Music Generation by Deep Learning – Challenges and Directions

Jean-Pierre Briot* François Pachet†

* Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, Paris, France
Jean-Pierre.Briot@lip6.fr

† Spotify Creator Technology Research Lab, Paris, France
francois@spotify.com

Abstract: In addition to traditional tasks such as prediction, classification and translation, deep learning is receiving growing attention as an approach for music generation, as witnessed by recent research groups such as Magenta at Google and CTRL (Creator Technology Research Lab) at Spotify. The motivation is in using the capacity of deep learning architectures and training techniques to automatically learn musical styles from arbitrary musical corpora and then to generate samples from the estimated (learnt) distribution. Meanwhile, a direct application of deep learning generation, such as feedforward on a feedforward or a recurrent architecture, reaches some limits as they tend to mimic the corpus learnt without incentive for creativity. Moreover, deep learning architectures do not offer direct ways for controlling generation (e.g., imposing some tonality or other arbitrary constraints). Furthermore, deep learning architectures alone are autistic automata which generate music autonomously without human user interaction, far from the objective of assisting musicians to compose and refine music. Issues such as: control, creativity and interaction are the focus of our analysis. In this paper, we list various limitations of a direct application of deep learning to music generation, analyze why the issues are not fulfilled and how to address them by possible approaches. Various examples of recent systems are cited as examples of promising directions.

1 Introduction

Deep learning has become a fast growing domain and is now used routinely for classification and prediction tasks, such as image and voice recognition, as well as translation. It has emerged about 10 years ago, in 2006, when a deep learning architecture very significantly outperformed standard techniques using handcrafted features on an image classification task [HOT06]. We may explain this success and reemergence of *artificial neural networks* architectures and techniques by the combination of:

1. *technical progress*, such as: *pre-training* (which resolved initial inefficient training of neural networks with many layers [HOT06][1]); *convolutions* (which provide motif translation invariance [CB98]); LSTM (Long Short-Term Memory, which resolved inefficient training of recurrent neural networks [HS97]);

2. availability of *massive data*;

3. availability of *efficient and cheap computing power* (notably, thanks to graphics processing units – GPU – initially designed for video games, and that have now one of their biggest market in deep learning applications).

There is no consensual definition for *Deep learning*. It is a *repertoire* of *machine learning (ML)* techniques, based on *artificial neural networks*. The common ground is the term *deep*, which means that there are *multiple layers* processing *multiple hierarchical levels of abstractions*, which are automatically extracted from data[2], as a way to express complex representations in terms of simpler representations. The technical foundation is mostly *artificial neural networks*, with many variants (*convolutional networks*, *recurrent networks*, *autoencoders*, *restricted Boltzmann machines*...). For more information about the history and various facets of deep learning, see, e.g., [GBC16].

Important part of current effort in deep learning is applied to traditional machine learning *tasks*[3]: *classification* and *prediction* (also named *regression*), – as a testimony of the initial DNA of neural networks: *linear regression* and *logistic regression* – and also translation. But a growing area of application of deep learning techniques is the *generation* of *content*: *text*, *images*, and *music*, the focus of this article.

[1] Although now days it has being replaced by other techniques, such as *batch normalization* and *deep residual learning*.

[2] That said, and although a 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.

[3] 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 [GBC16, Chapter 5]. Examples are: classification, regression, translation, anomaly detection...

The motivation for using deep learning, and more generally machine learning techniques, to generate musical content is its generality. As opposed to handcrafted models for, e.g., grammar-based (e.g., [Ste84]) and rule-based music generation systems (e.g., [Ebc88]), a machine-learning-based generation system is *agnostic*, as it learns a model from arbitrary corpus of music, and the same system may be used for various musical genres.

Therefore, as more large scale musical datasets of various contexts 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 in [FC16], benefits are: 1) it can make creation feasible when the desired application is too complex to be described by analytical formulations or manual brute force design; 2) 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.

Generation can take place by using *prediction* (e.g., to predict the pitch of the next note of a melody [SSBTK16]) or *classification* (e.g., to recognize the chord corresponding to a melody [MB17]), by using standard deep learning architectures: *feedforward* neural network architectures or *recurrent* neural network (RNN) architectures. Sampling is also an interesting alternative, as we will see in Sections 5.2, 7.1, 9.5 and 11.2.

2 Challenges

Meanwhile, direct application of deep learning (and neural networks) architectures and techniques (training, prediction, classification) to generation does suffer from severe limitations. This is precisely the objective of this article to study these limitations, the challenges that they pose and the solutions, complete or partial, as well as current research directions to address them. Following is a tentative list of challenges:

- Input-less (or input-low) generation (vs Accompaniment of an input),
- Variable length (vs Fixed length),
- Variability (vs Determinism),
- Control (ex: tonality conformance, maximum number of repeated notes...),
- Originality (vs Imitation),
- Incrementality (vs One-shot or Temporal generation),
- Interactivity (vs Automation),
- Adaptability (vs Nothing learnt from experience),
- Explainability (vs Black box).

3 Related Work

There are some recent surveys about the use of deep learning to generate musical content. In [BHP17], Briot *et al.* survey various systems through a multicriteria analysis (objective, representation, architecture, strategy). In [HCC17], Herremans *et al.* propose a function-oriented taxonomy for various kinds of music generation systems. In [PW99], Papadopoulos and Wiggins survey various AI-based methods for music generation. In [Gra14], Graves analyses the application of recurrent neural networks architectures to generate sequences (text, music...). [Cop00] by Cope and [Nie09] by Nierhaus are examples of books about various methods for algorithmic composition and music generation. In [FC16], Fiebrink and Caramiaux address the issue of using machine learning to generate creative music. Meanwhile, we are not aware of a comprehensive analysis dedicated to deep learning (which subsumes various types of traditional artificial neural networks techniques) that systematically analyzes limitations and challenges (such as control, creativity, interactivity...), solutions and directions, in other words that is *problem-oriented* and not application-oriented.

In this article, we assume that the reader know the basics of feedforward (also named multilayer Perceptron) as well as recurrent neural networks (RNN). Otherwise, please see, e.g., [GBC16, Chapters 6 and 10] or [BHP17, Chapter 5].

4 An Introductory Example

In order to illustrate the challenges, we will introduce a first very simple (naive) example of a deep learning system for generating music, using the most direct strategy for generation: *single step feedforward*, applied to a *feedforward* neural network architecture[4].

[4]For an introduction to feedforward neural network architectures, see, e.g., [GBC16, Chapter 6] or [BHP17, Chapter 5].

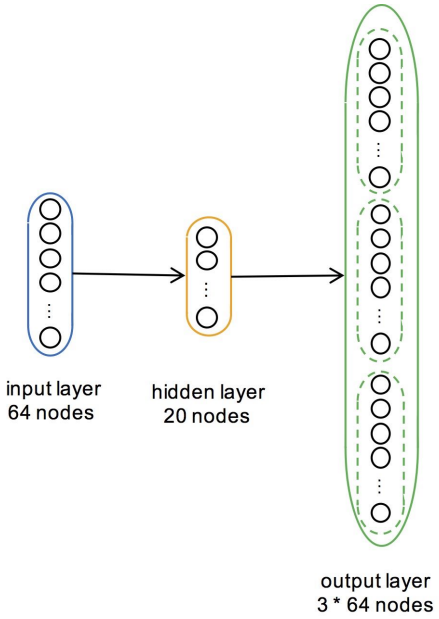

Figure 1: MicroBach architecture

4.1 Single Step Feedforward

4.1.1 Example: MicroBach Chorale Symbolic Generation System

Let us consider the following *task*: generating some *counterpoint accompaniment* of a given *melody*, representing a soprano voice, through three matching melodies, corresponding to alto, baryton and bass voices. We will use as a corpus the set of J.S. Bach polyphonic chorales music [Bac85]. Once trained on the dataset, composed of 352 examples, the system may be used to generate three counterpoint voices corresponding to an arbitrary melody provided as an input. Somehow, it does capture the pratique of Bach, who chose various given melodies for a soprano and composed the three additional ones (for alto, tenor and bass) in a *counterpoint* manner.

We need at first to decide what is the input as well as the output *representation*. We consider 4 measures of 4/4 music, with the minimal duration of a *note* being a half note (a quaver). We encode the *pitch* of the note corresponding to each successive time step by its corresponding MIDI value (and, e.g., value 0 if there is a silence)[5], followed by a value stating if the note *holds* onto next note, as a way to distinguish between a double length note and two successive identical notes.

The *input layer* of the *feedforward* architecture is composed of 4 measures * 8 notes * 2 hold informations = 64 nodes. The *output layer* is composed of 3 voices * 64 nodes = 192 nodes. The architecture chosen is a *feedforward network* with a single hidden layer (arbitrarily) composed of 20 nodes. and is shown at Figure 1. The *non linear function/unit* used for the hidden layer is the pretty standard ReLU (rectified linear unit)[6].

This system, named *MicroBach* and designed by Gaëtan Hadjeres and Jean-Pierre Briot[7], is an example of a *single step direct feedforward strategy* (see [BHP17, Section 6.1.1]). An example of generated chorale is shown at Figure 2.

5 Input-less (or input-low) generation

Although the generation is effective and produces Bach-like chorales, a first limitation is that it can only generate an output matching some input, in other words, it assumes that there is some *input*. But, sometimes we would like to generate music, not as an accompaniment of an existing music, but *from scratch*, or from a minimal set of information (e.g., a first note, some higher level features. . .), while of course based on the musical style learnt. Three main strategies are then possible:

- *Stacked autoencoders decoding*,
- *RBM sampling*,
- *RNN iterative feedforward*.

[5] A more standard encoding choice for symbolic music representation is to encode the pitch of a note through *one-hot encoding*, as a vector of pitches where the only non negative (1) value corresponds to the actual pitch. This *item encoding* strategy (see Section [BHP17, Section 4.4.5]) represents the discretization of the alternative analogical *value encoding* and is more robust, at the cost of a greater number of input (and output) nodes. See, e.g., an alternative one-hot encoding in [BHP17, Section 7.1.1.1].

