). In other words, i.e. that for all examples x, $h(x)$ is as close as possible[2] to y.

An example is shown at Figure 5.1, in the case of simple linear regression (with only one explanatory variable x).

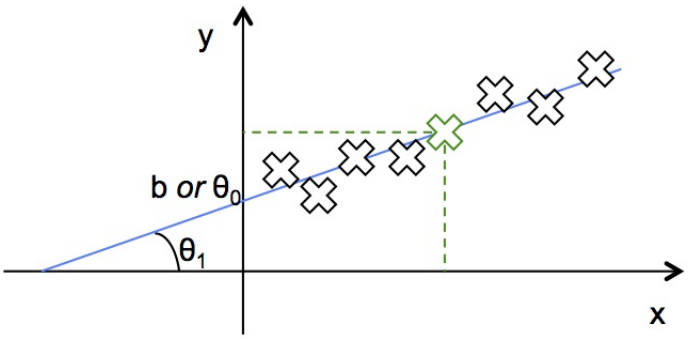

Fig. 5.1 Example of simple linear regression.

Examples are shown as black crosses. Once the model has been trained, values of the parameters are adjusted, illustrated as the blue line, which is mostly fit with the examples. Then, the model can be used for *prediction*, e.g., to estimate the value of y corresponding to a specific value of x, shown as the green cross.

5.1.3 Training Algorithm

The basic algorithm for training a linear regression model is actually pretty simple[3]:

- initialize each parameter (weight) θ_i and the bias b to a random or some more heuristic value[4];
- compute the values of the model h for all examples[5];

[2] Actually, the better fit is not necessarily the better hypothesis, because it may be have a low *generalization* (i.e., a low ability to predict yet unseen data). The optimal fit model/hypothesis would be to simply *memorize* the $< x, y >$ pairs, but it does not have any generalization ability. In practice, *overfitness* is a common and important problem and there are various strategies to minimize it (*regularization*, *dropout*...).

[3] See, e.g., [79] for details.

[4] *Pre-training* became a significant advance, as it improved *initialization* of the *parameters* by using actual training data, via sequential training of successive layers [23].

[5] Computing the cost for all examples is the best method but also computationally costly. There are numerous heuristic alternatives, in order to minimize the computational cost, e.g., *stochastic gradient descent*

- compute the *cost* (also named the *loss*), usually noted $J_\theta(h)$ (or $J(\theta)$, or $\mathcal{L}_\theta$, or $\mathcal{L}(\theta)$). It is the *distance* between $h(x)$ (the *prediction*) and y (the *actual value*) for *all* examples. It could be measured e.g., by a *mean squared error*;
- compute the *gradients*, i.e. the *partial derivatives* to each weight of the cost function $J_\theta(h)$, in respect to each weight θ_i as well as to the *bias* (weight) b;
- *search* for a new value for each weight and for each bias, guided by the value of each gradient;
- *iterate* until the error reaches *a minimum*[6], or after a certain number of iterations.

5.1.4 Architectural View

Let us now introduce at Figure 5.2 an architectural view of a linear regression model, as a precursor of a neural network. The weighted sum is represented as a *computational unit*[7] (drawn as a squared box), taking its inputs from the x_i nodes (circles). (In the example shown, there are 4 explanatory variables, x_1, x_2, x_3, x_4). There is a traditional convention to consider the bias as some specific weight, thus having a corresponding input node (that is *implicit*) and with constant value noted as $+1$.

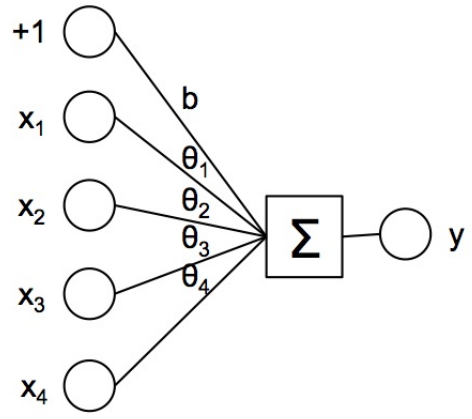

Fig. 5.2 Architectural model of linear regression.

Linear regression can also be generalized to *multivariate linear regression*, the case when there are multiple variables $y_1, \ldots y_p$ to be predicted, as illustrated at Figure 5.3 (with 3 predicted variables y_1, y_2, y_3).

(SGD), where one example is randomly chosen; *mini-batch gradient descent*, where a subset of examples is randomly chosen. See, e.g. [29, Section 5.9] for more details.

[6] If the cost function is *convex* (the case for linear regression), there is only one *global minimum*, thus there is a guarantee of finding the *optimal* model.

[7] In the following, we will use the term *node* for any component of a neural network, being just an *interface* (e.g., an input node), or a *computational unit* (e.g., a weighted sum or a function). We will use the term *unit* only in the case of a computational node. The term *neuron* may also be used in place of unit, as a way to emphasize the (partial) bio-inspiration from natural neural networks.

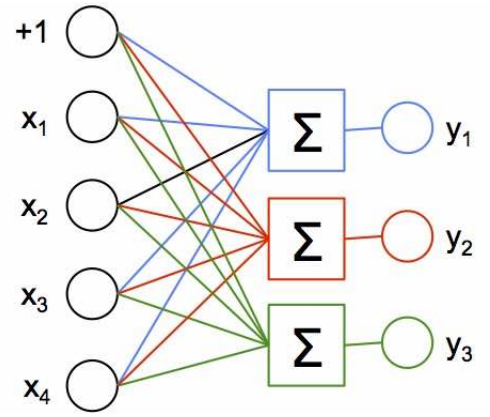

Fig. 5.3 Architectural model of multivariate linear regression.

5.2 Basic Building Block

Let us now also apply an *activation function* (AF) to each *weighted sum* unit computing y_j. This activation function allows us to introduce arbitrary *non linear functions*, in order for the model to be more general. The most common non linear function used[8] is *ReLU*, which stands for *Rectified linear unit*. It is simply a negative values filter: $ReLU(x) = max(0,x)$. It is shown at Figure 5.4 in blue color (as well as in green a smoother version named *softplus*).

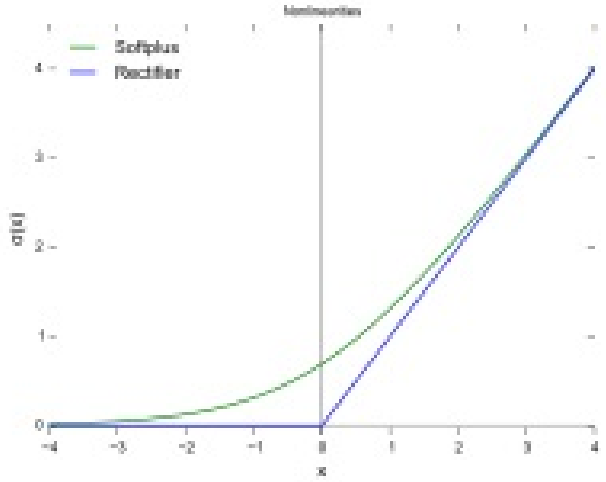

Fig. 5.4 ReLU function.

The resulting architectural representation is shown at Figure 5.5. This is a *basic building block* of neural networks and deep learning architectures.

This basic building block is actually already some working simple neural network with only two layers[9]. The *input layer*, on the left of the figure, is composed of the *input nodes*

[8] Historically speaking, the *sigmoid* function (used for *logistic regression*) was the most common. There were other alternatives, e.g., *hyperbolic tangent*. In the Perceptron, a *step* function was used (but is not differentiable, which turned it unusable to compute gradients in neural networks, see Section 5.3.1.4). ReLU is now the *de facto* standard as a non linear activation function for its effectiveness, non-saturation and its minimal computational cost.

[9] But, as it has no hidden layer, it still suffers from the linear separability limitation of the Perceptron.

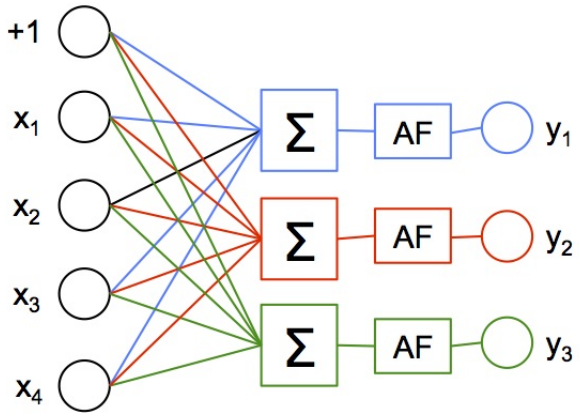

Fig. 5.5 Architectural model of the building block (multivariate linear regression with activation function).

x_i. The *bias node* is an *implicit* and specific input node. Its constant value is 1, thus it is usually noted as $+1$. The *output layer*, on the right of the figure, is composed of the *output nodes* y_j. The weight matrix is noted θ (or more often W)[10].

In order to compute the output, we simply *feedforward* the network, i.e. provide input data to the network (*feed in*) and compute the output values. This turns out to be quite efficient because computation reduces itself to *vector* (or matrix) to *matrix* multiplication (see Equation 5.2), which can be computed efficiently in *vectorized implementations*, thanks to specialized *linear algebra libraries* and furthermore to graphic processing units.

$$h(X) = AF(b + WX^{\mathrm{T}}) \tag{5.2}$$

where X is the matrix of all m examples, each example (vector of n values) being one of its row and X^{T} is the transposition of X.

5.3 Basic Architectures

From this basic building block, we can now derivate most of the four basic architectures (components) of deep learning used for music generation (and actually also for general purposes): *neural networks*, *autoencoders*, *restricted Boltzmann machines (RBMs)* and *recurrent networks (RNNs)*.

5.3.1 Multilayer Neural Network aka *Feedforward Neural Network*

A *multilayer neural network*, or simply a *neural network*, is an assemblage of successive layers of basic building blocks. The *first layer*, composed of input nodes, is called the *input layer*. The *last layer*, composed of output nodes, is called the *output layer*. Any layer within first (input) layer and last (output) layer is named a *hidden layer*.

The combination of hidden layer and non linear activation function make the neural network an universal approximator, able to overcome the non linear separability limitation[11].

[10] Its dimension is $p * n$ (p lines and n columns), where n is the number of input nodes and p the number of output nodes.

[11] The universal approximation theorem [46] states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate a wide variety of interesting functions when given appropriate parameters (weights).

See an example neural network composed of 4 layers at Figure 5.6, thus with 2 hidden layers.

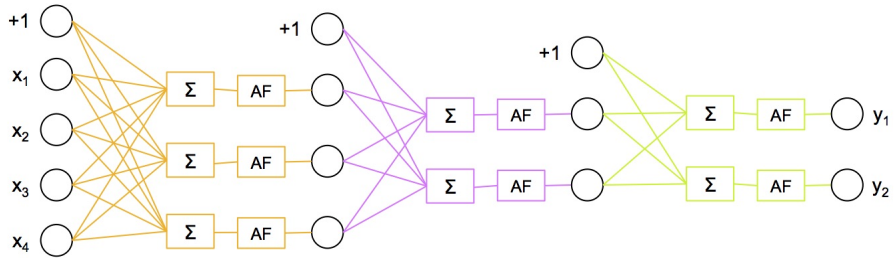

Fig. 5.6 Multilayer neural network.

The number of hidden layers was small in the first neural networks[12]. In deep networks it can be very large, e.g., the 27 layers of the GoogLeNet deep network architecture [99], illustrated at Figure 5.7, or the ResNet architecture [37] up to 152 layers.

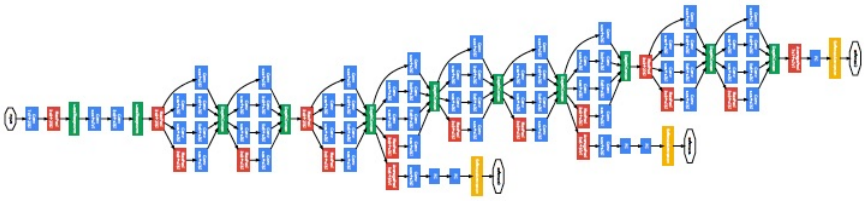

Fig. 5.7 GoogLeNet deep network architecture.

Let us simplify the figure, by omitting the sum and the non linear function units, resulting in Figure 5.8.

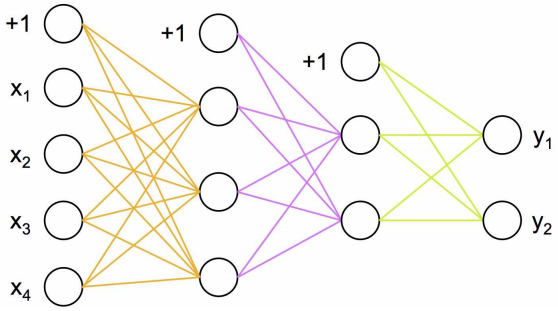

Fig. 5.8 Multilayer neural network – Simplified representation.

We can further abstract each layer by representing it as an oblong form (hiding the number of its nodes), see at Figure 5.9.

[12] The original Perceptron [89], ancestor of neural networks, had only an input layer and an output layer, thus without any hidden layer.

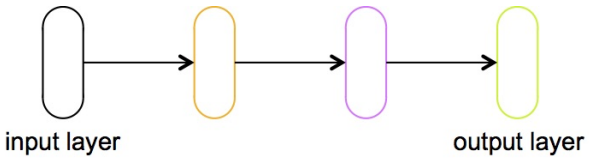

Fig. 5.9 Multilayer neural network – Abstract representation.

5.3.1.1 Output Activation Function

We have seen (in Section 5.2) that in modern neural networks, the activation function (AF) chosen for introducing non linearity at the output of each hidden layer is the ReLU function (see Section 5.2). But the output layer of a neural network has a special status. Basically, there are three possible types of activation function for the output layer:

- *identity* – This is the case for so-called *linear neural networks*, for a *prediction* task with output values being continuous. Therefore there is no non linear transformation at the last layer.
- *sigmoid* – It is used for binary classification, as in *logistic regression*[13] (which is indeed equivalent to linear regression with the addition of the sigmoid as activation function)[14]. The sigmoid function (usually written σ) is defined at Equation 5.3 and is displayed at Figure 5.10.
- *softmax* function[15] – This is the most common approach for a classification task with more than two classes, and also for a prediction task with a discrete value (as for instance to predict the pitch of a note), where a one-hot encoding is generally used (see Section 4.4.5). Indeed, this is a reformulation of a prediction (regression) task for a single value as a classification between different possible discrete values.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{5.3}$$

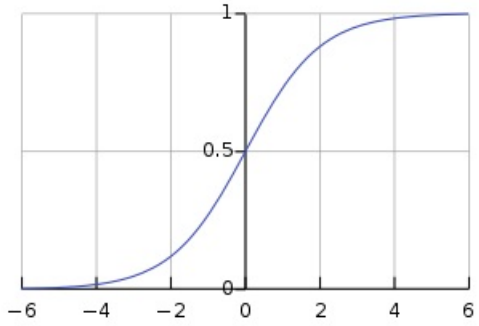

Fig. 5.10 Sigmoid function.

The softmax function actually represents a probability distribution over a discrete output variable with p possible values (in other words the probabibility of occurrence of each value j, knowing input value x, i.e. $P(y = j \mid x)$). Therefore, softmax ensures that the sum

[13] For this reason, the sigmoid function is also called the *logistic function*.

[14] See for instance [29, Chapter 6] for details about logistic regression.

[15] Softmax is a (variadic, i.e. which accepts a variable number of arguments) function, applied to the whole set of outputs.

of the probabilities for each possible value is equal to 1. The softmax function is defined at Equation 5.4[16] and an example of its use at Equation 5.5. Note that the σ notation is used, like for the logistic function, because in fact softmax is the generalization of sigmoid to the case of multiple values.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{j=1}^{p} e^{z_j}} \tag{5.4}$$

$$\sigma \begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix} \tag{5.5}$$

For a classification or a prediction task, we can simply select the value with the highest probability. But the distribution produced by the softmax can also be used as the basis for *sampling* to add variability and non determinism to the generation (see Section 6.3 and the system described in Section 7.3.1.2 as an example of use).

Note that the ReLU function is routinely used as the activation function in hidden layers, in order to introduce non linearity. But ReLU is never used as an activation function of the output layer (as it does not have the "separation" effect of the sigmoid).

5.3.1.2 Cost Function

The most common cost functions are *quadratic cost* (also known as *mean squared error* or *maximum likelihood*), *cross-entropy cost*, *Kullback-Leibler divergence* (*KL-divergence*). The choice of a cost function is actually very much related to the choice of output activation function. For instance, for a prediction (regression) task, a quadratic cost function is often chosen, joint with an identity of a sigmoid activation function. For a classification task, a cross-entropy cost function is often chosen. More details can be found, e.g., in [29, Chapter 6].

5.3.1.3 Feedforward Propagation

Feedforward propagation in a multilayer neural network consists in injecting input data on the output layer and propagating the computation through its successive layers until producing the output. This is very efficient, because it consists in successive matrix products (intercalated with AF activation function calls), with a recursive computation of the output of layer k as in Equation 5.6.

$$output^{(k)} = AF(b^{(k)} + W^{(k)} output^{(k-1)}) \tag{5.6}$$

Multilayer neural networks are therefore often also named *Feedforward neural networks* or also *Multilayer Perceptron* (MLP)[17].

Note that neural networks are *deterministic*. This means that the same input will *deterministically always* produce the *same* output. This is a guarantee for prediction and classification purposes, but this may turn out to be a limitation considering our objective of generating new content.

Therefore some techniques based on sampling (see Section 6.3) may be introduced to introduce *variability* over the generation process. The basic idea (already mentioned in Section 5.3.1.1) is as following: for a softmax output activation function (usually associated to a one-hot encoding), rather than selecting the value with the highest probability as for

[16] Variable z is traditionally used to represent the output values (z_j), as y is reserved for the actual data.

[17] The original Perceptron was a neural network with no hidden layer, thus equivalent to our basic building block, with only one output node and with the step function as the activation function.

a deterministic generation, we may sample from the probability distribution represented by the softmax, in order to add variability and non determinism to the generation (see an example in Section 7.3.1.2).

5.3.1.4 Training a Neural Network

For the training phase, computing the derivatives becomes a bit more complex than for a basic building block (without hidden layer) as presented in Section 5.1.3. *Back-propagation* is the standard method to estimate the derivatives (gradients) for a multi-layer neural network, based on the *chain rule* [90] principle, in order to estimate the contribution by each weight to the final prediction error (see, e.g., [29, Chapter 6] for more details).

Note that, in the most common case, the cost function of a multilayer neural network is *not convex*, meaning that there are *multiple local minima*. Gradient descent, as well as other more sophisticated heuristics optimization methods, does not guarantee reaching the global optimum. But in practice a clever configuration of the model and the architecture (notably, its *hyperparameters*, see Section 5.4) and well tuned optimization heuristics, such as stochastic gradient descent (SGD), will lead to accurate solutions[18].

5.3.2 Recurrent Neural Network (RNN)

A *Recurrent Neural Network* (RNN) is a (feedforward) neural network extended to include *recurrent connexions*. This actually means a significant change in the way they are used. The idea is in considering not an input x and an output y but a *serie* (sequence) of inputs x_t and outputs y_t, indexed by a parameter t which represents the *index* or the *time*. The basic idea is that the outputs of a hidden layer (h_t) *reenter* into itself (with a specific weight matrix, that we note here W_r) as an additional input to compute next values (h_{t+1}) of the hidden layer. This way, the network can learn, not only based on *current* data, but also on *previous* one. Therefore, it can learn *series*, notably *temporal series* (the case of musical content).

We can see at Figure 5.11 the added *recurrent connexions*, signaled with a special little solid square (in order to distinguish them from *standard connexions*).

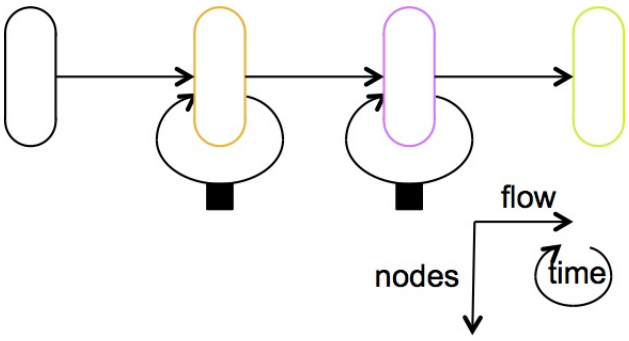

Fig. 5.11 Recurrent neural network (Folded).

[18] On this issue, see [12], which shows that: 1) local minima are located in a well-defined band; 2) SGD converges to that band; 3) reaching the global minimum becomes harder as the network size increases; 4) but that it is in practice irrelevant as global minimum often leads to overfitting.

The unfolded version is at Figure 5.12 and we represent it with a new axis representing the time dimension to illustrate the previous values of each layer (in lighter display).

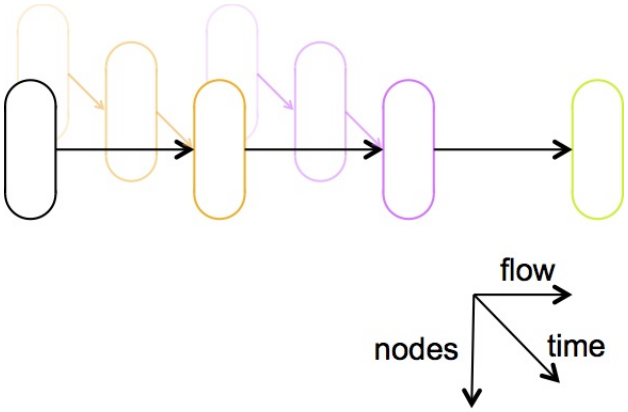

Fig. 5.12 Recurrent neural network (Unfolded).

Last, Figure 5.13 details the recurrent connexions for a given hidden layer (the second hidden layer). Recurrent connexions are in solid lines while standard connexions are in dashed lines. Note that a recurrent connexion fully connects all previous node states to current node states. The recurrent connexions matrix represents *parameter sharing* and relies on the assumption that the same parameters can be used for different time steps.

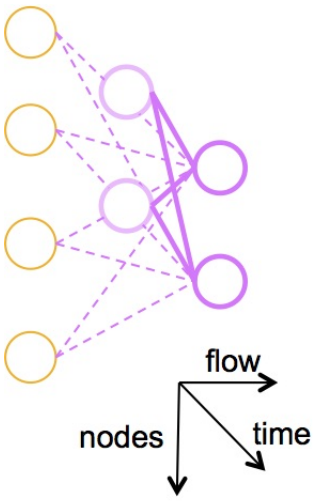

Fig. 5.13 Unfolded Recurrent layer.

Note that a recurrent network usually has identical input layer and output layer, as a recurrent network predicts next item, which will be used as next input in an iterative way in order to produce sequences.

A RNN can learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence, the output at each time step t being the conditional distribution $P(x_t|x_{t-1},...,x_1)$. In summary, recurrent networks (RNNs) are good at learning sequences and therefore are routinely used for natural text processing and for music generation.

A recurrent network is not trained as feedforward networks. The idea is to present an example element of a sequence (e.g., a note for learning melodies) as the input (x_t) and the actual next element of the sequence as the output (x_{t+1}). This will train the recurrent network to predict next element of the sequence. In practice, a RNN is not trained element by element, but with a sequence as an input and the same sequence shifted left of one item as the output. Therefore, the recurrent network will learn to predict next element for all successive elements of the sequence (and this will be done for various sequences).

5.3.3 Long Short-Term Memory (LSTM)

Recurrent networks suffered from a training problem caused by the difficulty to estimate gradients, because in *back-propagation through time (BPTT)*, recurrence brings repetitive multiplications and could thus lead to over *minimize* or *amplify* effects (this is called the *vanishing* or *exploding* gradient problem). This problem has been addressed and resolved by the *Long short-term memory (LSTM)* architecture, proposed by Hochreiter and Schmidhuber in 1997 [44]. As the solution has been quite effective, LSTM became the *de facto* standard for recurrent networks[19].

The idea behind LSTM is to secure information in protected memory *cells*, within a *block*[20], protected from the standard data flow of the recurrent network. Decisions about when *writing* to, *reading* from and *forgetting* (*erasing*), through the opening or closing of *gates*, are expressed at a distinct control level (*meta-level*), while being learnt during the training process. Therefore, gates are modulated by a *weight*, thus *differentiable* and suitable for back-propagation and standard learning process – each LSTM block learns how to maintain its memory as a function of its input in order to minimize loss. See a conceptual view of a LSTM cell at Figure 5.14[21]. We will not further detail here the inner mechanism of a LSTM cell (and block), because we may consider it here as a *black box* (please refer to, e.g., the original article [44]).

