Deep Learning Techniques for Music Generation – *A Survey*

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

[1] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France
[2] Sony Computer Science Laboratories, CSL-Paris, F-75005 Paris, France
[3] Spotify Creator Technology Research Lab, CTRL, F-75008 Paris, France

Abstract This paper is a survey and an analysis of different ways of using deep learning (deep artificial neural networks) to generate musical content.

We propose a methodology based on five dimensions for our analysis:

- *Objective – What musical content is to be generated?* E.g., melody, polyphony, accompaniment and counterpoint – *For what destination and for what use?* To be performed by a human(s) or by a machine.
- *Representation – What are the concepts to be manipulated?* E.g., waveform, spectrogram, note, chord, meter, and beat – *What format is to be used?* E.g., MIDI, piano roll and text – *How will the representation be encoded?* E.g., scalar, one-hot, and many-hot.
- *Architecture – What type of deep neural network is to be used?* E.g., feedforward network, recurrent network, autoencoder, and generative adversarial networks.
- *Challenges – What are the limitations and open challenges?* E.g., variability, interactivity and creativity.
- *Strategy – How do we model and control the process of generation?* E.g., single-step feedforward, decoder feedforward, sampling and input manipulation.

For each dimension, we conduct a comparative analysis of various models and techniques and we propose some tentative multidimensional typology. This typology is *bottom-up*, based on the analysis of many existing deep-learning based systems for music generation selected from the relevant literature. These systems are described in this survey/analysis and are used to exemplify the various choices of objective, representation, architecture, challenges and strategies. The final part of the paper includes some discussion and some prospects.

This paper is a simplified (weak DRM) version of the following book [14]: Jean-Pierre Briot, Gaëtan Hadjeres and François Pachet, Deep Learning Techniques for Music Generation, Computational Synthesis and Creative Systems, Springer Nature, 2019. Hardcover ISBN: 978-3-319-70162-2. eBook ISBN: 978-3-319-70163-9. Series ISSN: 2509-6575.

Chapter 1
Introduction

1.1 Context

Deep learning has recently become a fast growing domain and is now used routinely for classification and prediction tasks, such as image recognition, voice recognition or translation. It became popular in 2012, when a deep learning architecture significantly outperformed standard techniques relying on handcrafted features in an image classification competition, see more details in Section 5.1.

We may explain this success and reemergence of artificial neural network techniques by the combination of:

- availability of *massive data*;
- availability of *efficient and affordable computing power*[1];
- *technical advances*, such as:
  - *pre-training*, which resolved initially inefficient training of neural networks with many layers [74][2];
  - *convolutions*, which provide motif translation invariance [102];
  - LSTM (Long short-term memory), which resolved initially inefficient training of recurrent neural networks [77].

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*. This means that there are multiple layers processing multiple hierarchical levels of abstractions, which are automatically extracted from data[3]. Thus a deep architecture can manage and decompose complex representations in terms of simpler representations. The technical foundation is mostly artificial neural networks, as we will see in Chapter 5, with many extensions, such as: convolutional networks, recurrent networks, autoencoders, and 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 [58].

Driving applications of deep learning are traditional machine learning tasks[4]: *classification* (for instance, identification of images) and *prediction*[5] (for instance, of the weather) and also more recent ones such as *translation*.

But a growing area of application of deep learning techniques is the *generation of content*. Content can be of various kinds: images, text and music, the latter being the focus of our analysis. The motivation is in using now widely available various corpora to automatically learn musical *styles* and to generate *new* musical content based on this.

---

[1] Notably, thanks to graphics processing units (GPU), initially designed for video games, which have now one of their biggest markets in data science and deep learning applications.

[2] Although nowadays it has being replaced by other techniques, such as batch normalization [85] and deep residual learning [69].

[3] That said, although deep learning will automatically extract significant features from the data, manual choices of input representation, e.g., spectrum vs raw wave signal for audio, may be very significant for the accuracy of the learning and for the quality of the generated content, see Section 4.9.3.

[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 [58, Section 5.1.1]. Examples are: classification, regression and anomaly detection.

[5] As a testimony of the initial DNA of neural networks: *linear regression* and *logistic regression*, see Section 5.2.

1.2 Motivation

1.2.1 Computer-Based Music Systems

The first music generated by computer appeared in 1957. It was a 17 seconds long melody named "The Silver Scale" by its author Newman Guttman and was generated by a software for sound synthesis named Music I, developed by Mathews at Bell Laboratories. The same year, "The Illiac Suite" was the first score composed by a computer [72]. It was named after the ILLIAC I computer at the University of Illinois at Urbana-Champaign (UIUC) in the United States. The human "meta-composers" were Hiller and Isaacson, both musicians and scientists. It was an early example of algorithmic composition, making use of stochastic models (Markov chains) for generation as well as rules to filter generated material according to desired properties.

In the domain of sound synthesis, a landmark was the release in 1983 by Yamaha of the DX 7 synthesizer, building on groundwork by Chowning on a model of synthesis based on frequency modulation (FM). The same year, the MIDI[6] interface was launched, as a way to interoperate various software and instruments (including the Yamaha DX 7 synthesizer). Another landmark was the development by Puckette at IRCAM of the Max/MSP real-time interactive processing environment, used for real-time synthesis and for interactive performances.

Regarding algorithmic composition, in the early 1960s Iannis Xenakis explored the idea of stochastic composition[7] [188], in his composition named "Atrées" in 1962. The idea involved using computer fast computations to calculate various possibilities from a set of probabilities designed by the composer in order to generate samples of musical pieces to be selected. In another approach following the initial direction of "The Iliac Suite", grammars and rules were used to specify the style of a given corpus or more generally tonal music theory. An example is the generation in the 1980s by Ebcioğlu's composition software named CHORAL of a four-part chorale in the style of Johann Sebastian Bach, according to over 350 handcrafted rules [36]. In the late 1980s David Cope's system named Experiments in Musical Intelligence (EMI) extended that approach with the capacity to learn from a corpus of scores of a composer to create its own grammar and database of rules [24].

Since then, computer music has continued developing for the general public, if we consider, for instance, the GarageBand music composition and production application for Apple platforms (computers, tablets and cellphones), as an offspring of the initial Cubase sequencer software, released by Steinberg in 1989.

For more details about the history and principles of computer music in general, see, for example, the book by Roads [146]. For more details about the history and principles of algorithmic composition, see, for example, [117] and the books by Cope [24] or Dean and McLean [30].

1.2.2 Autonomy versus Assistance

When talking about computer-based music generation, there is actually some ambiguity about whether the objective is

- to design and construct *autonomous* music-making systems – a recent example being the deep-learning based Amper^{TM} system, aimed at the creation of original music for commercials and documentary; or
- to design and construct computer-based environments to *assist* human musicians (composers, arrangers, producers, etc.) – an example being the FlowComposer environment developed at Sony CSL-Paris [141], introduced in Section 6.10.4.

[6] Musical instrument digital interface, to be introduced in Section 4.7.1.

[7] One of the first documented case of *stochastic music*, long before computers, is the Musikalisches Wurfelspiel (Dice Music) by Wolfgang Amadeus Mozart. It was designed for using dice to generate music by concatenating randomly selected predefined music segments composed in a given style (Austrian waltz in a given key).

The quest for autonomous music-making systems may be an interesting perspective for exploring the process of composition[8] and it also serves as an evaluation method. An example of a musical Turing test[9] will be introduced in Section 6.13.2. It consists in presenting to various members of the public (from beginners to experts) chorales composed by J. S. Bach or generated by a deep learning system and played by human musicians[10]. As we will see in the following, deep learning techniques turn out to be very efficient at succeeding in such tests, due to their capacity to learn musical style from a given corpus and to generate new music that fits into this style. That said, we consider that such a test is more a means than an end.

A broader perspective is in assisting human musicians during the various steps of music creation: composition, arranging, orchestration, production, etc. Indeed, to compose or to improvise[11], a musician rarely creates new music from scratch. S/he reuses and adapts, consciously or unconsciously, features from various music that s/he already knows or has heard, while following some principles and guidelines, such as theories about harmony and scales. A computer-based musician assistant may act during different stages of the composition, to initiate, suggest, provoke and/or complement the inspiration of the human composer.

That said, as we will see, the majority of current deep-learning based systems for generating music are still focused on autonomous generation, although more and more systems are addressing the issue of human-level control and interaction.

1.2.3 Symbolic versus Sub-Symbolic AI

Artificial Intelligence (AI) is often divided into two main streams[12]:

- symbolic AI – dealing with high-level symbolic representations (e.g., chords, harmony...) and processes (harmonization, analysis...); and
- sub-symbolic AI – dealing with low-level representations (e.g., sound, timbre...) and processes (pitch recognition, classification...).

Examples of symbolic models used for music are rule-based systems or grammars to represent harmony. Examples of sub-symbolic models used for music are machine learning algorithms for automatically learning musical styles from a corpus of musical pieces. These models can then be used in a generative and interactive manner, to help musicians in creating new music, by taking advantage of this added "intelligent" memory (associative, inductive and generative) to suggest proposals, sketches, extrapolations, mappings, etc. This is now feasible because of the growing availability of music in various forms, e.g., sound, scores and MIDI files, which can be automatically processed by computers.

A recent example of an integrated music composition environment is FlowComposer [141], which we will introduce in Section 6.10.4. It offers various symbolic and sub-symbolic techniques, e.g., Markov chains for modeling style, a constraint solving module for expressing constraints, a rule-based module to produce harmonic analysis; and an audio mapping module to produce rendering. Another example of an integrated music composition environment is OpenMusic [2].

However, a deeper integration of sub-symbolic techniques, such as deep learning, with symbolic techniques, such as constraints and reasoning, is still an open issue[13], although some partial integrations in restricted contexts already exist (see, for example, Markov constraints in [137, 6] and an example of use for FlowComposer in Section 6.10.4).

---

[8] As Richard Feynman coined it: "What I cannot create, I do not understand."

[9] Initially codified in 1950 by Alan Turing and named by him the "imitation game" [173], the "Turing test" is a test of the ability for a machine to exhibit intelligent behavior equivalent to (and more precisely, indistinguishable from) the behavior of a human. In his imaginary experimental setting, Turing proposed the test to be a natural language conversation between a human (the evaluator) and a hidden actor (another human or a machine). If the evaluator cannot reliably tell the machine from the human, the machine is said to have passed the test.

[10] This is to avoid the bias (synthetic flavor) of a computer rendered generated music.

[11] Improvisation is a form of real time composition.

[12] With some precaution, as this division is not that strict.

[13] The general objective of integrating sub-symbolic and symbolic levels into a complete AI system is the Graal of AI.

1.2.4 Deep Learning

The motivation for using deep learning (and more generally machine learning techniques) to generate musical content is its *generality*. As opposed to handcrafted models, such as grammar-based [161] or rule-based music generation systems [36], a machine learning-based generation system can be agnostic, as it learns a model from an arbitrary corpus of music. As a result, the same system may be used for various musical genres.

Therefore, as more large scale musical datasets are made available, a machine learning-based generation system will be able to automatically learn a musical style from a corpus and to generate new musical content. As stated by Fiebrink and Caramiaux [45], some benefits are

- it can make creation feasible when the desired application is too complex to be described by analytical formulations or manual brute force design, and
- learning algorithms are often less brittle than manually designed rule sets and learned rules are more likely to generalize accurately to new contexts in which inputs may change.

Moreover, as opposed to structured representations like rules and grammars, deep learning is good at processing raw unstructured data, from which its hierarchy of layers will extract higher level representations adapted to the task.

1.2.5 Present and Future

As we will see, the research domain in deep learning-based music generation has turned hot recently, building on initial work using artificial networks to generate music (e.g., the pioneering CONCERT system developed in 1994 [127]), while creating an active stream of new ideas and challenges made possible thanks to the progress of deep learning. Let us also note the growing interest by some private actors in the computer-aided generation of artistic content, with the creation by Google in June 2016 of the Magenta research project [41] and the creation by Spotify in September 2017 of the Creator Technology Research Lab (CTRL) [47].

1.2.6 This Book

The lack (to our knowledge) of a comprehensive survey and analysis of this active research domain motivated the writing of this book, built in a *bottom-up* way from the analysis of numerous recent research works. The objective is to provide a comprehensive description of the issues and techniques for using deep learning to generate music, illustrated through the analysis of various architectures, systems and experiments presented in the literature. We also propose a conceptual framework and typology aimed at a better understanding of the design decisions for current as well as future systems.

1.3 Related Work

1.3.1 Books and Other Sources

To our knowledge, there are only a few partial attempts at analyzing the use of deep learning for generating music. In [13], a very preliminary version of this work, Briot *et al.* proposed a first survey of various systems through a multicriteria analysis (considering as dimensions the objective, representation, architecture and strategy). We have extended and consolidated this study by integrating as an additional dimension the challenges (after having analyzed them in [15]).

In [60], Graves presented an analysis focusing on recurrent neural networks and text generation. In [83], Humphrey *et al.* presented another analysis, sharing 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. On MIR applications of deep learning, see also the recent tutorial paper by Choi *et al.* [19].

One could also consult the proceedings of some recently created international workshops on the topic, such as

- the Workshop on Constructive Machine Learning (CML 2016), held during the 30th Annual Conference on Neural Information Processing Systems (NIPS 2016) [26];
- the Workshop on Deep Learning for Music (DLM), held during the International Joint Conference on Neural Networks (IJCNN 2017) [70]; and
- on the deep challenge of *creativity*, the related Series of International Conferences on Computational Creativity (ICCC) [48].

For a more general survey of computer-based techniques to generate music, the reader can refer to general books such as

- Roads' book about computer music [146];
- Cope's [24], Dean and McLean's [30] and/or Nierhaus' books [132] about algorithmic composition;
- a recent survey about AI methods in algorithmic composition [44]; and
- Cope's book about models of musical creativity [25].

About machine learning in general, some examples of texbooks are

- the textbook by Mitchell [120];
- a nice introduction and summary by Domingos [34]; and
- a recent, complete and comprehensive book about deep learning by Goodfellow *et al.* [58].

1.3.2 Other Models

We have to remember that there are various other models and techniques for using computers to generate music, such as rules, grammars, automata, Markov models and graphical models. These models are either *manually* defined by experts or are automatically *learnt* from examples by using various machine learning techniques. They will not be addressed in this book as we are concerned here with deep learning techniques. However, in the following section we make a quick comparison of deep learning and Markov models.

1.3.3 Deep Learning versus Markov Models

Deep learning models are not the only models able to learn musical style from examples. Markov chain models are also widely used, see, for example, [134]. A quick comparison (inspired by the analysis of Mozer in [127][14]) of the pros (+) and cons (–) of deep neural network models and Markov chain models is as follows:

+ Markov models are conceptually simple.
+ Markov models have a simple implementation and a simple learning algorithm, as the model is a transition probability table[15].
– Neural network models are conceptually simple but the optimized implementations of current deep network architectures may be complex and need a lot of tuning.
– Order 1 Markov models (that is, considering only the previous state) do not capture long-term temporal structures.

[14] Note that he made his analysis in in 1994, long before the deep learning wave.

[15] Statistics are collected from the dataset of examples in order to compute the probabilities.

– Order n Markov models (considering n previous states) are possible but require an explosive training set size[16] and can lead to plagiarism[17].
\+ Neural networks can capture various types of relations, contexts and regularities.
\+ Deep networks can learn long-term and high-order dependencies.
\+ Markov models can learn from a few examples.
– Neural networks need a lot of examples in order to be able to learn well.
– Markov models do not generalize very well.
\+ Neural networks generalize better through the use of distributed representations [76].
\+ Markov models are operational models (automata) on which some control on the generation could be attached[18].
– Deep networks are generative models with a distributed representation and therefore with no direct control to be attached[19].

As deep learning implementations are now mature and a large number of examples are available, deep learning-based models are in high demand for their characteristics. That said, other models (such as Markov chains, graphical models, etc.) are still useful and used and the choice of a model and its tuning depends on the characteristics of the problem.

1.4 Requisites and Roadmap

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

Chapter 1 Introduction (this chapter) introduces the purpose and rationale of the book.

Chapter 2 Method introduces the method of analysis (conceptual framework) and the five dimensions at its basis (objective, representation, architecture, challenges and strategy), dimensions that we discuss within the next four chapters.

Chapter 3 Objective concerns the different types of musical content that we want to generate (such as a melody or an accompaniment to an existing melody)[20], as well as their expected use (by a human and/or a machine).

Chapter 4 Representation provides an analysis of the different types of representation and techniques for encoding musical content (such as notes, durations or chords) for a deep learning architecture. This chapter may be skipped by a reader already expert in computer music, although some of the encoding strategies are specific to neural networks and deep learning architectures.

Chapter 5 Architecture summarizes the most common deep learning architectures (such as feedforward, recurrent or autoencoder) used for the 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 Challenges and Strategy provides an analysis of the various challenges that occur when applying deep learning techniques to music generation, as well as various strategies for addressing them. We will ground our study in the analysis of various systems and experiments surveyed from the literature. This chapter is the core of the book.

Chapter 7 Analysis summarizes the survey and analysis conducted in Chapter 6 through some tables as a way to identify the design decisions and their interrelations for the different systems surveyed[21].

---

[16] See the discussion in [127, page 249].

[17] By recopying too long sequences from the corpus. Some promising solution is to consider a variable order Markov model and to constrain the generation (through min order and max order constraints) on some sweet spot between junk and plagiarism [140].

[18] Examples are Markov constraints [137] and factor graphs [136].

[19] This issue as well as some possible solutions will be discussed in Section 6.9.1.

[20] Our proposed typology of possible objectives will turn out to be useful for our analysis because, as we will see, different objectives can lead to different architectures and strategies.

[21] And hopefully also for the future ones. If we draw the analogy (at some meta-level) with the expected ability for a model learnt from a corpus by a machine to be able to generalize to future examples (see Section 5.8.7), we hope that the conceptual framework presented in this book, (manually) inducted from a corpus of scientific and technical literature about deep-learning-based music generation systems, will also be able to help for the design and the understanding of future systems.

Chapter 8 Discussion revisits some of the open issues that were touched in during the analysis of challenges and strategies presented in Chapter 6.

Chapter 9 Conclusion draws together the topics we have addressed along with our findings.

A table of contents, a table of acronyms, a list of references and an index complete this book.

1.5 Limits

This book does not intend to be a general introduction to deep learning – a recent and broad spectrum book on this topic is [58]. We do not intend to get into all technical details of implementation, like engineering and tuning, as well as theory[22], as we wish to focus on the conceptual level, whilst providing a sufficient degree of precision. Also, although having a clear pedagogical objective, we do not provide some end-to-end tutorial with all the steps and details on how to implement and tune a complete deep learning-based music generation system. Last, as this book is about a very active domain and as our survey and analysis is based on existing systems, our analysis is obviously not exhaustive. We have tried to select the most representative proposals and experiments, while new proposals are being presented at the time of our writing. Therefore, we encourage readers and colleagues for any feedback and suggestions for improving this survey and analysis which is a still ongoing project.

[22] For instance, we will not develop the probabilistic foundation of neural networks and deep learning.

Chapter 2
Method

In our analysis, we consider five main *dimensions* to characterize different ways of applying deep learning techniques to generate musical content. This typology is aimed at helping the analysis of the various perspectives (and elements) leading to the design of different deep learning-based music generation systems[1].

The five dimensions that we consider are as follows

- **Objective**[2]
  Which consists in:
  - The musical *nature* of the content to be generated.
    Examples are a melody, a polyphony or an accompaniment; and
  - The *destination* and *use* of the content generated.
    Examples are a musical score to be performed by some human musician(s) or an audio file to be played.
- **Representation**
  The nature and format of the information (data) used to *train* and to *generate* musical content.
  Examples are signal, transformed signal (e.g., a spectrum, via a Fourier transform), piano roll, MIDI and text.
- **Architecture**
  The nature of the assemblage of processing *units* (the artificial neurons) and their *connexions*.
  Examples are a feedforward architecture, a recurrent architecture, an autoencoder architecture and generative adversarial networks.
- **Challenges**
  The nature of the qualities (requirements) that may be desired for music generation.
  Examples are content variability, interactivity and originality.
- **Strategy**
  The way the architecture will process representations in order to *generate*[3] the objective while matching desired requirements.
  Examples are iterative feedforward, decoder feedforward, sampling, and input manipulation.

Note that these five dimensions are not orthogonal. The choice of representation is partially determined by the objective and it also constrains the input and output (interfaces) of the architecture. A given type of architecture also usually leads to a default strategy of use, while new strategies may be designed in order to target specific challenges.

[1] In this book, *systems* refers to the various proposals (architectures, systems and experiments) about deep learning-based music generation that we have surveyed from the literature.

[2] We could have used the term *task* in place of *objective*. However, as task is a relatively well-defined and common term in the machine learning community (see Section 1.1 and [58, Chapter 5]), we preferred an alternative term.

[3] Note, that we consider here the strategy relating to the *generation phase* and not the strategy relating to the training phase, as they could be different.

The exploration of these five different dimensions and of their interplay is actually at the core of our analysis[4]. Each of the first three dimensions (objective, representation and architecture) will be analyzed with its associated typology in a specific chapter, with various illustrative examples and discussion. The challenges and strategy dimensions will be jointly analyzed within the same chapter (Chapter 6) in order to jointly illustrate potential issues (challenges) and possible solutions (strategies). As we will see, the same strategy may relate to more than one challenge and vice versa.

Last, we do not expect our proposed conceptual framework (and its associated five dimensions and related typologies) to be a final result, but rather a first step towards a better understanding of design decisions and challenges for deep learning-based music generation. In other words, it is likely to be further amended and refined, but we hope that it could help bootstrap what we believe to be a necessary comprehensive study.

[4] Let us remember that our proposed typology has been constructed in a *bottom-up* manner from the survey and analysis of numerous systems retrieved from the literature, most of them being very recent.

Chapter 3
Objective

The first dimension, the *objective*, is the nature of the musical content to be generated.

3.1 Facets

We may consider five main *facets* of an objective:

- *Type*
  The musical nature of the generated content.
  Examples are a melody, a polyphony or an accompaniment.
- *Destination*
  The entity aimed at using (processing) the generated content.
  Examples are a human musician, a software or an audio system.
- *Use*
  The way the destination entity will process the generated content.
  Examples are playing an audio file or performing a music score.
- *Mode*
  The way the *generation* will be conducted, i.e. with some human intervention (*interaction*) or without any intervention (*automation*).
- *Style*
  The musical style of the content to be generated.
  Examples are Johann Sebastian Bach chorales, Wolfgang Amadeus Mozart sonatas, Cole Porter songs or Wayne Shorter music. The style will actually be set though the choice of the dataset of musical examples (corpus) used as the training examples.

3.1.1 Type

Main examples of musical types are as follows:

- *Single-voice monophonic melody*, abbreviated as *Melody*
  It is a sequence of notes for a single instrument or vocal, with *at most* one note at the same time.

An example is the music produced by a monophonic instrument like a flute[1].

- *Single-voice polyphony* (also named *Single-track polyphony*), abbreviated as *Polyphony*
  It is a sequence of notes for a single instrument, where more than one note can be played at the same time.
  An example is the music produced by a polyphonic instrument such as a piano or guitar.
- *Multivoice polyphony* (also named *Multitrack polyphony*), abbreviated as *Multivoice* or *Multitrack*
  It is a set of multiple *voices/tracks*, which is intended for more than one voice or instrument.
  Examples are: a chorale with soprano, alto, tenor and bass voices or a Jazz trio with piano, bass and drums.
- *Accompaniment* to a given melody
  Such as
  - *Counterpoint*, composed of one or more melodies (voices); or
  - Sequence of *chords*, which provides some associated *harmony*.
- *Association of melody with chords*
  An example is what is named a *lead sheet*[2] and is common in Jazz. It may also include *lyrics*[3].

Note that the *type* facet is actually the most important facet, as it captures the musical nature of the objective for content generation. In this book, we will frequently identify an objective according to its *type*, e.g., a melody, as a matter of simplification. The next three facets – *destination*, *use* and *mode* – will turn out important when regarding the dimension of the *interaction* of human user(s) with the process of content generation.

3.1.2 Destination and Use

Main examples of destination and use are as follows:

- *Audio system*
  Which will *play* the generated content, as in the case of the generation of an audio file.
- *Sequencer software*
  Which will *process* the generated content, as in the case of the generation of a MIDI file.
- *Human(s)*
  Who will perform and *interpret* the generated content, as in the case of the generation of a music score.

3.1.3 Mode

There are two main modes of music generation:

- *Autonomous* and *Automated*
  Without any human intervention; or
- *Interactive* (to some degree)
  With some control interface for the human user(s) to have some interactive control over the process of generation.

---

[1] Although there are non-standard techniques to produce more than one note, the simplest one being to sing simultaneously as playing. There are also non-standard diphonic techniques for voice.

[2] Figure 4.13 at Chapter 4 Representation will show an example of a lead sheet.

[3] Note that lyrics could be generated too. Although this target is beyond the scope of this book, we will see later in Section 4.7.3 that, in some systems, music is encoded as a text. Thus, a similar technique could be applied to lyric generation.

As deep learning for music generation is recent and basic neural network techniques are non-interactive, the majority of systems that we have analyzed are not yet very interactive[4]. Therefore, an important goal appears to be the design of fully interactive support systems for musicians (for composing, analyzing, harmonizing, arranging, producing, mixing, etc.), as pioneered by the FlowComposer prototype [141] to be introduced in Section 6.10.4.

3.1.4 Style

As stated previously, the musical style of the content to be generated will be governed by the choice of the dataset of musical examples that will be used as training examples. As will be discussed further in Section 4.12, we will see that the choice of a dataset, notably properties like *coherence*, *coverage* (versus *sparsity*) and *scope* (specialized versus large breadth), is actually fundamental for good music generation.

[4] Some examples of interactive systems will be introduced in Section 6.14.

Chapter 4
Representation

A second dimension of our analysis, the *representation*, is about the way the musical content is represented. The choice of representation and its encoding is tightly connected to the configuration of the input and the output of the architecture, i.e. the number of input and output variables as well as their corresponding types.

We will see that, although a deep learning architecture can automatically extract significant *features* from the data, the choice of representation may be significant for the accuracy of the learning and for the quality of the generated content.

For example, in the case of an audio representation, we could use a spectrum representation (computed by a Fourier transform) instead of a raw waveform representation. In the case of a symbolic representation, we could consider (as in most systems) enharmony, i.e. A♯ being equivalent to B♭ and C♭ being equivalent to B, or instead preserve the distinction in order to keep the harmonic and/or voice leading meaning.

4.1 Phases and Types of Data

Before getting into the choices of representation for the various data to be processed by a deep learning architecture, it is important to identify the two main phases for a deep learning architecture: *training* phase and *generation* phase. There are three[1] main types of data to be considered:

- *Training phase*
  - *Training input*
    The dataset of examples used for training the deep learning system;
- *Generation phase*
  - *Generation input*
    The data that will 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[2] of the generation.

Depending on the objective, these three types of data may be equal or different. For instance:

- in the case of the generation of a melody (for example, in Section 6.5.1.2), both the training input and the generated output are melodies; whereas

[1] There may be more types of data depending on the complexity of the architecture, which may include *intermediate* processing steps.

[2] As stated in Section 3.1.1, we identify an objective by its type as a matter of simplification.

- in the case of the generation of a counterpoint accompaniment (for example, in Section 6.1.2), the generated output is a set of melodies.

4.2 Audio versus Symbolic

A big divide in terms of the choice of representation (both for input and output) is *audio* versus *symbolic*. This also corresponds to the divide between *continuous* and *discrete* variables. As we will see, their respective raw material is very different in nature, as are the types of techniques for possible processing and transformation of the initial representation[3]. They in fact correspond to different scientific and technical communities, namely *signal processing* and *knowledge representation*.

However, the actual processing of these two main types of representation by a deep learning architecture is basically the *same*[4]. Therefore, actual audio and symbolic architectures for music generation may be pretty similar. For example, the WaveNet audio generation architecture (to be introduced in Section 6.9.3.2) has been transposed to the MidiNet symbolic music generation architecture (in Section 6.9.3.3). This polymorphism (possibility of multiple representations leading to genericity) is an additional advantage of the deep learning approach.

That said, we will focus in this book on symbolic representations and on deep learning techniques for generation of symbolic music. There are various reasons for this choice:

- the grand majority of the current deep learning systems for music generation are symbolic;
- we believe that the essence of music (as opposed to sound[5]) is in the compositional process, which is exposed via symbolic representations (like musical scores or lead sheets) and is subject to analysis (e.g., harmonic analysis);
- covering the details and variety of techniques for processing and transforming audio representations (e.g., spectrum, cepstrum, MFCC[6], etc.) would necessitate an additional book[7]; and
- as stated previously, independently of considering audio or symbolic music generation, the principles of deep learning architectures as well as the encoding techniques used are actually pretty similar.

4.3 Audio

The first type of representation of musical content is audio *signal*, either in its raw form (waveform) or transformed.

4.3.1 Waveform

The most direct representation is the raw audio signal: the *waveform*. The visualization of a waveform is shown in Figure 4.1 and another one with a finer grain resolution is shown in Figure 4.2. In both figures, the x axis represents time and the y axis represents the amplitude of the signal.

[3] The initial representation may be transformed, through, e.g., data compression or extraction of higher-level representations, in order to improve learning and/or generation.

[4] Indeed, at the level of processing by a deep network architecture, the initial distinction between audio and symbolic representation boils down, as only *numerical* values and operations are considered.

[5] Without minimizing the importance of the orchestration and the production.

[6] Mel-frequency cepstral coefficients.

[7] An example entry point is the recent review by Wyse of audio representations for deep convolutional networks [187].

Fig. 4.1 Example of a waveform

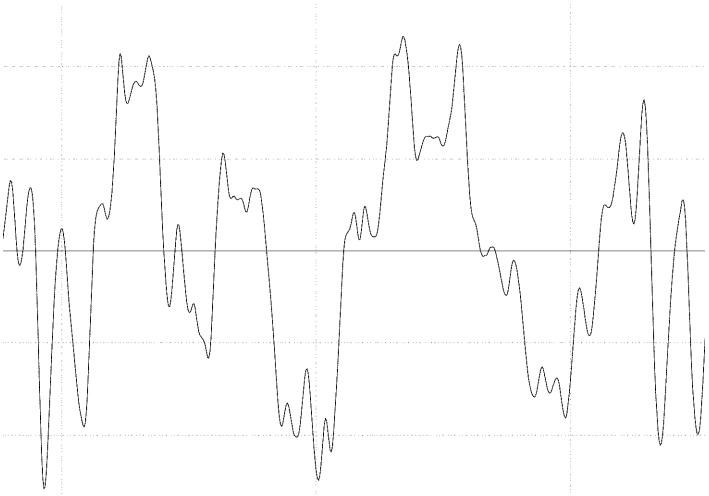

Fig. 4.2 Example of a waveform with a fine grain resolution. Excerpt from a waveform visualization (sound of a guitar) by Michael Jancsy reproduced from "https://plot.ly/~michaeljancsy/205.embed" with the permission of the author

The advantage of using a waveform is in considering the raw material untransformed, with its full initial resolution. Architectures that process the raw signal are sometimes named *end-to-end* architectures[8]. The disadvantage is in the computational load: low level raw signal is demanding in terms of both memory and processing.

4.3.2 Transformed Representations

Using transformed representations of the audio signal usually leads to data compression and higher-level information, but as noted previously, at the cost of losing some information and introducing some bias.

[8] The term *end-to-end* emphasizes that a system learns all features from raw unprocessed data – without any pre-processing, transformation of representation, or extraction of features – to produce the final output.

4.3.3 Spectrogram

A common transformed representation for audio is the *spectrum*, obtained via a *Fourier transform*[9]. Figure 4.3 shows an example of a *spectrogram*, a visual representation of a spectrum, where the *x* axis represents time (in seconds), the *y* axis represents the frequency (in kHz) and the third axis in color represents the intensity of the sound (in dBFS[10]).

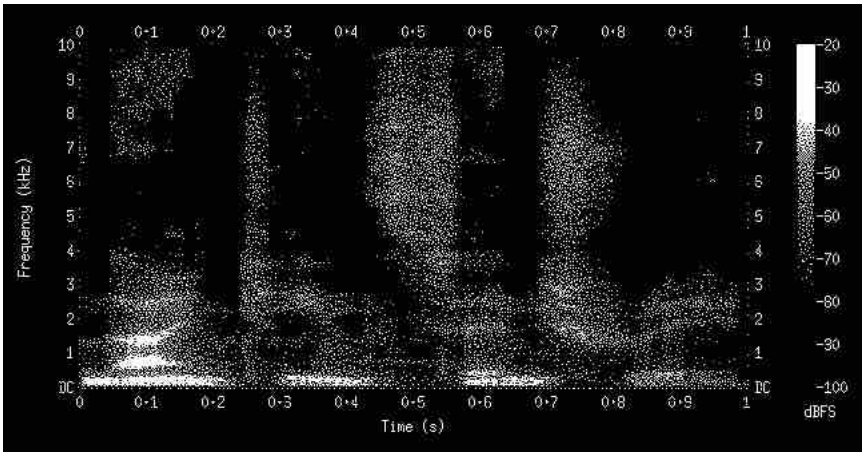

Fig. 4.3 Example of a spectrogram of the spoken words "nineteenth century". Reproduced from Aquegg's original image at "https://en.wikipedia.org/wiki/Spectrogram"

4.3.4 Chromagram

A variation of the spectrogram, discretized onto the tempered scale and independent of the octave, is a *chromagram*. It is restricted to *pitch classes*[11]. The chromagram of the C major scale played on a piano is illustrated in Figure 4.4. The *x* axis common to the four subfigures (a to d) represents time (in seconds). The *y* axis of the score (a) represents the note, the *y* axis of the chromagrams (b and d) represents the chroma (pitch class) and the *y* axis of the signal (c) represents the amplitude. For chromagrams (b and d), the third axis in color represents the intensity.

4.4 Symbolic

Symbolic representations are concerned with concepts like notes, duration and chords, which will be introduced in the following sections.

[9] The objective of the Fourier transform (which could be continuous or discrete) is the decomposition of an arbitrary signal into its elementary components (sinusoidal waveforms). As well as compressing the information, its role is fundamental for musical purposes as it reveals the *harmonic* components of the signal.

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

[11] A *pitch class* represents the name of the corresponding note independently of the octave position. Possible pitch classes are C, C♯ (or D♭), D, … A♯ (or B♭) and B. They are also named *chromas*.

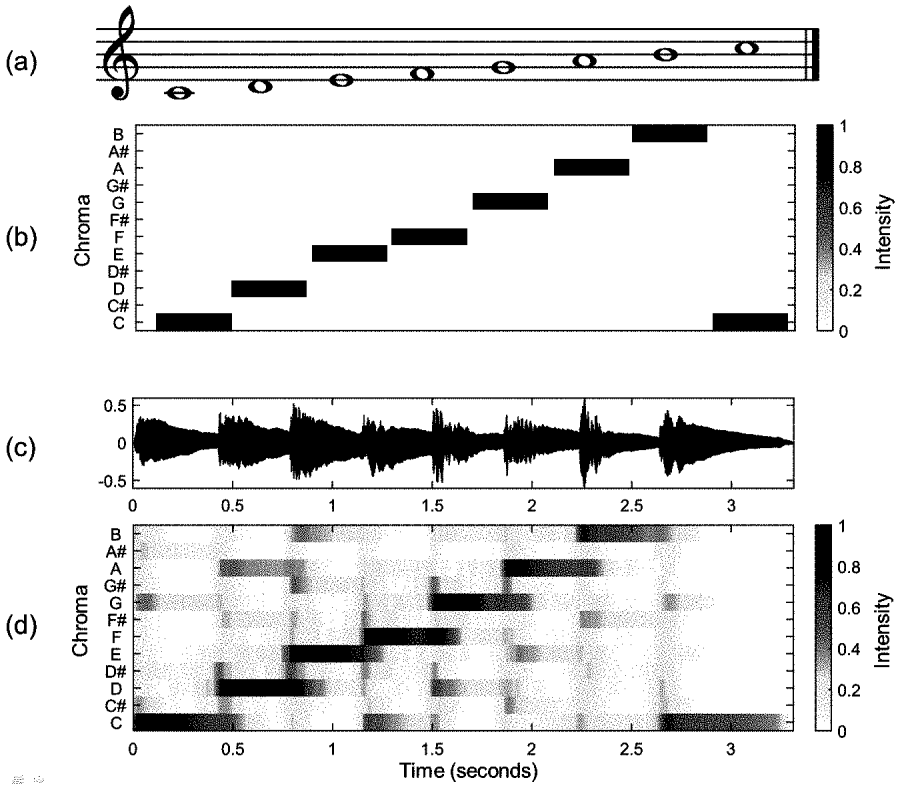

Fig. 4.4 Example of a chromagram. (a) Musical score of a C-major scale. (b) Chromagram obtained from the score. (c) Audio recording of the C-major scale played on a piano. (d) Chromagram obtained from the audio recording. Reproduced from Meinard Mueller's original image at "https://en.wikipedia.org/wiki/Chroma_feature" under a CC BY-SA 3.0 licence

4.5 Main Concepts

4.5.1 Note

In a symbolic representation, a note is represented through the following main features, and for each feature there are alternative ways of specifying its value:

- *Pitch* – specified by
  - frequency, in Hertz (Hz);
  - vertical position (height) on a score; or
  - *pitch notation*[12], which combines a musical note name, e.g., A, A♯, B, etc. – actually its pitch class – and a number (usually notated in subscript) identifying the pitch class octave which belongs to the $[-1,9]$ discrete interval. An example is A_4, which corresponds to A440 – with a frequency of 440 Hz – and serves as a general pitch tuning standard.
- *Duration* – specified by
  - *absolute* value, in milliseconds (ms); or

[12] Also named international pitch notation or scientific pitch notation.

  - *relative* value, notated as a division or a multiple of a reference note duration, i.e. the full note 𝅝. Examples are a quarter[13] 𝅘𝅥 and an eighth note 𝅘𝅥𝅮.

- *Dynamics* – specified by
  - *absolute* and *quantitative* value, in decibels (dB); or
  - *qualitative* value, an annotation on a score about how to perform the note, which belongs to the discrete set {***ppp***, ***pp***, ***p***, ***f***, ***ff***, ***fff***}, from pianissimo to fortissimo.

4.5.2 Rest

Rests are important in music as they represent intervals of silence allowing a pause for breath[14]. A rest can be considered as a special case of a note, with only one feature, its duration, and no pitch or dynamics. The duration of a rest may be specified by

- *absolute* value, in milliseconds (ms); or
- *relative* value, notated as a division or a multiple of a reference rest duration, the full rest 𝄻 having the same duration as a full note 𝅝. Examples are a quarter rest 𝄽 and an eighth rest 𝄾, corresponding respectively to a quarter 𝅘𝅥 and an eighth note 𝅘𝅥𝅮.

4.5.3 Chord

A representation of a *chord* could be

- *implicit* and *extensional*, enumerating the exact notes composing it. This permits the specification of the precise octave as well as the position (voicing) for each note, see an example in Figure 4.5; or
- *explicit* and *intensional*, by using a chord symbol combining
  - the pitch class of its root note, e.g., C, and
  - the *type*, e.g., major, minor, dominant seventh, or diminished[15].

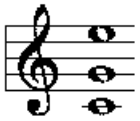

Fig. 4.5 C major chord with an open position/voicing: 1-5-3 (root, 5th and 3rd)

We will see that the extensional approach (explicitly listing all component notes) is more common for deep learning-based music generation systems, but there are some examples of systems representing chords explicitly with the intensional approach, as for instance the MidiNet system to be introduced in Section 6.9.3.3.

[13] Named a *quaver* in British English.

[14] As much for appreciation of the music as for respiration by human performer(s)!

[15] There are some abbreviated notations, frequent in Jazz and popular music, for instance C minor = Cmin = Cm = C-; C major seventh = CM7 = Cmaj7 = CΔ (and many more types of chords in modern music).

4.5.4 Rhythm

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

4.5.4.1 Beat and Meter

A *beat* is the unit of pulsation in music. Beats are grouped into measures, separated by *bars*[16]. The number of beats in a measure as well as the duration between two successive beats constitute the rhythmic signature of a measure and consequently of a piece of music[17]. This *time signature* is also often named *meter*. It is expressed as the fraction $numberOfBeats/BeatDuration$, where

- $numberOfBeats$ is the number of beats within a measure; and
- $beatDuration$ is the duration between two beats. As with the relative duration of a note (see Section 4.5.1) or of a rest, it is expressed as a division of the duration of a full note 𝅝.

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

4.5.4.2 Levels of Rhythm Information

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

- *None* – only notes and their durations are represented, without any explicit representation of measures. This is the case for most systems.
- *Measures* – measures are explicitly represented. An example is the system described in Section 6.5.1.2[18].
- *Beats* – information about meter, beats, etc. is included. An example is the C-RBM system described in Section 6.9.5.1, which allows us to impose a specific meter and beat stress for the music to be generated.

4.6 Multivoice/Multitrack

A *multivoice* representation, also named *multitrack*, considers independent various voices, each being a different vocal range (e.g., soprano, alto…) or a different instrument (e.g., piano, bass, drums…). Multivoice music is usually modeled as parallel tracks, each one with a distinct sequence of notes[19], sharing the same meter but possibly with different strong (stressed) beats[20].

Note that in some cases, although there are simultaneous notes, the representation will be a single-voice polyphony, as introduced in Section 3.1.1. Common examples are polyphonic instruments like a piano or a guitar. Another example

[16] Although (and because) a bar is actually the *graphical entity* – the line segment "|" – separating measures, the term bar is also often used, specially in the United States, in place of measure. In this book we will stick to the term *measure*.

[17] For more elaborate music, the meter may change within different portions of the music.

[18] It is interesting to note that, as pointed out by Sturm *et al.* in [163], the generated music format also contains bars separating measures and that there is no guarantee that the number of notes in a measure will always fit to a measure. However, errors rarely occur, indicating that this representation is sufficient for the architecture to learn to count, see [55] and Section 6.5.1.2.

[19] With possibly simultaneous notes for a given voice, see Section 3.1.1.

[20] Dance music is good at this, by having some syncopated bass and/or guitar not aligned on the strong drum beats, in order to create some bouncing pulse.

is a drum or percussion kit, where each of the various components, e.g., snare, hi-hat, ride cymbal, kick, etc., will usually be considered as a distinct note for the same voice.

The different ways to encode single-voice polyphony and multivoice polyphony will be further discussed in Section 4.11.2.

4.7 Format

The format is the language (i.e. grammar and syntax) in which a piece of music is expressed (specified) in order to be interpreted by a computer[21].

4.7.1 MIDI

Musical Instrument Digital Interface (MIDI) is a technical standard that describes a protocol, a digital interface and connectors for interoperability between various electronic musical instruments, softwares and devices [121]. MIDI carries event messages that specify real-time note performance data as well as control data. We only consider here the two most important messages for our concerns:

- *Note on* – to indicate that a note is played. It contains
  - a *channel number*, which indicates the instrument or track, specified by an integer within the set $\{0, 1, \ldots, 15\}$;
  - a MIDI *note number*, which indicates the note *pitch*, specified by an integer within the set $\{0, 1, \ldots, 127\}$; and
  - a *velocity*, which indicates how loud the note is played[22], specified by an integer within the set $\{0, 1, \ldots, 127\}$.

  An example is "Note on, 0, 60, 50" which means "On channel 1, start playing a middle C with velocity 50";
- *Note off* – to indicate that a note ends. In this situation, the velocity indicates how fast the note is released. An example is "Note off, 0, 60, 20" which means "On channel 1, stop playing a middle C with velocity 20".

Each note event is actually embedded into a track chunk, a data structure containing a delta-time value which specifies the timing information and the event itself. A *delta-time value* represents the time position of the event and could represent

- a *relative metrical* time – the number of *ticks* from the beginning. A reference, named the *division* and defined in the file header, specifies the number of ticks per quarter note ♩; or
- an *absolute* time – useful for real performances, not detailed here, see [121].

An example of an excerpt from a MIDI file (turned into readable ascii) and its corresponding score are shown in Figures 4.6 and 4.7. The division has been set to 384, i.e. 384 ticks per quarter note ♩ (which corresponds to 96 ticks for a sixteenth note 𝅘𝅥𝅯).

In [81], 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. In their experiment, they concatenate tracks end-to-end and thus 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. Piano roll, to be introduced in next section, does not have this limitation but at the cost of another limitation.

[21] The standard format for humans is a musical score.

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

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

Fig. 4.6 Excerpt from a MIDI file

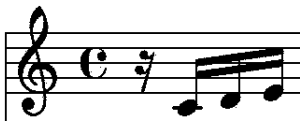

Fig. 4.7 Score corresponding to the MIDI excerpt

4.7.2 Piano Roll

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

Fig. 4.8 Automated piano and piano roll. Reproduced from Yaledmot's post "https://www.youtube.com/watch?v=QrcwR7eijyc" with the permission of YouTube

An example of a modern piano roll representation (for digital music systems) is shown in Figure 4.9. The x axis represents time and the y axis the pitch.

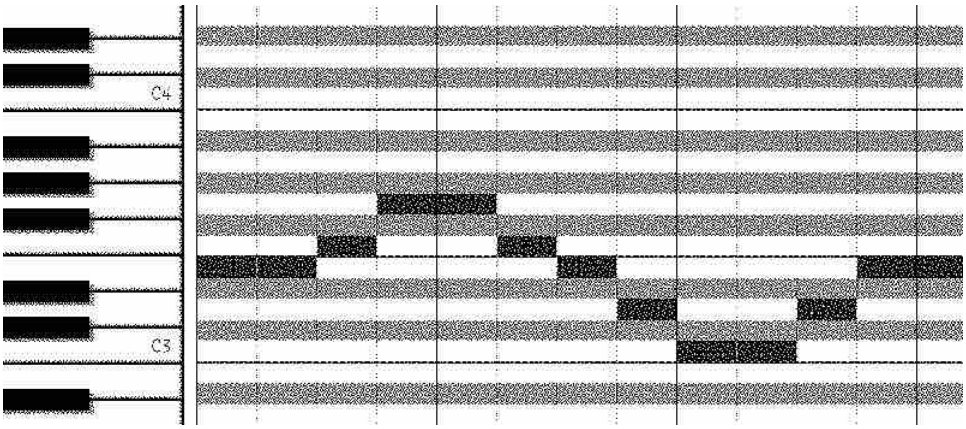

Fig. 4.9 Example of symbolic piano roll. Reproduced from [66] with the permission of Hao Staff Music Publishing (Hong Kong) Co Ltd.

There are several music environments using piano roll as a basic visual representation, in place of or in *complement* to a score, as it is more intuitive than the traditional score notation[23]. An example is Hao Staff piano roll sheet music [66], shown in Figure 4.9 with the time axis being horizontal rightward and notes represented as green cells. Another example is tabs, where the melody is represented in a piano roll-like format [79], in complement to chords and lyrics. Tabs are used as an input by the MidiNet system, to be introduced in Section 6.9.3.3.

The piano roll is one of the most commonly used representations, although it has some limitations. An important one, compared to MIDI representation, is that there is no note off information. As a result, there is no way to distinguish between a long note and a repeated short note[24]. In Section 4.9.1, we will look at different ways to address this limitation. For a more detailed comparison between MIDI and piano roll, see [81] and [180].

4.7.3 Text

4.7.3.1 Melody

A melody can be encoded in a textual representation and processed as a *text*. A significant example is the ABC notation [182], a *de facto* standard for folk and traditional music[25]. Figures 4.10 and 4.11 show the original score and its associated ABC notation for a tune named "A Cup of Tea", from the repository and discussion platform The Session [91].

The first six lines are the header and represent *metadata*: T is the title of the music, M is the meter, L is the default note length, K is the key, etc. The header is followed by the main text representing the melody. Some basic principles of the encoding rules of the ABC notation are as follows[26]:

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

[23] Another notation specific to guitar or string instruments is a *tablature*, in which the six lines represent the chords of a guitar (four lines for a bass) and the note is specified by the number of the fret used to obtain it.

[24] Actually, in the original 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.

[25] Note that the ABC notation has been designed *independently* of computer music and machine learning concerns.

[26] Please refer to [182] for more details.

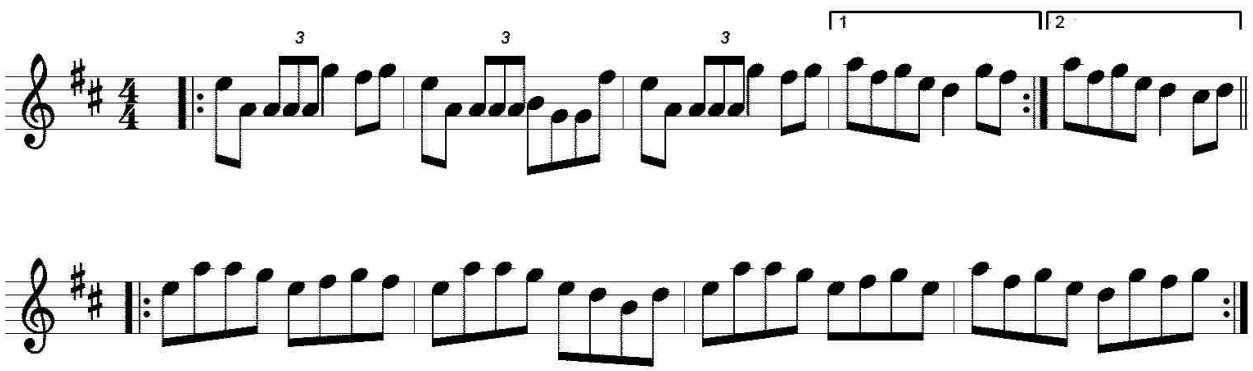

Fig. 4.10 Score of "A Cup of Tea" (Traditional). Reproduced from The Session [91] with permission of the manager

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

Fig. 4.11 ABC notation of "A Cup of Tea". Reproduced from The Session [91] with permission of the manager

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

Note that the ABC notation can only represent monophonic melodies[27].

In order to be processed by a deep learning architecture, the ABC text is usually transformed from a character vocabulary text into a *token* vocabulary text in order to properly consider concepts which could be noted on more than one character, e.g., `g2`. Sturm *et al.*'s experiment, described in Section 6.5.1.2, uses a token-based notation named the folk-rnn notation [163]. A tune is enclosed within a "`<s>`" begin mark and an "`<\s>`" end mark. Last, all example melodies are transposed to the same C root base, resulting in the notation of the tune "A Cup of Tea" shown in Figure 4.12.

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

Fig. 4.12 Folk-rnn notation of "A Cup of Tea". Reproduced from [163] with the permission of the authors

[27] Note that rests may be expressed in the ABC notation through the `z` letter. Their durations are expressed as for notes, e.g., `z2` is a double length rest.

4.7.3.2 Chord and Polyphony

When represented extensionally, chords are usually encoded with simultaneous notes as a vector. An interesting alternative extensional representation of chords, named Chord2Vec[28], has recently been proposed in [112][29]. Rather than thinking of chords (vertically) as vectors, it represents chords (horizontally) as sequences of constituent notes. More precisely,

- a chord is represented as an arbitrary length-ordered sequence of notes; and
- chords are separated by a special symbol, as with sentence markers in natural language processing.

When using this representation for predicting neighboring chords, a specific compound architecture is used, named RNN Encoder-Decoder which will be described in Section 6.9.2.3.

Note that a somewhat similar model is also used for polyphonic music generation by the BachBot system [110] which will be introduced in Section 6.16.1. In this model, for each time step, the various notes (ordered in a descending pitch) are represented as a sequence and a special delimiter symbol "| | |" indicates the next time frame.

4.7.4 Markup Language

Let us mention the case of general text-based structured representations based on markup languages (famous examples are HTML and XML). Some markup languages have been designed for music applications, like for instance the open standard MusicXML [57]. The motivation is to provide a common format to facilitate the sharing, exchange and storage of scores by musical software systems (such as score editors and sequencers). MusicXML, as well as similar languages, is not intended for direct use by humans because of its verbosity, which is the down side of its richness and effectiveness as an interchange language. Furthermore, it is not very appropriate as a direct representation for machine learning tasks for the same reasons, as its verbosity and richness would create too much overhead as well as bias.

4.7.5 Lead Sheet

Lead sheets are an important representation format for popular music (Jazz, Pop, etc.). A *lead sheet* conveys in upto a few pages the score of a melody and its corresponding chords via an intensional notation[30]. Lyrics may also be added. Some important information for the performer, such as the composer, author, style and tempo, is often also present. An example of lead sheet in shown in Figure 4.13.

Paradoxically, few systems and experiments use this rich and concise representation, and most of the time they focus on the notes. Note that Eck and Schmidhuber's Blues generation system, to be introduced in Section 6.4.1.1, 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 [42], resulting in the Lead Sheet Data Base (LSDB) repository [135], which includes more than 12,000 lead sheets.

Note that there are some alternative notations, notably tabs [79], where the melody is represented in a piano roll-like format (see Section 4.7.2) and complemented with the corresponding chords. An example of use of tabs is the MidiNet system to be analyzed in Section 6.9.3.3.

[28] Chord2Vec is inspired by the Word2Vec model for natural language [118].

[29] For information, there is another similar model, also named Chord2Vec, proposed in [82].

[30] See Section 4.5.3.

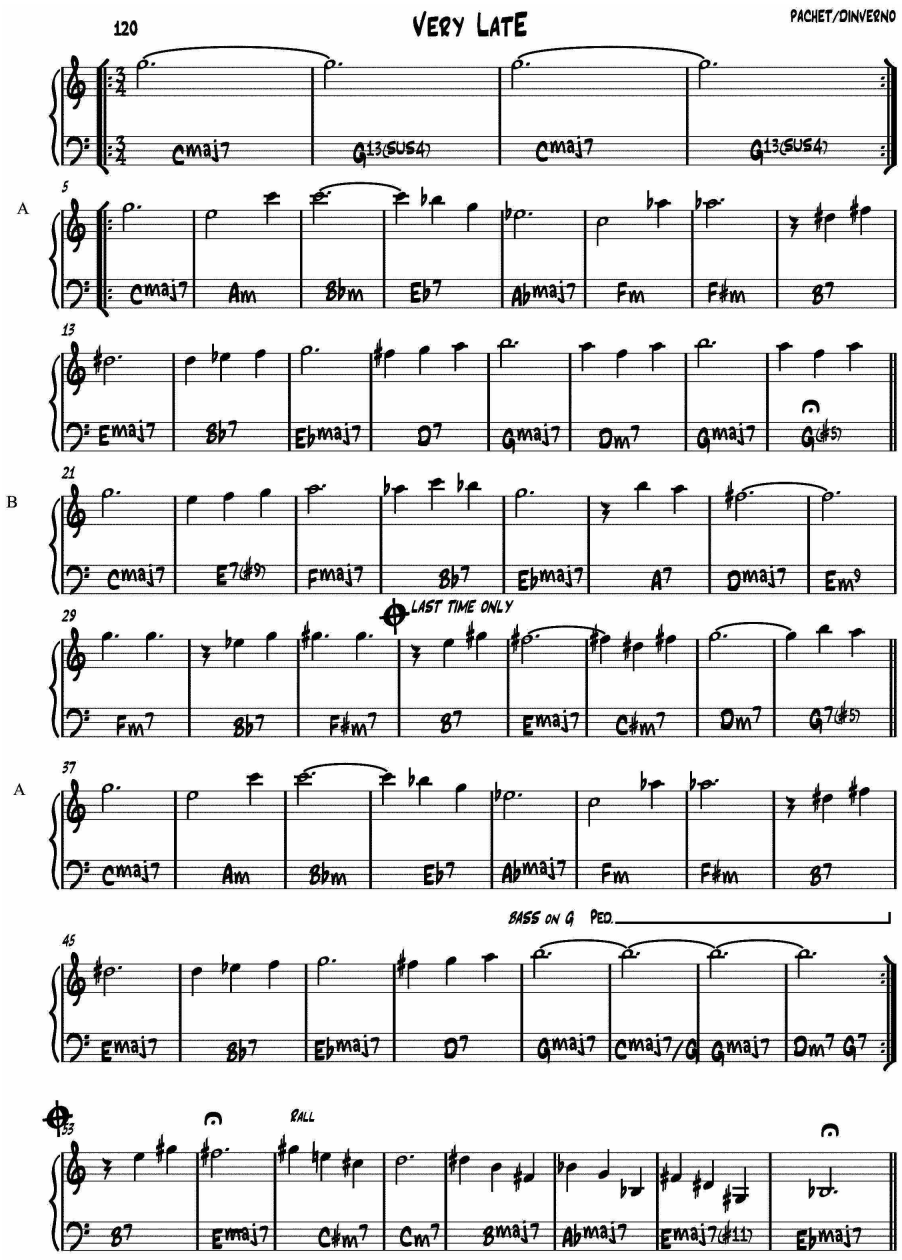

Fig. 4.13 Lead sheet of "Very Late" (Pachet and d'Inverno). Reproduced with the permission of the composers

4.8 Temporal Scope and Granularity

The representation of time is fundamental for musical processes.

4.8.1 Temporal Scope

An initial design decision concerns the *temporal scope* of the representation used for the generation input and for the generated output, that is the way the representation will be interpreted by the architecture with respect to time, as illustrated in Figure 4.14:

- *Global* – in this first case, the temporal scope of the representation is the *whole* musical piece. The deep network architecture (typically a feedforward or an autoencoder architecture, see Sections 5.8 and 5.9) will process the input and produce the output within a *single step*[31]. Examples are the MiniBach and DeepHear systems introduced in Sections 6.1.2 and 6.3.1.1, respectively.
- *Time step* (or *time slice*) – in this second case, the most frequent one, the temporal scope of the representation is a *local time slice* of the musical piece, corresponding to a specific temporal moment (time step). The granularity of the processing by the deep network architecture (typically a recurrent network) is a *time step* and generation is iterative[32]. Note that the time step is usually set to the *shortest note duration* (see more details in Section 4.8.2), but it may be larger, e.g., set to a measure in the system as discussed in [172].
- *Note step* – this third case was proposed by Mozer in [127] in his CONCERT system [127], see Section 6.5.1.1. In this approach there is *no fixed time step*. The granularity of processing by the deep network architecture is a *note*. This strategy uses a distributed encoding of duration that allows to process a note of any duration in a single network processing step. Note that, by considering as a single processing step a note rather than a time step, the number of processing steps to be bridged by the network is greatly reduced. The approach proposed later on by Walder in [180] is similar.

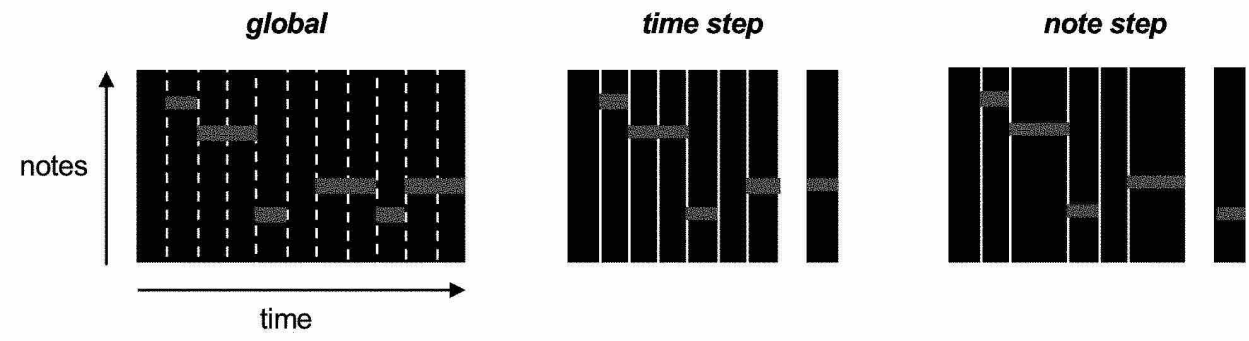

Fig. 4.14 Temporal scope for a piano roll-like representation

Note that a global temporal scope representation actually also considers time steps (separated by dash lines in Figure 4.14). However, although time steps are present at the representation level, they will not be interpreted as distinct processing steps by the neural network.

Also note that in the case of a global temporal scope representation the musical content generated has a *fixed length* (the number of time steps), whereas in the case of a time step or a note step temporal scope the musical content generated has an *arbitrary length*, because generation is iterative as we will see in Section 6.4.1.

[31] In Chapter 6, we will name it the *single-step feedforward strategy*, see Section 6.1.1.

[32] In Chapter 6, we will name it the *iterative feedforward strategy*, see Section 6.4.1.

4.8.2 Temporal Granularity

In the case of a global or a time step temporal scope, the granularity of the time step, corresponding to the granularity of the time *discretization*, must be defined. There are two main strategies:

- The most common strategy is to set the time step to a *relative duration*, the smallest duration of a note in the corpus (training examples/dataset), e.g., a sixteenth note 𝅘𝅥𝅯. To be more precise, as stated by Todd in [172], the time step should be the *greatest common factor* of the durations of all the notes to be learned. This ensures that the duration of every note will be properly represented with a whole number of time steps. One immediate consequence of this "leveling down" is the number of processing steps necessary, independent of the duration of actual notes.
- Another strategy is to set the time step to a fixed *absolute duration*, e.g., 10 milliseconds. This strategy permits us to capture expressiveness in the timing of each note during a human performance, as we will see in Section 4.10.

Note that in the case of a note step temporal scope, there is no uniform discretization of time (no fixed time step) and no need for.

4.9 Metadata

In some systems, additional information from the score may also be explicitly represented and used as *metadata*, such as

- note tie,
- fermata,
- harmonics,
- key,
- meter, and
- the instrument associated to a voice.

This extra information may lead to more accurate learning and generation.

4.9.1 Note Hold/Ending

An important issue is how to represent if a note is held, i.e. tied to the previous note[33]. This is actually equivalent to the issue of how to represent the ending of a note.

In the MIDI representation format, the end of a note is explicitly stated (via a "Note off" event[34]). In the piano roll notation discussed in Section 4.7.2, there is no explicit representation of the ending of a note and, as a result, one cannot distinguish between two repeated quarter notes 𝅘𝅥𝅘𝅥 and a half note 𝅗𝅥.

The main possible techniques are

- to introduce a *hold/replay* representation, as a dual representation of the sequence of notes. This solution is used, for example, by Mao *et al.* in their DeepJ system [116] (to be analyzed in Section 6.9.3.4), by introducing a replay matrix similar to the piano roll-type matrix of notes;
- to divide the size of the time step[35] by two and always mark a *note ending* with a special tag, e.g., 0. This solution is used, for example, by Eck and Schmidhüber in [37], and will be analyzed in Section 6.4.1.1;

[33] A tied note on a music score specifies how a note duration extends across a single measure. In our case, the issue is how to specify that the duration extends across a single *time step*. Therefore, we consider it as metadata information, as it is specific to the representation and its processing by a neural network architecture.

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

[35] See Section 4.8.2 for details of how the value of the time step is defined.

- to divide the size of the time step as before but instead mark a *new note beginning*. This solution is used by Todd in [172]; or
- to use a special *hold* symbol "__" in place of a note to specify when the previous note is held. This solution was proposed by Hadjeres *et al.* in their DeepBach system [65] to be analyzed in Section 6.13.2.

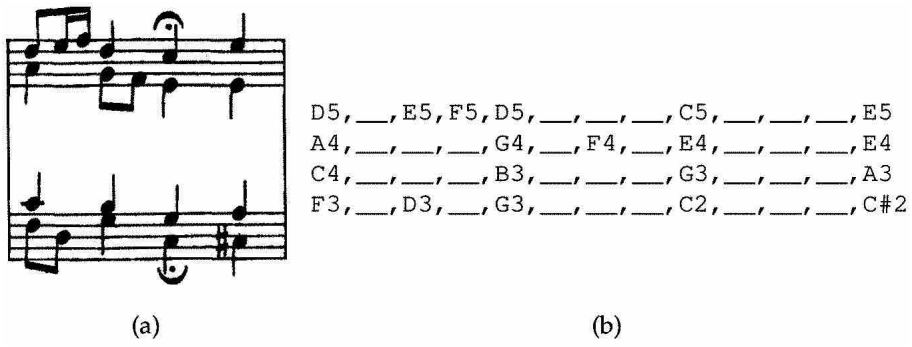

Fig. 4.15 a) Extract from a J. S. Bach chorale and b) its representation using the hold symbol "__". Reproduced from [65] with the permission of the authors

This last solution considers the hold symbol as a note, see an example in Figure 4.15. The advantages of the hold symbol technique are

- it is simple and uniform as the hold symbol is considered as a note; and
- there is no need to divide the value of the time step by two and mark a note ending or beginning.

The authors of DeepBach also emphasize that the good results they obtain using Gibbs sampling rely exclusively on their choice to integrate the hold symbol into the list of notes (see [65] and Section 6.13.2). An important limitation is that the hold symbol only applies to the case of a monophonic melody, that is it cannot directly express held notes in an unambiguous way in the case of a single-voice polyphony. In this case, the single-voice polyphony must be reformulated into a multivoice representation with each voice being a monophonic melody; then a hold symbol is added separately for each voice. Note that in the case of the replay matrix, the additional information (matrix row) is for each possible note and not for each voice.

We will discuss in Section 4.11.7 how to encode a hold symbol.

4.9.2 Note Denotation (versus Enharmony)

Most systems consider *enharmony*, i.e. in the tempered system $A\sharp$ is *enharmonically equivalent* to (i.e. has the same pitch as) $B\flat$, although harmonically and in the composer's intention they are different. An exception is the DeepBach system, described in Section 6.13.2, which encodes notes using their real names and not their MIDI note numbers. The authors of DeepBach state that this additional information leads to a more accurate model and better results [65].

4.9.3 Feature Extraction

Although deep learning is good at processing raw unstructured data, from which its hierarchy of layers will extract higher-level representations adapted to the task (see Section 1.2.4), some systems include a preliminary step of automatic *feature extraction*, in order to represent the data in a more compact, characteristic and discriminative form. One

motivation could be to gain efficiency and accuracy for the training and for the generation. Moreover, this *feature-based representation* is also useful for indexing data, in order to control generation through compact labeling (see, for example, the DeepHear system in Section 6.3.1.1), or for indexing musical units to be queried and concatenated (see Section 6.9.7.1).

The set of *features* can be defined *manually* (*handcrafted*) or *automatically* (e.g. by an autoencoder, see Section 5.9). In the case of handcrafted features, the bag-of-words (BOW) model is a common strategy for natural language text processing, which may also be applied to other types of data, including musical data, as we will see in Section 6.9.7.1. It consists in transforming the original text (or arbitrary representation) into a "bag of words" (the vocabulary composed of all occurring words, or more generally speaking, all possible tokens); then various measures can be used to characterize the text. The most common is *term frequency*, i.e. the number of times a term appears in the text[36].

Sophisticated methods have been designed for neural network architectures to automatically compute a vector representation which preserves, as much as possible, the relations between the items. Vector representations of texts are named *word embeddings*[37]. A recent reference model for natural language is the Word2Vec model [118]. It has recently been transposed to the Chord2Vec model for the vector encoding of chords, as described in [112] (see Section 4.5.3).

4.10 Expressiveness

4.10.1 Timing

If training input examples are processed from conventional scores or MIDI-format libraries, there is a good chance that the music is perfectly *quantized* – i.e., note onsets[38] are exactly aligned onto the tempo – resulting in a mechanical sound without *expressiveness*. One approach is to consider symbolic records – in most cases recorded directly in MIDI – from real human *performances*, with the musician interpreting the tempo. An example of a system for this purpose is Performance RNN [159], which will be analyzed in Section 6.6.1. It follows the *absolute time duration* quantization strategy, presented in Section 4.8.2.

4.10.2 Dynamics

Another common limitation is that many MIDI-format libraries do not include *dynamics* (the volume of the sound produced by an instrument), which stays fixed throughout the whole piece. One option is to take into consideration (if present on the score) the annotations made by the composer about the dynamics, from pianissimo ***ppp*** to fortissimo ***fff***, see Section 4.5.1. As for tempo expressiveness, addressed in Section 4.10.1, another option is to use real human performances, recorded with explicit dynamics variation – the velocity field in MIDI.

4.10.3 Audio

Note that in the case of an audio representation, expressiveness as well as tempo and dynamics are entangled within the whole representation. Although it is easy to control the global dynamics (global volume), it is almost impossible

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

[37] The term *embedding* comes from the analogy with *mathematical embedding*, which is an injective and structure-preserving mapping. Initially used for natural language processing, it is now often used in deep learning as a general term for *encoding* a given representation into a vector representation. Note that the term embedding, which is an abstract model representation, is often also used (we think, abusively) to define a specific instance of an embedding (which may be better named, for example, a *label*, see [164] and Section 6.3.1.1).