[6] $ReLU(x) = max(0, x)$.

[7] The system is actually a strong simplification of the *DeepBach* system (see Section 11.2), with the same corpus – but a simplified representation – and objective.

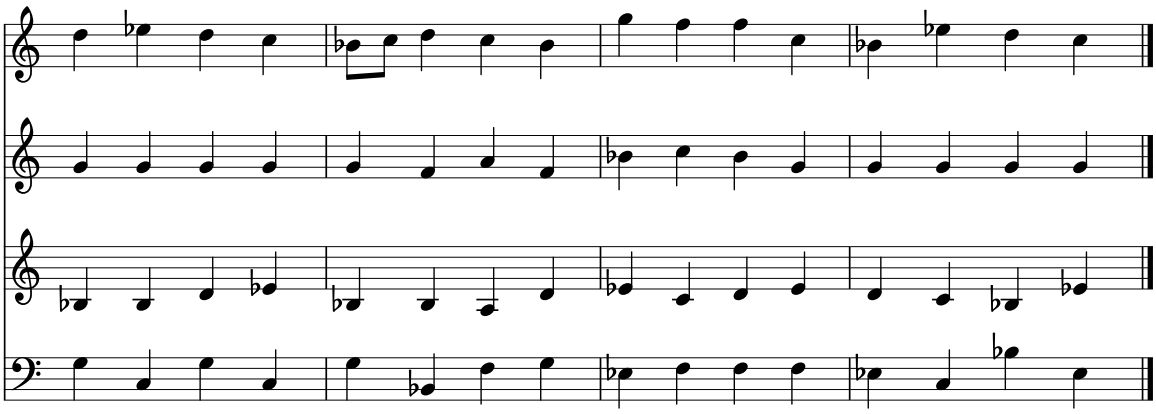

Figure 2: Example of counterpoint generated from the first voice

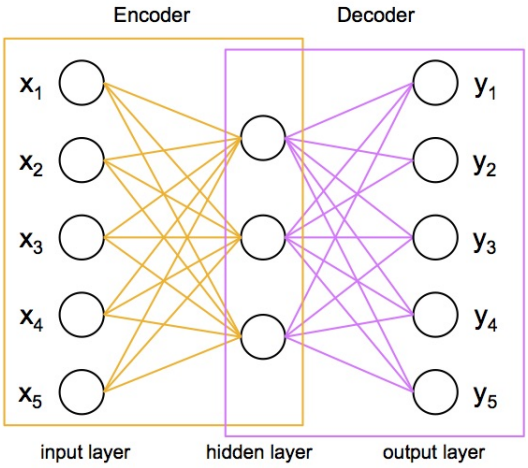

Figure 3: Autoencoder

5.1 Stacked Autoencoders Decoding

A first strategy is in automatically extracting a high-level and minimal set of parameters to describe the type of music learnt and then to use these parameters as the input to parameterize the generation of music. For this purpose, stacked autoencoders are used.

5.1.1 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 (in other words the output layer *mirrors* the input layer). The hidden layer is usually smaller (smaller number of nodes), resulting in the characteristic diabolo-shape, as shown at Figure 3.

Training an autoencoder is done by using conventional *supervised* learning, but with the output values equal to the input values[8], with the autoencoder task being to learn the *identity* function. As the hidden layer has usually fewer nodes than the input layer, the *encoder* part has to compress information and the *decoder* part has to reconstruct, as well as possible, the initial information. This forces the autoencoder to *discover* significant (discriminating) *features* to *encode* useful information[9].

Therefore, autoencoders may be used to automatically extract higher level *features* [LRM+12]. The set of features extracted are often also named an *embedding*[10]. Once trained, in order to extract features from an input, one just needs to feed forward the input data and gather the activations.

5.1.2 Stacked Autoencoders

Stacked autoencoders are *hierarchically nested* autoencoders, as illustrated at Figure 4. The pipeline of encoders will

[8] This is sometimes called *self-taught* (or *self-supervised*) learning [LRM+12].

[9] In order to further guide the autoencoder, additional constraints may be used, such as activation *sparsity*, i.e., at most one node (neuron) is active, in order to enforce *specialization* of each node of the hidden layer as a specific *feature detector*.

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

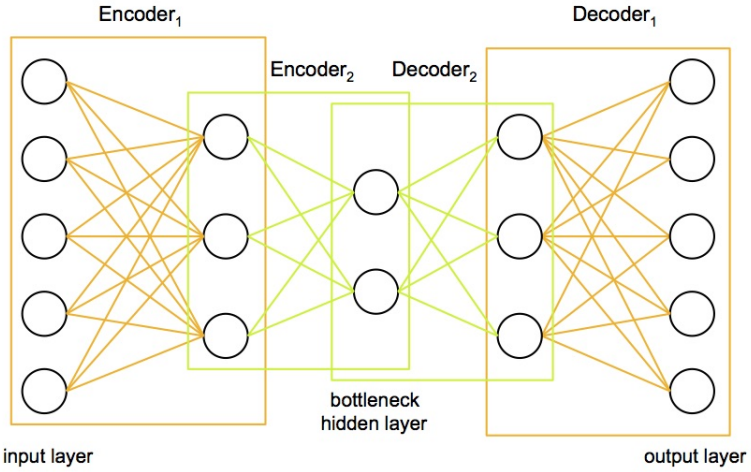

Figure 4: Stacked autoencoders architecture

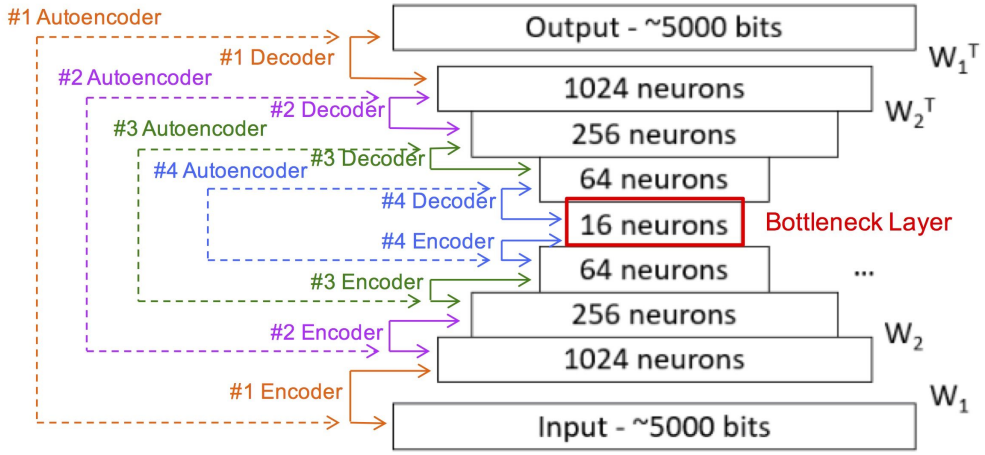

Figure 5: DeepHear stacked autoencoders architecture

increasingly compress data and extract increasingly higher-level features. Stacked autoencoders (which are indeed deep networks) are therefore heavily used for automated feature extraction, to be used, e.g., for music information retrieval.

5.1.3 Generating by Decoding

The *stacked autoencoders decoding* strategy uses hierarchically nested autoencoders to extract *features* from a corpus of musical content through the encoding process. In essence, the hierarchical encoding/decoding of the corpus is supposed to have extracted *discriminating features* of the corpus, thus characterizing its variations through a few features.

Generation is done by creating a *seed* value corresponding to the *innermost hidden layer* (called the *bottleneck hidden layer*) of the stacked autoencoders, inserting it at the bottleneck hidden layer (in other words, at the exact middle of the encoder/decoder stack, see Figure 6) and feedforwarding to the chain of decoders. This will *reconstruct* a corresponding musical content, arbitrarily complex and long, at the output layer. Therefore, only a simple seed information can generate an arbitrarily complex and long musical content.

5.1.4 #1 Example: DeepHear Ragtime Melody Symbolic Music Generation System

An example of this strategy is *DeepHear* system by Sun [Sun17]. The representation used is *piano roll* with *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 segments (thus 64 time steps). The number of input nodes is around 5000, which means having a vocabulary of about 80 possible note values. The architecture is shown at Figure 5 and is composed of 4 stacked autoencoders (with decreasing numbers of hidden units, down to 16 units).

Generation is performed by inputing random data as the seed into the 16 units bottleneck hidden layer and then feedforwarding the decoder to produce an output (in the same 4 measures format of the training examples), as shown at Figure 6.

Sun remarks that the system does some amount of plagiarizing. Some generated music is almost recopied from the corpus. Sun explains this because of the small size of the bottleneck hidden layer (only 16 units) [Sun17]. He

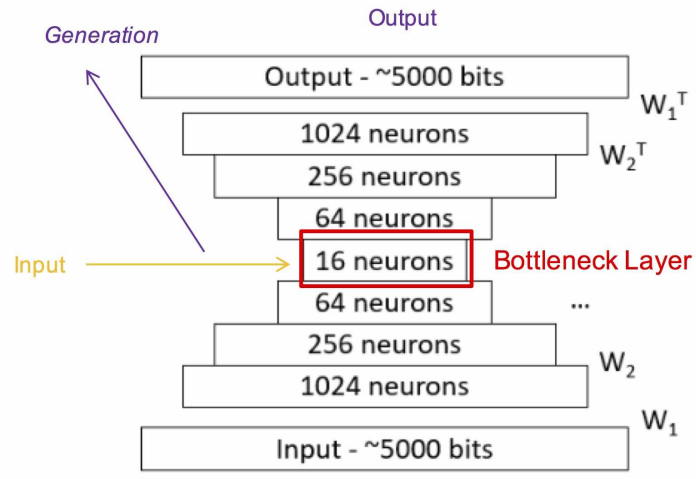

Figure 6: Generation in DeepHear

measured the similarity (defined as the percentage of notes in a generated piece that are also in one of the training pieces) and found that, on average, its value is 59.6%, which it is indeed quite high, but does not prevent most of generated pieces to sound different.