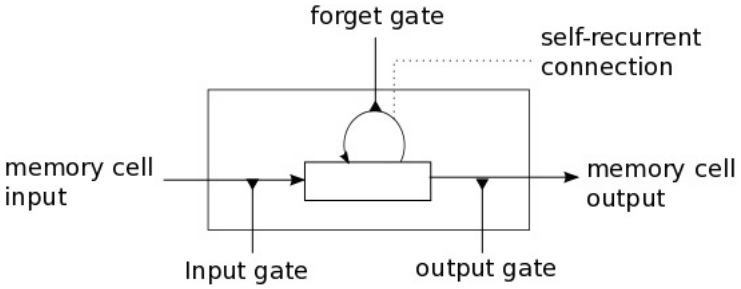

Fig. 5.14 LSTM architecture.

Note that a more general model of *memory* with access customized through training has been recently proposed: *Neural Turing Machines (NTM)* [32]. In this model, memory is global and has *read* and *write operations* with differentiable controls, thus subject to learning through back-propagation. The memory to be accessed, specified by *location* or by *contents*, is controlled via an *attention* (*focusing*) mechanism.

---

[19] Although, there are a few subsequent analog proposals, such as *Gated Recurrent Units (GRU)*, but LSTM tend to remain the *de facto* standard. See a comparative analysis of LSTM and GRU in [13].

[20] Cells within a same block *share* input, output and forget gates, i.e. although each cell might hold a different value in its memory, all cell memories within a block are read, written or erased *all at once* [44].

[21] Reproduced from [101].

5.3.4 Autoencoder

An *autoencoder* is a neural network with one hidden layer and with an additional *constraint*: the number of output nodes is equal to the number of input nodes[22]. The output layer actually *mirrors* the input layer. It is shown at Figure 5.15, with its specific symmetric diabolo or sand-timer shape aspect.

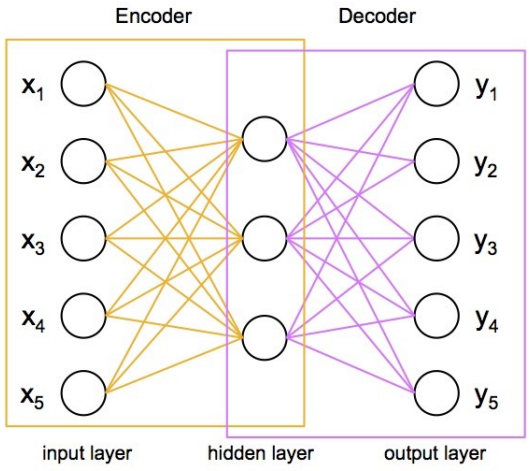

Fig. 5.15 Autoencoder.

Training an autoencoder is done by using conventional *supervised* learning, but with the output equal to the input[23]. In practice, the autoencoder tries to learn the *identity* function. As the hidden layer has often fewer nodes than the input layer, the *encoder* part has to compress information and the *decoder* part has to reconstruct, as well as possible, the initial information[24]. This forces the autoencoder to *discover* significant (discriminating) *features* to *encode* useful information (into the hidden layer, composed of what is sometimes named *latent variables*). In order to further guide the autoencoder, additional constraints may be used, such as activation *sparsity*[25], that is that at most one node (neuron) is active, in order to enforce *specialization* of each node of the hidden layer as a specific *feature detector*. Therefore, autoencoders may be used to automatically extract higher level *features* [63]. The set of features extracted are often also named an *embedding*[26]. Once trained, in order to extract features from an input, one just needs to feed forward the input data and gather the activations. As we will see in Section 7.1.3.1, one may use hierarchical stacks of autoencoders (named *stacked autoencoders*, see Section 5.3.5) to progressively extract more abstract features.

[22] The bias is not counted/considered here, as it is an implicit additional input node. Also, for simplification, biases are not represented on the figure.

[23] This is sometimes called *self-taught* (or *self-supervised*) learning [63].

[24] Compared to traditional dimension reduction algorithms, such as *Principal Component Analysis (PCA)*, this approach has 2 advantages: the number of features may be arbitrary (and not necessarily smaller than the number of input parameters) and feature extraction is non linear.

[25] They are named *sparse autoencoders*.

[26] See the definition of embedding in Section 4.4.4.

5.3.5 Stacked Autoencoders

The idea of *stacked autoencoders* is to hierarchically nest successive autoencoders with decreasing hidden layer (decreasing numbers of hidden units), as illustrated at Figure 5.16.

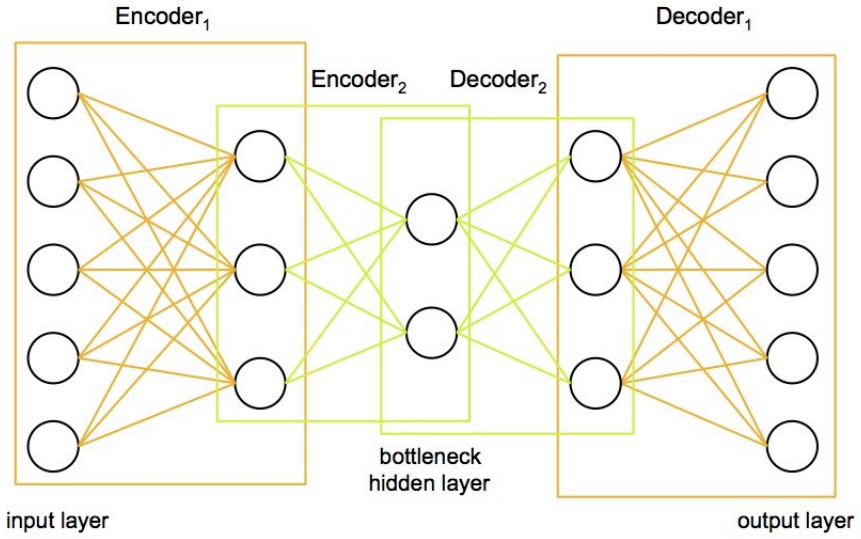

Fig. 5.16 Stacked autoencoders architecture.

The chain of encoders will increasingly compress data and extract higher-level features. Stacked autoencoders (which are indeed deep networks) are therefore used for feature extraction. They are also useful for music generation, as we will see in Section 6.1.3, e.g., in the DeepHear system in Section 7.1.3.1. This is because the *innermost* hidden layer, sometimes named *bottleneck hidden layer*, provides a compact and high-level encoding as a seed for generation (by the chain of decoders).

5.3.6 Restricted Boltzmann Machine (RBM)

A *Restricted Boltzmann Machine* (*RBM*) [42] is a *generative stochastic* artificial neural network that can learn a *probability distribution* over its set of inputs. Its name comes as it is a restricted (constrained) form[27] of (general) *Boltzmann Machines* [43], named after the *Boltzmann distribution* in statistical mechanics, which is used in their sampling function. The architectural restriction of a RBM (see at Figure 5.17) is that:

- they are organized in *layers*, just as neural networks and autoencoders are, and more precisely two layers:
  - the *visible* layer (analog to both the input layer and the output layer)
  - and the *hidden* layer (analog to the hidden layer of an autoencoder);
- and, as for standard neural networks, there cannot be connections between nodes within a same layer.

[27] Which actually makes them practical, as opposed to the general form, that, besides its interest suffers from a learning scalability limitation.

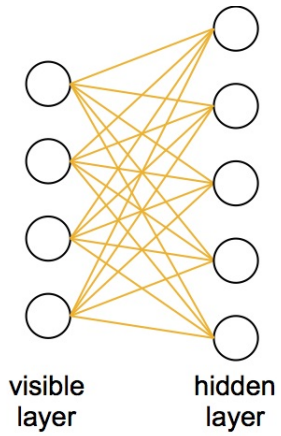

Fig. 5.17 Restricted Boltzmann Machine architecture.

A RBM bears some similarity in spirit and objective with an autoencoder. However, there are two main differences:

- a RBM has *no ouput* – the input acting also as the output;
- a RBM is *stochastic* (and therefore, *not deterministic*, as opposed to neural networks and autoencoders;
- values manipulated are *booleans*.

RBMs became popular after Hinton used them for *pre-training* deep neural networks [23], after designing a fast specific learning algorithm for them, named *contrastive divergence* [39].

A RBM is an architecture dedicated to learn distributions. Moreover, it can learn efficiently with few examples. For musical applications, this is interesting for, e.g., learning (and generating) chords, as the combinatorial of possible notes forming a chord is large and the amount of examples is usually small. See an example of application in Section 7.2.1.1.

Training a RBM has some similarity to the training of an autoencoder, with the practical difference that because there is no decoder part with an output layer mirroring the input layer, the RBM will alternate between two steps:

- *feedforward step*, to encode the input (visible layer) into the hidden layer, by making predictions about hidden layer nodes activations,
- and *backward step*, to decode/reconstruct back the input (visible layer), by making predictions about visible layer nodes activations.

We will not detail the technique behind RBM here. Note that the reconstruction process does something different from the case of autoencoders (based on regression for prediction or classification), and is known as *generative learning*[28].

After the training phase has been done, in the *generation* phase, a *sample* can be drawn from the model by randomly initializing visible layer vector v (following a standard uniform distribution) and running *sampling*[29] (see Section 6.3) until convergence. To this end, as for the training phase, hidden nodes and visible nodes are alternately updated (as during the training phase). In practice, convergence is reached when the *energy* stabilizes[30].

[28] See, e.g., a nice introduction in [78].

[29] More precisely *Gibbs sampling (GS)*, see [59].

[30] The *energy* of a *configuration* (the pair of vectors of values of visible and hidden layer nodes). is expressed as $E(v,h) = -a^{\mathrm{T}}v - b^{\mathrm{T}}h - v^{\mathrm{T}}Wh$, where v and h, respectively, are the visible and the hidden layers, W is the matrix of weights associated with the connections between visible and hidden nodes and a and b, respectively, the bias weights for visible and hidden nodes. For more details, see, e.g., [29, Section 16.2.4].

In the standard RBM, units (visible and hidden) are Boolean (with a *Bernoulli distribution*). One extension is with *multinomial* (with more than 2 discrete values) visible units, although the hidden units are Booleans. In this case, the logistic function for visible units is replaced by the softmax function. A further extension is with continuous units (visible and hidden), taking arbitrary real values (usually within the [0 1] range), see e.g., the C-RBM system in Section 7.7.1.1.

5.3.7 Variational Autoencoder

Variational Autoencoders *(VAE)* [57] are getting increasing attention because of their generative built-in capacities. The intuition is actually relatively simple: the idea is to enforce that the encoded representation in the hidden layer of the autoencoder (its latent variables) follows a *Gaussian distribution*[31]. This is enforced by adding a specific term to the cost function, computing the cross-entropy between the values of the latent variables and a Gaussian distribution[32]. The Gaussian distribution is controlled by some σ hyperparameter, the standard deviation of the Gaussian distribution. A small value will make the model learnt closer to the actual examples. For more details about VAEs, a nice tutorial is [19].

As for an autoencoder, the VAE will learn the identity, but furthermore the decoder part will learn the relation between a Gaussian distribution of the latent variables and the learnt examples. As a result, sampling from the VAE is immediate, one just needs to sample a value following a Gaussian distribution for the hidden layer (latent variables), input it into the decoder and feedforward the decoder to generate an output corresponding to the distribution of the examples. This is in contrast to the need for indirect and computationally expensive strategies such as Gibbs sampling for other architectures such as RBMs (see Section 5.3.6).

Variational autoencoders are elegant. As a result, they are one of the current approach explored for generating content. An example of use to generate musical content is presented in Section 7.3.4.1.

5.3.8 Convolutional Architectural Pattern

Convolutional architectures for deep learning have become a common place for *image applications*. The concept has been originally inspired from a model of human vision and carefully adapted and improved to neural networks by Le Cun, notably for handwritten character and object recognition [17], resulting in efficient results, by exploiting the strong spatially local *correlation* present in natural images.

The basic idea[33] is for the network not to learn from *individual elements* (in the case of images, *pixels*), but from an area (named a *convolution*), formed by *nearby elements*, while sharing the *same connexion weights* for this whole area. Figure 5.18[34] shows a conceptual representation of the basic principle of a convolution.

Note that convolutions are less frequent for music applications. The reason is, we believe, that it is not as effective as for images where motives are invariant in all dimensions.

---

[31] Actually other distribution models may be used, but Gaussian distribution (also named Gaussian law or *normal law*) is usually chosen for its generality.

[32] The actual implementation is more complex and has some tricks (e.g., the encoder actually generates a mean vector and a standard deviation vector), that we will not present here.

[33] For more information, including about *hyperparameters* named *depth*, *stride* and *zero-padding*, as well as the types of *pooling*, see, e.g., [65] or [29, Chapter 9].

[34] Reproduced from [3].

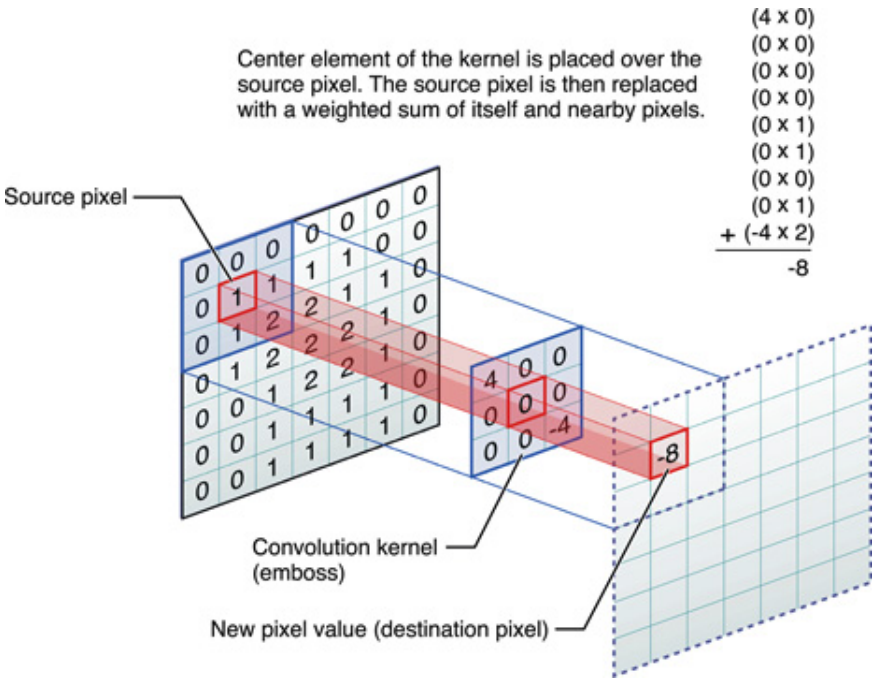

Fig. 5.18 Convolution.

For musical applications, it could be interesting to apply convolutions to the *time dimension*, in order to model temporally invariant motives. This convolutional approach is actually the basis for *time-delay neural networks* [60]. The convolution operation allows a network to share parameters across time [29, page 374], like for RNNs[35], but it is shallow (it applies to only a small number of temporal neighboring members of the input). This is in contrast to RNNs which share parameters in a deep way (for all time steps). Therefore, RNNs are much more frequent than convolutional networks for musical applications.

Meanwhile, we notice the recent occurrence of some convolutional architectures as an alternative to RNN architectures, following the pioneering *WaveNet* system [108]. They present a stack of causal convolutional layers, somehow analog to recurrent layers (see details in [108]). An adaptation from audio to MIDI is the recent *MidiNet* system [119]. Another example is the *C-RBM Convolutional restricted Boltzmann machine* architecture described in Section 7.7.1.1.

If we now consider the *pitch dimension*, in most cases pitch intervals are not considered metrically invariants, thus convolutions should not *a priori* apply to the pitch dimension. An exception is Johnson's architecture [53], analyzed in Section 7.3.3.1, that explicitly looks for invariance in pitch (although this seems to be a rare choice), and accordingly uses a convolutional RNN over the pitch dimension.

This issue of using or not convolutions for musical applications will be further discussed in Section 10.1.

Note that convolutional is an *architectural pattern*, as it may be applied internally to almost any architecture listed[36].

[35] Indeed RNNs are invariant in time, as remarked in [53], since each time step is a single iteration of the network.

[36] As discussed above, even a *Convolutional RNN* could be used as long as convolution is not on the time dimension, otherwise its time invariance would compete with the time invariance offered by the RNN.

5.3.9 *Conditioning* Architectural Pattern

The idea of conditioning (sometimes also named *conditional architecture*) is to *condition* the architecture on some extra *conditioning* information, which could be arbitrary, e.g., a class label or data from other modalities. Examples are: a bass line or a beat structure, in the rhythm generation system described in Section 7.4.1.1; a chord progression in the MidiNet architecture described in Section 7.9.1.1; a musical genre or an instrument, in the WaveNet architecture described in Section 7.5.1.1. The objective is to have some control over the data generation process.

In practice, the conditioning information is usually fed into the architecture as an additional input layer, see an illustration at Figure 5.19[37]. This distinction between standard input and conditioning input follows a good architectural modularity principle.

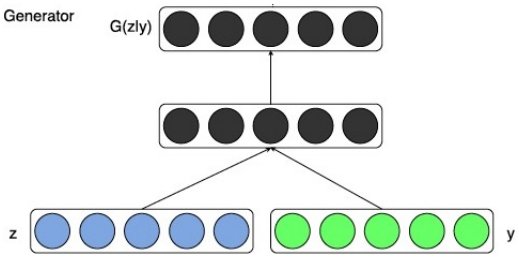

Fig. 5.19 Conditioning architecture

Note that in case of a *conditioning recurrent* architecture, there are two options: *global conditioning* or *local conditioning*, depending if the c conditioning input is shared for all time steps or is specific to each time step (c_t), see e.g., the WaveNet architecture[38] in Section 7.5.1.1.

5.3.10 *Generative Adversarial Networks (GAN)* Architectural Pattern

A significant conceptual and technical innovation was introduced in 2014 by Goodfellow et al. with *Generative adversarial networks* (GAN) [30]. The conceptual idea is to train simultaneously two networks[39] (see at Figure 5.20[40]):

- a *generative model* (or *generator*) G, whose objective is to transform random noise vectors into faked *samples*, which resemble real samples drawn from a distribution of real images
- and a *discriminative model* (or *discriminator*) D, that estimates the probability that a sample came from the training data rather than from G.

This corresponds to a *minimax* two-player game, with one unique (final) solution: G recovers the training data distribution and D outputs $1/2$ everywhere. The generator is then

[37] This figure is a part of an abstract representation of the Conditional GAN architecture [72], a first attempt at combining conditioning and adversarial architectural patterns.

[38] Actually, WaveNet is not a recurrent architecture, but a time-invariant convolutional architecture, in practice almost equivalent to a RNN.

[39] In the original version, two feedback networks are used. But we will see that other networks may be used, e.g., recurrent networks in the C-RNN-GAN architecture (see Section 7.8.1.1) and convolutional feedforward networks in the MidiNet architecture (see Section 7.9.1.1).

[40] Reproduced from [1].

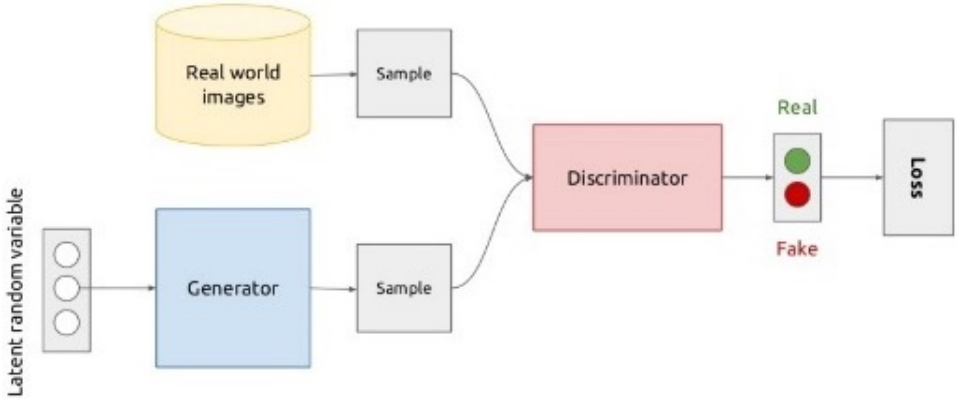

Fig. 5.20 Generative adversarial networks architecture

able to produce user-appealing synthetic samples from noise vectors. The discriminator may then be discarded.

$$\min_G \max_D V(G,D) = log(D(x)) + log(1 - D(G(z))) \tag{5.7}$$

In the minimax equation (Equation 5.7):

- $V(G,D)$ is the objective, which D will try to maximize and which G will try to minimize;
- $D(x)$ represents the probability (according to D) that input x came from the real data;
- $D(G(z))$ represents the probability (according to D) that input $G(z)$ has been produced by G from z random noise;
- $1 - D(G(z))$ represents the probability (according to D) that input $G(z)$ has *not* been produced by G from z random noise;

It is thus D's objective to classify correctly real data (maximize $(D(x)$ term) as well as synthetic data (maximize $1 - D(G(z))$ term), thus to maximize the sum of the two terms: $V(G,D)$. On the opposite, the generator's objective is to minimize $V(G,D)$. Actual training is organized with successive turns between the training of the generator and the training of the discriminator.

One of the initial motivation for GAN is for classification tasks to better prevent adversaries to manipulate deep networks to force misclassification of inputs (this vulnerability is analyzed in detail in [100]). But it also improves the generation of samples hard to distinguish from the actual corpus examples, thus addresses the generation task (which is our focus here).

Note that training based on a minimax objective is known to be challenging to optimize [118], with risks of non converging oscillations. Thus, careful selection of the model and its hyperparameters are important [29, page 701]. There are also some improved techniques, such as *feature matching* and others, to improve training [92].

To generate music, random noise is used as an input to the generator G, whose goal is to transform random noises into the objective, e.g., melodies. An example of the use of GAN for generating music (melodies) is the MidiNet system, described in Section 7.9.1.1.

5.4 Hyperparameters

In addition to the *parameters* of the model, which are the weights of the connexions between nodes, a model includes also *hyperparameters*, which are parameters at an (*architectural meta-level*, about both *structure* and *control*). Examples of *structural* hyperparameters

are: number of layers, number of nodes, non linear activation used. Examples of *control* hyperparameters, mainly concerned with the learning process, are: optimization function, learning rate, regularization strategy, regularization parameters. A good setting of values for hyperparameters is fundamental to the efficiency and accuracy of neural networks for a given application, a *grid search* being a possible general engineering strategy for hyperparameters optimization.

5.5 Reinforcement Learning

Reinforcement learning (RL) may appear at first look a bit outside of our interest in deep learning architectures, as it has distinct objectives and models. Meanwhile, recently the two approaches have been combined. The first move, in 2013, was to use deep learning architectures to implement efficiently reinforcement learning techniques, resulting in *deep reinforcement learning* [74]. The second move, in 2016, is directly related to our concerns, as it explores the use of reinforcement learning to control music generation, resulting in the *RL-Tuner* architecture [52].

Let us start by some reminder of the basic concepts of reinforcement learning, illustrated at Figure 5.21[41]:

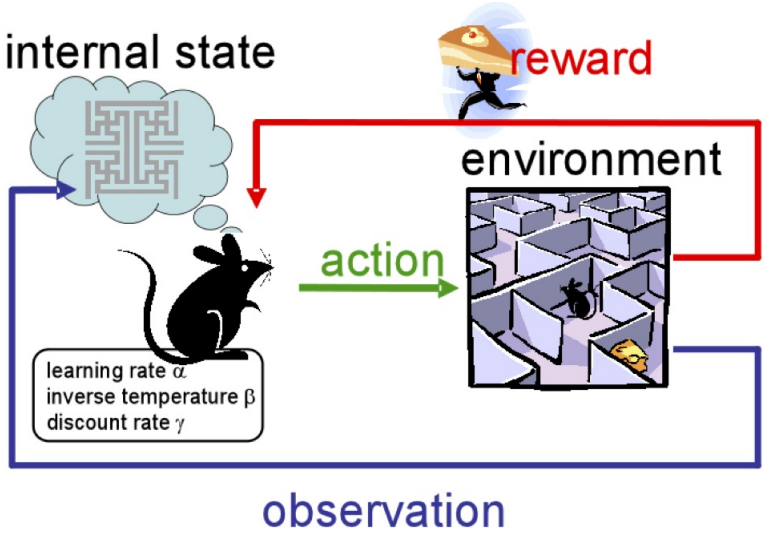

Fig. 5.21 Reinforcement learning – Conceptual model.