[38] An onset refers to the beginning of a musical note (or sound).

to separately control the dynamics of a single instrument or voice[39]. It is also possible to slightly adjust the tempo, for instance by using audio time stretching techniques, but it is also almost impossible to separately control the tempo of a single instrument or voice.

4.11 Encoding

Once the format of a representation has been chosen, the issue still remains of how to *encode* this representation (composed of a set of variables, e.g., pitch or dynamics) into a set of *inputs* (also named *input nodes* or *input variables*) for the neural network architecture[40].

4.11.1 Strategies

At first, let us consider the three possible types for a variable:

- *Continuous* variables – an example is the pitch of a note defined by its frequency in Hertz, that is a real value within the $]0, +\infty[$ interval[41].
  The straightforward way is to directly encode the variable[42] as a *scalar* whose domain is real values. We call this strategy *value encoding*.
- *Discrete integer* variables – an example is the pitch of a note defined by its MIDI note number, that is an integer value within the $\{0, 1, \ldots, 127\}$ discrete set[43].
  The straightforward way is to encode the variable as a real value *scalar*, by casting the integer into a real. This is another case of *value encoding*.
- *Categorical* variables[44] – an example is a component of a drum kit; an element within a set of possible values: {snare, high-hat, kick, middle-tom, ride-cymbal, etc.}.
  The usual strategy is to encode a categorical variable as a *vector* having as its length the number of possible elements, in other words the cardinality of the set of possible values. Then, in order to represent a given element, the corresponding element of the encoding vector is set to 1 and all other elements to 0. Therefore, this encoding strategy is usually called *one-hot encoding*[45]. This frequently used strategy is also often employed for encoding discrete integer variables, such as MIDI note numbers.

4.11.2 From One-Hot to Many-Hot and to Multi-One-Hot

Note that a one-hot encoding of a note corresponds to a piano roll representation, with as many lines (input nodes) as there are possible pitches. Note also that while a one-hot encoding of a piano roll representation of a *monophonic*

[39] More generally speaking, audio source separation, sometimes coined as the *cocktail party effect*, is known to be a very difficult problem, see the original article in [17] and a more recent survey in [68].

[40] See Section 5.7 for more details about the input nodes of a neural network architecture.

[41] The notation $]0, +\infty[$ is for an open interval excluding its endpoints. An alternative notation is $(0, +\infty)$.

[42] In practice, the different variables are also usually scaled and normalized, in order to have similar domains of values ($[0, 1]$ or $[-1, +1]$) for all input variables, in order to ease learning convergence.

[43] See our summary of MIDI specification in Section 4.7.1.

[44] In statistics, a *categorical variable* is a variable that can take one of a limited – and usually fixed – number of possible values. In computer science it is usually referred as an *enumerated type*.

[45] 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).

melody (with one note at a time) is straightforward, a one-hot encoding of a *polyphony* (with simultaneous notes, as for a guitar playing a chord) is not. One could then consider

- *many-hot encoding* – where all elements of the vector corresponding to the notes or to the active components are set to 1;
- *multi-one-hot encoding* – where different voices or tracks are considered (for multivoice representation, see Section 4.6) and a one-hot encoding is used for each different voice/track; or
- *multi-many-hot encoding* – which is multivoice representation with simultaneous notes for at least one or all of the voices.

4.11.3 Summary

The various approaches for encoding are illustrated in Figure 4.16, showing from left to right

- a scalar continuous value encoding of A_4 (A440), the real number specifiying its frequency in Hertz;
- a scalar discrete integer value encoding of A_4 (A440), the integer number specifiying its MIDI note number;
- a one-hot encoding of A_4 (A440);
- a many-hot encoding of a D minor chord (D_4, F_4, A_4);
- a multi-one-hot encoding of a first voice with A_4 and a second voice with D_3; and
- a multi-many-hot encoding of a first voice with a D minor chord (D_4, F_4, A_4) and a second voice with C_3 (corresponding to a minor seventh on bass).

4.11.4 Binning

In some cases, a continuous variable is transformed into a discrete domain. A common technique, named *binning*, or also *bucketing*, consists of

- dividing the original domain of values into smaller intervals[46], named *bins*; and
- replacing each bin (and the values within it) by a *value representative*, often the central value.

Note that this binning technique may also be used to reduce the cardinality of the discrete domain of a variable. An example is the Performance RNN system described in Section 6.6.1, for which the initial MIDI set of 127 values for note dynamics is reduced into 32 bins.

4.11.5 Pros and Cons

In general, value encoding is rarely used except for audio, whereas one-hot encoding is the most common strategy for symbolic representation[47].

A counterexample is the case of the DeepJ symbolic generation system described in Section 6.9.3.4, which is, in part, inspired by the WaveNet audio generation system. DeepJ's authors state that: "We keep track of the dynamics of every note in an N x T dynamics matrix that, for each time step, stores values of each note's dynamics scaled between 0 and 1, where 1 denotes the loudest possible volume. In our preliminary work, we also tried an alternate representation

[46] This can be automated through a learning process, e.g., by automatic construction of a decision tree.

[47] Let us remind that, at the level of processing by a deep network, the distinction between audio and symbolic representation boils down to nothing, as only numerical values and operations are considered. In fact the general principles of a deep learning architecture are independent of that distinction and this is one of the vectors of the generality of the approach.

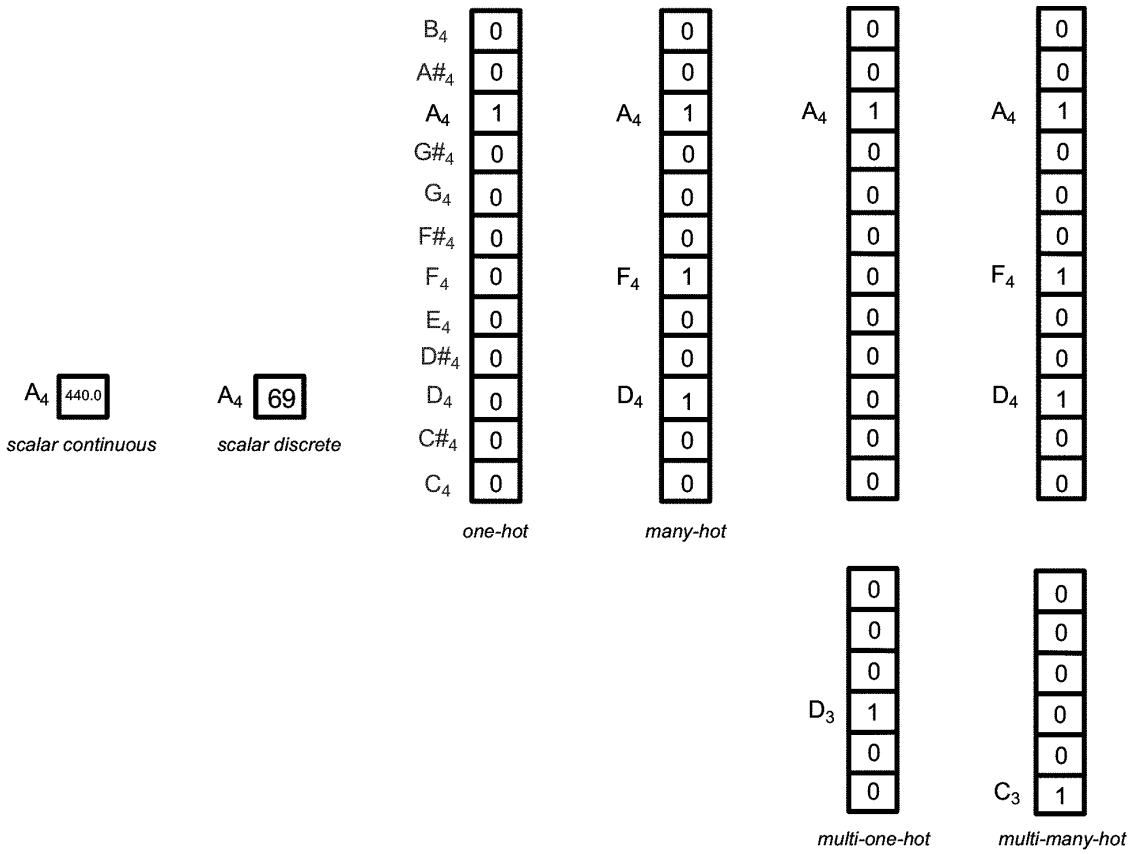

Fig. 4.16 Various types of encoding

of dynamics as a categorical value with 128 bins as suggested by Wavenet [176]. Instead of predicting a scalar value, our model would learn a multinomial distribution of note dynamics. We would then randomly sample dynamics during generation from this multinomial distribution. Contrary to Wavenet's results, our experiments concluded that the scalar representation yielded results that were more harmonious." [116].

The advantage of value encoding is its compact representation, at the cost of sensibility because of numerical operations (approximations). The advantage of one-hot encoding is its robustness (discrete versus analog), at the cost of a high cardinality and therefore a potentially large number of inputs.

It is also important to understand that the choice of one-hot encoding at the *output* of the network architecture is often (albeit not always) associated to a *softmax* function in order to compute the probabilities of each possible value, for instance the probability of a note being an A, or an A♯, a B, a C, etc. This actually corresponds to a *classification task* between the possible values of the categorical variable. This will be further analyzed in Section 5.8.3.

4.11.6 Chords

Two methods of encoding chords, corresponding to the two main alternative representations discussed in Section 4.5.3, are

- *implicit* and *extensional* – enumerating the exact notes composing the chord. The natural encoding strategy is many-hot. An example is the RBM-based polyphonic music generation system described in Section 6.3.2.3; and

- *explicit* and *intensional* – using a chord symbol combining a pitch class and a type (e.g., D minor). The natural encoding strategy is multi-one-hot, with an initial one-hot encoding of the pitch class and a second one-hot encoding of the class type (major, minor, dominant seventh, etc.). An example is the MidiNet system[48] described in Section 6.9.3.3.

4.11.7 Special Hold and Rest Symbols

We have to consider the case of special symbols for hold ("hold previous note", see Section 4.9.1) and rest ("no note", see Section 4.5.2) and how they relate to the encoding of actual notes.

First, note that there are some rare cases where the rest is actually *implicit*:

- in MIDI format – when there is no "active" "Note on", that is when they are all currently "closed" by a corresponding "Note off"; and
- in one-hot encoding – when all elements of the vector are equal to 0.

Now, let us consider how to encode hold and rest depending on how a note pitch is encoded:

- *value encoding* – In this case, one needs to add two extra boolean variables (and their corresponding input nodes) *hold* and *rest*. This must be done for each possible independent voice in the case of a polyphony; or
- *one-hot encoding* – In that case (the most frequent and manageable strategy), one just needs to extend the vocabulary of the one-hot encoding with two additional possible values: *hold* and *rest*. They will be considered at the same level, and of the same nature, as possible notes (e.g., A_3 or C_4) for the input as well as for the output.

4.11.8 Drums and Percussion

Some systems explicitly consider drums and/or percussion. A drum or percussion kit is usually modeled as a single-track polyphony by considering distinct simultaneous "notes", each "note" corresponding to a drum or percussion component (e.g., snare, kick, bass tom, hi-hat, ride cymbal, etc.), that is as a many-hot encoding.

An example of a system dedicated to rhythm generation is described in Section 6.9.3.1. It follows the single-track polyphony approach. In this system, each of the five components is represented through a binary value, specifying whether or not there is a related event for current time step. Drum events are represented as a binary word[49] of length 5, where each binary value corresponds to one of the five drum components; for instance, 10010 represents simultaneous playing of the kick (bass drum) and the high-hat, following a many-hot encoding.

Note that this system also includes – as an additional voice/track – a condensed representation of the bass line part and some information representing the meter, see more details in Section 6.9.3.1. The authors [113] argue that this extra explicit information ensures that the network is aware of the beat structure at any given point.

Another example is the MusicVAE system (see Section 6.11.1), where nine different drum/percussion components are considered, which gives 2^9 possible combinations, one-hot encoded[50] in $2^9 = 512$ different tokens.

[48] In MidiNet, the possible chord types are actually reduced to only major and minor. Thus, a boolean variable can be used in place of one-hot encoding.

[49] In this system, encoding is made in text, similar to the format described in Section 4.7.3 and more precisely following the approach proposed in [20].

[50] Note that the rhythm generation system mentioned above (and analyzed in Section 6.9.3.1) uses a many-hot encoding, whereas MusicVAE uses a one-hot encoding.

4.12 Dataset

The choice of a dataset is fundamental for good music generation. At first, a dataset should be of sufficient size (i.e. contain a sufficient number of examples) to guarantee accurate learning[51]. As noted by Hadjeres in [62]: "the tradeoff between the size of a dataset and its coherence is one of the major issues when building deep generative models. If the dataset is very heterogeneous, a good generative model should be able to distinguish the different subcategories and manage to generalize well. On the contrary, if there are only slight differences between subcategories, it is important to know if the "averaged model" can produce musically-interesting results."

4.12.1 Transposition and Alignment

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 accuracy and generalization of the learnt model (see Section 5.8.8). In the musical domain, a natural and easy way is *transposition*, i.e. to transpose all examples in all keys. In addition to artificially augmenting the dataset[52], this provides a key (tonality) invariance of all examples and thus makes the examples more generic. Moreover, this also reduces sparsity in the training data. This transposition technique is, for instance, used in the C-RBM system [100] described in Section 6.9.5.1.

An alternative approach is to transpose (align) all examples into a *single common key*. This has been advocated for the RNN-RBM system [10] to facilitate learning, see Section 6.8.1.

4.12.2 Datasets and Libraries

A practical issue is the availability of datasets for training systems and also for evaluating and comparing systems and approaches. There are some reference datasets in the image domain (e.g., the MNIST[53] dataset about handwritten digits [104]), but none yet in the music domain. However, various datasets or libraries[54] have been made public, with some examples listed below:

- the Classical piano MIDI database [96];
- the JSB Chorales dataset[55] [1];
- the LSDB (Lead Sheet Data Base) repository [135], with more than 12,000 lead sheets (including from all Jazz and bossa nova song books), developed within the Flow Machines project [42];
- the MuseData library, an electronic library of Classical music with more than 800 pieces, from CCARH in Stanford University [71];
- the MusicNet dataset [170], a collection of 330 freely-licensed Classical music recordings together with over 1 million annotated labels (indicating timing and instrumental information);
- the Nottingham database, a collection of 1,200 Folk tunes in the ABC notation [49], each tune consisting of a simple melody on top of chords, in other words an ABC equivalent of a lead sheet;

[51] Neural networks and deep learning architectures need lots of examples to function properly. However, one recent research area is about learning from scarce data.

[52] This is named *dataset augmentation*.

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

[54] The difference between a dataset and a library is that a dataset is almost ready for use to train a neural network architecture, as all examples are encoded within a single file and in the same format, although some extra data processing may be needed in order to adapt the format to the encoding of the representation for the architecture or vice-versa; whereas a library is usually composed of a set of files, one for each example.

[55] Note that this dataset uses a quarter 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 4.9.1.

- the Session [91], a repository and discussion platform for Celtic music in the ABC notation containing more than 15,000 songs;
- the Symbolic Music dataset by Walder [181], a huge set of cleaned and preprocessed MIDI files;
- the TheoryTab database [79], a set of songs represented in a tab format, a combination of a piano roll melody, chords and lyrics, in other words a piano roll equivalent of a lead sheet;
- the Yamaha e-Piano Competition dataset, in which participants MIDI performance records are made available [189].

Chapter 5
Architecture

5.1 History

Deep networks are a natural evolution of neural networks, themselves being an evolution of the Perceptron, proposed by Rosenblatt in 1957 [151]. Historically speaking[1], the Perceptron was criticized by Minsky and Papert in 1969 [119] for its inability to classify *nonlinearly separable domains*[2]. Their criticism also served in favoring an alternative approach of Artificial Intelligence, based on symbolic representations and reasoning.

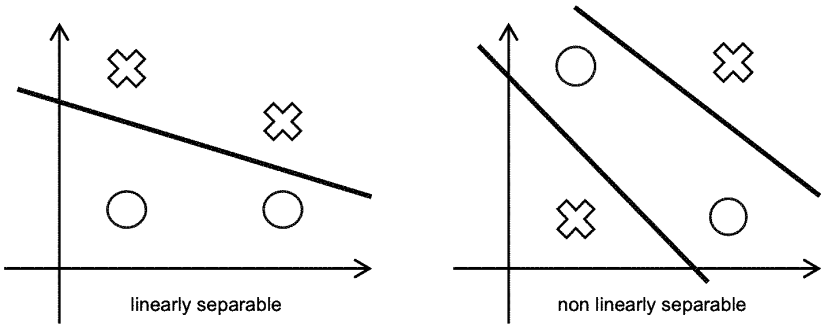

Fig. 5.1 Example and counterexample of linear separability

Neural networks reappeared in the 1980s, thanks to the idea of *hidden layers* joint with nonlinear units, to resolve the initial linear separability limitation, and to the *backpropagation* algorithm, to train such multilayer neural networks [152].

In the 1990s, neural networks suffered declining interest[3] because of the difficulty in training efficiently neural networks with many layers[4] and due to the competition from *support vector machines* (SVM) [179], which were efficiently designed to maximize the *separation margin* and had a solid formal background.

[1] See, for example, [58, Section 1.2] for a more detailed analysis of key trends in the history of deep learning.

[2] A simple example and a counterexample of linear separability (of a set of four points within a 2-dimensional space and belonging to green cross or red circle classes) are shown in Figure 5.1. The elements of the two classes are linearly separable if there is at least one straight line separating them. Note that the discrete version of the counterexample corresponds to the case of the exclusive or (XOR) logical operator, which was used as an argument by Minsky and Papert in [119].

[3] Meanwhile, convolutional networks started to gain interest, notably though handwritten digit recognition applications [103]. As Goodfellow *et al.* in [58, Section 9.11] put it: "In many ways, they carried the torch for the rest of deep learning and paved the way to the acceptance of neural networks in general."

[4] Another related limitation, although specific to the case of recurrent networks, was the difficulty in training them efficiently on very long sequences. This was resolved in 1997 by Hochreiter and Schmidhuber with the *Long short-term memory* (LSTM) architecture [77], presented in Section 5.11.3.

An important advance was the invention of the *pre-training* technique[5] by Hinton *et al.* in 2006 [74], which resolved this limitation. In 2012, an image recognition competition (the ImageNet Large Scale Visual Recognition Challenge [153]) was won by a deep neural network algorithm named AlexNet[6], with a stunning margin[7] over the other algorithms which were using handcrafted features. This striking victory was the event which ended the prevalent opinion that neural networks with many hidden layers could not be efficiently trained[8].

5.2 Introduction to Neural Networks

The purpose of this section is to review, or to introduce, the basic principles of *artificial neural networks*. Our objective is to define the key *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, like autoencoders, recurrent networks, RBMs, etc., which are used in musical applications. We will not describe extensively the techniques of neural networks and deep learning, for example covered in the recent book [58].

5.2.1 Linear Regression

Although bio-inspired (biological neurons), 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 $y \in \mathbb{R}$ and a set of one[9] or more *explanatory variable(s)* x_1 ... x_n, with $x_i \in \mathbb{R}$, jointly noted as vector x. A simple example is to predict the value of a house, depending on some factors (e.g., size, height, location...).

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

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

- h is the *model*, also named *hypothesis*, as this is the hypothetical best model to be discovered, i.e. learnt;
- b is the *bias*, representing the *offset*, also sometimes noted θ_0; and
- θ_1 ... θ_n are the *parameters* of the model, the *weights*, corresponding to the explanatory variables x_1 ... x_n.

5.2.2 Linear Algebra Representation

Equation 5.1 may be made more compact thanks to a linear algebra notation leading to Equation 5.2, where

$$h(x) = b + \theta^{\mathrm{T}} x \tag{5.2}$$

[5] Pre-training consists in prior training in *cascade* (one layer at a time, also named *greedy layer-wise unsupervised training*) of each hidden layer [74] [58, page 528]. It turned out to be a significant improvement for the accurate training of neural networks with several layers [40]. That said, pre-training is now rarely used and has been replaced by other more recent techniques, such as *batch normalization* and *deep residual learning*. But its underlying techniques are useful for addressing some new concerns like *transfer learning*, which deals with the issue of *reusability* (of what has been learnt, see Section 8.3).

[6] AlexNet was designed by the SuperVision team headed by Hinton and composed of Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton [95]. AlexNet is a deep convolutional neural network with 60 million parameters and 650,000 neurons, consisting of five convolutional layers, some followed by max-pooling layers, and three globally-connected layers.

[7] On the first task, AlexNet won the competition with a 15% error rate whereas other teams did not achieve better than a 26% error rate.

[8] Interestingly, the title of Hinton *et al.*'s article about pre-training [74] is about "deep belief nets" and does not mention the term "neural nets" because, as Hinton remembers it in [97]: "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."

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

- b and $h(x)$ are scalars;
- θ is a column vector, that is a matrix of dimension $n \times 1$, consisting of a single column of n elements: $\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$;
- θ^{T} is the transposition of column vector θ and is a a row vector, that is a matrix of dimension $1 \times n$, consisting of a single row of n elements: $\begin{bmatrix} \theta_1 & \theta_2 & \dots & \theta_n \end{bmatrix}$; and
- x is a vector consisting of a single column of n elements: $\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$.

5.2.3 Model Training

The purpose of training a linear regression model is to find the values for each weight θ_i (also denoted W_i) and the bias b that best fit the actual training data/examples, i.e. various pairs of values (x, y). In other words, we want to find the parameters and bias values such that for all values of x, $h(x)$ is as close as possible[10] to y, according to some measure named the *cost* or also the *loss*, that represents the *distance* between $h(x)$, the prediction, and y, the actual value, for *all* examples. It is usually[11] notated $J_\theta(h)$ and could be measured, for example, by a mean squared error, which measures the average squared difference, shown in Equation 5.3, where m is the number of examples and $(x^{(i)}, y^{(i)})$ is the ith example pair.

$$J_\theta(h) = 1/m \sum_{i=1}^{m} (y^{(i)} - h(x^{(i)}))^2 \tag{5.3}$$

An example is shown in Figure 5.2 for the case of simple linear regression, i.e. with only one explanatory variable x. Training data are shown as blue solid dots. Once the model has been trained, values of the parameters are adjusted, illustrated by the blue solid bold line which mostly fits the examples. Then, the model can be used for *prediction*, e.g., to provide a good estimate of the actual value of y for a given value of x by computing $h(x)$ (shown in green).

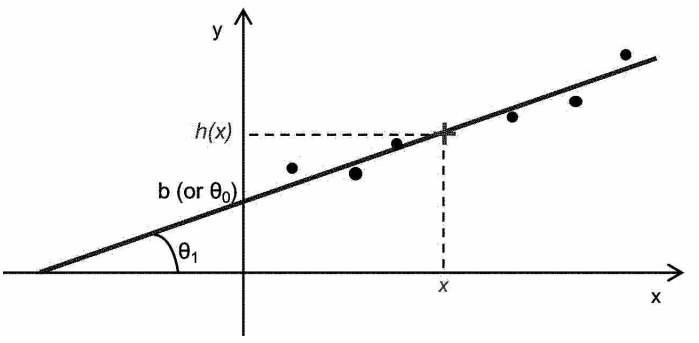

Fig. 5.2 Example of simple linear regression

[10] Actually, for the neural networks that are more complex (nonlinear models) than linear regression and that will be introduced in Section 5.8, the best fit to the training data is not necessarily the best hypothesis because it may have a low *generalization*, i.e. a low ability to predict *yet unseen data*. This issue, named *overfitting*, will be introduced in Section 5.8.7.

[11] Or also $J(\theta)$, $\mathscr{L}_\theta$ or $\mathscr{L}(\theta)$.

5.2.4 Gradient Descent Training Algorithm

The basic algorithm for training a linear regression model, using the simple *gradient descent* method, is actually pretty simple[12]:

- initialize each parameter θ_i and the bias b to a random or some heuristic value[13];
- compute the values of the model h for all examples[14];
- compute the *cost* $J_\theta(h)$ (also named the *loss*), as, e.g., by Equation 5.3;
- compute the *gradients* $\frac{\partial J_\theta(h)}{\partial \theta_i}$ which are the *partial derivatives* of the cost function $J_\theta(h)$ with respect to each θ_i, as well as to the bias b;
- *update simultaneously*[15] all parameters θ_i and the bias according to the update rule[16] shown in Equation 5.4, with α being the *learning rate*.

$$\theta_i := \theta_i - \alpha \frac{\partial J_\theta(h)}{\partial \theta_i} \tag{5.4}$$

  This represents an update in the opposite direction of the gradients in order to decrease the cost $J_\theta(h)$, as illustrated in Figure 5.3; and
- *iterate* until the error reaches *a minimum*[17], or after a certain number of iterations.

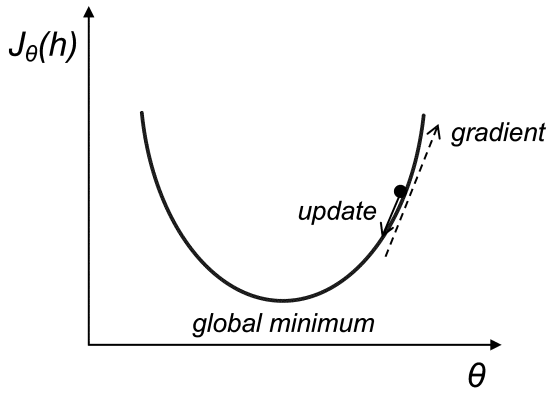

Fig. 5.3 Gradient descent

[12] See, e.g., [130] for more details.

[13] Pre-training led to a significant advance, as it improved the initialization of the parameters by using actual training data, via sequential training of the successive layers [40].

[14] Computing the cost for all examples is the best method but also computationally costly. There are numerous heuristic alternatives to minimize the computational cost, e.g., *stochastic gradient descent* (SGD), where one example is randomly chosen, and *minibatch gradient descent*, where a subset of examples is randomly chosen. See, for example, [58, Sections 5.9 and 8.1.3] for more details.

[15] A simultaneous update is necessary for the algorithm to behave correctly.

[16] The rule may also be notated as $\theta := \theta - \alpha \nabla_\theta J_\theta(h)$, where $\nabla_\theta J_\theta(h)$ is the vector of gradients $\frac{\partial J_\theta(h)}{\partial \theta_i}$.

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

5.3 Machine Learning Components

Note that searching for values that minimize the cost function is an *optimization* problem. One of the most simple optimization algorithms is gradient descent, introduced in previous section. There are various more sophisticated algorithms, such as stochastic gradient descent (SGD), Nesterov accelerated gradient (NAG), Adagrad, BFGS, etc. (see, for example, [58, Chapter 9] for more details). Optimization is indeed one of the most important components of machine learning algorithms. In his introduction to machine learning [34], Domingos describes machine learning algorithms through three components:

- *representation* – the way to represent the model – in our case, a *neural network*;
- *evaluation* – the way to evaluate and compare models – the *cost function*; and
- *optimization* – the way to identify (search among models for) a best model.

5.3.1 From Model to Architecture

Let us now introduce in Figure 5.4 a graphical representation of a linear regression model, as a precursor of a neural network. The *architecture* represented is actually the computational representation of the model[18].

The weighted sum is represented as a *computational unit*[19], drawn as a squared box with a $\sum$, taking its inputs from the x_i nodes, drawn as circles.

In the example shown, there are four explanatory variables: x_1, x_2, x_3 and x_4. Note that there is a convention of considering the bias as some special case of weight, thus having a corresponding input node named the *bias node*, which is *implicit*[20] and has a constant value notated as $+1$.

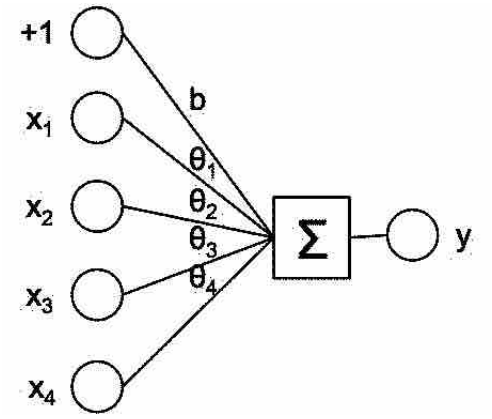

Fig. 5.4 Architectural model of linear regression

Linear regression can also be generalized to *multivariate linear regression*, the case when there are multiple variables y_1 ... y_p to be predicted, as illustrated in Figure 5.5 with three predicted variables: y_1, y_2 and y_3, each subnetwork represented in a different color.

[18] We mostly use the term *architecture* as, in this book, we are concerned with the way to implement and compute a given model and also with the relation between an architecture and a representation.

[19] We use the term *node* for any component of a neural network, whether it is just an *interface* (e.g., an input node) or a *computational unit* (e.g., a weighted sum or a function). We use the term *unit* only in the case of a computational node. The term *neuron* is also often used in place of unit, as a way to emphasize the inspiration from biological neural networks.

[20] However, as will be explained later in Section 5.8, bias nodes rarely appear in illustrations of non-toy neural networks.

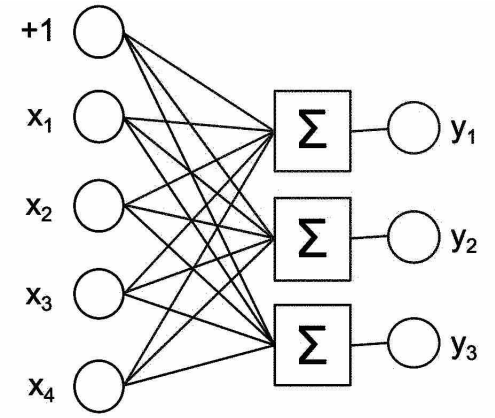

Fig. 5.5 Architectural model of multivariate linear regression

5.3.2 Activation Function

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 *nonlinear functions*.

- From an *engineering* perspective, a nonlinear function is necessary to overcome the linear separability limitation of the single layer Perceptron (see Section 5.1).
- From a *biological inspiration* perspective, a nonlinear function can capture the *threshold* effect for the activation of a neuron through its incoming signals (via its dendrites), determining whether it fires along its output (axone).

Historically speaking, the *sigmoid* function which is used for *logistic regression*, is the most common. The sigmoid function (usually written σ) is defined in Equation 5.5 and is shown in Figure 5.6. It will be further analyzed in Section 5.8.3. An alternative is the *hyperbolic tangent*, often noted *tanh*. But ReLU is now widely used for its simplicity and effectiveness. ReLU, which stands for *rectified linear unit*, is defined in Equation 5.5 and is shown in Figure 5.7.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5.5}$$

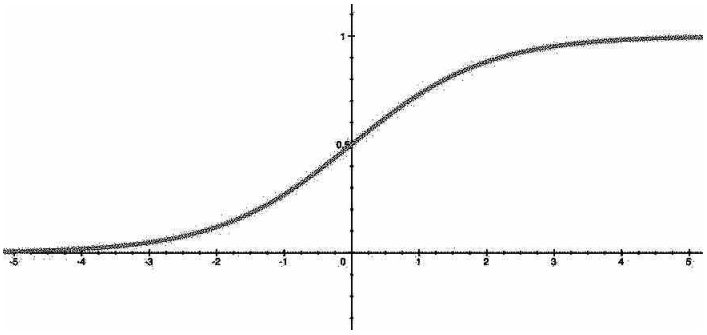

Fig. 5.6 Sigmoid function

$$ReLU(x) = max(0,x) \tag{5.6}$$

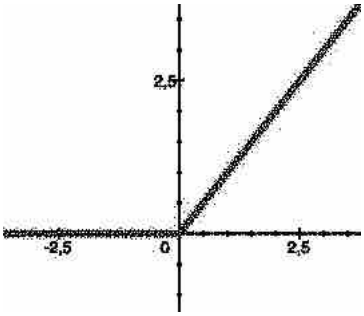

Fig. 5.7 ReLU function

5.4 Basic Building Block

The resulting architectural representation (of multivariate linear regression with activation functions) is shown in Figure 5.8. This is a *basic building block* of neural networks and deep learning architectures.

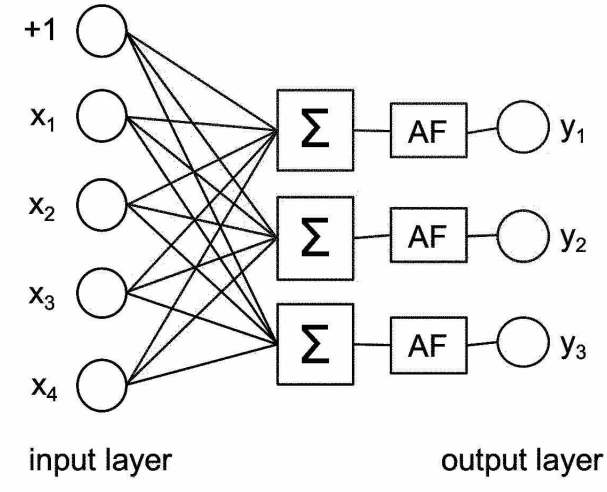

Fig. 5.8 Building block architecture

Although simple, this basic building block, is actually a working neural network. It has two layers[21]:

- The *input layer*, on the left of the figure, is composed of the *input nodes* x_i and the *bias node* which is an *implicit* and specific input node with a constant value of 1, therefore usually denoted as $+1$.
- The *output layer*, on the right of the figure, is composed of the *output nodes* y_j.

5.4.1 Linear Algebra Representation

By using linear algebra representation and notation, as for linear regression in Section 5.2.2, we can consider the bias as a vector b and the weights as a matrix W where

[21] Although, as we will see in Section 5.8.2, it will be considered as a single-layer neural network architecture. As it has no hidden layer, it still suffers from the linear separability limitation of the Perceptron.

- the b bias vector is a column vector of dimension $p \times 1$, with b_j representing the weight of the connexion between the bias input node and the jth sum operation corresponding to the jth output node;
- the W weight matrix is a matrix of dimension $n \times p$, that is with n rows and p columns, with $W_{i,j}$ representing the weight of the connexion between the ith input node and the jth sum operation corresponding to the jth output node;
- n is the number of input nodes (without considering the bias node); and
- p is the number of output nodes.

In the architecture illustrated in Figure 5.9, where the biases as well as two examples of weights are represented, $n = 4$ (the number of rows and of input nodes) and $p = 3$ (the number of columns and of output nodes). The corresponding b bias vector and W weight matrix are shown in Equations 5.7 and 5.8.

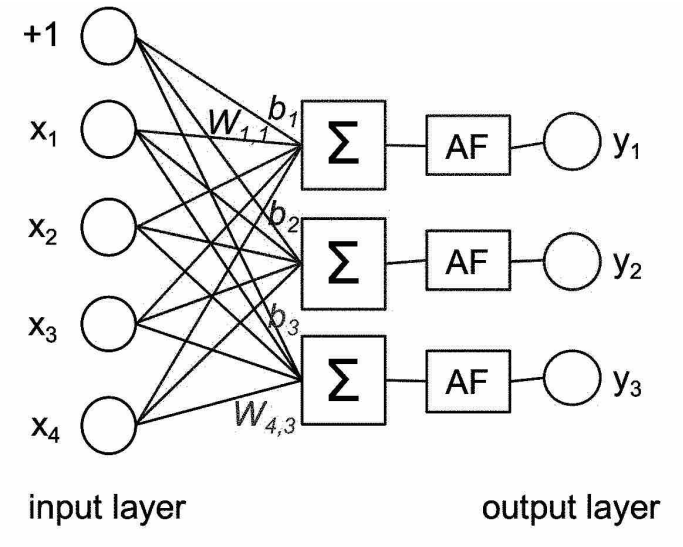

Fig. 5.9 Building block architecture showing the biases and two examples of weights

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{5.7}$$

$$W = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \\ W_{4,1} & W_{4,2} & W_{4,3} \end{bmatrix} \tag{5.8}$$

5.4.2 Training

Training a basic block is essentially the same as training a linear regression model, which has been described in Section 5.2.3.

5.5 Machine Learning

5.5.1 Definition

Let us now reflect a bit on the meaning of training a model, whether it is a linear regression model (Section 5.2.1) or the basic block architecture presented in Section 5.4. Therefore, let us consider what machine learning actually means. Our starting point is the following concise and general definition of machine learning provided by Mitchell in [120]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Note that the word *performance* covers different meanings:

1. the execution of (the action to perform) an action, notably an artistic act such as a musician playing a piece of music;
2. a *measure* (criterium of evaluation) of that action, notably for a computer system its *efficiency* in performing a task, in terms of time and memory; or
3. a measure of the *accuracy* in performing a task, i.e. the ability to predict or classify with minimal errors.

In the remainder of the book, in order to try to minimize ambiguity, we will use the terms as following:

- *performance* as an act by a musician,
- *efficiency* as a measure of computational ability, and
- *accuracy* as a measure of the quality of a prediction or a classification.

5.5.2 Categories

We may now consider the three main categories of machine learning with regard to the nature of the experience conveyed by the examples:

- *supervised learning* – the dataset is fixed and a correct (expected) answer[22] is associated to each example, the general objective being to *predict answers* for new examples. Examples of tasks are regression (prediction), classification and translation;
- *unsupervised learning* – the dataset is fixed and the general objective is in *extracting information*. Examples of tasks are feature extraction, data compression (both performed by *autoencoders*, to be introduced in Section 5.9), probability distribution learning (performed by *RBMs*, to be introduced in Section 5.10), series modeling (performed by *recurrent* networks, to be introduced in Section 5.11), clustering and anomaly detection; and
- *reinforcement learning*[23] – the experience is incremental through successive actions within an environment, with some feedback providing information about the *value* of the action, the general objective being to *learn a strategy*. Examples of tasks are game playing and robot navigation.

5.6 Feedforward Computation

After it has been trained, we can use the basic building block neural network defined in Section 5.4 for prediction. Therefore, we simply *feedforward* the network, i.e. provide input data to the network (*feed in*) and compute the output

[22] It is usually named a *label* in the case of a *classification* task and a *target* in the case of a *prediction/regression* task.

[23] To be introduced in Section 5.15.

values. This corresponds to the equation[24] shown in Equation 5.9, where W^{T} in Equation 5.10 is the transposition of the weight matrix W in Equation 5.8.

$$h(x) = AF(b + W^{\mathrm{T}}x) \tag{5.9}$$

$$W^{\mathrm{T}} = \begin{bmatrix} W_{1,1} & W_{2,1} & W_{3,1} & W_{4,1} \\ W_{1,2} & W_{2,2} & W_{3,2} & W_{4,2} \\ W_{1,3} & W_{2,3} & W_{3,3} & W_{4,3} \end{bmatrix} \tag{5.10}$$

The computation of the prediction is as follows (Equation 5.11), where $h_j(x)$ is the prediction of the jth variable y_j:

$$\begin{aligned} h(x) &= h(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}) = AF(b + W^{\mathrm{T}}x) \\ &= AF(\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} & W_{1,4} \\ W_{2,1} & W_{2,2} & W_{2,3} & W_{2,4} \\ W_{3,1} & W_{3,2} & W_{3,3} & W_{3,4} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}) \\ &= AF(\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + W_{1,4}x_4 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + W_{2,4}x_4 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + W_{3,4}x_4 \end{bmatrix}) \\ &= AF(\begin{bmatrix} b_1 + (W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + W_{1,4}x_4) \\ b_2 + (W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + W_{2,4}x_4) \\ b_3 + (W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + W_{3,4}x_4) \end{bmatrix}) = \begin{bmatrix} h_1(x) \\ h_2(x) \\ h_3(x) \end{bmatrix} \end{aligned} \tag{5.11}$$

Feedforwarding simultaneously a set of examples is easily expressed as a matrix to matrix multiplication, by substituting the single vector example in Equation 5.9 with a matrix of examples. Successive columns of the matrix of examples X correspond to the different examples. We use a superscript notation $X^{(k)}$ to denote the kth example, the kth column of the X matrix, to avoid confusion with the subscript notation x_i which is used to denote the ith input variable. Therefore, $X_i^{(k)}$ denotes the ith variable of the kth example. The new Equations 5.12 and 5.13, are as follows, with the corresponding predictions $h(X^{(k)})$ being successive columns of the resulting output matrix:

$$h(X) = AF(b + W^{\mathrm{T}}X) \tag{5.12}$$

[24] Note that an alternative formulation is $x^{\mathrm{T}}W$ which is equal to $(W^{\mathrm{T}}x)^{\mathrm{T}}$, see [58, Section 2.2] for details. The choice could be an optimization decision for the implementation platform, the first formulation being more efficient in the case of sparse input.

$$
\begin{aligned}
h(X) &= h(\begin{bmatrix} X_1^{(1)} & X_1^{(2)} & \dots & X_1^{(m)} \\ X_2^{(1)} & X_2^{(2)} & \dots & X_2^{(m)} \\ X_3^{(1)} & X_3^{(2)} & \dots & X_3^{(m)} \\ X_4^{(1)} & X_4^{(2)} & \dots & X_4^{(m)} \end{bmatrix}) = AF(b + W^{\mathrm{T}} X) \\
&= AF(\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} & W_{1,4} \\ W_{2,1} & W_{2,2} & W_{2,3} & W_{2,4} \\ W_{3,1} & W_{3,2} & W_{3,3} & W_{3,4} \end{bmatrix} \times \begin{bmatrix} X_1^{(1)} & X_1^{(2)} & \dots & X_1^{(m)} \\ X_2^{(1)} & X_2^{(2)} & \dots & X_2^{(m)} \\ X_3^{(1)} & X_3^{(2)} & \dots & X_3^{(m)} \\ X_4^{(1)} & X_4^{(2)} & \dots & X_4^{(m)} \end{bmatrix}) \\
&= \begin{bmatrix} h_1(X^{(1)}) & h_1(X^{(2)}) & \dots & h_1(X^{(m)}) \\ h_2(X^{(1)}) & h_2(X^{(2)}) & \dots & h_2(X^{(m)}) \\ h_3(X^{(1)}) & h_3(X^{(2)}) & \dots & h_3(X^{(m)}) \end{bmatrix}
\end{aligned}
\tag{5.13}
$$

Note that the main computation[25] is a product of matrices. This can be computed very efficiently, by using linear algebra vectorized implementation libraries and furthermore with specialized hardware like graphics processing units (GPUs).

5.7 Architectures

From this basic building block, we will describe in the following sections the main *types* of *deep learning architectures* used for music generation (as well as for other purposes):

- feedforward,
- autoencoder,
- restricted Boltzmann machine (RBM), and
- recurrent (RNN).

We will also introduce *architectural patterns* (see Section 5.16.1) which could be applied to them:

- convolutional,
- conditioning, and
- adversarial.

5.8 Multilayer Neural Network *aka* Feedforward Neural Network

A *multilayer neural network*, also named a *feedforward 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*; and
- any layer *between* the input layer and the output layer is named a *hidden layer*.

An example of a multilayer neural network with two hidden layers is illustrated in Figure 5.10.

The combination of a hidden layer and a nonlinear activation function makes the neural network a *universal approximator*, able to overcome the *linear separability* limitation[26].

[25] Apart from the computation of the AF activation function. In the case of ReLU this is fast.

[26] The universal approximation theorem [80] states that a feedforward 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). However, there is no guarantee that the neural network will learn them!

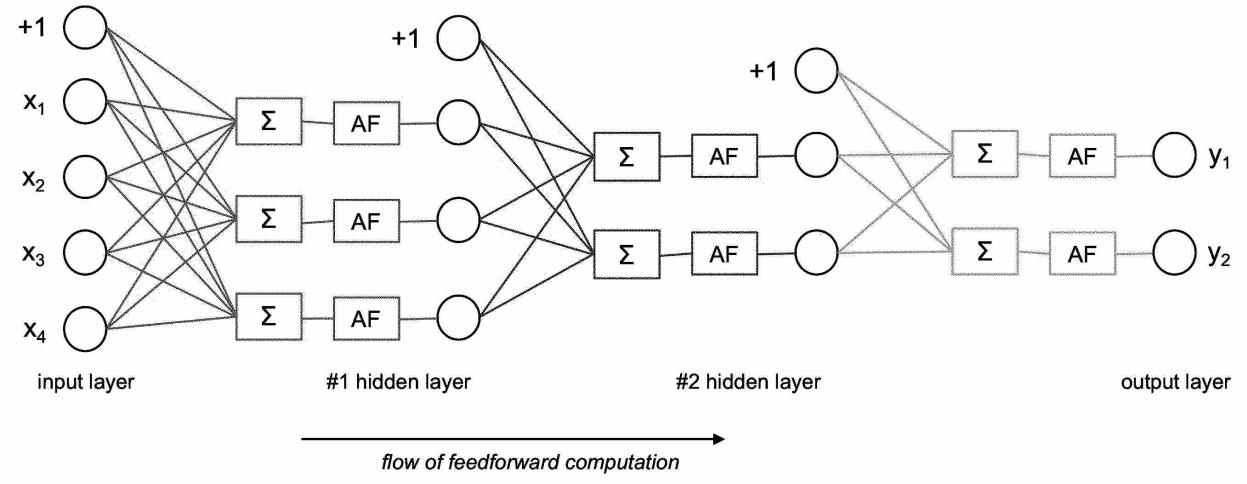

Fig. 5.10 Example of a feedforward neural network (detailed)

5.8.1 Abstract Representation

Note that, in the case of practical (non-toy) illustrations of neural network architectures, in order to simplify the figures, bias nodes are very rarely illustrated. With a similar objective, the sum and nonlinear function units are also almost always omitted, resulting in a more abstract view such as that shown in Figure 5.11.

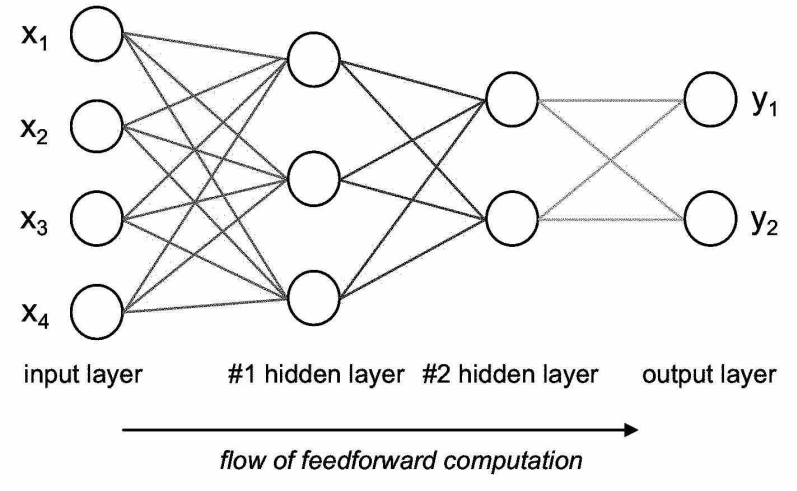

Fig. 5.11 Example of feedforward neural network (simplified)

We can further abstract each layer by representing it as an oblong form (by hiding its nodes)[27] as shown in Figure 5.12.

[27] It is sometimes pictured as a rectangle, see Figure 5.13, or even as a circle, notably in the case of recurrent networks, see Figure 5.23.

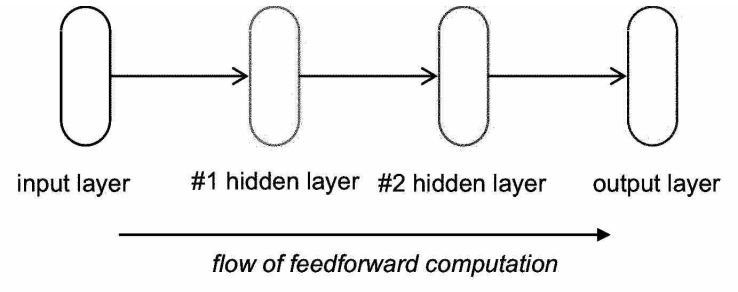

Fig. 5.12 Example of a feedforward neural network (abstract)

5.8.2 Depth

The architecture illustrated in Figure 5.12 is called a 3-layer neural network architecture, also indicating that the *depth* of the architecture is three. Note that the number of layers (depth) is indeed three and not four, irrespective of the fact that summing up the input layer, the output layer and the two hidden layers gives four and not three. This is because, by convention, only layers with weights (and units) are considered when counting the number of layers in a multilayer neural network; therefore, the input layer is not counted. Indeed, the input layer only acts as an input interface, without any weight or computation.

In this book, we will use a superscript (power) notation to denote the number of layers of a neural network architecture. For instance, the architecture illustrated in Figure 5.12 could be denoted as Feedforward^3.

The depth of the first neural network architectures was small. The original Perceptron [151], the ancestor of neural networks, has only an input layer and an output layer without any hidden layer, i.e. it is a single-layer neural network. In the 1980s, conventional neural networks were mostly 2-layer or 3-layer architectures.

For modern deep networks, the depth can indeed be very large, deserving the name of *deep* (or even *very deep*) networks. Two recent examples, both illustrated in Figure 5.13, are

- the 27-layer GoogLeNet architecture [166]; and
- the 34-layer (up to 152-layer!) ResNet architecture[28] [69].

Note that depth *does* matter. A recent theorem [38] states that there is a simple radial function[29] on $\mathbb{R}^d$, expressible by a 3-layer neural network, which cannot be approximated by any 2-layer network to more than a constant accuracy unless its width is exponential in the dimension d. Intuitively, this means that reducing the depth (removing a layer) means exponentially augmenting the width (the number of units) of the layer left. On this issue, the interested reader may also wish to review the analyses in [3] and [175].

Note that for both networks pictured in Figure 5.13, the flow of computation is vertical, upward for GoogLeNet and downward for ResNet. These are different conventions from the one that we have used so far, which is horizontal, from left to right. Unfortunately, there is no consensus in the literature about the notation for the flow of computation. Note that in the specific case of recurrent networks, to be introduced in Section 5.11, the consensus notation is vertical, upward.

[28] It introduces the technique of *residual learning*, reinjecting the input between levels and estimating the residual function $h(x) - x$, a technique aimed at very deep networks, see [69] for more details.

[29] A radial function is a function whose value at each point depends only on the distance between that point and the origin. More precisely, it is radial *if and only if* it is invariant under all rotations while leaving the origin fixed.

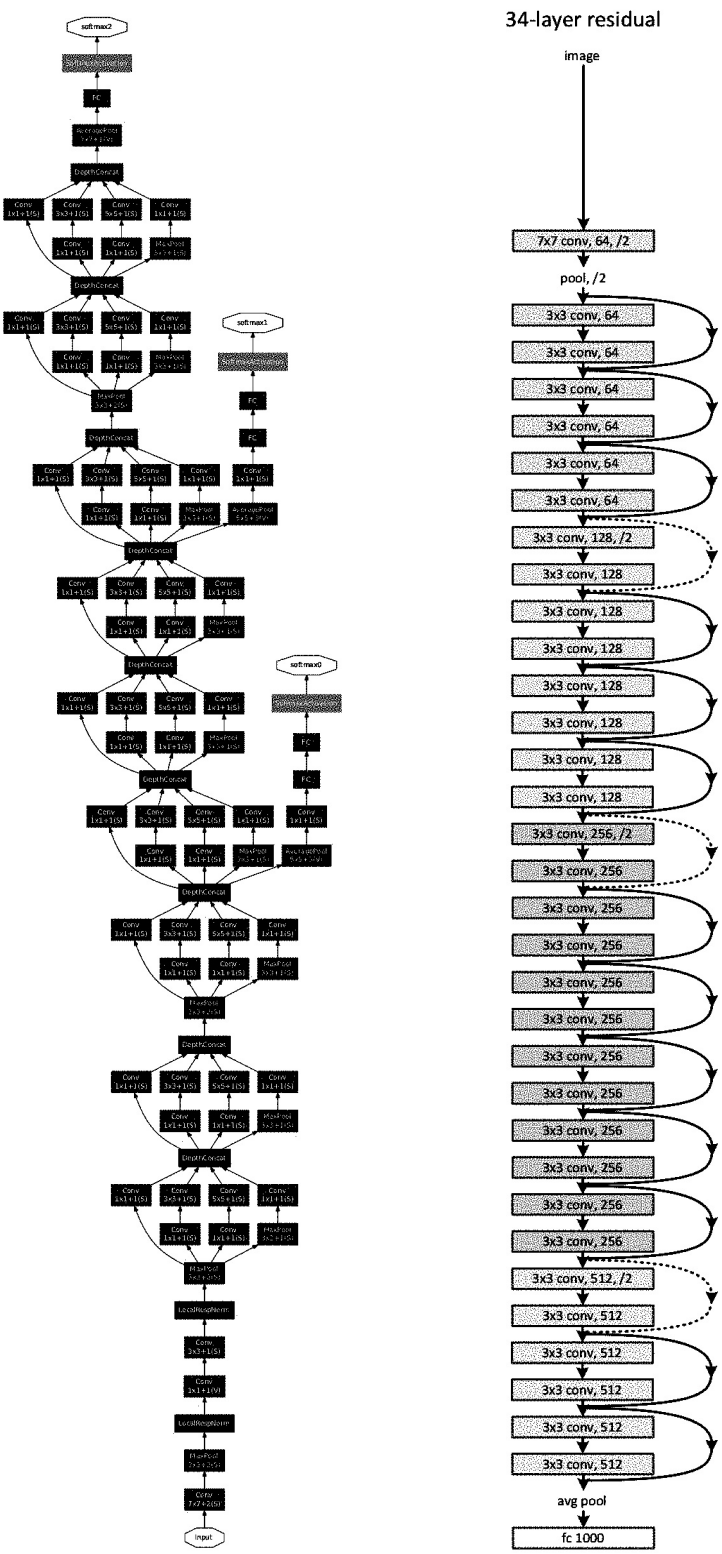

Fig. 5.13 (left) GoogLeNet 27-layer deep network architecture. Reproduced from [166] with the permission of the authors. (right) ResNet 34-layer deep network architecture. Reproduced from [69] with the permission of the authors.

5.8.3 Output Activation Function

We have seen in Section 5.4 that, in modern neural networks, the activation function (AF) chosen for introducing nonlinearity at the output of each hidden layer is the ReLU function. But the output layer of a neural network has a special status. Basically, there are three main possible types of activation function for the output layer:

- *identity* – this is the case for so-called *linear neural networks*, for prediction tasks with continuous output values. Therefore, there is no nonlinear transformation at the last layer;
- *sigmoid* – this is used for binary classification tasks, as in logistic regression[30]. The sigmoid function (usually written σ) has been defined in Equation 5.5 and shown in Figure 5.6. Note its specific shape, which provides a "separation" effect, used for binary decision (classification); or
- *softmax* – this is the most common approach for classification tasks with more than two classes, and also for prediction tasks with a discrete value (for instance predicting the pitch of a note)[31], where a one-hot encoding is generally used (see Section 4.11).

The softmax function actually represents a *probability distribution* over a discrete output variable with p possible values (in other words, the probability of the occurrence of each value j, knowing the input value x, i.e. $p(y = j|x)$). Therefore, softmax ensures that the sum of the probabilities for each possible value is equal to 1. The softmax function is defined[32] in Equation 5.14 and an example of its use is shown in Equation 5.15. Note that the σ notation is used for the softmax function, as for the sigmoid function, because softmax is the generalization of sigmoid to the case of multiple values, being a variadic function, that is one which accepts a variable number of arguments.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{j=1}^{p} e^{z_j}} \tag{5.14}$$

$$\sigma \begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix} \tag{5.15}$$

For a classification or prediction task, we can simply select the value with the highest probability. But the distribution produced by the softmax function can also be used as the basis for *sampling*, in order to add variability and nondeterminism to the generation (this will be detailed in Section 6.5).

Last, note that the ReLU function is routinely used as the activation function in hidden layers, in order to introduce nonlinearity. But ReLU is never used as an activation function in the output layer, as it does not have the "separation" effect of sigmoid.

5.8.4 Cost Function

The most common cost functions are

- *quadratic cost*, also known as *mean squared error* or *maximum likelihood*;
- *cross-entropy*; and
- *Kullback-Leibler divergence* (KL-divergence).

The choice of a cost function is actually closely related to the choice of output activation function. For instance, for a prediction (regression) task, a quadratic cost function is often chosen, along with no (i.e. identity) output activation function. For a classification task, a cross-entropy cost function is often chosen.

[30] For details about logistic regression, see, for example, [58, Chapter 6]. For this reason, the sigmoid function is also called the *logistic function*.

[31] Indeed, this is the reformulation of a prediction (regression) task for a single value as a classification task between different possible discrete values.

[32] Note that z is traditionally used to represent the output value, as y is reserved for the actual data.

More details and principles[33] can be found, for example, in [58, Section 6.2.1] and [58, Section 5.5], respectively.

5.8.5 Feedforward Propagation

Feedforward propagation in a multilayer neural network consists in injecting input data[34] into the input layer and propagating the computation through its successive layers[35] until the output is produced. This can be implemented very efficiently because it consists in a pipelined computation of successive vectorized matrix products (intercalated with AF activation function calls). Each layer k is processed as in Equation 5.16, which is a generalization of Equation 5.9, where $b^{(k)}$ and $W^{(k)}$ are respectively the bias and the weight matrix of the k layer.

$$output^{(k)} = AF(b^{(k)} + W^{(k)\mathrm{T}} output^{(k-1)}) \tag{5.16}$$

Multilayer neural networks are therefore often also named *feedforward neural networks* or *multilayer Perceptron* (MLP)[36].

Note that neural networks are *deterministic*. This means that the same input will deterministically *always* produce the *same* output. This is a useful guarantee for prediction and classification purposes but may be a limitation for generating new content. However, this may be compensated by *sampling* from the resultant probability distribution (see Sections 5.8.3 and 6.5).

5.8.6 Training

For the training phase[37], computing the derivatives becomes a bit more complex than for the basic building block (with no hidden layer) presented in Section 5.2.4. *Backpropagation* is the standard method of estimating the derivatives (gradients) for a multilayer neural network. It is based on the *chain rule* principle [152], in order to estimate the contribution of each weight to the final prediction error, that is the cost. See, for example, [58, 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 may be *multiple local minima*. Gradient descent, as well as other more sophisticated heuristic optimization methods, does not guarantee the global optimum will be reached. But in practice a clever configuration of the model (notably, its *hyperparameters*, see Section 5.8.9) and well-tuned optimization heuristics, such as stochastic gradient descent (SGD), will lead to accurate solutions[38].

5.8.7 Overfitting

A fundamental issue for neural networks (and more generally speaking for machine learning algorithms) is their *generalization* ability, that is their capacity to perform well on *yet unseen data*. In other words, we do not want a

[33] The underlying principle of *maximum likelihood estimation*, not explained here.

[34] The x part of an example, for the generation phase as well as for the training phase.

[35] Feedforward computation for one layer has been introduced in Section 5.6.

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

[37] Let us remember that this a case of supervised learning (see Section 5.4.2).

[38] On this issue, see [21], 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 and 4) in practice this is irrelevant as the global minimum often leads to overfitting (see next section).

neural network to just perform well on the training data[39] but also on future data[40]. This is actually a fundamental dilemma, the two opposing risks being

- *underfitting* – when the *training error* (error measure on the *training data*) is large; and
- *overfitting* – when the *generalization error* (expected error on *yet unseen data*) is large.

A simple illustrative example of underfit, good fit and overfit models for the same training data (the green solid dots) is shown in Figure 5.14.

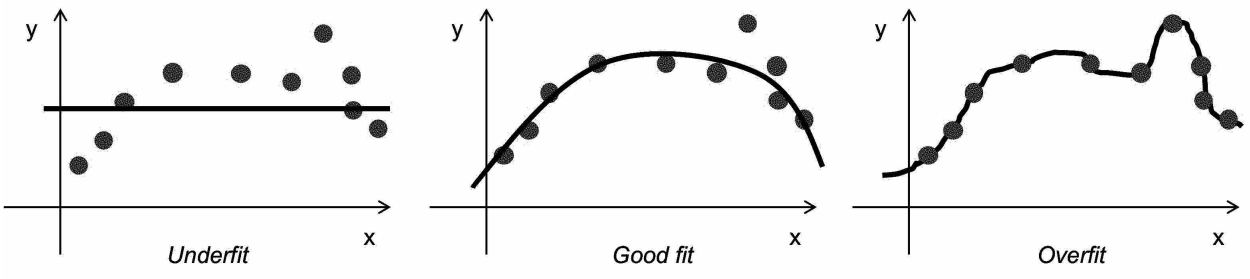

Fig. 5.14 Underfit, good fit and overfit models

In order to be able to estimate the potential for generalization, the dataset is actually divided into two portions, with a ratio of approximately 70/30:

- the *training set* – which will be used for training the neural network; and
- the *validation set*, also named *testing set* – which will be used to estimate the capacity of the model for generalization.

5.8.8 Regularization

There are various techniques to control overfitting, i.e., to improve generalization. They are usually named *regularization* and some examples of well-known techniques are

- *weight decay* (also known as L^2), by penalizing over-preponderant weights;
- *dropout*, by introducing random disconnections;
- *early stopping*, by storing a copy of the model parameters every time the error on the validation set reduces, then terminating after an absence of progress during a pre-specified number of iterations, and returning these parameters; and
- *dataset augmentation*, by data synthesis (e.g., by mirroring, translation and rotation for images; by transposition for music, see Section 4.12.1), in order to augment the number of training examples.

We will not further detail regularization techniques, see, for example, [58, Section 7].

[39] Otherwise, the best and simpler algorithm would be a memory-based algorithm, which simply *memorizes* all (x,y) pairs. It has the best fit to the training data but it does not have any generalization ability.

[40] Future data is not yet known but that does not mean that it is *any kind* of (random) data, otherwise a machine learning algorithm would not be able to learn and generalize well. There is indeed a fundamental assumption of regularity of the data corresponding to a task (e.g., images of human faces, Jazz chord progressions, etc.) that neural networks will exploit.

5.8.9 Hyperparameters

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

Examples of *structural* hyperparameters, mainly concerned with the architecture, are

- number of layers,
- number of nodes, and
- nonlinear activation function.

Examples of *control* hyperparameters, mainly concerned with the learning process, are

- optimization procedure,
- learning rate, and
- regularization strategy and associated parameters.

Choosing proper values for (tuning) the various hyperparameters is fundamental both for the efficiency and the accuracy of neural networks for a given application. There are two approaches for exploring and tuning hyperparameters: *manual tuning* or *automated tuning* – by algorithmic exploration of the multidimensional space of hyperparameters and for each sample evaluating the generalization error. The three main strategies for automated tuning are

- *random search* – by defining a distribution for each hyperparameter, sampling configurations, and evaluating them;
- *grid search* – as opposed to random search, exploration is systematic on a small set of values for each hyperparameter; and
- *model-based optimization* – by building a model of the generalization error and running an optimization algorithm over it.

The challenge of automated tuning is its computational cost, although trials may be run in parallel. We will not detail these approaches here; however, further information can be found in [58, Section 11.4].

Note that this tuning activity is more objective for conventional tasks such as prediction and classification because the evaluation measure is objective, being the error rate for the testing data. When the task is the generation of new musical content, tuning is more subjective because there is no preexisting evaluation measure. It then turns out to be more *qualitative*, for instance through a manual evaluation of generated music by musicologists. This evaluation issue is addressed in Section 8.5.

5.8.10 Platforms and Libraries

Various platforms[41], such as CNTK, MXNet, PyTorch and TensorFlow, are available as a foundation for developing and running deep learning systems[42], They include libraries of

- basic architectures, such as the ones we are presenting in this chapter;
- components, for example optimization algorithms;
- runtime interfaces for running models on various hardware, including GPUs or distributed Web runtime facilities; and
- visualization and debugging facilities.

Keras is an example of a higher-level framework to simplify development, with CNTK, TensorFlow and Theano as possible backends. ONNX is an open format for representing deep learning models and was designed to ease the transfer of models between different platforms and tools.

[41] See, for example, the survey in [142].

[42] There are also more general libraries for machine learning and data analysis, such as the SciPy library for the Python language, or the language R and its libraries.

5.9 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[43]. The output layer actually *mirrors* the input layer. It is shown in Figure 5.15, with its peculiar symmetric diabolo (or sand-timer) shape aspect.

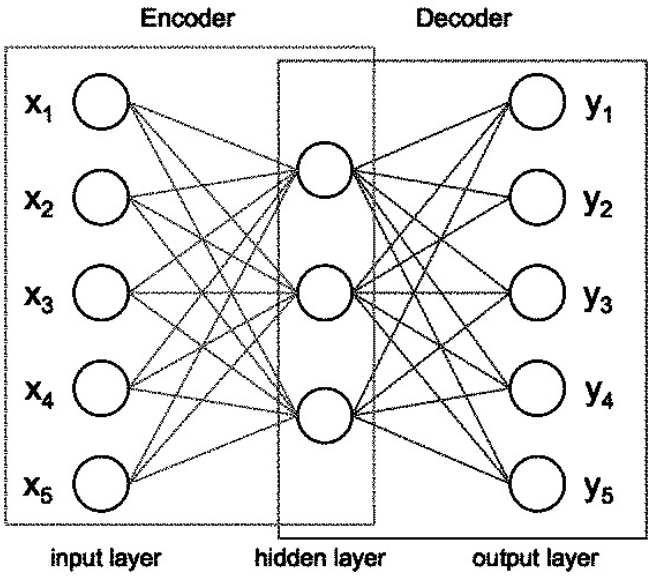

Fig. 5.15 Autoencoder architecture

Training an autoencoder represents a case of *unsupervised learning*, as the examples do not contain any additional label information (the effective value or class to be predicted). But the trick is that this is implemented using conventional supervised learning techniques, by presenting output data equal to the input data[44]. In practice, the autoencoder tries to learn the *identity* function. As the hidden layer usually has fewer nodes than the input layer, the *encoder* component (shown in yellow in Figure 5.15) must *compress* information while the *decoder* (shown in purple) has to *reconstruct*, as accurately as possible, the initial information[45]. This forces the autoencoder to *discover* significant (discriminating) *features* to encode useful information into the hidden layer nodes (also named the *latent variables*[46]). Therefore, autoencoders may be used to automatically extract high-level *features* [101]. The set of features extracted are often named an *embedding*[47]. Once trained, in order to extract features from an input, one just needs to feedforward the input data and gather the activations of the hidden layer (the values of the latent variables).

Another interesting use of decoders is the high-level control of content generation. The latent variables of an autoencoder constitute a compact representation of the common features of the learnt examples. By instantiating these latent variables and decoding the embedding, we can generate a new musical content corresponding to the values of the latent variables. We will explore this strategy in Section 6.3.1.

---

[43] The bias is not counted/considered here as it is an implicit additional input node.

[44] This is sometimes called *self-supervised* learning [101].

[45] Compared to traditional dimension reduction algorithms, such as principal component analysis (PCA), this approach has two advantages: 1) feature extraction is nonlinear – the case of *manifold learning*, see [58, Section 5.11.3] and Section 5.9.2 –, and 2) in the case of a sparse autoencoder (see next section), the number of features may be arbitrary (and not necessarily smaller than the number of input parameters).

[46] In statistics, *latent variables* are variables that are not directly observed but are rather inferred (through a mathematical model) from other variables that are observed (directly measured). They can serve to reduce the dimensionality of data.

[47] See the definition of embedding in Section 4.9.3.

5.9.1 Sparse Autoencoder

A *sparse autoencoder* is an autoencoder with a *sparsity* constraint, such that its hidden layer units are inactive most of the time. The objective is to enforce the *specialization* of each unit in the hidden layer as a specific *feature detector*.

For instance, a sparse autoencoder with 100 units in its hidden layer and trained on 10×10 pixel images will learn to detect edges at different positions and orientations in images, as shown in Figure 5.16. When applied to other input domains, such as audio or symbolic music data, this algorithm will learn useful features for those domains too.

The sparsity constraint is implemented by adding an additional term to the cost function to be minimized, see more details in [129] or [58, Section 14.2.1].