5.1.5 #2 Example: deepAutoController Audio Music Generation System

deepAutoController, by Sarroff and Casey [SC14], is similar to DeepHear (see Section 5.1.4) in that it also uses *stacked autoencoders*, but the representation is *audio* (more precisely a *spectrogram* generated by Fourier transformation). The dataset is composed of 8000 songs of 10 musical genres. The system also provides a user interface (see Section 12) to interactively control the generation by, e.g., select a given input (to be inserted at the bottleneck hidden layer), generate a random input, control (by scaling or muting) the activation of a given unit.

5.2 RBM Sampling

A *Restricted Boltzmann Machine* (RBM) [HS06] (see [GBC16, Section 16.7.1]) is a neural network architecture aimed at learning probability distributions, e.g., *vertical* correlations between *simultaneous* notes learnt from a corpus of *chords*. Once trained, one may *draw samples* from a RBM in order to generate a *content* according to the distribution learnt.

5.2.1 RBM

The name *Restricted Boltzmann Machine* (*RBM*) [HS06] comes both from: *Boltzmann distribution* in statistical mechanics, which is used in their sampling function, and the fact that it is a *restricted* and practical form of general *Boltzmann Machines* [HS86]. In practice, a RBM is a *generative stochastic* artificial neural network that can learn a *probability distribution* over its set of inputs. It is organized in *layers*, just as neural networks and autoencoders are, and more precisely two layers (see Figure 7):

- the *visible* layer (analog to both the *input layer* and the *output layer* of an *autoencoder*),
- the *hidden* layer (analog to the *hidden layer* of an *autoencoder*).

A RBM bears some similarity in spirit and objective with an autoencoder (see Section 5.1.1), but with some important differences:

- a RBM has *no ouput* – the *input* acting *also* as the *output*,
- a RBM is *stochastic* (and therefore, *not deterministic*, as opposed to neural networks and autoencoders),
- values manipulated are *booleans*[11].

RBMs became popular after Hinton used them for *pre-training* deep neural networks [EBC+10], after designing a fast specific learning algorithm for them, named *contrastive divergence* [Hin02].

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

[11] Note that there are also versions of RBM with *continuous values*, as for neural networks, see for instance Section 9.5.1.

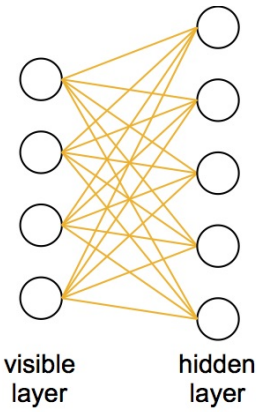

Figure 7: Restricted Boltzmann machine (RBM) architecture

Training a RBM has some similarity to the training of an autoencoder (we will not detail it here, see, e.g., [GBC16, Section 20.2]), with the practical difference that because there is no decoder part with an output layer mirroring the input layer, the RBM will alternate between two steps:

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

5.2.2 RBM Sampling

Sampling is the action of generating an element (a *sample*) from a *stochastic* model with a given *probability distribution*. The main issue is to ensure that the samples generated match a given distribution. Therefore, various sampling strategies have been proposed: *Metropolis-Hastings*, *Gibbs (GS)*, *block Gibbs*, etc. Please see, e.g., [GBC16, Chapter 17] for details about sampling and various sampling algorithms.

In the case of a RBM, after the training phase has been done, in the *generation* phase, a *sample* can be drawn from the model by randomly initializing the visible layer vector v (following a standard uniform distribution) and running *sampling*[12] until convergence[13]. To this end, as for the training phase, hidden nodes and visible nodes are alternately updated (as during the training phase).

5.2.3 Example: RBM-based Polyphonic Music Generation System

In [BLBV12], Boulanger-Lewandowski *et al.* at first propose to use a *restricted Boltzmann machine* (RBM) [HS06] to model polyphonic music. Their prior objective is actually to model polyphonic music (by learning a model from a corpus) in order to improve transcription of polyphonic music from audio. But they also discuss the generation of samples of the model learnt as a qualitative evaluation and finally for music generation [BL15]. They use a more general non-Boolean model of RBM, where variables have real values and not just Boolean values. In their first experiment, the RBM will learn from the corpus the distribution of possible simultaneous notes, i.e., a repertoire of chords.

Once having trained the RBM, they can sample from the RBM by *block Gibbs sampling*, by performing alternative steps of sampling hidden layer nodes (considered as variables) from visible layer nodes. Figure 8 shows various samples, each column representing a specific sample vector of notes, with the name of the chord below where the analysis is unambiguous.

5.3 RNN Iterative Feedforward

5.3.1 RNN

A *Recurrent Neural Network* (RNN) is a (feedforward) neural network extended to include *recurrent connexions*[14]. The basic idea is that the outputs of a hidden layer *reenter* into itself as an additional input to compute next values of the hidden layer. This way, the network can learn, not only based on *current* data, but also on *previous* one. Therefore, it can learn *series*, notably *temporal series* (the case of musical content). RNNs are routinely used, e.g., for natural

[12] More precisely *Gibbs sampling (GS)*, see [Lam16].

[13] In practice, convergence is reached when the *energy* stabilizes. The *energy* of a *configuration* (the pair of vectors of values of visible and hidden layer nodes) is expressed as $E(v,h) = -a^{\mathrm{T}}v - b^{\mathrm{T}}h - v^{\mathrm{T}}Wh$, where v and h, respectively, are the visible and the hidden layers, W is the matrix of weights associated with the connections between visible and hidden nodes and a and b, respectively, the bias weights for visible and hidden nodes. For more details, see, e.g., [GBC16, Section 16.2.4].

[14] For an introduction to recurrent neural network architectures, see, e.g., [GBC16, Chapter 10] or [BHP17, Chapter 5].

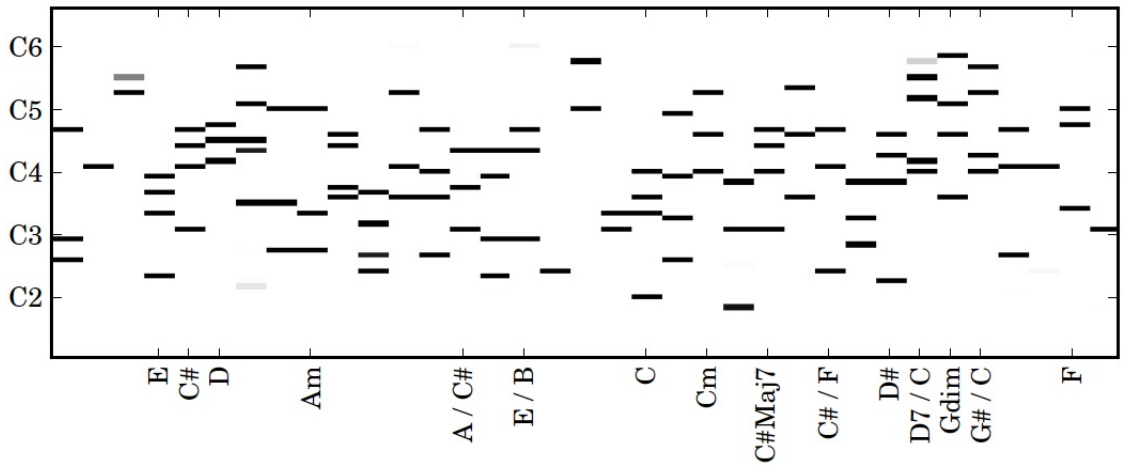

Figure 8: Samples generated by the RBM trained on Bach chorales

text processing and for music generation. The *de facto* standard for recurrent networks is *Long short-term memory (LSTM)* [HS97], that will be not detailed here.

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

5.3.2 Iterative Generation by a RNN

The *recurrent* nature of a *recurrent network* (RNN), which learns to predict next step information, can be used to generate sequences of *arbitrary* lengths[15]. The typical usage, as described by [Gra14] for text generation, is to enter some *seed* information as the first item (e.g., the first note of a melody), to generate by feedforward the next item (e.g., next note), which in turn is used as an input and so on, until producing a sequence of the length desired. Therefore, we name this strategy *iterative feedforward*. An example is the generation of melodies described in Section 5.3.3.

5.3.3 Example: Blues Melody Symbolic Generation System

In [ES02], Eck and Schmidhuber describe a double experiment done with a *recurrent network* architecture using LSTMs. The format of representation is *piano roll*, with 2 types of sequences: *melody* and *chords*, although chords are represented as notes. The melodic range as well as the chord vocabulary is strongly constrained, as the corpus is about 12-bar blues and is handcrafted (melodies and chords). The 13 possible notes extend from middle C (C_4) to tenor C (C_5). The 12 possible chords extend from C to B.

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

In their second experiment, the objective is to simultaneously learn and generate melody and chord sequences. The architecture has an input layer with 25 nodes (corresponding to the one-hot encoding of the 12 chords vocabulary and of the 13 melody notes vocabulary), one hidden layer with 8 LSTM blocks (4 chord blocks and 4 melody blocks[16], containing 2 cells each[17]) and an output layer with 25 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 next time step chord, using it as next input and so on, until generating a sequence of chords. Figure 10 shows an example of melody and chord generated.

6 Variable length

An important limitation of the first strategy used, *single step feedforward* (see Section 4.1), as well as the two next ones, *stacked autoencoders decoding* (see Section 5.1) and *RBM sampling* (see Section 5.2), is that the size (length) of the music generated is *fixed*. It is actually fixed by the architecture, namely the number of nodes of the output layer.

In contrast, the *iterative feedforward* generation strategy on a RNN (see Section 5.3) is of *variable* length. As generation is iterative, by inputing a seed information as first item of the series into the RNN, which produces next item, used as next input and so on, the generation goes on as long as the iteration, in other words, the size of the generated sequence (e.g., a melody) is arbitrary long.