- An *agent* within an *environment* sequentially selects and performs *actions* in some environment,
- each action performed brings it to a new *state*,
- with the reception (by the agent) of a *reward* (*reinforcement signal*), which represents some *adequation* of the action to the environment (the situation).
- The objective of *reinforcement learning* is for the agent to learn a near optimal *policy* (sequence of actions) in order to maximize its *cumulated rewards* (named its *gain*).

There are many approaches and algorithms for reinforcement learning (for a more detailed presentation, please refer to, e.g., [54]). Among them, *Q-learning* [34] turned out

[41] Reproduced from [20].

being a relatively simple and efficient method, thus widely used. The name comes from the objective to learn (estimate) the Q function $Q^*(s,a)$, which represents the expected gain for a given pair $< s,a >$, where s is a state and a an action, and then continuing by choosing actions optimally. The agent will manage a table, called the *Q-Table*, with values corresponding to all possible pairs. As long as the agent incrementally explores the environment, the table is updated with hopefully increasingly accurate expected values.

A recent combination of reinforcement learning (more specifically Q-learning) and deep learning, named *deep reinforcement learning* has been proposed [74] in order to make learning more efficient. As the Q-Table could be huge, the idea is to use a deep neural network in order to approximate the expected values of the Q-Table through the learning of many replayed experiences.

A further optimization, named *Double Q-learning* [111] *decouples* the *action selection* from the *evaluation*, in order to avoid value overestimation. The task of the Target Q-Network is to estimate the gain (Q), while the task of the Q-Network is to select next action.

Reinforcement learning appears as a promising approach for incremental adaptation of the music generated, e.g., based on the *feedback* from listeners (this issue will be addressed in Section 10.7). Meanwhile, some significant move was made in using reinforcement learning to inject control into the generation of music by deep learning architectures, through the reward mechanism, as described in Section 6.6 and exemplified by the RL-Tuner architecture (see Section 7.10.1.1).

5.6 Compound Architectures

Often *compound* architectures are used. Some cases are *homogeneous* compound architectures, combining various instances of a same architecture, e.g., *stacked autoencoders* (see Section 5.3.5), and most cases are *heterogeneous* compound architectures, combining various types of architectures, e.g., a *recurrent autoencoder* (also named *RNN Encoder-Decoder*, see Section 5.6.2).

5.6.1 Composition Types

We will see that, from an architectural point of view, various types of composition[42] may be used:

- *composition* – At least two architectures, of the same type or of different types, are combined. An example is the RNN-RBM architecture, combining a RNN architecture and a RBM architecture (see Section 5.6.3);
- *nesting* – One architecture is nested into the other one. An example is a Stacked Autoencoders architecture (see Section 5.3.5). Another example is the RNN Encoder-Decoder architecture, where two RNN architectures are nested within the encoder part and the decoder part of an autoencoder (see Section 5.6.2);
- *integration* – Two types of architectures are merged within a single one. An example is Johnson's Hexahedria architecture, where a feedforward architecture and a RNN architecture are merged (see Section 7.3.3.1);
- *pattern instantiation* – An architectural pattern is instantiated onto some given architecture(s). A first example is the C-RBM architecture, which applies the convolutional

[42] We are taking inspiration from concepts and terminology from programming languages and software architectures [94], such as: *feature*, *instantiation*, *nesting* and *pattern* [26].

architectural pattern is instantiated *into* a RBM architecture (see Section 7.25). Another example is the C-RNN-GAN architecture, where the GAN (generative adversarial networks architectural pattern is instantiated *onto* a RNN architecture (see Section 7.8.1.1). We could therefore denote it as a GAN(RNN) architecture.

5.6.2 RNN Encoder-Decoder

The idea of encapsulating two identical recurrent networks (RNNs) into an autoencoder was initially proposed in [10] as a technique to encode a variable-length sequence learnt by a recurrent network into another variable-length sequence produced by another recurrent network. The motivation and application target is translation from a language to another language, resulting in sentences of possibly different lengths. The idea is to use a fixed-length vector representation as a pivot representation through some encoding and decoding architecture. Therefore they name this architecture a *RNN Encoder-Decoder*, illustrated at Figure 5.22, with circles representing the successive hidden states of the encoder and of the decoder.

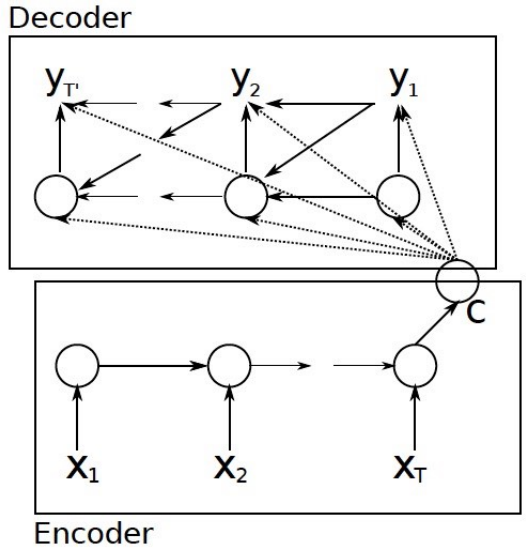

Fig. 5.22 RNN Encoder-Decoder architecture.

An example of use for music generation is the VRAE system described in Section 7.3.4.1.

5.6.3 Polyphonic Recurrent Network

The *RNN-RBM* architecture combines a RBM architecture and a recurrent (LSTM) architecture, by *coupling* them to associate the vertical perspective (simultaneous notes) with the horizontal perspective (temporal sequences of notes) of a polyphony to be generated (see Figure 7.18 and Section 7.3.2.1).

5.6.4 Further Compound Architectures

One may further combine architectures already compound, examples are:

- *Convolutional Generative Adversarial Networks* – a *Generative Adversarial Networks* (GAN) architecture with *convolution*, summarized as GAN(Convolutional(Feedforward)), an example being the MidiNet architecture, see Section 7.9.1.1;
- *Recurrent Generative Adversarial Networks* – a *Generative Adversarial Networks* (GAN) architecture where both the generator and the discriminator are *recurrent network architectures* (RNN), summarized as GAN(Conditional(RNN)), an example being the C-RNN-GAN architecture, see Section 7.8.1.1.

There are also some more specific (ad-hoc) compound architecture, as for instance:

- Johnson's Hexahedria architecture combines intimately a feedforward architecture with a recurrent architecture, by *integrating* them, as an alternative to the RNN-RBM architecture (See Section 7.3.3.1);
- The *DeepBach* architecture combines two feedforward architectures with two recurrent architectures (see Section 7.2.2.1).

Chapter 6
Strategy

A strategy is the way the deep learning architecture is used to attain the objective. We have surveyed and analyzed various systems and experiments using deep learning architectures to generate musical content (see Section 7) and derived in a *bottom-up* way several *strategies*. The most *straightforward* strategy of generation is using a feedforward architecture in a feedforward mode to generate music, but there are several other ways.

We will see that there are some obvious relations between architectures and strategies, e.g., a feedforward network architecture is generally used with the *feedforward strategy* and a RBM architecture is used with the *sampling strategy*. But we will also see that a given strategy may apply to various architectures, e.g., the *feedforward strategy* may be applied to a feedforward network architecture, but also to a recurrent architecture, an autoencoder and a generative adversarial networks architecture, and an architecture may be used with more than one strategy, e.g., a RBM may be used with the *sampling strategy* or with the *input manipulation strategy*. This motivates this fourth dimension classifying the strategy.

6.1 Feedforward

The most direct strategy is using the *prediction* or *classification* task of a neural network to generate a musical content. After having trained the neural network with a dataset of examples corresponding to a corpus, we *feedforward* the network with an example of input, e.g., a melody, in order to produce a corresponding output, e.g., an accompaniment.

6.1.1 Single-Step Feedforward

If the architecture is a *feedforward network*, generation is done in a *single step* of feedforward. Therefore, we name this strategy *single-step feedforward*. An example is the generation of a chorale as an accompaniment in a counterpoint manner of a given melody by the MiniBach system, described in Section 7.1.1.1.

6.1.2 Iterative Feedforward

The *recurrent* nature of a *recurrent network* (RNN), which learns to predict next step information, can be used to generate sequences of *arbitrary* lengths[1]. The typical usage is to

[1] The length of the sequence produced is not predefined, as it depends on how many iterations are done.

enter some *seed* information as the first item (e.g., the first note of a melody), to generate by feedforward the next item (e.g., next note), which in turn is used as an input and so on, until producing a sequence of the length desired. Therefore, we name this strategy *iterative feedforward*. An example is the generation of melodies described in Section 7.1.2.1.

6.1.3 Decoder Feedforward

The *decoder feedforward strategy* uses an autoencoder to extract *features*[2] from a corpus of musical content through the encoding process. In essence, the hierarchical encoding/decoding of the corpus is supposed to have extracted discriminating features of the corpus[3], thus characterizing its variations through a few features.

Generation is done by creating a *seed* value for the hidden layer of the autoencoder. and then feedforward this seed value to its decoder part to *reconstruct* by simple feedforward a corresponding musical content, with the same format (number of parameters) as the output layer of the autoencoder[4].

The strategy is more effective when using hierarchically nested autoencoders, having successive encoding and then mirrored decoding of *hierarchical* abstractions. This leads to a quite compact and high-level innermost hidden layer (the *bottleneck hidden layer*, see Section 5.3.5). This bottleneck hidden layer is used for inserting the seed value. Therefore, only a simple seed information inserted at the exact middle of the encoder/decoder stack (see Figure 7.10) can generate an arbitrarily complex and long musical content. An example is the generation of melodies by the DeepHear system described in Section 7.1.3.1.

This strategy may also be applied in an iterative way to a *recurrent autoencoder*, such as the *RNN Encoder-Decoder* architecture (see Section 5.6.2), by inserting a seed into the decoder (which in this case is a recurrent network) to produce a sequence. An example is the generation of melodies by the VRAE system, described in Section 7.3.4.1.

6.2 Conditioning

The concept of conditioning has been introduced as an architectural pattern in Section 5.3.9. We consider also an associated strategy, that we name the *conditioning strategy*, because this does not only involve the architecture, but it is also an approach to parameterize and control the generation of musical content.

6.3 Sampling

Sampling is the action of generating an element (a *sample*) from a *stochastic* model with a given *probability distribution*. An example is the generation of a sequence from a *Markov chain*. The main issue is to ensure that the samples generated match a given distribution. Therefore, various sampling strategies have been proposed: *Metropolis-Hastings*, *Gibbs (GS)*, *block Gibbs sampling*, etc. Please see, e.g., [29, Chapter 17] for details about sampling and various sampling algorithms.

---

[2] Also sometimes named an *embedding*, see Section 4.4.4.

[3] Therefore, to enforce this specialization, sparse autoencoders are often used (see Section 5.3.4).

[4] Because for an autoencoder, input and output formats are identical, this means also with the same format as the input layer, i.e., the trained corpus.

We just introduce the idea of one of the most basic sampling strategies, the *Metropolis-Hastings* algorithm. It works by generating a sequence of sample values in such a way that, as more and more sample values are generated, the distribution of values more closely approximates the target distribution. Sample values are produced *iteratively*, with the distribution of the next sample being dependent only on the current sample value[5]. Each successive sample is generated through a *generate-and-test* strategy, i.e. to generate a prospective candidate and accept or reject (based on a defined *probability density*) and regenerate it.

For musical content, we may consider two different levels:

- *item level* or *vertical* dimension – at the level of a compound musical item, e.g., a chord. In this case, the distribution is about the relations between the components of the chord, i.e. describing the probability of notes to occur together;
- *sequence* level or *horizontal* dimension – at the level of a sequence of items, e.g., a melody composed of successive notes. In that case, the distribution is about the sequence of notes, i.e. it describes the probability of the occurrence of a specific note after a given note.

The RBM (Restricted Boltzmann Machine) architecture is generally used for the vertical dimension, which notes should be played together[6]. An example of sampling (more precisely, block Gibbs sampling) from a RBM to generate chords is in Section 7.2.1.1.

Typically, a RNN architecture is used for the horizontal dimension. In order to be able to sample, the output of the RNN should be a probability distribution, i.e. the output activation of the RNN should be a *softmax*. In a deterministic manner, one would simply select the value with the highest probability. This would deterministically produce (always) the same result (e.g., the same note) from the same input (typically, the previous note, in an iterative feedforward strategy). But the distribution produced by the softmax can also be used as the basis for *sampling* in order to add *variability* and non determinism to the generation. Then, one just needs to sample from the distribution, i.e. select among the possible values depending on their respective probabilities, to generate a stochastic result (for next note). Examples of using such strategy are the CONCERT system described in Section 7.3.1.1 and the system described in Section 7.3.1.2.

An interesting combination of the vertical and the horizontal dimensions is the RNN-RBM two-level architecture, described in Section 7.3.2.1. In this system, sampling at the *item/vertical* level – i.e. which notes should be played together – is governed by a RBM and sampling at the *sequence/horizontal* level – i.e. which item comes next in the sequence – is governed by a RNN.

The process of sampling can also be controlled by introducing an external mechanism to restrict the set of possible solutions in the sampling process, according to some pre-defined constraints, as for instance in the C-RBM system, described in Section 7.7.1.1.

6.4 Input Manipulation

This strategy is about *input manipulation*, as has been pioneered for images by DeepDream. The initial input content, or a brand new (randomly generated) input content, is incrementally manipulated in order to match a target property. Examples are:

[5] I.e. theoretically speaking, following Markov constraint, and practically speaking being efficient, as it does not have to refer to all previous samples. Note that the Metropolis-Hastings algorithm family is actually a *Markov chain Monte Carlo (MCMC)* method that relies on repeated random sampling to obtain numerical results, based on iteratively constructing a Markov chain that has the desired *equilibrium distribution*. For more details, see, e.g., [59] or [29, Chapter 17].

[6] As noted in Section 5.3.6, a RBM is an architecture dedicated to learn distributions and that can learn efficiently from few examples. This is particularly interesting for learning and generating chords, as the combinatorial of possible notes forming a chord is large and the amount of examples is usually small.

- *maximizing* the *activation* of a specific *unit*, to exaggerate some visual element specific to this unit, in DeepDream (see Section 9.1.1);
- *maximizing* some *similarity* to a given *target*, in order to create a consonant melody, in DeepHear (see Section 7.6.1.1).

Interestingly, this is done by *reusing* standard *training* mechanisms, namely *back-propagation* to compute the gradients and *gradient descent* to minimize the cost.

6.5 Adversarial

The concept of generative adversarial networks (GAN) has been introduced as an architectural pattern in Section 5.3.10. We consider also an associated strategy, that we name the *adversarial strategy*, because this does not only involve the architecture, but also is a specific approach for generation of musical content.

6.6 Reinforcement

The idea of the *reinforcement strategy* is to reformulate the generation of musical content as a reinforcement learning problem (see Section 5.5), while using the output of a trained recurrent network as an objective and adding user defined constraints as an additional objective, e.g., some tonality rules according to music theory. The motivation is to overcome the limitations of the iterative feedforward strategy which, although it generates a sequence consistent with the learnt corpus, does not offer a way on imposing global constraints on the generation.

Let us consider the case of a melody formulated as a reinforcement learning problem. The state represents the musical content (a partial melody) generated so far and the action represents the selection of next item (a note) to be generated. Let us now consider a recurrent network (RNN) trained on the chosen corpus of melodies. Once trained, the RNN will be used as a reference for the reinforcement learning architecture.

The reward of the reinforcement learning architecture is defined as a combination of two objectives:

- adherence to *what has been learnt*, by measuring the similarity of the action selected, i.e. next note to be generated, to the *note predicted by the recurrent network* in a similar state (partial melody generated so far);
- adherence to *user-defined constraints* (e.g., consistency with current tonality, avoidance of excessive repetitions...), by measuring how well they are fulfilled.

In summary, the reinforcement learning architecture is rewarded to *mimic* the RNN, while being also rewarded to enforce some user-defined constraints.

This strategy has been proposed through the RL-Tune architecture, described in Section 6.6.

6.7 Unit Selection

The *unit selection strategy* is about querying successive *musical units* (e.g., a melody within a measure) from a data base and to *concatenate* them in order to generate some sequence according to some user characteristics. Querying is using features which have been automatically extracted by an autoencoder. Concatenation, i.e. what unit next?, is controlled by

two LSTMs, each one for a different criterium, in order to achieve some balance between direction and transition).

This strategy, as opposed to most of the other ones that are *bottom-up*, is *top-down*, as it starts with a structure and fills it. An example is described in Section 7.11.1.1.

6.8 Compound Strategies

Note that, as for compound architectures (see Section 5.6), strategies can be *combined* for a single system.

An example is in using both the *sampling strategy* and the *input manipulation strategy* in the C-RBM system described in Section 7.7.1.1. Training a RBM is done an iterated alternative of sampling to control the distribution and input manipulation (through gradient descent), to enforce a set of constraints.

Chapter 7
Systems

In the following, we analyze systems and experiments proposed by various researchers within our common analysis framework, based on the specifications and details that each paper provides. Some papers are very well detailed, some not. In some cases, descriptions are incomplete, which forces us to try to infer (*estimate*) missing information[1].

We will list the various systems and experiments that we have analyzed following the ordering of the strategies that we have presented (in Section 6) and the ordering of the architecture dimension as a second order.

7.1 Feedforward Strategy

This is the most naive and direct strategy, using a feedforward architecture in the feedforward mode to produce music. Interestingly, we have not found (yet) an example of such a simple strategy (because the music generated that way has some limitations). Thus, for pedagogic as well as evaluation reasons, we have created our own system and experiment.

Note that this strategy is not appropriate when the objective is to generate a musical content from scratch (with no generation input).

7.1.1 Feedforward Architecture

7.1.1.1 Hadjeres and Briot's MiniBach Chorale Generation System

This system is an example of single step direct feedforward strategy, named *MiniBach* and designed by Gaëtan Hadjeres and Jean-Pierre Briot. It is an accompaniment system that, from a melody, generates accompanying melodies in the style of the corpus learnt, the set of J.S. Bach polyphonic chorales music[2][4]. To compose his chorales, Bach chose various given melodies for a soprano and composed the three additional ones (for alto, tenor and bass) in a *counterpoint* manner.

The architecture, a *feedforward network*, is shown at Figure 7.1. Input layer has 2,832 nodes and output layer has 2,896 nodes. There is a single hidden layer with 200 units. The non linear function used for the hidden layer is ReLU. The output layer activation function

[1] Any further information about a given system and experiment, from the authors of the experiments or from researchers who managed a more thorough analysis, being a commentary and moreover a correction, is very welcome.

[2] The system is actually a strong simplification of the *DeepBach* system (see Section 7.2.2.1), with the same corpus – but a simplified representation – and objective.

is *softmax*. The input and output representations are *piano roll* with *one-hot* encoding. Time quantization (time slice) is set at the 16th note, half of the minimal note duration used in the corpus. The dataset includes 352 examples.

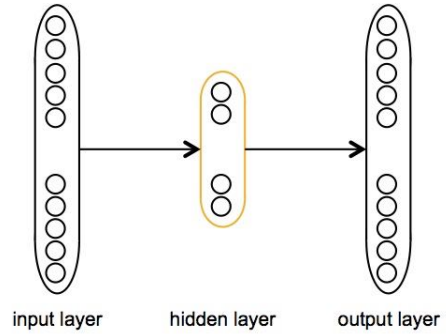

Fig. 7.1 MiniBach architecture.

An example of melody and its three counterpoint voices generated for a single measure is shown at Figure 7.2. The result is acceptable but not as convincing as the refined (and more complex) DeepBach system, presented in Section 7.2.2.1 (the counterpoint voices generated by DeepBach for the same melody are shown at Figure 7.3 as a comparison). Some limitations of MiniBach are its determinism and the fixed duration of the generation.

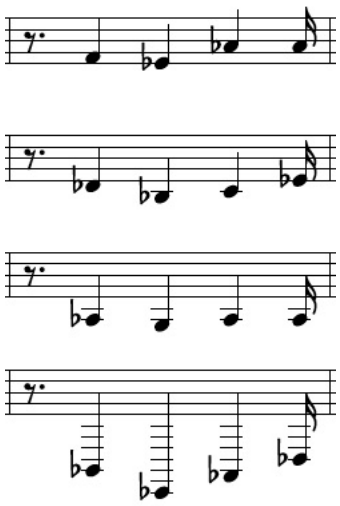

Fig. 7.2 Example of chorale generated by MiniBach.

7.1.2 RNN Architecture

7.1.2.1 Eck and Schmidhuber's Blues Melody Generation System

In [21], Douglas Eck and Jürgen Schmidhuber describe a double experiment done with a *recurrent network* architecture using LSTMs. The format of representation is *piano roll*, with 2 types of sequences: *melody* and *chords*, although chords are represented as notes.

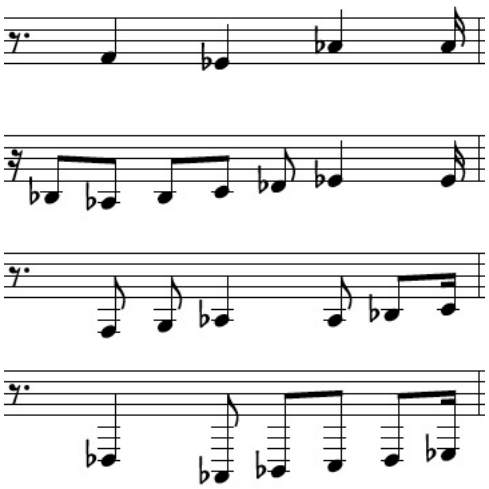

Fig. 7.3 Example of chorale generated by DeepBach.

The melodic range as well as the chord vocabulary is strongly constrained, as the corpus is about 12-bar blues and is handcrafted (melodies and chords). The 13 possible notes extend from middle *C* (C_4) to tenor *C* (C_5). The 12 possible chords extend from *C* to *B*. See at Figure 7.4 the possible melody notes as well as the *chords notes* (chords represented as notes). Figure 7.5 shows a sample of possible chords.

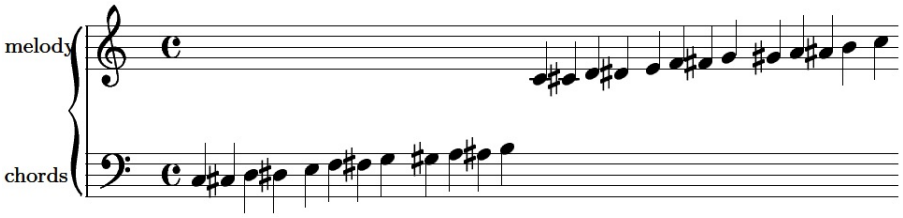

Fig. 7.4 Possible note and chord values.

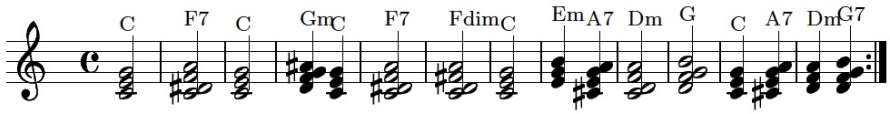

Fig. 7.5 A sample of possible chords.

One-hot encoding is used. Time quantization (time slice) is set at the eighth note, half of the minimal note duration used in the corpus, a quarter note. With 12 measures, this means 96 time steps. An example of chord sequence training example is shown at Figure 7.6.

In the first experiment, the objective is to learn and generate chord sequences. The first architecture used is as following: an input layer with 12 nodes (corresponding to the one-hot encoding of the 12 chords vocabulary), one hidden layer with 4 LSTM blocks containing 2 cells each[3] and an output layer with 12 nodes (identical to the input layer).

[3] The distinctions in a LSTM between cells and blocks is explained in Section 5.3.3.

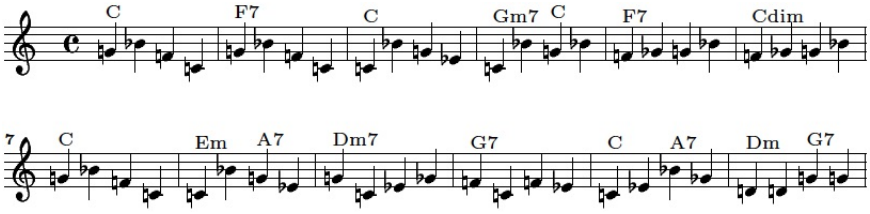

Fig. 7.6 A chord training example.