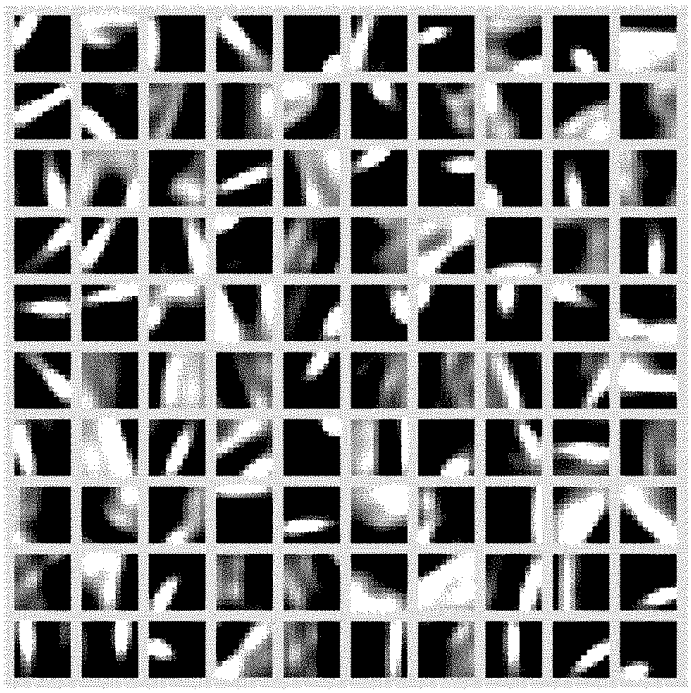

Fig. 5.16 Visualization of the input image motives that maximally activate each of the hidden units of a sparse autoencoder architecture. Reproduced from [129] with the permission of the author

5.9.2 Variational Autoencoder

A *variational autoencoder* (VAE) [94] has the added constraint that the encoded representation, the latent variables denoted by z, follows some prior probability distribution $p(z)$. Usually, a *Gaussian distribution*[48] is chosen for its generality.

This constraint is implemented by adding a specific term to the cost function, by computing the cross-entropy between the values of the latent variables and the prior distribution[49]. For more details about VAEs, an example of tutorial could be found in [33] and there is a nice introduction of its application to music in [148].

[48] Also named *normal distribution*.

[49] 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 detail here.

As with an autoencoder, a VAE will learn the identity function, 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 for the latent variables $z \sim p(z)$, i.e. z following distribution $p(z)$;
- input it into the decoder; and
- feedforward the decoder to generate an output corresponding to the distribution of the examples, more precisely $x \sim p(x|z)$ following the conditional probability distribution learnt by the decoder.

This is in contrast to the need for indirect and computationally expensive strategies such as Gibbs sampling for other architectures such as RBM, to be introduced in Section 5.10.

By construction, a variational autoencoder is representative of the dataset that it has learnt, that is, for any example in the dataset, there is at least one setting of the latent variables which causes the model to generate something very similar to that example [33]. A very interesting characteristic of the variational autoencoder architecture for generation purposes – therefore often considered as one type of a class of models named *generative models* – is in the meaningful exploration of the latent space, as a variational autoencoder is able to learn a "smooth" latent space mapping to realistic examples.

Note that this general objective is named *manifold learning* and more generally *representation learning* [7], that is the learning of a representation capturing the topology of a set of examples. As defined in [58, Section 5.11.3], a *manifold* is a connected set of points (examples) that can be approximated by a smaller number of dimensions, each one corresponding to a local direction of variation. An intuitive example is a 2D map capturing the topology of cities dispersed on the 3D earth, where a movement on the map corresponds to a movement on the earth.

Once learnt by a VAE, the latent representation (a vector of latent variables) can be used to explore the latent space with various operations to control/vary the generation of content. Some examples of operations on the latent space, as proposed in [148] and [149] for the MusicVAE system described in Section 6.11.1, are

- translation;
- interpolation;
- averaging of some points;
- addition or subtraction of an attribute vector capturing a given characteristic[50].

Figure 5.17 shows an interesting comparison of melodies resulting from

- interpolation in the *data space*, that is the space of representation of melodies; and
- interpolation in the *latent space*, which is then decoded into the corresponding melodies.

The interpolation in the latent space produces more meaningful and interesting melodies, as can be heard in [147] and [150]. More details about these experiments will be provided in Section 6.11.1.

Variational autoencoders are therefore elegant and promising models, and as a result they are currently among the hot approaches explored for generating content with controlled variations. Application to music generation will be illustrated in Sections 6.9.2.3 and 6.11.1.

5.9.3 Stacked Autoencoder

The idea of a *stacked autoencoder* is to hierarchically nest successive autoencoders with decreasing numbers of hidden layer units. An example of a 2-layer stacked autoencoder[51], i.e. two nested autoencoders that we could notate as Autoencoder^2, is illustrated in Figure 5.18.

[50] This attribute vector is computed as the average latent vector for a collection of examples sharing that attribute (characteristic).

[51] Note that the convention in this case is to count and notate the number of nested autoencoders, i.e. the number of hidden layers. This is different from the depth of the *whole* architecture, which is double. For instance, a 2-layer stacked autoencoder results in a 4-layer whole architecture, as shown in Figure 5.18.

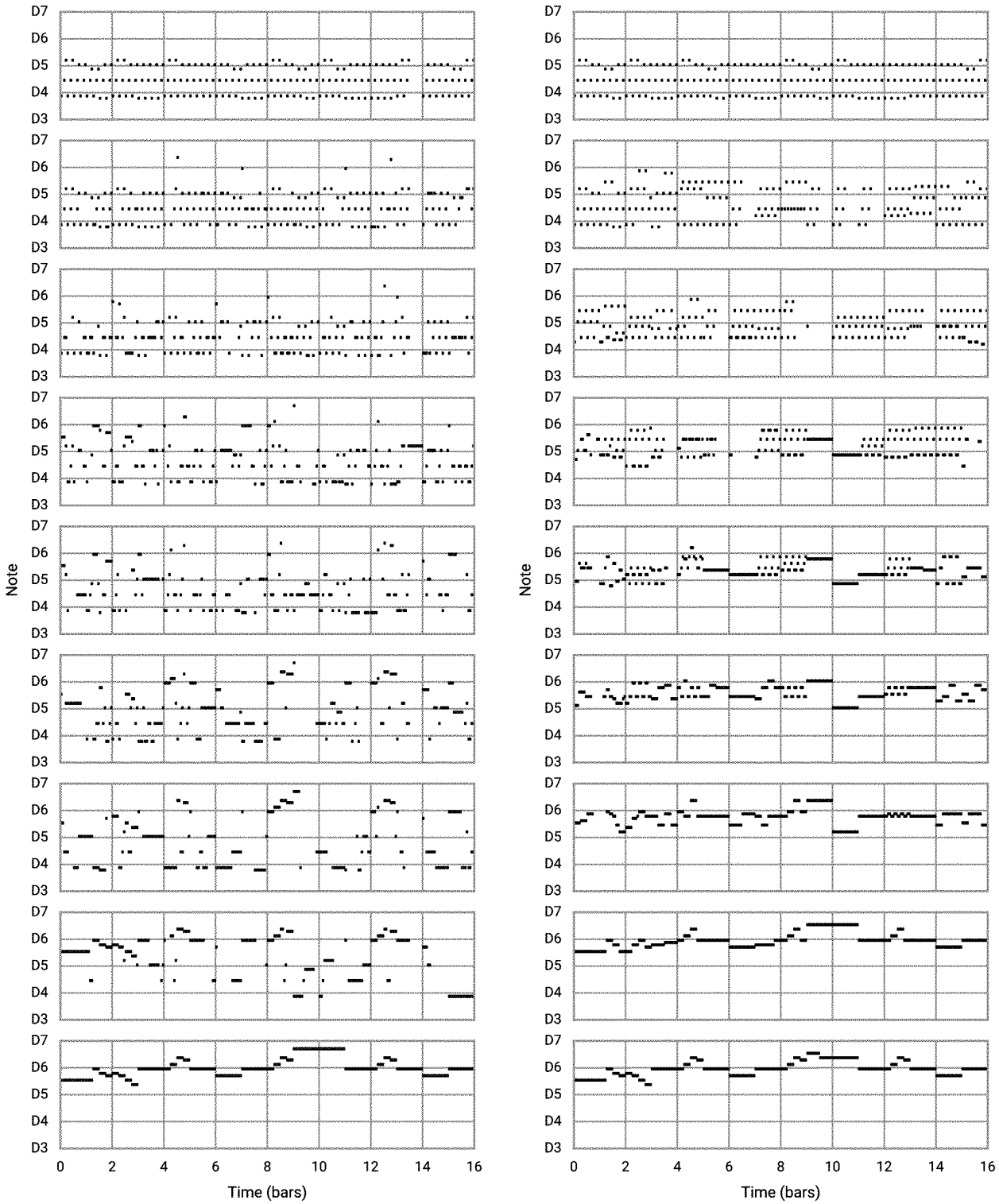

Fig. 5.17 Comparison of interpolations between the top and the bottom melodies by (left) interpolating in the data (melody) space and (right) interpolating in the latent space and decoding it into a melody. Reproduced from [149] with the permission of the authors

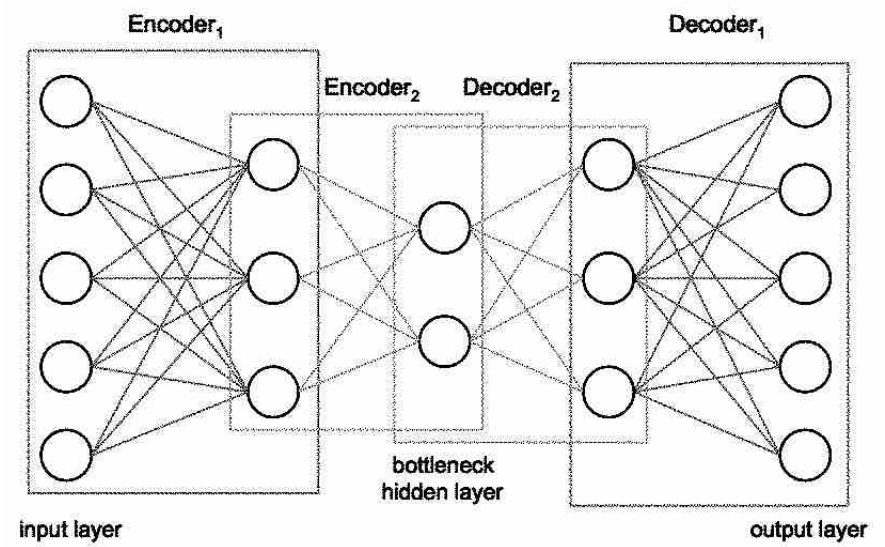

Fig. 5.18 A 2-layer stacked autoencoder architecture, resulting in a 4-layer full architecture

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 (an example will be introduced in Section 6.9.7.1). They are also useful for music generation, as we will see in Section 6.3.1. This is because the *innermost hidden layer*, sometimes named the *bottleneck hidden layer*, provides a compact and high-level encoding (embedding) as a seed for generation (by the chain of decoders).

5.10 Restricted Boltzmann Machine (RBM)

A *restricted Boltzmann machine* (RBM) [75] is a *generative stochastic* artificial neural network that can learn a *probability distribution* over its set of inputs. Its name comes from the fact that it is a restricted (constrained) form[52] of a (general) *Boltzmann machine* [76], named after the *Boltzmann distribution* in statistical mechanics, which is used in its sampling function. The architectural restrictions of an RBM (see Figure 5.19) are that

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

An RBM bears some similarity in spirit and objective to an autoencoder. However, there are some important differences:

- an RBM has *no ouput* – the input also acts as the output;
- an RBM is *stochastic* (and therefore *not deterministic*, as opposed to a feedforward network or an autoencoder);
- an RBM is trained in an *unsupervised learning* manner, with a specific algorithm (named *contrastive divergence*, see Section 5.10.1), whereas an autoencoder is trained using a standard supervised learning method, with the same data as input and output; and

[52] Which actually makes RBM practical, as opposed to the general form, which besides its interest suffers from a learning scalability limitation.

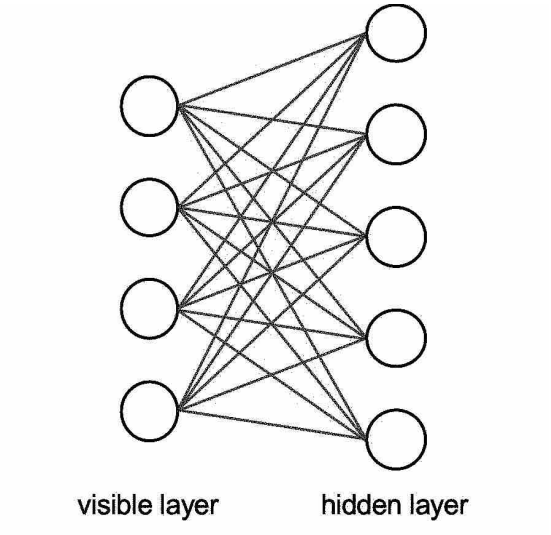

Fig. 5.19 Restricted Boltzmann machine (RBM) architecture

- the values manipulated are *booleans*[53].

RBMs became popular after Hinton designed a specific fast learning algorithm for them, named *contrastive divergence* [73], and used them for *pre-training* deep neural networks [40] (see Section 5.1).

An RBM is an architecture dedicated to learning distributions. Moreover, it can learn efficiently from only a few examples. For musical applications, this is interesting for learning (and generating) chords, as the combinatorial nature of possible notes forming a chord is large and the number of examples is usually small. We will see an example of such an application in Section 6.3.2.3.

5.10.1 Training

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

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

We will not detail here the learning technique behind RBMs, see, for example, [58, Section 20.2]. Note that the reconstruction process is different than that for autoencoders, which is based on regression, and is a case of *generative learning*[54].

[53] Although there are extensions with Multinoulli (categorical) or continuous values, see Section 5.10.3.

[54] See, for example, a nice introduction in [131].

5.10.2 Sampling

After the training phase has been completed, 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*[55] until convergence. To this end, hidden nodes and visible nodes are alternately updated (as during the training phase).

In practice, convergence is reached when the energy stabilizes. The *energy* of a *configuration* (the pair of visible and hidden layers) is expressed[56] in the Equation 5.17, where

$$E(v,h) = -a^{\mathrm{T}}v - b^{\mathrm{T}}h - v^{\mathrm{T}}Wh \tag{5.17}$$

- 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, are the bias weights for visible and hidden nodes.

5.10.3 Types of Variables

Note that there are actually three possibilities for the nature of RBM variables (units, visible or hidden):

- *Boolean* or *Bernoulli* – this is the case of standard RBMs, in which units (visible and hidden) are Boolean, with a *Bernoulli distribution* (see [58, Section 3.9.2]);
- *multinoulli* – an extension with *multinoulli* units[57], i.e. with more than two possible discrete values; and
- *continuous* – another extension with continuous units, taking arbitrary real values (usually within the $[0,1]$ range). An example is the C-RBM architecture analyzed in Section 6.9.5.1.

5.11 Recurrent Neural Network (RNN)

A *recurrent neural network* (RNN) is a feedforward neural network extended with *recurrent connexions* in order to learn series of items (e.g., a melody as a sequence of notes). The input of the RNN is an element x_t[58] of the sequence, where t represents the *index* or the *time*, and the expected output is next element x_{t+1}. In other words the RNN will be trained to predict the next element of a sequence.

In order to do so, the output of the hidden layer (h_t) *reenters* itself as an additional input (with a specific corresponding weight matrix). This way, the RNN can learn, not only based on the *current* item but also on its *previous* own state, and thus, recursively, on the whole of the previous sequence. Therefore, an RNN can learn sequences, notably *temporal sequences*, as in the case of musical content.

An example of RNN (with two hidden layers) is shown in Figure 5.20. Recurrent connexions are signaled with a solid square, in order to distinguish them from standard connexions. The unfolded version of the visual representation is in Figure 5.21, with a new diagonal axis representing the time dimension, in order to illustrate the previous step

[55] More precisely *Gibbs sampling (GS)*, see [98]. Sampling will be introduced in Section 6.3.2.1.

[56] For more details, see, for example, [58, Section 16.2.4].

[57] As explained by Goodfellow *et al.* in [58, Section 3.9.2]: ""Multinoulli" is a term that was recently coined by Gustavo Lacerdo and popularized by Murphy in [128]. The multinoulli distribution is a special case of the multinomial distribution. A multinomial distribution is the distribution over vectors in $\{0,...,n\}^k$ representing how many times each of the k categories is visited when n samples are drawn from a multinoulli distribution. Many texts use the term "multinomial" to refer to multinoulli distributions without clarifying that they refer only to the $n = 1$ case."

[58] This x_t notation – or sometimes s_t to stress the fact that it is a sequence – is very common but unfortunately introduces possible confusion with the notation of x_i as the ith input variable. The context – recurrent versus nonrecurrent network – usually helps to discriminate, as well as the use of the letter t (for time) as the index. An example of an exception is the RNN-RBM system analyzed in Section 6.8.1, which uses the $x^{(t)}$ notation.

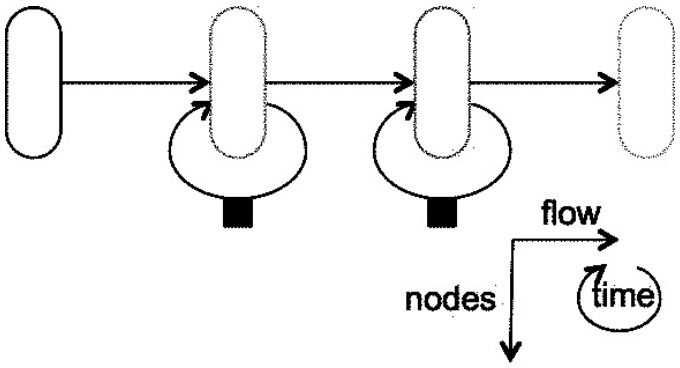

Fig. 5.20 Recurrent neural network (folded)

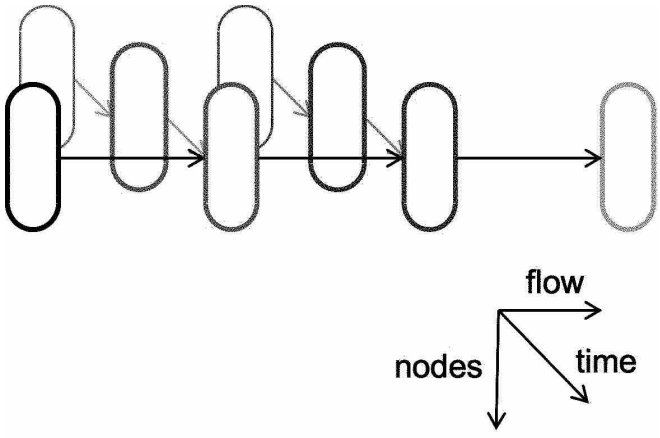

Fig. 5.21 Recurrent neural network (unfolded)

value of each layer (in thinner and lighter color). Note that, as for standard connexions (shown in yellow solid lines), recurrent connexions (shown in purple dashed lines) fully connect all previous step nodes to its current step nodes (with a specific weight matrix), as illustrated in Figure 5.22.

An RNN can learn a probability distribution over a sequence by being trained to predict the next element at time step t in a sequence as being the conditional probability $p(s_t|s_{t-1},...,s_1)$, also notated as $p(s_t|s_{<t})$, that is the probability $p(s_t)$ given all previous elements generated s_1, s_2, ... , s_{t-1}. In summary, recurrent networks (RNNs) are good at learning sequences and therefore are routinely used for natural text processing and for music generation.

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

5.11.1 Architectural Notation

A more frequent architectural notation for an RNN is actually showing the flow upwards and time rightwards, see the folded version (of an RNN with only one hidden layer) in Figure 5.23 and the unfolded version in Figure 5.24, with h_t being the value of the hidden layer at step t, and x_t and y_t being the values of the input and output at step t.

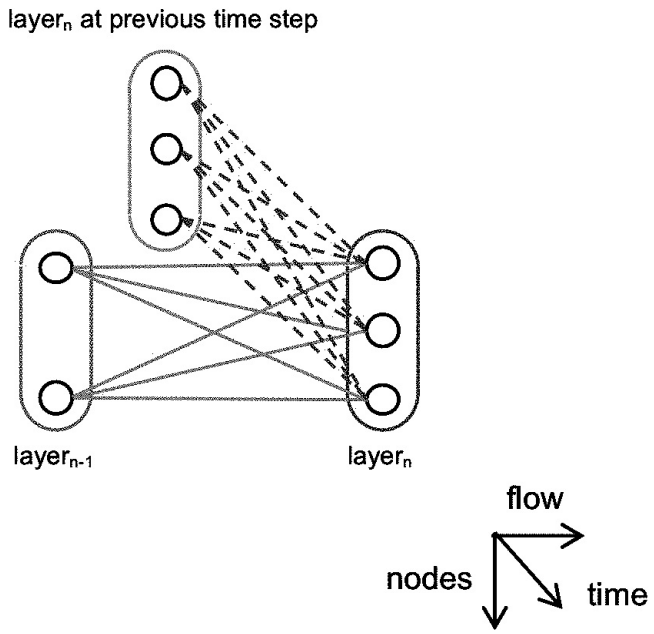

Fig. 5.22 Standard connexions vs recurrent connexions (unfolded)

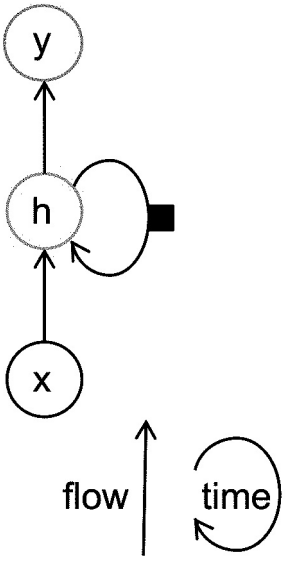

Fig. 5.23 Recurrent neural network (folded)

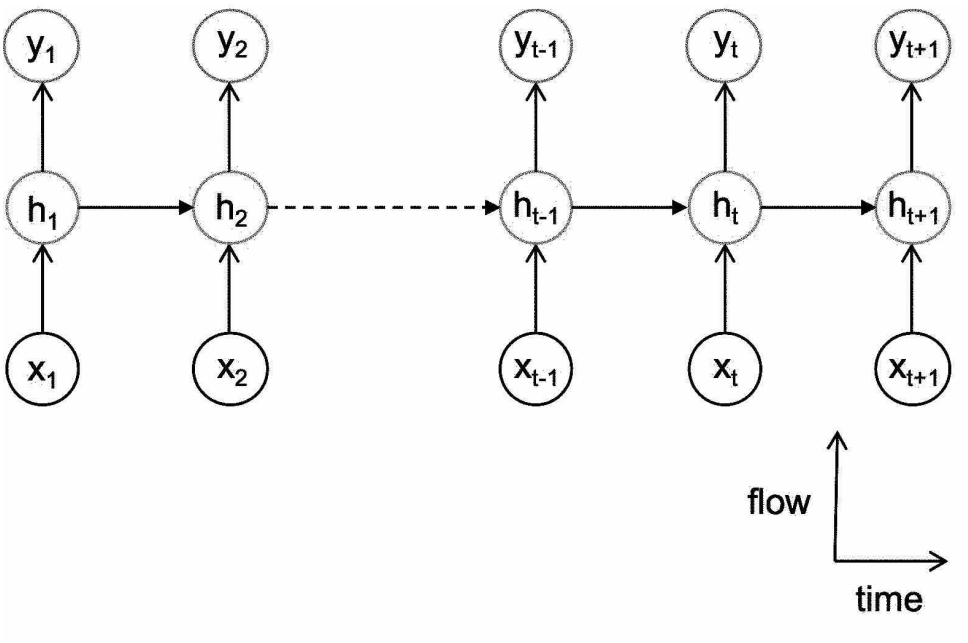

Fig. 5.24 Recurrent neural network (unfolded)

5.11.2 Training

A recurrent network is not trained in exactly the same manner as a feedforward network. The idea is to present an example element of a sequence (e.g., a note within a melody) as the input x_t and the next element of the sequence (the next note) x_{t+1} as the output y_t. This will train the recurrent network to predict the next element of the sequence. In practice, an RNN is very rarely trained element by element but with a sequence as an input and the same sequence shifted left by one step/item as the output. See an example in Figure 5.25[59]. Therefore, the recurrent network will learn to predict[60] the next element for all successive elements of the sequence.

The backpropagation algorithm to compute gradients for feedforward networks, introduced in Section 5.8.6, has been extended into a *backpropagation through time* (BPTT) algorithm for recurrent networks. The intuition is in unfolding the RNN through time and considering an ordered sequence of input-output pairs, but with every unfolded copy of the network sharing the same parameters, and then applying the standard backpropagation algorithm. More details may be found, for example, in [58, Section 10.2.2].

Note that training a recurrent network is usually considered as a case of supervised learning as, for each item, the next item is presented as the expected prediction. But we could also consider this as a case of unsupervised learning as the examples do not contain any additional label information (effective value or class to be predicted) apart from the recurrent information about the next item, which is *intrinsically* present within a sequence.

5.11.3 Long Short-Term Memory (LSTM)

Recurrent networks suffered from a training problem caused by the difficulty of estimating gradients because in back-propagation through time recurrence brings repetitive multiplications, and could thus lead to over *amplify* or *minimize*

[59] The end of the sequence is marked by a special symbol.

[60] Pictured as dashed arrows.

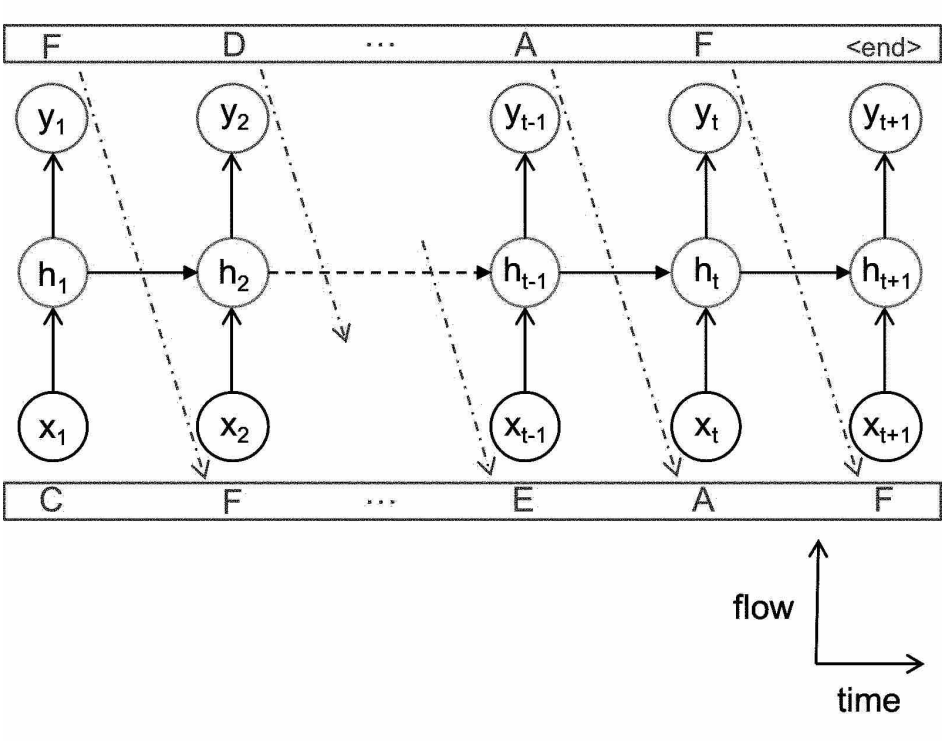

Fig. 5.25 Training a recurrent neural network

effects[61]. This problem has been addressed and resolved by the *long short-term memory* (LSTM) architecture, proposed by Hochreiter and Schmidhuber in 1997 [77]. As the solution has been quite effective, LSTM has become the de facto standard for recurrent networks[62].

The idea behind LSTM is to secure information in memory *cells*, within a *block*[63], protected from the standard data flow of the recurrent network. Decisions about *writing* to, *reading* from and *forgetting* (*erasing*) the values of cells within a block are performed by the opening or closing of *gates* and are expressed at a distinct control level (*meta-level*), while being learnt during the training process. Therefore, each gate is modulated by a *weight* parameter, and thus is suitable for backpropagation and standard training process. In other words, 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 an LSTM cell in Figure 5.26. We will not further detail here the inner mechanism of an LSTM cell (and block) because we may consider it here as a *black box* (please refer to, for example, the original article [77]).

Note that a more general model of memory with access customized through training has recently been proposed: neural Turing machines (NTM) [61]. In this model, memory is global and has *read* and *write operations* with differentiable controls, and thus is subject to learning through backpropagation. The memory to be accessed, specified by *location* or by *content*, is controlled via an *attention* mechanism (to be addressed in Section 5.16.3).

[61] This has been coined as the *vanishing or exploding gradient problem* and also as the *challenge of long-term dependencies* (see, for example, [58, Section 10.7]).

[62] Although, there are a few subsequent but similar proposals, such as *gated recurrent units* (GRUs). See a comparative analysis of LSTM and GRU in [22].

[63] Cells within the same block *share* input, output and forget gates. Therefore, 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* [77].

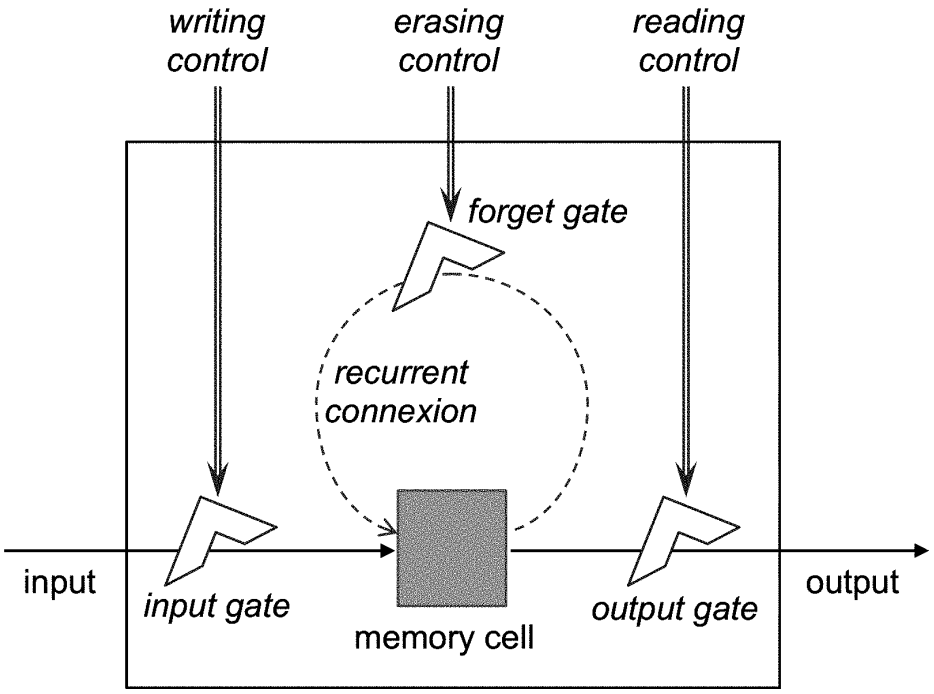

Fig. 5.26 LSTM architecture (conceptual)

5.12 Convolutional Architectural Pattern

Convolutional neural network (CNN or ConvNet) architectures for deep learning have become common place for image applications. The concept was originally inspired by both a model of human vision and the *convolution* mathematical operator[64]. It has been carefully adapted to neural networks and improved by Le Cun, at first for handwritten character and object recognition [102]. This resulted in efficient and accurate architectures for pattern recognition, exploiting the spatial local *correlation* present in natural images.

5.12.1 Principles

The basic idea[65] is to

- *slide* a matrix (named a *filter*, a *kernel* or a *feature detector*) through the entire image (seen as the input matrix); and

[64] In mathematics, a convolution is a mathematical operation on two functions sharing the same domain (usually noted $f * g$) that produces a third function which is the integral (or the sum in the discrete case – the case of images made of pixels) of the pointwise multiplication of the two functions varying within the domain in an opposing way. In the case of a continuous domain $[low \;\; high]$:

$$(f * g)(x) = \int_{low}^{high} f(x-t)g(t)dt$$

In the discrete case:

$$(f * g)(n) = \sum_{m=low}^{high} f(n-m)g(m)$$

[65] Inspired by the nice intuitive explanation provided by Karn in [90]. For more technical details see, for example, [108] or [58, Chapter 9].

- for each mapping position:
  - compute the dot product of the filter with each mapped portion of the image; and
  - then sum up all elements of the resulting matrix;
- resulting in a new matrix (composed of the different sums for each sliding/mapping position), named *convolved feature*, or also *feature map*.

The size of the feature map is controlled by three hyperparameters:

- *depth* – the number of filters used;
- *stride* – the number of pixels by which we slide the filter matrix over the input matrix; and
- *zero-padding* – the padding of the input matrix with zeros around its border[66].

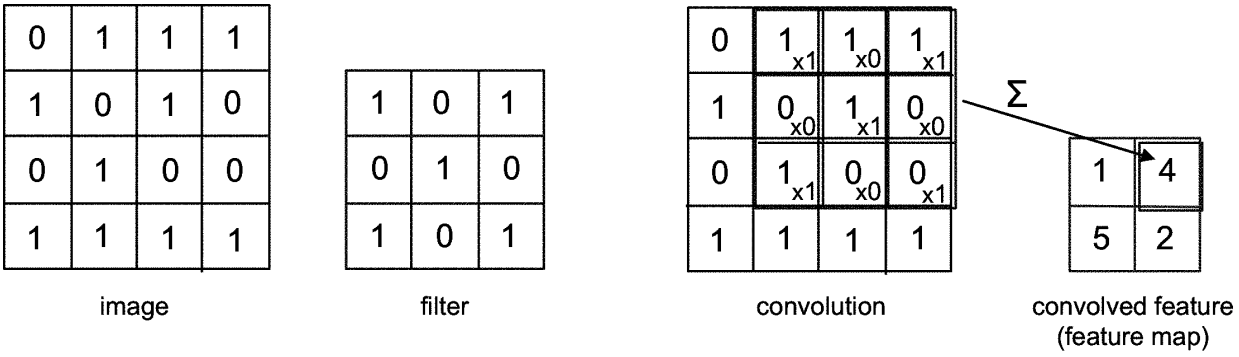

Fig. 5.27 Convolution, filter and feature map. Inspired by Karn's data science blog post [90]

An example is illustrated in Figure 5.27 with some simple settings: depth = 1, stride = 1 and no zero-padding. Various filter matrixes can be used with different objectives, such as detection of different features (e.g., edges or curves) or other operations such as sharpening or blurring.

The parameter sharing used by the convolution (because of the shared fixed filter) brings the important property of *equivariance* to translation, i.e. a motif in an image can be detected independently of its location [58, Chapter 9].

5.12.2 Stages

A convolution usually consists of three successive *stages*:

- a *convolution stage*, as described in Section 5.12.1;
- a *nonlinear rectification stage*, sometimes named detector stage, which applies a nonlinear operation, usually ReLU; and
- a *pooling stage*, also named *subsampling*, to reduce the dimensionality.

[66] Zero-padding allows mapping of the filter up to the borders of the image. It also avoids shrinking the representation, which otherwise would be problematic when using multiple consecutive convolutional layers [58, Section 9.5].

5.12.3 Pooling

The motivation for *pooling* is to reduce the dimensionality of each feature map while retaining significant information. Operations used for pooling are, for example, max, average and sum. In addition to reducing the dimensionality of data, pooling brings the important property of the *invariance* to small transformations, distortions and translations in the input image. This provides an overall robustness to the processing [90]. Like convolution, pooling has hyperparameters to control the process. A simple example of max pooling with stride = 2 is illustrated in Figure 5.28.

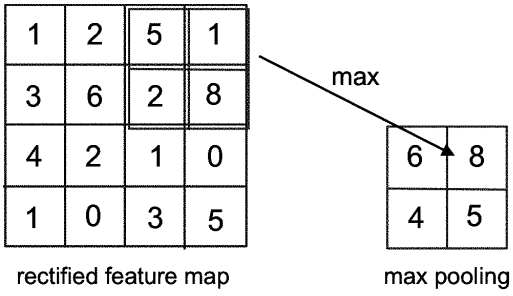

Fig. 5.28 Pooling. Inspired by Karn's data science blog post [90]

5.12.4 Multilayer Convolutional Architecture

A typical example of a convolutional architecture with successive layers – each one including the three stages of convolution, nonlinearity and pooling – is illustrated in Figure 5.29. The final layer is a fully connected layer, like in standard feedforward networks, and typically ends up in a softmax in order to classify image types.

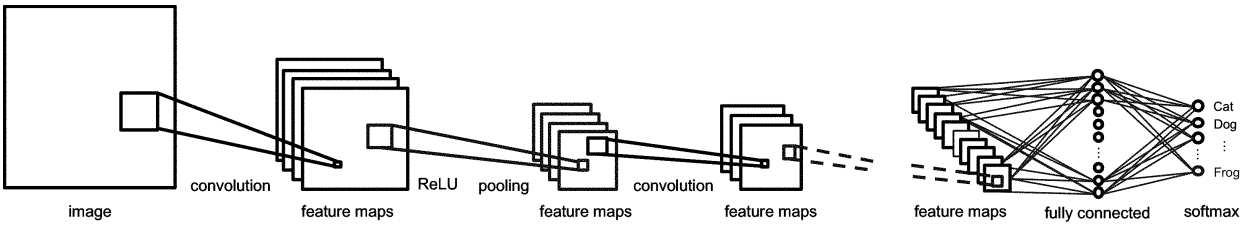

Fig. 5.29 Convolutional deep neural network architecture. Inspired by Karn's data science blog post [90]

Note that a convolution is an *architectural pattern*, as it may be applied internally to almost any architecture listed.

5.12.5 Convolution over Time

For musical applications, it could be interesting to apply convolutions to the *time dimension*[67], in order to model temporally invariant motives. Therefore, the convolution operation will share parameters across time [58, page 374],

[67] This approach is actually the basis for *time-delay neural networks* [99].

like for RNNs[68]. However, the sharing of parameters is *shallow*, as it applies only to a small number of temporal neighboring members of the input, in contrast to RNNs that share parameters in a *deep* way, for *all* time steps. RNNs are indeed much more frequent than convolutional networks for musical applications.

That said, we have noticed the recent occurrence of some convolutional architectures as an alternative to RNN architectures, following the pioneering WaveNet architecture for audio [176], described in Section 6.9.3.2. WaveNet presents a stack of causal convolutional layers, somewhat analogous to recurrent layers. Another example is the C-RBM architecture, described in Section 6.9.5.1.

If we now consider the *pitch dimension*, in most cases pitch intervals are not considered invariants, and thus convolutions should not *a priori* apply to the pitch dimension[69].

This issue of convolution versus recurrence (recurrent networks) for musical applications will be further discussed in Section 8.2.

5.13 Conditioning Architectural Pattern

The idea of a *conditioning* (sometimes also named *conditional*) architecture is to parametrize the architecture based on some extra *conditioning* information, which could be arbitrary, e.g., a class label or data from other modalities. The objective is to have some control over the data generation process. Examples of conditioning information are

- a *bass line* or a *beat structure* in the rhythm generation system to be described in Section 6.9.3.1;
- a *chord progression* in the MidiNet system to be described in Section 6.9.3.3;
- some *positional constraints on notes* in the Anticipation-RNN system to be described in Section 6.9.3.5; and
- a *musical genre* or an *instrument* in the WaveNet system to be described in Section 6.9.3.2.

In practice, the conditioning information is usually fed into the architecture as an additional and specific input layer, shown in purple in Figure 5.30.

The conditioning layer could be

- a simple input layer. An example is a tag specifying a musical genre or an instrument in the WaveNet system to be described in Section 6.9.3.2; or
- some output of some architecture, being
  - the same architecture, as a way to condition the architecture on some history[70]. An example is the MidiNet system to be described in Section 6.9.3.3, in which history information from previous measure(s) is injected back into the architecture; or
  - another architecture. An example is the DeepJ system to be described in Section 6.9.3.4, in which two successive transformation layers of a style tag produce an embedding used as the conditioning input.

In the case of *conditioning* a time-invariant architecture – recurrent or convolutional over time – there are two options

- *global conditioning* – if the conditioning input is shared for all time steps; and
- *local conditioning* – if the conditioning input is specific to each time step.

The WaveNet architecture, which is convolutional over time (see Section 5.12.5), offers the two options, as will be analyzed in Section 6.9.3.2.

[68] Indeed, RNNs are invariant in time, as remarked in [87].

[69] An exception is Johnson's architecture [87], analyzed in Section 6.8.2, which explicitly looks for invariance in pitch (although this seems to be a rare choice) and accordingly uses an RNN over the pitch dimension.

[70] This is close in spirit to a recurrent architecture (RNN).

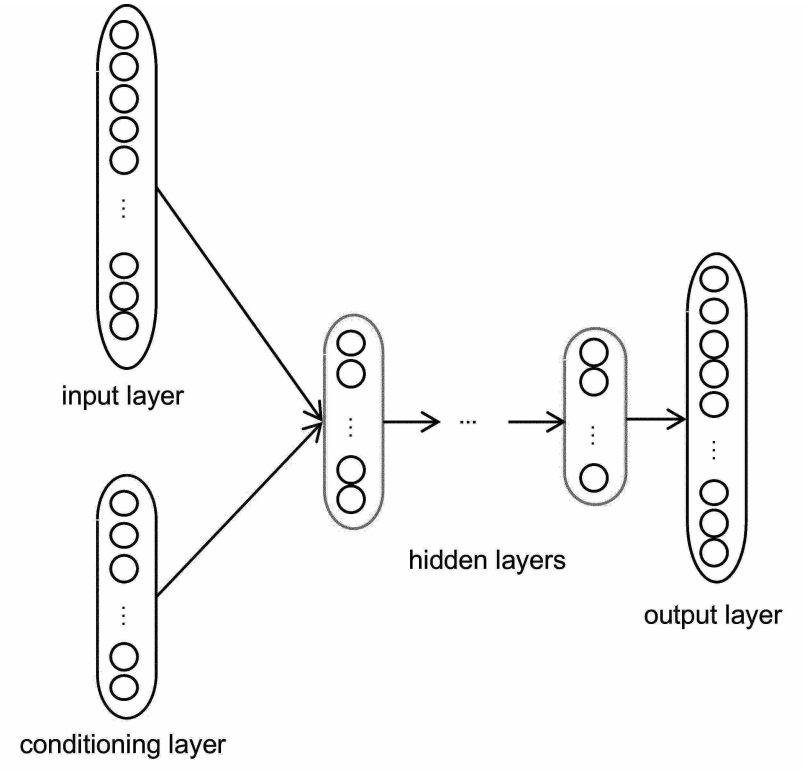

Fig. 5.30 Conditioning architecture

5.14 Generative Adversarial Networks (GAN) Architectural Pattern

A significant conceptual and technical innovation was introduced in 2014 by Goodfellow *et al.* with the concept of *generative adversarial networks* (GAN) [59]. The idea is to train simultaneously two neural networks[71], as illustrated in Figure 5.31:

- a *generative model* (or *generator*) G, whose objective is to transform a random noise vector into a synthetic (faked) *sample*, which resembles real samples drawn from a distribution of real images; and
- a *discriminative model* (or *discriminator*) D, which estimates the probability that a sample came from the real data rather than from the generator G[72].

This corresponds to a *minimax* two-player game, with one unique (final) solution[73]: G recovers the training data distribution and D outputs $1/2$ everywhere. The generator is then able to produce user-appealing synthetic samples from noise vectors. The discriminator may then be discarded.

The minimax relationship is defined in Equation 5.18.

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_{\text{Data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \tag{5.18}$$

- $D(x)$ represents the probability that x came from the real data (i.e. the *correct* estimation by D); and

[71] In the original version, two feedforward networks are used. But we will see that other networks may be used, e.g., recurrent networks in the C-RNN-GAN architecture (Section 6.9.2.4) and convolutional feedforward networks in the MidiNet architecture (Section 6.9.3.3).

[72] In some ways, a GAN represents an automated Turing test setting, with the discriminator being the evaluator and the generator being the hidden actor.

[73] It corresponds to the Nash equilibrium of the game. In game theory, the intuition of a Nash equilibrium is a solution where no player can benefit by changing strategies while the other players keep theirs unchanged, see, for example, [133].

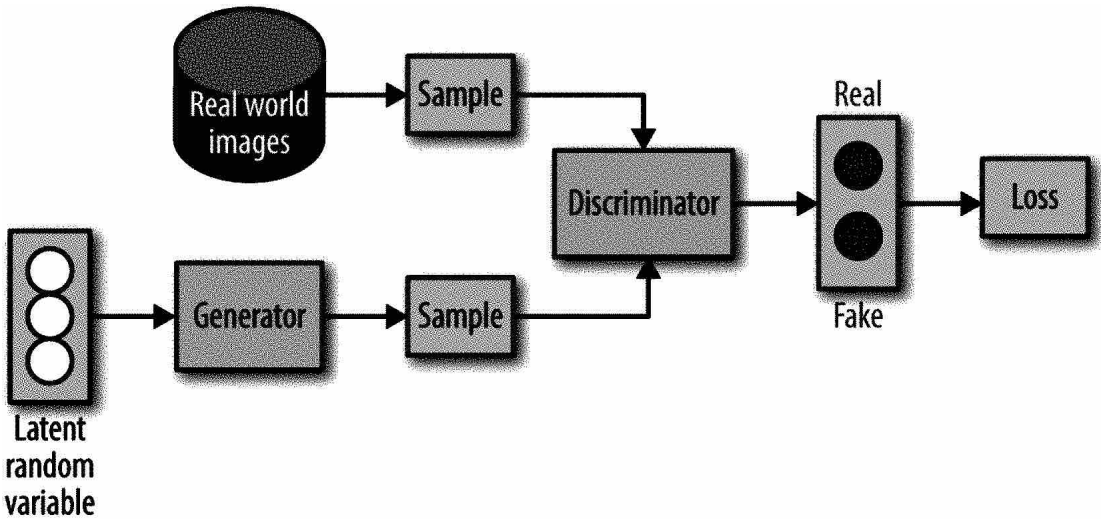

Fig. 5.31 Generative adversarial networks (GAN) architecture. Reproduced from [144] with the permission of O'Reilly Media

- $\mathbb{E}_{x\sim p_{\text{Data}}}[\log D(x)]$ is the expectation[74] of $\log D(x)$ with respect to x being drawn from the real data.

  It is thus D's objective to estimate correctly *real data*, that is to maximize the $\mathbb{E}_{x\sim p_{\text{Data}}}[\log D(x)]$ term.

- $D(G(z))$ represents the probability that $G(z)$ came from the real data (i.e. the *uncorrect* estimation by D);
- $1-D(G(z))$ represents the probability that $G(z)$ did not come from the real data, i.e. that it was generated by G (i.e. the *correct* estimation by D); and
- $\mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$ is the expectation of $\log(1-D(G(z)))$ with respect to $G(z)$ being produced by G from z random noise.

It is thus also D's objective to estimate correctly *synthetic data*, that is to maximize the $\mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$ term.

In summary, it is D's objective to estimate correctly both *real data* and *synthetic data* and thus to maximize both $\mathbb{E}_{x\sim p_{\text{Data}}}[\log D(x)]$ and $\mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$ terms, i.e. to maximize $V(G,D)$. On the opposite side, G'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 motivations for GAN was for classification tasks to prevent adversaries from manipulating deep networks to force misclassification of inputs (this vulnerability is analyzed in detail in [167]). However, from the perspective of content generation (which is our interest), GAN improves the generation of samples, which become hard to distinguish from the actual corpus examples.

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[75]. An example of the use of GAN for generating music is the MidiNet system, to be described in Section 6.9.3.3.

---

[74] The expectation, or expected value, of some function $f(x)$ with respect to a probability distribution $p(x)$, usually notated as $\mathbb{E}_{x\sim p}$, is the average (mean) value that f takes on when x is drawn from p.

[75] In that respect, generation from a GAN has some similarity with generation by decoding hidden layer variables of a variational autoencoder (Section 5.9.2), as in both cases generation is done from latent variables. An important difference is that, by construction, a variational autoencoder is representative of the whole dataset that it has learnt, that is, for any example in the dataset, there is at least one setting of the latent variables which causes the model to generate something very similar to that example [33]. A GAN does not offer such guarantee and does not offer a smooth generation control interface over the latent space (by, e.g., interpolation or attribute arithmetics, see Section 5.9.2), but it can usually generate better quality (better resolution) images than a variational autoencoder [115]. Note that the resolution limitation for a VAE may be a problem too for audio generation of music, but it appears *a priori* less a direct concern in the case of symbolic generation of music.

5.14.1 Challenges

Training based on a minimax objective is known to be challenging to optimize [190], with a risk of nonconverging oscillations. Thus, careful selection of the model and its hyperparameters is important [58, page 701]. There are also some newer techniques, such as *feature matching*[76], among others, to improve training [154].

A recent proposed alternative both to GANs and to autoencoders is *generative latent optimization* (GLO) [8]. 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, and as a discriminator-less GAN. It can also be used, as for a VAE (variational autoencoder) introduced in Section 5.9.2, to control generation by exploring the latent space. GLO has been tested on images but not yet on music. This potential new approach needs more evaluation.

5.15 Reinforcement Learning

Reinforcement learning (RL) may appear at first glance to be outside of our interest in deep learning architectures, as it has distinct objectives and models. However, the two approaches have recently been combined. The first move, in 2013, was to use deep learning architectures to efficiently implement reinforcement learning techniques, resulting in *deep reinforcement learning* [122]. The second move, in 2016, is directly related to our concerns, as it explored the use of reinforcement learning to control music generation, resulting in the RL-Tuner architecture [86] to be described in Section 6.9.6.1.

Let us start with a reminder of the basic concepts of reinforcement learning, illustrated in Figure 5.32

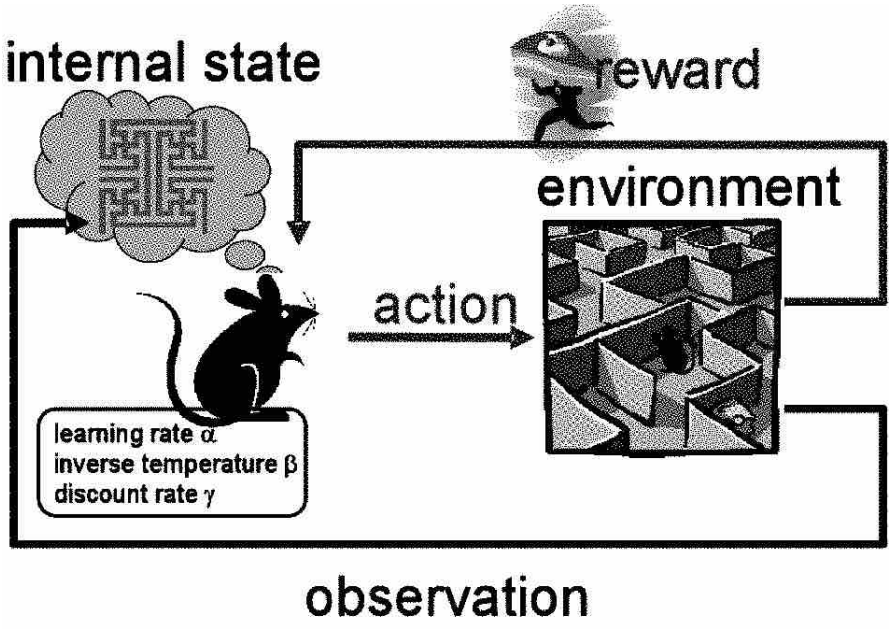

Fig. 5.32 Reinforcement learning – conceptual model. Reproduced from [35] with the permission of SAGE Publications, Inc./Corwin

- an *agent* within an *environment* sequentially selects and performs *actions* in an environment;
- where each action performed brings it to a new *state*;

[76] Feature matching changes the objective for the generator (and accordingly its cost function) to minimize the statistical difference between the features of the real data and the generated samples, see more details in [154].

- the agent receives a *reward* (*reinforcement signal*), which represents the *fitness* of the action to the environment (current situation);
- the objective of the agent being to learn a near optimal *policy* (sequence of actions) in order to maximize its *cumulated rewards* (named its *gain*).

Note that the agent does not know beforehand the model of the environment and the reward, thus it needs to balance between *exploring* to learn more and *exploiting* (what it has learned) in order to improve its gain – this is the *exploration exploitation dilemma*.

There are many approaches and algorithms for reinforcement learning (for a more detailed presentation, please refer, for example, to [89]). Among them, *Q-learning* [183] turned out to be a relatively simple and efficient method, and thus is 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, for an agent choosing actions optimally (i.e. by following the optimal policy π^*). The agent will manage a table, called the *Q-table*, with values corresponding to all possible pairs. As the agent explores the environment, the table is incrementally updated, with estimates becoming more accurate.

A recent combination of reinforcement learning (more specifically Q-learning) and deep learning, named *deep reinforcement learning*, has been proposed [122] in order to make learning more efficient. As the Q-table could be huge[77], 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* [178] *decouples* the *action selection* from the *evaluation*, in order to avoid value overestimation. The task of the first network, named the Target Q-Network, is to estimate the gain (Q), while the task of the Q-Network is to select the next action.

Reinforcement learning appears to be a promising approach for incremental adaptation of the music to be generated, e.g., based on the *feedback* from listeners (this issue will be addressed in Section 6.15). Meanwhile, a significant move has been made in using reinforcement learning to inject some control into the generation of music by deep learning architectures, through the reward mechanism, as described in Section 6.9.6.

5.16 Compound Architectures

Often *compound* architectures are used. Some cases are *homogeneous* compound architectures, combining various instances of the same architecture, e.g., a stacked autoencoder (see Section 5.9.3), and most cases are *heterogeneous* compound architectures, combining various types of architectures, e.g., an RNN Encoder-Decoder which combines an RNN and an autoencoder, see Section 5.16.3.

5.16.1 Composition Types

We will see that, from an architectural point of view, various types of composition[78] may be used:

- *Composition* – at least two architectures, of the same type or of different types, are combined, such as
  - a bidirectional RNN (Section 5.16.2) combining two RNNs, forward and backward in time; and
  - the RNN-RBM architecture (Section 5.16.5) combining an RNN architecture and an RBM architecture.
- *Refinement* – one architecture is *refined* and *specialized* through some additional constraint(s)[79], such as

[77] Because of the high combinatorial nature when the number of possible states and possible actions is huge.

[78] We are taking inspiration from concepts and terminology in programming languages and software architectures [157], such as *refinement*, *instantiation*, *nesting* and *pattern* [50].

[79] Both cases are refinements of the standard autoencoder architecture through additional constraints, in practice adding an extra term onto the cost function.

  - a sparse autoencoder architecture (Section 5.9.1); and
  - a variational autoencoder (VAE) architecture (Section 5.9.2).

- *Nested* – one architecture is nested into the other one, for example

  - a stacked autoencoder architecture (Section 5.9.3); and
  - the RNN Encoder-Decoder architecture (Section 5.16.3), where two RNN architectures are nested within the encoder and decoder parts of an autoencoder, which we could therefore also notate as Autoencoder(RNN, RNN).

- *Pattern instantiation* – an architectural pattern is instantiated onto a given architecture(s), for example

  - the C-RBM architecture (Section 6.49) that instantiates the convolutional architectural pattern onto an RBM architecture, which we could notate as Convolutional(RBM);
  - the C-RNN-GAN architecture (Section 6.9.2.4), where the GAN architectural pattern is instantiated onto an RNN architecture, which we could notate as GAN(RNN, RNN); and
  - the Anticipation-RNN architecture (Section 6.9.3.5) that instantiates the conditioning architectural pattern onto an RNN with the output of another RNN as the conditioning input, which we could notate as Conditioning(RNN, RNN).

5.16.2 Bidirectional RNN

Bidirectional recurrent neural networks (bidirectional RNNs) were introduced by Schuster and Paliwal [156] to handle the case when the prediction depends not only on the previous elements but also on the *next* elements, as for instance with speech recognition. In practice, a bidirectional RNN combines[80]

- a first RNN that moves *forward* through time and begins from the *start* of the sequence; and
- a second symmetric RNN that moves *backward* through time and begins from the *end* of the sequence.

The output y_t of the bidirectional RNN at step t combines

- the output h_t^f at step t of the hidden layer of the "forward RNN", and
- the output h_{N-t+1}^b at step $N-t+1$ of the hidden layer of the "backward RNN".

An illustration is in Figure 5.33. Examples of use are

- the C-RNN-GAN architecture (Section 6.9.2.4) that encapsulates a bidirectional RNN into the discriminator of a GAN; and
- the MusicVAE architecture (Section 6.11.1) that encapsulates a bidirectional RNN into the encoder of a VAE (variational autoencoder).

5.16.3 RNN Encoder-Decoder

The idea of encapsulating two identical recurrent networks (RNNs) into an autoencoder, named the *RNN Encoder-Decoder*[81], was initially proposed in [18] as a technique to encode a variable-length sequence learnt by a recurrent network into another variable-length sequence produced by another recurrent network[82]. The motivation and application target is the translation from one language to another, resulting in sentences of possibly different lengths.

The idea is to use a fixed-length vector representation as a *pivot* representation between an encoder and a decoder architecture, see the illustration in Figure 5.34. The hidden layer(s) h_t^e of the encoder will act as a memory which

[80] See more details in [156].

[81] We could also notate it as Autoencoder(RNN, RNN).

[82] This is named *sequence-to-sequence learning* [165].

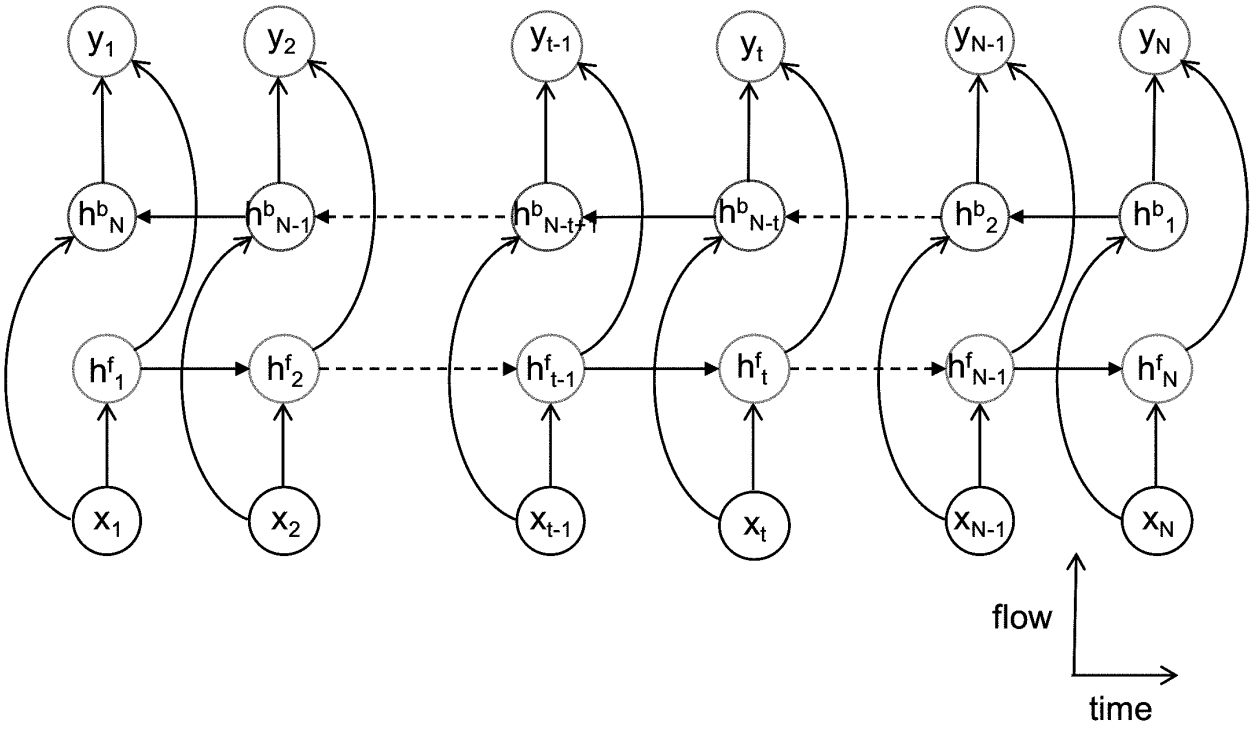

Fig. 5.33 Bidirectional RNN architecture

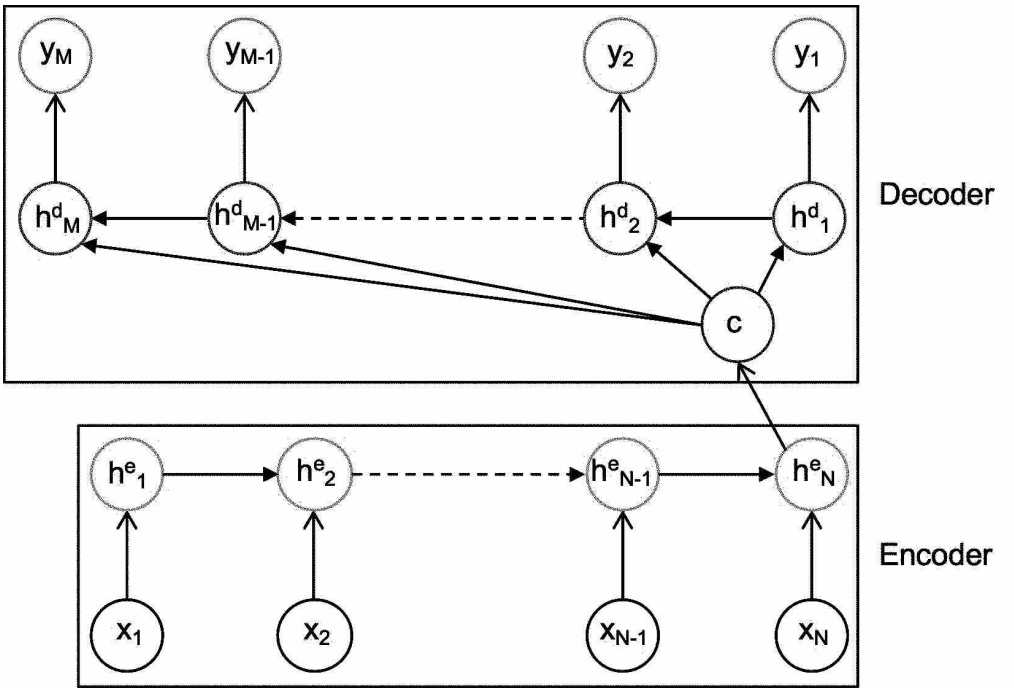

Fig. 5.34 RNN Encoder-Decoder architecture. Inspired from [18]

- iteratively accumulates information about some input sequence of length N, while reading its successive x_t elements[83], resulting in a final state h_N^e;
- which is passed to the decoder as the summary c of the whole input sequence; and
- the decoder then iteratively generates the output sequence of length M, by predicting the next item y_t given its hidden state h_t^d and the summary (as a conditioning additional input) c[84].

The two components of the RNN Encoder-Decoder are jointly trained to minimize the cross-entropy between input and output. See in Figure 5.35 the example of the Audio Word2Vec architecture for processing audio phonetic structures [23].

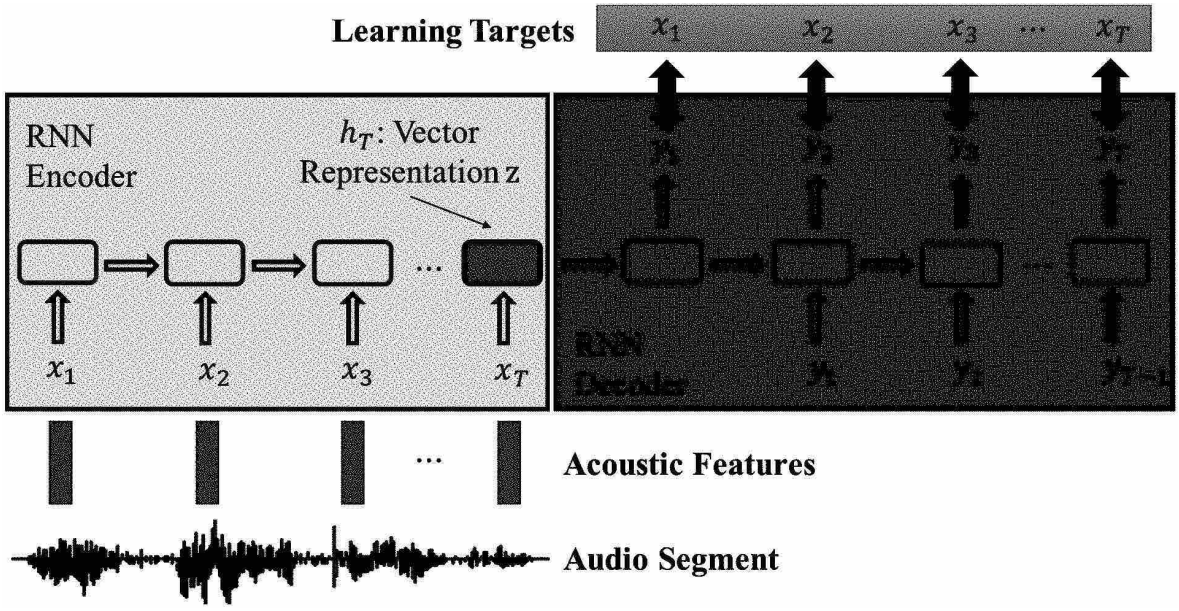

Fig. 5.35 RNN Encoder-Decoder audio Word2Vec architecture. Reproduced from [23] with the permission of the authors

One limitation of the RNN Encoder-Decoder approach is the difficulty for the summary to memorize very long sequences[85]. Two possible directions are

- using an *attention mechanism*; and
- using a *hierarchical model*, as proposed in the MusicVAE architecture, to be introduced in Section 6.11.1.

The idea of an attention mechanism is to focus at each time step on specific elements of the input sequence. This is modeled by weighted connexions onto the sequence elements (or onto the sequence of hidden units). Therefore it is differentiable and subject to backpropagation-based learning at a meta-level, as with LSTM gate control described in Section 5.11.3. For more details, see, for example, [58, Section 12.4.5.1].

[83] The end of the sequence is marked by a special symbol, as when training an RNN, see Section 5.11.2.

[84] As noted by Goodfellow *et al.* in [58, Section 10.4], an alternative is to use the summary c only to initialize the initial hidden state of the decoder h_0^d. This is, for instance, the strategy chosen in the GLSR-VAE architecture described in Section 6.9.2.3.

[85] In text translation applications, sentences have a limited size.

5.16.4 Variational RNN Encoder-Decoder

An interesting development is a *variational* version of the RNN Encoder-Decoder, in other words a variational autoencoder (VAE) encapsulating two RNNs. We could notate it as Variational(Autoencoder(RNN, RNN)). The objective is to combine

- the *variational* property of the VAE for controlling the generation[86]; and
- the *sequence* generation property of the RNN.

Examples of its application to music generation will be introduced in Section 6.9.2.3.

5.16.5 Polyphonic Recurrent Networks

The RNN-RBM architecture, to be introduced in Section 6.8.1, combines an 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.

5.16.6 Further Compound Architectures

It is possible to further combine architectures that are already compound, for example

- the WaveNet architecture (Section 6.9.3.2), which is a conditioning convolutional feedforward architecture with some tag as the conditioning input, which we could notate as Conditioning(Convolutional(Feedforward), Tag); and
- the VRASH architecture (Section 6.9.3.6), which is a variational autoencoder encapsulating RNNs with the decoder being conditioned on history, which we could notate as Variational(Autoencoder(RNN, Conditioning(RNN, History))).

There are also some more specific (ad hoc) compound architectures, for example

- Johnson's Hexahedria architecture (Section 6.8.2), which combines two layers recurrent on the time dimension with two other layers recurrent on the pitch dimension, as an integrated alternative to the RNN-RBM architecture; and
- The DeepBach architecture (Section 6.13.2), which combines two feedforward architectures with two recurrent architectures.

5.16.7 The Limits of Composition

There is a natural tendency to explore possible combinations of different architectures with the hope of combining their respective features and merits. An example of a sophisticated compound architecture is the VRASH architecture (Section 6.9.3.6), which combines

- variational autoencoder;
- recurrent networks; and
- conditioning (on the decoder).

However, note that

[86] See Section 5.9.2.

- not all combinations make sense. For instance, recurrence and convolution over the time dimension would compete, as discussed in Section 5.12; and
- there is no guarantee that combining a maximal variety of types will make a sound and accurate architecture[87].