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

[16] Recurrent connexions between chord and melody blocks are asymetric, as the authors wanted to ensure the preponderant role of chords, see [ES02] and [BHP17, Section 7.1.2.2] for details and discussion.

[17] Cells within a same block *share* input, output and forget gates, i.e. although each cell might hold a different value in its memory, all cell memories within a block are read, written or erased *all at once* [HS97].

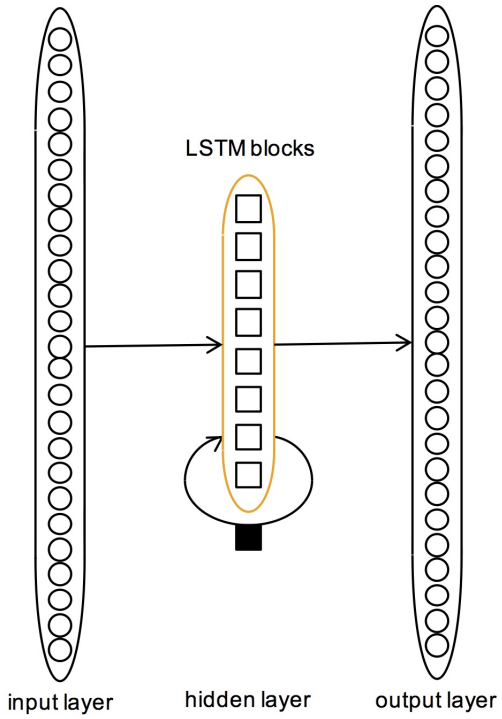

Figure 9: Blues generator second experiment architecture

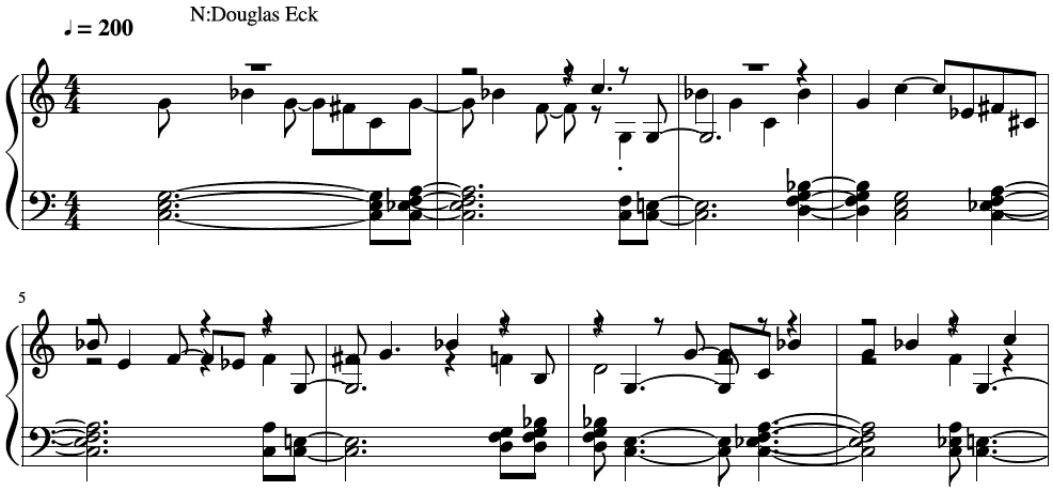

Figure 10: Example of Blues generated (excerpt)

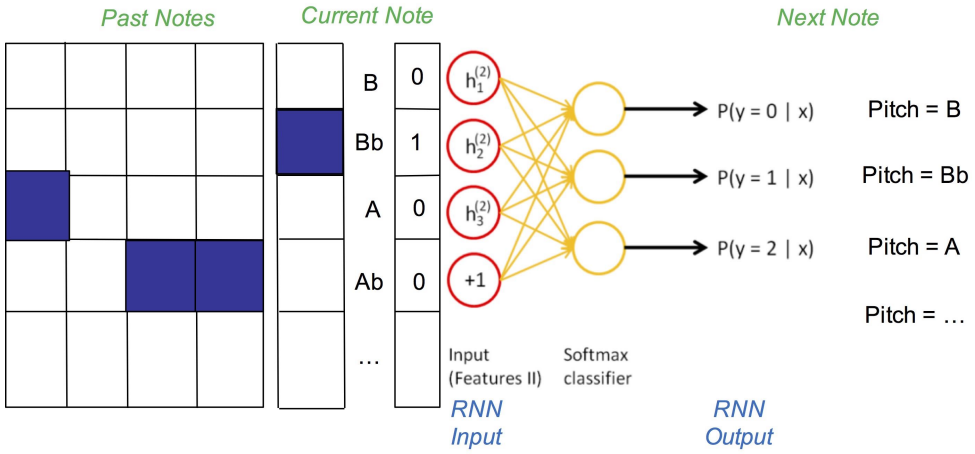

Figure 11: Sampling the output

7 Variability

A limitation of the *iterative feedforward* strategy on a RNN, as illustrated by the Blues generation experiment described in Section 5.3.3, is that generation is *deterministic*. Indeed, a neural network is deterministic[18], feedforwarding the *same input* will always produce the *same output*. As the generation of next note, of next next note, etc., is deterministic, the same seed note will lead to the same generated series of notes (the actual melody length depends on the number of iterations). As there are only 12 possible input values (the 12 pitch classes), there are only 12 possible melodies.

7.1 Sampling

Fortunately, as we will see the solution is simple. The assumption is that the output representation is *one-hot* encoded (See Section 4). In other words, the output representation is a *piano roll* or alike, the output activation layer is *softmax* or alike, and generation is modeled as a *classification* task[19], see Figure 11. The default *deterministic* strategy consists in choosing the class (the note/pitch) with the *highest probability*. We can then easily switch to a *non deterministic* strategy, by introducing *sampling* from the output which corresponds (through the softmax) 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.

8 A First Discussion

The challenges that we discussed so far have been solved. They were more *limitations* than real challenges. Meanwhile, we saw that some limitations may conflict which each other and with some objectives. For instance, we saw in Section 6 that the *iterative feedforward* strategy allows *variable length* generation. But, because the iterative feedforward strategy generates *homogeneous* sequences, this constrains the *nature* (and the *structure*) of the generated *output*, which must be *isomorphic* to the input (same number of nodes, see Section 5.3.1) and thus cannot be *arbitrary*. Other examples of conflicts will be discussed in Section 15. Now we will address deeper challenges which are harder to address.

9 Control

A deep architecture generates *musical content* matching the corpus learnt. This capacity of induction from a corpus without any explicit modeling nor programming is an important ability, as discussed in Section 1 [FC16]. Meanwhile, like a fast car needs a good steering wheel, 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.

9.1 Dimensions of control strategies

Such arbitrary control is a deep and difficult issue for deep learning architectures and techniques because neural networks are not designed to be controlled, moreover, no internal control, as, e.g., opposed to Markov chains, where one can attach constraints on the internal structure in order to control the generation[21]. As a result, as we will

[18] There are stochastic versions of artificial neural networks – a RBM is an example – but they are not mainstream.
[19] In the case of the chorale generation example (see Section 4), a classification between the possible notes/pitches.
[20] The chance of sampling a given class/note corresponds to its corresponding probability.
[21] Two examples are Markov constraints [PR11] and factor graphs [PPR17].

see, strategies for controlling deep learning generation will rely on some *external* intervention at various *entry points* (hooks) and *levels*:

- *input*,
- *output*,
- *input* and *output*,
- *encapsulation/reformulation*.

We will also see that some strategies (such as output sampling control, see Section 9.5.1) are more *bottom-up* and other ones, (such as structure imposition, see Section 9.5.1, or unit selection, see Section 9.8) are more *top-down*. Last, there is also a continuum between *partial* solutions (such as conditioning/parametrization, see Section 9.3) and more *general* solutions (such as reinforcement, see Section 9.6).

9.2 Sampling

Sampling from a *stochastic architecture* (such as a restricted Boltzmann machine – RBM, see Section 5.2), or from a *deterministic architecture* (in order to introduce *variability*, see Section 7.1), may be an entry point for *control* if we introduce *constraints* on the sampling process (this is called *constraint sampling*, see, e.g., Section 9.5.1). This is usually implemented by a *generate-and-test* approach, where valid solutions are picked from a set of generated random samples from the model, which could be a very costly process and moreover with no guarantee to succeed. A key and difficult issue is how to *guide* the sampling process in order to fulfill the *objectives* (constraints). As a result, this strategy is rarely used, at least *alone*. We will see in Section 9.5 that it may be combined with another strategy (namely, *input manipulation*) as as way to *correct* the control done by that other strategy in order to *realign* the samples with the learnt distribution.

9.3 Conditioning

The idea of *conditioning* (sometimes also named *conditional architecture*) is to *condition* the architecture on some extra (*conditioning*) information, which could be arbitrary, e.g., a class label or data from other modalities. Examples are:

- a *bass line* or a *beat structure*, in the rhythm generation system described in Section 9.3.1;
- a *chord progression*, in the MidiNet architecture described in Section 9.3.3;
- a *musical genre* or an *instrument*, in the WaveNet architecture described in Section 9.3.2.

In practice, the conditioning information is usually fed into the architecture as an *additional input layer* (e.g., see Figure 12). This distinction between *standard input* and *conditioning input* follows a good architectural *modularity* principle. Conditioning is a way to have some degree of *parametrized control* over the generation process.

9.3.1 #1 Example: Makris et al. Rhythm Symbolic Generation System

The system proposed by Makris *et al.* [MKPKK17] is specific in that it is dedicated to the generation of sequences of *rhythm*. Another specificity is the experiment they do in considering the possibility to *condition* the generation relative to some information, such as a given *beat* or a given *bass line*.