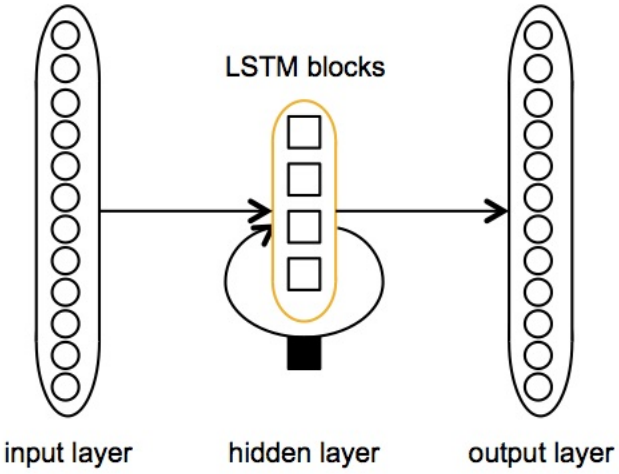

Fig. 7.7 Eck's Blues First version architecture.

Generation is performed by presenting a seed chord (represented by a note) and by iteratively feedforwarding the network, producing the prediction of next time step chord, using it as next input and so on, until generating a sequence of chords.

7.1.2.2 Eck and Schmidhuber Blues Melody and Chord Generation System

In the second experiment, both chords and melody are learnt at the same time. The architecture used is an extension of the previous one: an input layer with 25 nodes (corresponding to the one-hot encoding of the 12 chords vocabulary and the 13 melody notes vocabulary), one hidden layer with 8 LSTM block (4 chord blocks and 4 melody blocks, as we will see below) containing 2 cells each and an output layer with 25 nodes (identical to the input layer). The separation between chords and melody is ensured as follows:

- 4 blocks are fully connected to the input nodes and to the output nodes corresponding to chords,
- 4 blocks are fully connected to the input nodes and to the output nodes corresponding to melody,
- chord blocks have recurrent connections to themselves *and* to the melody blocks,
- melody blocks have recurrent connections *only* to themselves.

In other words, as Eck states it [21]: "melody information does not reach the cell blocks responsible for processing chords." He argues that this decision makes sense because: "in real music improvisation the person playing melody (the soloist) is for the most part following the chord structure supplied by the rhythm section. However this architectural choice presumes that we know ahead of time how to segment chords from melodies. When

working with jazz sheet music, chord changes are almost always provided separately from melodies and so this does not pose a great problem."

Generation is performed as in the first experiment, by presenting a seed note or a sequence of notes (up to 24) and by feedforwarding the network to iteratively generate a sequence of notes and a sequence of chords. Eck reports satisfying results (Blues feeling) on both tasks.

Meanwhile, this experiment, although it became a reference, has obvious limitations. The training set is handcrafted, the representation of chords is limited and the generation is deterministic. Other systems based on an iterative feedforward strategy, such as CONCERT (see Section 7.3.1.1) and the system by Sturm et al. (see Section 7.3.1.2), use sampling from the probability distribution output in order to provide generation variability.

7.1.3 Stacked Autoencoders Architecture

7.1.3.1 Sun's DeepHear Ragtime Melody Generation System

Felix Sun in his *DeepHear* system [98] uses a *decoder feedforward* strategy on a *stacked autoencoders* architecture. The representation used is *piano roll* with *one-hot* encoding. The quantization (time step) is a sixteenth note. The corpus is 600 measures of Scott Joplin's ragtime music, split into 4 measures segments (thus 64 time steps). The number of input nodes is around 5000, which means having a vocabulary of about 80 possible note values. The architecture is shown at Figure 7.8 and is composed of 4 stacked autoencoders (with decreasing numbers of hidden units, down to 16 units).

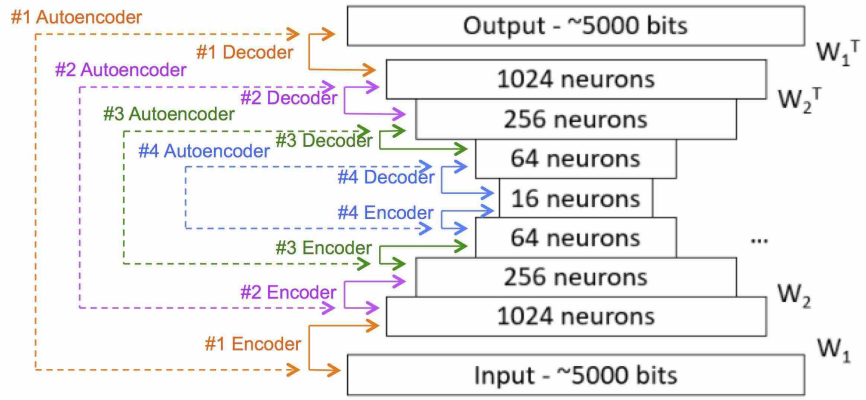

Fig. 7.8 DeepHear architecture.

Training the autoencoders is done *one layer at a time*, as for *pre-training* deep networks (see [29, page 528] for more information about pre-training). Then, as for pre-training neural networks, a *fine-tuning* final training is performed, with the same example provided as input and as output, see Figure 7.9.

Generation is performed by inputing random data as the seed into the 16 units middle (and thinnest) hidden layer and then feedforwarding the decoder to produce an output (in the same 4 measures format of the training examples), as shown at Figure 7.10.

Sun remarks that the system does some amount of plagiarizing. Some generated music is almost recopied from the corpus. Sun explains this because of the small size of the middle layer (only 16 units). He measured the similarity (defined as the percentage of notes in a generated piece that are also in one of the training pieces) and found that, on average, its

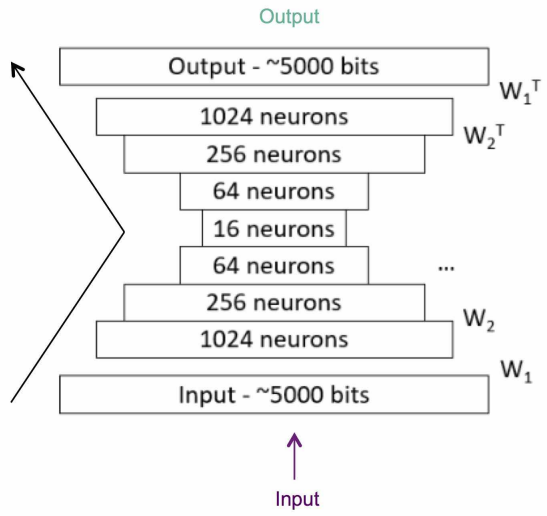

Fig. 7.9 Training DeepHear.

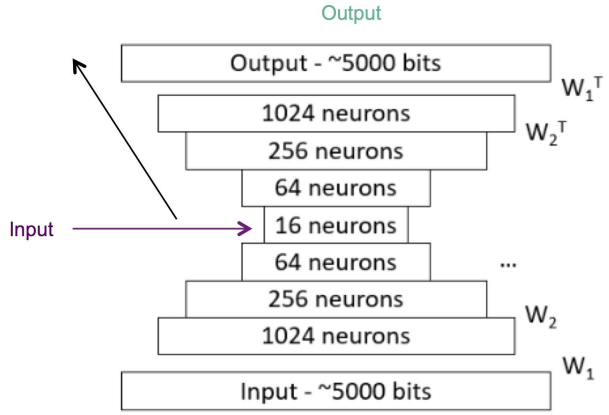

Fig. 7.10 Generation in DeepHear.

value is 59.6%, which it is indeed quite high, but does not prevent most of generated pieces to sound different.

7.1.3.2 Sarroff and Casey's Audio Generation System

The system named *deepAutoController*, by Andy Sarroff and Michael Casey, described in [93], is similar to DeepHear melody generation experiment (see Section 7.1.3.1) in that it also uses *stacked autoencoders*, but the representation is *audio* (more precisely a spectrogram generated by Fourier transformation). The system also provides some user interface to interactively control the generation by, e.g., selecting a given input (to be feedforwarded into the decoders), or generate a random input, controlling (by scaling or muting) the activation of a given unit. The dataset is composed of 8000 songs of 10 musical genres.

7.2 Sampling Strategy

7.2.1 RBM Architecture

7.2.1.1 Boulanger-Lewandowski *et al.*'s RBM Polyphonic System

In the work described in [7], Nicolas Boulanger-Lewandowski *et al.* at first propose to use a *restricted Boltzmann machine (RBM)* [42] to model polyphonic music. Their prior objective is actually to model polyphonic music (by learning a model from a corpus) in order to improve transcription of polyphonic music from audio. But they also discuss the generation of samples of the model learnt as a qualitative evaluation and finally for music generation [6]. They use a more general non-Boolean (also named Bernoulli) model of RBM, where variables have real values and not just Boolean values. In their first experiment the RBM will learn from the corpus the distribution of possible simultaneous notes, i.e., a repertoire of chords.

Once trained the RBM, they can sample from the RBM by *block Gibbs sampling*, by performing alternative steps of sampling hidden layer parameters from visible layer parameters (see Section 6.3 for more details about sampling strategies). Figure 7.11 shows various samples, each column representing a specific sample vector of notes, with the name of the chord below where the analysis is unambiguous.

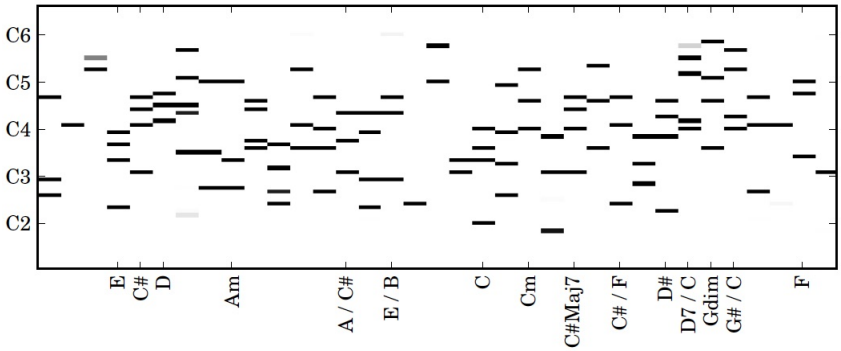

Fig. 7.11 Samples generated by the RBM trained on Bach chorales.

7.2.2 Feedforward & RNN Architectures

7.2.2.1 Hadjeres *et al.*'s DeepBach Chorale System

Gaëtan Hadjeres *et al.* DeepBach architecture[4] [36] is specialized for Bach chorales. The architecture, shown at Figure 7.12, is *compound*, combining two *recurrent* (LSTMs) and two *feedforward* networks. As opposed to standard use of recurrent networks, where a single time direction is considered, DeepBach architecture considers the two directions *forward* in time and *backwards* in time. Therefore, two recurrent networks (more precisely, LSTMs, with 200 nodes) are used, one summing up past information and another summing up information coming from the future, together with a non recurrent neural network for notes occurring at the same time. Their three outputs are merged and passed as the input of

[4] The MiniBach architecture, described in Section 7.1.1.1, is actually a deterministic single-step feedforward major simplification of the DeepBach architecture.

a final feedforward neural network with one hidden layer with 200 units and using ReLU. The final output activation function is *softmax*. The first 4 lines of the example data on top of the Figure 7.12 correspond to the 4 voices. The two bottom lines correspond to metadata (fermata and beat information). Actually this architecture is replicated 4 times, one for each voice (4 in a chorale).

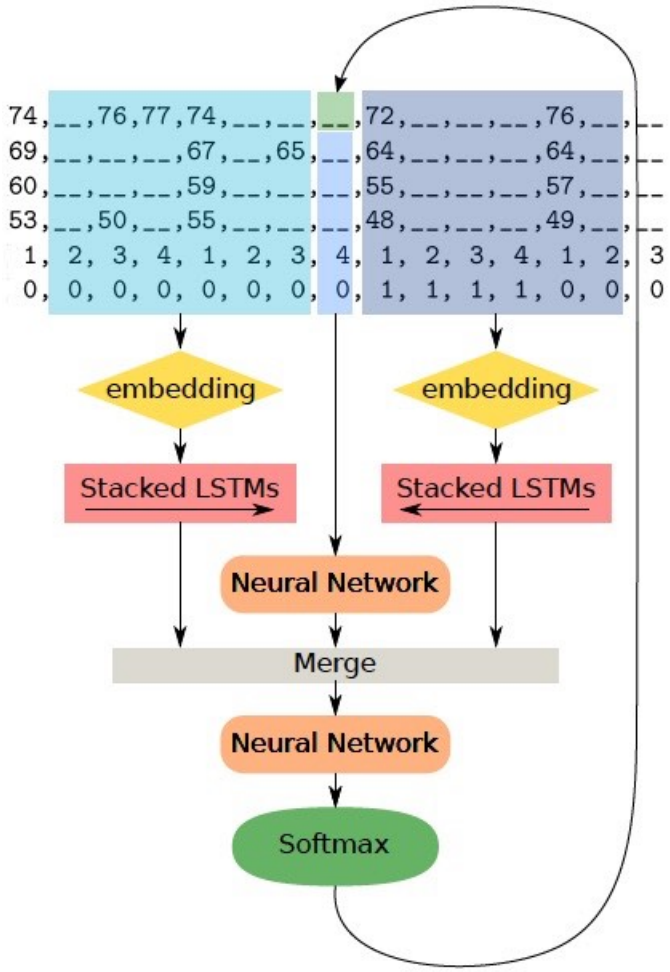

Fig. 7.12 DeepBach architecture.

This architectural choice somehow matches real compositional practice of Bach chorales. Indeed, when reharmonizing a given melody, it is often simpler to start from the cadence and write music *backwards*.

The initial corpus is the set of J.S. Bach polyphonic chorales music [4], where the composer chose various given melodies for a soprano and composed the three additional ones (for alto, tenor and bass) in a *counterpoint* manner. Contrary to other approaches, which transpose all chorales to the same key (usually in C major or A minor)[5], the dataset is augmented by adding all chorale transpositions which fit within the vocal ranges defined by the initial corpus. This leads to a total corpus of 2,503 chorales. The vocal ranges contains less than 30 different pitches for each voice[6].

[5] See Section 4.4.8.

[6] More precisely: 21 for the soprano, alto and tenor parts; and 28 for the bass part.

The choice of the representation in DeepBach has some specificities. Rhythm is modeled by simply adding a *hold symbol* (_) encoding whether or not the preceding note is held to the list of existing notes. This representation is unambiguous, compact and well-suited to the sampling method[7]. Another specifity of DeepBach is that the representation consists in encoding notes using their *real names* and not their MIDI pitches (e.g., *F#* is different from a *Gb*)[8]. Also the fermata symbol for Bach Chorales is explicitly considered as it helps producing structure and coherent phrases. More details can be found in [36].

Note that training is not done in the conventional way for neural networks. The objective is to predict the value of current note for a a given voice (shown in light green, on top center of Figure 7.12), using as information surrounding contextual notes (and their associated metadata), more precisely: the current notes for the 3 other voices (the thin rectangle in light blue, in the center), the 6 previous notes (the rectangle in light turquoise blue, on the left) for all voices, the 6 future notes (the rectangle in light magenta, on the right) for all voices. The training set is formed on-line by repeatedly randomly selecting a note in a voice from an example of the corpus and its surrounding context (as defined above).

Generation is done by sampling, using a *pseudo-Gibbs sampling* procedure, analog but computationally simpler than *Gibbs sampling* procedure[9], to produce a set of values (each note) of a polyphony, following the distribution that the network has learnt. Note that, for generation of notes, as opposed to most strategies (notably, iterative feedforward strategy), where instantiation of notes is done incrementally from left to right (along the time dimension), in DeepBach, instantiation of notes is done globally (for the whole temporal range) and incrementally.

An example of chorale generated is shown at Figure 7.13. As opposed to many experiments, a systematic evaluation has been conducted (with more than 1,200 human subjects, from experts to novices, via a questionnaire on the Web) and results are very positive, with a significant difficulty to discriminate between chorales composed by Bach and generated by DeepBach. Also a user interface has been developed to help the user controlling partial regeneration of chorales. This is made possible by the incremental nature of the generation.

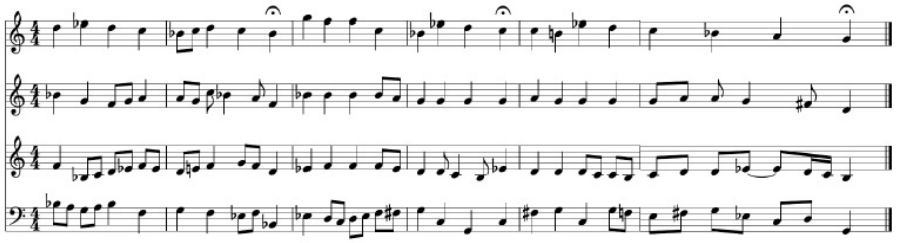

Fig. 7.13 Example of chorale generated by DeepBach.

[7] The authors emphasize the choice of data representation with respect to the sampling procedure, more precisely that the fact that they obtain good results using Gibbs sampling relies exclusively on their choice to integrate the hold symbol into the list of notes.

[8] This means, as opposed to most of systems, considering *enharmony*, see Section 4.4.6.

[9] The difference, based on the non assumption of compatibility of conditional distributions, as well as the algorithm, are detailed and discussed in [36].

7.3 Sampling & Feedforward Strategies

7.3.1 RNN Architecture

7.3.1.1 Mozer's CONCERT Music Generation System

CONCERT (an acronym for *con*nectionist *c*omposer of *er*udite *t*unes) by Michael Mozer is actually among of the first systems for generating music based on recurrent networks, presented in 1994 [77]. It is based on a *recurrent network* architecture. The input and output representation includes three aspects of a note: *pitch*, *duration* and *harmonic chord* accompaniment. The representation of a pitch is inspired by a psychological pitch representation space of Shepard based on two dimensions (circle of chromas and circle of fifths) [95], see Figure 7.14, resulting in a specific representation and encoding (see the details in [77]).

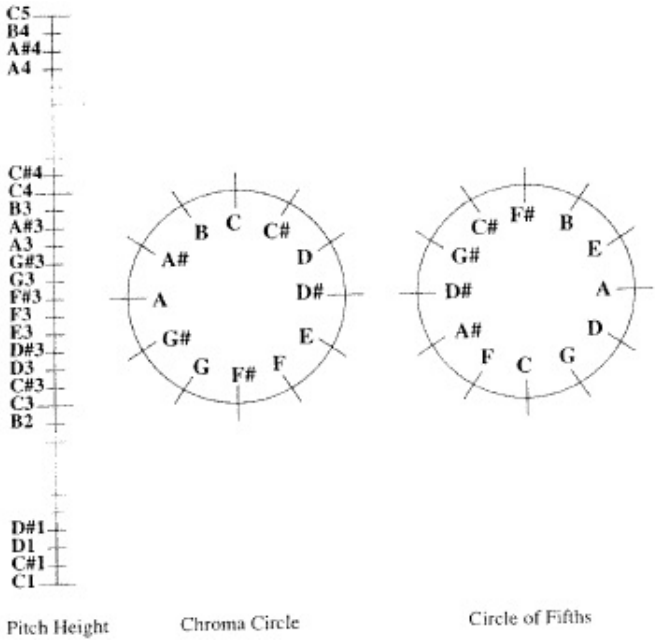

Fig. 7.14 Shepard's pitch representation.

The representation of durations and chords is also specific and rich. The representation of duration is based on 1/3 and 1/4 *beat cycles*. The representation of chords is based not only on the set of pitches (notes) composing the chord but also on the fours harmonics of each component pitch. To represent next note to be predicted, the CONCERT system actually uses both this rich and distributed representation (named next-node-distributed, see at Figure 7.15) and a more concise and traditional representation (named next-node-local), in order to be more intelligible. The activation function is the *sigmoid* function rescaled to the $[-1\ +1]$ range and the cost function is *quadratic cost*.

In the generation mode, the output is interpreted as a probability distribution over a set of possible notes as a basis for deciding next note in a non deterministic way, following the *sampling strategy* (see Section 6.3).

CONCERT has been tested on different examples, notably after training with melodies of J. S. Bach. Figure 7.16 shows an example of melody generated based on the Bach training set. Although now a bit dated, CONCERT has been a pioneering work and the discussion in the article about representation of music is still actual and quite interesting.

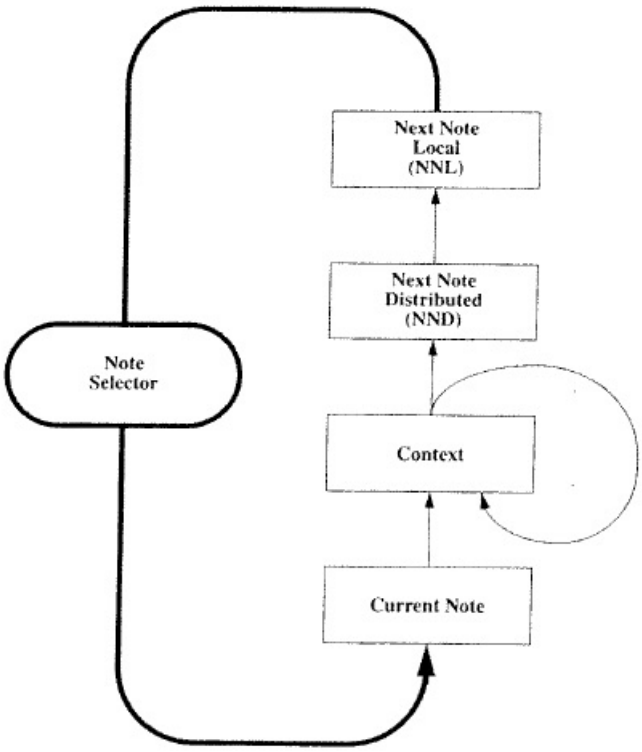

Fig. 7.15 CONCERT architecture.

Note also that CONCERT is representative of the early generation systems, before the advent of deep learning architectures, when representation was designed with rich hand-crafted features. One of the interest of using deep learning architectures is that such kind of rich and deep representation is automatically extracted and managed by the architecture.

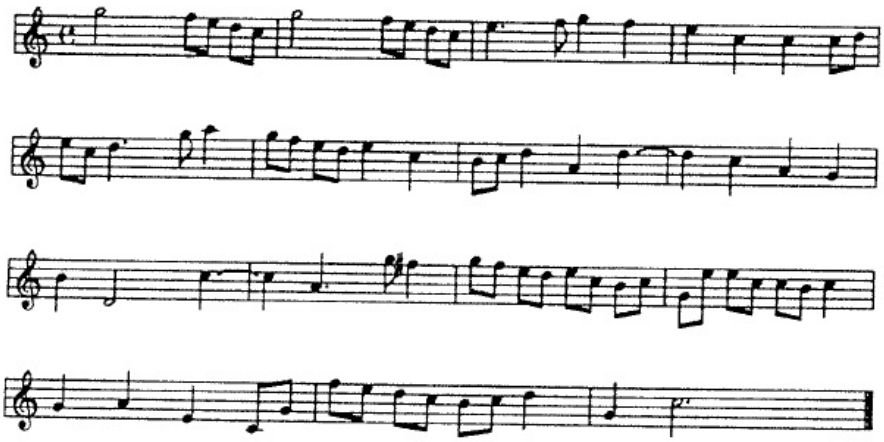

Fig. 7.16 Example of generation by CONCERT based on the Bach training set.

7.3.1.2 Sturm *et al.*'s Celtic Melody Generation System

Another representative example is the system by Bob Sturm *et al.* to generate Celtic music melodies [97]. The architecture used is a *recurrent network* with 3 hidden layers with 512 LSTM cells each. The foundation of the representation chosen is *textual*, namely the *ABC notation* [115] (see Section 4.3.3). This textual (ABC) notation is slightly transformed, in what the authors name a *folk-rnn* textual notation, in order to represent and discriminate the various types of tokens considered (time signature token, key token, and the 4 types of tokens corresponding to music events) and to avoid ambiguity[10]. The number of input and output nodes is equal to the number of characters in its vocabulary (i.e., with a *one-hot* encoding). The output of the network is a probability distribution over its vocabulary.

Training the recurrent network is done in an iterative way, as the network learns to predict the next item. Once trained, the generation is done by inputing a random item or a specific item (e.g., corresponding to a specific starting note), feedforward it to generate the output (which is a probability distribution output over the vocabulary), sample from this probability distribution, and use each selected vocabulary element as subsequent input, in order to iteratively produce a melody.

The final step is to decode the ABC representation generated into a MIDI format melody to be played. See at Figure 7.17 an example of a melody generated. One may see and listen to results on [96]. The results are very convincing, with melodies generated with a clear Celtic style.