We will see in Chapter 6 that an important additional design dimension is the *strategy*, which governs how an architecture will process representations in order to reach a given objective with some expected properties (the *challenges*).

[87] As in the case of a good cook, whose aim is not to simply mix *all* possible ingredients but to discover original successful combinations.

Chapter 6
Challenges and Strategy

We are now reaching the core of this book. This chapter will analyze in depth how to apply the architectures presented in Chapter 5 to learn and generate music. We will first start with a naive, straightforward strategy, using the basic prediction task of a neural network to generate an accompaniment for a melody.

We will see that, although this simple direct strategy does work, it suffers from some limitations. We then will study these limitations, some relatively simple to solve, some more difficult and profound – the challenges. We will analyze various strategies[1] for each challenge, and illustrate them though different systems[2] taken from the relevant literature. This also provides an opportunity to study the possible relationships between architectures and strategies.

6.1 An Introductory Example

6.1.1 Single-Step Feedforward Strategy

The most direct strategy is using the prediction or the classification task of a neural network in order to generate musical content. Let us consider the following objective: for a given melody we want to generate an accompaniment, for example, a counterpoint. We will consider a dataset of examples, each one being a pair $(melody, counterpoint\,melody(ies))$. We then train a feedforward neural network architecture in a supervised learning manner on this dataset. Once trained, we can choose an arbitrary melody and feedforward it into the architecture in order to produce a corresponding counterpoint accompaniment, in the style of the dataset. Generation is completed in a single-step of feedforward processing. Therefore, we have named this strategy the *single-step feedforward strategy*.

6.1.2 Example: MiniBach Chorale Counterpoint Symbolic Music Generation System

Let us consider the following objective: generating a counterpoint accompaniment to a given melody for a soprano voice, through three matching parts, corresponding to alto, tenor and bass voices. We will use as a corpus the set of J. S. Bach's polyphonic chorales [4]. As we want this first introductory system to be simple, we consider only 4 measures long excerpts from the corpus. The dataset is constructed by extracting all possible 4 measures long excerpts from the original 352 chorales, also transposed in all possible keys. Once trained on this dataset, the system may be used to generate three counterpoint voices corresponding to an arbitrary 4 measures long melody provided as an input.

[1] Remember, and this will be important for the following sections, that, as stated in Chapter 2, we consider here the strategy related to the *generation phase* and not the training phase (which could be different).

[2] As proposed in Chapter 2, we use the term *systems* for various proposals – architectures, models, prototypes, systems and related experiments – for deep learning-based music generation, collected from the related literature.

Somehow, it does capture the practice of J. S. Bach, who chose various melodies for a soprano and composed the three additional voices melodies (for alto, tenor and bass) in a counterpoint manner.

First, we need to decide the input as well as the output representations. We represent four measures of 4/4 music. Both the input and the output representations are symbolic, of the piano roll type, with one-hot encoding for each voice, i.e. a multi-one-hot encoding for the output representation, which we could notate as one-hot$\times$3. The three first voices (soprano, alto and tenor) have a scope of 20 possible notes plus an additional token to encode the hold symbol[3], while the last voice (bass) has a scope of 27 possible notes plus the hold. Time quantization (the value of the time step) is set at the sixteenth note, which is the minimal note duration used in the corpus. The input representation has a size of 21 possible notes $\times$ 16 time steps $\times$ 4 measures, i.e. $21 \times 16 \times 4 = 1,344$, while the output representation has a size of $(21 + 21 + 28) \times 16 \times 4 = 4,480$.

The architecture, a feedforward network, is shown in Figure 6.1. As explained previously and because of the mapping between the representation and the architecture, the input layer has 1,344 nodes and the output layer 4,480. There is a single hidden layer with 200 units. The nonlinear activation function used for the hidden layer is ReLU. The output layer activation function is softmax for each voice. For each voice and each time step, the predicted note is the one with the highest probability. The characteristics of this system, named MiniBach[4], are summarized in our multidimensional conceptual framework in Table 6.1, with the notation Feedforward^2 meaning a 2-layer feedforward architecture (with 1 hidden layer). An example of a generated chorale is shown in Figure 6.2.

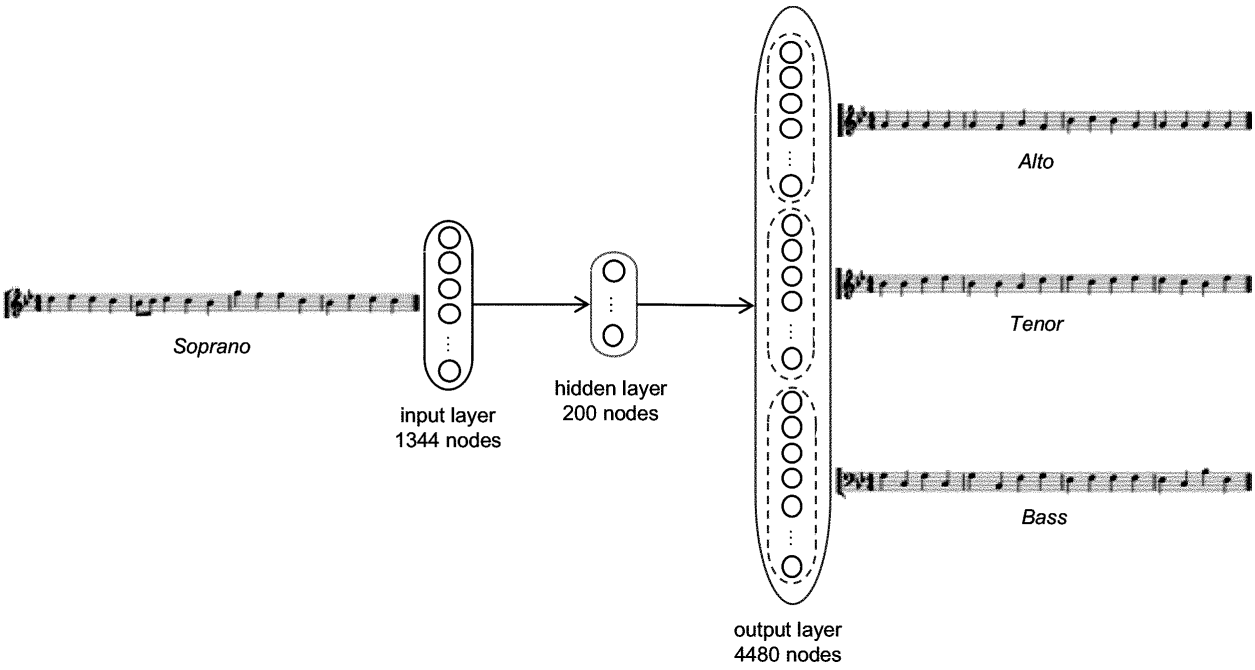

Fig. 6.1 MiniBach architecture

[3] When a note is held, the hold symbol "_" is used in place of the name of the note (see Section 4.9.1). Note that, as a simplification, we do not consider rests.

[4] MiniBach is actually a strong simplification – but with the same objective, corpus and representation principles – of the DeepBach system to be introduced in Section 6.13.2.

Objective	Multivoice; Counterpoint; Chorale; Bach
Representation	Symbolic; Piano roll; One-hot×(1+3); Hold
Architecture	Feedforward[2]
Strategy	Single-step feedforward

Table 6.1 MiniBach summary

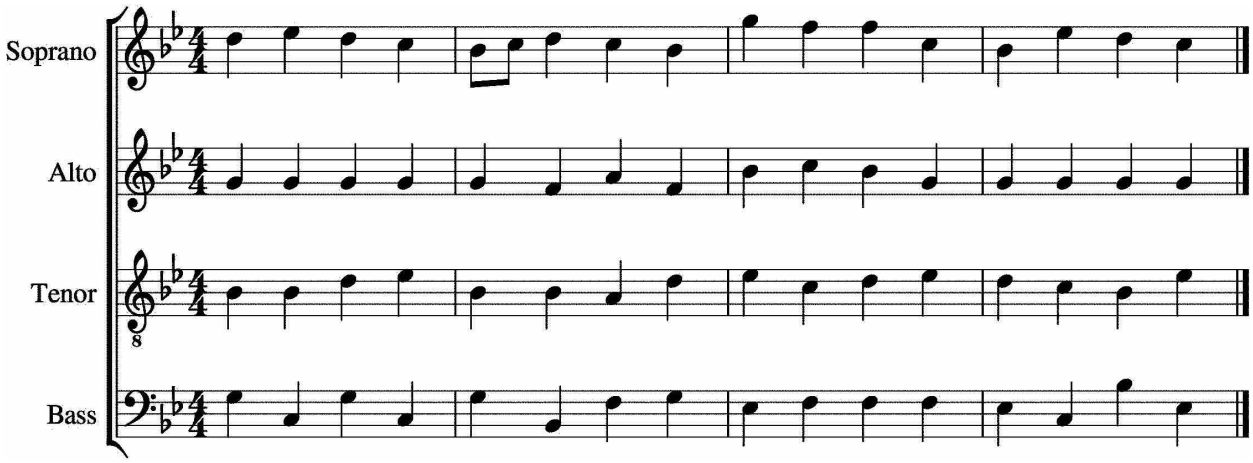

Fig. 6.2 Example of a chorale generated by MiniBach

6.1.3 A First Analysis

The chorales produced by MiniBach look convincing at first glance. But, independently of a qualitative musical evaluation, where an expert could detect some defects, objective limitations of MiniBach appear:

- A structural limitation is that the music produced (as well as the input melody) has a *fixed size* (one cannot produce a longer or shorter piece of music).
- The same melody will always produce exactly the *same* accompaniment because of the *deterministic* nature of a feedforward neural network architecture.
- The generated accompaniment is produced in a *single atomic step*, without any possibility of human intervention (i.e. without *incrementality* and *interactivity*).

6.2 A Tentative List of Limitations and Challenges

Let us now introduce a tentative list of limitations (in most cases, properties not fulfilled) and challenges[5]:

- *Ex nihilo* generation (vs accompaniment);
- Length variability (vs fixed length);
- Content variability (vs determinism);
- Expressiveness (vs mechanization);
- Melody-harmony consistency;
- Control (e.g., tonality conformance, maximum number of repeated notes…);
- Style transfer;
- Structure;

[5] Our shallow distinction between a limitation and a challenge is as follows: *limitations* have relatively well-understood solutions, whereas *challenges* are more profound and still the subject of open research.

- Originality (vs imitation);
- Incrementality (vs one-shot generation);
- Interactivity (vs automation);
- Adaptability (vs no improvement through usage); and
- Explainability (vs black box).

We will analyze them with possible matching solutions and illustrate them through various examples systems.

6.3 *Ex Nihilo* Generation

The MiniBach system is good at generating an accompaniment (a counterpoint composed of three distinct melodies) matching an input melody. This is an example of supervised learning, as training examples include both an input (a melody) and a corresponding output (accompaniment).

Now suppose that our objective is to generate a melody on its own – not as an accompaniment of some input melody – while being based on a style learnt from a corpus of melodies. A standard feedforward architecture and its companion single-step feedforward strategy, such as those used in MiniBach (described in Section 6.1.2), are not appropriate for such an objective.

Let us introduce some strategies to generate new music content *ex nihilo* or from minimal *seed* information, such as a starting note or high-level description.

6.3.1 Decoder Feedforward

The first strategy is based on an autoencoder architecture. As explained in Section 5.9, through the training phase an autoencoder will specialize its hidden layer into a detector of features characterizing the type of music learnt and its variations[6]. One can then use these features as an *input interface* to *parameterize* the generation of musical content. The idea is then to:

- *choose* a *seed* as a vector of values corresponding to the hidden layer units;
- *insert* it in the hidden layer; and
- *feedforward* it through the decoder.

This will produce a *new* musical content corresponding to the features, in the same format as the training examples.

In order to have a minimal and high-level vector of features, a stacked autoencoder (see Section 5.9.3) is often used. The seed is then inserted at the *bottleneck hidden layer* of the stacked autoencoder[7] and feedforwarded through the chain of decoders. Therefore, a simple seed information can generate an arbitrarily long, although fixed-length, musical content.

6.3.1.1 #1 Example: DeepHear Ragtime Melody Symbolic Music Generation System

An example of this strategy is the DeepHear system by Sun [164]. The representation used is piano roll with a one-hot encoding. The quantization (time step) is a sixteenth note. The corpus used is 600 measures of Scott Joplin's ragtime music, split into 4 measures long segments, thus with 64 time steps. The number of input nodes is around 5,000, which provides a vocabulary of about 80 possible note values. The architecture is shown in Figure 6.3 and is a 4-layer stacked autoencoder with a decreasing number of hidden units, down to 16 units.

[6] To enforce this specialization, sparse autoencoders are often used (see Section 5.9.1).

[7] In other words, at the exact middle of the encoder/decoder stack, as shown in Figure 6.5.

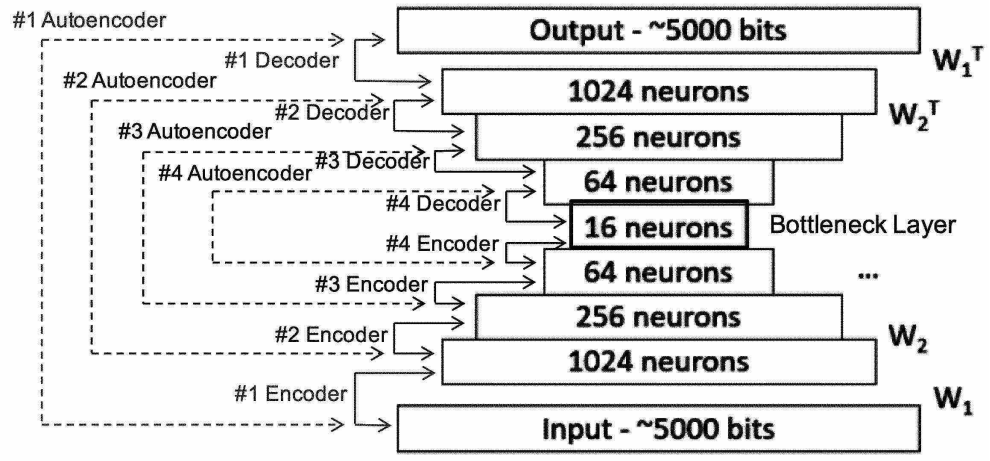

Fig. 6.3 DeepHear stacked autoencoder architecture. Extension of a figure reproduced from [164] with the permission of the author

After a pre-training phase[8], final training is performed, with each provided example used both as an input and as an output, in the self-supervised learning manner (see Section 5.9) shown in Figure 6.4.

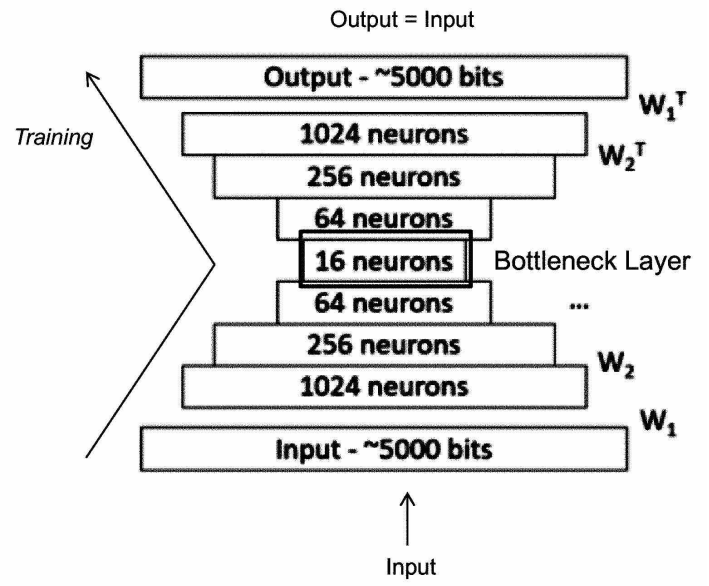

Fig. 6.4 Training DeepHear. Extension of a figure reproduced from [164] with the permission of the author

Generation is performed by inputing random data as the seed into the 16 bottleneck hidden layer units[9] (shown within a red rectangle) and then by feedforwarding it into the chain of decoders to produce an output (in the same 4 measures long format as the training examples), as shown in Figure 6.5. We summarize the characteristics of DeepHear$_M$[10] in Table 6.2.

[8] We do not detail pre-training here, please refer to, for example, [58, page 528].

[9] The units of the hidden layer represent an embedding (see Section 4.9.3), of which an arbitrary instance is named by Sun a *label*.

[10] We notate DeepHear$_M$ this DeepHear melody generation system, where *M* stands for melody, because another experiment with the same DeepHear architecture but with a different objective will be presented later on in Section 6.9.4.1.

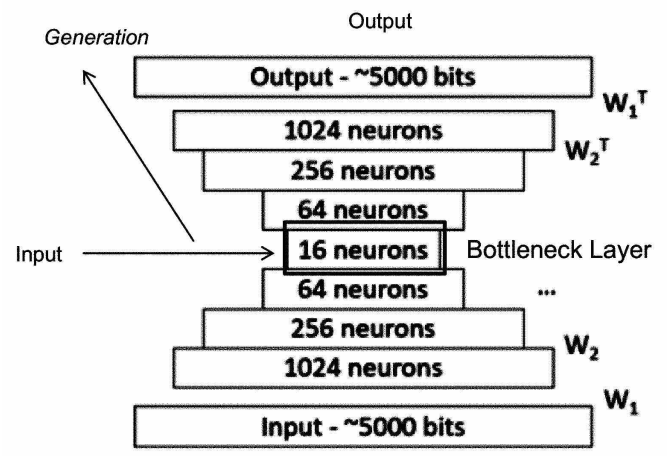

Fig. 6.5 Generation in DeepHear. Extension of a figure reproduced from [164] with the permission of the author

Objective	Melody; Ragtime
Representation	Symbolic; Piano roll; One-hot
Architecture	Stacked autoencoder = Autoencoder[4]
Strategy	Decoder feedforward

Table 6.2 DeepHear$_M$ summary

In [164], Sun remarks that the system produces a certain amount of plagiarism. Some generated music is almost recopied from the corpus. He states that this is because of the small size of the bottleneck hidden layer (only 16 nodes) [164]. 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, it is 59.6%, which is indeed quite high, although it does not prevent most of generated pieces from sounding different.

6.3.1.2 #2 Example: deepAutoController Audio Music Generation System

The deepAutoController system, by Sarroff and Casey [155], is similar to DeepHear (see Section 6.3.1.1) in that it also uses a stacked autoencoder. But the representation is *audio*, more precisely a spectrum generated by Fourier transform, see [155] for more details. The dataset is composed of 8,000 songs of 10 musical genres, leading to 70,000 frames of magnitude Fourier transforms[11]. The entire data is normalized to the $[0,1]$ range. The cost function used is quadratic cost. The architecture is a 2-layer stacked autoencoder, the bottleneck hidden layer having 256 units and the input and output layers having 1,000 nodes. The authors report that increasing the number of hidden units does not appear to improve the model performance.

The system, summarized at Table 6.3, also provides a user interface, analyzed in Section 6.14, to interactively control the generation, e.g., selecting a given input (to be inserted at the bottleneck hidden layer), generating a random input, and controlling (by scaling or muting) the activation of a given unit.

[11] As the authors state in [155]: "We chose to use frames of magnitude FFTs (Fast Fourier transforms) for our models because they may be reconstructed exactly into the original time domain signal when the phase information is preserved, the Fourier coefficients are not altered, and appropriate windowing and overlap-add is applied. It was thus easier to subjectively evaluate the quality of reconstructions that had been processed by the autoencoding models."

Objective	Audio; User interface
Representation	Audio; Spectrum
Architecture	Stacked autoencoder = Autoencoder^2
Strategy	Decoder feedforward

Table 6.3 deepAutoController summary

6.3.2 Sampling

Another strategy is based on sampling. *Sampling* is the action of generating an element (a *sample*) from a *stochastic* model according to a *probability distribution*.

6.3.2.1 Sampling Basics

The main issue for sampling is to ensure that the samples generated match a given distribution. The basic idea is to generate 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 thus produced *iteratively*, with the distribution of the next sample being dependent only on the current sample value. Each successive sample is generated through a *generate-and-test* strategy, i.e. by generating a prospective candidate, accepting or rejecting it (based on a defined *probability density*) and, if needed, regenerating it. Various sampling strategies have been proposed: Metropolis-Hastings algorithm, Gibbs sampling (GS), block Gibbs sampling, etc. Please see, for example, [58, Chapter 17] for more details about sampling algorithms.

6.3.2.2 Sampling for Music Generation

For musical content, we may consider two different levels of probability distribution (and sampling):

- *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; and
- *sequence*-level or *horizontal* dimension – at the level of a sequence of items, e.g., a melody composed of successive notes. In this 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.

An RBM (restricted Boltzmann machine) architecture is generally[12] used to model the vertical dimension, i.e. which notes should be played together. As noted in Section 5.10, an RBM architecture is dedicated to learning distributions and can learn efficiently from few examples. This is particularly interesting for learning and generating chords, as the combinatorial nature of possible notes forming a chord is large and the number of examples is usually small. An example of a *sampling strategy* applied on an RBM for the horizontal dimension will be presented in Section 6.3.2.3.

An RNN (recurrent neural network) architecture is often used for the horizontal dimension, i.e. which note is likely to be played after a given note, as will be described in Section 6.4.1. As we will see in Section 6.5.1, a sampling strategy may be also added to enforce variability.

We will see in Section 6.8.1 that a compound architecture named RNN-RBM may combine and *articulate*[13] these two different approaches:

- an RBM architecture with a sampling strategy for the vertical dimension; and
- an RNN architecture with an iterative feedforward strategy for the horizontal dimension.

[12] A counterexample is the C-RBM convolutional RBM architecture, to be introduced in Section 6.9.5.1, which models both the vertical dimension (simultaneous notes) and the horizontal dimension (sequence of notes) for single-voice polyphonies.

[13] This issue of how to articulate vertical and horizontal dimensions, i.e. harmony with melody, will be further analyzed in Section 6.8.

An alternative approach is to use sampling as the *unique* strategy for both dimensions, as witnessed by the DeepBach system to be analyzed in Section 6.13.2.

6.3.2.3 Example: RBM-based Chord Music Generation System

In [10], Boulanger-Lewandowski *et al.* propose to use a restricted Boltzmann machine (RBM) [75] to model polyphonic music. Their objective is actually to improve the transcription of polyphonic music from audio. But prior to that, the authors discuss the generation of samples from the model that has been learnt as a qualitative evaluation and also for music generation [11]. In their first experiment, the RBM learns from the corpus the distribution of possible simultaneous notes, i.e. a repertoire of chords.

The corpus is the set of J. S. Bach's chorales (as for MiniBach, described in Section 6.1.2). The polyphony (number of simultaneous notes) varies from 0 to 15 and the average polyphony is 3.9. The input representation has 88 binary visible units that span the whole range of piano from A_0 to C_8, following a many-hot encoding. The sequences are aligned (transposed) onto a single common tonality (e.g., C major/minor) to ease the learning process.

One can sample from the RBM through block Gibbs sampling, by performing alternative steps of sampling the hidden layer nodes (considered as variables) from the visible layer nodes (see Section 5.10). Figure 6.6 shows various examples of samples. The vertical axis represents successive possible notes. Each column represents a specific sample composed of various simultaneous notes, with the name of the chord written below when the analysis is unambiguous. Table 6.4 summarizes this RBM-based chord generation system, which we notate RBM_C (where C stands for chords).

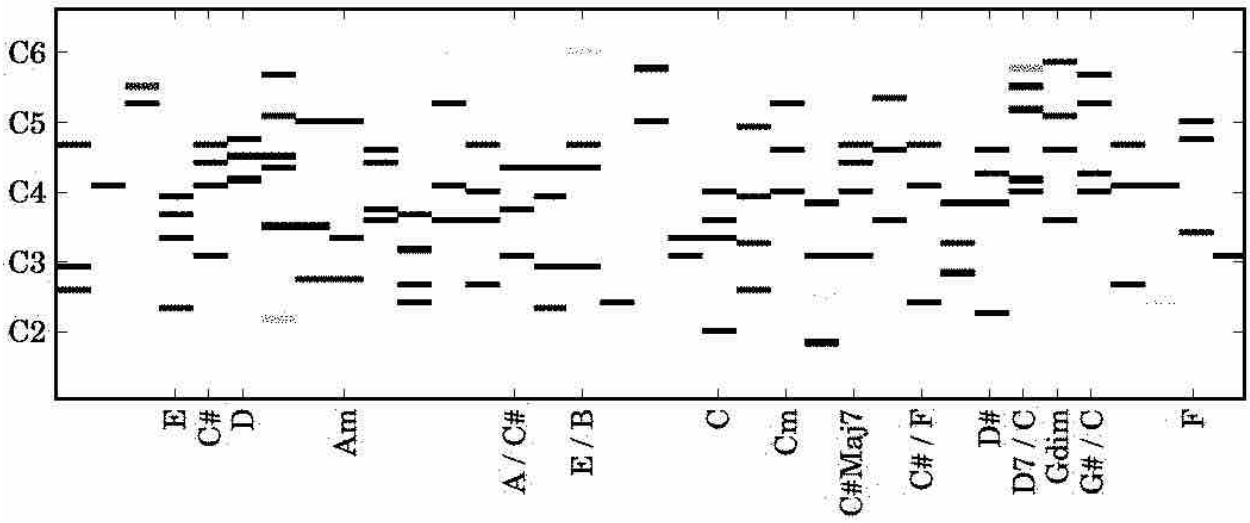

Fig. 6.6 Samples generated by the RBM trained on J. S. Bach chorales. Reproduced from [10] with the permission of the authors

Objective	Simultaneous notes (Chord)
Representation	Symbolic; Many-hot
Architecture	RBM
Strategy	Sampling

Table 6.4 RBM_C summary

6.4 Length Variability

An important limitation of single-step feedforward (Section 6.1.1) and decoder feedforward (Section 6.3.1) strategies is that the length of the music generated (more precisely the number of times steps or measures) is *fixed*. It is actually fixed by the architecture, namely the number of nodes of the output layer[14]. To generate a longer (or shorter) piece of music, one needs to reconfigure the architecture and its corresponding representation.

6.4.1 Iterative Feedforward

The standard solution to this limitation is to use a recurrent neural network (RNN) and to perform generation in an *iterative* way, time step by time step. Indeed, the recurrent nature of a recurrent neural network, which learns to predict information for the next step, can be used to generate sequences of *arbitrary* length. The typical usage, as described in [60], for text generation, is to

- select some *seed* information as the *first* item (e.g., the first note of a melody);
- *feedforward* it into the recurrent network in order to produce the *next* item (e.g., next note);
- use this next item as the next input to produce the *next next* item; and
- repeat this process until a *sequence* (e.g., of notes, i.e. a melody) of the desired length is produced.

Therefore, we have named this the *iterative time step feedforward* strategy, or more simply the *iterative feedforward* strategy[15]. Note that the iterative feedforward strategy is one kind of *seed-based generation* (see Section 6.3) as the full sequence (e.g., a melody) is generated iteratively from the initial seed item (e.g., the starting note).

6.4.1.1 #1 Example: Blues Chord Sequence Symbolic Music Generation System

In [37], Eck and Schmidhuber describe a double experiment undertaken with a recurrent network architecture using LSTMs[16]. In their first experiment, the objective is to learn and generate chord sequences. The format of representation is piano roll, with two types of sequences: melody and chords, although chords are represented as notes. The melodic range as well as the chord vocabulary is strongly constrained, as the corpus consists of 12 measures long 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.

A one-hot encoding is used. Time quantization (time step) is set at the eighth note, half of the minimal note duration used in the corpus, which is a quarter note. With 12 measures long music this equates to 96 time steps. An example of chord sequence training example is shown in Figure 6.7.

The architecture for this first experiment is: an input layer with 12 nodes (corresponding to a one-hot encoding of the 12 chord vocabulary), a hidden layer with four LSTM blocks containing two cells each[17] and an output layer with 12 nodes (identical to the input layer).

Generation is performed by presenting a *seed* chord (represented by a note) and by iteratively feedforwarding the network, producing the prediction of the next time step chord, using it as the next input and so on, until a sequence of chords has been generated. This system, which we notate Blues_C (where C stands for chords), is summarized in Table 6.5.

[14] In the case of an RBM, the number of nodes of the input layer (which also has the role of an output layer).

[15] In [127], Mozer names it the *note-to-note* composition technique.

[16] This was actually the first experiment in using LSTMs to generate music.

[17] See in Section 5.11.3 for the difference between LSTM cells and blocks.

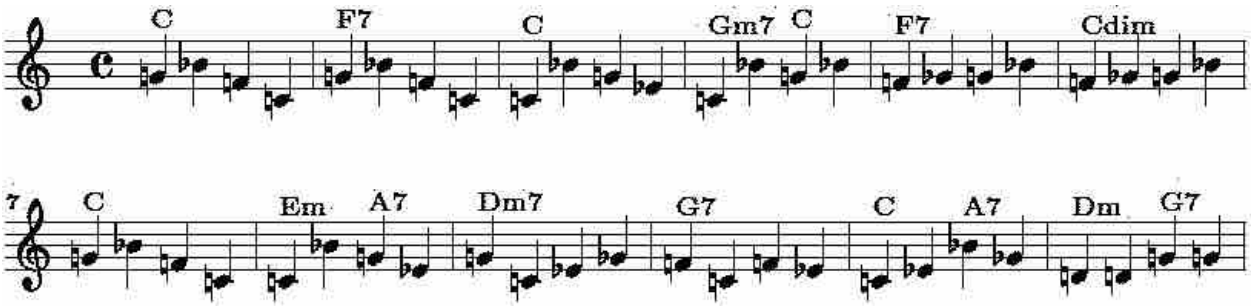

Fig. 6.7 A chord training example for blues generation. Reproduced from [37] with the permission of the authors

Objective	Chord sequence; Blues
Representation	Symbolic; One-hot; Note end; Chord as note
Architecture	LSTM
Strategy	Iterative feedforward

Table 6.5 Blues_C summary

6.4.1.2 #2 Example: Blues Melody and Chords Symbolic Music Generation System

In Eck and Schmidhuber's second experiment [37], the objective is to simultaneously generate melody and chord sequences. The new architecture is an extension of the previous one: it has an input layer with 25 nodes (corresponding to a one-hot encoding of the 12 chord vocabulary and to a one-hot encoding of the 13 melody note vocabulary), a hidden layer with eight LSTM blocks (four chord blocks and four melody blocks, as we will see below), containing two cells each, and an output layer with 25 nodes (identical to the input layer).

The separation between chords and melody is ensured as follows:

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

Generation is performed by presenting a seed (note and chord) and by feedforwarding it into the network, producing the prediction of the next time step note and chord, and so on, until a sequence of notes with chords is generated. Figure 6.8 shows an example of the melody and chords generated. Table 6.6 summarizes this second system, which we notate Blues_{MC} (where *MC* stands for melody and chords).

Objective	Melody + Chords; Blues
Representation	Symbolic; One-hot$\times$2; Note end; Chord as note
Architecture	LSTM
Strategy	Iterative feedforward

Table 6.6 Blues_{MC} summary

This second experiment is interesting in that it *simultaneously* generates melody *and* chords. Note that in this second architecture, recurrent connexions are *asymmetric* as the authors wanted to ensure the preponderant role of chords. Chord blocks have recurrent connexions to themselves but also to melody blocks, whereas melody blocks do not have recurrent connexions to chord blocks. This means that chord blocks will receive previous step information about chords *and* melody, whereas melody blocks cannot use previous step information about chords. This somewhat *ad hoc* configuration of the recurrent connexions in the architecture is a way to control the interaction between harmony and melody in a master-slave manner. The control of the interaction and consistency between melody and harmony is indeed an effective issue and it will be further addressed in Section 6.8 where we will analyze alternative approaches.

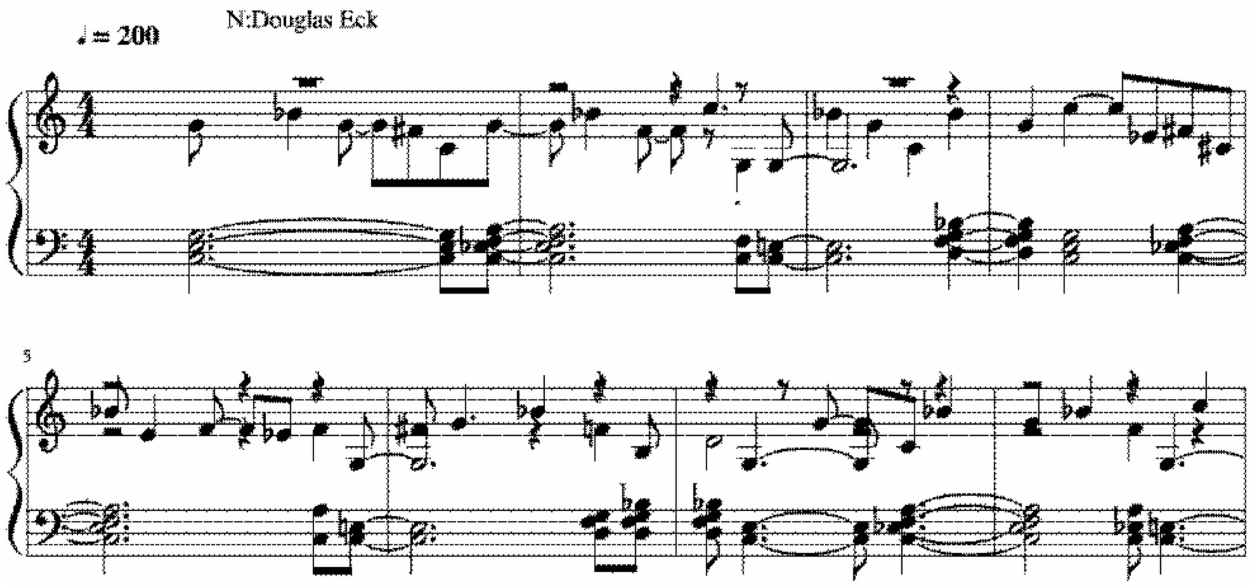

Fig. 6.8 Example of blues generated (excerpt). Reproduced with the permission of the authors

6.5 Content Variability

A limitation of the iterative feedforward strategy on an RNN, as illustrated by the blues generation experiment described in Section 6.4.1.2, is that generation is *deterministic*. Indeed, a neural network is deterministic[18]. As a consequence, feedforwarding the *same input* will always produce the *same output*. As the generation of the next note, the next next note, etc., is deterministic, the *same* seed note will lead to the *same* generated series of notes[19]. Moreover, as there are only 12 possible input values (the 12 pitch classes), there are only 12 possible melodies.

6.5.1 Sampling

Fortunately, as we will see, the usual solution is quite simple. The assumption is that the output representation of the melody is one-hot encoded. In other words, the output representation is of a piano roll type, the output activation layer is softmax and generation is modeled as a classification task. See an example in Figure 6.9, where $p(x_t = \mathrm{C}|x_{<t})$ represents the conditional probability for the element (note) x_t at step t to be a C given the previous elements $x_{<t}$ (the melody generated so far).

The default *deterministic* strategy consists in choosing the class (the note) with the *highest probability*, i.e. $\mathrm{argmax}_{x_t} p(x_t|x_{<t})$, that is A♭ in Figure 6.9. We can then easily switch to a *nondeterministic* strategy, by *sampling* the output which corresponds (through the softmax function) to a probability distribution between possible notes. By sampling a note following the produced distribution[20], we introduce *stochasticity* in the process and thus *variability* in the generation.

[18] There are stochastic versions of artificial neural networks – an RBM is an example – but they are not mainstream.

[19] The actual length of the melody generated depends on the number of iterations.

[20] The chance of sampling a given class/note is its corresponding probability. In the example shown, A♭ has around one chance in two of being selected and B♭ one chance in four.

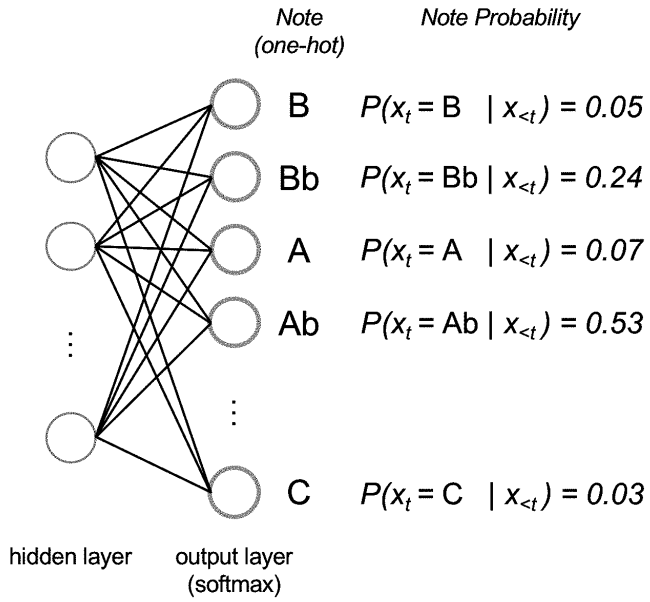

Fig. 6.9 Sampling the softmax output

6.5.1.1 #1 Example: CONCERT Bach Melody Symbolic Music Generation System

CONCERT (an acronym for CONnectionist Composer of ERudite Tunes) developed by Mozer [127] in 1994, was actually one of the first systems for generating music based on recurrent networks (and before LSTM). It is aimed at generating melodies, possibly with some chord progression as an accompaniment.

The input and output representation includes three aspects of a note: pitch, duration and *harmonic chord accompaniment*. The representation of a pitch, named PHCCCH, is inspired by the psychological pitch representation space of Shepard [158], and is based on five dimensions, as illustrated in Figure 6.10.

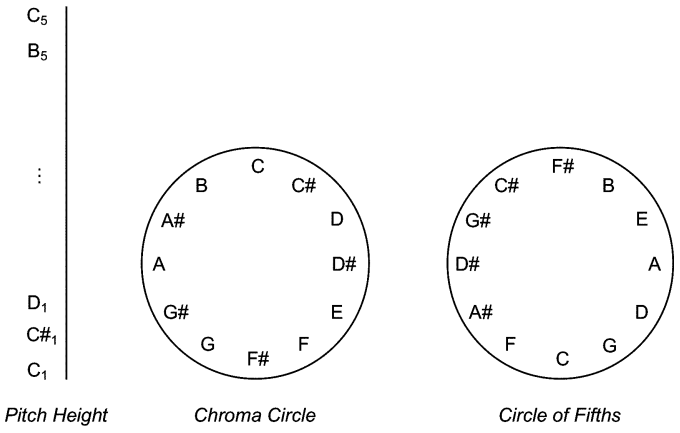

Fig. 6.10 CONCERT PHCCCH pitch representation. Inspired by [158] and [127]

The three main components are as follows:

- the pitch height (PH),

- the (modulo) chroma circle (CC) cartesian coordinates, and
- the (harmonic) circle of fifths (CH) cartesian coordinates.

The motivation is in having a more musically meaningful representation of the pitch by capturing the similarity of octaves and also the harmonic similarity between a note and its fifth. The proximity of two pitches is determined by computing the Euclidean distance between their PHCCCH representations, that distance being invariant under transposition. The encoding of the pitch height is through a scalar variable scaled to range from -9.798 for C_1 to +9.798 for C_5. The encoding of the chroma circle and of the circle of fifths is through a six binary value vector, for the reasons detailed in [127]. The resulting encoding includes 13 input variables, with some examples shown in Table 6.7. Note that a rest is encoded as a pitch with a unique code.

Pitch	*PH*	*CC*						*CH*					
C_1	-9.798	+1	+1	+1	-1	-1	-1	-1	-1	-1	+1	+1	+1
$F\sharp_1$	-7.349	-1	-1	-1	+1	+1	+1	+1	+1	+1	-1	-1	-1
G_2	-2.041	-1	-1	-1	-1	+1	+1	-1	-1	-1	-1	+1	+1
C_3	0	+1	+1	+1	-1	-1	-1	-1	-1	-1	+1	+1	+1
$D\sharp_3$	1.225	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
E_3	1.633	-1	+1	+1	+1	+1	+1	+1	-1	-1	-1	-1	-1
A_4	8.573	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
C_5	9.798	+1	+1	+1	-1	-1	-1	-1	-1	-1	+1	+1	+1
Rest	0	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1

Table 6.7 Examples of PHCCCF pitch representation

Durations are considered at a very fine-grain level, each beat (a quarter note) being divided into twelfths, thus having a duration of 12/12. This choice allows to represent binary (two or four divisions) as well as ternary (three divisions) rhythms. In a similar way to the representation of pitch, a duration is represented through a scalar and two circle coordinates, for 1/4 and 1/3 beat cycles, as illustrated in Figure 6.11, resulting in five dimensions directly encoded through a five binary value vector (see more details in [127]). The temporal scope is a *note step*[21], that is the granularity of processing by the architecture is a note.

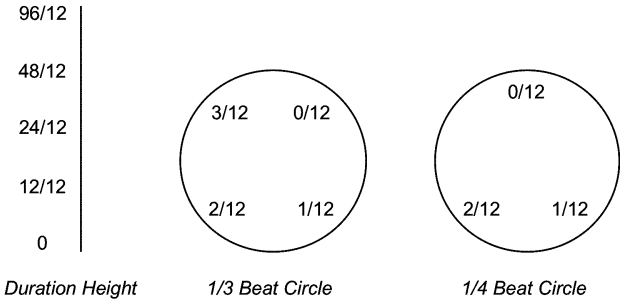

Fig. 6.11 CONCERT duration representation. Inspired by [127]

Chords are represented in an extensional way as a triad or a tetrachord, through the root, the third (major or minor) and the fifth (perfect, augmented or diminished), with the possible addition of a seventh component (minor or major). To represent the next note to be predicted, the CONCERT system actually uses both this rich and distributed representation (named next-node-distributed, see Figure 6.12) 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.

[21] And not a fixed time step, as in Section 6.4.1.1, see Section 4.8.1.

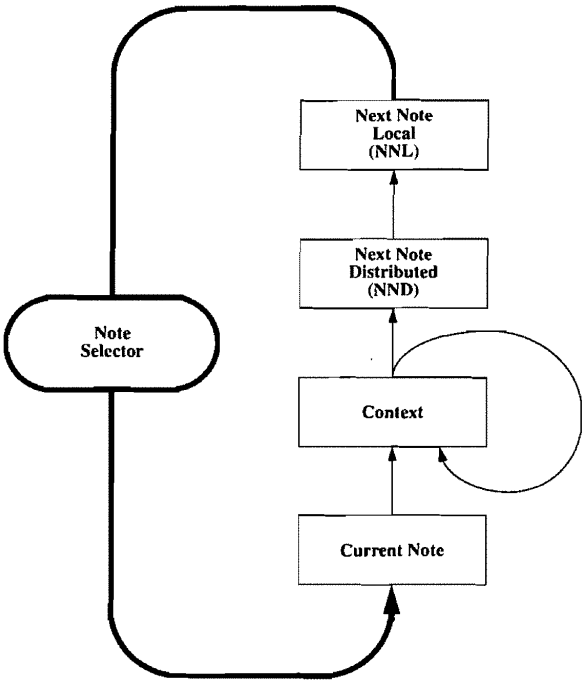

Fig. 6.12 CONCERT architecture. Reproduced from [127] with the permission of Taylor & Francis (www.tandfonline.com)

In the generation phase, the output is interpreted as a probability distribution over a set of possible notes as a basis for deciding the next note in a nondeterministic way, following the *sampling* strategy.

CONCERT has been tested on different examples, notably after training with melodies of J. S. Bach. Figure 6.13 shows an example of a melody generated based on the Bach training set. Although now a bit dated, CONCERT has been a pioneering model and the discussion in the article about representation issues is still relevant.

Note also that CONCERT (which is summarized in Table 6.8) is representative of the early generation systems, before the advent of deep learning architectures, when representations were designed with rich handcrafted features. One of the benefits of using deep learning architectures is that this kind of rich and deep representation may be automatically extracted and managed by the architecture.

Objective	Melody + Chords
Representation	Symbolic; Harmonics; Harmony; Beat
Architecture	RNN
Strategy	Iterative feedforward; Sampling

Table 6.8 CONCERT summary

6.5.1.2 #2 Example: Celtic Melody Symbolic Music Generation System

Another representative example is the system by Sturm *et al.* to generate Celtic music melodies [163]. The architecture used is a recurrent network with three hidden layers, which we could notate[22] as LSTM^3, with 512 LSTM cells in each layer.

[22] Note that, as explained in Section 5.8.2, we notate the number of hidden layers without considering the input layer.

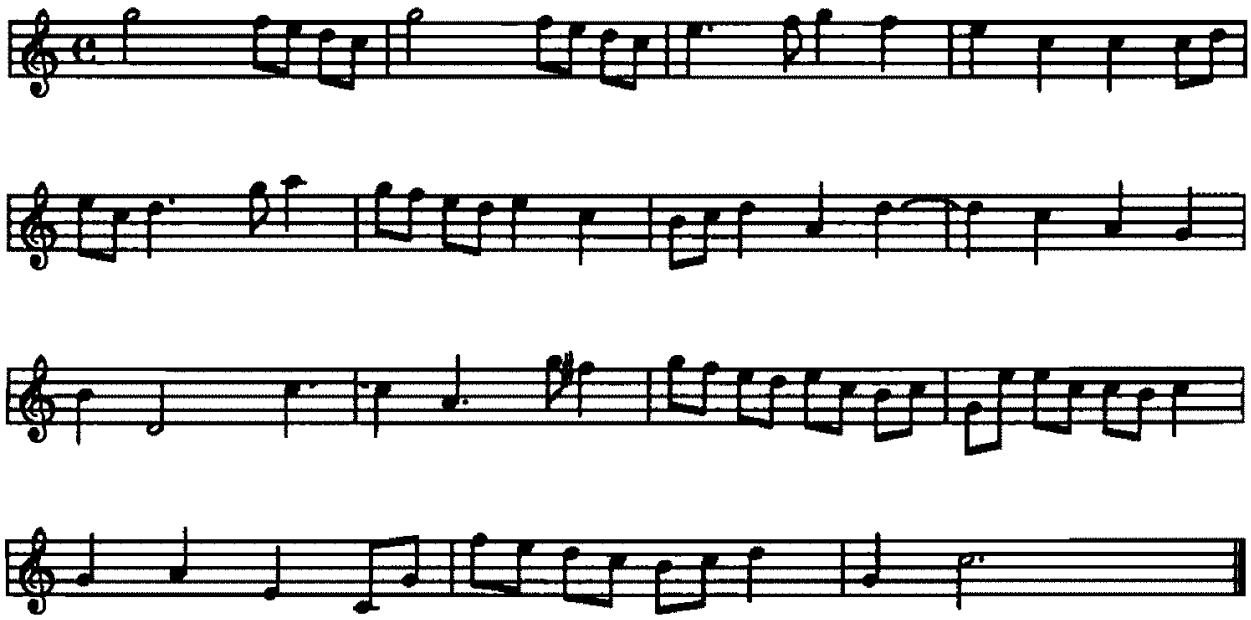

Fig. 6.13 Example of melody generation by CONCERT based on the J. S. Bach training set. Reproduced from [127] with the permission of Taylor & Francis (www.tandfonline.com)

The corpus comprises folk and Celtic monophonic melodies retrieved from a repository and discussion platform named The Session [91]. Pieces that were too short, too complex (with varying meters) or contained errors were filtered out, leaving a dataset of 23,636 melodies. All melodies are aligned (transposed) onto the single C key. One of the specificities is that the representation chosen is *textual*, namely the token-based *folk-rnn* notation, a transformation of the character-based ABC notation (see Section 4.7.3). The number of input and output nodes is equal to the number of tokens in the vocabulary (i.e. with a one-hot encoding), in practice equal to 137. 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 token or a specific token (e.g., corresponding to a specific starting note), feedforwarding it to generate the output, sampling from this probability distribution, and using each selected vocabulary element as a subsequent input, in order to iteratively produce a melody.

The final step is to decode the folk-rnn representation generated into a MIDI format melody to be played. See in Figure 6.14 for an example of a melody generated. One may also see and listen to results on [162]. The results are very convincing, with melodies generated in a clear Celtic style. The system is summarized in Table 6.9.

As observed in [62]: "It is interesting to note that in this approach the bar lines and the repeat bar lines are given explicitly and are to be predicted as well. This can cause some issues, since there is no guarantee that the output sequence of tokens would represent a valid song in ABC format. There could be too many notes in one bar for example, but according to the authors, this rarely occurs. This would tend to show that such an architecture is able to learn to count."[23]

Objective	Melody
Representation	Symbolic; Text; Token-based; One-hot
Architecture	LSTM[3]
Strategy	Iterative feedforward; Sampling

Table 6.9 Celtic system summary

[23] On this issue, see also [55].

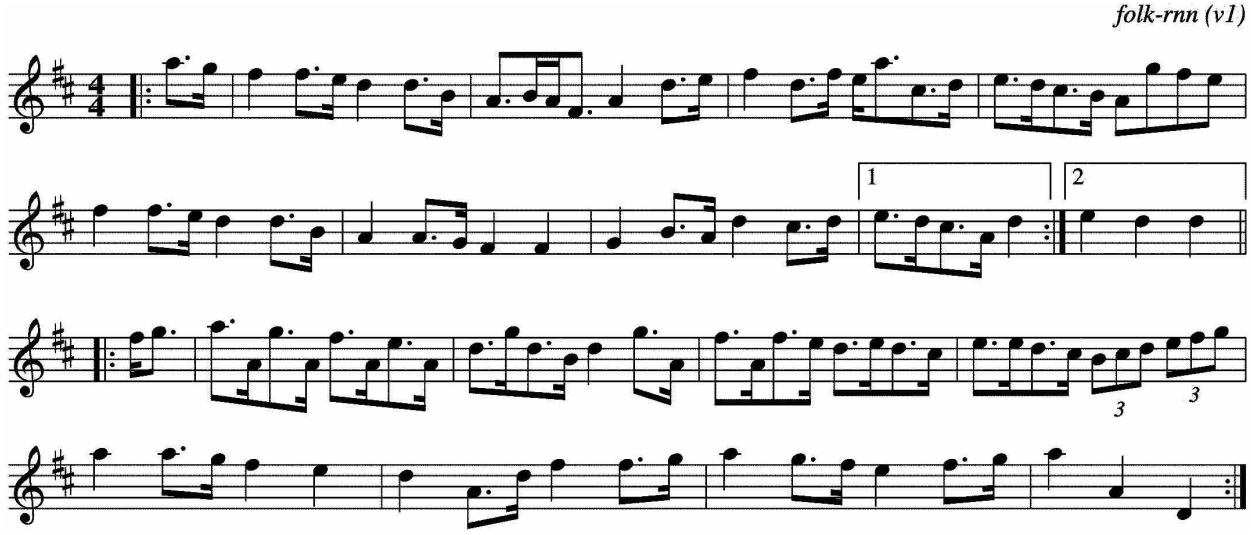

Fig. 6.14 Score of "The Mal's Copporim" automatically generated. Reproduced from [163] with the permission of the authors

6.6 Expressiveness

One limitation of most existing systems is that they consider fixed dynamics (amplitude) for all notes as well as an exact quantization (a fixed tempo), which makes the music generated too mechanical, without *expressiveness* or *nuance*.

A natural approach resides in considering representations recorded from real performances and not simply scores, and therefore with musically grounded (by skilled human musicians) variations of tempo and of dynamics, as discussed in Section 4.10.

Note that an alternative approach is to automatically *augment* the generated music information (e.g., a standard MIDI piece) with slight transformations on the amplitude and/or the tempo. An example is the CyberJoão system [27], which performs bossa nova guitar accompaniment with expressiveness, through automatic retrieval[24] and application of rhythmic patterns[25].

As noted in Section 4.10.3, in the case of an audio representation, expressiveness is implicit to the representation. However, it is almost impossible to separately control the expressiveness (dynamics or tempo) of a single instrument or voice as the representation is global.

6.6.1 Example: Performance RNN Piano Polyphony Symbolic Music Generation System

In [159], Simon and Oore present their architecture and methodology named Performance RNN. It is an LSTM-based recurrent neural network architecture. One of the specificities is in the dataset characteristics, as the corpus is composed of recorded human performances, with records of exact timing as well as dynamics for each note played. The corpus used is the Yamaha e-Piano Competition dataset, whose participants MIDI performance records are made available to the public [189]. It captures more than 1,400 performances by skilled pianists. To create additional training examples, some time stretching (up to 5% faster or slower) as well as some transposition (up to a major third) is applied.

[24] By a mixed use of production rules and case-based reasoning (CBR).

[25] These patterns have been manually extracted from a corpus of performances by the guitarist and singer João Gilberto, one of the inventors of the Bossa nova style. One could also consider automatic extraction, as, for example, in [29].

The representation is adapted to the objective. At first look, it resembles a piano roll with MIDI note numbers but it is actually a bit different. Each time slice is a multi-one-hot vector of the possible values for each of the following possible events:

- start of a new note – with 128 possible values (MIDI pitches),
- end of a note – with 128 possible values (MIDI pitches),
- time shift – with 100 possible values (from 10 miliseconds to 1 second), and
- dynamics – with 32 possible values (128 MIDI velocities quantized into 32 bins[26]).

An example of a performance representation is shown in Figure 6.15.

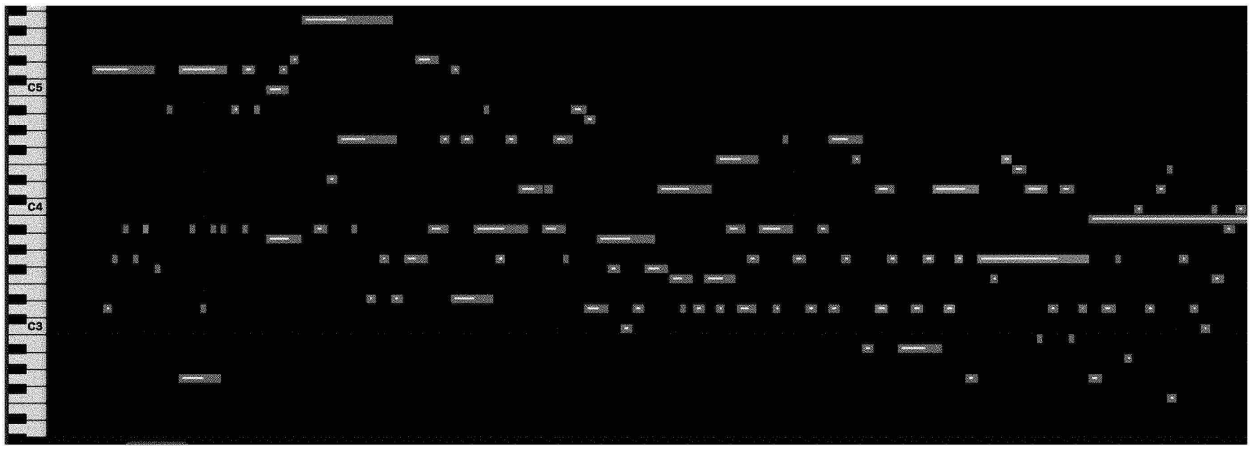

Fig. 6.15 Example of Performance RNN representation. Reproduced from [159] with the permission of the authors

Some control is made available to the user, referred to as the *temperature*, which controls the randomness of the generated events in the following way:

- a temperature of 1.0 uses the exact distribution predicted,
- a value smaller than 1.0 reduces the randomness and thus increases the repetition of patterns, and
- a larger value increases the randomness and decreases the repetition of patterns.

Examples are available on the web page [159]. Performance RNN is summarized in Table 6.10.

Objective	Polyphony; Performance control
Representation	Symbolic; One-hot×4; Time shift; Dynamics
Architecture	LSTM
Strategy	Iterative feedforward; Sampling

Table 6.10 Performance RNN summary

6.7 A First Discussion

The initial limitations that we have discussed so far have been rather easily solved. They were indeed more limitations than real challenges, although we could see that some limitations may conflict with each other and with some objectives.

[26] See the description of the binning transformation in Section 4.11.

For instance, the iterative feedforward strategy coupled with an RNN architecture allows variable length generation, as discussed in Section 6.4. But it restricts the type of objective to the generation of sequences (e.g., a melody). It cannot directly generate an accompaniment (e.g., a counterpoint) as the input layer and the output layer of the RNN architecture must be isomorphic. However, note that a solution to generate an accompaniment would be to use an RNN Encoder-Decoder compound architecture as introduced in Section 5.16.3, because it uses two distinct RNNs in order to decouple input and output sequences. Also note that the use of an RNN Encoder-Decoder will combine the decoder feedforward strategy with the iterative feedforward strategy, as in the VRAE system to be described in Section 6.9.2.3.

Some other examples of conflicts will be discussed in Section 6.17. Before that, we will continue to analyze challenges and possible solutions or directions.

6.8 Melody-Harmony Interaction

When the objective is to generate simultaneously a melody with an accompaniment, expressed through some harmony or counterpoint[27], an issue is the musical consistency between the melody and the harmony. Although a general architecture such as MiniBach (Section 6.1.2) is supposed to have learnt correlations, interactions between vertical and horizontal dimensions are not always explicitly considered.

We have analyzed in Section 6.4.1.2 an example of a specific architecture to generate simultaneously melody and chords, with explicit relations between them (i.e. chords can use previous step information about melody but not the opposite). However, this architecture is a bit *ad hoc*. In the following sections, we will analyze some more general architectures having in mind interactions between melody and harmony.

6.8.1 #1 Example: RNN-RBM Polyphony Symbolic Music Generation System

In [10], Boulanger-Lewandowski *et al.* have associated to the RBM-based architecture introduced in Section 6.3.2.3 a recurrent network (RNN) in order to represent the temporal sequence of notes. The idea is to *couple* the RBM to a deterministic RNN with a single hidden layer, such that

- the RNN models the *temporal sequence* to produce successive outputs, corresponding to successive time steps,
- which are *parameters*, more precisely the *biases*, of an RBM that models the *conditional probability distribution* of the *accompaniment notes*, i.e. which notes should be played together.

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). The resulting architecture named RNN-RBM is shown in Figure 6.16, and can be interpreted as follows:

- the bottom line represents the temporal sequence of RNN hidden units $u^{(0)}$, $u^{(1)}$, ..., $u^{(t)}$, where $u^{(t)}$ notation means[28] the value of the RNN hidden layer u at time (index) t; and
- the upper part represents the sequence of each RBM instance at time t, which we could notate RBM$^{(t)}$, with
  - $v^{(t)}$ its visible layer with $b_v^{(t)}$ its bias,
  - $h^{(t)}$ its hidden layer with $b_h^{(t)}$ its bias, and
  - W the weight matrix of connexions between the visible layer and the hidden layer.

[27] Harmony and counterpoint are dual approaches of accompaniment with different focus and priorities. Harmony focuses on the *vertical* relations between simultaneous notes, as objects on their own (*chords*), and then considers the horizontal relations between them (e.g., harmonic cadences). Conversely, counterpoint focuses on the *horizontal* relations between successive notes for each simultaneous melody (a *voice*), and then considers the vertical relations between their progression (e.g., to avoid parallel fifths). Note that, although their perspectives are different, the analysis and control of relations between vertical and horizontal dimensions are their shared objective.

[28] Note that the usual notation would be u_t, as the $u^{(t)}$ notation is usually reserved to index dataset examples (tth example), see Section 5.11.

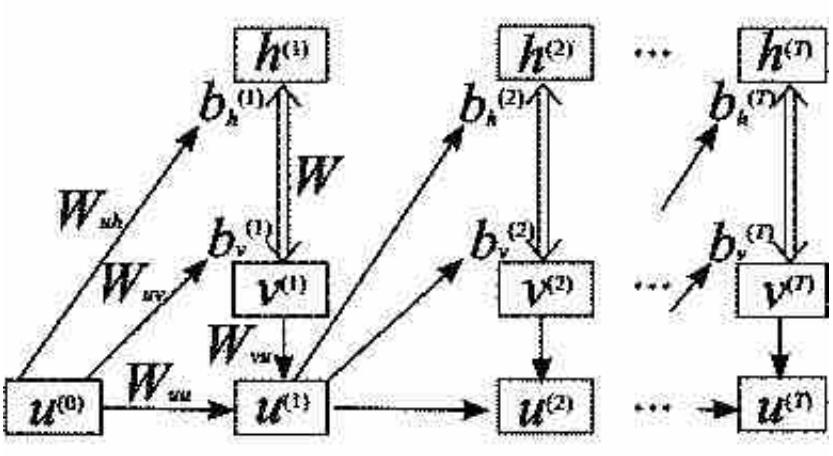

Fig. 6.16 RNN-RBM architecture. Reproduced from [11] with the permission of the authors

There is a specific training algorithm, which we will not detail here, please refer to [10]. During generation, each t time step of processing is as follows:

- compute the biases $b_v^{(t)}$ and $b_h^{(t)}$ of RBM$^{(t)}$, via Equations 6.1 and 6.2 respectively,
- sample from RBM$^{(t)}$ by using block Gibbs sampling to produce $v^{(t)}$, and
- feedforward the RNN with $v^{(t)}$ as the input, using the RNN hidden layer value $u^{(t-1)}$, in order to produce the RNN new hidden layer value $u^{(t)}$ via Equation 6.3, where
  - W_{vu} is the weight matrix and b_u the bias for the connexions between the input layer of the RBM and the hidden layer of the RNN; and
  - W_{uu} is the weight matrix for the recurrent connexions of the hidden layer of the RNN.

$$b_v^{(t)} = b_v + W_{uv}u^{(t-1)} \tag{6.1}$$

$$b_h^{(t)} = b_h + W_{uh}u^{(t-1)} \tag{6.2}$$

$$u^{(t)} = \tanh(b_u + W_{uu}u^{(t-1)} + W_{vu}v^{(t)}) \tag{6.3}$$

Note that the biases $b_v^{(t)}$ and $b_h^{(t)}$ of RBM$^{(t)}$ are variable for each time step, in other words they are *time dependent*, whereas the weight matrix W for the connexions between the visible and the hidden layer of RBM$^{(t)}$ is *shared* for all time steps (for all RBMs), i.e. it is *time independent*[29].

Four different corpora have been used in the experiments: Classical piano, Folk tunes, orchestral Classical music and J. S. Bach chorales. Polyphony varies from 0 to 15 simultaneous notes, with an average value of 3.9. A piano roll representation is used with many-hot encoding of 88 units representing pitches from A_0 to C_8. Discretization (time step) is a quarter note. All examples are aligned onto a single common tonality: C major or minor. An example of a sample generated in a piano roll representation is shown in Figure 6.17. The quality of the model has made RNN-RBM, summarized at Table 6.11, one of the reference architectures for polyphonic music generation.

[29] W_{uv}, W_{uh}, W_{uu} and W_{vu} weight matrices are also shared and thus time independent.

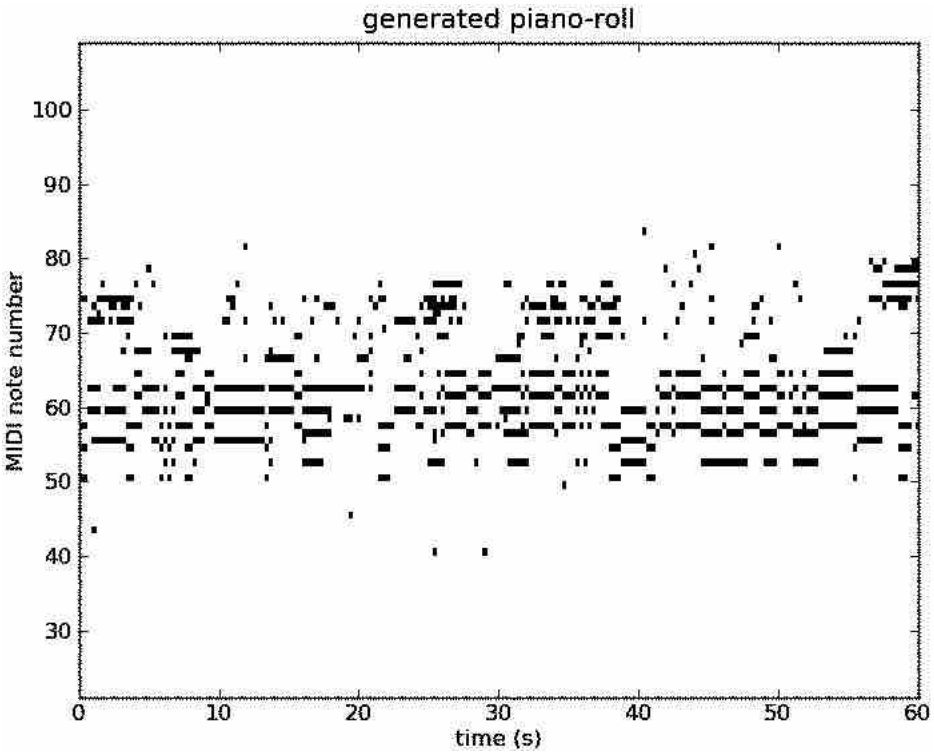

Fig. 6.17 Example of a sample generated by RNN-RBM trained on J. S. Bach chorales. Reproduced from [11] with the permission of the authors

Objective	Polyphony
Representation	Symbolic; Many-hot
Architecture	RBM-RNN
Strategy	Iterative feedforward; Sampling

Table 6.11 RNN-RBM summary

6.8.1.1 Other RNN-RBM Systems

There have been a few systems following on from and extending the RNN-RBM architecture, but they are not significantly different and furthermore they have not been thoroughly evaluated. However, it is worth mentioning the following:

- the RNN-DBN architecture[30], using multiple hidden layers [56]; and
- the LSTM-RTRBM architecture, using an LSTM instead of an RNN [111].

6.8.2 #2 Example: Hexahedria Polyphony Symbolic Music Generation Architecture

The system for polyphonic music proposed by Johnson in his Hexahedria blog [87] is hybrid and original in that it *integrates* into the same architecture

[30] This is apparently the state of the art for the J. S. Bach Chorales dataset in terms of cross-entropy loss.

- a first part made of two RNNs (actually LSTM) layers, each with 300 hidden units, recurrent over the *time dimension*, which are in charge of the *temporal* horizontal aspect, that is the relations between notes in a sequence. Each layer has connections across time steps, while being independent across notes; and
- a second part made of two other RNN (LSTM) layers, with 100 and 50 hidden units, recurrent over the *note dimension*, which are in charge of the *harmony* vertical aspect, that is the relations between simultaneous notes within the same time step. Each layer is independent between time steps but has transversal directed connexions between notes.

We can notate this architecture as LSTM^{2+2} in order to highlight the two successive 2-level recurrent layers, recurrent in two different dimensions (time and note). The architecture is actually a kind of integration within a single architecture[31] of the RNN-RBM architecture described in previous section. The main originality is in using recurrent networks not only on the time dimension but also on the note dimension, more precisely on the pitch class dimension. This latter type of recurrence is used to model the occurence of a simultaneous note based on other simultaneous notes. Like for the time relation, which is oriented – towards the future –, the pitch class relation is oriented – towards higher pitch, from C to B.