The architecture is a combination of a *recurrent network* (more precisely, a LSTM) and a *feedforward network*, representing the *conditioning layer*. The LSTM (2 stacked LSTM layers with 128 or 512 units) is in charge of the *drums* part, while the feedforward network is in charge of the *bass line* and the metrical structure (*beat*) information. These two networks are then merged, resulting in the architecture illustrated at Figure 13. 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 [MKPKK17].

9.3.2 #2 Example: WaveNet Speech and Music Audio Generation System

WaveNet by van der Oord *et al.* [vdODZ+16] is a system for generating raw *audio* waveforms. It has been experimented on generation for three audio domains: multi-speaker, text-to-speech (TTS) and music.

The architecture (illustrated at Figure 14 and not detailed here) is also made conditioning, by adding an additional input. There are actually two options: *global conditioning* or *local conditioning*, depending if the conditioning input is shared for *all* time steps or is specific to *each* time step. An example of application of conditioning WaveNet for a text-to-speech application domain is to feed linguistic features (e.g., North American English or Mandarin Chinese speakers) in order to generate speech with a better prosody.

The authors conducted preliminary work on conditioning music models to generate music given a set of tags specifying, e.g., *genre* or *instruments*, and state that their preliminary attempt is promising [vdODZ+16].

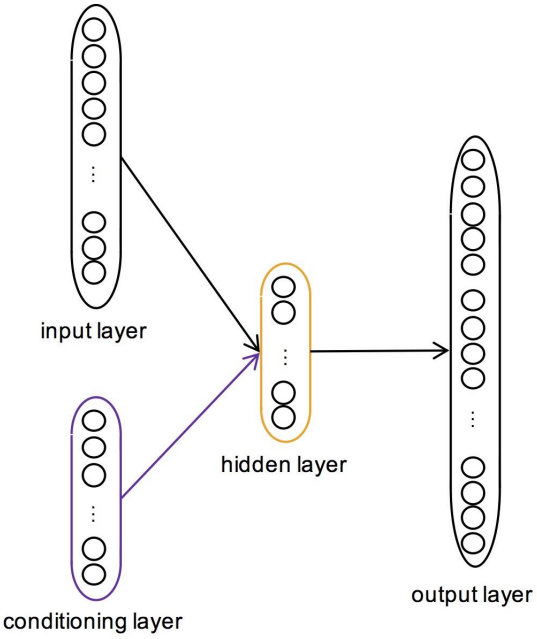

Figure 12: Conditioning architecture

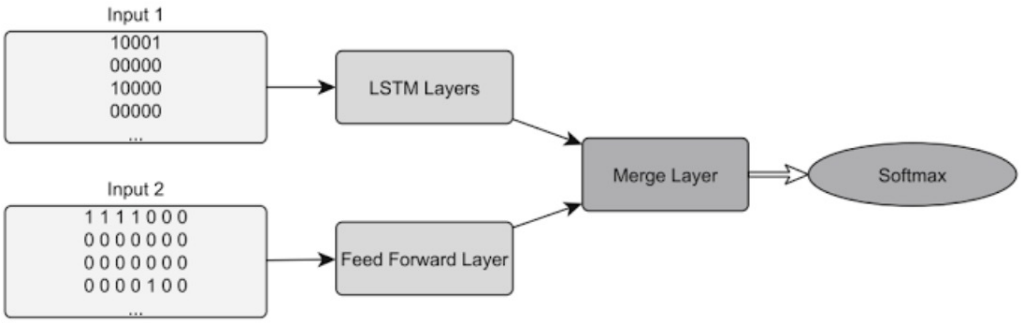

Figure 13: Rhythm generation architecture

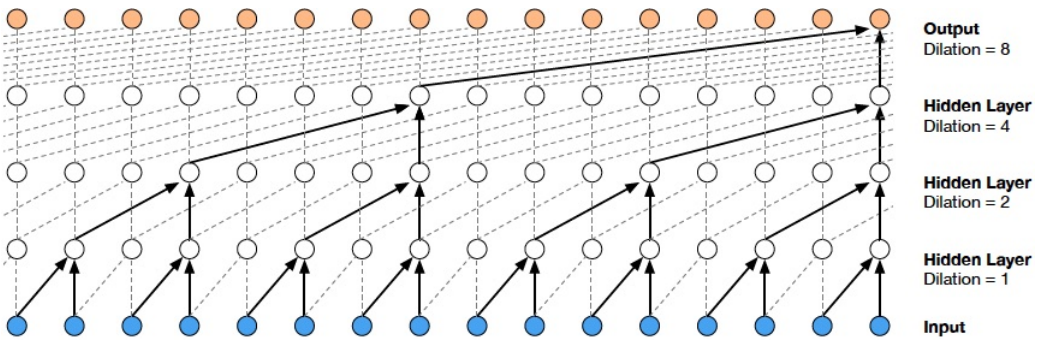

Figure 14: WaveNet architecture

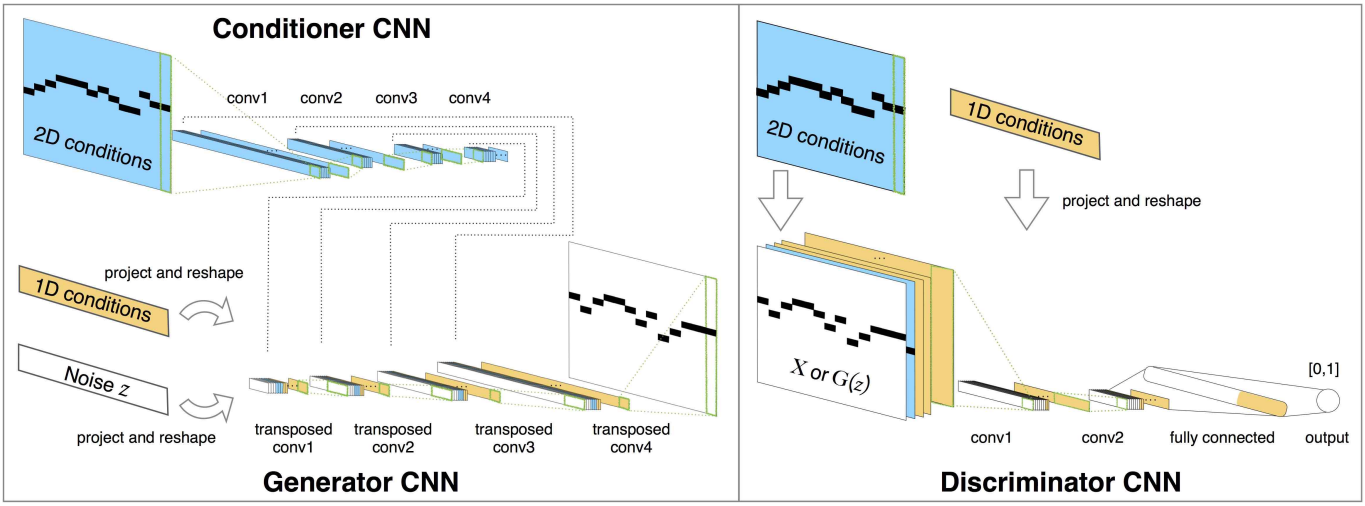

Figure 15: MidiNet architecture

9.3.3 #3 Example: MidiNet Pop Music Melody Symbolic Generation System

In [YCY17], Yang *et al.* propose the MidiNet architecture as an architecture both *adversarial*[22] and *convolutional*, to generate pop music melodies. The architecture is illustrated at Figure 15. It is composed of a generator and a discriminator network, both convolutional networks. The generator includes 2 fully-connected layers (with 1,024 and 512 units respectively) followed by 4 convolutional layers. The conditioner includes 4 convolutional layers but with a reverse architecture. The discriminator includes 2 convolutional layers followed by some fully connected layers and the final output activation function used being *cross-entropy*.

The *conditioning* mechanism incorporates information from previous measures (as a memory mechanism, analog to a recurrent network). The authors show how their system can generate music by following a chord progression, or by following a few starting notes (a priming melody). The conditioner also allows to incorporate information from previous measures to intermediate layers and therefore consider history as would do a recurrent network. In addition, the authors discuss two methods to control *creativity* (see Section 10).

9.4 Input Manipulation

This strategy about *input manipulation* has been pioneered for images by DeepDream. 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 being applied to the output but to the *input*, *before generation*. Examples are:

- *maximizing* the *similarity* to a given *target*, in order to create a *consonant melody*, in DeepHear (see Section 9.4.1);
- *maximizing* the *activation* of a specific *unit*, to *exaggerate* some visual element specific to this unit, in DeepDream (see Section 9.4.2);
- *maximizing both* the *content similarity* to some initial image *and* the *style similarity* to a reference style image, to perform *style transfer* (see Section 9.4.3);
- *maximizing* the *similarity* of *structure* to some reference music, to perform *style imposition* (see Section 9.5.1).

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

9.4.1 #1 Example: DeepHear Ragtime Counterpoint Symbolic Generation System

A second experiment has been conducted by Sun (see his first experiment in Section 5.1.4) with a different objective: harmonize a melody, while using the same architecture and what had already been learnt[23]. The idea is to find an embedding – the values of the 16 units of the bottleneck hidden layer of the stacked autoencoders – which will result in some generated output matching as much as possible a given melody. Therefore, a simple distance (error) function is defined to represent the distance (*similarity*) between two melodies (in practice, the number of non matched notes). Then just remains a gradient descent onto the embedding, guided by the gradients corresponding to the error function, until finding a sufficiently similar decoded melody.

[22] See Section 10.2.1 and [GBC16, Section 20.10.4].

[23] It is a simple example of *transfer learning* (see [GBC16, Section 15.2]), with a same domain and a same training done, but onto a different production task.