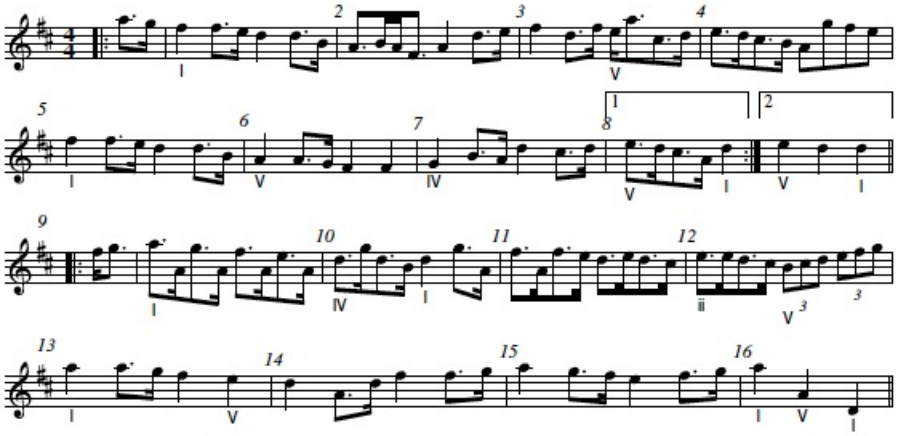

Fig. 7.17 Score of "The Mal's Copporim" automatically generated.

7.3.1.3 Liang's BachBot Chorale Generation System

Like Sturm *et al.*'s Celtic music system (see Section 7.3.1.2), the *BachBot* system by Feynman Liang [66] is based on a *recurrent network* architecture (and more precisely a LSTM architecture), but its objective is to generate *polyphonic* music (chorales, in the style of Bach's chorales). Liang used a grid search in order to select optimal setting for hyperparameters of the architecture (number of layers, RNN size...). The selected architecture has 3 layers and as Liang notes: "Depth matters! Increasing num_layers can yield up to 9% lower validation loss. The best model is 3 layers deep, any further and overfitting occurs." As has been already noted in Section 4.3.4, one of the specificity of BachBot is in encoding

[10] The authors mention as an example that otherwise C can mean a pitch, part of a pitch (Ĉ), a letter in a title (A Cup of Tea), or part of a key designation (K:Cmin).

multiple voices as a sequence[11] and with a special delimiter symbol to indicate the next time frame. All examples (of the training dataset) are transposed in the same key.

Also, although preliminary, some interesting study is about the analysis of the specialization of some of the nodes (neurons) of the network, through some correlation analysis with some specific motives and progressions (see more details in [66, chapter 5]).

7.3.2 RNN-RBM Architecture

7.3.2.1 Boulanger-Lewandowski *et al.*'s RNN-RBM Polyphonic System

The RBM-based architecture, presented in Section 7.2.1.1 has been extended by the same authors [7] with a recurrent network (RNN) to represent the temporal sequence of notes. The idea is to *couple* the RBM to a deterministic recurrent neural network (RNN), such as:

- the RNN models the *temporal sequence* to produce successive outputs (corresponding to successive time steps),
- that are *parameters* (biases) of a RBM that models the *conditional distribution* of the *accompaniment notes* (which notes should be played with which other notes).

In other words, the objective is to combine a *horizontal view* (temporal sequence) and a *vertical view* (combination of notes for a particular time step). See the resulting architecture, named *RNN-RBM*, at Figure 7.18. The bottom part represents the temporal sequence of RBM parameters $u^{(t)}$ computed by the RNN. The upper part represents the sequence of each RBM instance, with $v^{(t)}$ as its visible layer and $h^{(t)}$ as its hidden layer, with the visible layer coupled to $u^{(t)}$.

Note that the RNN outputs values for the biases (bias ${b_h}^{(t)}$ of the hidden layer and bias ${b_h}^{(t)}$ of the visible layer) of the RBM, but not its weights. This means that the biases are variables for each time step (time dependent) but the weights are shared for all time steps (time independent).

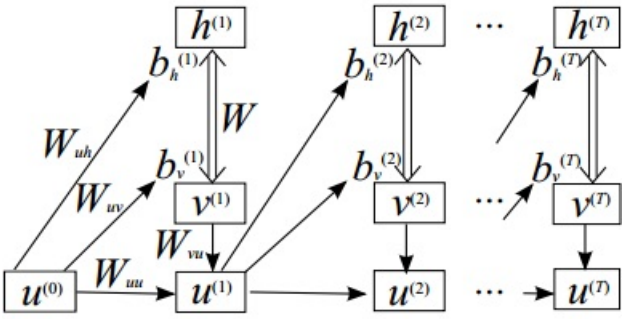

Fig. 7.18 RNN-RBM architecture.

During generation, each step (t time slice) of processing is as follows:

- compute the biases ($b_v^{(t+1)}$ and $b_h^{(t+1)}$) of RBM$^{(t+1)}$ with Equations 7.1 and 7.2;
- sample from the RBM by using *block Gibbs sampling* to produce $u^{(t+1)}$;
- feedforward the RNN with $v^{(t+1)}$ as the input in order to produce $v^{(t+1)}$ with Equation 7.3.

[11] An ordered sequence – in a descending pitch, i.e. soprano voice being the first one.

$$b_v^{(t+1)} = b_v + W_{uv}u^t \tag{7.1}$$

$$b_h^{(t+1)} = b_h + W_{uh}u^t \tag{7.2}$$

$$u^{(t+1)} = \tanh(b_u + W_{uu}u^{(t)} + W_{vu}v^{(t+1)}) \tag{7.3}$$

There is a specific training algorithm, not detailed here, please refer to [7].

Four different corpora have been used in the experiments: classical piano, folk tunes, orchestral classical music and Bach chorales. Polyphony varies from 0 to 15 (simultaneous notes, with an average value of 3.9). A *piano roll* representation is used with *one-hot* encoding of 88 units representing pitches from A_0 to C_8. Discretization (time step) is a quarter note. All examples have been transposed to a single common tonality (C major or minor). An example sample in a piano roll representation is shown at Figure 7.19. The quality of the model made RNN-RBM one of the reference architecture for polyphonic music generation.

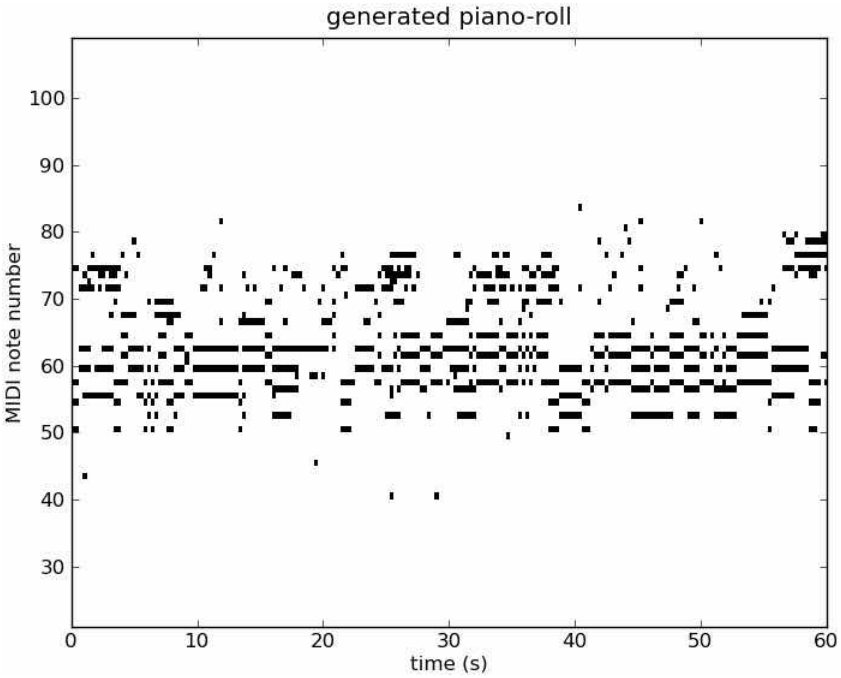

Fig. 7.19 Sample generated by the RNN-RBM trained on Bach chorales.

7.3.2.2 Other RNN-RBM Systems

There have been a few following systems, inspired by and extending the RNN-RBM architecture, but with not much significant difference and furthermore with no evaluation. Let us mention:

- the *RNN-DBN* architecture[12], using multiple hidden layers [28],
- using a LSTM instead of a RNN [67],
- using a GRU instead of a RNN [14].

[12] This is apparently the state of the art for Bach Chorales dataset in term of cross entropy loss.

7.3.3 Feedforward-RNN Architecture

7.3.3.1 Johnson's Hexahedria Polyphony Generation Architecture

The system proposed by Daniel Johnson in his Hexahedria blog [53] for polyphonic music is hybrid and original, in that it combines both non recurrent (feedforward) and recurrent layers within the same network architecture. The intuitive idea is to combine (more precisely, integrate) in the *same* architecture one *recurrent* neural network and one *non recurrent* neural network. The recurrent half architecture is in charge of the *temporal* aspect (relations between notes in a sequence) whereas the following non recurrent half architecture is in charge of the *harmonic* aspect (relations between notes within a same time step)[13]. The architecture has 4 hidden layers, the first two layers being recurrent (with 300 units each) and the last two layers being non recurrent (with 100 and 50 units, respectively), but, with transversal directed connexions between the units of each of these two last layers (this is an original feature). The resulting architecture is shown at Figure 7.20 in its folded form (folding the recurrent connexions) and at Figure 7.21 in its unfolded form.

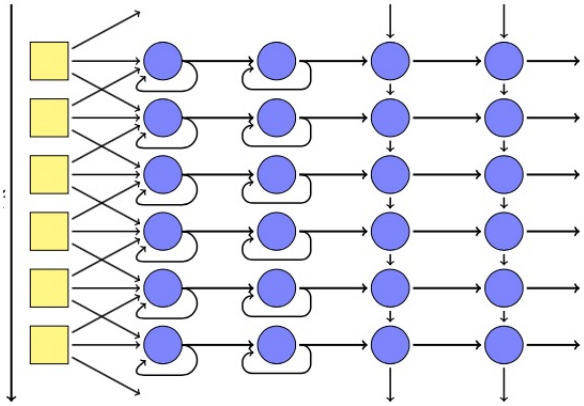

Fig. 7.20 Hexahedria architecture (Folded).

Three axes are represented:

- *flow axis*, represented horizontally and directed from left to right. It represents the flow of (feedforward) computation through the architecture, from the input layer to the output layer;
- *note axis*, represented vertically and directed from up to down. It represents the connexions between units of each non recurrent layer;
- *time axis*, represented diagonally and directed from up left to right down. It represents the time steps and the propagation of the memory within a same unit of each recurrent layer.

The input representation used is *piano roll*, with the pitch represented as the MIDI note value. More specific information is added: pitch classes, previous note played, how many times a pitch class has been played in the previous time step, beat (position within the measure, assuming 4/4 time signature) (all in one-hot encoding). The output representation is piano roll, in order to represent the possibility of more than one note at the same time.

Generation is done in an iterative way (iterative feedforward strategy), as for most of recurrent networks.

[13] The architecture is actually some kind of integration of the *RNN-RBM* architecture described in Section 7.18.

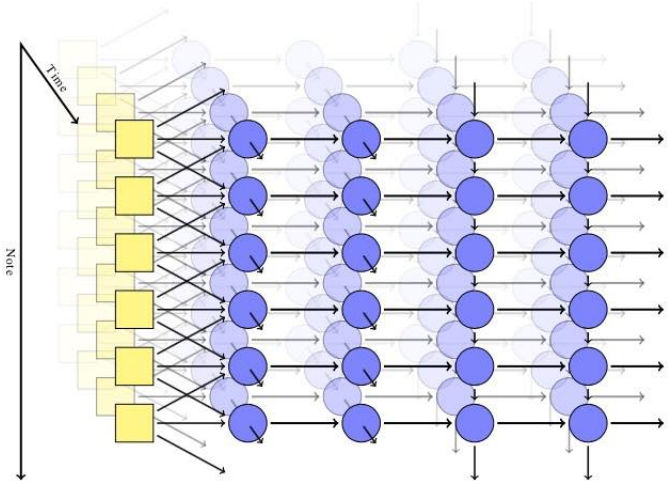

Fig. 7.21 Hexahedria architecture (Unfolded).

7.3.4 Variational RNN Encoder-Decoder Architecture

7.3.4.1 Fabius and van Amersfoort's VRAE Video Game Melody Generation System

In [24], Otto Fabius and Joost van Amersfoort propose to use a *variational autoencoder* (VAE) instead of a simple autoencoder. Moreover the encoder and the decoder parts are recurrent networks (with a single hidden layer, in practice LSTM are used). They name accordingly their proposed architecture a *Variational Recurrent Auto-Encoder (VRAE)*. This compound architecture follows the *RNN Encoder-Decoder* architecture proposed by [10][14] and introduced in Section 5.6.2.

The encoder is a RNN that reads each symbol of an input sequence sequentially. As it reads each symbol x_t, the hidden state (layer) of the Encoder RNN h_t^e changes. After reading the end of the sequence (marked by an end-of-sequence symbol), the hidden state of the Encoder RNN c is a summary of the whole input sequence. The decoder is another RNN which is trained to generate the output sequence by predicting the next symbol y_t given its hidden state h_t^d and the summary c. The two components of the RNN Encoder-Decoder are jointly trained to minimize the cross-entropy between input and output. The RNN Encoder-Decoder architecture is illustrated at Figure 5.22, with circles representing the successive hidden states of the encoder (h_t^e) and of the decoder (h_t^d).

The corpus used in the experiment of the VRAE architecture is a set of video game songs from the 80's and 90's. They are divided into various shorter parts of 50 time steps. A *one-hot* encoding of 49 possible pitches is used (pitches with too little amount of occurrences of notes were not considered). Experiments have been conducted with various number of hidden layer units (latent variables) equal to 2 and also equal to 20. Training takes place as for training recurrent networks, i.e., presenting for each input note next note as the output. Music generated is a kind of medley of the corpus used for training. The result is positive, but the low musical quality of the corpus hampers a careful evaluation.

[14] The initial motivation of the RNN Encoder-Decoder architecture is to be able to encode a variable-length sequence into another variable-length sequence. The application target is translation from a language to another language, resulting in sentences of possibly different lengths. The idea is to use a fixed-length vector representation as a pivot representation through some encoding and decoding (autoencoder) architecture.

7.3.4.2 Tikhonov and Yamshchikov's VRASH Melody Generation System

The system described by Alexey Tikhonov and Ivan Yamshchikov in [105], although similar to VRAE (see Section 7.3.4.1), uses a different representation (they separately encode in *one-hot* manner the pitch, the octave and the duration informations). The training dataset is composed of various songs (different epochs and genres), starting from MIDI files and with several filtering and normalization (see details in [105]). As for the architecture, they use a 4 layers LSTM[15] for the encoder and the decoder.

Last, they experimented with using the output of the decoder back into the decoder as a way to include the previous note generated as an added information (they call their final architecture *VRASH*, as for *Variational Recurrent Autoencoder Supported by History*). It is illustrated at Figure 7.22. In their evaluation, they state that the melodies generated are only slightly closer to the corpus (using a cross entropy measure) than when not adding history information, but that qualitatively the results are better.

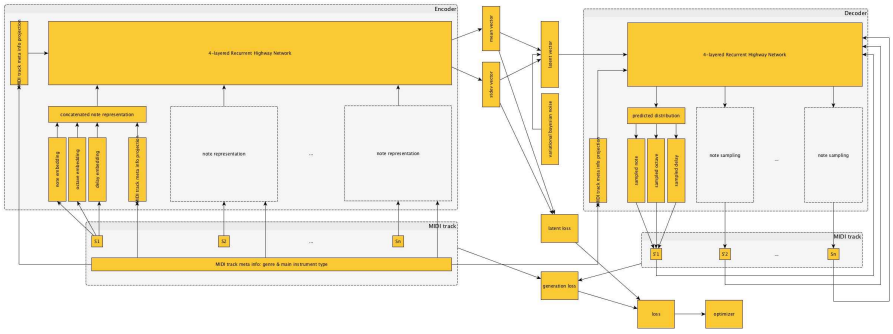

Fig. 7.22 VRASH architecture.

Last, note that the objective of chorales polyphony generation performed by the DeepBach system has been reformulated using variational autoencoders (encapsulating RNN architectures, as for VRAE), with a refined regularization of the hidden layer (latent variables), named *GLSR-VAE* (for *Geodesic Latent Space Regularization for the Variational AutoEncoder*), in [35]. The objective is to have a better control of the moves within the space of the latent variables.

7.4 Conditioning & Feedforward Strategies

7.4.1 Feedforward-RNN Architecture

7.4.1.1 Makris *et al.*'s Rhythm Generation System

The system proposed by Dimos Makris *et al.* [69] is specific in that it is dedicated to the generation of sequences of *rhythm*. Another specificity is the experiment they do in considering the possibility to *condition* the generation relative to some information, such as a given *beat* or a given *bass line*. The corpus is composed of 45 drums and bass patterns, with 16 measures each and in 4/4 time signature, collected from rock bands tablatures from the Web and converted to MIDI. The drums part representation includes 5 components (represented individually in an analog way to simultaneous musical voices): kick, snare,

[15] To be more precise, some recent evolution, named *Recurrent Highway Networks (RHN)* [120].

any tom event, open or closed hi-hats, crash or ride cymbals, each one having a binary value (with or without related event).

Encoding is made in text, similar in spirit to Sturm *et al.*'s Celtic system (see Section 7.3.1.2), more precisely following the approach proposed in [11]. Thus, for each time step, a binary word of length 5 represents the drums events occurring, e.g., `10010` represents simultaneous playing of kick and high-hat. The bass line part representation models two informations: if there is a note or a rest through a binary value and the evolution of the pitch through 3 possible binary values: steady (same), upward and downward, e.g., `1001` represents a descending bass line. Last, some additional information represents the metrical structure (the beat structure): start of the measure, half of a measure and third of a measure through 3 binary values, e.g., `111` is the value for the starting of each measure. The authors argue that this extra explicit information ensures that the network is aware of the beat structure at any given point.

The architecture is a combination of a *recurrent network* (more precisely, a LSTM) and a *feedforward network*. The LSTM (2 stacked LSTM layers with 128 or 512 units) is in charge of the drums part while the feedforward network is in charge of the bass line and the metrical structure (beat) information. These two networks are then merged, resulting in the architecture illustrated at Figure7.23. The feedforward network represents the *conditioning layer*. The authors report that this conditioning layer (bass line and beat information) improves the quality of the learning and of the generation. It may also be used in oder to mildly influence the generation. More details may be found in the article [69].

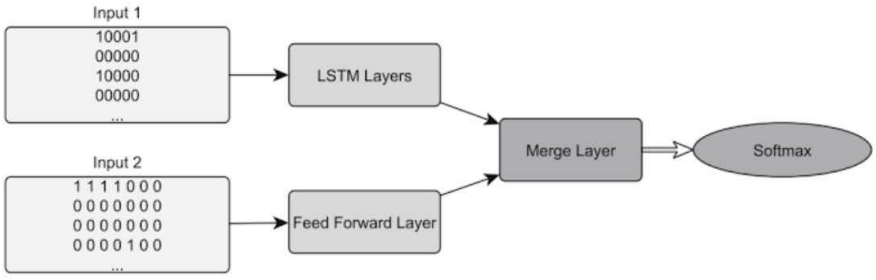

Fig. 7.23 Rhythm generation architecture.

7.5 Sampling & Conditioning & Feedforward Strategies

7.5.1 Convolutional(Feedforward) Architecture

7.5.1.1 Van der Oord *et al.*'s WaveNet Audio Generation System

WaveNet, by Aäron van der Oord *et al.* [108], is a system for generating raw *audio* waveforms. It has been experimented on generation for three audio domains: multi-speaker, text-to-speech (TTS) and music. Representation is a data compression from raw audio – typically stored as a sequence of 16-bit integer values, i.e., 65,536, per time step – onto 256 values, in order to be more tractable.

Two musical datasets are used: the MagnaTagATune dataset [62], which consists of about annotated 200 hours of music audio; and the YouTube piano dataset, which consists of about 60 hours of solo piano music obtained from YouTube videos.

The architecture is an extended *convolutional network*, called a *dilated causal convolution*, with the following 2 main additions:

- *causal* – The prediction at time step can only depend on previous time steps (and cannot depend on any of the future time steps)[16];
- *dilated* – This is used to reduce the amount of layers while keeping a large amount of input nodes.

Although the basis of the architecture is a *feedforward network* and not a recurrent network, the transversal causal connexions make it analog to a recurrent network[17]. The architecture is also made conditioning, by adding an additional input h. There are actually two options: *global conditioning* or *local conditioning*, depending if the conditioning input is shared for all time steps or is specific to each time step (h_t). The resulting architecture is illustrated at Figure 7.24. An example of application of conditioning WaveNet for a text-to-speech application domain is to feed linguistic features in order to generate speech with a better prosody.

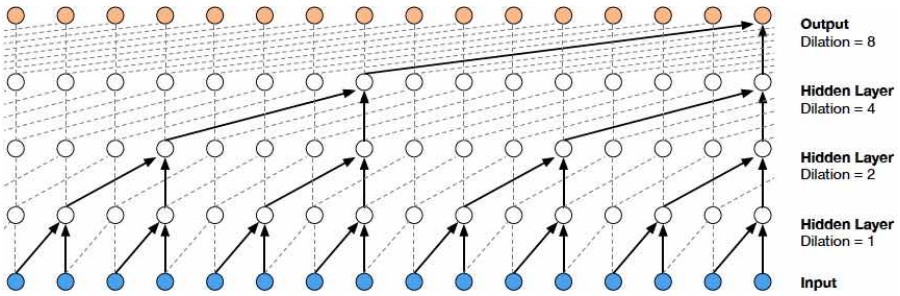

Fig. 7.24 WaveNet architecture.

For generation, at each time step a value is drawn from the probability distribution computed by the network. This value is then fed back into the input and a new prediction for the next step is made[18]. The first experiments in the music generation domain have not used the conditioning possibility, with interesting results.

The authors mention that enlarging the length of the time window (receptive field) processed was crucial to obtain samples that sounded musical.

They have also conducted preliminary work on conditioning music models to generate music given a set of tags specifying, e.g., genre or instruments, and state that their preliminary attempt is promising.

7.6 Input Manipulation & Feedforward Strategies

7.6.1 Stacked Autoencoders Architecture

7.6.1.1 Sun's DeepHear Ragtime Counterpoint Generation System

A second experiment has been conducted by Felix Sun (see his first experiment in Section 7.1.3.1) with a different objective: harmonize a melody, although using the same architecture and what had already been learnt[19]. The idea is to find an embedding – the values

[16] The convolution is in fact a *semi-convolution*, only towards previous time steps.

[17] On this issue, see Section 10.1.

[18] In an iterative feedforward strategy, as for a recurrent network.

[19] It is a simple example of transfer learning, with a same domain and a same training done, but onto a different production task.

of the 16 units of the bottleneck hidden layer of the stacked autoencoders – which will result in some generated output matching as much as possible a given melody. Therefore, a simple distance (error) function is defined to represent the distance (*similarity*) between two melodies (in practice, the number of non matched notes). Then just remains a gradient descent onto the embedding, guided by the gradients corresponding to the error function, until finding a sufficiently similar decoded melody.

Although this is not real harmonization (see Section 7.2.1.1 for an example of experiment specific for harmonization), but rather generation of a similar (consonant) melody, the results (tested on not Ragtime melodies) produce some naive counterpoint with a ragtime flavor.

7.7 Sampling & Input Manipulation Strategies

7.7.1 Convolutional(RBM) Architecture

7.7.1.1 Lattner *et al.*'s Convolutional RBM Constrained Sampling

The system presented by Stefan Lattner *et al.* in [61] combines *sampling* strategy and *input manipulation* strategy. The starting point is to use a restricted Boltzmann machine (RBM) to learn local structure, seen as the *musical texture*, of a corpus of musical pieces. The additional idea is in imposing through *constraints* some more *global structure* (form, e.g., AABA, as well as tonality), seen as as a *structural template* inspired from an existing musical piece, onto the generated new piece. These constraints, about structure, tonality and meter, will guide an iterative generation through a search process, manipulating the input, based on gradient descent.