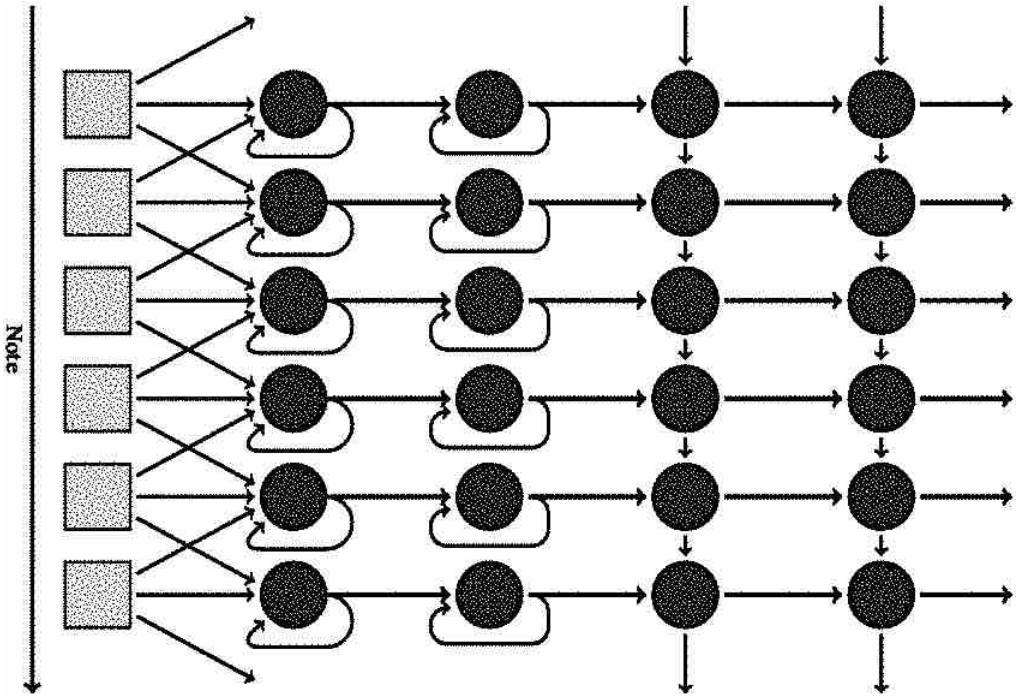

Fig. 6.18 Hexahedria architecture (folded). Reproduced from [87] with the permission of the author

The resulting architecture is shown in Figure 6.18 in its folded form and in Figure 6.19[32] in its unfolded form, with three axes represented:

- the *flow axis*, shown horizontally and directed from left to right, represents the flow of (feedforward) computation through the architecture, from the input layer to the output layer;
- the *note axis*, shown vertically and directed from top to bottom, represents the connexions between units corresponding to successive notes of each of the two last (note-oriented) recurrent hidden layers; and

[31] We will see in Section 6.8.3 an alternative architecture, named Bi-Axial LSTM, where each of the 2-level time-recurrent layers is encapsulated into a different architectural module.

[32] Our unfolded pictorial representation of an RNN shown in Section 5.11 was actually inspired by Johnson's Hexahedria pictorial representation.

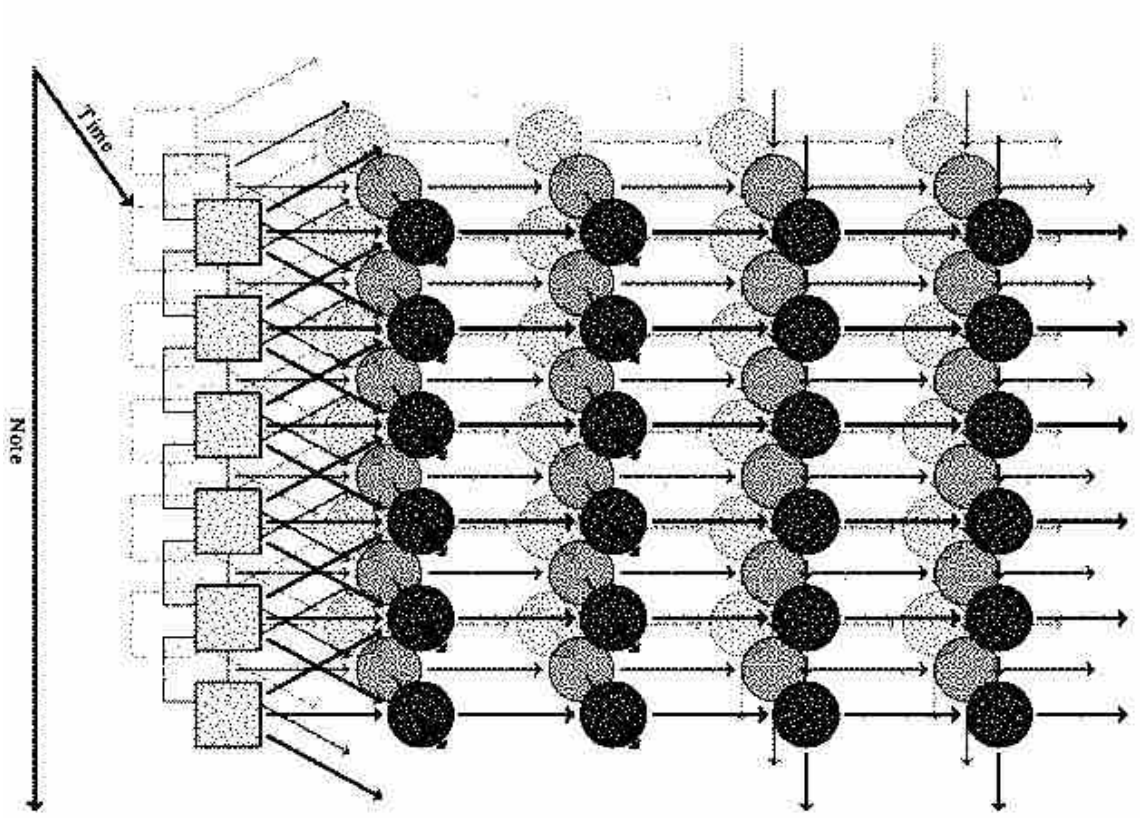

Fig. 6.19 Hexahedria architecture (unfolded). Reproduced from [87] with the permission of the author

- the *time axis*, only in the unfolded Figure 6.19, shown diagonally and directed from top left to bottom right, represents the time steps and the propagation of the memory within a same unit of the two first (time-oriented) recurrent hidden layers.

The dataset is constructed by extracting 8 measures long parts from MIDI files from the Classical piano MIDI database [96]. The input representation used is piano roll, with the pitch represented as the MIDI note number. More specific information is added: the pitch class, the previous note played (as a way to represent a possible hold), how many times a pitch class has been played in the previous time step and the beat (the position within the measure, assuming a 4/4 time signature). The output representation is also a piano roll, in order to represent the possibility of more than one note at the same time. Generation is done in an iterative way (i.e. following the iterative feedforward strategy), as for most recurrent networks. The system is summarized in Table 6.12.

Objective	Polyphony
Representation	Symbolic; Piano roll; Hold; Beat
Architecture	LSTM^{2+2}
Strategy	Iterative feedforward; Sampling

Table 6.12 Hexahedria summary

6.8.3 #3 Example: Bi-Axial LSTM Polyphony Symbolic Music Generation Architecture

Johnson recently proposed an evolution of his original Hexahedria architecture, described in Section 6.8.2, named Bi-Axial LSTM (or BALSTM) [88].

The representation used is piano roll, with note hold and rest tokens added to the vocabulary. Various corpora are used: the JSB Chorales dataset, a corpus of 382 four-part chorales by J. S. Bach [1]; the MuseData library, an electronic classical music library from CCARH in Stanford [71]; the Nottingham database, a collection of 1,200 folk tunes in ABC notation [49]; and the Classical piano MIDI database [96]. Each dataset is transposed (aligned) into the key of C major or C minor.

The probability of playing a note depends on two types of information:

- all notes at previous time steps – this is modeled by the *time-axis module*; and
- all notes within the current time step that have already been generated (the order being lowest to highest) – this is modeled by the *note-axis module*.

There is an additional front end layer, named "Note Octaves", which transforms each note into a vector of all its possible corresponding octave notes (i.e. an extensional version of pitch classes). The resulting architecture is illustrated in Figure 6.20[33]. The "x2" represents the fact that each module is stacked twice (i.e. has two layers).

The time-axis module is recurrent in time (as for a classical RNN), the LSTM weights being shared across notes in order to gain note transposition invariance. The note-axis module[34] is recurrent in note. For each note input of the note-axis module, $\oplus$ represents the concatenation of the corresponding output from the time-axis module with the already predicted lower notes. Sampling (into a binary value, by using a coin flip) is applied to each note output probability in order to compute the final prediction (whether that note is played or not).

As pointed out by Johnson [88], during the training phase, as all the notes at all time steps are known, the training process may be accelerated by processing each layer independently (e.g., on a GPU), by running input through the two time-axis layers in parallel across all notes, and using the two note-axis layers to compute probabilities in parallel across all time steps.

The generation phase is sequential and iterative for each time step (by following both the iterative feedforward strategy and the sampling strategy).

The Bi-Axial LSTM system, summarized at Table 6.13, has been evaluated and compared to some other architectures. The author reports noticeably better results with Bi-Axial LSTM, the greatest improvements being on the MuseData [71] and the Classical piano MIDI database [96] datasets, and states in [88] that: "It is likely due to the fact that those datasets contain many more complex musical structures in different keys, which are an ideal case for a translation-invariant architecture." Note that an extension of the Bi-Axial LSTM architecture with conditioning, named DeepJ, will be introduced in Section 6.9.3.4.

Objective	Polyphony
Representation	Symbolic; Piano roll; Hold; Rest
Architecture	Bi-Axial LSTM = LSTM$^2\times2$
Strategy	Iterative feedforward; Sampling

Table 6.13 Bi-Axial LSTM summary

[33] This figure comes from the description of another system based on the Bi-Axial LSTM architecture, named DeepJ, which will be described in Section 6.9.3.4.

[34] Note that, as opposed to Johnson's first architecture (that we refer to as Hexahedria, and which has been introduced in Section 6.8.2), which integrates the 2-level time-recurrent layers with the 2-level note-recurrent layers within a single architecture and therefore notated as LSTM^{2+2}, the Bi-Axial LSTM architecture explicitly separates each 2-level time-recurrent layers into distinct architectural modules and is therefore notated as LSTM$^2\times2$.

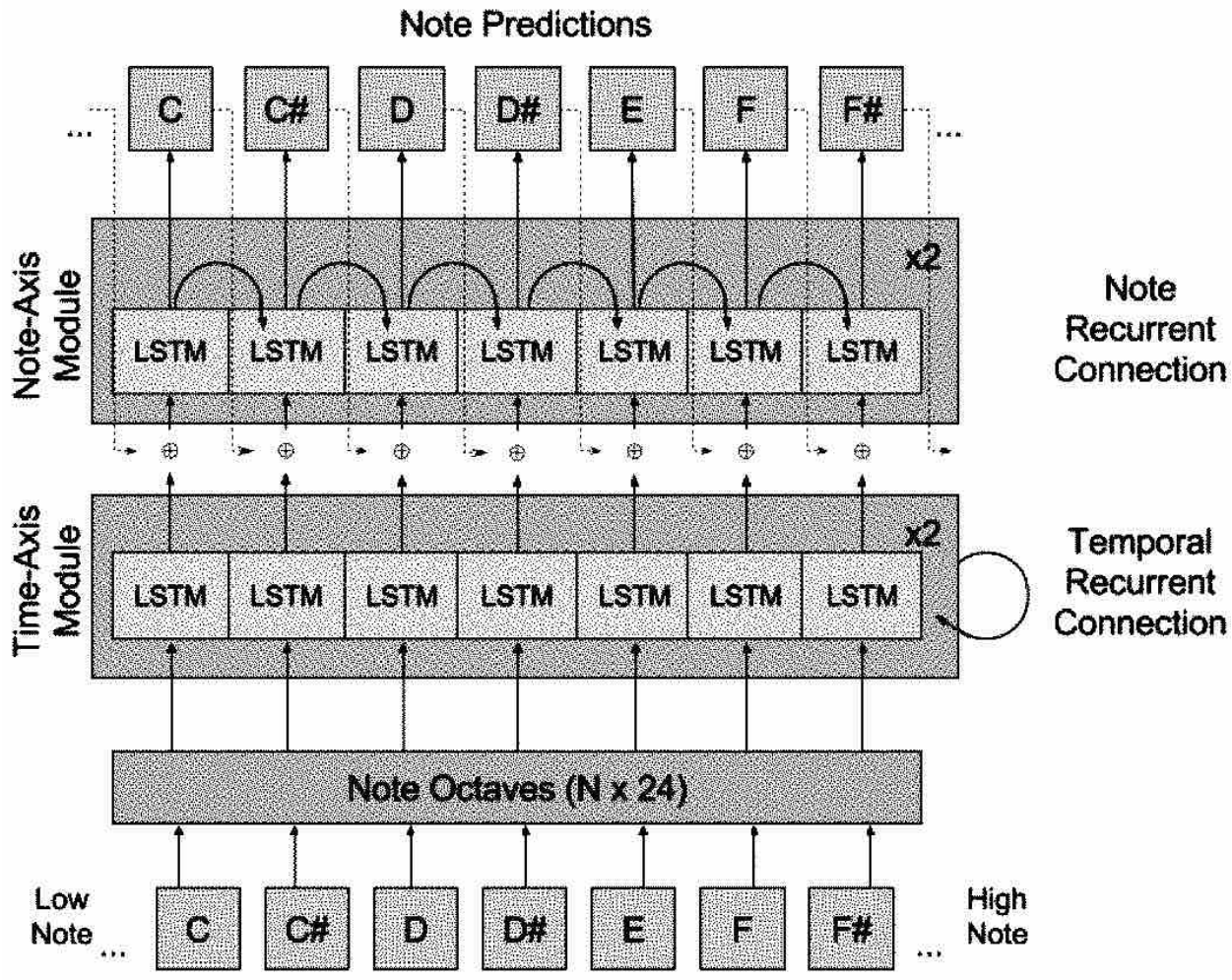

Fig. 6.20 Bi-Axial LSTM architecture. Reproduced from [116] with the permission of the authors

6.9 Control

A deep architecture generates musical content matching the style learnt from the corpus. This capacity of induction from a corpus without any explicit modeling or programming is an important ability, as discussed in Chapter 1 and also in [45]. However, like a fast car that needs a good steering wheel, control is also needed as musicians usually want to *adapt* ideas and patterns *borrowed* from other contexts to their own objective and context, e.g., transposition to another key, minimizing the number of notes, finishing with a given note, etc.

6.9.1 Dimensions of Control Strategies

Arbitrary control is a difficult issue for deep learning architectures and techniques because neural networks have not been designed to be controlled. In the case of Markov chains, they have an operational model on which one can attach constraints to control the generation[35]. However, neural networks do not offer such an operational entry point and the

[35] Two examples are Markov constraints [137] and factor graphs [136].

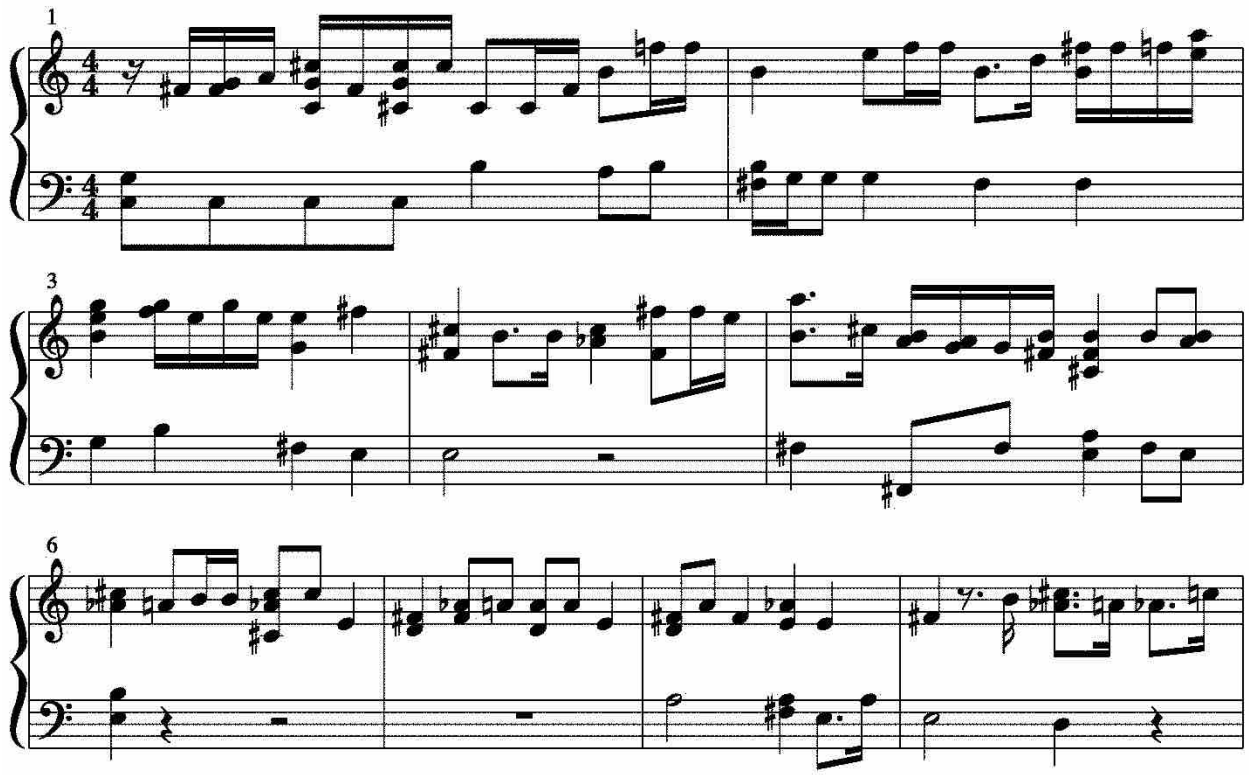

Fig. 6.21 Example of Bi-Axial LSTM generated music (excerpt). Reproduced from [88]

distributed nature of their representation does not provide a clear relation to the structure of the content generated. Therefore, as we will see, most of strategies for controlling deep learning generation rely on *external* intervention at various *entry points* (hooks) and *levels*:

- input,
- output,
- input *and* output, and
- encapsulation/reformulation.

Various control *strategies* can be employed:

- sampling,
- conditioning,
- input manipulation,
- reinforcement, and
- unit selection.

We will also see that some strategies (such as sampling, see Section 6.9.2) are more *bottom-up* and others (such as structure imposition, see Section 6.9.5.1, or unit selection, see Section 6.9.7) are more *top-down*. Lastly, there is also a continuum between *partial* solutions (such as conditioning/parametrization, see Section 6.9.3) and more *general* approaches (such as reinforcement, see Section 6.9.6).

6.9.2 Sampling

Sampling from a stochastic architecture (such as a restricted Boltzmann machine (RBM), see Section 6.3.2), or from a deterministic architecture (in order to introduce *variability*, see Section 6.5.1), may be an entry point for control if we introduce *constraints* into the sampling process. This is called *constrained sampling*, see for example the C-RBM system in Section 6.9.5.1.

Constrained sampling is usually implemented by a *generate-and-test* approach, where valid solutions are picked from a set of random samples generated from the model. But this could be a very costly process and, moreover, with no guarantee of success. A key and difficult issue is therefore how to *guide* the sampling process in order to fulfill the constraints.

6.9.2.1 Sampling for Iterative Feedforward Generation

In the case of an iterative feedforward strategy on a recurrent network, some refinements in the sampling procedure can be made.

In Section 6.5.1, we introduced the technique of sampling the softmax output of a recurrent network in order to introduce content variability. However, this may sometimes lead to the generation of an unlikely note (with a low probability). Moreover, as noted in [62], generating such a "wrong" note can have a cascading effect on the remaining of the generated sequence.

A counter measure consist in adjusting a learnt RNN model (conditional probability distribution $p(s_t|s_{<t})$, as defined in Section 5.11) by not considering notes with a probability under a certain threshold. The new model, with a probability distribution $p_{threshold}(s_t|s_{<t})$, is defined in Equation 6.4 following [184], where:

$$p_{threshold}(s_t|s_{<t}) := \begin{cases} 0 \text{ if } p(s_t|s_{<t})/max_{s_t} p(s_t|s_{<t}) < threshold, \\ p(s_t|s_{<t})/z \text{ otherwise.} \end{cases} \tag{6.4}$$

- $max_{s_t} p(s_t|s_{<t})$ is the note maximum probability,
- $threshold$ is the threshold hyperparameter, and
- z is a normalization constant.

A slightly more sophisticated version interpolates between the original distribution $p(x_t|x_{<t})$ and the $\text{argmax}_{x'_t} p(x'_t|x_{<t})$ deterministic variant[36], with some temperature user control hyperparameter (see more details in [62, Section 4.1.1.3]).

This technique will be further generalized and combined with the conditioning strategy in order to control the generation of notes at specific positions via positional constraints. This will be exemplified by the Anticipation-RNN system to be introduced in Section 6.9.3.5.

6.9.2.2 Sampling for Incremental Generation

In the case of an incremental generation (to be introduced in Section 6.13), the user may select

- on which part (e.g., a given part of a melody and/or a given voice) sampling will occur (or reoccur), and
- the interval of possible values on which sampling will occur.

In the case of the DeepBach system (to be introduced at Section 6.14.2), this will be the basis for introducing user control on the generation, notably to regenerate only some parts of a music, to restrict note range, and to impose some basic rhythm.

[36] See Section 6.5.1.

6.9.2.3 Sampling for Variational Decoder Feedforward Generation

Another interesting case is the use of sampling for *generative models*, such as variational autoencoders (VAEs) and generative adversarial networks (GANs), to be introduced in Section 6.9.2.4. We will see that some nice control of the sampling, e.g., to produce an interpolation, averaging or attribute modification, will produce meaningful variations in the content generated by the decoder feedforward strategy. Moreover, as has been discussed in Section 5.9.2, a variational autoencoder (VAE) is interesting for its ability for controlling generation over significant dimensions that have been learnt.

#1 Example: VRAE Video Game Melody Symbolic Music Generation System

In [43], Fabius and van Amersfoort propose the extension of the RNN Encoder-Decoder architecture to the case of a variational autoencoder (VAE), which is therefore named a variational recurrent autoencoder (VRAE). Both the encoder and the decoder encapsulate an RNN (actually an LSTM), as has been explained in Section 5.16.3. In terms of strategy, the VRAE combines the iterative feedforward strategy with the decoder feedforward strategy and the sampling strategy.

The corpus used in the experiment is a set of MIDI files of eight video game songs from the 1980 and 1990 (Sponge Bob, Super Mario, Tetris...), which are divided into various shorter parts of 50 time steps. A one-hot encoding of 49 possible pitches is used (pitches with too few occurrences of notes were not considered). Experiments have been conducted with 2 or 20 hidden layer units (latent variables). Training takes place as for training recurrent networks, i.e. for each input note presenting the next note as the output.

After the training phase, the latent space vector can be sampled and used by the decoder to iteratively generate a melody. This could be done by random sampling or also by interpolating between the values of the latent variables corresponding to different songs that have been learnt, creating a sort of "medley" of these songs. Figure 6.22 visualizes the organisation of the encoded data in the latent space, each color representing the data points from one song. The result is positive, but the low musical quality of the corpus hampers a careful evaluation. The VRAE system is summarized in Table 6.14.

Objective	Melody; Video game songs
Representation	Symbolic; MIDI; One-hot
Architecture	Variational(Autoencoder(LSTM, LSTM))
Strategy	Decoder feedforward; Iterative feedforward; Sampling

Table 6.14 VRAE summary

#2 Example: GLSR-VAE Melody Symbolic Music Generation System

The architecture proposed by Hadjeres and Nielsen in [64] is based on a variational autoencoder (VAE) architecture (Section 5.9.2), but it proposes an improvement in the control of the variation in the generation, named *geodesic latent space regularization* (GLSR), with a system named GLSR-VAE.

The starting point is that a straight line between two points in the latent space will not necessarily produce the *best* interpolation in the generated content domain space. The idea is to introduce a regularization to relate variations in the latent space to variations in the attributes of the decoded elements. The details of the definition of the added cost term may be found in [64].

The experiment consists in generating chorale melodies in the style of J. S. Bach. The dataset comprises monophonic soprano voices from the J. S. Bach chorales corpus [4].

GLSR-VAE shares the principles of representation initiated by the DeepBach system (Section 6.13.2), that is

- one-hot encoding of a note,

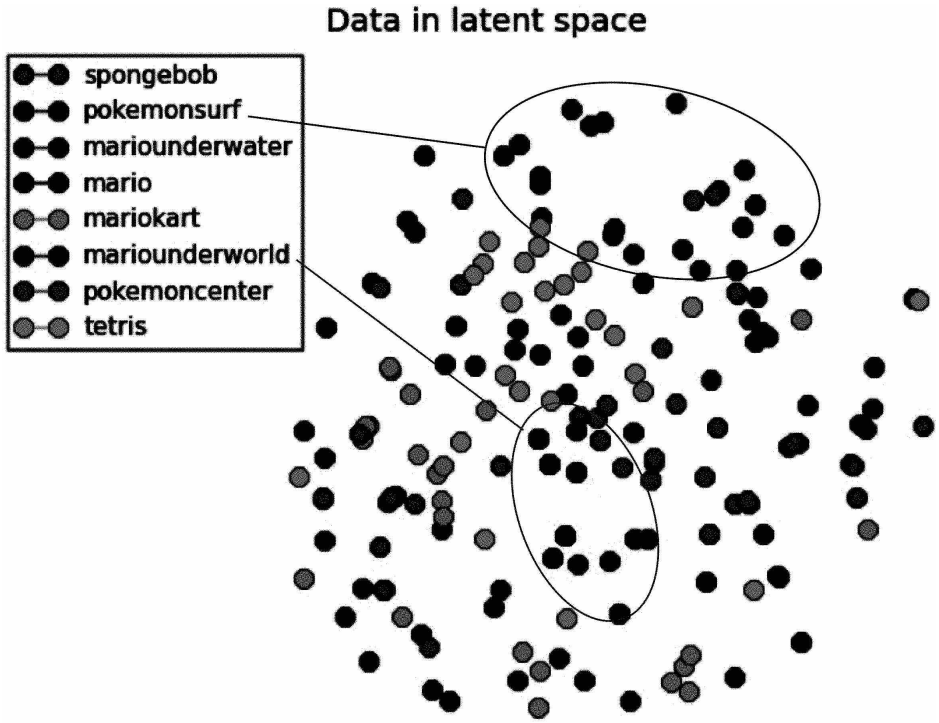

Fig. 6.22 Visualization of the VRAE latent space encoded data. Extended from [43] with the permission of the authors

- with the addition to the vocabulary of the hold symbol "_" and the rest symbol to specify, respectively, a note repetition and a rest (see Section 4.11.7), and
- using the names of the notes (with no enharmony, e.g., F♯ and G♭ are considered to be different, see Section 4.9.2).

Quantization is at the level of a sixteenth note. The latent variable space is set to 12 dimensions (12 latent variables).

In the experiments conducted, regularization is executed on a first dimension representing the number of notes (named z_1). Figure 6.23 shows the organisation of the encoded data in the latent space, with the number of notes z_1 being the abscissa axis, with from left to right an effective progressive increase in the number of notes (shown with scales of colors). Figure 6.24 shows examples of the melodies generated (each 2-measures long, separated by double bar lines) while increasing z_1, showing a progressive correlated densification of the melodies generated.

GLSR-VAE is summarized in Table 6.15. More examples of sampling from variational autoencoders will be described in Sections 6.11.1.

Objective	Melody; Bach
Representation	Symbolic; Piano roll; One-hot; Hold; Rest; Fermata; No enharmony
Architecture	Variational(Autoencoder(LSTM, LSTM)); Geodesic regularization
Strategy	Decoder feedforward; Sampling

Table 6.15 GLSR-VAE summary

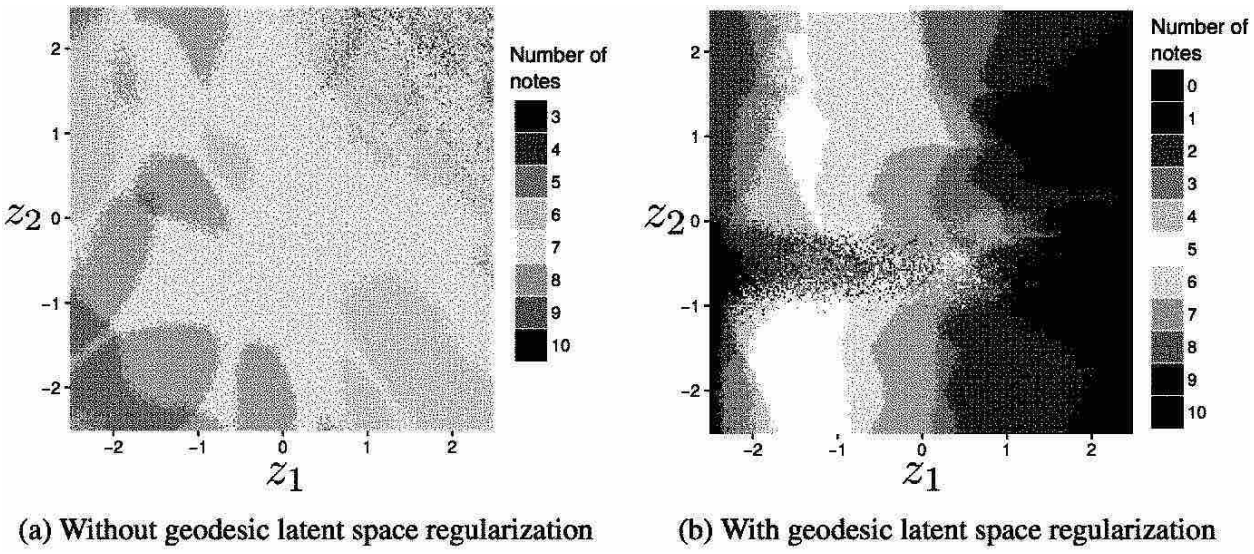

Fig. 6.23 Visualization of GLSR-VAE latent space encoded data. Reproduced from [64] with the permission of the authors

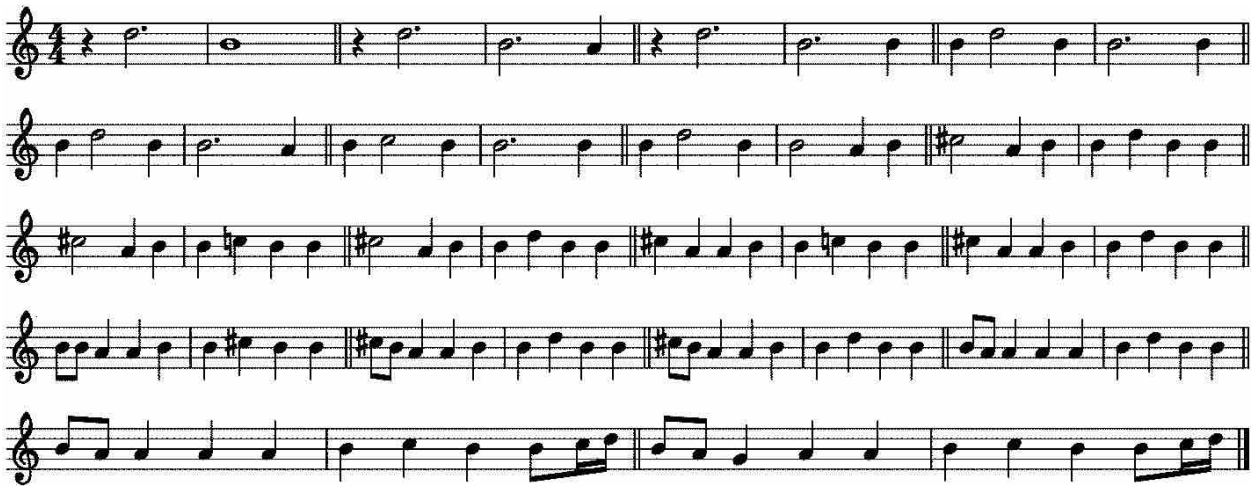

Fig. 6.24 Examples of 2-measures long melodies (separated by double bar lines) generated by GLSR-VAE. Reproduced from [64] with the permission of the authors

6.9.2.4 Sampling for Adversarial Generation

Another example of a generative model is a generative adversarial networks (GAN) architecture. In such an architecture, after having trained the generator in an adversarial way, generation of content is done by sampling latent random variables.

Example: Mogren's C-RNN-GAN Classical Polyphony Symbolic Music Generation System

The objective of Mogren's C-RNN-GAN [123] system is the generation of single voice polyphonic music. The representation chosen is inspired by MIDI and models each musical event (note) via four attributes: duration, pitch, intensity and time elapsed since the previous event, each attribute being encoded as a real value scalar. This allows the repre-

sentation of simultaneous notes (in practice up to three). The musical genre of the corpus is classical music, retrieved in MIDI format from the Web and contains 3,697 pieces from 160 composers.

C-RNN-GAN is based on a generative adversarial networks (GAN) architecture, with both the generator and the discriminator being recurrent networks[37], more precisely each having two LSTM layers with 350 units each. A specificity is that the discriminator (but not the generator) has a bidirectional recurrent architecture, in order to take context from both the past and the future for its decisions. The architecture is shown in Figure 6.25 and summarized in Table 6.16.

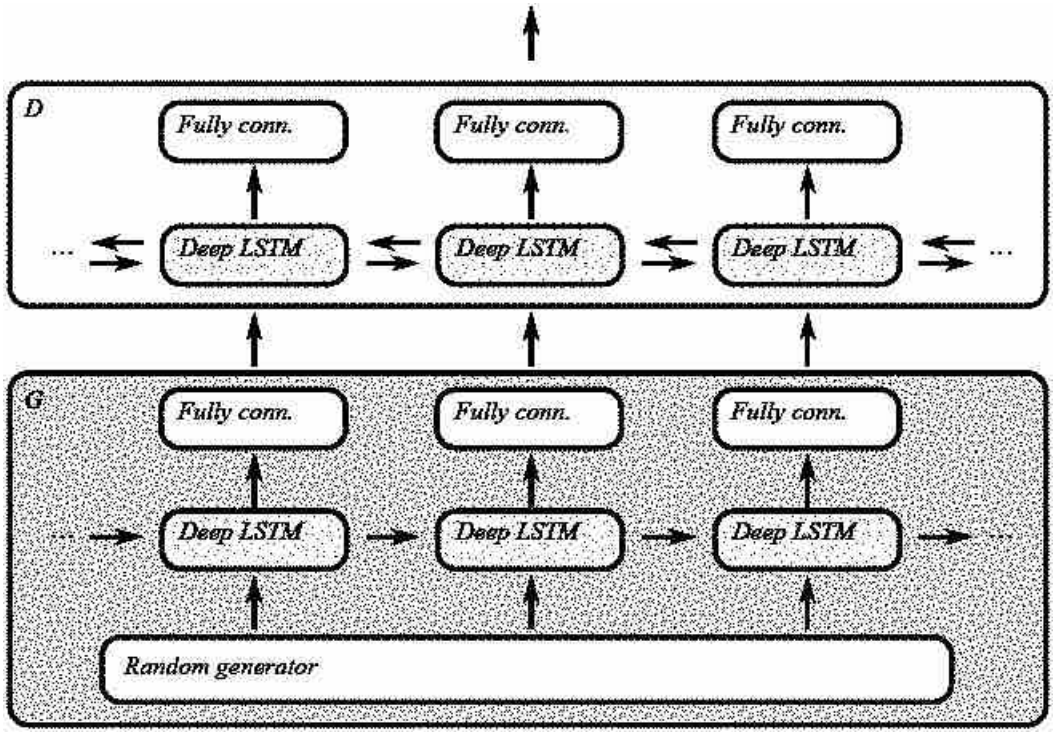

Fig. 6.25 C-RNN-GAN architecture. Reproduced from [123] with the permission of the authors

The discriminator is trained, in parallel to the generator, to classify if a sequence input is coming from the real data. Similar to the case of the encoder part of the RNN Encoder-Decoder, which summarizes a musical sequence into the values of the hidden layer (see Section 5.16.3), the bidirectional RNN decoder part of the C-RNN-GAN summarizes the sequence input into the values of the two hidden layers (forward sequence and backward sequence) and then classifies them.

An example of generated music is shown in Figure 6.26. The author conducted a number of measurements on the generated music. He states that the model trained with feature matching[38] achieves a better trade-off between structure and surprise than the other variants. Note that this is consonant with the use of the feature matching regularization technique to control creativity in MidiNet (to be introduced in Section 6.9.3.3). C-RNN-GAN is summarized in Table 6.16.

[37] This generative GAN architecture encapsulates two recurrent networks, in the same spirit that the generative VRAE variational autoencoder architecture encapsulates two recurrent networks as explained in Section 6.9.2.3.

[38] A regularization technique for improving GANs, see Section 5.14.1.

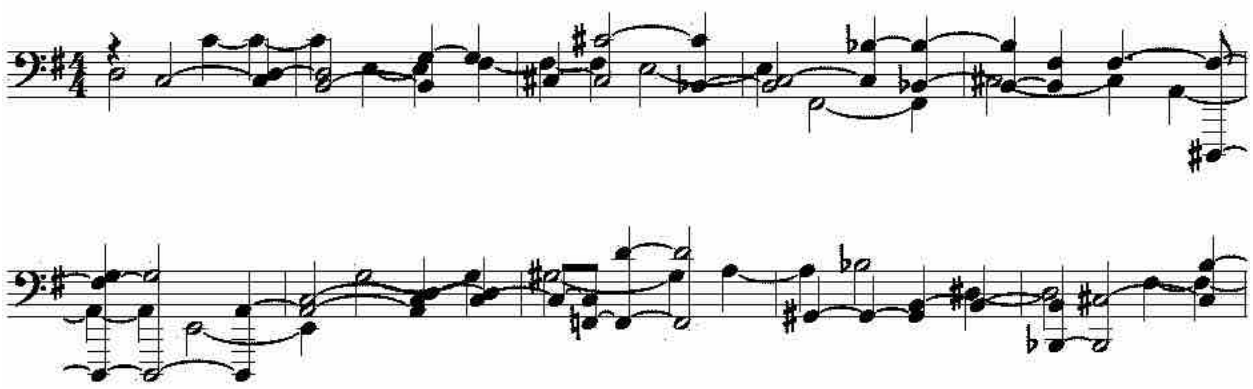

Fig. 6.26 C-RNN-GAN generated example (excerpt). Reproduced from [123] with the permission of the authors

Objective	Polyphony
Representation	Symbolic; MIDI; Value encoding[4]
Architecture	GAN(Bidirectional-LSTM[2], LSTM[2])
Strategy	Iterative feedforward; Sampling[39]

Table 6.16 C-RNN-GAN summary

6.9.2.5 Sampling for Other Generation Strategies

Sampling may also be combined with other strategies for content generation, as for instance

- *Conditioning*, as a way to *parametrize* generation with constraints, in Section 6.9.3.5, or
- *Input manipulation*, as a way to *correct* the manipulation performed in order to *realign* the samples with the learnt distribution, in Section 6.9.5.

6.9.3 Conditioning

The idea of *conditioning* (sometimes also named *conditional architecture*) is to condition the architecture on some extra 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 architecture (Section 6.9.3.1),
- a *chord progression*, in the MidiNet architecture (Section 6.9.3.3),
- the *previously generated note*, in the VRASH architecture (Section 6.9.3.6),
- some *positional constraints on notes*, in the Anticipation-RNN architecture (Section 6.9.3.5),
- a *musical genre* or an *instrument*, in the WaveNet architecture (Section 6.9.3.2), and
- a *musical style*, in the DeepJ architecture (Section 6.9.3.4).

In practice, the conditioning information is usually fed into the architecture as an additional input layer (for example, see Figure 5.30). This distinction between *standard input* and *conditioning input* follows a good architectural modularity principle[40]. Conditioning is a way to have some degree of parametrized control over the generation process.

The conditioning layer could be

- a simple input layer. An example is a tag specifying a musical genre or an instrument in the WaveNet system (Section 6.9.3.2),

[40] Note that we do not consider conditioning as a strategy because we consider that the essence of conditioning relates to the *conditioning architecture*, whereas we are concerned with the strategy relating to the *generation phase*. Generation uses a conventional strategy (e.g., single-step feedforward, iterative feedforward...) depending on the type of the architecture (e.g., feedforward, recurrent...).

- some output of some architecture, being
  - the same architecture, as a way to condition the architecture on some history[41] – an example is the MidiNet system (Section 6.9.3.3) in which history information from previous measure(s) is injected back into the architecture, or
  - another architecture – examples are the rhythm generation system (Section 6.9.3.1) in which a feedforward network in charge of the bass line and the metrical structure information produces the conditioning input, and the DeepJ system (Section 6.9.3.4) in which two successive transformation layers of a style tag produce an embedding used as the conditioning input.

If the architecture is time-invariant – i.e. recurrent or convolutional over time –, there are two options

- *global conditioning* – if the conditioning input is shared for all time steps, or
- *local conditioning* – if the conditioning input is specific to each time step.

The WaveNet architecture, which is convolutional over time (see Section 5.12.5), offers the two options, as will be analyzed in Section 6.9.3.2.

6.9.3.1 #1 Example: Rhythm Symbolic Music Generation System

The system proposed by Makris *et al.* [113] is specific in that it is dedicated to the generation of sequences of rhythm. Another specificity is the possibility to condition the generation relative to some particular information, such as a given beat or bass line.

The corpus includes 45 drum and bass patterns, each 16-measures long in 4/4 time signature, from three different rock bands and converted to MIDI. The representation of drums is described in Section 4.11.8 and summarized as follows. Different drum components (kick, snare, toms, hi-hat, cymbals) are considered as distinct simultaneous voices, following a many-hot approach, and encoded in text as a binary word of length 5, e.g., 10010 represents the simultaneous playing of kick and high-hat.

The representation also includes a condensed representation of the bass line part. It captures the voice leading perspective of the bass[42], by specifying the pitch difference direction for the bass between two successive time steps. This is represented in a binary word of length 4, the first digit specifying the existence of a bass event (1) or a rest event (0), while the three remaining digits specify the 3 possible directions for voice leading: steady (000), upward (010) and downward (001). Last, the representation includes some additional information representing the metrical structure (the beat structure), also through binary words. See further details in [113].

The architecture is a combination of a recurrent network (more precisely, an LSTM) and a feedforward network, representing the conditioning layer. The LSTM (two 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 information. The outputs of these two networks are then merged[43], resulting in the architecture illustrated in Figure 6.27. The authors report that the conditioning layer (bass line and beat information) improves the quality of the learning and of the generation. It may also be used in order to mildly influence the generation. More details may be found in the article [113]. The architecture is summarized in Table 6.17.

An example of a rhythm pattern generated is shown at Figure 6.28 with at Figure 6.29 the use of a specific and more complex bass line as a conditioning input which produces a rhythm more elaborate. The piano roll like visual representation shows in its five successive lines (downwards) the kick, snare, toms, hi-hat and cymbals components events.

[41] This is close in spirit to a recurrent architecture (RNN).

[42] The voice leading of the bass has proven a valuable aspect in harmonization systems, see, e.g., [67].

[43] Note that in this system, the conditioning layer is added to the main architecture at its output level and not at its input level. Therefore an additional feedforward merge layer is introduced. We could notate the resulting architecture as Conditioning(Feedforward(LSTM^2), Feedforward).

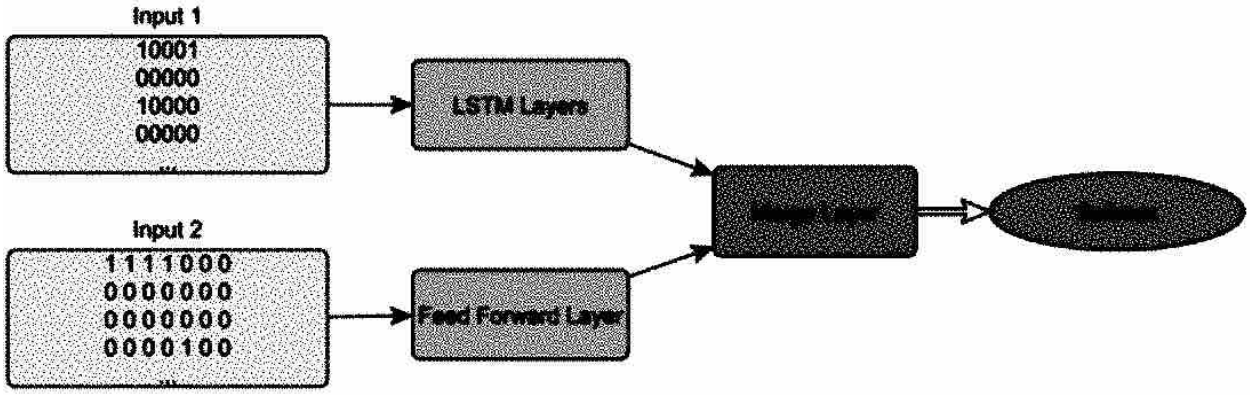

Fig. 6.27 Rhythm generation architecture. Reproduced from [113] with the permission of the authors

Fig. 6.28 Example of a rhythm pattern generated. Reproduced from [113] with the permission of the authors

Fig. 6.29 Example of a rhythm pattern generated with a specific bass line as the conditioning input. Reproduced from [113] with the permission of the authors

Objective	Multivoice; Rhythm; Drums
Representation	Symbolic; Beat; Drums; Many-hot; Bass line; Note; Rest; Hold
Architecture	Conditioning(Feedforward($LSTM^2$), Feedforward)
Strategy	Iterative feedforward; Sampling

Table 6.17 Rhythm system summary

6.9.3.2 #2 Example: WaveNet Speech and Music Audio Generation System

WaveNet, by van der Oord *et al.* [176], is a system for generating raw audio waveforms, quite innovative in that respect. It has been tested in three audio domains: multi-speaker, text-to-speech (TTS) and music.

The architecture is based on a convolutional feedforward network with no pooling layer. Convolutions are constrained in order to ensure that the prediction only depends on previous time steps, and are therefore named *causal convolutions*. The actual implementation is optimized through the use of *dilated convolution* (also called "à trous"), where the convolution filter is applied over an area larger than its length by skipping input values with a certain step. Incrementally dilated successive convolution layers[44] enable networks to have very large receptive fields with just a few layers while preserving the input resolution throughout the network as well as computational efficiency (see [176] for more details). The architecture is illustrated in Figure 6.30.

Another specificity of WaveNet is in the training/generation asymmetry: during the training phase, predictions for all time steps can be made in parallel, whereas during the generation phase, predictions are sequential and iterative (following the iterative feedforward strategy).

The WaveNet architecture is made conditioning, as a way to guide the generation, by adding an additional tag as a conditioning input. We could thus notate the architecture as Conditioning(Convolutional(Feedforward), Tag).

There are actually two options:

- *global conditioning*, if the conditioning input is shared for *all* time steps; and
- *local conditioning*, if the conditioning input is specific to *each* time step.

An example of conditioning for a text-to-speech application domain is to feed in linguistic features from different speakers, e.g., North American or Mandarin Chinese English speakers, in order to generate speech with a specific prosody.

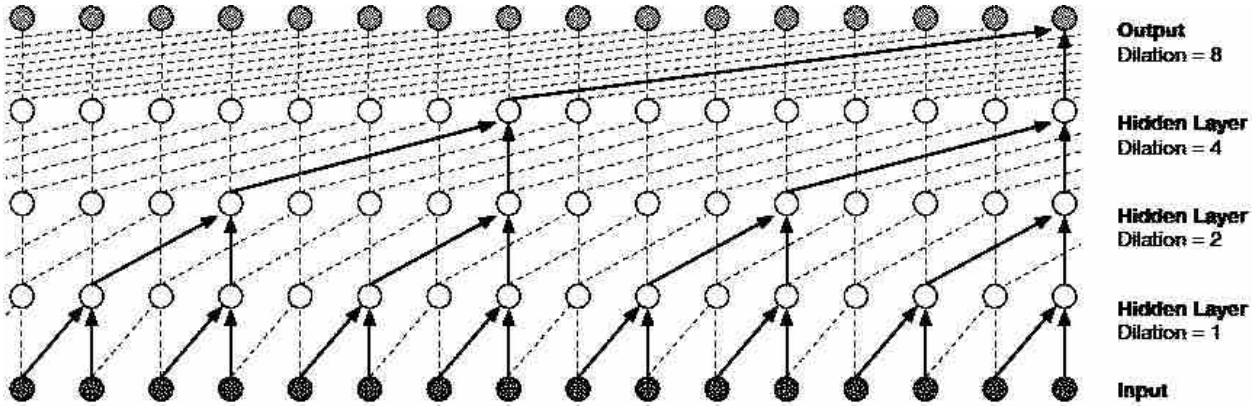

Fig. 6.30 WaveNet architecture. Reproduced from [176] with the permission of the authors

The authors also conducted preliminary work on conditioning models to generate music given a set of tags specifying, for example, genre or instruments. They state (without further details) that their preliminary attempt is promising [176]. WaveNet is summarized in Table 6.18.

6.9.3.3 #3 Example: MidiNet Pop Music Melody Symbolic Music Generation System

In [191], Yang *et al.* propose the MidiNet architecture, which is both adversarial and convolutional, for the generation of single or multitrack pop music monophonic melodies.

[44] The dilation is doubled for every layer up to a limit and then repeated, e.g., 1, 2, 4, . . . , 512, 1, 2, 4, . . . , 512, . . .

Objective	Audio
Representation	Audio; Waveform
Architecture	Conditioning(Convolutional(Feedforward), Tag); Dilated convolutions
Strategy	Iterative feedforward; Sampling

Table 6.18 WaveNet summary

The corpus used is a collection of 1,022 pop music songs from the TheoryTab[45] online database [79] that provides two channels per tab, one for the melody and the other for the underlying chord progression. This allows two versions of the system: one with only the melody channel and another that additionally uses chords to condition melody generation. After all the preprocessing steps, the dataset is composed of 526 MIDI tabs (representing 4,208 measures). Data augmentation is then performed by circularly shifting all melodies and chords to any of the 12 keys, leading to a final dataset of 50,496 measures of melody and chord pairs for training.

The representation is obtained by transforming each channel of MIDI files into a one-hot encoding of 8 measures long piano roll representations, using one of the encodings to represent silence (rest) and neglecting the velocity of the note events. The time step is set at the smallest note, a sixteenth note. All melodies have been transposed in order to fit within the two-octave interval $[C_4, B_5]$[46]. Note that the current representation does not distinguish between a long note and two short repeating notes, and the authors mention considering future extensions in order to emphasize the note onsets.

For chords, instead of using a many-hot vector extensional representation of dimension 24 (for the two octaves), the authors state that they found it more efficient to use an intensional representation of dimension 13: 12 for the pitch-class (key) and 1 for the chord type (major or minor).

The architecture[47] is illustrated in Figures 6.31 and 6.32. It is composed of a generator and a discriminator, which are both convolutional networks. The generator includes two fully-connected layers (with 1,024 and 512 units respectively) followed by four convolutional layers. The generator is conditioned by a module (named Conditioner CNN in Figure 6.31) which includes four convolutional layers with a reverse architecture. The conditioning mechanism incorporates

- history information from previous measures (as a memory mechanism, analog to a RNN), and
- the chord sequence (only for the generator). The discriminator includes two convolutional layers followed by some fully connected layers and the final output activation function is cross-entropy.

The discriminator is also conditioned, but without specific conditioner layers. We could thus notate the architecture as

GAN(Conditioning(Convolutional(Feedforward),
Convolutional(Feedforward(History, Chord sequence))),
Conditioning(Convolutional(Feedforward), History)).

The conditioning information could be

- only about the previous measure – named "1D conditions" (shown in yellow in Figures 6.31 and 6.32); or
- about various previous measures – named "2D conditions" (shown in blue).

Both cases are illustrated in Figure 6.31. The authors report experiments performed with different variants:

- melody generation with conditioning on the previous measure (with previous measure as 2D conditions for the generator and as 1D conditions for the discriminator[48]);

[45] Tabs are piano roll-like leadsheets, including melody, lyrics and notation of chords.

[46] However, the authors considered all the 128 MIDI note numbers (corresponding to the $[C_0, G_{10}]$ interval) in a one-hot encoding and state in [191] that: "In doing so, we can detect model collapsing more easily, by checking whether the model generates notes outside these octaves."

[47] The architecture is complex, please see further details in [191].

[48] To ensure that the discriminator distinguishes between real and generated melodies only from the present measure.

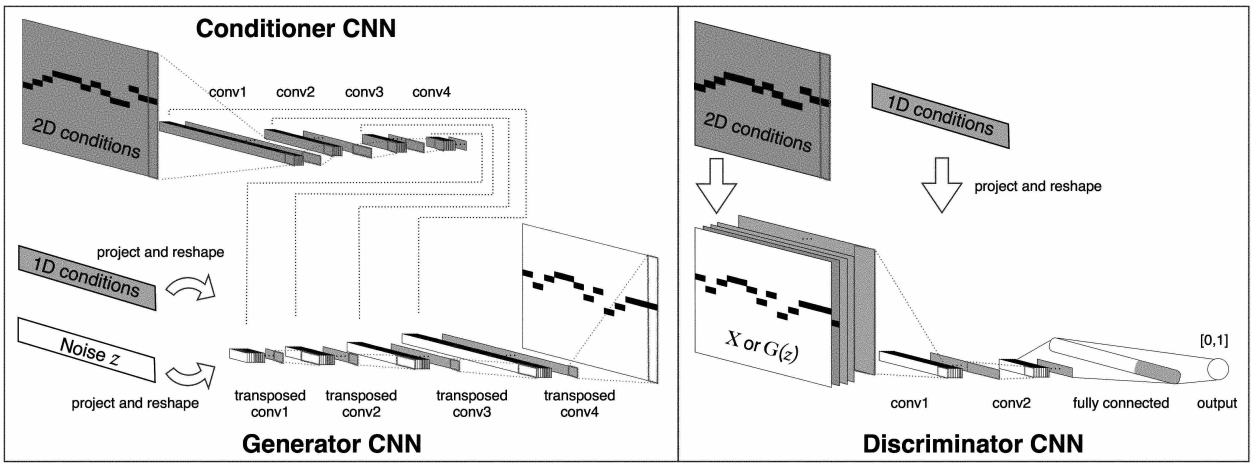

Fig. 6.31 MidiNet architecture. Reproduced from [191] with the permission of the authors

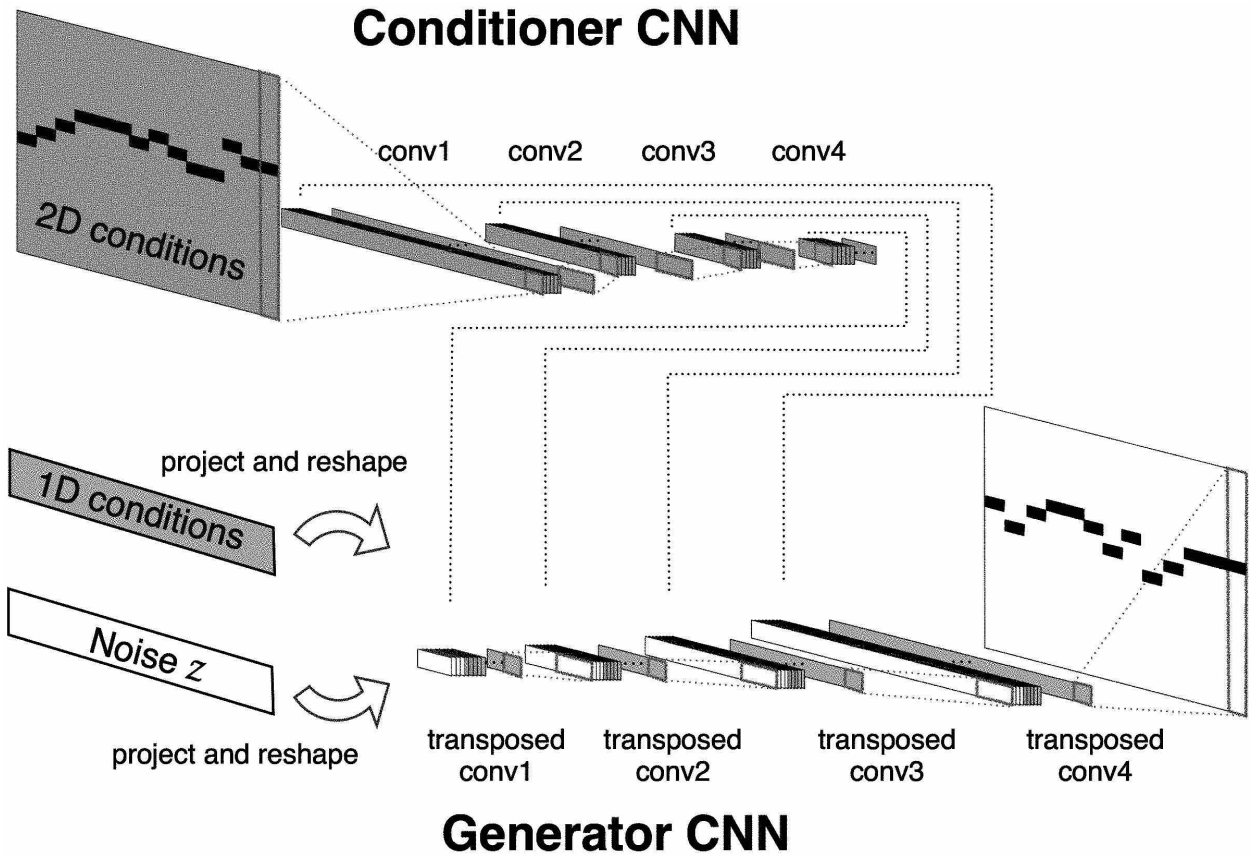

Fig. 6.32 Architecture of the MidiNet generator. Reproduced from [191] with the permission of the authors

- melody generation with conditioning on the previous measure and on the chord sequence (with chord sequence as 1D conditions for the generator, or alternatively also as 2D conditions only for its last convolutional layer in order to highlight the chord condition); and
- melody generation with conditioning on the previous measure and on the chord sequence in a creative mode (with chord sequence as 2D conditions for all convolutional layers of the generator).

For the second variant, which they name *stable mode*, the authors report that the generation is more chord-dominant and stable, in other words it closely follows the chord progression and seldom generates notes violating chord constraints. For the third variant, named *creative mode*[49], the generator sometimes violates the constraint imposed by the chords, to better adhere to the melody of the previous measure. In other words, the creative mode allows a better balance between melody following over chord following. The authors state in [191] that: "Such violations sometimes sound unpleasant, but can be sometimes creative. Unlike the previous two variants, we need to listen to several melodies generated by this model to handpick good ones. However, we believe such a model can still be useful for assisting and inspiring human composers."

MidiNet in summarized in Table 6.19.

Objective	Melody + Chords; Pop music; Melody vs chords following balance
Representation	Symbolic; Chords; Piano roll; One-hot; Rest
Architecture	GAN(Conditioning(Convolutional(Feedforward), Convolutional(Feedforward(History, Chord sequence))), Conditioning(Convolutional(Feedforward), History))
Strategy	Iterative feedforward; Sampling

Table 6.19 MidiNet summary

6.9.3.4 #4 Example: DeepJ Style-Specific Polyphony Symbolic Music Generation System

In [116], Mao *et al.* propose a system named DeepJ, with the objective of being able to control the style of music generated. In their experiment, they consider 23 styles, each corresponding to a different composer (from Johann Sebastian Bach to Pyotr Ilyich Tchaikovsky) with his/her specific style[50]. They encode the style – or a combination of styles[51] – as a many-hot representation over all possible styles (i.e. composers). Composers are grouped into musical genres. Thus a genre is specified (extensionally) as an equal combination of the styles (composers) of that genre. For example, if the Baroque genre is defined by composers 1 to 4, the Baroque style would be equal to $[0.25, 0.25, 0.25, 0.25, 0, 0, ...]$. We will see below, when detailing the architecture, that this somewhat simplistic user-defined style encoding will be automatically transformed through the learning phase into an adaptive distributed representation.

The foundation of the architecture is the Bi-Axial LSTM architecture proposed by Johnson in [88] (see Section 6.8.3). Music representation is based on piano roll, modeling a note through its MIDI note number, within a truncated range (originally within the $\{0, 1, \dots\ 127\}$ discrete set, truncated to $\{36, 37, \dots\ 84\}$, i.e. four octaves) in order to reduce note input dimensionality. Quantization is 16 time steps per measure, i.e. a time step with the value of a sixteenth note. The representation is similar to that for Bi-Axial LSTM. DeepJ representation uses a replay matrix, dual to the piano roll matrix of notes, in order to distinguish between a held note and a replayed note. DeepJ representation also includes information about dynamics through a scalar variable[52] within the $[0, 1]$ interval. But the main addition is the use of *style conditioning*, via global conditioning[53], as in WaveNet.

[49] On the challenge of creativity, see Section 6.12.

[50] In other words, they identify a style to a composer.

[51] In the case of a combination of several styles, the vector must be normalized in order for its sum to be equal to 1.

[52] The authors comment that they have also tried an alternate representation of dynamics as a categorical value (one-hot encoding) with 128 bins (as in WaveNet, see Section 6.9.3.2), which is actually the original MIDI discretization. But: "Contrary to WaveNet's results, our experiments concluded that the scalar representation yielded results that were more harmonious." [116]

[53] This means that the conditioning input is shared for *all* time steps, see Section 6.9.3.2.

As has been noted, the user-defined style encoding is too simplistic to be used as it is. Musical styles are not necessarily orthogonal to each other and may share many characteristics. The first transformation layer linearly transforms the user-defined many-hot encoding of the style into a first embedding (a set of hidden/latent variables, pictured as the yellow Embedding box in Figure 6.33). The second transformation layer transforms this first embedding in a nonlinear way (through a tanh activation[54]) into a second embedding of the style (pictured as the lower yellow Fully-Connected box) to be added as a conditioning input to the time-axis module. A similar transformation and conditioning is performed for the note-axis module. Further details and discussion may be found in [116]. DeepJ is summarized at Table 6.20.

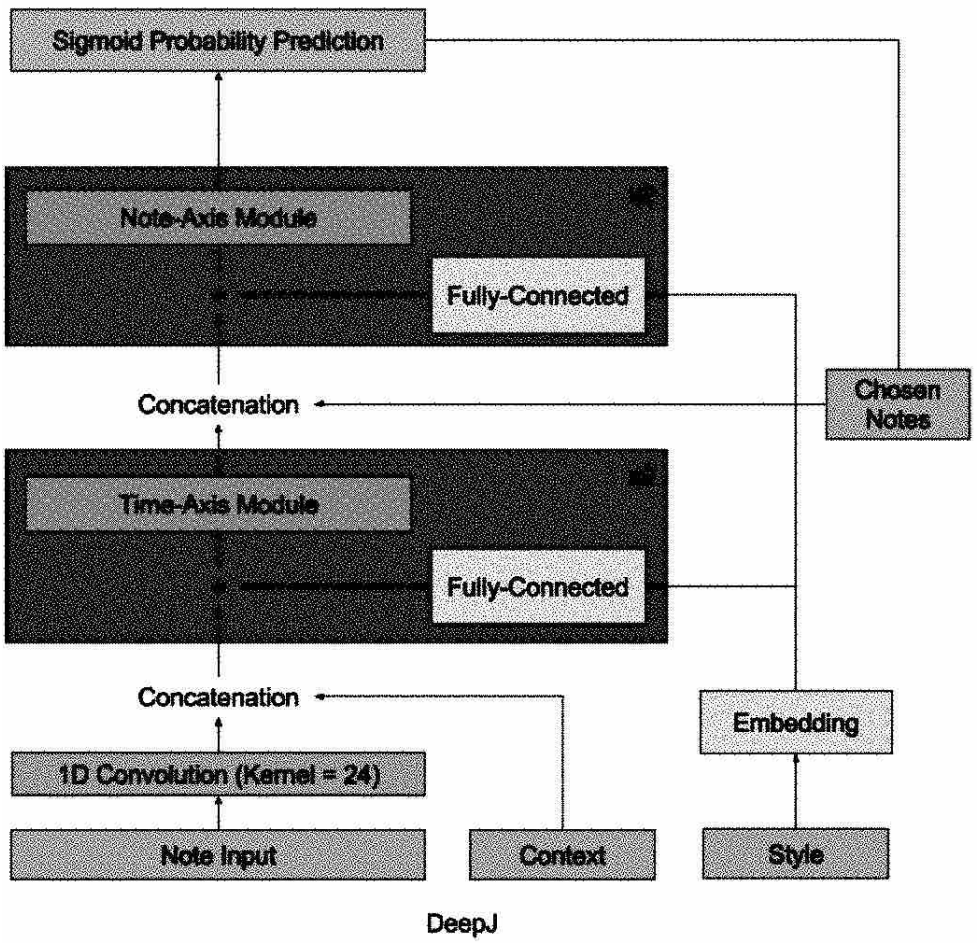

Fig. 6.33 DeepJ architecture. Reproduced from [116] with the permission of the authors

The authors have conducted an initial subjective evaluation with human listeners comparing music generated by DeepJ (an example is shown in Figure 6.34) and by Bi-Axial LSTM. They report that DeepJ compositions were usually preferred and they comment that the style conditioning makes generated music more stylistically consistent. They also conducted a second subjective evaluation in order to verify whether DeepJ can generate stylistically distinct music (correctly identified by human listeners). The authors report no statistically significant differences between the classification accuracy for DeepJ music and real composers music. A more objective analysis has also been undertaken by visualizing the style embedding space, shown in Figure 6.35, with each composer pictured as a dot and each cluster as a color (blue, yellow and red are for baroque, classical and romantic clusters, respectively). The authors found that

[54] Hyperbolic tangent function.

composers from similar periods do cluster together (same color) and point out the interesting result that Ludwig van Beethoven appears at the limit between the classical and romantic clusters.

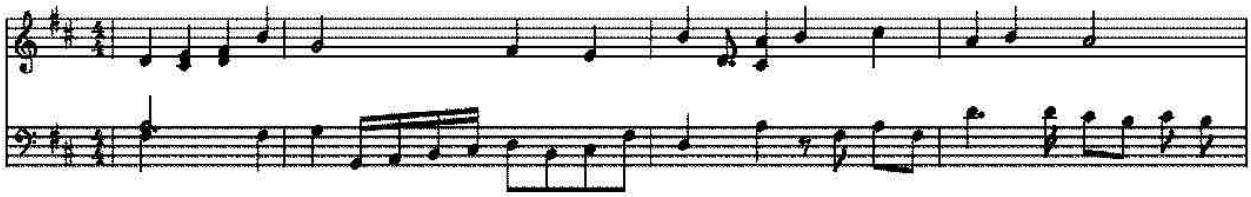

Fig. 6.34 Example of baroque music generated by DeepJ. Reproduced from [116] with the permission of the authors

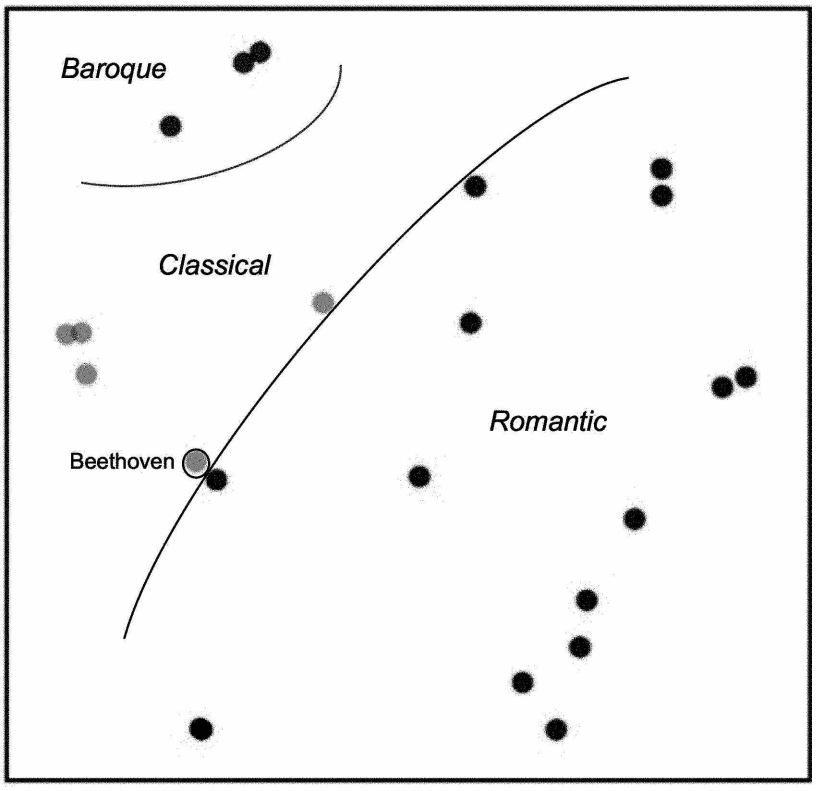

Fig. 6.35 Visualization of DeepJ embedding space. Extended from [116] with the permission of the authors

Objective	Polyphony; Classical; Style
Representation	Symbolic; Piano roll; Replay matrix; Rest; Style; Dynamics
Architecture	Conditioning(Bi-Axial LSTM, Embedding) = Conditioning($\text{LSTM}^2 \times 2$, Embedding)
Strategy	Iterative feedforward; Sampling

Table 6.20 DeepJ summary

6.9.3.5 #5 Example: Anticipation-RNN Bach Melody Symbolic Music Generation System

In [63], Hadjeres and Nielsen propose a system named Anticipation-RNN for generating melodies with unary constraints on notes (to enforce a given note at a given time position to have a given value). The limitation when using a standard iterative feedforward strategy for generation is that enforcing the constraint at time i may retrospectively invalidate the distribution of the previously generated items[55], as shown in [139]. The idea is then to condition the RNN on information summarizing the set of further (in time) constraints, as a way to anticipate oncoming constraints, in order to generate notes with a correct distribution.