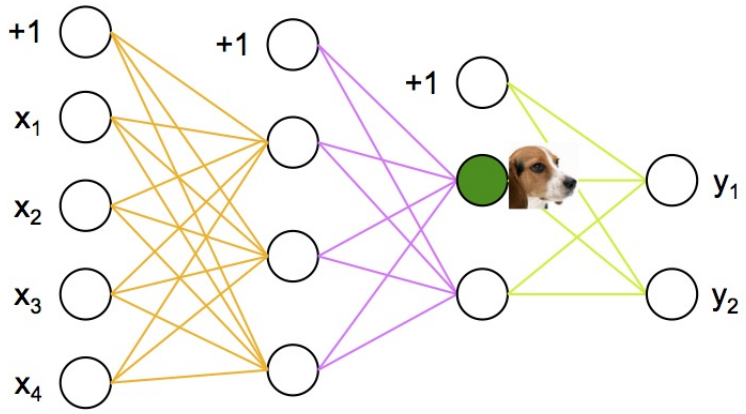

Figure 16: DeepDream architecture

Although this is not real harmonization (see Section 5.2.3 for an example of experiment specific for harmonization), but rather generation of a similar (consonant) melody, the results (tested on Ragtime melodies) do produce some naive counterpoint with a ragtime flavor.

9.4.2 #2 Example: Deep Dream Psychedelic Images Generation System

DeepDream by Mordvintsev *et al.* [MOT15] has become famous for generating psychedelic versions of standard images. The idea is to use a deep feedforward neural network architecture (see Figure 16) and to use it to *guide* the incremental alteration of an initial input image, in order to maximize the potential occurrence (pareidolia) of a specific visual feature (visual element, identified by the activation of a given unit, filled in green in Figure 16).

The method is as follows:

- the network is first trained on a large images dataset;
- instead of minimizing the cost function, the objective is to *maximize* the *activation* of a specific *node*(s) (which has been identified to activate for a specific pattern, e.g., a dog face – this node being represented in green in the left part of Figure 16);
- an initial image (e.g., of a tree) is *iteratively* slightly *altered* (e.g., by jitter[24]), under gradient ascent control, in order to *maximize* the *activation* of that specific node. This will favor emergence of occurrences of that specific pattern in the image (see, e.g., the right part of Figure 16).

Note that the activation maximization of a specific *higher layer node* (the case here, see Figure 16) will favor the emergence in the initial image (e.g., a tree) of that specific *high-level pattern* (e.g., a dog face at Figure 17), whereas the activation maximization of a specific *lower layer node* (Figure 18) will result in *texture insertion* (e.g., in a cloud, see Figure 19).

One may transpose the DeepDream objective to music, by maximizing the activation of a specific node (specially if the role of a node has been identified, through a <node/layer activation, musical motif> correlation analysis, as, e.g., in Section 14.1).

Somehow, a similar strategy has been used in the second experiment of DeepHear, described in Section 9.4.1, to manipulate the values of the bottleneck hidden layer of a stacked autoencoders architecture, in order to produce (through the decoder) a melody similar (*consonant*) to another target melody.

9.4.3 #3 Example: Style Transfer Painting Generation System

The idea in this approach, pioneered by Gatys *et al.* [GEB15] and designed for images, is to use a deep learning architecture to independently capture:

- features of a first image (named the *content*),
- and the *style* (as a correlation between features) of a second image (named the *style*),
- and then, to use gradient-based learning 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 *content* information of the first image (the content reference), by feed-forwarding it into the network and by storing *units activations* for each layer;

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

Figure 17: DeepDream example of resulting image

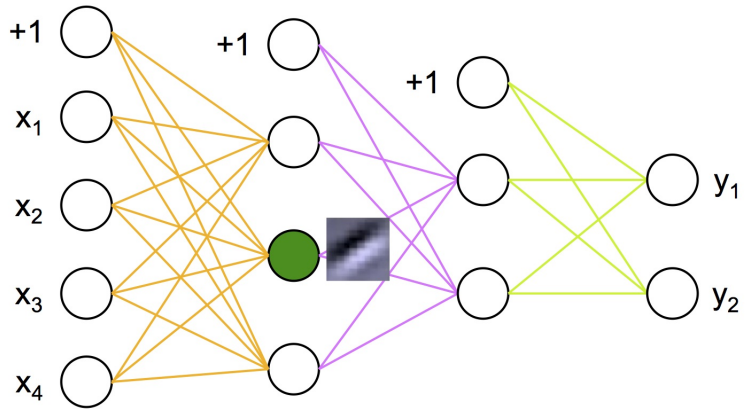

Figure 18: Deep Dream architecture focused on a lower level unit

Figure 19: Example of lower layer unit maximization resulting image

- capture *syle* information of the second image (the style reference), by feed-forwarding it into the network and by storing *feature spaces*, which are *correlations* between units activations for each layer;
- synthesize an hybrid image in the following way:
  - at first, generate a random image, then define it as current image,
  - and then *iterate*, until reaching the *two targets* (both *content similarity* and *style similarity*):
    - capture the *contents* and the *style* information of current image,
    - compute the *content cost* (distance between reference and current content) and the *style cost* (distance between reference and current style),
    - compute the corresponding *gradients*, through standard back-propagation,
    - *update* current image guided by the gradients;

The architecture and process are summarized at Figure 20 (more details may be found in [GEB15]). The content image (on the right) is a photography of Neckarfront in Tübingen[25] in Germany and the style image is the painting "The Starry Night" by Vincent van Gogh.

Note that one may balance between content and style objectives[26] (α/β ratio) in order to favor one or the other. In addition, the complexity of the capture may also be adjusted via the number of hidden layers used. These variations are shown at Figure 21 with: *rightwards* an *increasing* α/β content/style objectives ratio and *downwards* an *increasing* number of hidden layers used for matching style information. The style image is the painting "Composition VII" by Wassily Kandinsky.

9.4.4 #4 Example: Musical Style Transfer

Transposing this style transfer technique to music is a tempting direction. However, the style of a piece of music is less easy to capture via correlation of activations. For audio it has already been experimented independently by [UL16] and [FYR16], usually using a spectrogram (and not a direct wave signal) as input. The result is acceptable, but not as interesting as in the case of painting style transfer, being more similar to a sound merging of the style and of the content. In their study [FYR16], 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 believe that this is because of the *anisotropy*[27] of global music content representation. In the case of an image, the correlations between visual elements (pixels) are equivalent whatever the direction (horizontal axis, vertical axis, diagonal axis or any arbitrary direction), in other words correlations are *isotropic*. In the case of a global representation of musical content, where the horizontal dimension represents time and the vertical dimension represents the notes (pitches), this uniformity does not hold anymore, as horizontal correlations represent *temporal* correlations and vertical correlations represent *harmonic* correlations, of very different nature (see Figure 22).

[25] The location of the researchers.

[26] Through the α and β parameters, see on the top of Figure 20 the total loss defined as $\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$.

[27] Isotropy means uniformity in all orientations, whereas anisotropy, its opposite, means dependence on directions.