The objective is the generation of *polyphonic* music. The representation used is *piano roll*, with 512 time steps and a range of 64 pitches (corresponding to MIDI pitch numbers `28` to `92`). The corpus is Mozart sonatas. Each piece is transposed into all possible keys in order to have sufficient training data for all possible keys. The architecture is a *Convolutional Restricted Boltzmann Machine* (C-RBM), i.e. a RBM with convolution (see Section 5.3.8), with 512 input nodes and 2,048 hidden units. Units are not Boolean, as in standard RBM, and are continuous, as for neural networks. Convolution is only performed on the time dimension, in order to model temporally invariant motives, but not pitch invariant motives (there are correlations between notes over the whole pitch range), which would break the notion of tonality[20]. The resulting *convolutional restricted Boltzmann machine with constraints* architecture is summarized at Figure 7.25.

Training is done for the C-RBM through contrastive divergence (more precisely, a more advanced version, named *persistent contrastive divergence*).

Generation is done by *sampling* but with some *constraints*. Three types of constraints are considered:

- *self-similarity* – The purpose is to specify a global structure (e.g. AABA) in the generated music piece. This is modeled by minimizing the distance (mean squared error) between the self-similarity matrixes of a target and of the intermediate solution;
- *tonality constraint* – The purpose is to specify a key (tonality). To estimate the key in a given temporal window, they compare the distribution of pitch classes in the window

[20] As the authors state [61], "Tonality is another very important higher order property in music. It describes perceived tonal relations between notes and chords. This information can be used to, for example, determine the key of a piece or a musical section. A key is characterized by the distribution of pitch classes in the musical texture within a (temporal) window of interest. Different window lengths may lead to different key estimates, constituting a hierarchical tonal structure (on a very local level, key estimation is strongly related to chord estimation)."

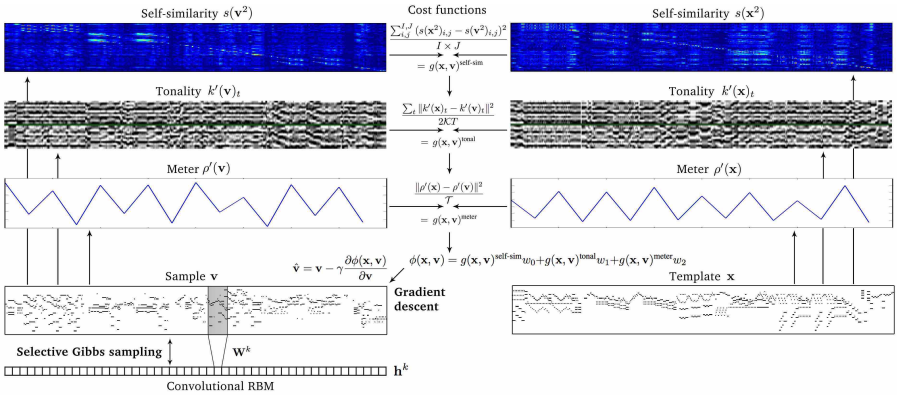

Fig. 7.25 C-RBM Architecture.

with so-called key profiles u^{mode} (i.e. paradigmatic relative pitch-class strengths for specific scales and modes) [102], in practice with the two major and minor modes. They are repeated in the time and in the pitch dimension of the piano roll matrix, with a modular octave shift in the pitch dimension. The resulting key estimation vectors are combined (see the article for more details) to obtain a combined key estimation vector. In the same way as for self-similarity, a distance between the target and the intermediate solution key estimation is minimized;

- *meter constraint* – The purpose is to impose a specific meter (also named a *time signature*, e.g., 3/4, 4/4...) and its related rhythmic pattern (e.g., relatively strong accents on the first and the third beat of a measure in a 4/4 meter). As note intensities are not encoded in the data, only note onsets are considered. The relative occurrence of note onsets within a measure is constrained to follow that of a target piece.

Generation is performed via *Constrained Sampling* (CS), an external mechanism to restrict the set of possible solutions in the sampling process according to some pre-defined constraints. A costly approach is *generate-and-test*, where valid solutions are picked from a set of random samples from the model. In more elaborated methods (e.g., in [84, 85], where hard constraints are imposed in music generation with Markov chains), the initial model is extended in order to sample constrained solutions in a more directed way. Depending on the nature of the constraints, this may be a very difficult problem, still not addressed in the literature.

A sample v is generated from the model in the following way. At first, an instance is randomly initialized, following the standard uniform distribution. A step of Constrained Sampling (CS) is composed of n runs of Gradient Descent (GD) optimization followed by p runs of *selective Gibbs sampling* (SGS)[21]. A *simulated annealing* algorithm is applied in order to decrease exploration in relation to a decrease of variance over solutions. Please see details in the original paper [61]. Figure 7.26 show an example of generated sample in piano roll format. The article [61] details in different figures the steps of constrained sampling.

The results are interesting and promising. One current limitation, stated by the authors, is that constraints only apply to high-level structure. Initial attempts at imposing low-level structure constraints were challenging because, as constraints are never purely content-invariant, when trying to transfer low-level structure, it can happen that the template piece gets exactly reconstructed in the GD phase. Therefore, creating constraints for low-level structure would have to be accompanied by increasing their content-invariance. Another issue is convergence and satisfaction of the constraints. As, discussed by the authors, their

[21] Selective Gibbs sampling (SGS) is an authors' variant of *Gibbs sampling* (GS), to keep corrections local [61].

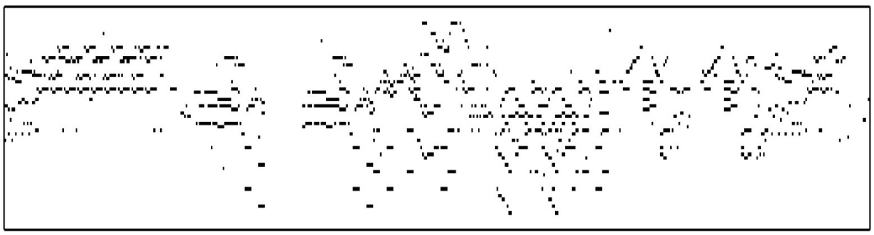

Fig. 7.26 Piano roll sample generated by C-RBM.

approach is not exact, as for instance for the Markov constraints approach (for Markov chains) proposed in [84].

7.8 Adversarial & Feedforward Strategies

7.8.1 GAN(RNN) Architecture

7.8.1.1 Mogren's C-RNN-GAN Classical Polyphony Generation System

Olof Mogren system, named *C-RNN-GAN* [75], has its objective the generation of *single voice polyphonic* music. The representation chosen is inspired from MIDI and models each musical event (note) via 4 attributes: duration, pitch, intensity and time spent since the previous event. This allows to represent simultaneous notes (in practice up to 3). The domain is classical music and retrieved in MIDI format from the Web and contains 3,697 pieces from 160 composers.

The architecture is a *generative adversarial networks* (GAN), with both the generator and the discriminator being *recurrent networks*, more precisely LSTM architectures, with 2 hidden layers and 350 units each. A specificity is that the discriminator has a bidirectional recurrent architecture, in order to take contexts from both past and future for its decisions. The architecture is shown at Figure 7.27.

Two examples of generated music are shown at Figure 7.28. The author conducted a number of measurements (4 metrics) on the generated music. The author states that the model trained with feature matching[22] gets a better trade-off between structure and surprise than the other variants. Note that this is consonant with the use of this regularization technique to control creativity trade-off in MidiNet (see Section 7.9.1.1).

7.9 Adversarial & Conditioning & Feedforward Strategies

7.9.1 GAN(Conditional(Convolutional(Feedforward))) Architecture

7.9.1.1 Yang *et al.*'s MidiNet Pop Melody Generation System

Li-Chia Yang *et al.* propose the MidiNet architecture as a both an *adversarial* and *convolutional* architecture to generate pop music melodies [119].

The dataset is a set of 1,022 pop music melodies of pop music, retrieved from the library of structured tablatures TheoryTab [45]. The representation used is derived from MIDI

[22] A regularization technique for improving GANs, see Section 5.3.10.

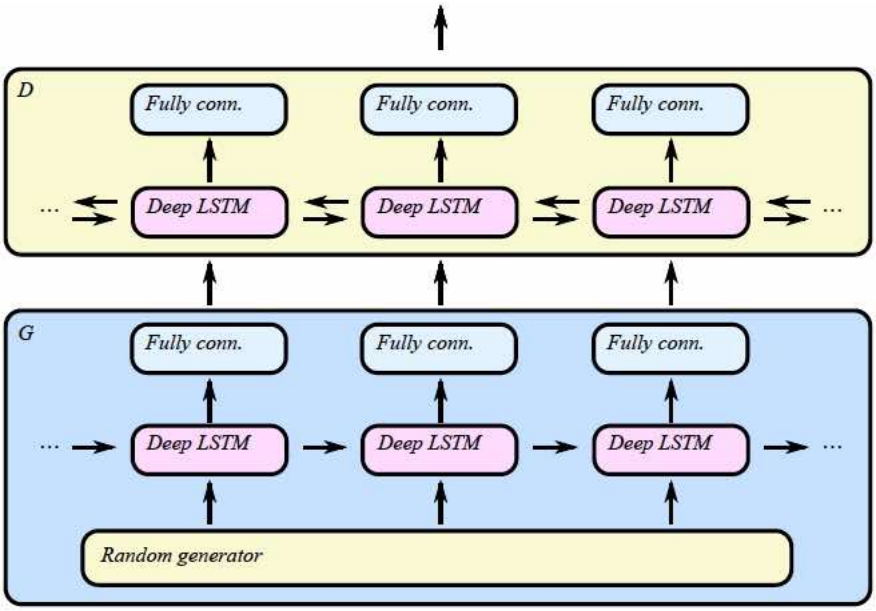

Fig. 7.27 C-RNN-GAN architecture.

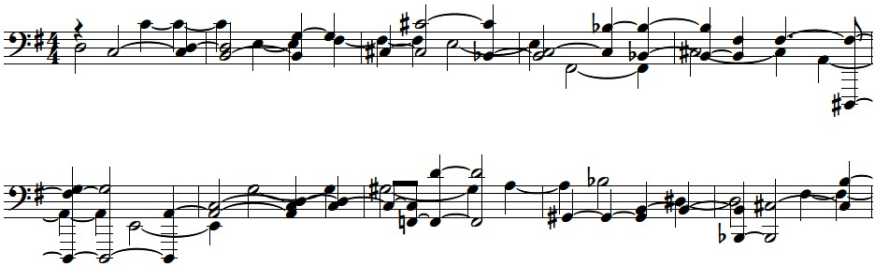

Fig. 7.28 C-RNN-GAN generated examples.

and uses a *piano roll* like format. The quantization is set at a 16th note. All melodies are normalized within a same 2 octaves range and they are transposed into all keys, leading to a final dataset of 50,496 bars of melodies.

There is an extended version of the architecture that uses chords as an additional input to *condition* the generation. A *one-hot* representation is used for the chords to specify its key and an additional binary parameter indicates if it is major or minor. An important specificity of the system is that the granularity for generation is the measure (and not the smallest note). The *conditioning* mechanism incorporates information from previous measures (as a memory mechanism, analog to a recurrent network).

The MidiNet architecture is illustrated at Figure 7.29. It is composed of a generator and a discriminator network, both convolutional networks. The generator includes 2 fully-connected layers (with 1,024 and 512 units respectively) followed by 4 convolutional layers. The conditioner includes 4 convolutional layers but with a reverse architecture. The discriminator includes 2 convolutional layers followed by some fully connected layers and the final output activation function used being *cross entropy*.

As for us GANs (see Section 5.3.10), the generator uses random noises as inputs, thus it can generate melodies from scratch. Meanwhile, the conditioner provides the capacity to exploit arbitrary prior knowledge available, represented as a matrix. The authors show how their system can generate music by following a chord progression, or by following a few starting notes (a priming melody). The conditioner also allows to incorporate information

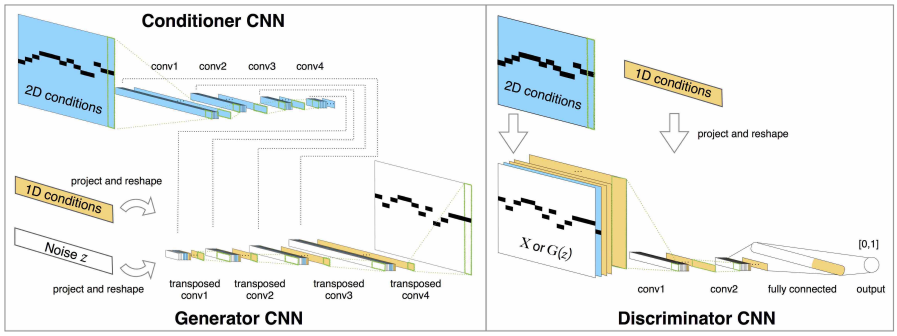

Fig. 7.29 MidiNet architecture.

from previous measures to intermediate layers and therefore consider history as would do a recurrent network[23]. In addition, the authors discuss two methods to control creativity:

- by restricting the conditioning by inserting the conditioning data only in the intermediate convolution layers of G;
- by decreasing the values of the two control parameters of feature matching regularization, in order to less enforce the distributions of real and generated data to be close.

Figure 7.30 shows some melodies generated in three successive experiments: 1) without chord conditioning; 2) with chord conditioning; 3) with chord conditioning in creative mode. The authors also conduct some external evaluation and comparative analysis.

(a) MidiNet model 1

(b) MidiNet model 2

(c) MidiNet model 3

Fig. 7.30 Examples generated by MidiNet.

7.10 Reinforcement & Feedforward Strategies

7.10.1 RNN Architectures

7.10.1.1 Jaques *et al.*'s RL-Tuner Melody Generation System

The *reinforcement strategy* has been pioneered by the *RL-Tuner* architecture [52] by Natasha Jaques *et al.*. The objective is to control the generation of melodies with user constraints.

The architecture, illustrated at Figure 7.31, consists in two deep Q network architectures and two *recurrent network* (RNN) architectures.

[23] On this issue, see the discussion recurrent vs convolutional in Section 10.1.

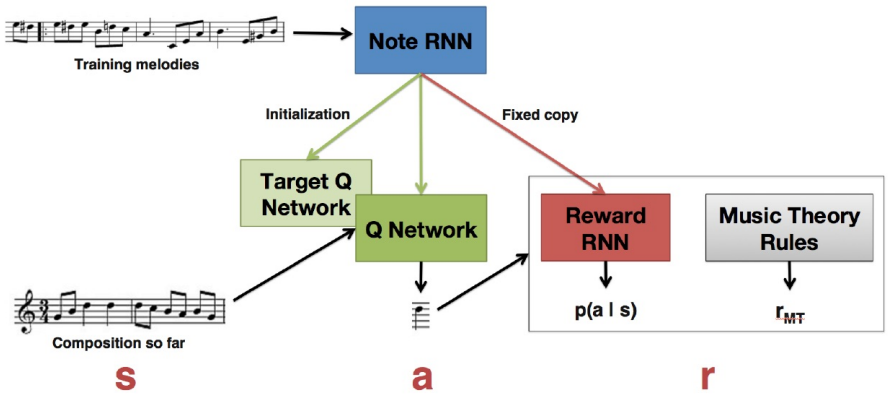

Fig. 7.31 RL-Tuner architecture.

- The initial RNN (named *Note RNN*, as its objective is to predict and generate next note) is trained on the dataset of melodies, in a similar way to the experiments by Eck and Schmidhuber on using a RNN to generate melodies following the *iterative feedforward* strategy (see Section 7.1.2.1).
- Then, a fixed copy is made, named the *Reward RNN*, and will be used for the reinforcement learning (to generate the dataset-based reward).
- The *Q Network* architecture task is to learn to select next note (next action a) from the generated (partial) melody so far (current state a).
- The Q Network is trained in parallel to the other Q Network, named the *Target Q Network*, which estimates the value of the gain and which has been initialized from what the Note RNN has learnt.
- The Q Network's reward r combines two measures:
  - adherence to *what has been learnt*, by measuring the similarity with the *note predicted by the Reward RNN recurrent network*;
  - adherence to *user-defined constraints* (in practice according to some musical theory rules, e.g., consistency with current tonality, avoidance of excessive repetitions...), by measuring how well they are fulfilled.

Therefore, this reinforcement strategy allows to combine arbitrary user given control with a style learnt by the recurrent network. Note that in the general model of reinforcement learning, the reward is not predefined (the agent does not know beforehand the model of the environment and the rewards, thus it needs to balance between exploring to learn more and exploit in order to improve its gain – the *exploration exploitation dilemna*). In the case of RL-Tuner, the reward is pre-defined and dual: handcrafted for the music theory rules and learnt from the dataset for the musical style. But we can see the opportunity to insert any kind of control, including incremental control by the user although it may be that a feedback at the granularity of each note generated may be too demanding and also not accurate[24]. We will discuss in Section 10.7 the issue of learning from user feedback.

[24] As Miles Davis coined it: "If you hit a wrong note, it's the next note that you play that determines if it's good or bad."

7.11 Unit Selection & Feedforward Strategies

7.11.1 Autoencoder & RNN Architectures

7.11.1.1 Bretan *et al.*'s Unit Selection and Concatenation Melody Generation System

This strategy has been pioneered by Mason Bretan *et al.* [8]. The idea is to generate music from a concatenation of music units, queried from a database. The key process here is *unit selection*, which is based both on two criteria: *semantic relevance* and *concatenation cost*. The idea of unit selection to generate sequences is actually inspired by a technique commonly used in Text-to-speech (TTS) systems.

The objective is to generate melodies. The corpus considered is a dataset of 4,235 lead sheets in various musical styles (jazz, folk, rock, classical...), and 120 jazz solo transcriptions. The granularity of a musical unit is a measure. (This means roughly 170,000 units in the dataset). The dataset is augmented by transpositions as well as interval alterations, while restricting it to a five octave range (MIDI notes `36` to `99`) and each unit is transposed so that all pitches are covered.

The architecture includes one *autoencoder* and two LSTM *recurrent networks*. The first step is feature extraction. 10 types of features, manually handcrafted, are considered, following a *bag-of-words* (BOW) approach (see Section 4.4.4), e.g., counts of a certain pitch class, counts of certain pitch class rhythm tuple, if first note is tied to previous measure... (see details in [8]). This results in 9,675 actual features (parameters). Note that all of them have integer values (`0` or `1` for the Boolean features and rests are represented using a negative pitch value). Therefore each unit is described (indexed) as a 9,675 size feature vector.

The autoencoder used is a double *stacked autoencoders* architecture, as described at Figure 7.32. Once trained, the usual self-taught way for autoencoders, on the set of feature vectors, it will become a new feature extractor encoding a 9,675 size feature vector into a 500 size vector *embedding*.

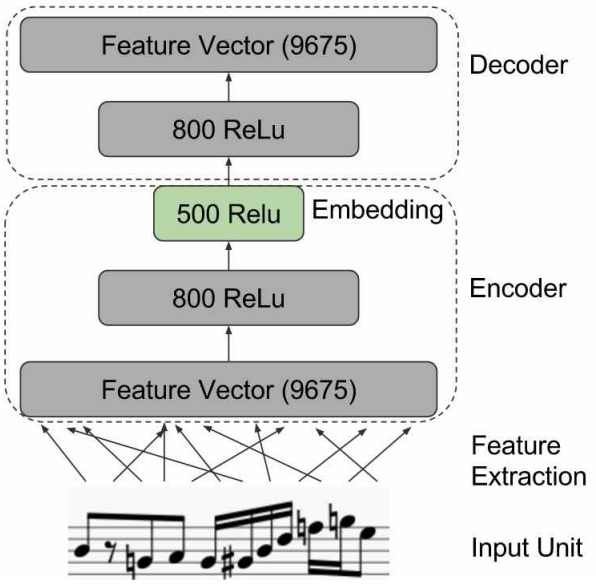

Fig. 7.32 Stacked autoencoders architecture.

The remaining issue to be able to generate a melody is the following: from a given musical unit, how to select a best (or at least, very good) candidate as a successor musical unit? Two criteria are considered:

- *successor semantic relevance* – Based on a model of transition between units, as learnt by a LSTM recurrent network. In other words, that relevance is based on the distance to the (ideal) next unit as predicted by the model. This LSTM architecture has 2 hidden layers, each with 128 units. Input and output layers have 512 units (corresponding to the format of the embedding);
- *concatenation cost* – Based on another model of transition[25], this time between the last note of the unit and the first note of the next unit, as learnt by another LSTM recurrent network. That second LSTM architecture has hidden layers and its input and output layers have about 3,000 units, as this corresponds to the one-hot encoding of the characterization of an individual note (as defined by its pitch and its duration).

The combination of the two criteria (illustrated at Figure 7.33) is handled by the following heuristic-based dynamic ranking process:

1. rank all units with the input seed through successor semantic relevance;
2. take the 5% top and re-rank them based on the concatenation cost;
3. rerank the same top 5% based on their combined (successor and concatenation) ranks;
4. select the unit with the highest combined rank.

The process is iterated in order to generate an arbitrary length melody, This iterative generation of a melody may look like the *iterative feedforward strategy* for a LSTM, but with two important differences: 1) the embedding of the next musical unit item is computed through a multi-criteria ranking algorithm, 2) the actual unit is queried from a database with the embedding as the index.

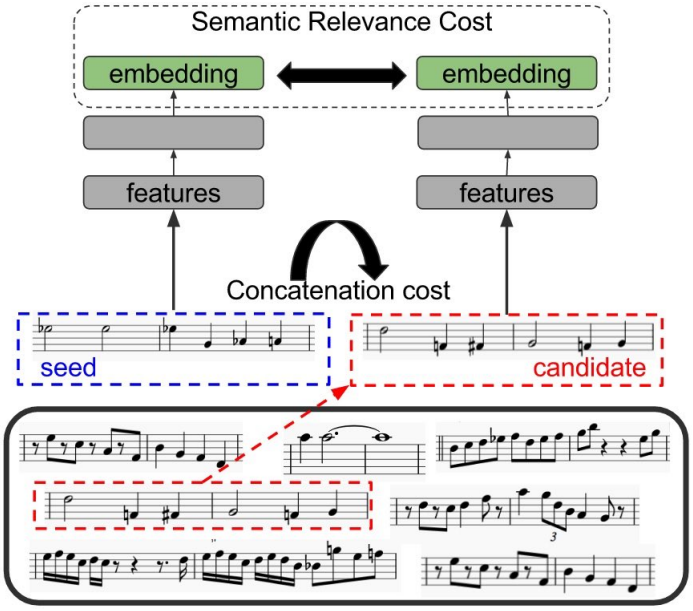

Fig. 7.33 Unit selection based on semantic cost.

Some initial external human evaluation has been conducted by the authors. They found out that music generated using units of one or two measure durations tended to be ranked

[25] At a more fine-grained level, note-to-note level, than the previous one.

higher according to naturalness and likeability than units of four measures or note-level generation, with an ideal unit length appearing to be one measure.

Chapter 8
Analysis

We now conduct a preliminary analysis and summary of the various systems surveyed, through our referential via one table. This is an opportunity to analyze relations between dimensions. Some are trivial (e.g., adversarial strategy is related to GAN architecture), but the analysis table may help to explore existing combinations, frequent or unfrequent correlations and also to point out combinations that have not yet been explored.

8.1 Referencing the Systems Analyzed

We first reference and summarize the various systems analyzed in Table 8.1.

8.1.1 Abbreviations

Because of width space limitation, we also introduce abbreviations to reference the various architectures and strategies in Tables 8.2 and 8.3, respectively.

8.1.2 Analysis Table

Table 8.4 is a global summary of the systems surveyed in terms of the two dimensions architecture and strategy. (Because of width space limitation, we cannot include also the objective and representation dimensions).

X^n means n instances of that architecture.