Therefore, a second RNN architecture, named Constraint-RNN, is used that functions backward in time. Its outputs are used as additional inputs for the main RNN, which the authors name Token-RNN. The complete architecture is illustrated in Figure 6.36, with the following notation and meaning:

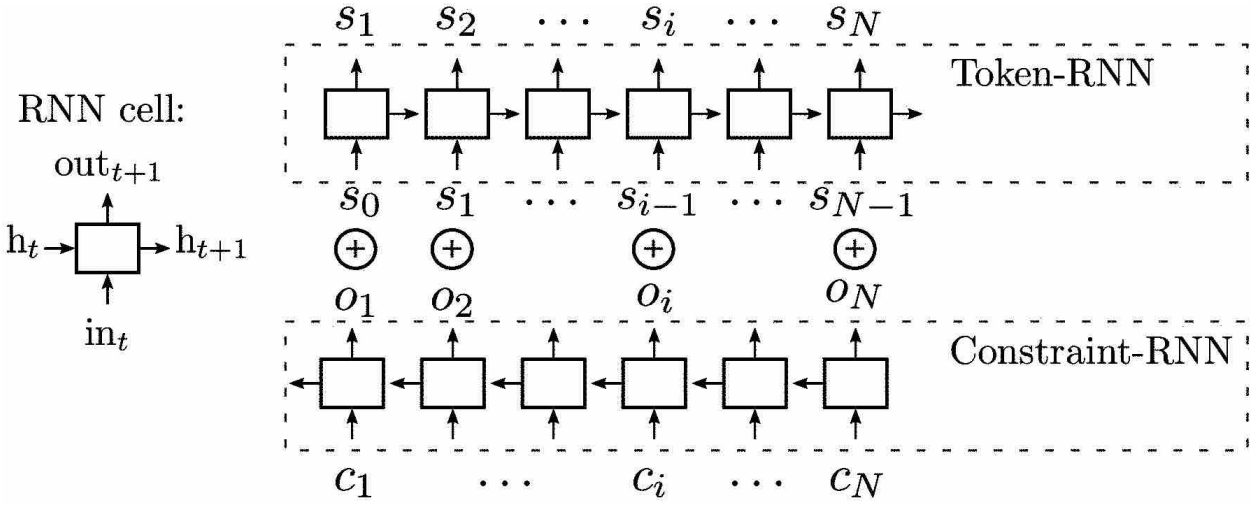

Fig. 6.36 Anticipation-RNN architecture. Reproduced from [63] with the permission of the authors

- c_i is a *positional constraint*; and
- o_i is the output at index i (after i iterations) of Constraint-RNN – it summarizes constraints information from step i to the final step N (the end of the sequence). It will be concatenated ($\oplus$) to input s_{i-1} of Token-RNN in order to predict the next item s_i.

Note that Anticipation-RNN is not a symmetric bidirectional recurrent architecture, as in the case of the C-RNN-GAN architecture analyzed in Section 6.9.2.4, because what is processed backwards is another sequence (of the constraints associated to the first sequence). Both RNNs (Constraint-RNN and Token-RNN) are implemented as a 2-layer LSTM.

The corpus used is the set of soprano voice melodies extracted from the four-voice Chorales of J. S. Bach. Data synthesis is performed by transposing in all keys within the original voice range and by pairing them with some sorted set of constraints[56].

Anticipation-RNN shares the principles of representation initiated by the DeepBach system to be presented in Section 6.13.2, that is one-hot encoding with the addition of the hold symbol "__" and the rest symbol to specify, respectively, a note repetition and a rest, and using the names of the notes with no enharmony. Quantization is at the level of a sixteenth note.

[55] As the authors put it, imposing a constraint on time index i "twists" the conditional probability distribution $p(s_t|s_{<t})$ for $t < i$.

[56] This is done to reduce the combinatorial explosion, as one does not need to construct all possible pairs $(melody, constraint)$ as long as the coverage is sufficient for good learning.

Three examples of melodies generated with the same set of positional constraints (each one indicated with a green note within a green rectangle) are shown in Figure 6.37. The model is indeed able to anticipate each positional constraint by adjusting its direction towards the target (lower-pitched or higher-pitched note). Further details and analysis of the results are provided in [63]. Anticipation-RNN is summarized in Table 6.21.

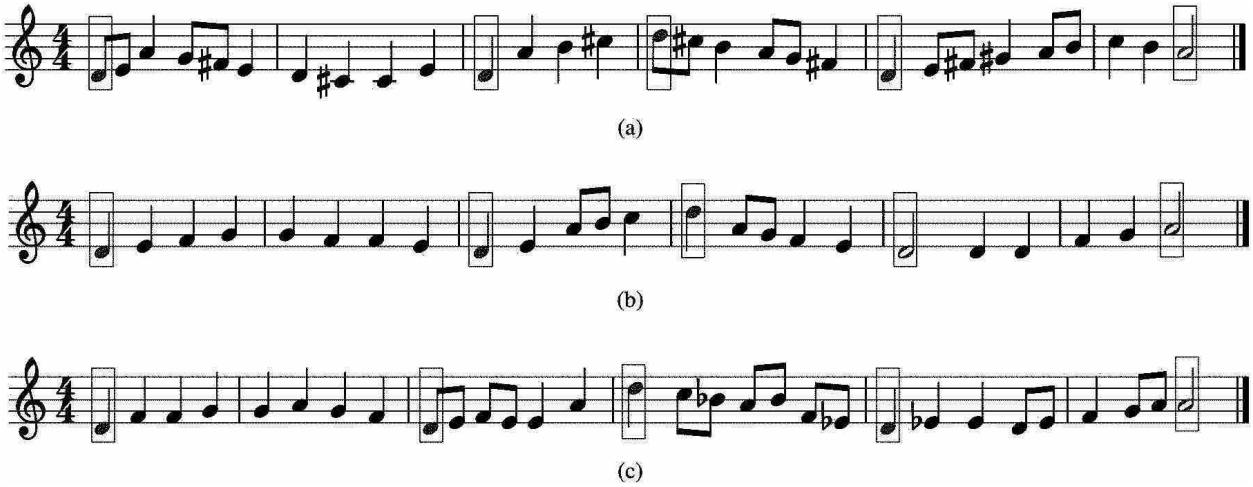

Fig. 6.37 Examples of melodies generated by Anticipation-RNN. Reproduced from [63] with the permission of the authors

Objective	Melody; Bach
Representation	Symbolic; One-hot; Hold; Rest; No enharmony
Architecture	Conditioning(LSTM^2, LSTM^2)
Strategy	Iterative feedforward; Sampling

Table 6.21 Anticipation-RNN summary

6.9.3.6 #6 Example: VRASH Melody Symbolic Music Generation System

The system described by Tikhonov and Yamshchikov in [171], although similar to VRAE (see Section 6.9.2.3), uses a different representation, separately encoding in a multi-one-hot manner the pitch, the octave and the duration. The training dataset is composed of various songs (different epochs and genres), derived from MIDI files following filtering and normalization (see the details in [171]). The architecture has four LSTM[57] layers for the encoder and for the decoder.

The authors have experimented with feeding the output of the decoder back into the decoder as a way of including the previously generated note as an additional information (therefore, they have named their final architecture VRASH, for variational recurrent autoencoder supported by history). It is illustrated in Figures 6.38 and 6.39 and summarized in Table 6.22. In their evaluation, the authors 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.

[57] To be more precise, a recent evolution named recurrent highway networks (RHNs) [192].

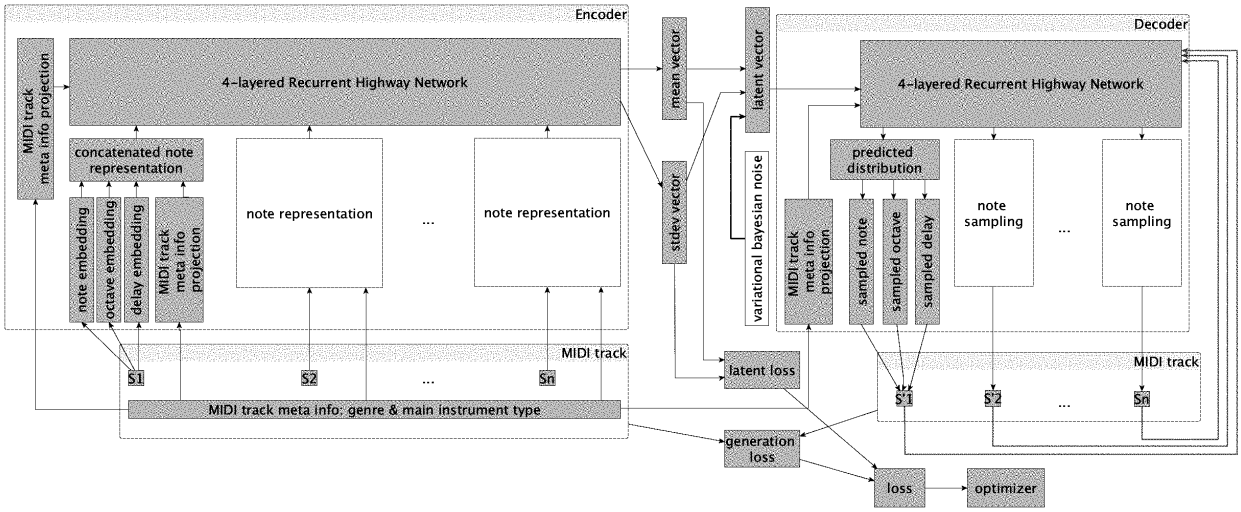

Fig. 6.38 VRASH architecture. Reproduced from [171] with the permission of the authors

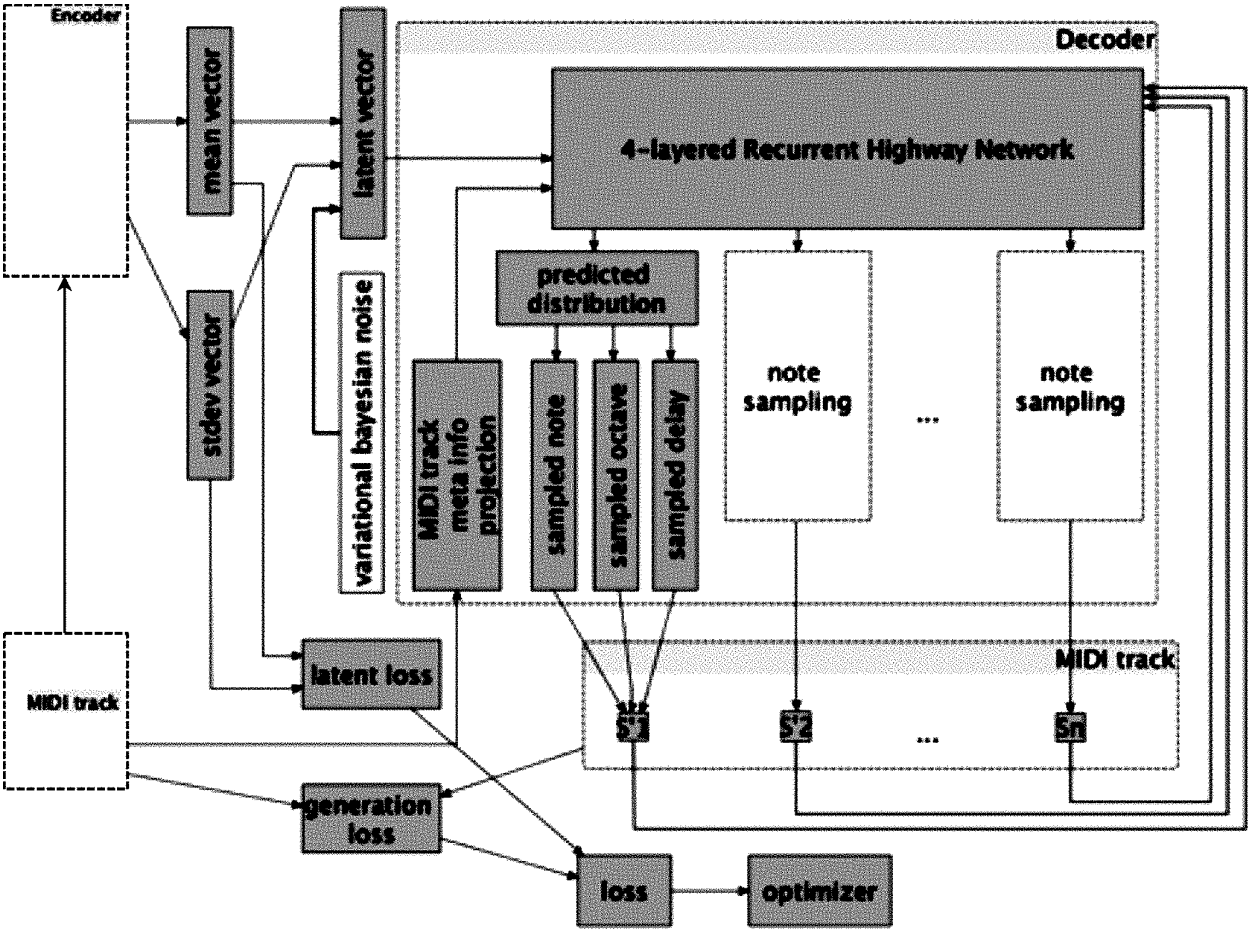

Fig. 6.39 VRASH architecture with a focus on the decoder. Extended from [171] with the permission of the authors

Objective	Melody
Representation	Symbolic; MIDI; Multi-one-hot
Architecture	Variational(Autoencoder(LSTM^4, Conditioning(LSTM^4, History)))
Strategy	Decoder feedforward; Iterative feedforward; Sampling

Table 6.22 VRASH summary

6.9.4 Input Manipulation

The *input manipulation* strategy was pioneered for images by Deep Dream. The idea is that the initial input content, or a brand new (randomly generated) input content, is incrementally *manipulated* in order to match a target *property*. Note that control of the generation is *indirect*, as it is not applied to the output but to the input, *before* generation. Examples of target properties are

- maximizing the *similarity* to a given *target*, in order to create a consonant melody, as in DeepHear$_C$ (Section 6.9.4.1);
- maximizing the *activation* of a specific *unit*, to amplify some visual element associated to this unit, as in Deep Dream (Section 6.9.4.3);
- maximizing the *content similarity* to some initial image *and* the *style similarity* to a reference style image, to perform *style transfer* (Section 6.9.4.4); and
- maximizing the *similarity* of the *structure* to some reference music, to perform *style imposition* (Section 6.9.5.1).

Interestingly, this is done by reusing standard training mechanisms, namely backpropagation to compute the gradients, as well as gradient descent (or ascent) to minimize the cost (or to maximize the objective).

6.9.4.1 #1 Example: DeepHear Ragtime Counterpoint Symbolic Music Generation System

In [164], in addition to the generation of melodies (described in Section 6.3.1.1), Sun proposed to use DeepHear for a different objective: to harmonize a melody, while using the *same* architecture as well as what has already been learnt[58]. We notate this second experiment DeepHear$_C$, where C stands for counterpoint, in order to distinguish it from DeepHear$_M$ for melody generation (Section 6.3.1.1).

The idea is to find a label instance of the embedding, i.e. a set of values for the 16 units of the bottleneck hidden layer of the stacked autoencoder, which will result in a decoded output resembling a given melody. Therefore, a simple distance (error) function is defined to represent the distance (similarity) between two melodies (in practice, the number of unmatched notes). Then a gradient descent is conducted on the variables of the embedding, guided by the gradients corresponding to the error function, until a sufficiently similar decoded melody is found.

Although this is not a real counterpoint[59], but rather the generation of a similar (consonant) melody, the results (tested on ragtime melodies) do produce a naive counterpoint with a ragtime flavor.

Note that in DeepHear$_C$ (summarized in Table 6.23), the input manipulated is the input of the innermost decoder (the starting point of the chain of decoders) and not the main input of the full architecture. Whereas, in the case of the Deep Dream system to be introduced in Section 6.9.4.3, this is the main input of the full (feedforward) architecture which is manipulated.

6.9.4.2 Relation to Variational Autoencoders

Note that in the case of the manipulation of the hidden layer units of an autoencoder (or a stacked autoencoder, the case of DeepHear$_C$), the input manipulation strategy does have some analogy with variational autoencoders, such as

[58] It is a simple example of *transfer learning* (see [58, Section 15.2]), using the same domain and the same training but for a different task.

[59] As, for example, in the case of MiniBach (Section 6.1.2) or DeepBach (Section 6.13.2) for real counterpoint generation.

Objective	Accompaniment; Ragtime
Representation	Symbolic; Piano roll; One-hot
Architecture	Autoencoder[4]
Strategy	Input manipulation; Decoder feedforward

Table 6.23 DeepHear$_C$ summary

for instance the VRAE system (Section 6.9.2.3) or the GLSR-VAE system (Section 6.9.2.3). Indeed in both cases, there is some exploration of possible values for the hidden units in order to generate variations of musical content by the decoder (or the chain of decoders). The important difference is that

- in the case of a variational autoencoder, the exploration of values is *user-directed*, although it could be guided by some principle, e.g., geodesic in GLSR-VAE, interpolation or attribute vector arithmetics in MusicVAE (Section 6.11.1), whereas
- in the case of input manipulation, the exploration of values is *automatically guided* by the gradient descent (or ascent) mechanism, the user having previously specified a cost function to be minimized (or an objective to be maximized).

6.9.4.3 #2 Example: Deep Dream Psychedelic Images Generation System

Deep Dream, by Mordvintsev *et al.* [125], has become famous for generating psychedelic versions of standard images. The idea is to use a deep convolutional feedforward neural network architecture (see Figure 6.40) and to use it to *guide* the incremental alteration of an initial input image, in order to maximize the potential occurrence of a specific visual motif[60] correlated to the activation of a given unit.

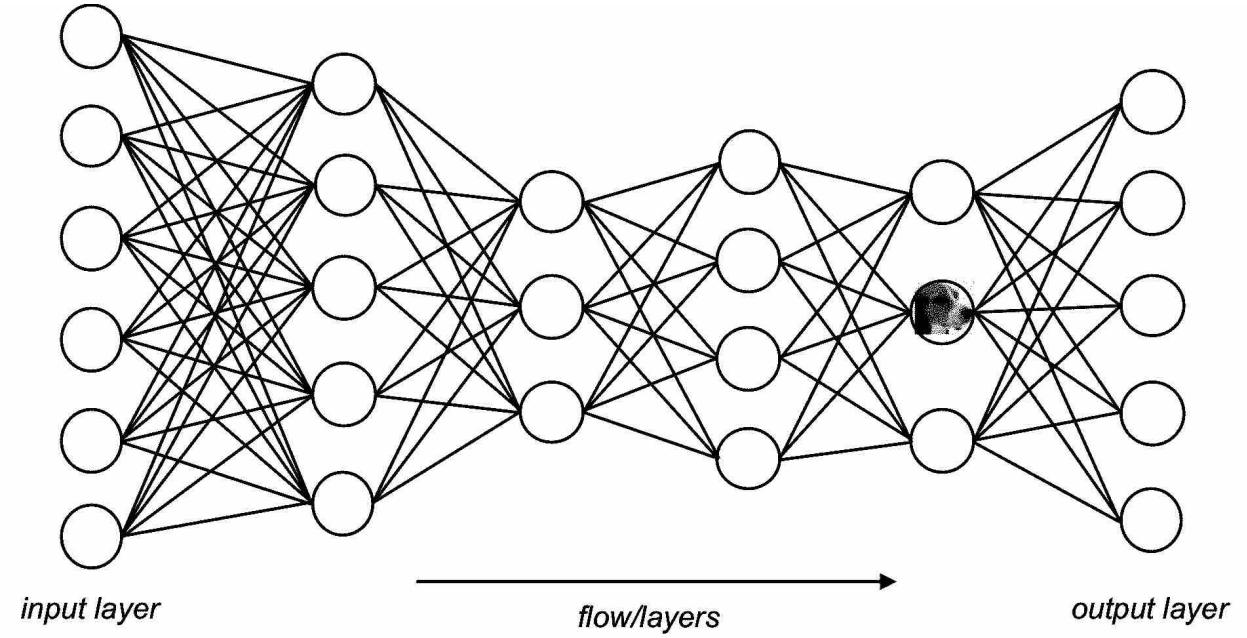

Fig. 6.40 Deep Dream architecture (conceptual)

[60] To create a pareidolia effect, where a pareidolia is a psychological phenomenon in which the mind responds to a stimulus, like an image or a sound, by perceiving a familiar pattern where none exists.

The method is as follows:

- the network is first trained on a large dataset of images;
- instead of minimizing the cost function, the objective is to *maximize* the *activation* of some specific *unit*(s) which has (have) been identified to activate for some specific visual feature(s), e.g., a dog's face, see Figure 6.40[61];
- an initial image is *iteratively* slightly altered (e.g., by jitter[62]), under *gradient ascent* control, in order to maximize the activation of the specific unit(s). This will favor the emergence of the correlated visual motif (motives), see Figure 6.41.

Note that

- the activation maximization of a *higher-layer* unit(s), as in Figure 6.40, will favor the emergence in the image of a correlated *high-level* motif (motives), like a dog's face (see Figure 6.41[63]); whereas
- the activation maximization of a *lower-layer* unit(s), as in Figure 6.42, will result in *texture insertion* (see Figure 6.43).

One may imagine a direct transposition of the Deep Dream approach to music, by maximizing the activation of a specific node[64].

6.9.4.4 #3 Example: Style Transfer Painting Generation System

The idea in this approach, named *style transfer*, pioneered by Gatys *et al.* [51] and designed for images, is to use a deep convolutional feedforward architecture to independently capture

- the features of a first image (named the *content*), and
- the style (as a correlation between features) of a second image (named the *style*).

Gradient-based learning is then used to guide the incremental modification of an initially random third image, with the double objective of matching both the content *and* the style descriptions. More precisely, the method is as follows:

- capture the *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 the *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; and
- synthesize a hybrid image.

The hybrid image is created by generating a random image, defining it as current image, and then iterating the following loop until the *two targets* (*content similarity* and *style similarity*) are reached:

- capture the *contents* and the *style* information of the 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 backpropagation, and
- *update* the current image guided by the gradients.

---

[61] Instead of exactly prescribing which feature(s) we want the network to amplify, an alternative is to let the network make that decision, by picking a layer and asking the network to enhance whatever it has detected [125].

[62] Adding a small random noise displacement of pixels.

[63] As the authors put it in [125]: "The results are intriguing – even a relatively simple neural network can be used to over-interpret an image, just like as children we enjoyed watching clouds and interpreting the random shapes. This network was trained mostly on images of animals, so naturally it tends to interpret shapes as animals."

[64] Particularly if the role of a node has been identified, through a correlation analysis between node/layer activations and musical motives, as, for example, in Section 6.16.1.

Fig. 6.41 Deep Dream. Example of a higher-layer unit maximization transformation. Created by Google's Deep Dream. Original picture: Abbey Road album cover, Beatles, Apple Records (1969). Original photography by Iain Macmillan

The architecture and process are summarized in Figure 6.44 (more details may be found in [52]). The content image (on the right) is a photography of Tübingen's Neckarfront in Germany[65] (shown in Figure 6.45) and the style image (on the left) is the painting "The Starry Night" by Vincent van Gogh (1889).

Examples of transfer for the same content (Tübingen's Neckarfront) and the styles "The Starry Night" by Vincent van Gogh (1889) and "The Shipwreck of the Minotaur" by J. M. W. Turner (1805) are shown in Figures 6.46 and 6.47, respectively.

[65] The location of the researchers.

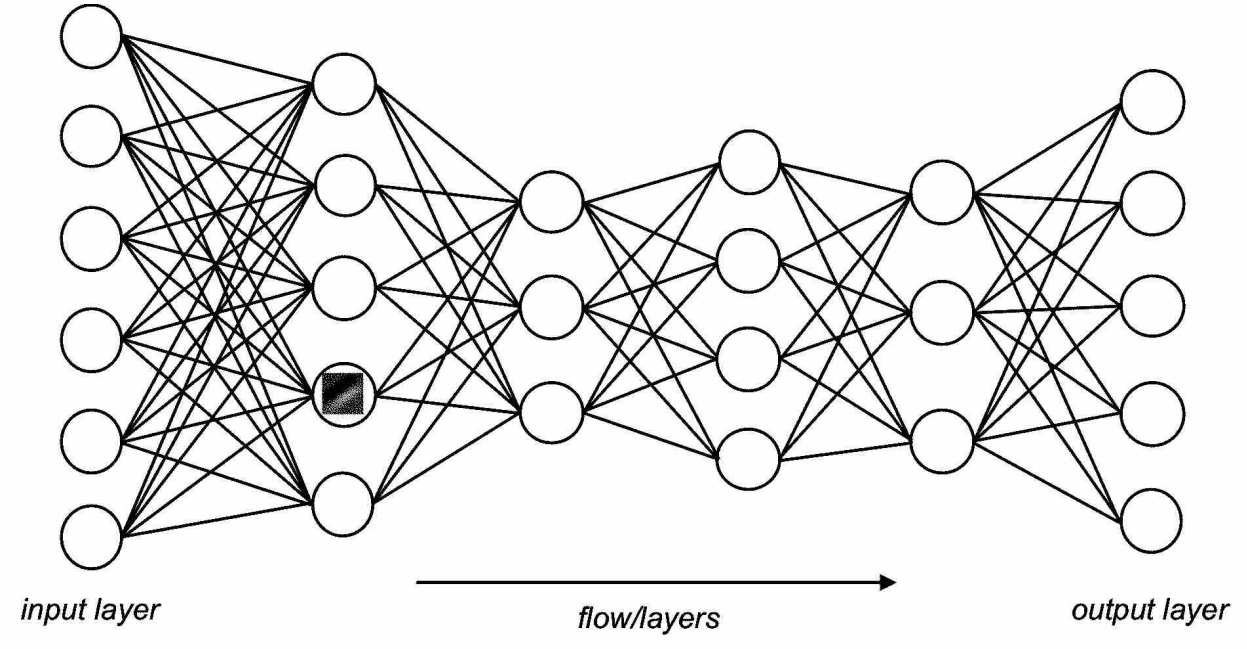

Fig. 6.42 Deep Dream architecture focusing on a lower-level unit

Note that one may balance content and style targets[66] (α/β ratio) in order to favor content or style. In addition, the complexity of the capture may also be adjusted via the number of hidden layers used. These variations are shown in Figure 6.48: rightwards an increasing α/β content/style objectives ratio and downwards an increasing number of hidden layers used (from 1 to 5) for capturing the style. The style image is the painting "Composition VII" by Wassily Kandinsky (1913).

6.9.4.5 Style Transfer vs Transfer Learning

Note that although style transfer shares some of the general objectives of *transfer learning*, it is actually different (in terms of objective and techniques). Transfer learning is about reusing what has been learnt by a neural network architecture for a specific task and applying it to *another* task and/or domain (e.g., another type of classification). Transfer learning issues will be touched upon in Section 8.3.

6.9.4.6 #4 Example: Music Style Transfer

Transposing this style transfer technique to music (*music style transfer*) is a tempting direction. However, as we will see, the style of a piece of music is more multidimensional and could be related to different types of music representation (composition, performance, sound, etc.), and is thus more difficult to capture via such a simple correlation of activations. Therefore, we will analyze this issue as a specific challenge in Section 6.10.

[66] Through the α and β parameters, see at the top of Figure 6.44 the total loss defined as $\mathscr{L}_{total} = \alpha\mathscr{L}_{content} + \beta\mathscr{L}_{style}$.

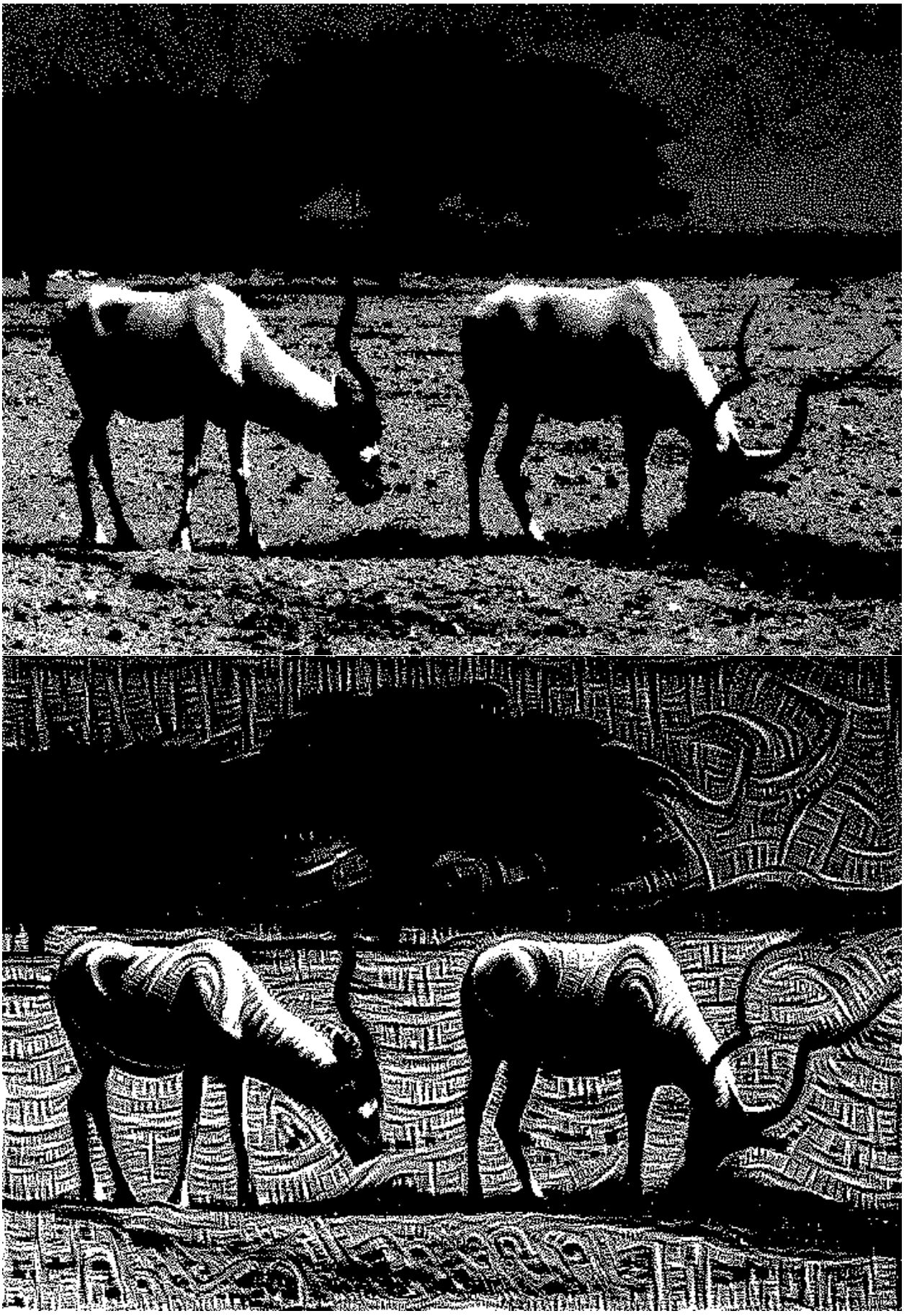

Fig. 6.43 Deep Dream. Example of a lower-layer unit maximization transformation. Reproduced from [125] under a CC BY 4.0 licence. Original photography by Zachi Evenor

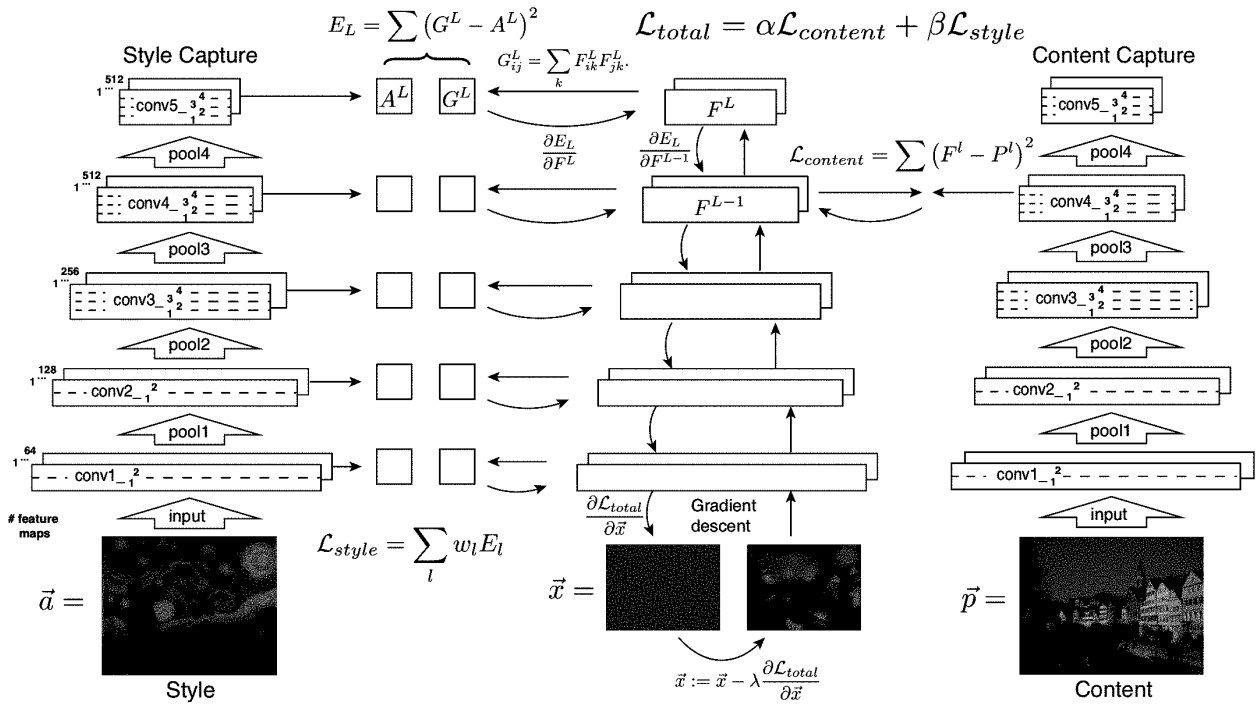

Fig. 6.44 Style transfer full architecture/process. Extension of a figure reproduced from [51] with the permission of the authors

6.9.5 Input Manipulation and Sampling

An example of the combination of the input manipulation strategy with the sampling strategy, thus acting both on the input and the output[67], is exemplified in the following section.

6.9.5.1 Example: C-RBM Polyphony Symbolic Music Generation System

In the system presented by Lattner *et al.* in [100], the starting point is to use a restricted Boltzmann machine (RBM) to learn the local structure, seen as the *musical texture*, of a corpus of musical pieces. The additional idea is to impose, through constraints, a more *global structure* (form, e.g., AABA, as well as tonality), seen as a *structural template* inspired by an existing musical piece, on the new piece to be generated. This is called *structure imposition*[68], also earlier coined as *templagiarism* (short for template plagiarism) by Hofstadter [78]. These constraints, concerning structure, tonality and meter, will guide an iterative generation through a search process, manipulating the input, based on gradient descent.

The actual objective is the generation of polyphonic music. The representation used is piano roll, with 512 time steps and a range of 64 notes (corresponding to MIDI note numbers 28 to 92). The corpus is Wolfgang Amadeus Mozart's sonatas. Each piece is transposed into all possible keys in order to have sufficient training data for all possible keys (this also helps reduce sparsity in the training data). The architecture is a convolutional restricted Boltzmann machine (C-RBM) [106], i.e. an RBM with convolution, with $512 \times 64 = 32{,}768$ input nodes and 2,048 hidden units. Units have continuous and not Boolean values, as for standard RBMs (see Section 5.10.2). Convolution is only performed on the

[67] Interestingly, the input is actually equal to the output because the architecture used is an RBM (see Section 5.10), where the visible layer acts both as input *and* output.

[68] This is an example of score-level *composition style transfer* (see Section 6.9.4.6).

Fig. 6.45 Tübingen's Neckarfront. Photography by Andreas Praefcke. Reproduced from [51] with the permission of the authors

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[69].

Training of the C-RBM is undertaken using the RBM-specific contrastive divergence algorithm (see Section 5.10, more precisely a more advanced version named persistent contrastive divergence). Generation is performed by sampling 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 (measured through a mean squared error) between the self-similarity matrixes of the reference 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, the distribution of pitch classes in the window is compared with the so-called key profiles of the reference (i.e. paradigmatic relative pitch-class strengths for specific scales and modes [168], in practice the major and minor modes). They are repeated in the time and pitch dimensions of the piano roll matrix, with a modulo octave shift in the pitch dimension. The resulting key estimation vectors are combined (see the article for more details) to obtain

[69] As the authors state in [100]: "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 estimations, constituting a hierarchical tonal structure (on a very local level, key estimation is strongly related to chord estimation)."

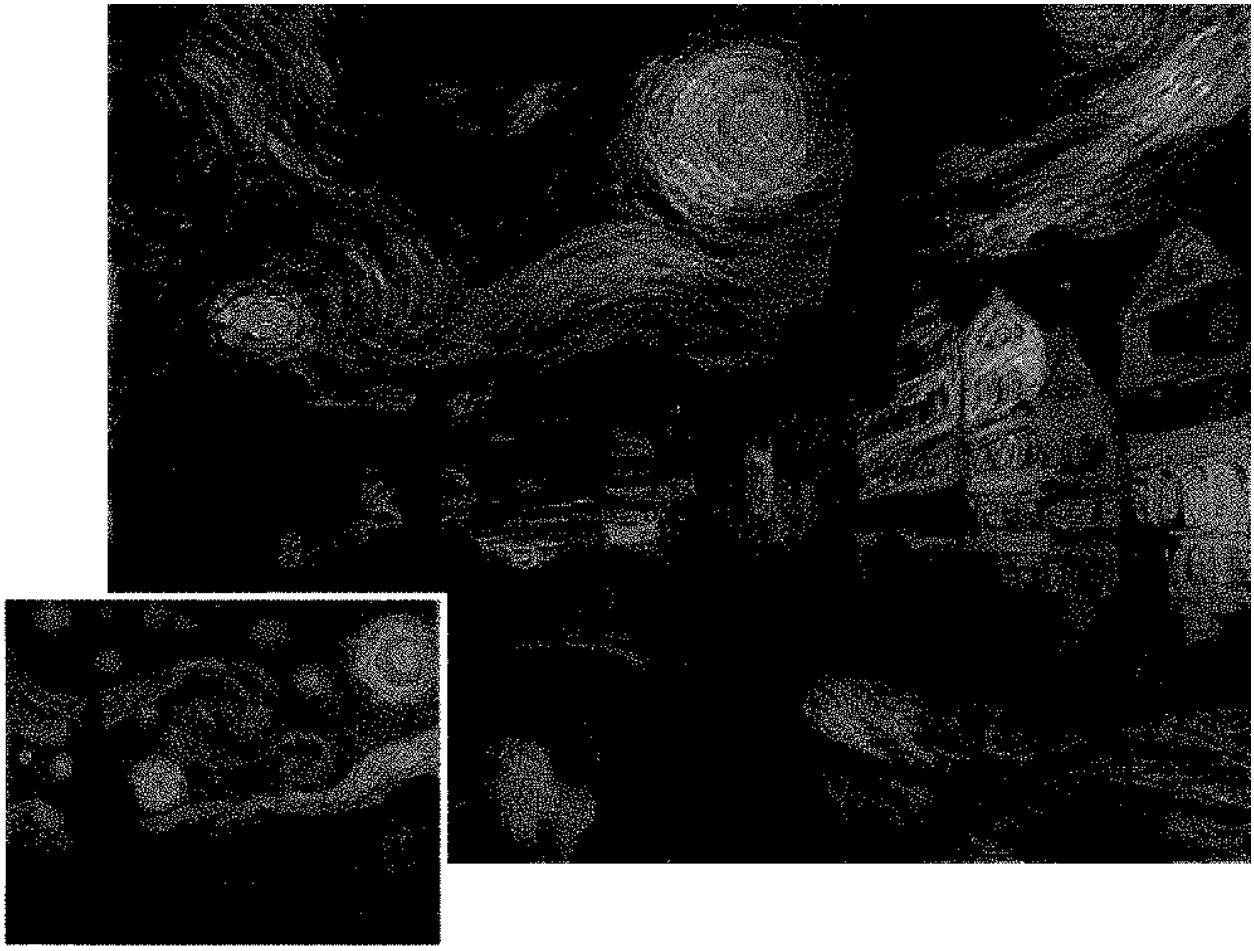

Fig. 6.46 Style transfer of "The Starry Night" by Vincent van Gogh (1889) on Tübingen's Neckarfront photography. Reproduced from [51] with the permission of the authors

an overall key estimation vector. In the same way as for self-similarity, the distance between the target and the intermediate solution key estimations is minimized.

- *Meter constraint* – the purpose is to impose a specific *meter* (also named a *time signature*, e.g., 3/4, 4/4, see Section 4.5.4) 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 the reference.

Generation is performed via *constrained sampling* (CS), a mechanism used to restrict the set of possible solutions in the sampling process according to some pre-defined constraints. The principles of the process, illustrated in Figure 6.49, are as follows:

- A sample is randomly initialized following the standard uniform distribution.
- A step of constrained sampling (CS) is performed comprising
  - n runs of gradient descent (GD) optimization to impose the high-level structure, and
  - p runs of selective Gibbs sampling (SGS)[70] to selectively realign the sample onto the learnt distribution.

[70] Selective Gibbs sampling (SGS) is the authors' variant of Gibbs sampling (GS).

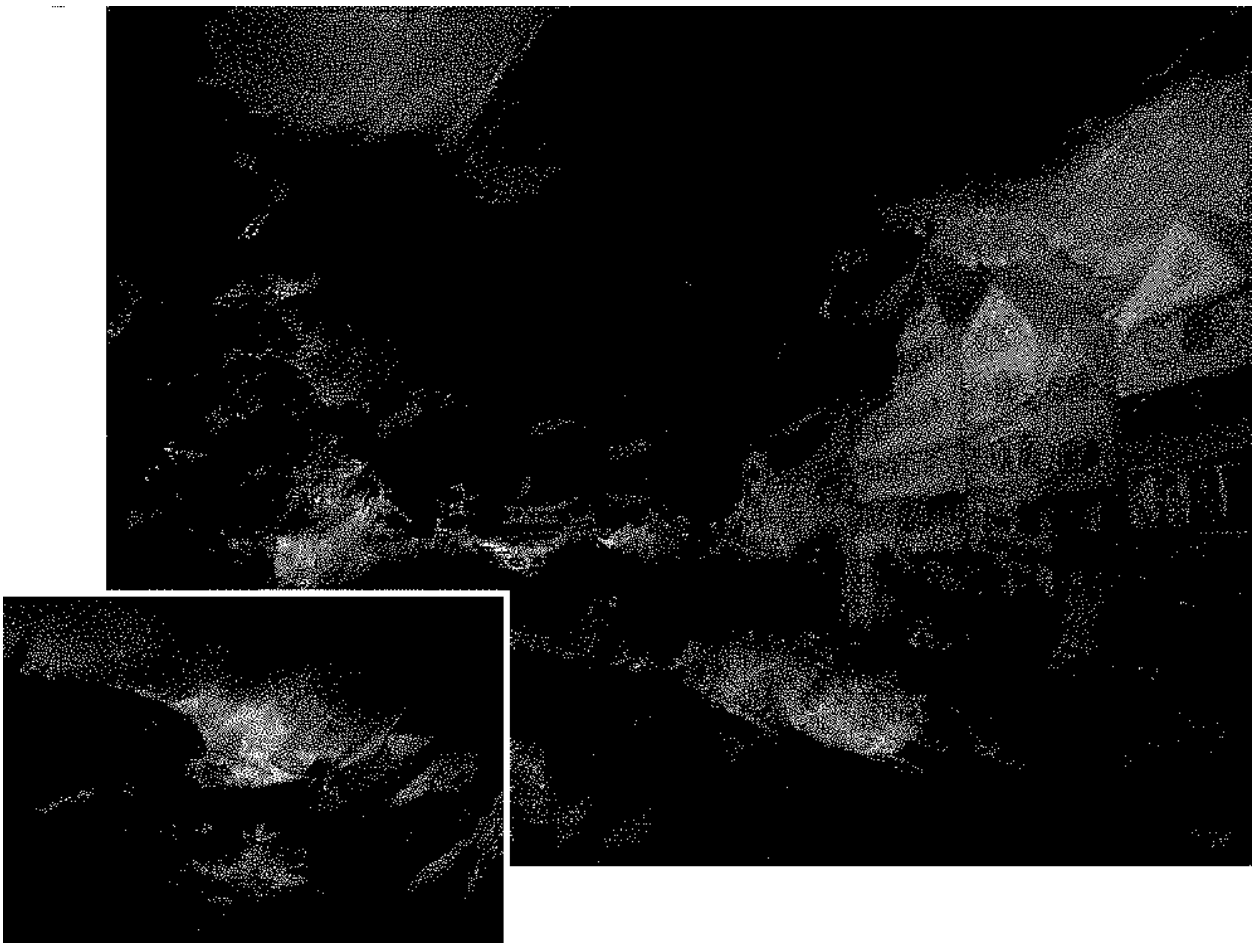

Fig. 6.47 Style transfer of "The Shipwreck of the Minotaur" by J. M. W. Turner (1805) on Tübingen's Neckarfront photography. Reproduced from [51] with the permission of the authors

- A simulated annealing algorithm is applied in order to decrease exploration in relation to a decrease of variance over solutions.

The different steps of constrained sampling are further detailed in [100]. Figure 6.50 shows an example of a generated sample in piano roll format.

The results for the C-RBM (summarized in Table 6.24) are interesting and promising. One current limitation, stated by the authors, is that constraints only apply to the high-level structure. Initial attempts at imposing low-level structure constraints are challenging because, as constraints are never purely content-invariant, when trying to transfer low-level structure, the template piece can be exactly reconstructed in the GD phase. Therefore, creating constraints for low-level structure would have to be accompanied by an increase in their content invariance. Another issue is convergence and satisfaction of the constraints. As discussed by the authors, their approach is not exact, as opposed to the Markov constraints approach (for Markov chains) proposed in [137].

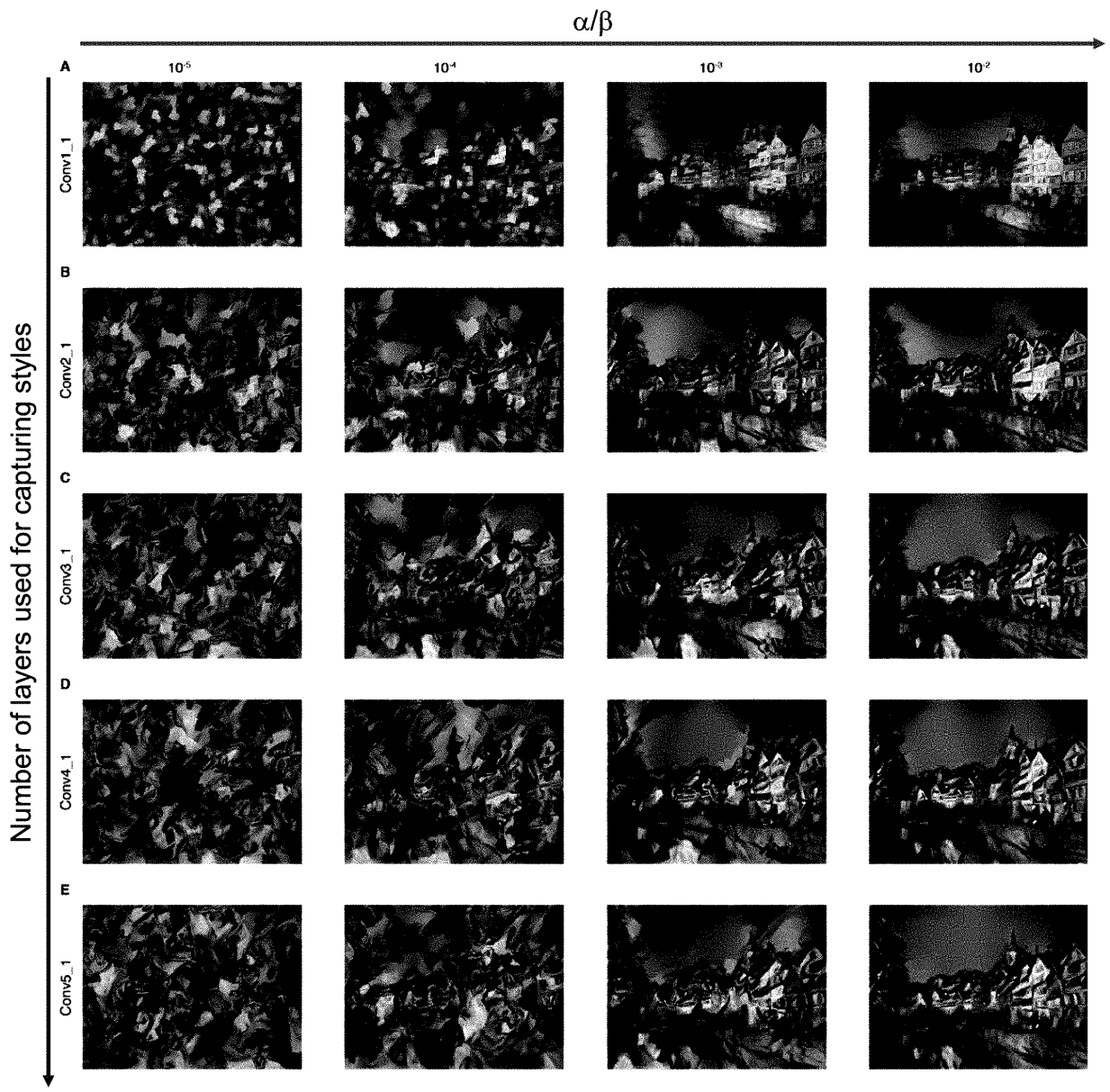

Fig. 6.48 Variations on the style transfer of "Composition VII" by Wassily Kandinsky (1913) on Tübingen's Neckarfront photography. Reproduced from [51] with the permission of the authors

Objective	Polyphony; Style imposition
Representation	Symbolic; Piano-roll; Rest; Many-hot; Meter
Architecture	Convolutional(RBM)
Strategy	Input manipulation; Sampling

Table 6.24 C-RBM summary

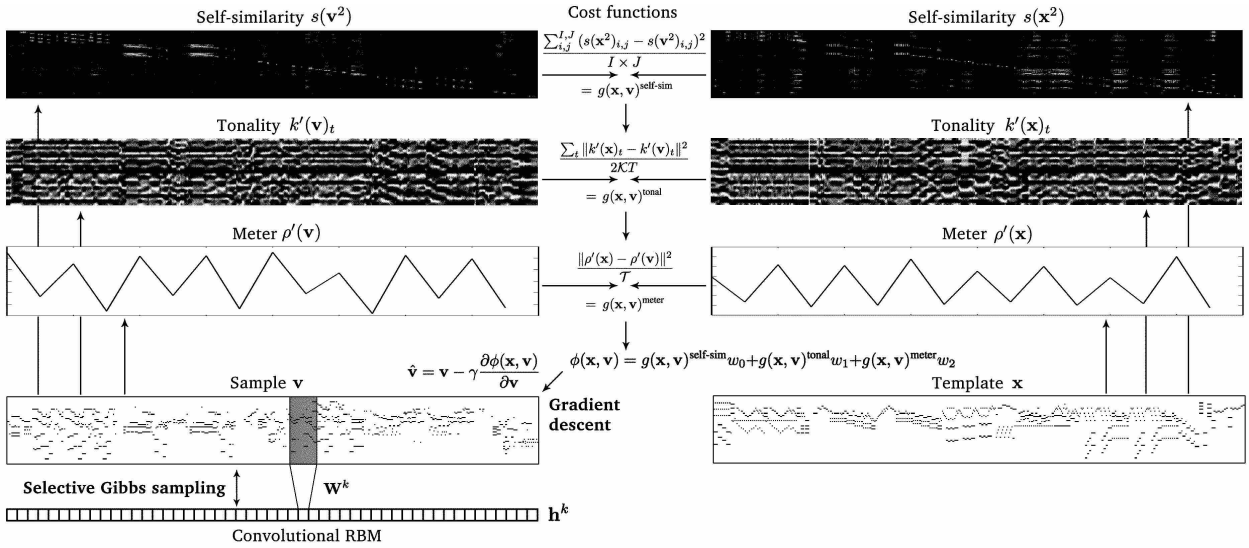

Fig. 6.49 C-RBM architecture. Reproduced from [100] with the permission of the authors

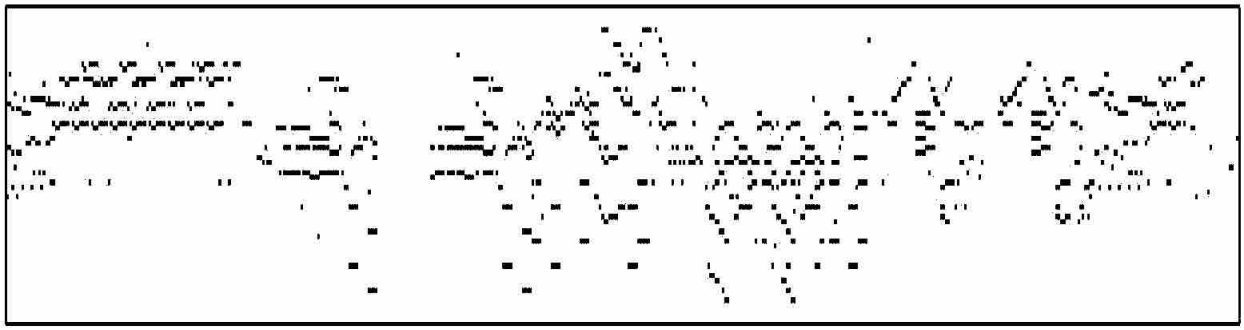

Fig. 6.50 Piano roll sample generated by C-RBM. Reproduced with the permission of the authors

6.9.6 Reinforcement

The idea of the *reinforcement strategy* is to reformulate the generation of musical content as a *reinforcement learning problem*: using the similarity to the output of a recurrent network trained on the dataset as a reward and adding user defined constraints, e.g., some tonality rules according to music theory, as an additional reward.

Let us consider the case of a monophonic 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 the next note to be generated.

Let us now consider a recurrent neural 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. the next note to be generated, to the note predicted by the recurrent network in a similar state (i.e. the partial melody generated so far); and
- adherence to *user-defined constraints* (e.g., consistency with current tonality, avoidance of excessive repetitions, etc.), by measuring how well they are fulfilled.

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

6.9.6.1 Example: RL-Tuner Melody Symbolic Music Generation System

The reinforcement strategy was pioneered in the RL-Tuner architecture by Jaques *et al.* [86]. This architecture, illustrated in Figure 6.51, consists in two deep Q network reinforcement learning architectures[71] and two recurrent neural network (RNN) architectures.

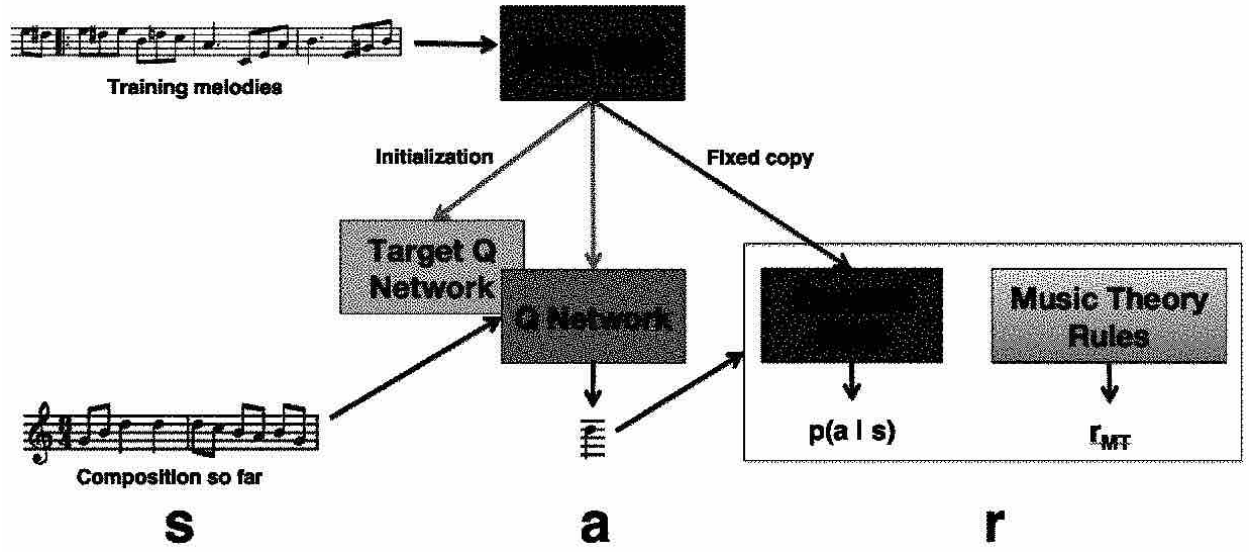

Fig. 6.51 RL-Tuner architecture. Reproduced from [86] with the permission of the authors

- The initial RNN, named Note RNN, is trained on the dataset of melodies for the task of predicting and generating the next note, following the iterative feedforward strategy.
- A fixed copy of Note RNN is made, named Reward RNN, which will be used by the reinforcement learning architecture as a reference.
- The Q Network architecture task is to learn to select the next note (next action a) from the generated (partial) melody so far (current state s).
- The Q Network is trained in parallel to the other Q Network, named Target Q Network, which estimates the value of the gain (accumulated rewards) and which has been initialized from what Note RNN has learnt.
- Q Network's reward r combines two rewards, as defined in previous section:
  - adherence to *what has been learnt*, measured by the probability of Reward RNN to play that note, in practice $\log p(a|s)$, the log probability for the next note being a given a melody s; and
  - adherence to *music theory constraints*, in practice[72]:
    - staying in key,
    - beginning and ending with the tonic note,
    - avoiding excessively repeated notes,
    - preferring harmonious intervals,

[71] An implementation of the Q-learning reinforcement learning strategy through a deep learning architecture [178].

[72] This list of musical theory constraints has been selected from [53], see more details in [86].

- resolving large leaps,
- avoiding continuously repeating extrema notes,
- avoiding high auto-correlation,
- playing motifs, and
- playing repeated motifs.

The total reward $r(s,a)$[73] is defined by Equation 6.5, where r_{MT} is the reward concerning music theory and c is a parameter controlling the balance between the two competing constraints.

$$r(s,a) = \log p(a|s) + r_{MT}(a,s)/c \tag{6.5}$$

Figure 6.52 shows the evolution during the training phase of the two types of rewards (adherence to Note RNN and to music theory), with three different reinforcement learning algorithms: Q-learning, Ψ-learning and G-learning (see details in [86]).

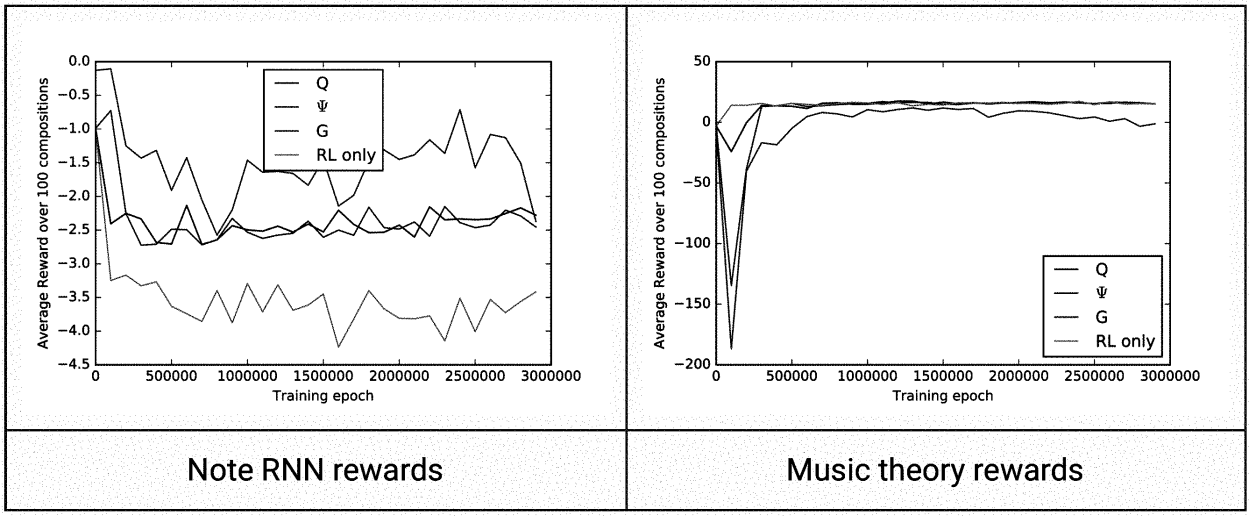

Fig. 6.52 Evolution during training of the two types of rewards for the RL-Tuner architecture. Reproduced from [86] with the permission of the authors

The corpus used for the experiments is a set of monophonic melodies extracted from a corpus of 30,000 MIDI songs. The time step is set at a sixteenth note. The one-hot encoding (of dimension 38) considers three octaves of notes plus two special events: note off (encoded as 0) and no note (a rest, encoded as 1). The MIDI note number is translated in order to start the lowest note (C_3) as a 2 (B_5 is encoded as 37) and have special events smoothly integrated within the integer encoding. Note that, as melodies are monophonic, playing a different note implicitly ends the last played note without requiring an explicit note off event, which results in a more compact representation. Note RNN (and its copy Reward RNN) have one LSTM layer with 100 cells.

In summary, the reinforcement strategy allows arbitrary user given constraints (control) to be combined with a style learnt by the recurrent network.

Note that in the case of RL-Tuner, the reward is known beforehand and dual purpose: *handcrafted* for the music theory rules and *learnt* from the dataset by an RNN for the musical style. Therefore, there is an opportunity to add another type of reward, an *interactive* feedback by the *user* (see Section 6.14). However, a feedback at the granularity of each note generated may be too demanding and, moreover, not that accurate[74]. We will discuss in Section 6.15 the issue of learning from user feedback. RL-Tuner is summarized in Table 6.25.

[73] Which means the reward to be received when from state s action a is chosen.

[74] 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."

Objective	Melody
Representation	Symbolic; One-hot; Note-off; Rest
Architecture	LSTM×2 + RL
Strategy	Iterative feedforward; Reinforcement

Table 6.25 RL-Tuner summary

6.9.7 Unit Selection

The *unit selection strategy* is about querying successive musical units (e.g., one measure long melodies) from a database and concatenating them in order to generate a 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 a balance between *direction* and *transition*.

This strategy, as opposed to most of the other ones, which are bottom-up, is *top-down*, as it starts with a structure and fills it.

6.9.7.1 Example: Unit Selection and Concatenation Symbolic Melody Generation System

This strategy was pioneered by Bretan *et al.* [12]. The idea is to generate music from a concatenation of musical units, queried from a database. The key process here is unit selection, which is based on two criteria: *semantic relevance* and *concatenation cost*. The idea of unit selection to generate sequences was 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…) and 120 Jazz solo transcriptions. The granularity of a musical unit is a measure. This means there are roughly 170,000 units in the dataset. The dataset is restricted to a five octaves range (MIDI note numbers 36 to 99) and augmented by transposing each unit in all keys so that all possible pitches are covered.

The architecture includes one autoencoder and two LSTM recurrent networks. The first step is feature extraction: 10 features, manually handcrafted, are considered, following a *bag-of-words* (BOW) approach (see Section 4.9.3), e.g., counts of a certain pitch class, counts of a certain pitch class rhythm tuple, whether the first note is tied to the previous measure, etc. This results in 9,675 actual features. Most of features have integer values, with the exception of rests being represented using a negative pitch value and of some Boolean features. Therefore, each unit is described (indexed) as a feature vector of size 9,675.

The autoencoder used has a 2-layer stacked autoencoder architecture, as illustrated in Figure 6.53. Once trained on the set of feature vectors, in the usual self-supervised way for autoencoders (see Section 5.9), the autoencoder becomes a features extractor encoding a feature vector of size 9,675 into an *embedding* vector of size 500.

There is one remaining issue for generating a melody: how to select the best (or at least, a very good) candidate from a given (current, named seed by the authors) musical unit as a successor musical unit? Two criteria are considered:

- *Successor semantic relevance* – based on a model of transition between units, as learnt by an LSTM recurrent network. In other words, relevance is based on the distance to the (ideal) next unit as predicted by the model. This first LSTM architecture has two hidden layers, each with 128 units. The input and output layers have 512 units (corresponding to the format of the embedding).
- *Concatenation cost* – based on another model of transition[75] between the last note of current unit and the first note of the next unit, as learnt by another LSTM recurrent network. This second LSTM architecture is multilayer and its input and output layers have about 3,000 units, corresponding to a multi-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 in Figure 6.54, with current (seed) unit in blue and next (candidate) unit in red) is handled by a heuristic-based dynamic ranking process:

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

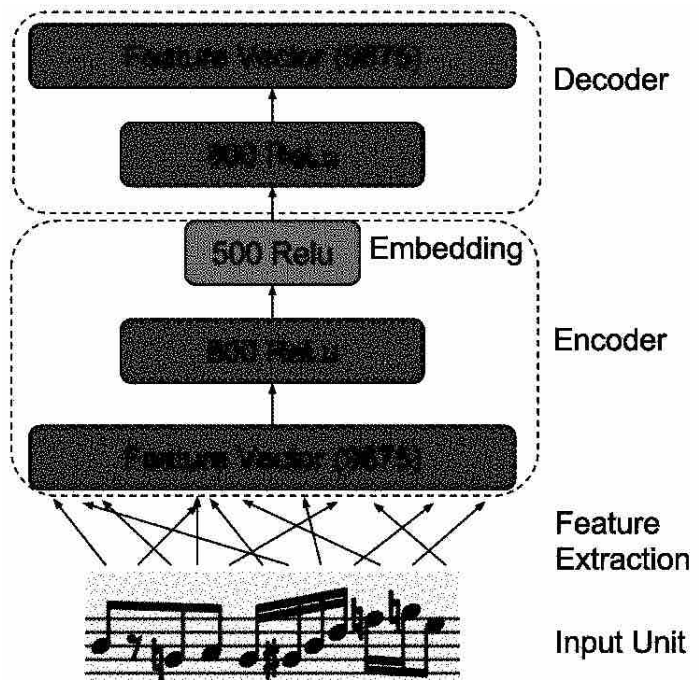

Fig. 6.53 Unit selection indexing architecture. Reproduced from [12] with the permission of the authors

1. rank all musical units according to their successor semantic relevance with current musical unit[76];
2. take the top 5% and re-rank them according to the combination of their successor semantic relevance and their concatenation cost; and
3. select the musical unit with the highest combined rank.

The process is iterated in order to generate successive musical units and thus a melody of arbitrary length. This may at first look like an iterative feedforward generation from a recurrent network (see Section 6.4.1.2), but there are two important differences:

- the label (instance of the embedding) of the next musical unit is computed through a multicriteria ranking algorithm; and
- the actual unit is queried from a database with the label as the index.

Initial external human evaluation has been conducted by the authors. They found that music generated using one or two measures long units tend to be ranked higher according to naturalness and likeability than four measures long units or note-level generation, with an ideal unit length appearing to be one measure.

Note that the unit selection strategy does not directly provide control, but it does provide *entry points* for control as one may extend the selection framework (currently based on two criteria: successor semantic relevance and concatenation cost) with user defined constraints/criteria. The system is summarized in Table 6.26.

Objective	Melody
Representation	Symbolic; Rest; BOW Features
Architecture	Autoencoder^2 + LSTM$\times$2
Strategy	Unit selection; Iterative feedforward

Table 6.26 Unit selection summary

[76] The initial musical unit of a melody to be generated may be chosen by the user or sorted.