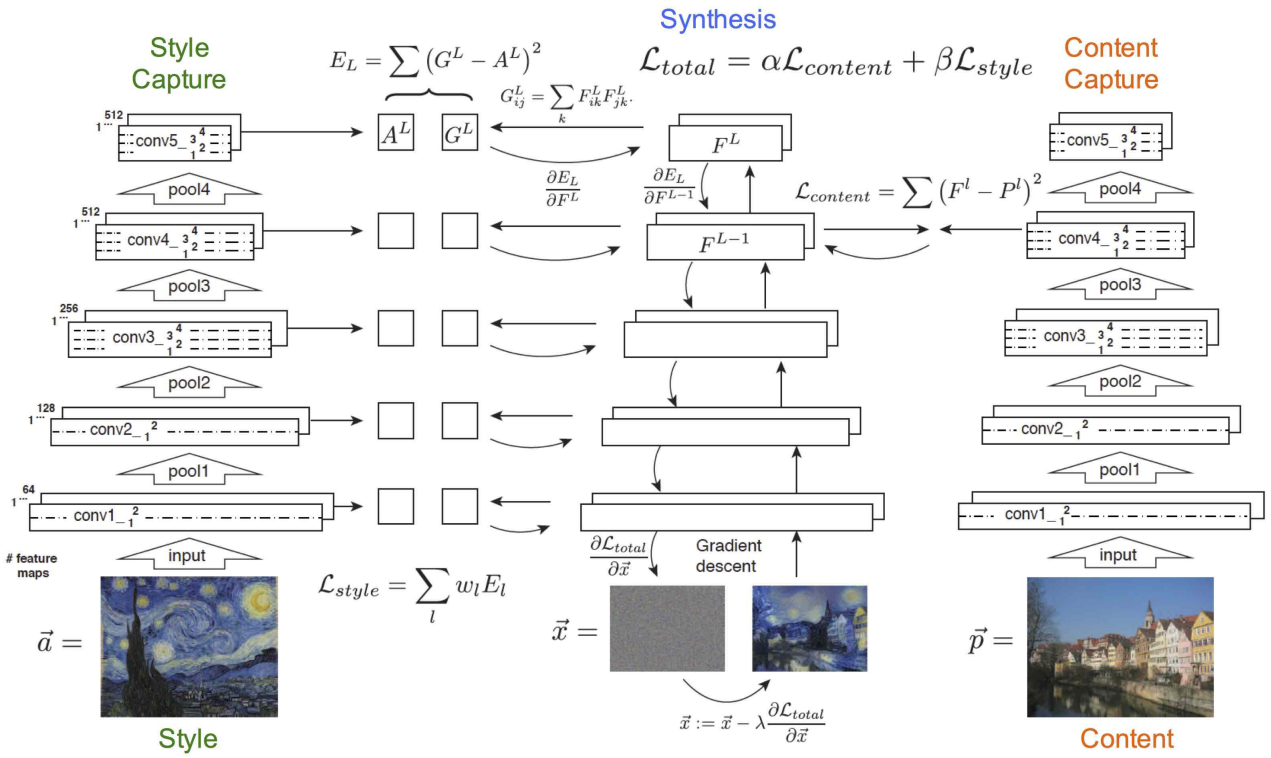

Figure 20: Style transfer full architecture/process

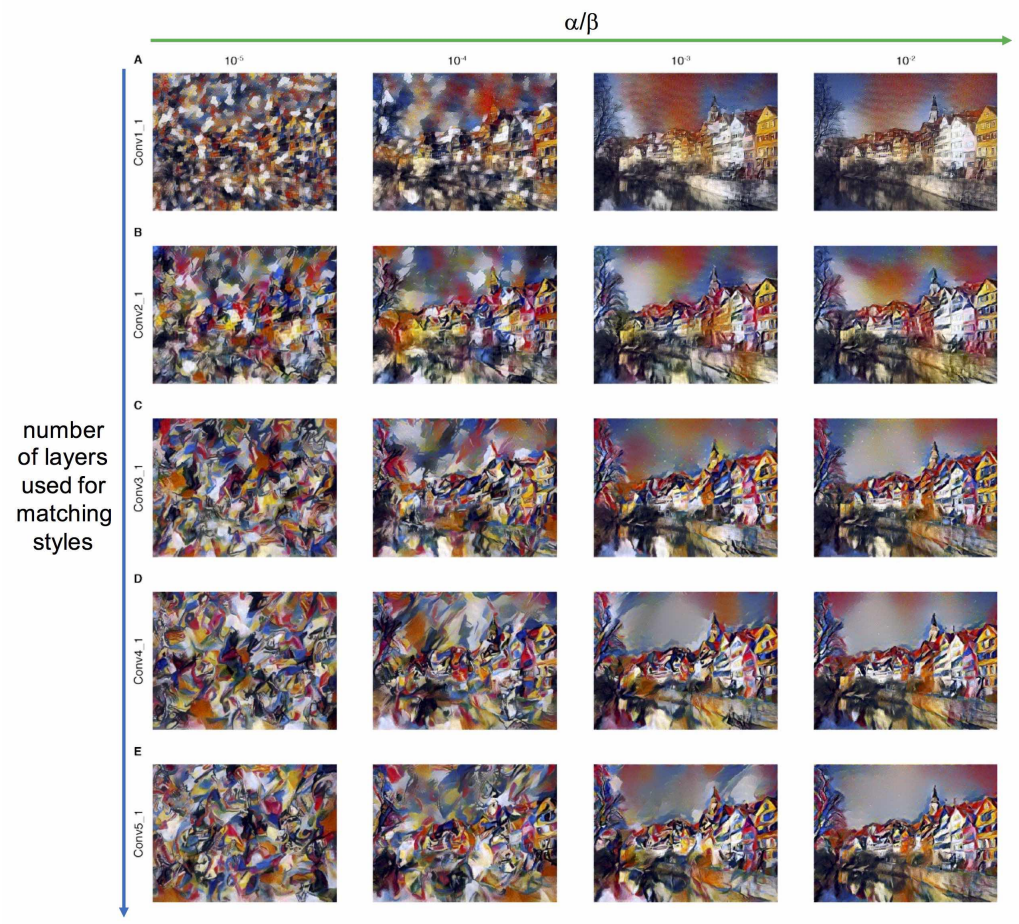

Figure 21: Style transfer variations

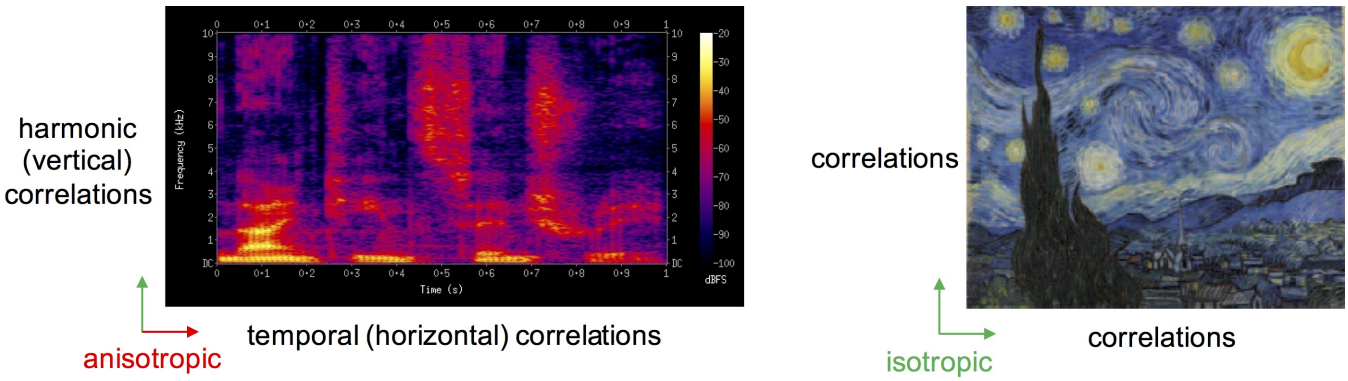

Figure 22: Anisotropic music vs isotropic image

Note that an alternative direction would be to use a *temporal series* representation, the type of memory representation that recurrent networks learn. Indeed, for music, recurrent networks are more frequently used than feedforward networks. Using pre-encoding of temporal series, e.g., such as the RNN Encoder-Decoder (see [BHP17, Section 5.6.2]) may be a path to explore.

9.5 Input Manipulation and Sampling

Another example of input manipulation, combined with sampling, thus acting both on the input and the output[28], is exemplified below.

9.5.1 Example: C-RBM Polyphony Symbolic Generation System

In the system presented by Lattner *et al.* in [LGW16], 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 in imposing through *constraints* some more *global structure* (form, e.g., AABA, as well as tonality), seen as as a *structural template* inspired from the reference of an existing musical piece, onto the generated new piece. This is called *structure imposition*. These constraints, about 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 pitches (corresponding to MIDI pitch numbers 28 to 92). The corpus is Mozart sonatas. Each piece is transposed into all possible keys in order to have sufficient training data for all possible keys. (This also helps at reducing sparsity in the training data). The architecture is a *convolutional restricted Boltzmann Machine* (C-RBM), i.e. a RBM with convolution[29], with 512*64 = 32,768 input nodes and 2,048 hidden units. Units are not Boolean, as for standard RBM, and are continuous, as for neural networks. Convolution is only performed on the time dimension, in order to model temporally invariant motives, but not pitch invariant motives (there are correlations between notes over the whole pitch range), which would break the notion of tonality[30].

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

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

- *self-similarity* – The purpose is to specify a *global structure* (e.g. AABA) in the generated music piece. This is modeled by minimizing the distance (mean squared error) between the self-similarity matrixes of 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 so-called key profiles of the reference (i.e. paradigmatic relative pitch-class strengths for specific scales and modes) [Tem11], in practice with the two major and minor modes. They are repeated in the time and in the pitch dimension of the piano roll matrix, with a modulo octave shift in the pitch dimension. The resulting key estimation vectors are combined (see the article for more details) to obtain a combined key estimation vector. In the same way as for self-similarity, a distance between the target and the intermediate solution key estimation is minimized;

[28] Interestingly, *the input is actually equal to the output*, because the architecture used is a RBM (restricted Boltzmann machine, see Section 5.2.1), where the *visible layer* acts *both* as *input* and *output*.

[29] See, e.g., [GBC16, Chapter 9] or [BHP17, Section 5.3.8] for introduction or/and details about convolutional architectures.

[30] As the authors state [LGW16], "Tonality is another very important higher order property in music. It describes perceived tonal relations between notes and chords. This information can be used to, for example, determine the key of a piece or a musical section. A key is characterized by the distribution of pitch classes in the musical texture within a (temporal) window of interest. Different window lengths may lead to different key estimates, constituting a hierarchical tonal structure (on a very local level, key estimation is strongly related to chord estimation)."