Reference name	*Original name*	*Authors*	*Refs*	*Section*	*Summary*
BachBot	BachBot	F. Liang	[66]	7.3.1.3	Bach chorale – LSTM.
Blues_M		D. Eck & J. Schmidhuber	[21]	7.1.2.1	Blues melody – LSTM.
$\text{Blues}_{M\&C}$		D. Eck & J. Schmidhuber	[21]	7.1.2.2	Blues melody and chords – LSTM.
Celtic		B. Sturm *et al.*	[97]	7.3.1.2	Celtic melody – text encoding – LSTM.
CONCERT	CONCERT	M. Mozer	[77]	7.3.1.1	Melody – RNN.
C-RBM	C-RBM	S. Lattner *et al.*	[61]	7.7.1.1	Polyphony – Convolutional RBM + input manipulation
C-RNN-GAN	C-RNN-GAN	O. Mogren	[75]	7.8.1.1	Polyphony – GAN(RNN).
deepAC	deepAuto-Controller	A. Sarroff & M. Casey	[93]	7.1.3.2	Audio – Stacked autoencoders + interface
DeepBach	DeepBach	G. Hadjeres *et al.*	[36]	7.2.2.1	Bach Chorale – Feedforward & LSTM + sampling.
DeepHear_M	DeepHear	F. Sun	[98]	7.1.3.1	Ragtime melody – Stacked autoencoders.
DeepHear_C	DeepHear	F. Sun	[98]	7.6.1.1	Ragtime counterpoint – Stacked autoencoders + input manipulation.
Hexahedria		D. Johnson	[53]	7.3.3.1	Polyphony – Feedforward & recurrent.
MidiNet	MidiNet	L.-C. Yang *et al.*	[119]	7.9.1.1	Pop melody – GAN(Convolutional(Feed-forward)) + conditional.
MiniBach	MiniBach	G. Hadjeres & J.-P. Briot		7.1.1.1	Bach Chorale – Feedforward.
Rhythm		D. Makris *et al.*	[69]	7.4.1.1	Rhythm – Feedforward & Conditional(RNN).
RL-Tuner	RL-Tuner	N. Jaques *et al.*	[52]	7.10.1.1	Melody – RNN & reinforcement.
RNN-RBM	RNN-RBM	N. Boulanger-Lewandowski	[7]	7.3.2.1	Polyphony – RNN, RBM + sampling
UnitSelection		M. Bretan *et al.*	[8]	7.11.1.1	Melody – Autoencoder, RNN + Unit selection.
VRAE	VRAE	O. Fabius & J. van Amersfoort	[24]	7.3.4.1	Video game melody – Variational Autoencoder (RNN).
VRASH	VRASH	A. Tikhonov & I. Yamshchikov	[105]	7.3.4.2	Various genre melody – Variational Autoencoder(RNN) + history.
WaveNet	WaveNet	A. van der Oord *et al.*	[108]	7.5.1.1	Audio – Conditional Convolutional(Feedforward)) + conditional.

Table 8.1 Table of references to the systems analyzed.

Fd	Feedforward
RNN	Recurrent
AE	Autoencoder
StAE	Stacked Autoencoders
RBM	Restricted Boltzmann Machine
VAE	Variational Autoencoder
CnV	Convolutional
CnD	Conditioning
GAN	Generative Adversarial Networks
RL	Reinforcement Learning
Cp	Compound

文字

Table 8.2 Abbreviations of the Architectures.

sFd	Single-step Feedforward
iFd	Iterated Feedforward
CnD	Conditioning
S	Sampling
iM	Input Manipulation
A	Adversarial
R	Reinforcement
uS	Unit selection
Cp	Compound

Table 8.3 Abbreviations of the Strategies.

Dimensions →	*Architecture*											*Strategy*								
↓ *Systems*	Fd	RNN	AE	StAE	RBM	VAE	CnV	CnD	GAN	RL	Cp	sFd	iFd	CnD	S	iM	A	R	uS	Cp
BachBot		X											X		X					X
Blues$_M$		X											X							
Blues$_{M\&C}$		X											X							
Celtic		X											X		X					X
CONCERT		X											X		X					X
C-RBM					X		X								X	X				X
C-RNN-GAN		X^2							X		X		X				X			X
deepAC				X								X								
DeepBach	X^2	X^2									X				X					
DeepHear$_M$				X								X								
DeepHear$_C$				X								X				X				X
Hexahedria	X	X									X		X		X					X
MidiNet	X						X	X	X		X	X		X			X			X
MiniBach	X											X								
Rhythm	X	X						X			X	X	X	X						X
RNN-RBM		X			X						X		X		X					X
RL-Tuner		X^2								X^2	X		X				X			X
UnitSelection		X^2		X								X	X						X	X
VRAE		X^2				X					X	X	X		X					X
VRASH		X^2				X					X	X	X		X					X
WaveNet	X						X	X			X		X	X	X					X

Table 8.4 Table of systems according to the Architecture and Strategy dimensions.

Chapter 9
Other Sources of Inspiration

We now analyze some deep learning based generation systems applied to other domains (mainly images), that could be sources of inspiration for music generation.

9.1 Input Manipulation

9.1.1 Mordvintsev et al.*'s DeepDream Psychedelic Images Generation System*

DeepDream, by Alexander Mordvintsev *et al.* [76], has become famous for generating psychedelic versions of standard images. The idea is to use a deep feedforward neural network architecture (see Figure 9.1), and use it to guide the alteration of an initial input image in order to maximize the potential occurrence of a specific visual feature. The method is as follows:

- the network is first trained on a large images dataset;
- in place of minimizing the cost function, the objective is to *maximize* the activation of a specific node(s) (which has been identified to activate for a specific pattern, e.g., a dog face, this node being represented in green at Figure 9.1);
- an initial image (e.g., of a tree) is iteratively slightly altered (e.g., by jitter[1], under gradient ascent control) to maximize that specific activation. This will favor emergence of occurrences of that specific pattern in the image.

Note that the activation maximization of a specific *higher layer node* (the case here, see Figure 9.1) will favor the emergence in the initial image of that specific high-level pattern (e.g., a dog face at Figure 9.2), whereas the activation maximization of a specific *lower layer node* (Figure 9.3) will result in texture insertion (Figure 9.4).

As one could see, traditional approach for training via gradient estimation through back-propagation and gradient following has been displaced, from updating connexion *weights* in order to *minimize* cost, to updating *input* in order to *maximize* activation of a specific node.

One may imagine trying to transpose the DeepDream objective to music, by maximizing the activation of a specific node (specially if the role of a node has been identified, as, e.g., in the preliminary study presented in Section 7.3.1.3).

Some related (but slightly different) strategy has been applied for the second experiment of DeepHear, described in Section 7.6.1.1, to manipulate the values of the hidden layer of

[1] Adding some small random noise displacement of pixels.

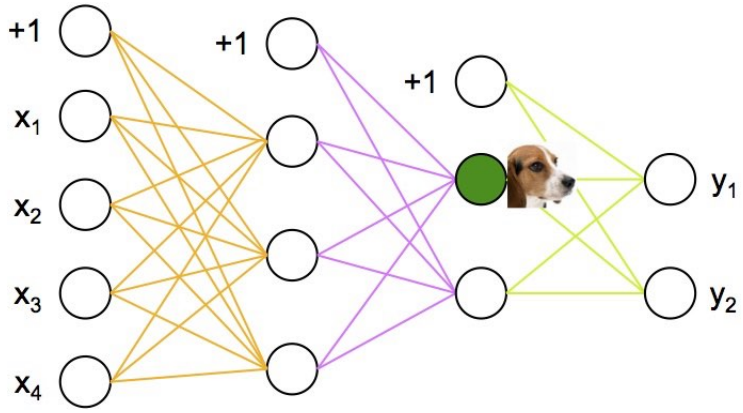

Fig. 9.1 DeepDream architecture.

Fig. 9.2 Example of higher layer unit maximization resulting image.

a stacked autoencoders architecture, in order to produce (through the decoder) a melody similar to another target melody.

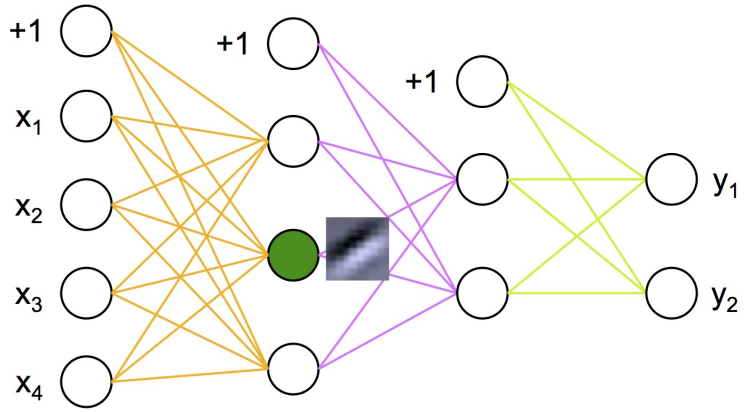

Fig. 9.3 Deep Dream architecture focused on a lower level unit.

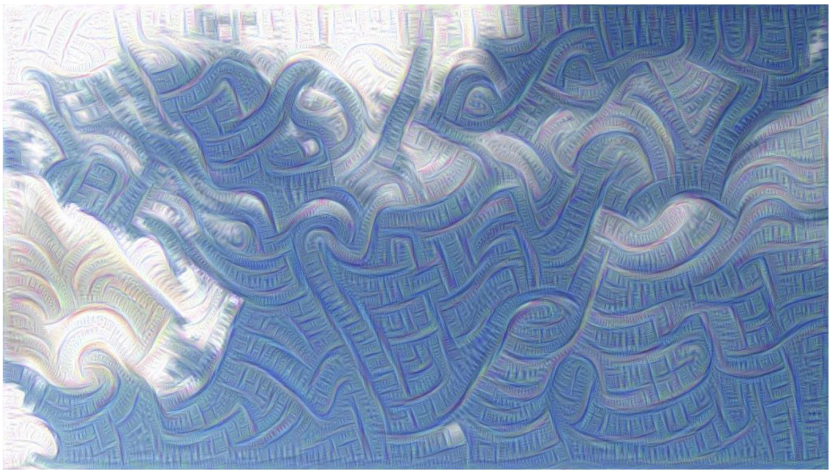

Fig. 9.4 Example of lower layer unit maximization resulting image.

9.2 Style Transfer

9.2.1 Gatys et al.*'s Style Transfer Painting Generation System*

The idea in this approach, pioneered by Leon Gatys *et al.* [27] and designed for images, is to use a neural network to independently capture:

- features (named the *content*) of an image
- and the *style* (as a correlation between features) of another image
- and then, gradient-based learning is used to guide the incremental modification of an initially random third image, with the double objective of matching *both* the content and the style description.

More precisely, the method is as follows:

- capture *content* information of the first image (the content reference), by feed-forwarding it into the network and by storing *units activations* for each layer;
- capture *syle* information of the second image (the style reference), by feed-forwarding it into the network and by storing *feature spaces*, which are *correlations* between units activations for each layer;
- synthesize an hybrid image in the following way:

- at first, generate a random image, then define it as current image
- and then *iterate*, until reaching the two targets (both content similarity and style similarity):
  - capture the *contents* and the *style* information of current image,
  - compute the *content cost* (distance between reference and current content) and the *style cost* (distance between reference and current style),
  - compute the corresponding gradients, through standard back-propagation,
  - update current image guided by the gradients;

More details may be found in [27]. At Figure 9.5 the architecture and process are summarized. The content image (on the right) is a photography of Neckarfront in Tübingen[2] in Germany and the style image is the painting "The Starry Night" by Vincent van Gogh.

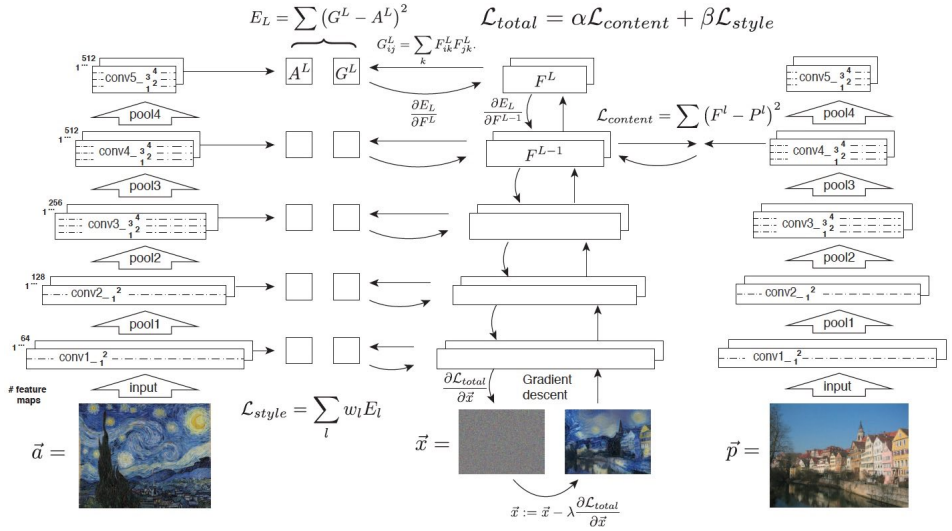

Fig. 9.5 Style transfer full architecture/process.

Note that one may balance between content and style objectives[3] in order to favor one or the other. In addition, the complexity of the capture may also be adjusted via the number of hidden layers used. These variations are shown at Figure 9.6 with, rightwards an increasing α/β content/style objectives ratio and downwards an increasing number of hidden layers used. The style image is the painting "Composition VII" by Wassily Kandinsky.

Transposing this style transfer technique to music is a tempting direction. Meanwhile, the style of a piece of music is less easy to capture globally via correlation of activations. For audio it looks easier and actually it has already been experimented independently by at least two researchers: [107] and [25]. The results are interesting. Transposing this to symbolic representation is more challenging, as a more frequent representation is through temporals series. But using pre-encoding of temporal series, e.g., such as the RNN Encoder-Decoder (see in Section 7.3.4.1) may be a path to explore.

[2] The location of the researchers.

[3] Through the α and β parameters, see on the top of Figure 9.5 the total loss defined as $\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$.

Fig. 9.6 Style transfer variations.

9.3 Originality-Driven Generation

9.3.1 Akin Kazakçi et al.*'s Non Digits Generation System*

This approach is interesting in that it addresses an important issue. Most of the experiments and systems that we have analyzed have as objective to generate music in a certain style, more precisely by favoring some *similarity* with the training corpus. The experiment by Akin Kazakçi *et al.* [55] favors *originality* . The domain is handwritten digits, the corpus being the reference MNIST dataset [64].

The architecture is a sparse convolutional autoencoder, with three successive encoding layers and one decoding layer. It is trained on the MNIST dataset. The process of generation of *new digits* is as follows:

- feedforward a randomly generated image into the autoencoder;
- reiterate feeding in the autoencoder previous output until reaching a *fixed point* (stable image).

Traces of successive transformations of a random image are shown at Figure 9.7. Figure 9.8 shows the typicality of new non-digits via a clusterization process (using the k-Means clustering non supervised algorithm, see [29, Section 5.8.2]). Then the clusters are projected onto a 2-dimensional space for visualization, using the standard t-SNE algorithm [110]. Colored clusters are the original MNIST types (digit classes from 0 to 9). The circles

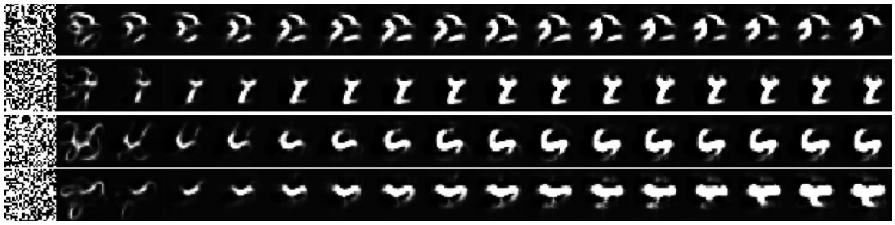

Fig. 9.7 4 examples (traces) of convergences to new non-digits.

mark 4 of the clusters of generated non-digits, with some of elements of each cluster shown in the associated black square.

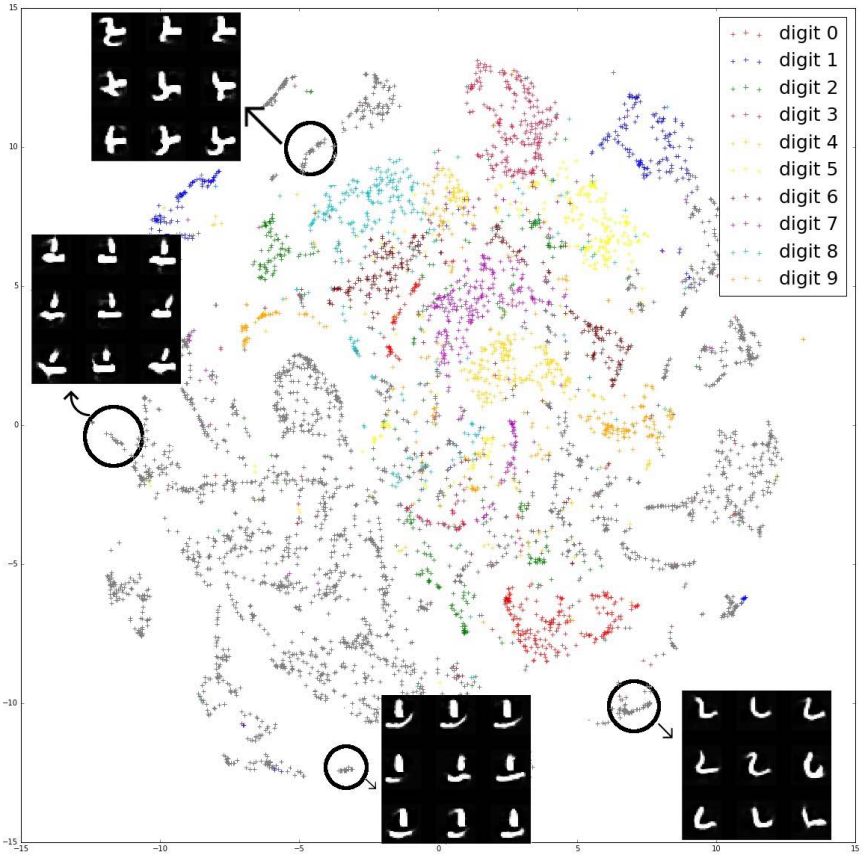

Fig. 9.8 The space of standard digits and generated non-digits.

9.3.2 Elgammal et al.*'s Creative Adversarial Networks Painting Generation System*

Ahmed Elgammal *et al.* propose in [22] to address the issue of *creativity* by extending a generative adversarial networks architecture (GAN) architecture into a *Creative Adversarial Networks (CAN)* architecture to "generate art by learning about styles and deviating from style norms."

Their assumption is that in a standard GAN architecture, the generator objective is to generate images that fool the discriminator and, as a consequence, the generator is trained to be *emulative* but not *creative*. In their proposed creative adversarial networks (CAN), the generator receives from the discriminator not just one but two signals. The first signal is analog to the case of the standard GAN (see Equation 5.7) and specifies how the discriminator believes that the generated item comes from the training dataset of real art pieces. The second signal is about easily the discriminator can classify the generated item into established styles. If there is strong ambiguity (i.e., the various classes are equiprobable), this means that the generated item is difficult to fit within the existing art styles. These two signals are thus contradictory forces and push the generator to explore the space for generating items that are at the same time close to the distribution of existing art pieces and with some style originality. The CAN architecture is illustrated at Figure 9.9.

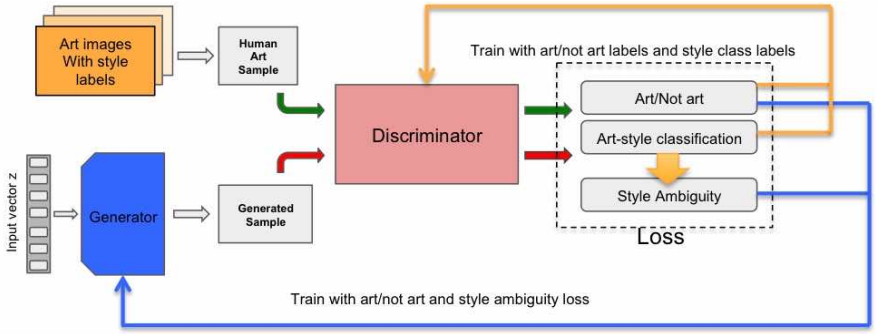

Fig. 9.9 CAN architecture.

Experiments have been done with a dataset of paintings from a WikiArt dataset 3 [116]. This collection has images of 81,449 paintings from 1,119 artists ranging from the 15th century to 20th century. It has been tagged with 25 possible painting styles (e.g., cubism, fauvism, high-renaissance, impressionism, pop-art, realism...). Some examples of generated images are shown at Figure 9.10.

Note that, as the authors discuss, the generated images indeed do not look like traditional art, in terms of standard genres (portrait, landscapes, religious paintings, still life, etc.). They have conducted a preliminary external evaluation and also a preliminary analysis of their approach. Their approach is actually relatively simple to implement and interesting to analyze further. It relies on an existing style classification and tends to reduce the idea of creativity to explore new styles (which indeed has some grounding in the art history). The necessary prior classification between different styles does have an important role and it will be interesting to experiment with other types of classification. As for the style transfer approach (described in Section 9.2.1), a musical style seems less easy to capture just as correlations of neuron activations or as music classification descriptors but, giving the large amount of techniques on classification and retrieval of music[4], experiments appear promising.

[4] See, e.g., the proceedings of the series of conferences of the International Society of Music Information Retrieval (ISMIR) [51].

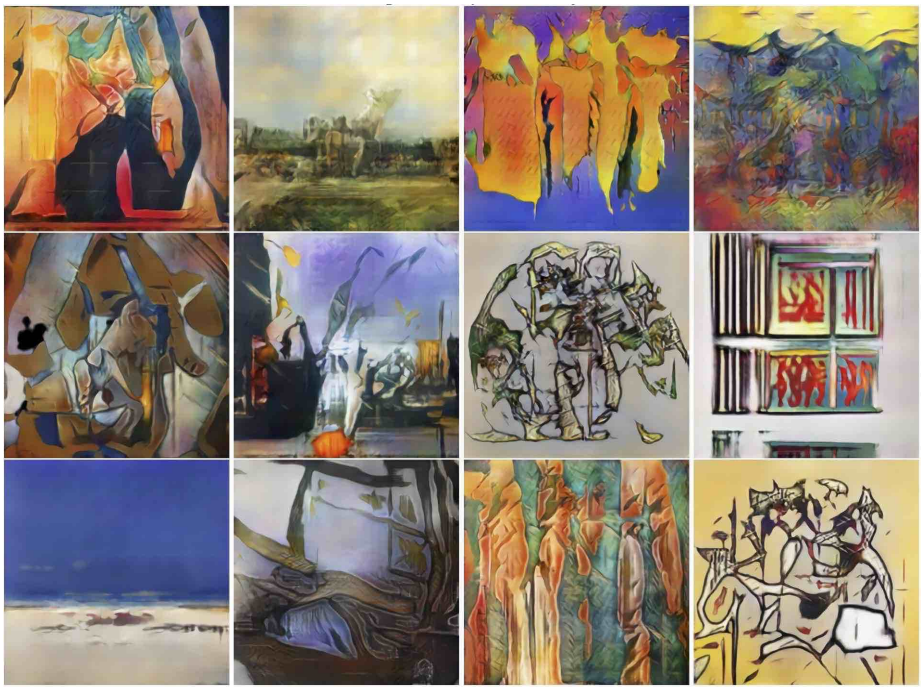

Fig. 9.10 Examples of images generated by CAN.

Chapter 10
Discussion

We now point out some potential lessons learnt and issues raised through our analysis.

10.1 Convolution vs Recurrent

As noted in Section 5.3.8, convolutional architectures, while prevalent for image applications, are rarely used in music applications, because, as opposed to images where motives are *a priori* invariant in all dimensions, in music a dimension such as *pitch* is *a priori* not metrically invariant (see a counter example in Section 7.3.3.1). Furthermore, recurrent architectures are more commonly used to model the invariance in time (see Section 5.3.8). We identified only one non recurrent (feedforward) network architecture using convolution in time: a convolutional RBM, described in Section 7.7.1.1. As there are many systems using non recurrent architectures (like feedforward networks or autoencoders), it seems interesting to study if their extension into a convolutional architecture on the time dimension would bring some effective gain, e.g., in accuracy.

A practical aspect is that convolutional networks are typically faster to train and more easily to parallelize than recurrent networks [109]. Also, the authors of WaveNet [108] (see Section 7.5.1.1) argue that layers of dilated convolutions allow the receptive field to grow longer in a much cheaper way than using LSTM units. One limitation is that, although convolution applied onto the time dimension allows sharing weights in a similar way to recurrent networks, it is limited, as it applies to only a small number of temporal neighboring members of the input. This is in contrast to a RNN that shares parameters in a deep way, for all time steps (see the related comparison in Section 5.3.8). Meanwhile, the authors of MidiNet [119] (see Section 7.9.1.1) argue that using a conditioning strategy to a convolutional architecture allows to incorporate information from previous measures to intermediate layers and therefore to consider history as would do a recurrent network.