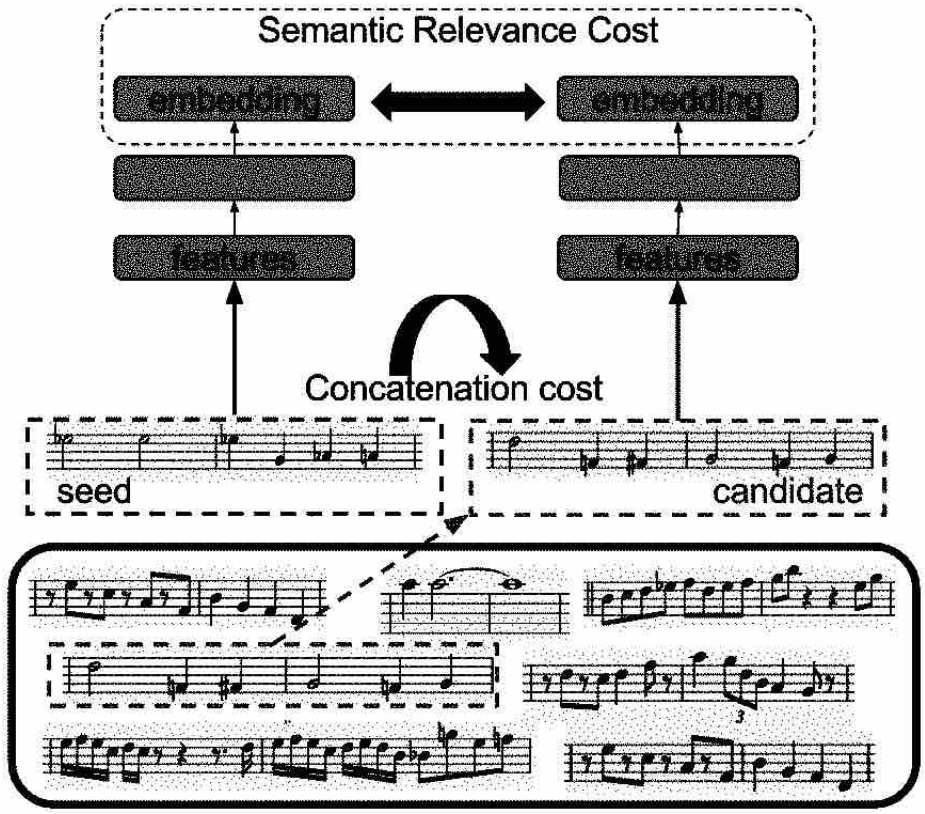

Fig. 6.54 Unit selection based on semantic cost. Reproduced from [12] with the permission of the authors

6.10 Style Transfer

In Section 6.9.4.6 we introduced style transfer as one example of using the input manipulation strategy to control content generation. The style transfer technique for images (proposed by Gatys *et al.* [51] and described in Section 6.9.4.4) is effective and relatively straightforward to apply. However, as opposed to paintings, where the common representation is two-dimensional and uniformly digitalized in terms of pixels, music is a much more complex object with various levels and models of representation (see Chapter 4 and also Section 6.10.2.2).

In their recent analysis, Dai *et al.* [28] consider three main levels (or dimensions) of representation and associated types of music style transfer:

- *score*-level, which they name *composition style transfer*;
- *sound*-level, which they name *timbre style transfer*; and
- *performance control*-level, which they name *performance style transfer*.

They state that music style transfer for each level (namely, composition, timbre and performance style transfer) are very different in nature. They also point out the issue of the interrelation and the *entanglement* of these different levels (and nature) of representation. Therefore, they point out the need for automated learning of the disentanglement[77] of different levels of music representation, in order to ease music style transfer.

[77] Disentanglement is the objective of separating the different factors governing variability in the data (e.g., in the case of human images, identity of the individual and facial expression, see, for example, [32]). Recent work on *disentanglement learning* can be found, for example, in [9]. Also note that variational autoencoders (VAEs, see Section 5.9.2) are currently among the promising approaches for disentanglement learning because, as Goodfellow *et al.* put it in [58, Section 20.10.3]: "Training a parametric encoder in combination with the generator network forces the model to learn a predictable coordinate system that the encoder can capture."

6.10.1 Composition Style Transfer

Style transfer at the composition level means working on symbolic representations. An example is *structure imposition*, i.e. transferring some existing structure (e.g., an AABA global structure) from an initial composition into another newly generated composition. The C-RBM system, presented in Section 6.9.5.1, implements this kind of structure imposition by considering separately three kinds of structures and associated constraints: global structure (e.g., AABA), tonality and meter (rhythm).

One may think that such structure descriptors are too low level to define a style. But they are an interesting first step, as one may consider higher-level style descriptors by aggregating such structure descriptors. Let us imagine, for instance, describing (and later on transferring) the style of a composer like Michel Legrand with his own way of repeating transpositions of motives.

Note that in the DeepJ system for controlling the style of the generation (Section 6.9.3.4) the objective is different, as the style is explicitly specified by the user via a set of musical examples, learnt and applied during generation time through conditioning.

6.10.2 Timbre Style Transfer

For timbre style transfer, based on audio representations, some researchers have straightforwardly applied Gatys *et al.*'s technique (Section 6.9.4.4) to sound (audio), using various kinds of sources (various styles of music as well as speech), as explained in next section.

6.10.2.1 Examples: Audio Timbre Style Transfer Systems

Examples of style transfer systems for audio (timbre) are:

- Ulyanov and Lebedev's system in [174], and
- Foote *et al.*'s system in [46].

These two systems both use a spectrogram (and not a direct wave signal) as their input representation[78]. In [187], Wyse points out two specificities (which he calls "two remarkable aspects") in the architecture of Ulyanov and Lebedev's system that differentiate it from the image style transfer technique:

- the network uses only a single layer. Therefore, the only difference between content and style comes from the difference between first-order and second-order (correlations) measures of activation; and
- the network was not pre-trained and uses random weights.

Wyse further adds in [187]: "The blog post claims this unintuitive approach generated results as good as any other, and the sound examples posted are indeed compelling." We also found convincing the examples of audio style transfer, although not as interesting as painting style transfer, as it sounds similar to some sound modulation/merging of both style and content signals. In their own analysis in [46], Foote *et al.* summarize the difficulty as follows: "On this level we draw one main conclusion: audio is dissimilar enough from images that we shouldn't expect work in this domain to be as simple as changing 2D convolutions to 1D." We will try to analyze some possible reasons for this in the next section.

Table 6.27 summarizes the main features of these audio (timbre) style transfer systems (which we will reference to as AST in Chapter 7).

[78] For a comparison of various audio representations for audio style transfer, see the recent analysis by Wyse [187].

Objective	Audio style transfer (AST)
Representation	Audio; Spectrum
Architecture	Convolutional(Feedforward)
Strategy	Input manipulation; Feedforward

Table 6.27 Audio (timbre) style transfer (AST) summary

6.10.2.2 Limits and Challenges

We believe that, in part, the difficulty of directly transposing image style transfer to music comes from the *anisotropy*[79] of global audio music content representation. In the case of a natural image, the correlations between visual elements (pixels) are equivalent whatever the direction (horizontal axis, vertical axis, diagonal axis or any arbitrary direction), i.e. correlations are *isotropic*. In the case of a global representation of audio data (where the horizontal dimension represents time and the vertical dimension represents the notes), this uniformity no longer holds as horizontal correlations represent *temporal* correlations and vertical correlations represent *harmonic* correlations, which are very different in nature (see an illustration in Figure 6.55).

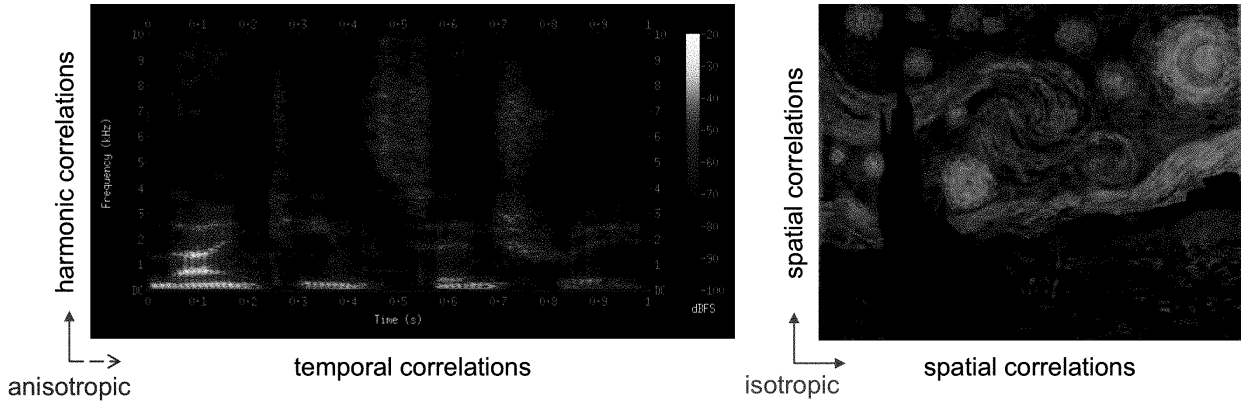

Fig. 6.55 Anisotropic music vs an isotropic image. Incorporating Aquegg's original image from "https://en.wikipedia.org/wiki/Spectrogram" and the painting "The Starry Night" by Vincent van Gogh (1889)

One direction could be to reformulate the capture of the style information, and therefore the nature of the correlations, in order to take into account the time dimension. Another (also hypothetical) direction could be to use a "time-compressed" representation, by considering the summary learnt by an RNN Encoder-Decoder (see Section 6.9.2.3).

6.10.3 Performance Style Transfer

Although it does not directly address performance style transfer, the Performance RNN system described in Section 6.6.1 provides a background representation for modeling performance (note onsets as well as dynamics). What remains to be undertaken to develop a *performance imposition* system could be along the following lines:

- model mappings between performance and other(s) features (e.g., mean duration of notes, modulation, etc.);
- learn mappings for a given corpus (musician, context, etc.) through correlation analysis, revealing the performance style of a given musician (and corpus); and
- transpose a mapping to an existing piece in order to transfer the performance style.

[79] Isotropy means invariance of properties regardless of the direction, whereas anisotropy means direction dependence.

As noted by Dai *et al.* in [28], performance style transfer is closely related to expressive performance rendering, see, for instance, the example of the CyberJoão system [27] introduced in Section 6.6 and a recent system based on deep learning in [114]. But it also requires the disentanglement of control (style) and score information (content) as well as the learning of the mappings discussed above. Thus, this is still a direction to be explored.

6.10.4 Example: FlowComposer Composition Support Environment

A good example of an interactive music composition environment addressing style transfer in different dimensions is the FlowComposer system [138, 141], developed by Pachet *et al.* during the Flow Machines project [42]. Note that it is based on Markov chain models and not (yet) deep learning models.

FlowComposer provides possibilities for music style transfer at the following levels:

- Composition style level – some style transfer may be performed, e.g., automated reharmonization based on a style (corpus) of selected music. See the examples of the automatic reharmonization of Yesterday by John Lennon and Paul McCartney (Figure 6.56) in the style of Michel Legrand (Figure 6.57) and Bill Evans (Figure 6.58).
- Timbre and performance style levels[80] – rendering may be done via style transfer, by automated mapping and extrapolation from a library of various instrumental audio performances (through the ReChord component [143]).

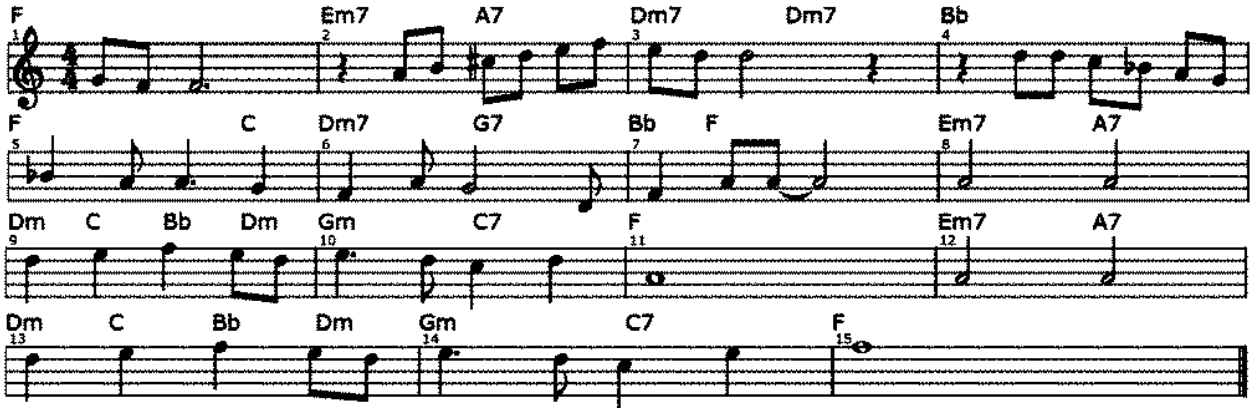

Fig. 6.56 Yesterday (Lennon/McCartney) (first 15 measures) – original harmonization. Reproduced from [141] with the permission of the authors

The FlowComposer control panel includes various fields to select composition style and sliders to set harmonisation conformance, inspiration, average note duration and chord changes, as shown in Figure 6.59. An example of a lead sheet generated in the style of Bill Evans is shown in Figure 6.60, with the following user defined characteristics: a 3/4 time signature, a constraint on the first (C7) and last (G7) chords, and a "max order" of four beats[81]. Color backgrounds indicate sequences of notes extracted as a whole from a given song in the chosen corpus (here all of Bill Evans' compositions in 3/4).

[80] Jointly, as there is no possibility yet for separating/disentangling these two concerns.

[81] This very interesting feature controls the maximum amount of successive notes (actually beats) copied from the corpus. It relies on the integration of a new constraint named MaxOrder [140] in the Markov constraints framework [137] underlying FlowComposer. This is one possible way to control originality (see Section 6.12).

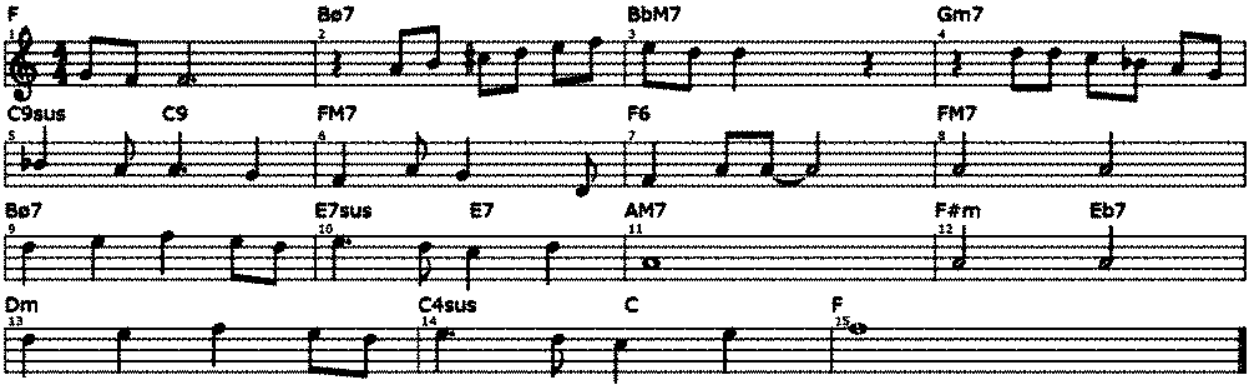

Fig. 6.57 Yesterday (Lennon/McCartney) (first 15 measures) – reharmonization by FlowComposer in the style of Michel Legrand. Reproduced from [141] with the permission of the authors

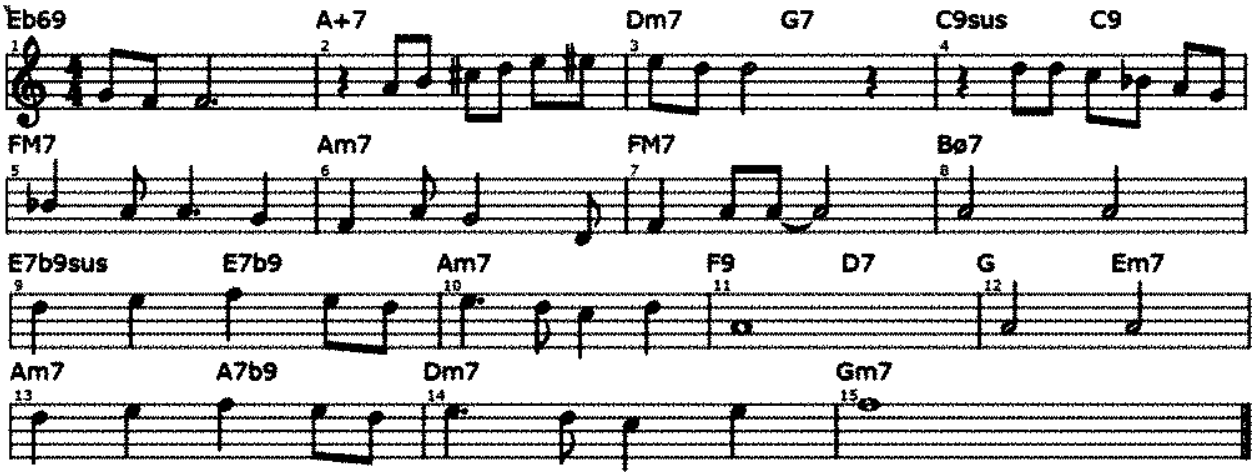

Fig. 6.58 Yesterday (Lennon/McCartney) (first 15 measures) – reharmonization by FlowComposer in the style of Bill Evans. Reproduced from [141] with the permission of the authors

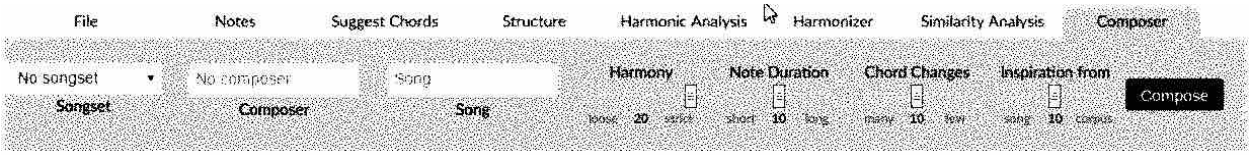

Fig. 6.59 Flow Composer control panel. Reproduced from [141] with the permission of the authors

6.11 Structure

One challenge is that most existing systems have a tendency to generate music with no clear structure or "sense of direction"[82]. In other words, although the style of the generated music corresponds to the corpus learnt, the music appears to wander without any higher organization, as opposed to human composed music which has some global organization (usually named a *form*) and identified components, such as

- an overture, an allegro, an adagio or a finale in classical music;
- an AABA or an AAB form in Jazz;

[82] Beside the technical improvements brought by LSTMs on the learning of long-term dependencies (see Section 5.11.3).

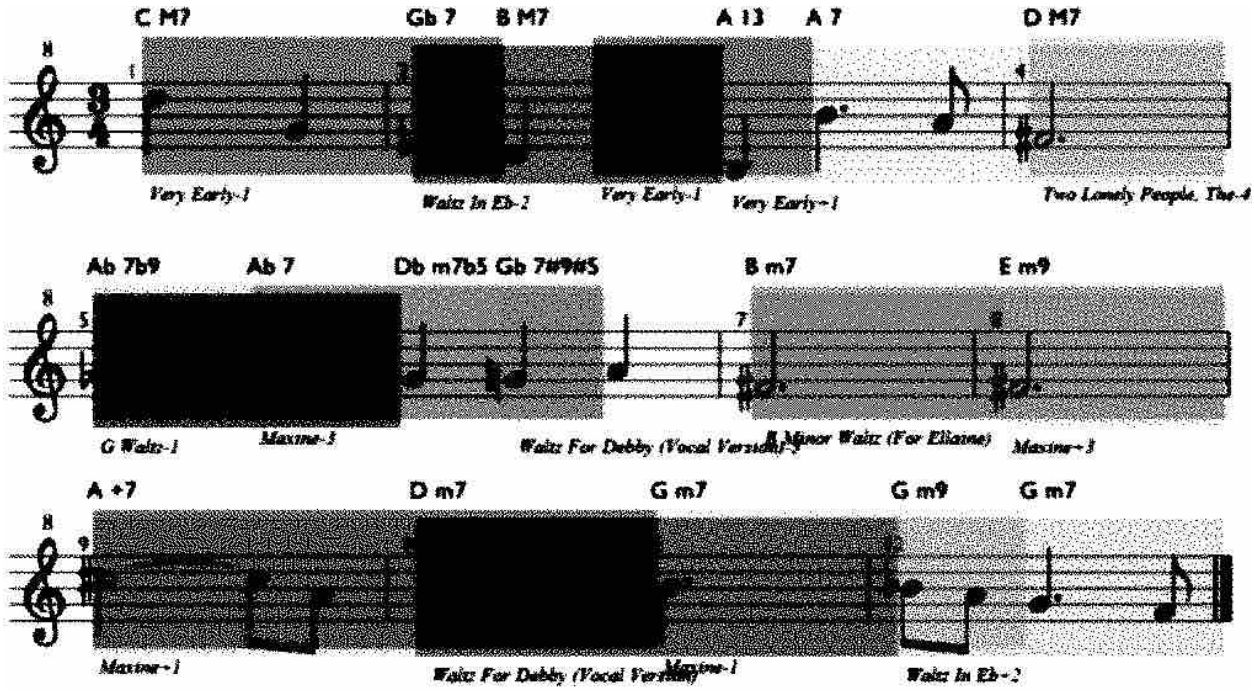

Fig. 6.60 Example of a Flow Composer interactively generated lead sheet. Reproduced from [138] with the permission of the authors

- a refrain, a verse or a bridge in song music.

Note that there are various possible levels of structure. For instance, an example of a finer-grain structure is at the level of melodic patterns that can be repeated, often being transposed in order to adapt to a new harmonic structure.

The reinforcement strategy (used by RL-Tuner in Section 6.9.6.1) and the structure imposition approach (used by C-RBM in Section 6.9.5.1) can both enforce (and/or transfer, see Section 6.10) some constraints, possibly high-level, on the generation. About structure imposition, see also a recent proposal combining two graphical models, one for chords and one for melody, for the generation of lead sheets with an imposed structure [136]. An alternative top-down approach is followed by the unit selection strategy (see Section 6.9.7), by generating an abstract sequence structure and filling it with musical units, although the structure is not yet very high-level as it effectively stays at the level of a measure.

A related challenge is not about the *imposition* of *preexisting* high-level structures, but about the capacity for *learning* high-level structures and, moreover, the capacity for *invention* (emergence) of high-level structures. Therefore, a natural direction is to explicitly consider and process different levels (hierarchies) of temporality and structure.

6.11.1 Example: MusicVAE Multivoice Hierarchical Symbolic Music Generation System

In [148], Roberts *et al.* propose an architecture named MusicVAE, based on a variational recurrent autoencoder (VRAE) with a 2-level hierarchical RNN within the decoder

The corpus comprises MIDI files collected from the web, from which three types of musical examples are extracted:

- monophonic melodies, 2 or 16 measures long;
- drum patterns, 2 or 16 measures long; and
- trio sequences with three different voices (melody, bass line and drum pattern), 16 measures long.

Encoding of monophonic melodies and bass lines is through tokens representing MIDI events: the 128 "Note on" events corresponding to the 128 possible MIDI note numbers (pitches) of the defined interval, the single[83] "Note off" event and the rest (silence) token. Encoding of drum patterns is done by mapping MIDI standard drum classes through a binning into 9 canonical classes one-hot encoded into $2^9 = 512$ categorical tokens representing all possible combinations. Quantization is at the sixteenth note.

The architecture follows the principles of a variational autoencoder encapsulating recurrent networks such as VRAE, with two differences:

- the encoder is a bidirectional recurrent network (see Section 5.16.2) – an LSTM with the input and output layers having 2,048 nodes and a single hidden layer of 512 cells; and
- the decoder is a hierarchical 2-level recurrent network, composed of
  - a high-level RNN named the conductor – an LSTM with the input and output layers having 512 nodes and a single hidden layer of 1,024 cells – that produces a sequence of embeddings; and
  - a bottom-layer RNN – an LSTM with two hidden layers of 1,024 cells – that uses each embedding as an initial state and also as an additional input concatenated to its previously generated token[84] to produce each subsequence. In order to prioritize the conductor RNN over the bottom-layer RNN, its initial state is reinitialized with the decoder generated embedding for each new subsequence. In the case of a multivoice trio (melody, bass and drums), there are three LSTMs, one for each voice.

The MusicVAE architecture is illustrated in Figure 6.61. The authors report that an equivalent "flat" MusicVAE architecture (without hierarchy), although accurate in modeling the style in the case of 2 measures long examples, was inaccurate in the case of 16 measures long examples, with a 27% error increase for the autoencoder reconstruction (0.883 accuracy for the flat architecture and 0.919 accuracy for the hierarchical architecture).

An example of trio music generated is shown in Figure 6.62. A preliminary evaluation has been conducted with listeners comparing three versions (flat architecture, hierarchical architecture and real music) for three types of music: melody, trio and drums. The results show a very significant gain with the hierarchical architecture, see more details in [148] and [160].

An interesting feature of the variational autoencoder architecture is in the capacity for exploring the latent space via various operations such as

- translation;
- interpolation – Figure 5.17 in Section 5.9.2 shows an interesting comparison of melodies resulting from interpolation in the data space (that is the space of representation of melodies) and interpolation in the latent space which is then decoded into the corresponding melodies. One can see (and hear) that the interpolation in the latent space produces much more meaningful and interesting melodies;
- averaging – Figure 6.63 shows an example of a melody (in the middle of the figure) generated from the combination (averaging) of the latent spaces of two melodies (at the top and bottom of the figure);
- addition or subtraction of an attribute vector capturing a given characteristic – Figure 6.64 shows an example of a melody (at the bottom of the figure) generated when a "high note density" attribute vector is added to the latent space of an existing melody (at the top of the figure). An attribute vector is computed as the average of the latent vectors for a collection of examples sharing that characteristic (attribute) (e.g., high density of notes, rapid change, high register, etc.).

Furthermore, Figure 6.65 shows the effect, as a percent change, of modifying individual attributes of 16 measures long melodies by adding (left matrix), or respectively subtracting (right matrix), attribute vectors in the latent space. The vertical axis of each correlation matrix denotes the attribute vector applied and the horizontal axis denotes the attribute measured. These correlation matrixes show that individual attributes can be modified without effecting others, except for the cases when correlations are expected, as for instance between the eighth and sixteenth note syncopations.

Audio examples are available in [150] and [147]. MusicVAE is summarized in Table 6.28.

[83] Only one "Note off" event is needed for all possible pitches as the melody is monophonic. It is used to differentiate a note held from two successive identical notes, see Section 4.9.1.

[84] Along the iterative feedforward strategy.

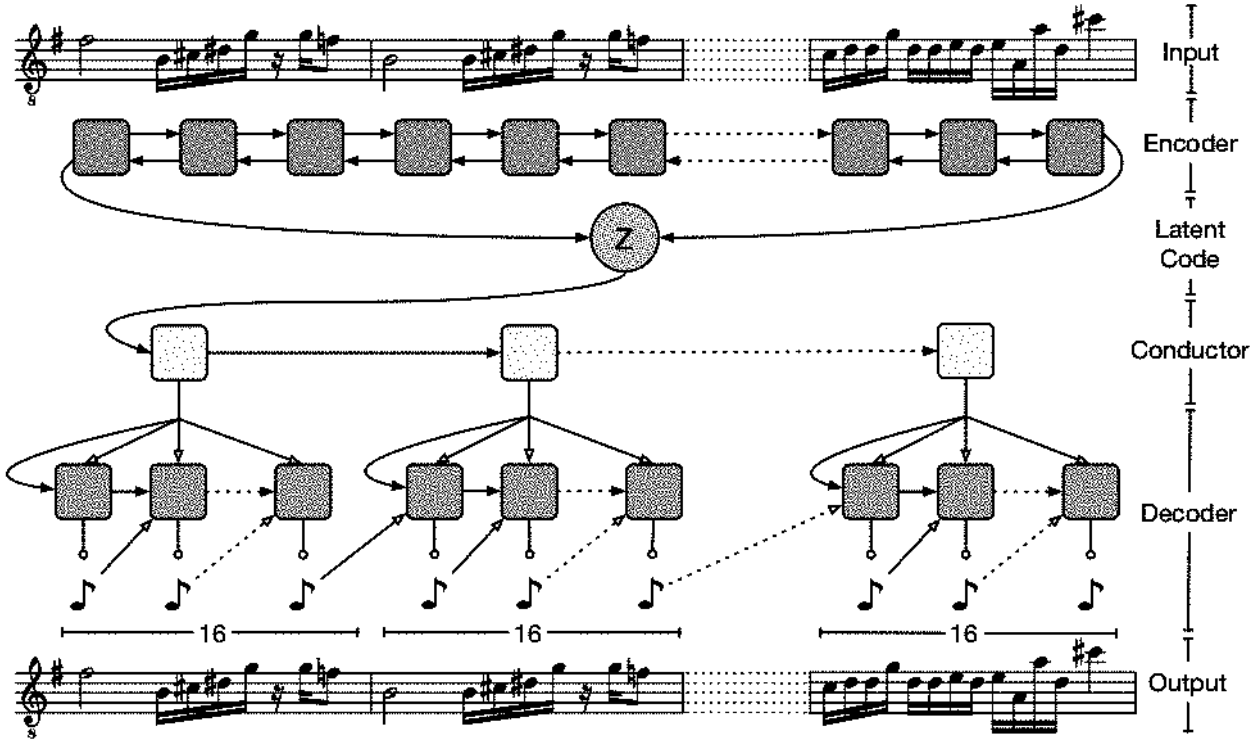

Fig. 6.61 MusicVAE architecture. Reproduced from [148] with the permission of the authors

Objective	Melody; Trio (Melody, Bass, Drums)
Representation	Symbolic; Drums; Note end; Rest
Architecture	Variational Autoencoder(Bidirectional-LSTM, Hierarchical^2-LSTM)
Strategy	Iterative feedforward; Sampling; Latent variables manipulation

Table 6.28 MusicVAE summary

6.12 Originality

The issue of the *originality* of the music generated is not only an artistic issue (*creativity*) but also an economic one, because it raises the issue of the copyright[85].

One approach is *a posteriori*, by ensuring that the generated music is not too similar (e.g., in not having recopied a significant number 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[86]. A solution for music generation from Markov chains has been proposed [140]. It is based on a variable order Markov model and constraints over the order of the generation through some min order and max order constraints, in order to attain some sweet spot between junk and plagiarism. However, there is not yet a solution for deep learning architectures.

Let us now analyze some recent directions for favoring originality in the generated musical content.

[85] On this issue, see the recent paper by Deltorn [31].

[86] Note that this addresses the issue of significant recopying from the training corpus, but it does not prevent a system from *reinventing* existing music outside of the training corpus.

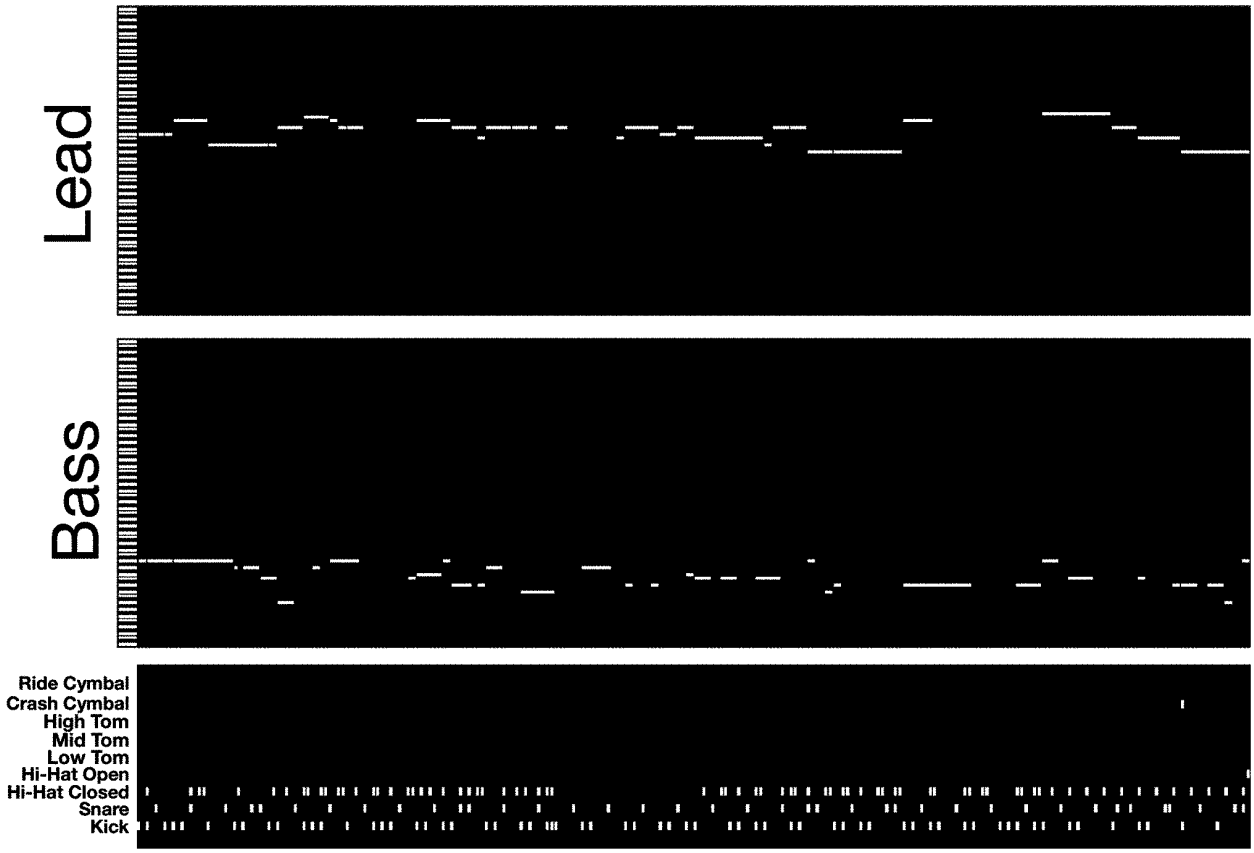

Fig. 6.62 Example of a trio music generated by MusicVAE. Reproduced from [149] with the permission of the authors

6.12.1 Conditioning

6.12.1.1 Example: MidiNet Melody Generation System

In their description of MidiNet [191] (see Section 6.9.3.3), the authors discuss two methods to control creativity:

- restricting the conditioning by inserting the conditioning data only in the intermediate convolution layers of the generator architecture; and
- decreasing the values of the two control parameters of feature matching regularization, in order to reduce the requirement for the closeness of the distributions of real data and generated distributions of real and generated data.

These experiments are interesting but they remain at the level of *ad hoc* tuning of the hyper-parameters of the architecture.

6.12.2 Creative Adversarial Networks

Another more systematic and conceptual direction is the concept of *creative adversarial networks (CAN)* proposed by Elgammal *et al.* [39], as an extension of the generative adversarial networks (GAN) architecture (introduced in Section 5.14).

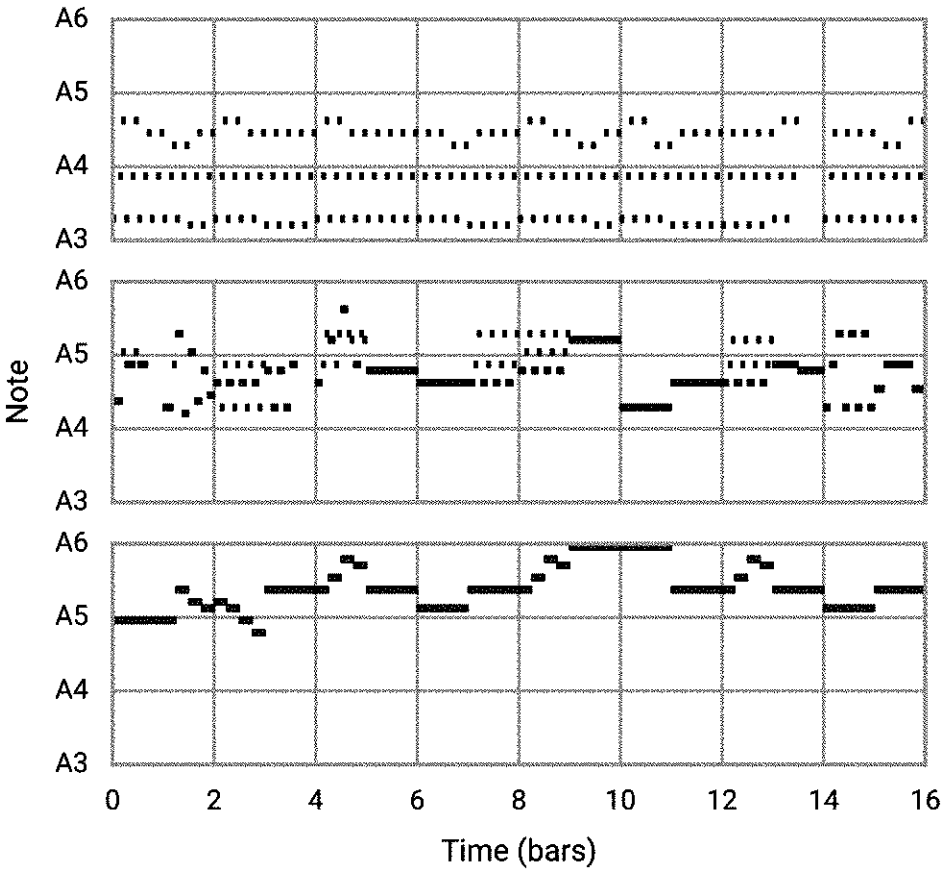

Fig. 6.63 Example of a melody generated (middle) by MusicVAE by averaging the latent spaces of two melodies (top and bottom). Reproduced from [148] with the permission of the authors

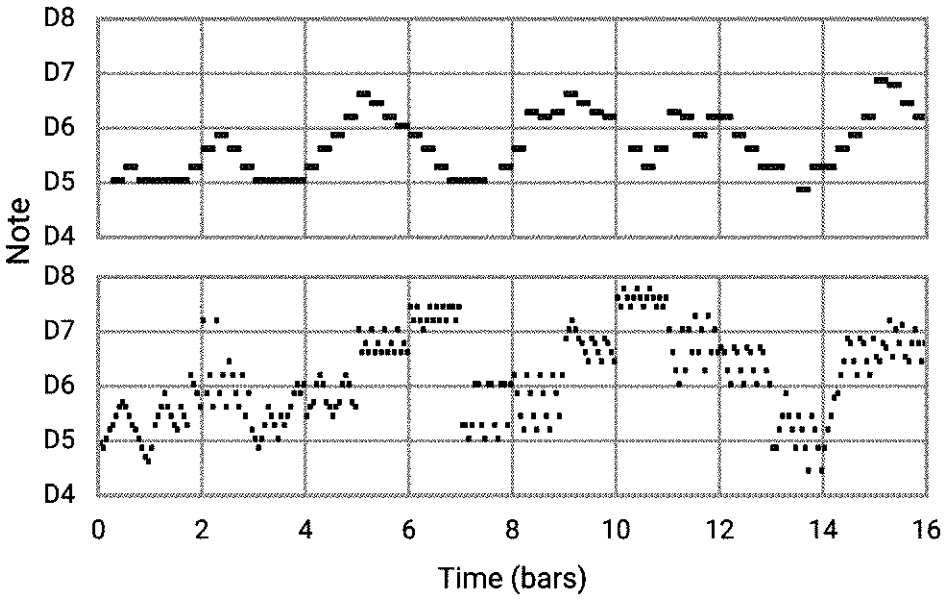

Fig. 6.64 Example of a melody generated (bottom) by MusicVAE by adding a "high note density" attribute vector to the latent space of an existing melody (top). Reproduced from [149] with the permission of the authors

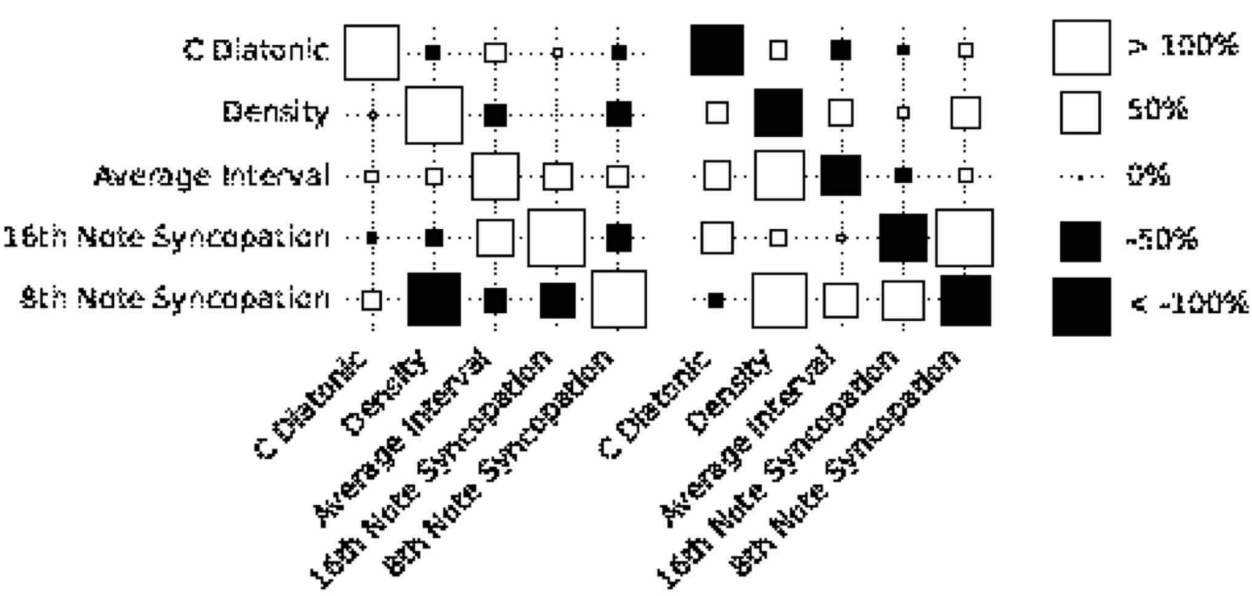

Fig. 6.65 Correlation matrices of the effect of adding (left) of subtracting (right) an attribute to other attributes in MusicVAE. Reproduced from [148] with the permission of the authors

6.12.2.1 Creative Adversarial Networks Painting Generation System

Elgammal *et al.* propose in [39] to address the issue of *creativity* by extending a generative adversarial networks (GAN) architecture into a creative adversarial networks (CAN) architecture to "generate art by learning about styles and deviating from style norms." [39].

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 the proposed creative adversarial networks (CAN) (illustrated in Figure 6.66), the generator receives from the discriminator not just one but *two* signals:

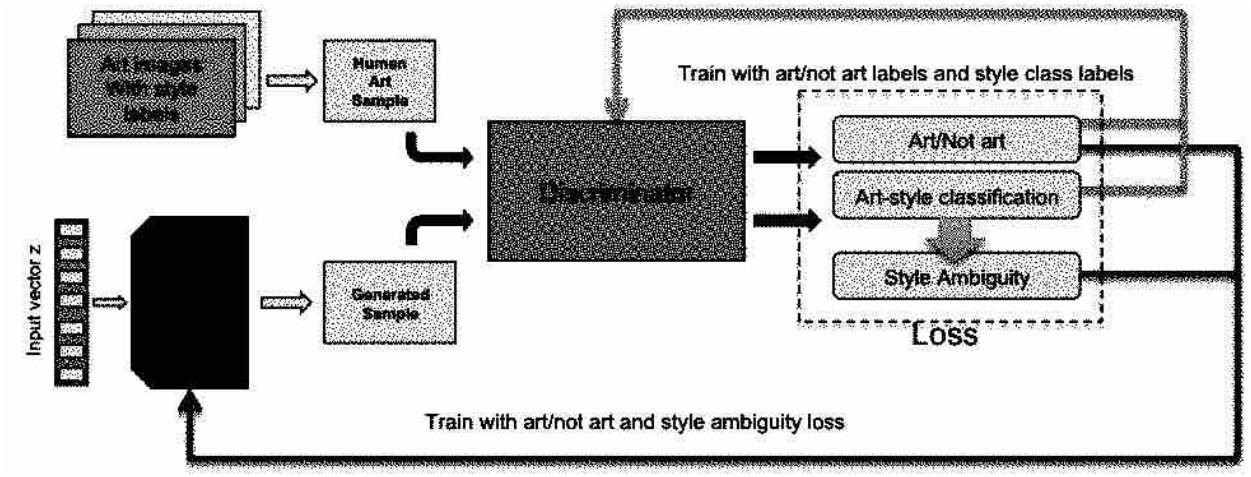

Fig. 6.66 Creative adversarial networks (CAN) architecture. Reproduced from [39] with the permission of the authors

- the first signal is analog to the case of the standard GAN (see Equation 5.18) and is the discriminator's estimation whether the generated sample is real or faked art; and
- the second signal is about how easily the discriminator can *classify* the generated sample into predefined *established styles*. If the generated sample is *style-ambiguous* (i.e. the various classes are *equiprobable*), this means that the sample is difficult to fit within the existing art styles, which may be interpreted as the creation of a *new style*.

These two signals are contradictory forces which push the generator to explore the space for generating items that are close to the distribution of existing art pieces *and* style-ambiguous.

Experiments have been done with paintings from a WikiArt dataset [185]. This collection has images of 81,449 paintings from 1,119 artists ranging from the fifteenth century to the twentieth century. It has been tagged with 25 possible painting styles (e.g., cubism, fauvism, high-renaissance, impressionism, pop-art, realism, etc.). Some examples of images generated by CAN are shown in Figure 6.67.

Fig. 6.67 Examples of images generated by CAN. Reproduced from [39] with the permission of the authors

As the authors discuss, the generated images are not recognized like traditional art, in terms of standard genres (portraits, landscapes, religious paintings, still lifes, etc.), as shown by a preliminary external human evaluation and also a preliminary analysis of their approach. Note that the CAN approach assumes the existence of a prior style classification and reduces the idea of creativity to exploring 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 ex-

periment with other types of classification, including styles which are automatically constructed. Experimenting with the transposition of the CAN approach to music generation appears as a tempting direction.

6.13 Incrementality

A straightforward use of deep architectures for generation leads to a one-shot generation of a musical content as a whole in the case of a feedforward or autoencoder network architecture, or to an iterative generation of time slices of a musical content in the case of a recurrent network architecture. This is a strong limitation if we compare this to the way a human composer creates and generates music, in most cases very incrementally, though successive refinements of arbitrary parts.

6.13.1 Note Instantiation Strategies

Let us review how notes are instantiated during generation. There are three main strategies:

- *Single-step feedforward* – a feedforward architecture processes in a single processing step a global representation which includes all time steps. An example is MiniBach (Section 6.1.2).
- *Iterative feedforward* – a recurrent architecture iteratively processes local representations corresponding to a single time step. An example is CONCERT (Section 6.5.1.1).
- *Incremental sampling* – a feedforward architecture incrementally processes a global representation which includes all time steps, by incrementally instantiating its variables (each variable corresponding to the possibility of a note at a specific time step). An example is DeepBach (Section 6.13.2).

These three strategies are compared and illustrated in Figure 6.68. The representation is piano roll type with two simultaneous voices (or tracks). The blue cells are the notes to be played. The rectangles with a thick line labeled as "current" indicate the parts being processed, whereas the parts in light grey indicate the parts already processed.

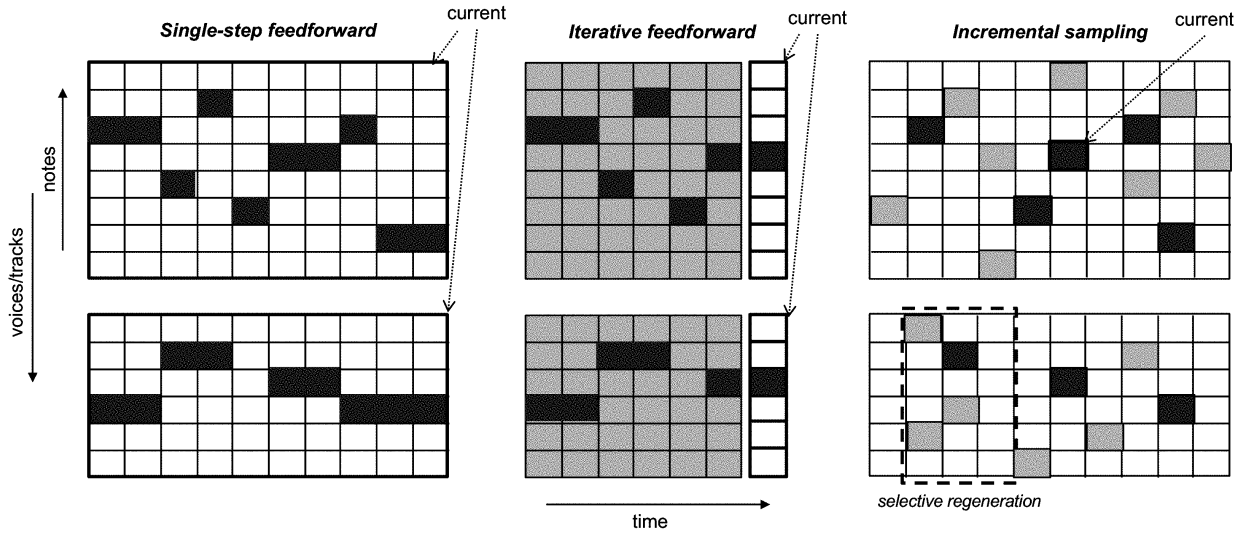

Fig. 6.68 Note generation/instantiation – three main strategies

In the case of the incremental sampling strategy (right part of Figure 6.68), at each processing step a new cell representing a triplet $(voice, note, time step)$, labeled as "current", is randomly chosen and instantiated. Triplets already instantiated are blue-filled if a note is to be played and light grey-filled otherwise.

Note that, with this incremental sampling strategy, it is possible to only generate or to *regenerate* an arbitrary part (slice) of the musical content, for a specific time interval between two time steps and/or for a specific subset of voices/tracks, without the need for regenerating the whole content. In Figure 6.68, the dashed rectangle indicates a zone selected by the user to perform a selective regeneration[87].

6.13.2 Example: DeepBach Chorale Multivoice Symbolic Music Generation System

Hadjeres *et al.* have proposed the DeepBach architecture[88] for the generation of J. S. Bach chorales [65]. The architecture, shown in Figure 6.69, combines two recurrent networks (LSTMs) and two feedforward networks. As opposed to the standard use of recurrent networks where a single time direction is considered[89], DeepBach architecture considers two directions: *forward* in time and *backward* in time[90]. Therefore, two recurrent networks (more precisely, LSTMs with 200 cells) are used, one summing up past information and another summing up information coming from the future, together with a nonrecurrent network in charge of notes occurring at the same time. Their three outputs are merged and passed to the input of a final feedforward neural network, with one hidden layer with 200 units. The final output activation function is softmax.

The initial corpus is the set of J. S. Bach's polyphonic (multivoice) chorales [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. 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 contain up to 28 different pitches for each voice[91].

The choice of the representation in DeepBach has some specificities. A hold symbol "_" is used to indicate whether a note is being held (see Section 4.9.1). The authors emphasize in [65] that this representation is well-suited to the sampling method used, 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. Another specificity is that the representation consists in encoding notes using their real names and not their MIDI note numbers (e.g., F♯ is considered separately from G♭, see Section 4.9.2). Last, the fermata symbol for Bach chorales is explicitly considered as it helps to produce structure and coherent phrases.

The first four lines of the example data at top of Figure 6.69 correspond to the four voices. The two bottom lines correspond to metadata (fermata and beat information). Actually this architecture is replicated four times, one for each voice (four in a chorale).

Training, as well as generation, is not done in the conventional way for neural networks. The objective is to predict the value of the current note for a given voice (shown in light green with a red "?", in top center of Figure 6.69), using as input information the surrounding contextual notes and their associated metadata, more precisely

- the three current notes for the three other voices (the thin rectangle in light blue in top center);
- the six previous notes (the rectangle in light turquoise blue in top left) for all voices; and
- the six next notes (the rectangle in light grey blue in top right) for all voices.

---

[87] With the single-step feedforward strategy, one could imagine selecting only the desired slice from the regenerated content and "copy/pasting" it into the previously generated content, but with the obvious absence of a guarantee that the old and the new parts will be consistent.

[88] The MiniBach architecture described in Section 6.1.2 is actually a deterministic single-step feedforward (major) simplification of the DeepBach architecture.

[89] An exception is, for example, the bidirectional recurrent architecture used in the C-RNN-GAN system analyzed in Section 6.9.2.4.

[90] The authors state that this architectural choice somewhat matches the real compositional practice of Bach chorales. Indeed, when reharmonizing a given melody, it is often simpler to start from the cadence and write music *backward* [65].

[91] 21 for the soprano, alto and tenor parts and 28 for the bass part.

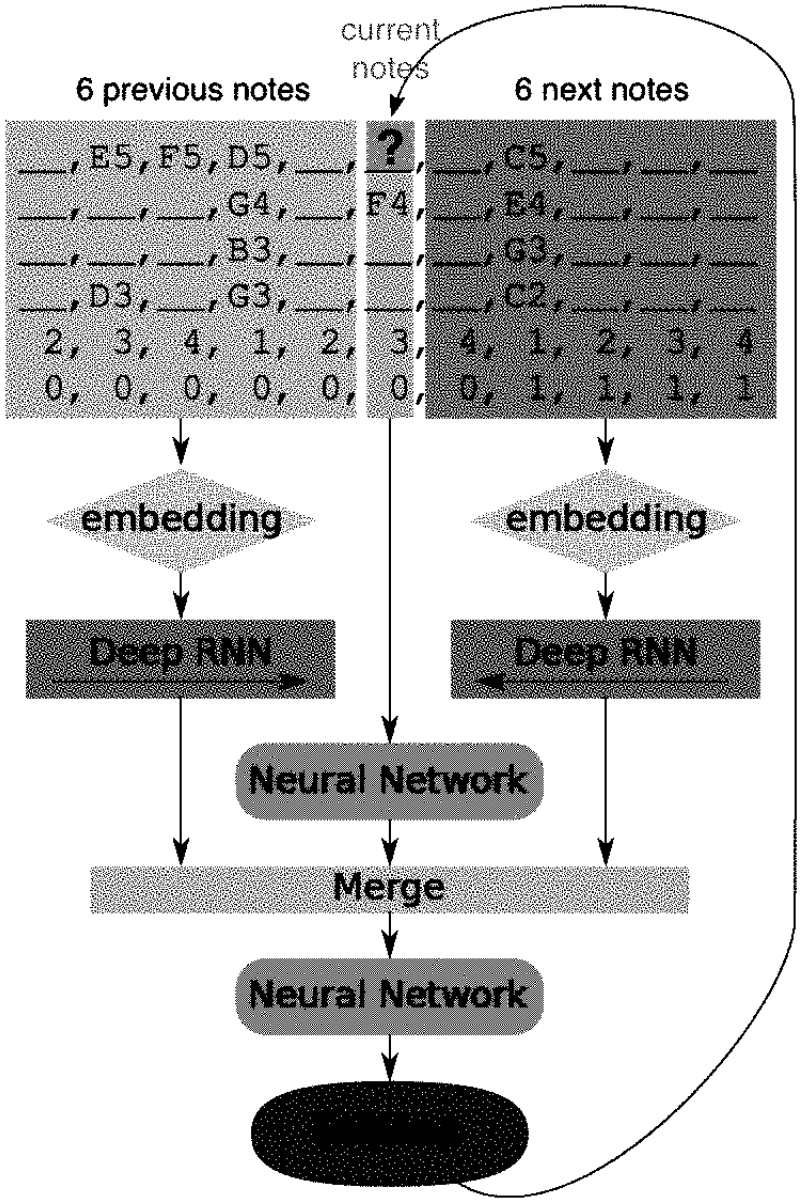

Fig. 6.69 DeepBach architecture. Reproduced from [65] with the permission of the authors

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 previously defined).

Generation is performed by incremental sampling, using a pseudo-Gibbs sampling algorithm analog to but computationally simpler than Gibbs sampling algorithm[92] (see Section 6.3.2.1), to produce a set of values (each note) of a polyphony, following the distribution that the network has learnt. The algorithm for generation by incremental sampling is shown in Figure 6.70 and has been illustrated in Figure 6.68.

[92] The difference with Gibbs sampling (based on the non-assumption of compatibility of conditional probability distributions) and the algorithm are detailed and discussed in [65].

Create four lists $V = (V_1; V_2; V_3; V_4)$ of length L;
Initialize them with random notes drawn from the ranges of the corresponding voices
(sampled uniformly or from the marginal distributions of the notes);
for m from 1 to *max number of iterations* **do**
Choose voice i uniformly between 1 and 4;
Choose time t uniformly between 1 and L;
Re-sample V_i^t from $p_i(V_i^t | V_{\setminus i,t}, \theta_i)$
end for

Fig. 6.70 DeepBach incremental generation/sampling algorithm

An example of a chorale generated[93] is shown in Figure 6.71. As opposed to many experiments, a systematic evaluation in a Turing-type test has been conducted (with more than 1,200 human subjects, from experts to novices, via a questionnaire on the Web[94]) and the results are very positive, showing a significant difficulty to discriminate between chorales composed by Bach and chorales generated by DeepBach. DeepBach is summarized in Table 6.29.

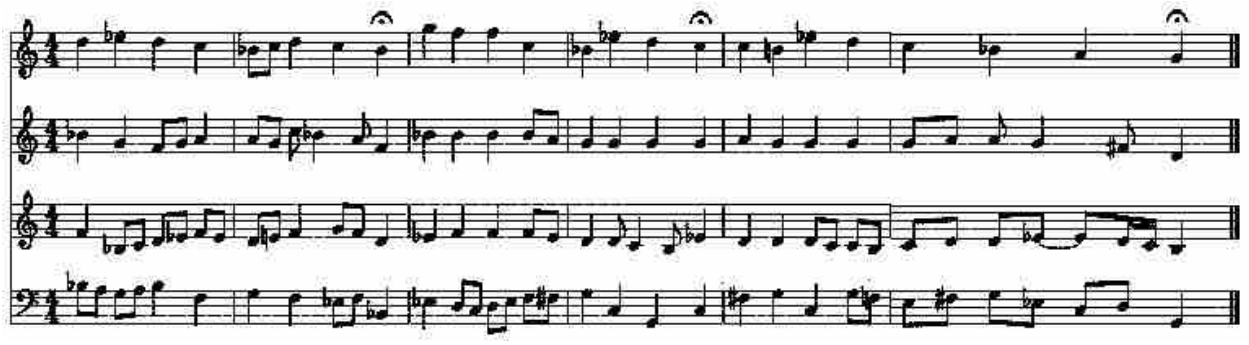

Fig. 6.71 Example of a chorale generated by DeepBach. Reproduced from [65] with the permission of the authors

Objective	Multivoice; Counterpoint; Chorale; Bach
Representation	Symbolic; Piano roll; One-hot; Hold; Rest; Fermata
Architecture	Feedforward×2 + LSTM×2
Strategy	Sampling

Table 6.29 DeepBach summary

6.14 Interactivity

An important issue is that, for most current systems, generation of musical content is an automated and autonomous process. Some *interactivity* with a human user(s) is fundamental to obtaining a companion system to help humans in their musical tasks (composition, counterpoint, harmonization, analysis, arranging, etc.) in an incremental and interactive manner. An example, already introduced in Section 6.10.4, is the FlowComposer prototype [141].

[93] We will see in Section 6.14.2 that DeepBach may also be used for a different objective: counterpoint accompaniment.

[94] An evaluation was also conducted during a live program on a Dutch TV channel.

A couple of examples of partially interactive incremental systems based on deep network architectures are deepAutoController (Section 6.3.1.2) and DeepBach (Section 6.13.2).

6.14.1 #1 Example: deepAutoController Audio Music Generation System

The deepAutoController system [155] introduced in Section 6.3.1.2 provides a user interface, shown in Figure 6.72, to interactively control the generation, for instance by

- selecting a given input,
- generating a random input to be feedforwarded into the decoder stack, or
- controlling (by scaling or muting) the activation of a given unit.

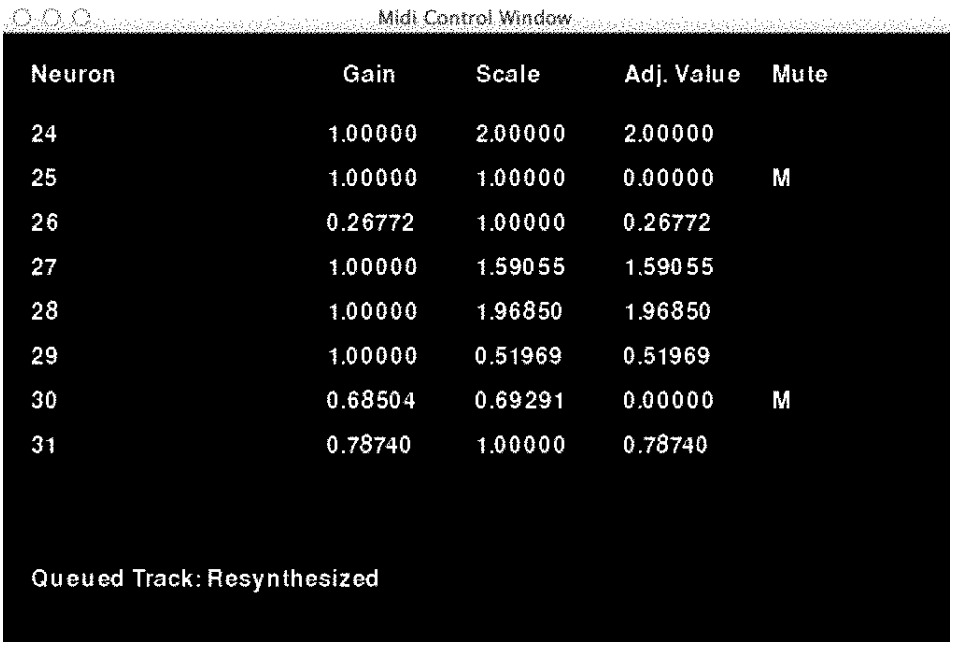

Fig. 6.72 Snapshot of a deepAutoController information window showing hidden units. Reproduced from [155] with the permission of the authors

6.14.2 #2 Example: DeepBach Chorale Symbolic Music Generation System

The user interface of DeepBach [65] (see Section 6.13.2) is implemented as a plugin for the MuseScore music editor (see Figure 6.73). It helps the human user to interactively select and control partial regeneration of chorales. This is made possible by the incremental nature of the generation (see Section 6.13). Moreover, the user can enforce some user-defined constraints, such as

- freezing a voice (e.g., the soprano) and resampling the other voices in order to reharmonize the fixed melody[95];

[95] In practice, this means changing from the original objective of generating a *4-voice polyphony* from scratch as discussed in Section 6.13.2, to generating a *3-voice counterpoint accompaniment* for a given melody.

- modifying the fermata list in order to impose an end to musical phrases at specific places;
- restricting the note range for a given voice and a given temporal interval; and
- imposing a rhythm by restricting the note range to the hold symbol (as it is considered as a note) in specific parts.

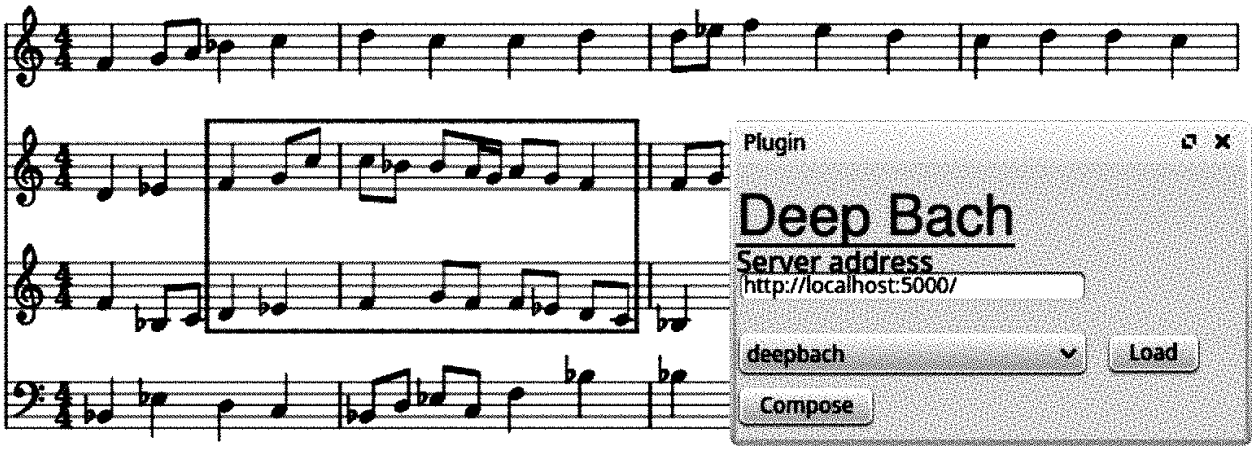

Fig. 6.73 DeepBach user interface. Reproduced from [65] with the permission of the authors

6.14.3 Interface Definition

Let us finally mention, at the junction between *control* and *interactivity*, the interesting discussion by Morris *et al.* in [126] on the issue of what control parameters (for music generation by a Markov chain trained model) should be *exposed* at the human user level. Some examples of user-level control parameters they have experimented with are as follows

- major vs minor;
- following melody vs following chords; and
- locking a feature (e.g., a chord).

6.15 Adaptability

One fundamental limitation of current deep learning architectures for the generation of musical content is that they paradoxically do *not* learn or adapt. Learning is applied during the *training* phase of the network, but no learning or adaptation occurs during the *generation* phase. However, one can imagine some feedback from a user, e.g., the composer, producer, listener, about the quality and the adequacy of the generated music. This feedback may be explicit, which puts a task on the user, but it could also be, at least partly, implicit and automated. For instance, the fact that the user quickly stops listening to the music just generated could be interpreted as negative feedback. On the contrary, the fact that the user selects a better rendering after a first quick listen to some initial reproduction could be interpreted as positive feedback.

Several approaches are possible. The most straightforward approach, considering the nature of neural networks and supervised learning, would be to add the newly generated musical piece to the training set and eventually[96] retrain

[96] Immediately, after some time, or after some amount of new feedback, as with a minibatch (see Section 5.2.4).

the network[97]. This would reinforce the number of positive examples and gradually update the learnt model and, as a consequence, future generations. However, there is no guarantee that the overall generation quality would improve. This could also lead the model to overfit and loose some generalization. Moreover, there is no direct possibility of negative feedback, as one cannot remove a badly generated example from the dataset because there is almost no chance that it was already present in the dataset.

At the junction between *adaptability* and *interactivity*, an interesting approach is that of interactive machine learning for music generation, as discussed by Fiebrink and Caramiaux [45]. They report on experience with a toolkit they designed, named Wekinator, to allow users to interactively modify the training examples. For instance, they argue in [45] that: "Interactive machine learning can also allow people to build accurate models from very few training examples: by iteratively placing new training examples in areas of the input space that are most needed to improve model accuracy (e.g., near the desired decision boundaries between classes), users can allow complicated concepts to be learned more efficiently than if all training data were representative of future data."

Another approach is to work not on the training dataset but on the generation phase. This leads us back to the issue of control (see Section 6.9), via, for example, a constrained sampling strategy, an input manipulation strategy or, obviously, a reinforcement strategy. The RL-Tuner framework (Section 6.9.6.1) is an interesting step in this direction. Although the initial motivation for RL-Tuner was to introduce musical constraints on the generation, by encapsulating them into an additional reward, this approach could also be used to introduce user feedback as an additional reward.

6.16 Explainability

A common critique of sub-symbolic approaches of Artificial Intelligence (AI)[98], such as neural networks and deep learning, is their *black box* nature, which makes it difficult to explain and justify their decisions [16]. Explainability is indeed a real issue, as we would like to be able to understand and explain what (and how) a deep learning system has learned from a corpus as well as why it ends up generating a given musical content.

6.16.1 #1 Example: BachBot Chorale Polyphonic Symbolic Music Generation System

Although preliminary, an interesting study conducted with the BachBot system concerns the analysis of the specialization of some of the units (neurons) of the network, through a correlation analysis with some specific motives and progressions.

BachBot, by Liang [110, 109], is a system designed to generate chorales in the style of J. S. Bach, an objective shared by DeepBach (Section 6.13.2). All examples from the dataset are aligned onto the same key. The initial representation is piano roll but it is encoded in text, in a similar way to the Celtic melody generation system described in Section 6.5.1.2.

One of the specificities of the encoding is the way simultaneous notes are encoded as a sequence of tokens, with a special delimiter symbol "|||" indicating the next time frame, with a constant time step of an eighth note. Actually, a chorale is considered in BachBot as a single-voice polyphony and not as a multivoice polyphony, as for instance in the cases of DeepBach (Section 6.13.2) and MiniBach (Section 6.1.2). Rests are encoded as empty frames. Notes are ordered in a descending pitch and are represented by their MIDI note number, with a boolean indicating if it is tied to a note at the same pitch from previous time step[99]. An example is shown in Figure 6.74, encoding two successive chords:

- notes B_3, $G\sharp_3$, E_3 and B_2, corresponding to a E major with a B as the bass (often notated as E/B in Jazz) with the duration of a quarter note, and repeated with a tied note;

[97] This could be done in the background.

[98] As opposed to symbolic approaches, see Section 1.2.3.

[99] This is equivalent to a hold "_" indication.

- a fermata, notated as "(.)"; and
- notes A_3, E_3, C_3 and A_2, corresponding to a A minor with the duration of an eighth note.

```
(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. 6.74 Example of score encoding in BachBot. Reproduced from [110]

The architecture is a recurrent network (LSTM). The author used a grid search in order to select the optimal setting for hyperparameters of the architecture (number of layers, number of units, etc.). The selected architecture has three layers and as the author notes in [110]: "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." Generation is done time step by time step, following the iterative feedforward strategy.

As for DeepBach (Sections 6.13.2 and 6.14.2), BachBot may be readapted from the initial 4-voice multivoice chorale generation objective to a melody 3-voice counterpoint accompaniment objective, see details in [110, Section 6.1]. However, as opposed to DeepBach architecture and representation which stay unchanged, in the case of BachBot both the architecture, the representation temporal scope and the strategy have to be *structurally changed* from a time step/iterative generation approach to a global/single-step generation approach (similar to MiniBach).

An interesting preliminary study by the author was the invitation of a musicologist to manually search for possible correlations between unit activation and specific motives and progressions, as shown in Figure 6.75. Some examples of the correlations found[100] are as follows:

- Neurons 64 and 138 of Layer 1 seem to detect (specifically) perfect cadences (V–I) with root position chords in the tonic key.
- Neuron 87 of Layer 1 seems to detect an I chord on the first downbeat and its reprise four measures later.
- Neuron 151 of Layer 1 seems to detect A minor cadences that end phrases 2 and 4.
- Neuron 37 of Layer 2 seems to be looking for I chords: strong peak for a full I and weaker for other similar chords (same bass).

BachBot is summarized in Table 6.30.

Objective	Multivoice; Chorale; Bach
Representation	Symbolic; Text; One-hot; Hold; Fermata
Architecture	$LSTM^3$
Strategy	Iterative feedforward; Sampling

Table 6.30 BachBot summary

[100] More details may be found in [110, chapter 5].

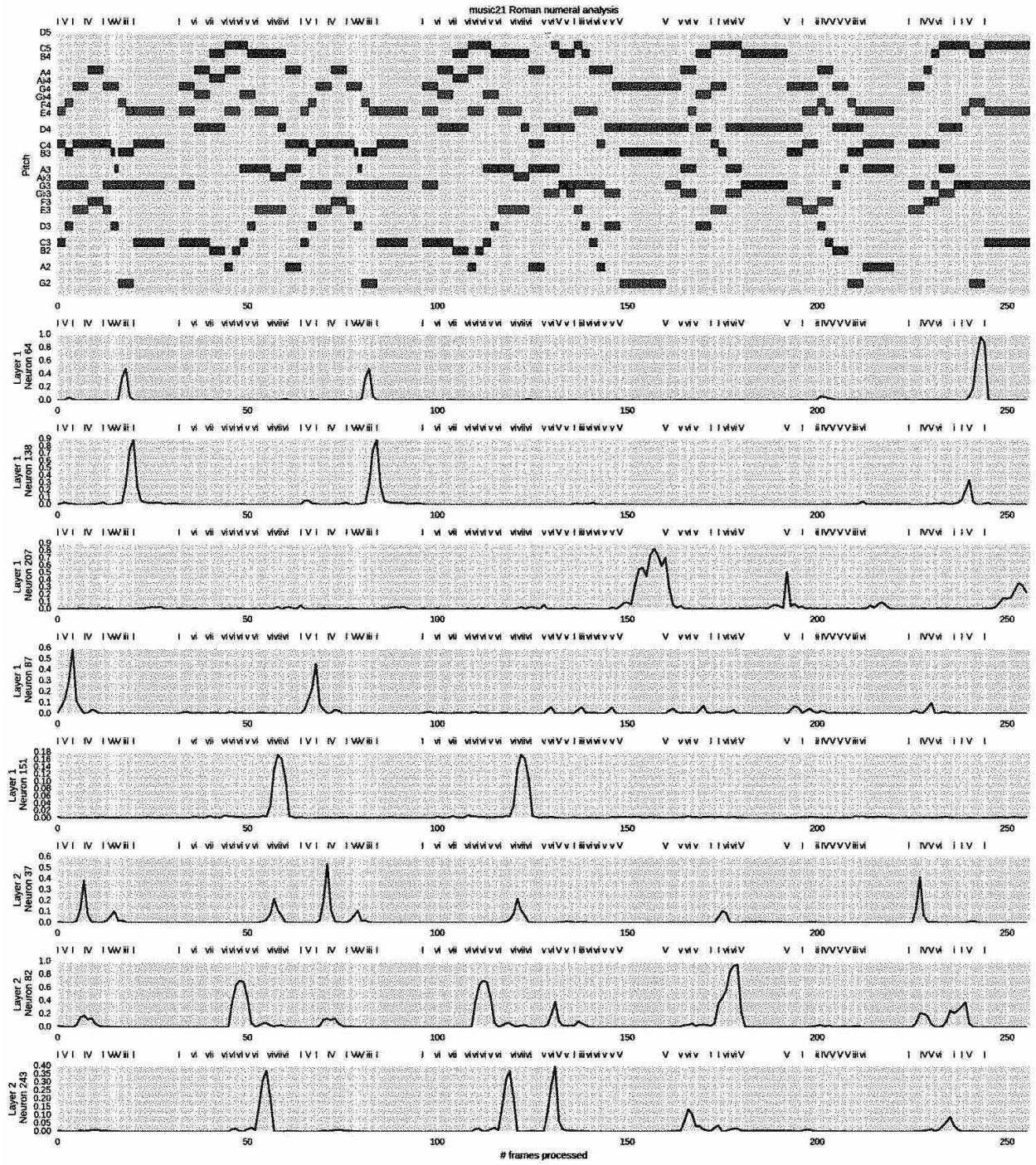

Fig. 6.75 Correlation analysis of BachBot layer/unit activation. Reproduced from [110]

6.16.2 #2 Example: deepAutoController Audio Music Generation System

In [155], the authors of deepAutoController (Section 6.3.1.2) discuss the musical effects of different controls over the units of the architecture: "The optimal parameters of the models were mostly inhibitory. Therefore the deactivation of a unit in a hidden layer yields a denser mixture of sounds at the output. Learning to play such an interface may prove difficult for new users, as one typically expects the opposite behavior from a musical synthesizer. <...> We explored models having non-negative weights by using an asymmetric weight decay as shown in [107]. The results are not presented here as they are preliminary. Reconstruction error in such models is worse than without non-negativity constraints. But we find informally that the models are somewhat more intuitive to play as synthesizers."

6.16.3 Automated Correlation Analysis

The two previous examples in Sections 6.16.1 and 6.16.2 are examples of a preliminary manual correlation analysis. Meanwhile, an active area of research relates to the understanding of the way deep learning architectures work and the explanation of their predictions or decisions via automated analyses. An example of such an approach is using saliency maps with the three following categories[101]:

- gradient sensitivity, to estimate how a small change to the input can affect the classification task (see, for example, [5]);
- signal methods, to isolate input patterns that stimulate neuron activation in higher layers (see, for example, [93]); and
- attribution methods, to decompose the value at a specific output neuron into contributions from the individual input dimensions[102] (see, for example, [124]).

Note that this type of analysis could also be used with a different objective: to optimize the configuration of the architecture by removing components that are considered to make no contribution and are therefore unnecessary, see, for example, [105] (with its provocative title).

6.17 Discussion

We can observe that the various limitations and challenges that we have analyzed may be partially dependent on one another and, furthermore, conflicting. Thus, resolving one may damper another. For instance, the sampling strategy used by DeepBach (Section 6.13.2) provides incrementality but the length of the generated music is fixed, whereas the iterative feedforward strategy allows variable and unbounded length but incrementality is only forward in time.

There is probably no general solution and, as for multicriteria decisions, the selection of architectures and strategies depends on preferences and priorities. Also, as already noted in Section 5.16.7, there is no guarantee that combining a variety of different architectures and/or strategies will make a sound and accurate system. As for a good cook, the best outcome is not achieved by simply mixing together all the possible ingredients. Therefore, it is important to continue to deepen our understanding and to explore solutions as well as their possible articulations. We hope that the survey and analysis conducted in this chapter and in the two next chapters provide a contribution to this understanding.

[101] Following Kindermans *et al.*'s study in [92], actually a critique of the reliability of saliency methods.

[102] With an approach analog to reverse correlation, which is used in neurophysiology for studying how sensory neurons add up signals from different sources and sum up stimuli at different times to generate a response (see, for example, [145]).

Chapter 7
Analysis

We now present a preliminary analysis and summary of the various systems surveyed, following our proposed five dimensions referential through various tables. This provides material for an analysis of the relations between the different dimensions and the corresponding design decisions.

7.1 Referencing and Abbreviations

At first, we reference in Table 7.1 the various systems that we have analyzed.

Then, because of space limitations, we introduce abbreviations for the various possible types for each dimension:

- *objectives* in Table 7.2;
- *representations* in Table 7.3;
- *architectures* in Table 7.4;
- *challenges* in Table 7.5; and
- *strategies* in Table 7.6.

System				
Reference name	*Original name*	*Authors*	*Refer.*	*Section*
Anticipation-RNN	Anticipation-RNN	G. Hadjeres & F. Nielsen	[63]	6.9.3.5
AST		D. Ulyanov & V. Lebedev	[174]	6.10.2.1
(Audio Style Transfer)		D. Foote *et al.*	[46]	6.10.2.1
BachBot	BachBot	F. Liang	[110]	6.16.1
Bi-Axial LSTM	Bi-Axial LSTM	D. Johnson	[88]	6.8.3
Blues_C		D. Eck & J. Schmidhüber	[37]	6.4.1.1
Blues_{MC}		D. Eck & J. Schmidhüber	[37]	6.4.1.2
Celtic		B. Sturm *et al.*	[163]	6.5.1.2
CONCERT	CONCERT	M. Mozer	[127]	6.5.1.1
C-RBM	C-RBM	S. Lattner *et al.*	[100]	6.9.5.1
C-RNN-GAN	C-RNN-GAN	O. Mogren	[123]	6.9.2.4
deepAutoController	deepAutoController	A. Sarroff & M. Casey	[155]	6.3.1.2
DeepBach	DeepBach	G. Hadjeres *et al.*	[65]	6.13.2
DeepHear_C	DeepHear	F. Sun	[164]	6.9.4.1
DeepHear_M	DeepHear	F. Sun	[164]	6.3.1.1
DeepJ	DeepJ	L.-C. Mao *et al.*	[116]	6.9.3.4
GLSR-VAE	GLSR-VAE	G. Hadjeres & F. Nielsen	[64]	6.9.2.3
Hexahedria		D. Johnson	[87]	6.8.2
MidiNet	MidiNet	L.-C. Yang *et al.*	[191]	6.9.3.3
MiniBach	MiniBach	G. Hadjeres & J.-P. Briot		6.1.2
MusicVAE	MusicVAE	A. Roberts *et al.*	[148]	6.11.1
Performance RNN	Performance RNN	I. Simon & S. Oore	[159]	6.6.1
RBM_C		N. Boulanger-Lewandowski *et al.*	[10]	6.8.1
Rhythm		D. Makris *et al.*	[113]	6.9.3.1
RL-Tuner	RL-Tuner	N. Jaques *et al.*	[86]	6.9.6.1
RNN-RBM	RNN-RBM	N. Boulanger-Lewandowski *et al.*	[10]	6.8.1
UnitSelection		M. Bretan *et al.*	[12]	6.9.7.1
VRAE	VRAE	O. Fabius & J. van Amersfoort	[43]	6.9.2.3
VRASH	VRASH	A. Tikhonov & I. Yamshchikov	[171]	6.9.3.6
WaveNet	WaveNet	A. van der Oord *et al.*	[176]	6.9.3.2

Table 7.1 Referencing the systems

Abbreviation	**Objective**
Type	
Au	Audio
Me	Melody (Single-voice monophonic melody)
Po	Polyphony (Single-voice polyphony)
CS	Chord sequence
MV	Multivoice (Multivoice/multitrack polyphony)
Dr	Drums
Co	Counterpoint accompaniment
ST	Style transfer
Destination & Use	
AR	Audio reproduction
SP	Software processing
HI	Human interpretation
Mode	
AG	Autonomous generation
IG	Interactive generation

Table 7.2 Abbreviations for the objectives

Abbreviation	**Representation**
Audio	
Wa	Waveform
Sp	Spectrum
Symbolic	
Concept	
No	Note
Re	Rest
Ch	Chord
Rh	Rhythm (Meter & Beats)
Dr	Drums
Format	
MI	MIDI
Pi	Piano roll
Te	Text
Temporal Scope	
Gl	Global
TS	Time step
Meta-Data	
NH	Note hold or ending
NE	No enharmony (Note denotation)
Fe	Fermata
FE	Feature extraction
Expressiveness	
To	Tempo
Dy	Dynamics
Encoding	
VE	Value encoding
OH	One-hot encoding
MH	Many-hot encoding
Dataset	
Tr	Transposition
Al	Alignment

Table 7.3 Abbreviations for the representations

Abbreviation	**Architecture**
Fd	Feedforward
Ae	Autoencoder
Va	Variational
RB	Restricted Boltzmann machine (RBM)
RN	Recurrent (RNN)
Cv	Convolutional
Cn	Conditioning
GA	Generative adversarial networks (GAN)
RL	Reinforcement learning (RL)
Cp	Compound

Table 7.4 Abbreviations for the architectures

Abbreviation	**Challenge**
EN	*Ex-nihilo* generation
LV	Length variability
CV	Content variability
Es	Expressiveness
MH	Melody-harmony consistency
Co	Control
St	Structure
Or	Originality
Ic	Incrementality
It	Interactivity
Ad	Adaptability
Ey	Explainability

Table 7.5 Abbreviations for the challenges

Abbreviation	**Strategy**
SF	Single-step feedforward
DF	Decoder feedforward
Sa	Sampling
IF	Iterative feedforward
IM	Input manipulation
Re	Reinforcement
US	Unit selection
Cp	Compound

Table 7.6 Abbreviations for the strategies

7.2 System Analysis

We summarize in Tables[1] 7.7 and 7.8 how each system is positioned in respect to each of the five dimensions: *objective*, *representation*, *architecture*, *strategy* and *challenges*.

System				
Name	*Objective*	*Representation*	*Architecture*	*Strategy*
Anticipat-ion-RNN	Melody; Bach	Symbolic; One-hot; Hold; Rest; No enharmony	Condition-ing(LSTM^2, LSTM^2)	Iterative feedforward; Sampling
AST	Audio style transfer	Audio; Spectrum	Convolution-al(Feedforward)	Input manipulation; Single-step feedforward
BachBot	Polyphony; Chorale; Bach	Symbolic; Text; One-hot; Hold; Fermata	LSTM^3	Iterative feedforward; Sampling
Bi-Axial LSTM	Polyphony	Symbolic; Piano roll; Hold; Rest	LSTM$\times$2	Iterative feedforward; Sampling
Blues$_C$	Chord sequence; Blues	Symbolic; One-hot; Note end; Chord as note	LSTM	Iterative feedforward
Blues$_{MC}$	Melody + Chords; Blues	Symbolic; One-hot$\times$2; Note end; Chord as note	LSTM	Iterative feedforward
Celtic	Melody	Symbolic; Text; Token-based; One-hot	LSTM^3	Iterative feedforward; Sampling
CONCERT	Melody + Chords	Symbolic; Harmonics; Harmony; Beat	RNN	Iterative feedforward; Sampling
C-RBM	Polyphony; Style impos-ition	Symbolic; Piano-roll; Rest; Many-hot; Meter	Convolution-al(RBM)	Input manipulation; Sampling
C-RNN-GAN	Polyphony	Symbolic; MIDI	GAN(Birectio-nal(LSTM^2), LSTM^2)	Iterative feedforward; Sampling
deepAuto-Controller	Audio; User interface	Audio; Spectrum	Autoencoder^2	Decoder feedforward
DeepBach	Multivoice; Counterpoint; Chorale; Bach	Symbolic; Piano roll; One-hot; Hold; Rest; Fermata; No enharmony	Feedforward$\times$2 + LSTM$\times$2	Sampling
DeepHear$_C$	Melody accompaniment	Symbolic; Piano roll One-hot	Autoencoder^4	Input manipulation; Decoder feedforward
DeepHear$_M$	Melody; Ragtime	Symbolic; Piano roll; One-hot	Autoencoder^4	Decoder feedforward
DeepJ	Polyphony; Classical; Style	Symbolic; Piano roll; Replay matrix; Rest; Style; Dynamics	Condition-ing(LSTM$^2\times$2, Embedding)	Iterative feedforward; Sampling
GLSR-VAE	Melody; Bach	Symbolic; Piano roll; One-hot; Hold; Rest Fermata; No enharmony	Variational(Auto-encoder(LSTM, LSTM); Geodesic regularization	Decoder feedforward; Sampling
Hexahedria	Polyphony	Symbolic; Piano roll; Hold; Beat	LSTM^{2+2}	Iterative feedforward; Sampling

Table 7.7 Summary (1/2)

We then analyze each system in a more detailed manner, dimension by dimension:

[1] This table is split in two because of vertical space limitations.

System				
Name	*Objective*	*Representation*	*Architecture*	*Strategy*
MidiNet	Melody + Chords; Pop; Melody vs Chords following	Symbolic; Chords Piano roll; One-hot; Rest	GAN(Conditioning(Convolutional(Feedforward, Convolutional(Feedforward(History, Chord sequence))), Conditioning(Convolutional(Feedforward), History))	Single-step feedforward; Sampling
MiniBach	Multivoice; Counterpoint; Chorale; Bach	Symbolic; Piano roll; One-hot×(1+3); Hold	Feedforward[2]	Single-step feedforward
MusicVAE	Melody; Trio (Melody, Bass, Drums)	Symbolic; Drums; Note end; Rest	Variational Autoencoder(Bidirectional-LSTM, Hierarchical[2]-LSTM)	Iterative feedforward; Sampling; Latent variables manipulation
Performance-RNN	Polyphony; Performance control	Symbolic; One-hot; Time shift; Dynamics	LSTM	Iterative feedforward; Sampling
RBM_C	Simultaneous notes (Chord)	Symbolic; Many-hot	RBM	Sampling
Rhythm	Multivoice; Rhythm; Drums	Symbolic; Beat; Drums; Bass line; Note; Rest; Hold	Conditioning(Feedforward(LSTM[2]), Feedforward)	Iterative feedforward; Sampling
RL-Tuner	Melody	Symbolic; One-hot; Note off; Rest	LSTM×2 + RL	Iterative feedforward; Reinforcement
RNN-RBM	Polyphony	Symbolic; Many-hot	RBM-RNN	Iterative feedforward; Sampling
Unit-Selection	Melody	Symbolic; Rest; BOW Features	Autoencoder[2] + LSTM×2	Unit selection; Iterative feedforward
VRAE	Melody; Video game songs	Symbolic;	Variational(Autoencoder(LSTM, LSTM)	Decoder feedforward; Iterative feedforward; Sampling
VRASH	Melody	Symbolic; MIDI; Multi-one-hot	Variational(Autoencoder(LSTM[4], Conditioning(LSTM[4], History)))	Decoder feedforward; Iterative feedforward; Sampling
WaveNet	Audio	Audio; Waveform	Conditioning(Convolutional(Feedforward), Tag); Dilated convolutions	Iterative feedforward; Sampling

Table 7.8 Summary (2/2)

- *objective* in Table 7.9;
- *representation* in Tables[2] 7.10 and 7.11;
- *architecture* and *strategy* in Table 7.12; and
- *challenges* in Table 7.13.

For each table, which analyzes each system (line) in respect to the possible types (columns) for a given dimension, the occurrence of an "X" at the crossing of a given line (system) and a given column (type) means that this system does match that given type for that dimension (e.g., follows some representation facet, is based on some type of architecture,

[2] This table is split in two because of horizontal space limitations.

fulfills some challenge. . .). Note that we base this analysis on how each system *is* presented in the literature referenced, and not as it could be further extended.

Furthermore, we use notations such as X^n and $X \times n$ to convey additional information about the number of occurrences of a type, for example

- X^n for a n-voice objective (e.g., 3-voice melodies for MiniBach in Table 7.9), or a n-layer architecture (e.g., a 2-layer architecture for GLSR-VAE in Table 7.12), and
- $X \times n$ for a n(multi)-one-hot encoding (e.g., a 4-one-hot encoding for Performance RNN in Table 7.11), or a n-instance compound architecture (e.g., 2 RNNs for RL-Tuner in Table 7.12).

	Objective												
	Type								Dest./Use			Mode	
	Au	Me	Po	CS	MV	Dr	Co	ST	AR	SP	HI	AG	IG
System													
Anticipation-RNN		X								X	X	X	
AST	X							X	X			X	
BachBot			X							X	X	X	
Bi-Axial LSTM			X							X	X	X	
Blues_C				X						X	X	X	
Blues_{MC}		X		X	X					X	X	X	
Celtic		X								X	X	X	
CONCERT		X		X	X					X	X	X	
C-RBM			X					X		X	X	X	
C-RNN-GAN			X							X	X	X	
deepAutoController	X								X			X	X
DeepBach		X^3			X		X			X	X	X	X
DeepHear_C		X					X			X	X	X	
DeepHear_M		X								X	X	X	
DeepJ			X							X	X	X	
GLSR-VAE		X								X	X	X	
Hexahedria			X	X	X					X	X	X	
MidiNet		X		X	X					X	X	X	
MiniBach		X^3			X		X			X	X	X	
MusicVAE		X			X	X				X	X	X	
Performance RNN			X							X	X	X	
RBM_C				X						X	X	X	
Rhythm					X	X				X	X	X	
RL-Tuner		X								X	X	X	
RNN-RBM			X							X	X	X	
UnitSelection		X								X	X	X	
VRAE		X								X	X	X	
VRASH		X								X	X	X	
WaveNet	X								X			X	

Table 7.9 System $\times$ Objective

	Representation												
	Audio		Concept					Format			TmpS	
	Wa	Sp	No	Re	Ch	Rh	Dr	MI	Pi	Te	Gl	TS
System												
Anticipation-RNN			X	X					X			X
Audio Style Transfer (AST)		X									X	
BachBot			X	X						X		X
Bi-Axial LSTM			X	X	X	X			X			X
Blues$_C$			X		X				X			X
Blues$_{MC}$			X		X				X			X
Celtic			X			X				X		X
CONCERT			X	X	X	X			X			X
C-RBM			X	X		X			X		X	
C-RNN-GAN			X	X				X				X
deepAutoController		X									X	
DeepBach			X	X					X		X	
DeepHear$_C$			X						X		X	
DeepHear$_M$			X						X		X	
DeepJ			X	X	X	X			X			X
GLSR-VAE			X	X								X
Hexahedria			X		X	X			X			X
MidiNet			X	X	X				X		X	
MiniBach			X						X		X	
MusicVAE			X	X		X	X	X		X		X
Performance RNN			X	X				X				X
RBM$_C$			X		X				X		X	
Rhythm			X	X		X	X		X			X
RL-Tuner			X	X					X			X
RNN-RBM			X		X				X			X
UnitSelection			X	X					X			
VRAE			X								X	X
VRASH			X						X		X	X
WaveNet	X											X

Table 7.10 System $\times$ Representation (1/2)

	Representation										
	Meta-data				Expr.		Encoding			DSet	
	NH	NE	Fe	FE	To	Dy	VE	OH	MH	Tr	Al
System											
Anticipation-RNN	X	X	X					X		X	
Audio Style Transfer (AST)					X	X	X				
BachBot	X		X					X			X
Bi-Axial LSTM	X			X				X	X		X
Blues_C	X							X			
Blues_{MC}	X							X×2			
Celtic								X			X
CONCERT							X	X	X		
C-RBM	X								X	X	
C-RNN-GAN	X						X^4				
deepAutoController					X	X	X				
DeepBach	X	X	X					X×(1+3)		X	
DeepHear_C								X			
DeepHear_M								X			
DeepJ	X			X		X	X	X	X		
GLSR-VAE	X	X	X					X		X	
Hexahedria	X						X	X	X		
MidiNet								X		X	
MiniBach	X	X						X×(1+3)		X	
MusicVAE	X							X			
Performance RNN	X				X	X	X	X×4		X	
RBM_C									X		X
Rhythm	X								X		
RL-Tuner	X							X			
RNN-RBM									X		X
UnitSelection				X			X	X		X	
VRAE											
VRASH								X×3			
WaveNet					X	X	X	X			

Table 7.11 System × Representation (2/2)

	Architecture										Strategy							
	Fd	Ae	Va	RB	RN	Cv	Cn	GA	RL	Cp	SF	DF	Sa	IF	IM	Re	US	Cp
System																		
Anticipation-RNN					$X^2 \times 2$		X			X			X	X				X
AST	X					X				X	X				X			X
BachBot					X^3								X	X				X
Bi-Axial LSTM					$X^2 \times 2$					X			X	X				X
Blues_C					X									X				
Blues_{MC}					X									X				
Celtic					X^3								X	X				X
CONCERT					X								X	X				X
C-RBM				X		X				X			X		X			X
C-RNN-GAN					$X^2 \times 2$			X		X			X	X				X
deepAutoController		X^2								X		X						
DeepBach	X×2				X×2					X			X					
DeepHear_C		X^4								X		X			X			X
DeepHear_M		X^4								X		X						
DeepJ					$X^2 \times 2$		X			X			X	X				X
GLSR-VAE		X	X		X^2					X		X	X	X				X
Hexahedria					X^{2+2}					X			X	X				X
MidiNet	X					X	X	X		X	X		X					X
MiniBach	X^2										X							
MusicVAE		X	X		X×2+2					X		X	X	X				X
Performance RNN					X								X	X				X
RBM_C				X									X					
Rhythm	X				X^2		X			X	X		X	X				X
RL-Tuner					X×2				X	X			X	X		X		X
RNN-RBM				X	X					X			X	X				X
UnitSelection		X^2			X×2					X				X			X	X
VRAE		X	X		X×2					X		X	X	X				X
VRASH		X	X		$X^4 \times 2$		X			X		X	X	X				X
WaveNet	X					X	X			X			X	X				X

Table 7.12 System × Architecture & Strategy

	Challenge										
	EN	LV	CV	Es	MH	Co	St	Or	Ic	It	Ad	Ey
System												
Anticipation-RNN	X	X	X			X			X			
AST						X						
BachBot	X	X	X		X				X			X
Bi-Axial LSTM	X	X	X		X				X			
Blues_C	X	X							X			
Blues_{MC}	X	X			X				X			
Celtic	X	X	X						X			
CONCERT	X	X	X						X			
C-RBM	X		X		X	X	X					
C-RNN-GAN	X	X	X						X			
deepAutoController	X					X				X		
DeepBach			X		X				X	X		
DeepHear_C	X				X							
DeepHear_M	X											
DeepJ	X	X	X	X		X			X			
GLSR-VAE	X	X	X			X			X			
Hexahedria	X	X	X		X				X			
MidiNet	X		X		X	X		X				
MiniBach					X							
MusicVAE	X	X	X			X	X		X			
Performance RNN	X	X	X	X		X			X			
RBM_C	X		X									
Rhythm	X	X	X	X		X			X			
RL-Tuner	X	X	X			X	X		X			
RNN-RBM	X	X	X		X				X			
UnitSelection	X	X	X			X	X		X			
VRAE	X	X	X						X			
VRASH	X	X	X						X			
WaveNet	X		X	X		X			X			

Table 7.13 System $\times$ Challenge

Note that, when considering the analysis regarding the challenges in Table 7.13, we have to keep in mind that

- the limitations and challenges are not of equal importance and difficulty; and
- the majority of the systems may be further extended in order to better address some of the challenges[3].

That said, we can see the emergence of some divide between: 1) systems using a global temporal representation and 2) systems using a time sliced representation, depending on the following requirements: *ex nihilo* generation and length variability. This will be further discussed in Section 8.1.

7.3 Correlation Analysis

The last series of tables analyse some correlations between the dimensions:

- *representation* with respect to the *objective* in Table 7.14;
- *architecture* and *strategy* with respect to the *objective* in Table 7.15;
- *architecture* and *strategy* with respect to the *representation* in Table 7.16;
- *strategy* with respect to the *architecture* in Table 7.17; and

[3] For instance, the RL-Tuner system has the potential for addressing interactivity and adaptability challenges, although, to our knowledge, not yet experimented.

- *architecture* and *strategy* with respect to the *challenge* in Table 7.18.

The occurrence of an "X" at the crossing of a given line (a given type for the first dimension) and a given column (a given type for the second dimension) means that there is (at least) a system which matches both types (for instance, is based on a particular architecture and follows a particular strategy). But, note that we base this analysis on how each system *is* presented in the literature referenced, and not as a potential compatibility analysis. In other words, the absence of an "X" means that there is no system reviewed in the book matching both types, but it does not mean that the two types are incompatible or that such a system may not be constructed. Therefore, we add an additional "x" notation to represent an *a priori* potential compatibility between types and furthermore a possible direction to be explored.

	Objective												
	Type								Dest./Use			Mode	
	Au	Me	Po	CS	MV	Dr	Co	ST	AR	SP	HI	AG	IG
Representation													
Audio													
Waveform	X							x	X			X	x
Spectrum	X							X	X			X	x
Symbolic													
Concept													
Note		X	X	X	X		X	X		X	X	X	X
Rest		X	X	X	X	X	X	X		X	X	X	X
Chord			X	X	X			X		X	X	X	x
Rhythm (Meter & Beats)		X	X	X	X	X	X	X		X	X	X	X
Drums			X		X	X		x		X	X	X	x
Format													
MIDI		X	X	X	X	X	X	x		X	X	X	x
Piano roll		X	X	X	X	X	X	X		X	X	X	X
Text		X	X	X		X	X	x		X	X	X	x
Temporal Scope													
Global	X	X	X	X	X	X	X	X	X	X	X	X	X
Time step	X	X	X	X	X	X	X	x	X	X	X	X	x
Meta-Data													
Note hold or ending		X	X	X	X	X	X	x		X	X	X	X
No enharmony (Note denotation)		X	x	x	X		X	x		X	X	X	X
Fermata		X	X	x	X		X	x		X	X	X	X
Feature extraction	x	X	x	x	x	x	x	x	X	X	X	X	x
Expressiveness													
Tempo	X	x	X	x	x	x	x	X	X	X	X	X	x
Dynamics	X	x	X	x	x	x	x	X	X	X	X	X	x
Encoding													
Value encoding	X	x	x	x	x	x	x	X	X	X	X	X	X
One-hot encoding	X	X		X	X^n	x^n	X^n	x	X	X	X	X	X
Many-hot encoding			X	x		X	x	X		X	X	X	x
Dataset													
Transposition		X	X	X	X		X	x		X	X	X	X
Alignment		X	X	x	x		x	x		X	X	X	x

Table 7.14 Representation × Objective

	Objective												
	Type								Dest./Use			Mode	
	Au	Me	Po	CS	MV	Dr	Co	ST	AR	SP	HI	AG	IG
Architecture													
Feedforward	X	X	x	x	X	x	X	X	X	X	X	X	X
Autoencoder	X	X	x	x	X	X	X		X	X	X	X	X
Variational	x	X	X	X	X	X			x	X	X	X	x
Restricted Boltzmann machine	x	x	X	X	x	x			x	X	X	X	x
Recurrent (RNN)	x	X	X	X	X	X		X	x	X	X	X	x
Convolutional	X	X	X	X	X	X	X		X	X	X	X	x
Conditioning	X	X	X	X	X	X	X		X	X	X	X	x
Generative adversarial networks	x	X	X	X	X	X			x	X	X	X	x
Reinforcement learning (RL)	x	X	X	X	X	X				X	X	X	x
Strategy													
Single-step feedforward	X	X	x	x	X	x	X	X	X	X	X	X	x
Decoder feedforward	X	X	x	x	x	x	x		X	X	X	X	X
Sampling	X	x	x	x	X	x	X		x	X	X	X	X
Iterative feedforward	X	X	X	X	X	X				X	X	X	x
Input manipulation	x	x	X	x	x	x	X	X	x	X	X	X	x
Reinforcement		X	x	x	x	x	x			X	X	X	x
Unit selection	x	X	x	x	x	x	x		x	X	X	X	x

Table 7.15 Architecture & Strategy $\times$ Objective

	Representation																								
	Audio		Concept					Format			TmpS		Meta-Data				Expr.		Encoding			DSet	
	Wa	Sp	No	Re	Ch	Rh	Dr	MI	Pi	Te	Gl	TS	NH	NE	Fe	FE	To	Dy	VE	OH	MH	Tr	Al
Objective																									
Type																									
Au	X	X	x	x	x	x	x	x	x	x	X	x				x	x	x	X	X			
Me			X	X		X		X	X	X	X	X	X	X	X	X	x	x	X	X		X	X
Po			X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X		X	X	X
CS			X	X	X	X		x	X	x	X	X	X	X	X	x	x	x	X	X	X	X	x
MV			X	X	X	X	X	X	X	X	X	X	X	X	X	x	x	x	X	X^n	X	X	x
Dr				X		X	X	X	X	X	X	X	X		X	x	x	x	x	X	X		
Co			X	X	x	X		x	X	X	X	x	X	X	X	x	x	x	x	X	x	X	x
ST			X	X	x	X	x	x	X	x	X	x	X	x	x	x	X	X	X	X	x	X	x
Destination & Use																							
AR	X	X	x	x	x	x	x	x	x	x	X					x	x	x	X	X	x		
SP			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
HI			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Mode																							
AG	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
IG	x	X	X	X	x	X	x	x	X	x	X	x	X	X	X	x	X	X	X	X	x	X	x
Architecture																							
Fd	X	X	X	X	X	X	X	x	X	x	X		X	X	X	x	X	X	X	X	X	X	x
Ae	x	X	X	X	x	X	X	X	X	X	X		X	X	X	X	X	X	X	X	x	X	x
Va	x	x	X	X	x	X	X	X	X	X	X		X	X	X	x	X	X	x	X	x	X	x
RB	x	x	X	X	X	X	x	x	X	x	X		X	x	X	x	X	X	x		X	X	X
RN			X	X	X	X	X	X	X	X		X	X	X	X	x	X	X	X	X	X	X	X
Cv	X	X	X	X	X	X	x	x	X	x	X	x	X	x	X	x	X	X	X	X	x	X	x
Cn	X	x	X	X	X	X	X	x	X	x	X	X	X	X	X	X	X	X	X	X	X	X	x
GA	x	x	X	X	X	x	x	X	X	x	X		X	x	X	x	X	X	X	X	x	X	x
RL			X	X	x	x	x	x	X	x		X	X	x	X	x	X	X	x	X	x	X	x
Strategy																							
SF	X	x	X	X	X	X	X	X	X	X	X		X	X	x	x	X	X	x	X	X	X	X
DF	x	X	X	X	x	X	X	X	X	X	X		X	X	X	x	X	X	X	X	x	X	x
Sa	X	x	X	X	X	X	X	X	X	X	X	X	X	X	X	x	X	X	X	X	X	X	X
IF	X		X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
IM	x	X	X	X	x	x	X	X	X	X	X		X	x	x	x	X	X	X	X	x	X	x
Re			X	X	x	x	X	X	X	X		X	X	x	X	x	X	X	x	X	x	X	x
US	x	x	X	X	x	x	X	X	X	X		X	X	x	x	X	X	X	X	X	x	X	x

Table 7.16 Architecture & Strategy × Representation

	Architecture								
	Fd	Ae	Va	RB	RN	Cv	Cn	GA	RL
Strategy									
Single-step feedforward	X					X	X	X	
Decoder feedforward		X	X			x	X	x	
Sampling	X	X	X	X	X	X	X	X	X
Iterative feedforward			X		X	X	X	X	
Input manipulation	X	X	x	X	x	X	x	x	
Reinforcement					X		x		X
Unit selection		X			X		x		

Table 7.17 Strategy × Architecture

	Challenge											
	EN	LV	CV	Es	MH	Co	St	Or	Ic	It	Ad	Ey
Architecture												
Feedforward						X				X		
Autoencoder	X		X			X				X		
Variational	X		X			X				X		
Restricted Boltzmann machine (RBM)	X		X			X						
Recurrent (RNN)	X	X	X		X	X			X	X		
Convolutional												
Conditioning					X	X		X				
Generative adversarial networks (GAN)	X		X			x		X				
Reinforcement learning (RL)	X	X	X			X		x	X	X	X	
Strategy												
Single-step feedforward												
Decoder feedforward	X					X						
Sampling	X		X			X		x	X	X		
Iterative feedforward	X	X				X	X		X	x		
Input manipulation	X		X		X	X	X		x	x		
Reinforcement	X	X	X			X	X		X	x	X	
Unit selection	X	X	X			X	X		X	x		

Table 7.18 Architecture & Strategy $\times$ Challenge

The analysis of the correlation tables allows us to draw a few first conclusions:

- audio representation is specific and dedicated to audio generation (Table 7.16);
- divide between global temporal representation and time slice representation – they have a strong coupling to feedforward and recurrent architectures, respectively (Tables 7.16 and 7.17);
- recurrent networks may not be used to generate accompaniment, except if encapsulated within autoencoders (Table 7.15).

Note that in Table 7.15, a feedforward architecture is usually correlated to an accompaniment objective, i.e. counterpoint accompaniment (Co), for the reason discussed in Section 6.3. However, the DeepBach architecture is a notable exception. Moreover, thanks to its *sampling only strategy*, it can generate an accompaniment as well as a complete chorale (see the discussion in Section 6.16.1).

Last, note that in Table 7.18, the columns corresponding to the expressiveness and explainability challenges are left empty. This is because these challenges are not to be solved with the lone choice of an architecture or a strategy.

We do not comment further these tables in this book. We consider them as a first version of analysis tools related to our proposed conceptual framework which could be tried out and improved, for investigating current as well as future systems.

Chapter 8
Discussion

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

8.1 Global versus Time Sliced

As we have seen in Sections 6.7 and 6.13.1, one important decision is to choose between

- a global representation, including all time steps – typically coupled with a feedforward or an autoencoder architecture; and
- a time-sliced representation, representing a single time step – typically coupled with a recurrent neural network (RNN) architecture.

The pros and cons are

- global representation
  - \+ allows arbitrary output, e.g., for the objective of generating some accompaniment through the single-step feedforward strategy on a feedforward architecture, as, for example, in the MiniBach system (Section 6.1.2);
  - \+ supports incremental instantiation (via sampling), as, for example, by the DeepBach system (Section 6.13.2);
  - – does not allow variable-length generation;
  - – does not allow seed-based generation for a feedforward architecture;
  - \+ allows seed-based generation through the decoder feedforward strategy on an autoencoder architecture, as, for example, in the DeepHear system (Section 6.3.1.1);
- time-sliced representation
  - –/+ does not allow arbitrary output, e.g., for the objective of generating some accompaniment[1];
  - +/– supports incremental instantiation, but only forward in time, through the iterative feedforward strategy on a recurrent network architecture;
  - \+ allows variable length generation, as, for example, in the CONCERT system (Section 6.5.1.1).

Actually some attempt at combining "the best of both worlds" seems to lie in using an RNN Encoder-Decoder architecture, as

- generation is iterative, which allows variable length content generation,
- while allowing arbitrary output generation, as the output sequence may have an arbitrary length and content[2], and

[1] Except for an RNN Encoder-Decoder (sequence-to-sequence) architecture (Section 5.16.3), which allows input and output sequences to be of different nature and length.

[2] As is the case for translation tasks.

- allowing the manipulation of a global temporal scope representation (the latent variables).

Moreover, in a variational version, such as, for example, VRAE, the latent space could be explored in a disciplined and meaningful manner, as, for example, in the GLSR-VAE and MusicVAE systems (Sections 6.9.2.3 and 6.11.1).

8.2 Convolution versus Recurrent

As noted in Section 5.12, convolutional architectures, while prevalent for image applications, are more seldom used than recurrent neural network (RNN) architectures in music applications. The few examples of nonrecurrent architectures using convolution in time that we have encountered and analyzed are

- WaveNet, a convolutional feedforward architecture for audio (Section 6.9.3.2);
- MidiNet, a GAN architecture encapsulating conditional convolutional architectures (Section 6.9.3.3)[3]; and
- C-RBM, a convolutional RBM (Section 6.9.5.1).

Let us try to list and analyze the relative pros of cons of using recurrent architectures or convolutional architectures to model time correlations:

- recurrent networks are popular and accurate, especially since the arrival of LSTM architectures;
- convolution should be used only in time 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 (a counterexample is the system analyzed in Section 6.8.2);
- convolutional networks are typically faster to train and easier to parallelize than recurrent networks [177];
- sharing weights by convolutions only applies to a small number of temporal neighboring members of the input, in contrast to a recurrent network that shares parameters in a deep way, for all time steps (see Section 5.12);
- the authors of WaveNet argue that the layers of dilated convolutions allow the receptive field to grow longer in a much cheaper way than using LSTM units;
- the authors of MidiNet argue that using a conditioning strategy for a convolutional architecture allows the incorporation of information from previous measures into intermediate layers and therefore considers history as a recurrent network would do.

A potential output of this initial comparative analysis is that, as there are many systems using nonrecurrent architectures (like feedforward networks or autoencoders), it may be interesting to study whether their extension into a convolutional architecture on the time dimension could bring some effective gain, in terms of efficiency and/or accuracy.

8.3 Style Transfer and 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 one context and use it in other contexts. Various cases may be considered, e.g., similar source and target domains, similar task, etc. This new research subdomain, named *transfer learning*, is about methodologies and techniques for the transfer of what has been learnt [58, Section 15.2].

We have not addressed this important issue in our analysis because it has not yet been specifically addressed for music generation, although we think that it will become an area of investigation. Meanwhile, an example, although still simplistic, is the way the DeepHear architecture and what it has learnt is transfered from the objective of generating a melody to the objective of generating a counterpoint (see Section 6.9.4.1). Another example is the way the

[3] A comparison between MidiNet and C-RNN-GAN, both using GANs but encapsulating a convolutional network versus a recurrent network, is also interesting.

DeepBach architecture allows to adapt the objective from *ex nihilo* chorale generation to multi-voice accompaniment (see Section 6.13.2).

Last, let us remember that style transfer is a very specific case of transfer learning in terms of objective and techniques (see Section 6.9.4.5)

8.4 Cooperation

All the systems surveyed are basically lone systems (although the architecture may be compound). A more cooperative approach is natural for handling complexity, heterogeneity, scalability and openness, as, for example, pioneered by multi-agent systems [186].

An example is the system proposed by Hutchings and McCormack [84]. It is composed of two agents:

- a *harmony* agent, based on an RNN (LSTM) architecture, in charge of the progression of chords; and
- a *melody* agent, based on a rule-based system, in charge of the melody.

The two agents work in a cooperative way and alternate between leading and accompanying roles (inspired by, for example, the way musicians function in a Jazz band). The authors relate the interesting dynamics between the two agents and also an intereresting balance between harmonic creativity and harmonic consistency[4]. This approach appears to be an interesting direction to pursue and extend with more agents and roles.

8.5 Evaluation

Evaluation of a system generating music mostly comprises qualitative evaluation of examples of generated music. For many experiments, evaluation is only preliminary, and in many cases, only conducted by the designers themselves. There are of course some exceptions, with more systematic and external evaluations (by a more or less expert public). When the corpus is very precise, e.g., in the case of J. S. Bach's chorales for the BachBot or the DeepBach systems (Sections 6.16.1 and 6.13.2), a Turing test may be conducted: a piece of music being presented to the public who has to decide if it is one of the original pieces or music generated by a computer. But this methodology is more limited when the objective is not to generate music belonging to a relatively narrow style (and corpus), as in the case of J. S. Bach chorales, but to generate more creative music. In this case, a Turing test may be counter productive. The preliminary experiments and evaluation protocol for the CAN system (Section 6.12.2) are, in that respect, an interesting direction.

Last, about the possibility of looking for more systematic objective criteria for evaluation, see, for example, the analysis by Theis *et al.* for the case of image generation [169]. The authors state that an evaluation of image generative models is *multicriteria* via different possible metrics, such as log-likelihood, Parzen window estimates, or qualitative visual fidelity, and that a good result with respect to one criterion does not necessarily imply a good result with respect to another criterion.

8.6 Specialization

A general issue is the hyper-specialization of systems designed for a specific objective and/or a specific type of corpus. This is witnessed by the diversity of the architectures and approaches surveyed. Note that this is a known issue for Artificial Intelligence (AI) research in general. There is some tendency towards hyper-specialized systems solving specific problems, especially in the case of competitions organized by conferences or other institutions, with the risk of loosing the initial objective of a general problem solving framework[5].

[4] On this issue, see Section 6.12.

[5] An interesting counterexample is the ongoing research and competition about general game playing [54].

Meanwhile, the general objective of generating interesting musical content is complex and still an opened issue. Thus, we need to work both on general approaches for general problems and specific approaches for specific sub-problems, as well as top-down and bottom-up approaches, while not losing interest in how to interpret, generalize and reuse advances and lessons learnt. We hope that the survey and analysis conducted in this book will contribute to this research agenda.

Chapter 9
Conclusion

The use of deep learning techniques for the creation of musical content, and more generally speaking creative artistic content, is nowadays getting increased attention. This book presented a survey and an analysis of various strategies and techniques for using deep learning to generate musical content. We have proposed a multi-criteria analysis based on five dimensions: objective, representation, architecture, challenges and strategy. We have analyzed and compared various systems and experiments proposed by various researchers in the literature. We hope that the conceptual framework provided in this book will help in understanding the issues and in comparing various approaches for using deep learning for music generation, and therefore contribute to this research agenda.

Acknowledgements

This research was partly 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. The authors thank CNRS, LIP6, Sony CSL and Spotify CTRL for their support and research environments. Jean-Pierre Briot also thanks CAPES, PUC-Rio and UNIRIO additional support.

References

1. Moray Allan and Christopher K. I. Williams. Harmonising chorales by probabilistic inference. *Advances in Neural Information Processing Systems*, 17:25–32, 2005.
2. Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer assisted composition at IRCAM: From PatchWork to Openmusic. *Computer Music Journal (CMJ)*, 23(3):59–72, September 1999.
3. Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep?, October 2014. arXiv:1312.6184v7.
4. Johann Sebastian Bach. *389 Chorales (Choral-Gesange)*. Alfred Publishing Company, 1985.
5. David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert Müller. How to explain individual classification decisions. *Journal of Machine Learning Research (JMLR)*, (11):1803–1831, June 2010.
6. Gabriele Barbieri, François Pachet, Pierre Roy, and Mirko Degli Esposti. Markov constraints for generating lyrics with style. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 115–120, Montpellier, France, August 2012.
7. Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 35(8):1798–1828, August 2013.
8. Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks, July 2017. arXiv:1707.05776v1.
9. Diane Bouchacourt, Emily Denton, Tejas Kulkarni, Honglak Lee, Siddharth Narayanaswamy, David Pfau, and Josh Tenenbaum (Eds.). NIPS 2017 Workshop on Learning Disentangled Representations: from Perception to Control, December 2017. https://sites.google.com/view/disentanglenips2017.
10. 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.
11. Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Chapter 14th – Modeling and generating sequences of polyphonic music with the RNN-RBM. In *Deep Learning Tutorial – Release 0.1*, pages 149–158. LISA lab, University of Montréal, September 2015. http://deeplearning.net/tutorial/deeplearning.pdf.
12. Mason Bretan, Gil Weinberg, and Larry Heck. A unit selection methodology for music generation using deep neural networks. In Ashok Goel, Anna Jordanous, and Alison Pease, editors, *Proceedings of the 8th International Conference on Computational Creativity (ICCC 2017)*, pages 72–79, Atlanta, GA, USA, June 2017.
13. Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation – A survey, September 2017. arXiv:1709.01620v1.
14. Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation, 2019.
15. Jean-Pierre Briot and François Pachet. Music generation by deep learning – Challenges and directions. *Neural Computing and Applications (NCAA)*, October 2018.
16. Davide Castelvecchi. The black box of AI. *Nature*, 538:20–23, October 2016.
17. E. Colin Cherry. Some experiments on the recognition of speech, with one and two ears. *The Journal of the Acoustical Society of America*, 25(5):975–979, September 1953.
18. 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.
19. Keunwoo Choi, György Fazekas, Kyunghyun Cho, and Mark Sandler. A tutorial on deep learning for music information retrieval, September 2017. arXiv:1709.04396v1.
20. Keunwoo Choi, György Fazekas, and Mark Sandler. Text-based LSTM networks for automatic music composition. In *1st Conference on Computer Simulation of Musical Creativity (CSMC 16)*, Huddersfield, U.K., June 2016.
21. Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks, January 2015. arXiv:1412.0233v3.
22. Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, December 2014. arXiv:1412.3555v1.
23. Yu-An Chung, Chao-Chung Wu, Chia-Hao Shen, Hung-Yi Lee, and Lin-Shan Lee. Audio Word2Vec: Unsupervised learning of audio segment representations using sequence-to-sequence autoencoder, June 2016. arXiv:1603.00982v4.
24. David Cope. *The Algorithmic Composer*. A-R Editions, 2000.
25. David Cope. *Computer Models of Musical Creativity*. MIT Press, 2005.
26. 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/.
27. Márcio Dahia, Hugo Santana, Ernesto Trajano, Carlos Sandroni, and Geber Ramalho. Generating rhythmic accompaniment for guitar: the cyber-João case study. In *Proceedings of the IX Brazilian Symposium on Computer Music (SBCM 2003)*, pages 7–13, Campinas, SP, Brazil, August 2003.
28. Shuqi Dai, Zheng Zhang, and Gus Guangyu Xia. Music style transfer issues: A position paper, March 2018. arXiv:1803.06841v1.
29. Ernesto Trajano de Lima and Geber Ramalho. On rhythmic pattern extraction in bossa nova music. In *Proceedings of the 9th International Conference on Music Information Retrieval (ISMIR 2008)*, pages 641–646, Philadelphia, PA, USA, September 2008.
30. Roger T. Dean and Alex McLean, editors. *The Oxford Handbook of Algorithmic Music*. Oxford Handbooks. Oxford University Press, 2018.

31. Jean-Marc Deltorn. Deep creations: Intellectual property and the automata. *Frontiers in Digital Humanities*, 4, February 2017. Article 3.
32. Guillaume Desjardins, Aaron Courville, and Yoshua Bengio. Disentangling factors of variation via generative entangling, October 2012. arXiv:1210.5474v1.
33. Carl Doersch. Tutorial on variational autoencoders, August 2016. arXiv:1606.05908v2.
34. Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM (CACM)*, 55(10):78–87, October 2012.
35. Kenji Doya and Eiji Uchibe. The Cyber Rodent project: Exploration of adaptive mechanisms for self-preservation and self-reproduction. *Adaptive Behavior*, 13(2):149–160, 2005.
36. Kemal Ebcioğlu. An expert system for harmonizing four-part chorales. *Computer Music Journal (CMJ)*, 12(3):43–51, Autumn 1988.
37. Douglas Eck and Jürgen Schmidhuber. A first look at music composition using LSTM recurrent neural networks. Technical report, IDSIA/USI-SUPSI, Manno, Switzerland, 2002. No. IDSIA-07-02.
38. Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks, May 2016. arXiv:1512.03965v4.
39. 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.
40. 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 (JMLR)*, (11):625–660, 2010.
41. Douglas Eck *et al.* Magenta Project, Accessed on 20/06/2017. https://magenta.tensorflow.org.
42. François Pachet *et al.* Flow Machines – Artificial Intelligence for the future of music, 2012. http://www.flow-machines.com.
43. Otto Fabius and Joost R. van Amersfoort. Variational recurrent auto-encoders, June 2015. arXiv:1412.6581v6.
44. Jose David Fernández and Francisco Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research (JAIR)*, (48):513–582, 2013.
45. Rebecca Fiebrink and Baptiste Caramiaux. The machine learning algorithm as creative musical tool, November 2016. arXiv:1611.00379v1.
46. Davis Foote, Daylen Yang, and Mostafa Rohaninejad. Audio style transfer – Do androids dream of electric beats?, December 2016. https://audiostyletransfer.wordpress.com.
47. Spotify for Artists. Innovating for writers and artists, Accessed on 06/09/2017. https://artists.spotify.com/blog/innovating-for-writers-and-artists.
48. The International Association for Computational Creativity. International Conferences on Computational Creativity (ICCC), Accessed on 17/05/2018. http://computationalcreativity.net/home/conferences/.
49. Eric Foxley. Nottingham Database, Accessed on 12/03/2018. https://ifdo.ca/˜seymour/nottingham/nottingham.html.
50. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
51. Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, September 2015. arXiv:1508.06576v2.
52. Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423. IEEE, June 2016.
53. Robert Gauldin. *A Practical Approach to Eighteenth-Century Counterpoint*. Waveland Press, 1988.
54. Michael Genesereth and Yngvi Björnsson. The international general game playing competition. *AI Magazine*, pages 107–111, Summer 2013.
55. Felix A. Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.
56. 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 Nature, 2014.
57. Michael Good. MusicXML for notation and analysis. In Walter B. Hewlett and Eleanor Selfridge-Field, editors, *The Virtual Score: Representation, Retrieval, Restoration*, pages 113–124. MIT Press, 2001.
58. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
59. 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.
60. Alex Graves. Generating sequences with recurrent neural networks, June 2014. arXiv:1308.0850v5.
61. Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines, December 2014. arXiv:1410.5401v2.
62. Gaëtan Hadjeres. *Interactive Deep Generative Models for Symbolic Music*. PhD thesis, Ecole Doctorale EDITE, Sorbonne Université, Paris, France, June 2018.
63. Gaëtan Hadjeres and Frank Nielsen. Interactive music generation with positional constraints using Anticipation-RNN, September 2017. arXiv:1709.06404v1.
64. Gaëtan Hadjeres, Frank Nielsen, and François Pachet. GLSR-VAE: Geodesic latent space regularization for variational autoencoder architectures, July 2017. arXiv:1707.04588v1.
65. Gaëtan Hadjeres, François Pachet, and Frank Nielsen. DeepBach: a steerable model for Bach chorales generation, June 2017. arXiv:1612.01010v2.
66. Jeff Hao. Hao staff piano roll sheet music, Accessed on 19/03/2017. http://haostaff.com/store/index.php?main_page=article.
67. Dominik Härnel. ChordNet: Learning and producing voice leading with neural networks and dynamic programming. *Journal of New Music Research (JNMR)*, 33(4):387–397, 2004.

68. Simon Haykin and Zhe Chen. The cocktail party problem. *Neural Computation*, 17(9):1875–1902, 2005.
69. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, December 2015. arXiv:1512.03385v1.
70. Dorien Herremans and Ching-Hua Chuan. Deep Learning for Music – Workshop Proceedings, May 2017. http://dorienherremans.com/dlm2017/.
71. Walter Hewlett, Frances Bennion, Edmund Correia, and Steve Rasmussen. MuseData – an electronic library of classical music scores, Accessed on 12/03/2018. http://www.musedata.org.
72. Lejaren A. Hiller and Leonard M. Isaacson. *Experimental Music: Composition with an Electronic Computer*. McGraw-Hill, 1959.
73. Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, August 2002.
74. 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.
75. Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
76. 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*, pages 282–317. MIT Press, Cambridge, MA, USA, 1986.
77. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
78. Douglas Hofstadter. Staring Emmy straight in the eye–and doing my best not to flinch. In David Cope, editor, *Virtual Music – Computer Synthesis of Musical Style*, pages 33–82. MIT Press, 2001.
79. Hooktheory. Theorytabs, Accessed on 26/07/2017. https://www.hooktheory.com/theorytab.
80. Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.
81. Allen Huang and Raymond Wu. Deep learning for music, June 2016. arXiv:1606.04930v1.
82. 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.
83. Eric J. Humphrey, Juan P. Bello, and Yann LeCun. Feature learning and deep architectures: New directions for music informatics. *Journal of Intelligent Information Systems (JIIS)*, 41(3):461–481, 2013.
84. 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 19–21, 2017, Proceedings*, number 10198 in Theoretical Computer Science and General Issues, pages 114–127. Springer Nature, 2017.
85. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, March 2015. http://arxiv.org/abs/1502.03167v3.
86. Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning, November 2016. arXiv:1611.02796.
87. Daniel Johnson. Composing music with recurrent neural networks, August 2015. http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/.
88. Daniel D. Johnson. Generating polyphonic music using tied parallel networks. 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 19–21, 2017, Proceedings*, number 10198 in Theoretical Computer Science and General Issues, pages 128–143. Springer Nature, 2017.
89. Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research (JAIR)*, (4):237–285, 1996.
90. Ujjwal Karn. An intuitive explanation of convolutional neural networks, August 2016. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.
91. Jeremy Keith. The Session, Accessed on 21/12/2016. https://thesession.org.
92. Pieter-Jan Kindermans, Sara Hooker, Julius Adebayo, Maximilian Alber, Kristof T. Schütt, Sven Dähne, Dumitru Erhan, and Been Kim. The (un)reliability of saliency methods. November 2017. arXiv:1711.00867v1.
93. Pieter-Jan Kindermans, Kristof T. Schütt, Maximilian Alber, Klaus-Robert Müller, Dumitru Erhan, Been Kim, and Sven Dähne. Learning how to explain neural networks: PatternNet and PatternAttribution, 2017. arXiv:1705.05598v2.
94. Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes, May 2014. arXiv:1312.6114v10.
95. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, volume 1 of *NIPS 2012*, pages 1097–1105, Lake Tahoe, NV, USA, 2012. Curran Associates Inc.
96. Bernd Krueger. Classical Piano Midi Page, Accessed on 12/03/2018. http://piano-midi.de/.
97. 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/.
98. 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.
99. 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.

100. Stefan Lattner, Maarten Grachten, and Gerhard Widmer. Imposing higher-level structure in polyphonic music generation using convolutional restricted Boltzmann machines and constraints. *Journal of Creative Music Systems (JCMS)*, 2(2), March 2018.
101. 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.
102. Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time-series. In Michael A. Arbib, editor, *The handbook of brain theory and neural networks*, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.
103. Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.
104. Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/.
105. Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan Kaufmann, 1990.
106. Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)*, pages 609–616, Montréal, QC, Canada, June 2009. ACM.
107. Andre Lemme, René Felix Reinhart, and Jochen Jakob Steil. Online learning and generalization of parts-based image representations by non-negative sparse autoencoders. *Neural Networks*, 33:194–203, September 2012.
108. Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Convolutional neural networks (CNNs / ConvNets) – CS231n Convolutional neural networks for visual recognition Lecture Notes, Winter 2016. http://cs231n.github.io/convolutional-networks/#conv.
109. Feynman Liang. BachBot, 2016. `https://github.com/feynmanliang/bachbot`.
110. 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.
111. 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.
112. 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.
113. 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 Nature, 2017.
114. Iman Malik and Carl Henrik Ek. Neural translation of musical style, August 2017. arXiv:1708.03535v1.
115. Stéphane Mallat. GANs vs VAEs, September 2018. Personal communication.
116. Huanru Henry Mao, Taylor Shin, and Garrison W. Cottrell. DeepJ: Style-specific music generation, January 2018. arXiv:1801.00887v1.
117. John A. Maurer. A brief history of algorithmic composition, March 1999. https://ccrma.stanford.edu/~blackrse/algorithm.html.
118. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, September 2013. arXiv:1301.3781v3.
119. Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.
120. Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
121. MIDI Manufacturers Association (MMA). MIDI Specifications, Accessed on 14/04/2017. https://www.midi.org/specifications.
122. 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.
123. Olof Mogren. C-RNN-GAN: Continuous recurrent neural networks with adversarial training, November 2016. arXiv:1611.09904v1.
124. Grégoire Montavon, Sebastian Lapuschkin, Alexander Binder, Wojciech Samek, and Klaus-Robert Müller. Explaining nonlinear classification decisions with deep Taylor decomposition. *Pattern Recognition*, (65):211–222, 2017.
125. Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Deep Dream, 2015. https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.
126. Dan Morris, Ian Simon, and Sumit Basu. Exposing parameters of a trained dynamic model for interactive music creation. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 784–791, Chicago, IL, USA, July 2008. AAAI Press.
127. 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.
128. Kevin P. Murphy. *Machine Learning: a Probabilistic Perspective*. MIT Press, 2012.
129. Andrew Ng. Sparse autoencoder – CS294A/CS294W Lecture notes – Deep Learning and Unsupervised Feature Learning Course, Winter 2011. https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf.
130. Andrew Ng. CS229 Lecture notes – Machine Learning Course – Part I Linear Regression, Autumn 2016. http://cs229.stanford.edu/notes/cs229-notes1.pdf.
131. Andrew Ng. CS229 Lecture notes – Machine Learning Course – Part IV Generative Learning algorithms, Autumn 2016. http://cs229.stanford.edu/notes/cs229-notes2.pdf.

132. Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer Nature, 2009.
133. Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, July 1994.
134. François Pachet. Beyond the cybernetic jam fantasy: The Continuator. *IEEE Computer Graphics and Applications (CG&A)*, 4(1):31–35, January/February 2004. Special issue on Emerging Technologies.
135. 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, PA, Brazil, November 2013.
136. 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), Suzhou, China, October 23–27, 2017*, pages 167–173, 2017.
137. François Pachet and Pierre Roy. Markov constraints: Steerable generation of Markov sequences. *Constraints*, 16(2):148–172, 2011.
138. François Pachet and Pierre Roy. Imitative leadsheet generation with user constraints. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 – Proceedings of the 21st European Conference on Artificial Intelligence*, Frontiers in Artificial Intelligence and Applications, pages 1077–1078. IOS Press, 2014.
139. François Pachet, Pierre Roy, and Gabriele Barbieri. Finite-length markov processes with constraints. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 635–642, Barcelona, Spain, July 2011.
140. Alexandre Papadopoulos, François Pachet, and Pierre Roy. Generating non-plagiaristic Markov sequences with max order sampling. In Mirko Degli Esposti, Eduardo G. Altmann, and François Pachet, editors, *Creativity and Universality in Language*, Lecture Notes in Morphogenesis. Springer Nature, 2016.
141. 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 Nature, 2016.
142. Aniruddha Parvat, Jai Chavan, and Siddhesh Kadam. A survey of deep-learning frameworks. In *Proceedings of the International Conference on Inventive Systems and Control (ICISC 2017)*, Coimbatore, India, January 2017.
143. Mathieu Ramona, Giordano Cabral, and François Pachet. Capturing a musician's groove: Generation of realistic accompaniments from single song recordings. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015) – Demos Track*, pages 4140–4141, Buenos Aires, Argentina, July 2015. AAAI Press / IJCAI.
144. Bharath Ramsundar and Reza Bosagh Zadeh. *TensorFlow for Deep Learning*. O'Reilly Media, March 2018.
145. Dario Ringach and Robert Shapley. Reverse correlation in neurophysiology. *Cognitive Science*, 28:147–166, 2004.
146. Curtis Roads. *The Computer Music Tutorial*. MIT Press, 1996.
147. Adam Roberts. MusicVae supplementary materials, Accessed on 27/04/2018. g.co/magenta/musicvae-samples.
148. Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. ACM, Montréal, QC, Canada, July 2018.
149. Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music, June 2018. arXiv:1803.05428v2.
150. Adam Roberts, Jesse Engel, Colin Raffel, Ian Simon, and Curtis Hawthorne. MusicVAE: Creating a palette for musical scores with machine learning, March 2018. https://magenta.tensorflow.org/music-vae.
151. Frank Rosenblatt. The Perceptron – A perceiving and recognizing automaton. Technical report, Cornell Aeronautical Laboratory, Ithaca, NY, USA, 1957. Report 85-460-1.
152. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.
153. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
154. Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs, June 2016. arXiv:1606.03498v1.
155. Andy M. Sarroff and Michael Casey. Musical audio synthesis using autoencoding neural nets, 2014. http://www.cs.dartmouth.edu/˜sarroff/papers/sarroff2014a.pdf.
156. Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, (11):2673–2681, 1997.
157. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
158. Roger N. Shepard. Geometric approximations to the structure of musical pitch. *Psychological Review*, (89):305–333, 1982.
159. Ian Simon and Sageev Oore. Performance RNN: Generating music with expressive timing and dynamics, 29/06/2017. https://magenta.tensorflow.org/performance-rnn.
160. Ian Simon, Adam Roberts, Colin Raffel, Jesse Engel, Curtis Hawthorne, and Douglas Eck. Learning a latent space of multitrack measures. In *Proceedings of the 19th International Society for Music Information Retrieval Conference (ISMIR 2018)*, Paris, France, September 2018.
161. Mark Steedman. A generative grammar for Jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.
162. 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/.

163. Bob L. Sturm, João Felipe Santos, Oded Ben-Tal, and Iryna Korshunova. Music transcription modelling and composition using deep learning. In *Proceedings of the 1st Conference on Computer Simulation of Musical Creativity (CSCM 16)*, Huddersfield, U.K., April 2016.
164. Felix Sun. DeepHear – Composing and harmonizing music with neural networks, Accessed on 21/12/2017. https://fephsun.github.io/2015/09/01/neural-music.html.
165. Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.
166. 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.
167. 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.
168. David Temperley. *The Cognition of Basic Musical Structures*. MIT Press, Cambridge, MA, USA, 2011.
169. Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models, 2015. arXiv:1511.01844.
170. John Thickstun, Zaid Harchaoui, and Sham Kakade. Learning features of music from scratch, December 2016. arXiv:1611.09827.
171. Alexey Tikhonov and Ivan P. Yamshchikov. Music generation with variational recurrent autoencoder supported by history, July 2017. arXiv:1705.05458v2.
172. Peter M. Todd. A connectionist approach to algorithmic composition. *Computer Music Journal (CMJ)*, 13(4):27–43, 1989.
173. A. M. Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, October 1950.
174. Dmitry Ulyanov and Vadim Lebedev. Audio texture synthesis and style transfer, December 2016. https://dmitryulyanov.github.io/audio-texture-synthesis-and-style-transfer/.
175. Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Abdelrahman Mohamed, Matthai Philipose, Matt Richardson, and Rich Caruana. Do deep convolutional nets really need to be deep (or even convolutional)?, May 2016. arXiv:1603.05691v2.
176. 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.
177. 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.
178. Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning, December 2015. arXiv:1509.06461v3.
179. Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer Nature, 1995.
180. Christian Walder. Modelling symbolic music: Beyond the piano roll, June 2016. arXiv:1606.01368.
181. Christian Walder. Symbolic Music Data Version 1.0, June 2016. arXiv:1606.02542.
182. Chris Walshaw. ABC notation home page, Accessed on 21/12/2016. http://abcnotation.com.
183. Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.
184. Raymond P. Whorley and Darrell Conklin. Music generation from statistical models of harmony. *Journal of New Music Research (JNMR)*, 45(2):160–183, 2016.
185. WikiArt.org. WikiArt – Visual Art Encyclopedia, Accessed on 22/08/2017. https://www.wikiart.org.
186. Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009.
187. Lonce Wyse. Audio spectrogram representations for processing with convolutional neural networks. In *Proceedings of the 1st International Workshop on Deep Learning for Music*, pages 37–41, Anchorage, AK, USA, May 2017.
188. Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition*. Indiana University Press, 1963.
189. Yamaha. e-Piano Junior Competition, Accessed on 19/03/2018. http://www.piano-e-competition.com/.
190. Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2Image: Conditional image generation from visual attributes, October 2016. arXiv:1512.00570v2.
191. 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.
192. Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks, July 2017. arXiv:1607.03474v5.