10.2 Adversarial and Beyond

The concept of Generative adversarial networks (GAN) is an important recent conceptual progress to construct generative networks. But actual training is challenging because of the min-max optimization objective (see Section 5.3.10). A recent proposed alternative both to GANs and to autoencoders is *Generative Latent Optimization* (GLO) [5]. It is an approach to train a generator without the need to learn a discriminator, by learning a mapping from noise vectors to images. GLO can thus be viewed both as an encoder-less autoencoder, or as a discriminator-less GAN. It can also be used for conditional generation. GLO has been

tested on images and not yet to music. It looks as a promising new approach in need for more evaluation.

10.3 Transfer Learning

Transfer learning is an important issue for deep learning and machine learning in general. As training can be a tedious process, the issue is to be able to *reuse*, at least partially, what has been learnt in some context and use it in other contexts[1]. Transfer learning is about methodologies and techniques for such transfer of what has been learnt [29, Chapter 15]. We have not addressed this important issue in our analysis, because is has not yet been specifically addressed in the music generation systems. But we think that it can become an area of investigation.

Note that style transfer (see Section 9.2) is actually a specific example of transfer learning. Also, although a simple and specific case, note that the way the DeepHear architecture and what it has learnt is reused is some transfer from the objective of generating a melody to the objective of generating a counterpoint (see Section 7.6.1.1).

10.4 Evaluation

For many experiments, *evaluation* is only preliminary, and in many cases, only conducted by the designer himself. There are of course exceptions, with some more systematic and external evaluations, e.g., for the DeepBach system (see Section 7.2.2.1). Moreover, as discussed in [103], evaluation is usually *multicriteria* (e.g., accuracy, likelihood...) and a good performance with respect to one criterion does not necessarily imply a good performance with respect to another criterion.

10.5 User Interaction

An important issue is that for most of current systems, content generation is an automated and autonomous process. Interactivity with the user is a fundamental concern in order to have a companion system and not an automated system, to help humans in their musical tasks (composition, counterpoint, harmonization, arranging, etc.) in an incremental and interactive manner (such as, e.g., showed by the FlowComposer prototype [87]). Some examples of such partially interactive incremental systems are deepAutoController (Section 7.1.3.2) and DeepBach (Section 7.2.2.1).

10.6 Cooperation

All systems surveyed are a unique generating system. A more cooperative approach is interesting as it is a natural approach to handle complexity, heterogeneity, scalability and openness, as, e.g., pioneered by multi-agent systems [117]. An exception[2] is the system proposed by Patrick Hutchings and Jon McCormack [50]. It is composed of two agents: an

[1] Various situations may be considered, e.g., similar source and target domains, same task, etc.

[2] For deep learning architectures, as there have already been various proposals for multi-agent based music generation systems.

harmony agent based on a RNN (LSTM) architecture in charge of the progression of chords and a melody agent based on a rule-based system in charge of melodies. The two agents work in a cooperative way and alternate between leading and accompanying roles (inspired by, e.g., the way Jazz musicians function). The authors relate the interesting dynamics between the two agents and also a seed for signs of creativity[3]. This approach appears as an interesting direction to pursue (and to be extended with more agents and roles).

10.7 Adaptation

One fundamental limitation of current deep learning architectures for the generation of musical content is that, *paradoxically*, they *do not learn* nor *adapt*. Learning is applied during the *training* phase of the network, but no learning or adaptation occurs during the *generation* phase. Meanwhile, one can imagine some *feedback* from a user, about the *quality* and the *adequacy* of the music generated. This feedback may be *explicit*, which puts a task on the user, but it could also be, at least partly *automated*. For instance, the fact that the user quickly stops listening to the music just generated could be interpreted as a *negative* feedback. On the contrary, the fact that the user selects a better rendering after a first quick hear at a straightforward MIDI type audio generation could be interpreted as a *positive* feedback.

Several approaches are possible. The more straightforward approach, considering the nature of neural networks and supervised learning, would be to add the new generated musical piece to the training set and eventually[4] retrain the network. (This could be done in the background). This would reinforce the amount of positive examples and gradually update the learnt model and future generations. However, there is no guarantee that the overall generation quality would improve. This could also lead for the model to *overfit* and loose some generalization. Moreover, there is no direct possibility of *negative feedback*, as one cannot remove from the dataset a bad example generated, because there is almost no chance that it was already present in the dataset.

Another approach is to work not on the *training* dataset but on the *generation* phase. This leads us to go back to the issue of control, via, e.g., constrained sampling strategy, input manipulation strategy and obviously reinforcement strategy, as the control objective and parameters should be *adaptive*. The RL-Tuner framework (see Section 7.10.1.1) is an interesting step towards this. Although, the initial motivation for RL-Tuner is to introduce musical constraints onto the generation, by encapsulating them into an additional reward, this approach could also be used to introduce user feedback as an additional reward (e.g., as in the following RL-based generation system [33]). Although such experiments are preliminary, they show the way for future exploration and research.

10.8 Structure

Some issue is that most of existing systems do not allow the emergence or manipulation of higher-level structures. An example of recurrent structure in the Jazz and song domain is a structure such as AABA or AAB. On the contrary, current systems have a tendency to generate music with no clear structure and sense of direction. The reinforcement strategy, such as in the RL-Tuner system [52] (see Section 7.10.1.1), and the controlled sampling strategy, such as in the C-RBM system [61] (see Section 7.7.1), are approaches to enforce some constraints, possibly high-level, onto the generation and therefore are also interesting

[3] On this issue, see Section 10.9.

[4] immediately or after some time or some amount of new feedbacks, as for a *mini-batch*, see Section 5.1.3.

directions for *structure imposition*. On this issue, see also, e.g., a recent proposal combining two graphical models, one for chords and one for melody, for the generation of lead sheets with an imposed structure [83]. Note that the unit selection strategy is also an interesting direction in that generation is typically top-down (generating a sequence abstract structure and filling it with musical units) as opposed to most of current systems, although the generated structure is not yet very high-level as it stays basically at the measure level.

10.9 Originality

The issue of the *originality* of the music generated is not only an artistic issue but also an economic one, because it raises the issue of the intellectual property and the copyright[5]. One approach is *a posteriori*, by ensuring that the generated music is not too similar (e.g., in not having recopied a significant amount of notes of a melody) to an existing piece of music. Therefore existing tools to detect similarities in texts may be used. Another approach, more systematic but even more challenging, is *a priori*, by ensuring that the music generated will not recopy a given portion of music from the training corpus[6]. A solution for music generation from Markov chains has been proposed [86] but there is none yet for generation from deep architectures.

One possible direction is the concept of creative adversarial networks *(CAN)* [22], as an extension of generative adversarial networks (GAN) architecture (see Section 9.3.2). The idea is to add to the GAN's objective a *style ambiguity* objective, which expresses the difficulty to properly classify data into existing style classes, meaning that it should belong to a new style class. The idea is interesting as a direction to further explore.

10.10 Explanation

A common criticism of *sub-symbolic* approaches of Artificial Intelligence (AI), such as neural networks and deep learning, often considered as *black boxes* is the issue of its explainability [9]. This is indeed a real issue, to be able to explain what and how a deep learning system has learned and how and why it generates a specific content. Being able to better understand this issue also indirectly addresses the issue of how to control the generation of a system. Although preliminary, some interesting study conducted with the BachBot system (presented in Section 7.3.1.3) is about the analysis of the specialization of some of the nodes (neurons) of the network, through some correlation analysis with some specific motives and progressions (see more details in [66, Chapter 5]). In the images domain, see also, e.g., the *a posteriori* discovery of the *cat neuron* in the deep unsupervised learning experiment described in [63].

10.11 Specialization

Another issue is the hyperspecialization of systems designed for a specific objective or/and a specific type of corpus. This is witnessed by the diversity of architectures and approaches surveyed. Note that this is a known issue for Artificial Intelligence (AI) research in general, hyper specializing in solving specific problems, specially in the case of contests organized

[5] On this issue, see a recent paper [18].

[6] Note that this addresses the issue of avoiding a significant recopy from the training corpus, but it does not prevent to *reinvent* an existing music outside of the training corpus.

by conferences or other institutions, and loosing a bit the more general objective of a general problem solving framework.

Of course, the general objective of generating interesting musical content is complex and yet far away, Thus, we need to work both on general approaches for general problems and specific approaches for specific sub problems, as well as more top-down or bottom-up approaches, while not loosing concerns on how to interpret, generalize and reuse advances and lessons learnt. We hope that this survey will contribute to this agenda.

Chapter 11
Conclusion

This book presented a survey and an analysis of various ways and techniques of using deep learning to generate musical content. We proposed a multi-criteria analysis based on four dimensions: objective, representation, architecture and strategy. We analyzed and compared various systems and experiments proposed by various researchers. We hope that this survey provides a conceptual framework to understand and compare various perspectives and approaches for using deep learning for music generation and therefore contributes to this research agenda.

References

1. Nam Hyuk Ahn. Generative adversarial network, January 2017. https://www.slideshare.net/nmhkahn/generative-adversarial-network-laplacian-pyramid-gan.
2. Moray Allan and Christopher K. I. Williams. Harmonising chorales by probabilistic inference. *Advances in neural information processing systems*, 17:25–32, 2005.
3. Apple. vImage programming guide, Accessed on 06/01/2017. https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage.
4. Johann Sebastian Bach. *389 Chorales (Choral-Gesange)*. Alfred Publishing Company, 1985.
5. Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks, July 2017. arXiv:1707.05776v1.
6. Nicolas Boulanger-Lewandowski. Chapter 14th – Modeling and generating sequences of polyphonic music with the RNN-RBM. In *DeepLearning Tutorial – Release 0.1*, pages 149–158. LISA lab, University of Montréal, September 2015. http://deeplearning.net/tutorial/deeplearning.pdf.
7. Nicolas Boulanger-Lewandowski, Yoshua Bengio, and 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-12)*, pages 1159–1166, Edinburgh, Scotland, U.K., 2012.
8. Mason Bretan, Gil Weinberg, and Larry Heck. A unit selection methodology for music generation using deep neural networks, December 2016. arXiv:1612.03789v1.
9. Davide Castelvecchi. The black box of AI. *Nature*, 538:20–23, October 2016.
10. Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN Encoder-Decoder for statistical machine translation, September 2014. arXiv:1406.1078v3.
11. Keunwoo Choi, George Fazekas, and Mark Sandler. Text-based LSTM networks for automatic music composition, April 2016. arXiv:1604.05358v1.
12. Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks, January 2015. arXiv:1412.0233v3.
13. Junyoung Chung, Caglar Gulcehre, Kyung Hyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, December 2014. arXiv:1412.3555v1.
14. Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014. arXiv:1412.3555.
15. David Cope. *The Algorithmic Composer*. A-R Editions, 2000.
16. Fabrizio Costa, Thomas Gärtner, Andrea Passerini, and François Pachet. Constructive Machine Learning – Workshop Proceedings, December 2016. http://www.cs.nott.ac.uk/ psztg/cml/2016/.
17. Yann Le Cun and Yoshua Bengio. Convolutional networks for images, speech, and time-series. In *The handbook of brain theory and neural networks*, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.
18. Jean-Marc Deltorn. Deep creations: Intellectual property and the automata. *Frontiers in Digital Humanities*, 4, February 2017. Article 3.
19. Carl Doersch. Tutorial on variational autoencoders, August 2016. arXiv:1606.05908v2.
20. Kenji Doya and Eiji Uchibe. The Cyber Rodent project: Exploration of adaptive mechanisms for self-preservation and self-reproduction. *Adaptive Behavior*, (2):149–160, 2005.
21. Douglas Eck and Jürgen Schmidhuber. A first look at music composition using LSTM recurrent neural networks. Technical report, IDSIA/USI-SUPSI, Manno, Switzerland, 2002. Technical Report No. IDSIA-07-02.
22. Ahmed Elgammal, Bingchen Liu, Mohamed Elhoseiny, and Marian Mazzone. CAN: Creative adversarial networks generating "art" by learning about styles and deviating from style norms, June 2017. arXiv:1706.07068v1.
23. Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, and Pascal Vincent. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, (11):625–660, 2010.
24. Otto Fabius and Joost R. van Amersfoort. Variational Recurrent Auto-Encoders, June 2015. arXiv:1412.6581v6.
25. Davis Foote, Daylen Yang, and Mostafa Rohaninejad. Audio style transfer – Do androids dream of electric beats?, December 2016. https://audiostyletransfer.wordpress.com.
26. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
27. Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, September 2015. arXiv:1508.06576v2.
28. Kratarth Goel, Raunaq Vohra, and J. K. Sahoo. Polyphonic music generation by modeling temporal dependencies using a RNN-DBN. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 217–224. Springer, 2014.
29. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

30. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozairy, Aaron Courville, and Yoshua Bengio. Generative adversarial nets, June 2014. arXiv:1406.2661v1.
31. Alex Graves. Generating sequences with recurrent neural networks, June 2014. arXiv:1308.0850v5.
32. Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines, December 2014. arXiv:1410.5401v2.
33. Sylvain Le Groux and Paul F.M.J. Verschure. Adaptive music generation by reinforcement learning of musical tension. In *Proceedings of the 7th Sound and Music Computing Conference (SMC'2010)*, Barcelona, Spain, 2010.
34. Christopher J. C. H. and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.
35. Gaëtan Hadjeres, Frank Nielsen, and François Pachet. GLSR-VAE: Geodesic latent space regularization for variational autoencoder architectures, July 2017. arXiv:1707.04588v1.
36. Gaëtan Hadjeres, François Pachet, and Frank Nielsen. DeepBach: a steerable model for Bach chorales generation, June 2017. arXiv:1612.01010v2.
37. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, December 2015. arXiv:1512.03385v1.
38. Dorien Herremans and Ching-Hua Chuan. Deep Learning for Music – Workshop Proceedings, May 2017. http://dorienherremans.com/dlm2017/.
39. Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, August 2002.
40. Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, July 2006.
41. Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, (18):1527–1554, 2006.
42. Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
43. Geoffrey E. Hinton and Terrence J. Sejnowski. Learning and relearning in Boltzmann machines. In David E. Rumelhart, James L. McClelland, and PDP Research Group, editors, *Parallel Distributed Processing – Explorations in the Microstructure of Cognition: Volume 1 Foundations*, page 282?317. MIT Press, Cambridge, MA, USA, 1986.
44. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
45. Hooktheory. Theorytabs, Accessed on 26/07/2017. https://www.hooktheory.com/theorytab.
46. Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.
47. Allen Huang and Raymond Wu. Deep learning for music, June 2016. arXiv:1606.04930v1.
48. Cheng-Zhi Anna Huang, David Duvenaud, and Krzysztof Z. Gajos. ChordRipple: Recommending chords to help novice composers go beyond the ordinary. In *Proceedings of the 21st International Conference on Intelligent User Interfaces (IUI'16)*, pages 241–250, Sonoma, CA, USA, March 2016. ACM.
49. Eric J. Humphrey, Juan P. Bello, and Yann LeCun. Feature learning and deep architectures: New directions for music informatics. *Journal of Intelligent Information Systems*, 41(3):461–481, 2013.
50. Patrick Hutchings and Jon McCormack. Using autonomous agents to improvise music compositions in real-time. In João Correia, Vic Ciesielski, and Antonios Liapis, editors, *Computational Intelligence in Music, Sound, Art and Design – 6th International Conference, EvoMUSART 2017, Amsterdam, The Netherlands, April 1921, 2017, Proceedings*, number 10198 in LNCS, pages 114–127. Springer, 2017.
51. ISMIR. International Society for Music Information Retrieval Conference(s) (Proceedings), Accessed on 23/08/2017. http://dblp.uni-trier.de/db/conf/ismir/.
52. Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning, November 2016. arXiv:1611.02796.
53. Daniel Johnson. Composing music with recurrent neural networks, August 2015. http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/.
54. Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, (4):237–285, 1996.
55. Akin Kazakçi, Mehdi Cherti, and Balázs Kégl. Digits that are not: Generating new types through deep neural nets. In François Pachet, Amilcar Cardoso, Vincent Corruble, and Fiammetta Ghedini, editors, *Proceedings of the 7th International Conference on Computational Creativity (ICCC'2016)*, pages 188–195, Paris, France, June 2016. https://arxiv.org/abs/1606.04345.
56. Jeremy Keith. The Session, Accessed on 21/12/2016. https://thesession.org.
57. Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes, May 2014. arXiv:1312.6114v10.
58. Andrey Kurenkov. A 'brief' history of neural nets and deep learning, Part 4, December 2015. http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-4/.
59. Patrick Lam. MCMC methods: Gibbs sampling and the Metropolis-Hastings algorithm, Accessed on 21/12/2016. http://pareto.uab.es/mcreel/IDEA2017/Bayesian/MCMC/mcmc.pdf.

60. Kevin J. Lang, Alex H. Waibel, and Geoffrey E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43, 1990.
61. Stefan Lattner, Maarten Grachten, and Gerhard Widmer. Imposing higher-level structure in polyphonic music generation using convolutional restricted Boltzmann machines and constraints, December 2016. arXiv:1612.04742v2.
62. Edith Law and Luis von Ahn. Input-agreement: A new mechanism for collecting data using human computation games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'2009)*, pages 1197–1206. ACM, 2009.
63. Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. In *29th International Conference on Machine Learning*, Edinburgh, U.K., 2012.
64. Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/.
65. Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Convolutional neural networks (CNNs / ConvNets) – Stanford University CS231n Convolutional neural networks for visual recognition Lecture Notes, Winter 2016. http://cs231n.github.io/convolutional-networks/#conv.
66. Feynman Liang. BachBot: Automatic composition in the style of Bach chorales – Developing, analyzing, and evaluating a deep LSTM model for musical style. Master's thesis, University of Cambridge, Cambridge, U.K., August 2016. M.Phil in Machine Learning, Speech, and Language Technology.
67. Qi Lyu, Zhiyong Wu, Jun Zhu, and Helen Meng. Modelling high-dimensional sequences with LSTM-RTRBM: Application to polyphonic music generation. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 4138–4139. AAAI Press, 2015.
68. Sephora Madjiheurem, Lizhen Qu, and Christian Walder. Chord2Vec: Learning musical chord embeddings. In *Proceedings of the Constructive Machine Learning Workshop at 30th Conference on Neural Information Processing Systems (NIPS'2016)*, Barcelona, Spain, December 2016.
69. Dimos Makris, Maximos Kaliakatsos-Papakostas, Ioannis Karydis, and Katia Lida Kermanidis. Combining LSTM and feed forward neural networks for conditional rhythm composition. In Giacomo Boracchi, Lazaros Iliadis, Chrisina Jayne, and Aristidis Likas, editors, *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings*, pages 570–582. Springer, 2017.
70. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, September 2013. arXiv:1301.3781v3.
71. Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.
72. Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, November 2014. arXiv:1411.1784v1.
73. MIDI Manufacturers Association (MMA). MIDI Specifications, Accessed on 14/04/2017. https://www.midi.org/specifications.
74. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning, December 2013. arXiv:1312.5602v1.
75. Olof Mogren. C-RNN-GAN: Continuous recurrent neural networks with adversarial training, November 2016. arXiv:1611.09904v1.
76. Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Deep Dream, 2015. https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.
77. Michael C. Mozer. Neural network composition by prediction: Exploring the benefits of psychophysical constraints and multiscale processing. *Connection Science*, 6(2–3):247–280, 1994.
78. Andrew Ng. Supervised Learning – Part IV: Generative Learning Algorithms – Stanford University CS229 Machine Learning Course Lecture Notes, Autumn 2016. http://cs229.stanford.edu/notes/cs229-notes2.pdf.
79. Andrew Ng. Supervised Learning – Parts I–III: Linear Regression; Classification and Logistic Regression; Generalized Linear Models – Stanford University CS229 Machine Learning Course Lecture Notes, Autumn 2016. http://cs229.stanford.edu/notes/cs229-notes1.pdf.
80. Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer, 2009.
81. François Pachet. Flow Machines Project, Accessed on 20/06/2017. http://www.flow-machines.com.
82. François Pachet, Jeff Suzda, and Daniel Martín. A comprehensive online database of machine-readable leadsheets for jazz standards. In Alceu de Souza Britto Junior, Fabien Gouyon, and Simon Dixon, editors, *Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR'2013)*, pages 275–280, Curitiba, Brazil, November 2013.
83. François Pachet, Alexandre. Papadopoulos, and Pierre Roy. Sampling variations of sequences for structured music generation. In *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR'2017)*, pages 23–27, Suzhou, China, October 2017. ISMIR.
84. François Pachet and Pierre Roy. Markov constraints: Steerable generation of Markov sequences. *Constraints*, 16(2):148–172, 2011.

85. Alexandre Papadopoulos, François Pachet, Pierre Roy, and Jason Sakellariou. Exact sampling for regular and Markov constraints with belief propagation. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP'2015)*, pages 341–350, Cork, Ireland, August–September 2015.
86. Alexandre Papadopoulos, Pierre Roy, and François Pachet. Avoiding plagiarism in Markov sequence generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2731–2737, Québec, PQ, Canada, July 2014.
87. Alexandre Papadopoulos, Pierre Roy, and François Pachet. Assisted lead sheet composition using FlowComposer. In Michel Rueher, editor, *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 769–785. Springer, 2016.
88. Curtis Roads. MIT Press, 1996.
89. Frank Rosenblatt. The Perceptron – A perceiving and recognizing automaton. Technical report, Cornell Aeronautical Laboratory, Ithaca, NY, USA, 1957. Report 85-460-1.
90. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.
91. David E. Rumelhart, James L. McClelland, and PDP Research Group. *Parallel Distributed Processing – Explorations in the Microstructure of Cognition: Volume 1 Foundations*. MIT Press, Cambridge, MA, USA, 1986.
92. Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs, June 2016. arXiv:1606.03498v1.
93. Andy M. Sarroff and Michael Casey. Musical audio synthesis using autoencoding neural nets, 2014. http://www.cs.dartmouth.edu/ sarroff/papers/sarroff2014a.pdf.
94. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
95. Roger N. Shepard. Geometric approximations to the structure of musical pitch. *Psychological Review*, (89):305–333, 1982.
96. Bob L. Sturm and João Felipe Santos. The endless traditional music session, Accessed on 21/12/2016. http://www.eecs.qmul.ac.uk/%7Esturm/research/RNNIrishTrad/.
97. Bob L. Sturm, João Felipe Santos, Oded Ben-Tal, and Iryna Korshunova. Music transcription modelling and composition using deep learning, April 2016. arXiv:1604.08723v1.
98. Felix Sun. DeepHear – Composing and harmonizing music with neural networks, Accessed on 21/12/2016. http://web.mit.edu/felixsun/www/index.html?neural-music.html.
99. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, September 2014. arXiv:1409.4842v1.
100. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, February 2014. arXiv:1312.6199v4.
101. Theano Development Team. Deeplearning 0.1 documentation, Accessed on 14/04/2017. http://deeplearning.net/tutorial/lstm.html.
102. David Temperley. *The Cognition of Basic Musical Structures*. MIT Press, Cambridge, MA, USA, 2011.
103. Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models, 2015. arXiv:1511.01844.
104. John Thickstun, Zaid Harchaoui, and Sham Kakade. Learning features of music from scratch, December 2016. arXiv:1611.09827.
105. Alexey Tikhonov and Ivan P. Yamshchikov. Music generation with variational recurrent autoencoder supported by history, July 2017. arXiv:1705.05458v2.
106. Peter M. Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.
107. Dmitry Ulyanov and Vadim Lebedev. Audio texture synthesis and style transfer, December 2016. https://dmitryulyanov.github.io/audio-texture-synthesis-and-style-transfer/.
108. Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio, December 2016. arXiv:1609.03499v2.
109. Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with PixelCNN decoders, June 2016. arXiv:1606.05328v2.
110. Laurens van der Maaten and Geoffrey H. Hinton. Visualizing data using t-SNE. *The Journal of Machine Learning Research*, 9:2579–2605, 2008.
111. Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning, December 2015. arXiv:1509.06461v3.
112. Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.
113. Christian Walder. Modelling symbolic music: Beyond the piano roll, June 2016. arXiv:1606.01368.
114. Christian Walder. Symbolic Music Data Version 1.0, June 2016. arXiv:1606.02542.

115. Chris Walshaw. abc notation home page, Accessed on 21/12/2016. http://abcnotation.com.
116. WikiArt.org. WikiArt, Accessed on 22/08/2017. https://www.wikiart.org.
117. Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009.
118. Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2Image: Conditional image generation from visual attributes, October 2016. arXiv:1512.00570v2.
119. Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. MidiNet: A convolutional generative adversarial network for symbolic-domain music generation. In *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR'2017)*, Suzhou, China, October 2017.
120. Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks, July 2017. arXiv:1607.03474v5.