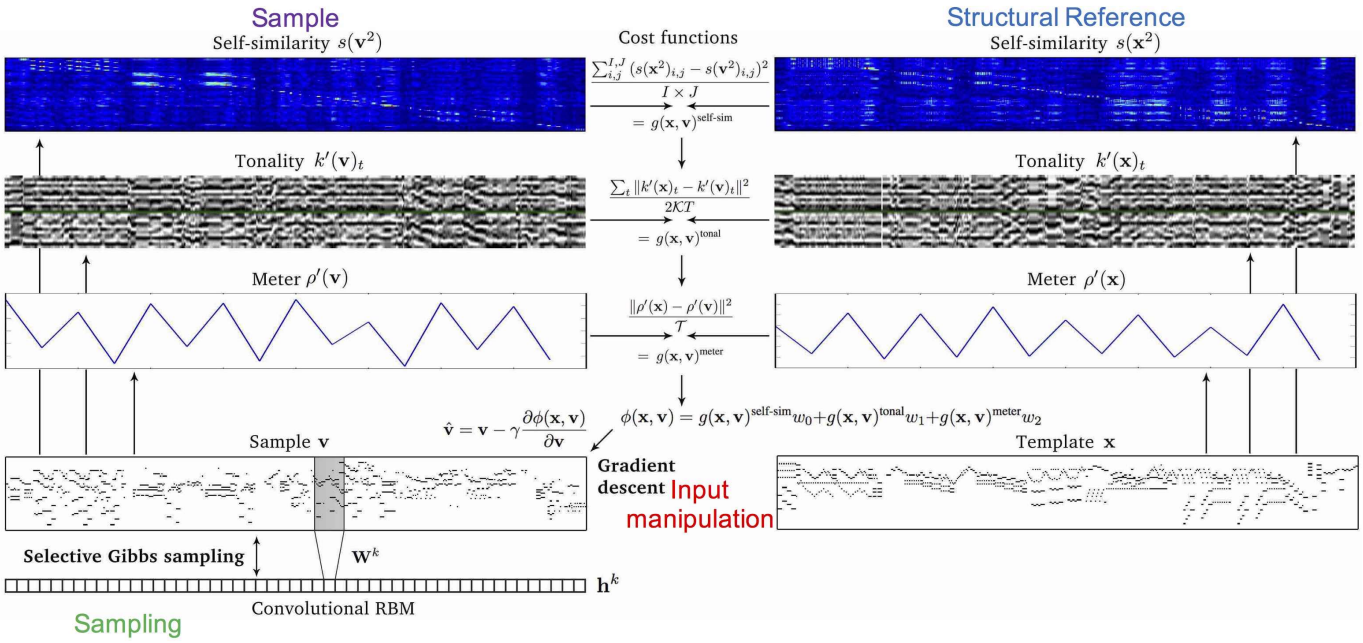

Figure 23: C-RBM Architecture

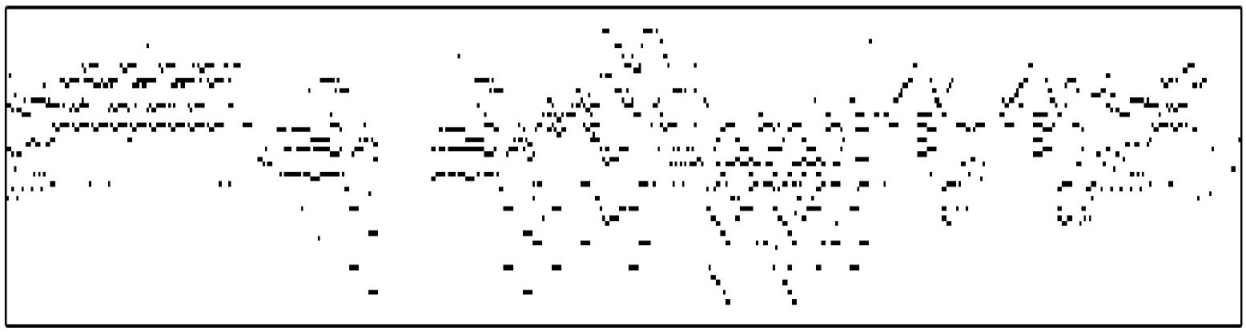

Figure 24: Piano roll sample generated by C-RBM

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

Generation is performed via *Constrained Sampling* (CS), a mechanism to restrict the set of possible solutions in the sampling process according to some pre-defined constraints. The principle of the process (illustrated at Figure 23) is as follows. At first, a sample is randomly initialized, following the standard uniform distribution. A step of Constrained Sampling (CS) is composed of n runs of Gradient Descent (GD) optimization to impose the high-level structure, followed by p runs of *selective Gibbs sampling* (SGS)[31] to selectively realign the sample onto the learnt distribution. A *simulated annealing* algorithm is applied in order to decrease exploration in relation to a decrease of variance over solutions.

Figure 24 show an example of generated sample in piano roll format. The article [LGW16] details in different figures the steps of constrained sampling.

Results are promising. One current limitation, stated by the authors, is that constraints only apply to the high-level structure. Initial attempts at imposing a low-level structure constraints were challenging because, as constraints are never purely content-invariant, when trying to transfer low-level structure, it can happen that the template piece gets exactly reconstructed in the GD phase. Therefore, creating constraints for low-level structure would have to be accompanied by increasing their content invariance. Another issue is convergence and satisfaction of the constraints. As discussed by the authors, their approach is not exact, as for instance by the Markov constraints approach (for Markov chains) proposed in [PR11].

[31] Selective Gibbs sampling (SGS) is an authors' variant of *Gibbs sampling* (GS).

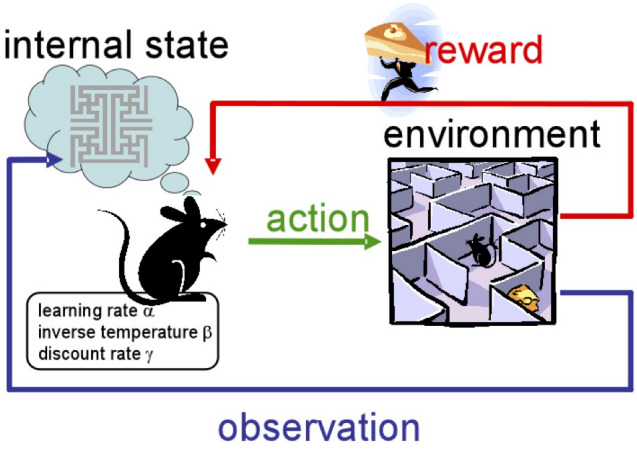

Figure 25: Reinforcement learning (Conceptual model) – Reproduced from [DU05]

9.6 Reinforcement

The idea of the *reinforcement strategy* is to *reformulate* the generation of musical content as a *reinforcement learning problem*, while using the output of a trained recurrent network as an *objective* and adding user defined constraints, e.g., some tonality rules according to music theory, as an *additional objective*.

9.6.1 Reinforcement Learning

Let us remind the basic concepts of reinforcement learning, illustrated at Figure 25:

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

There are many approaches and algorithms for reinforcement learning (for a more detailed presentation, please refer to, e.g., [KLM96]). Among them, *Q-learning* [HD92] turned out being a relatively simple and efficient method, thus widely used. The name comes from the objective to learn (estimate) the *action value function* $Q^*(s, a)$, which represents the expected gain for a given pair $< s, a >$, where s is a state and a an action, and then using it to choose actions optimally. The agent will manage a table, called the *Q-Table*, with values corresponding to all possible pairs. As long as the agent incrementally explores the environment, the table is updated with hopefully increasingly accurate expected values.

A recent combination of reinforcement learning (more specifically Q-learning) and deep learning, named *deep reinforcement learning* has been proposed [MKS+13] in order to make learning more efficient. As the Q-Table could be huge, the idea is to use a deep neural network in order to approximate the expected values of the Q-Table through the learning of many replayed experiences. A further optimization, named *Double Q-learning* [vHGS15] *decouples* the *action selection* from the *evaluation*, in order to avoid value overestimation. The task of the Target Q-Network is to estimate the gain (Q), while the task of the Q-Network is to select next action.

9.7 Reinforcement-based Control

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

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

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

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

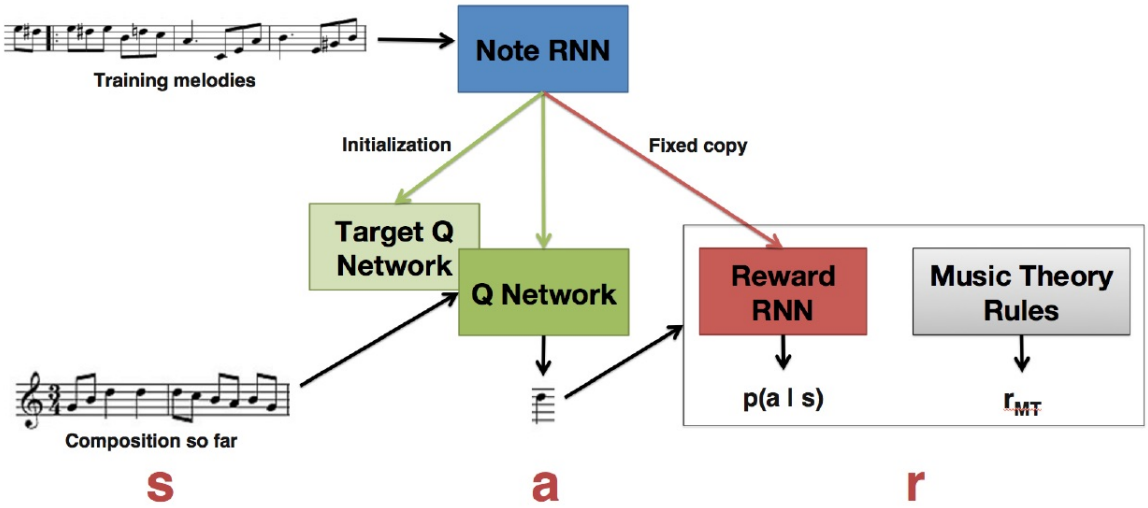

Figure 26: RL-Tuner architecture

9.7.1 Example: RL-Tuner Melody Symbolic Generation System

The *reinforcement strategy* has been pioneered by the *RL-Tuner* architecture [JGTE16] by Jaques *et al.* The architecture, illustrated at Figure 26, consists in two deep Q network reinforcement learning architectures[32] and two *recurrent network* (RNN) architectures.

- The initial RNN (named *Note RNN*, as its objective is to predict and generate next note) is trained on the dataset of melodies, in a similar way to the experiments by Eck and Schmidhuber on using a RNN to generate melodies following the *iterative feedforward* strategy (see Section 5.3.3);
- Then, a fixed copy is made, named the *Reward RNN*, and will be used for the reinforcement learning (to generate the dataset-based reward);
- The *Q Network* architecture task is to learn to select next note (next action a) from the generated (partial) melody so far (current state a);
- The Q Network is trained in parallel to the other Q Network, named the *Target Q Network*, which estimates the value of the gain and which has been initialized from what the Note RNN has learnt;
- The Q Network's reward r combines two objectives: adherence to *what has been learnt* and adherence to *user-defined constraints*, as defined in Section 9.7.

Therefore, the reinforcement strategy allows to combine arbitrary user given control with a style learnt by the recurrent network. Note that in the general model of reinforcement learning, the reward is not predefined: the agent does not know beforehand the model of the environment and the reward, thus it needs to balance between exploring to learn more and exploit in order to improve its gain – the *exploration exploitation dilemna*. In the case of RL-Tuner, the reward is pre-defined and dual: handcrafted for the music theory rules and learnt from the dataset by a RNN for the musical style. Therefore, there is an opportunity to insert *arbitrary* kind of control, including *incremental* control by the *user* (see Section 12), although a feedback at the granularity of each note generated may be too demanding and moreover not very accurate[33]. We will discuss in Section 13 the issue of learning from user feedback.

9.8 Unit Selection

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

This strategy, as opposed to most of the other ones, that are *bottom-up*, is *top-down*, as it starts with a structure and fills it. An example is described in Section 9.8.1.

[32] An implementation of the Q-learning reinforcement learning strategy through deep learning architectures [vHGS15].
[33] 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."

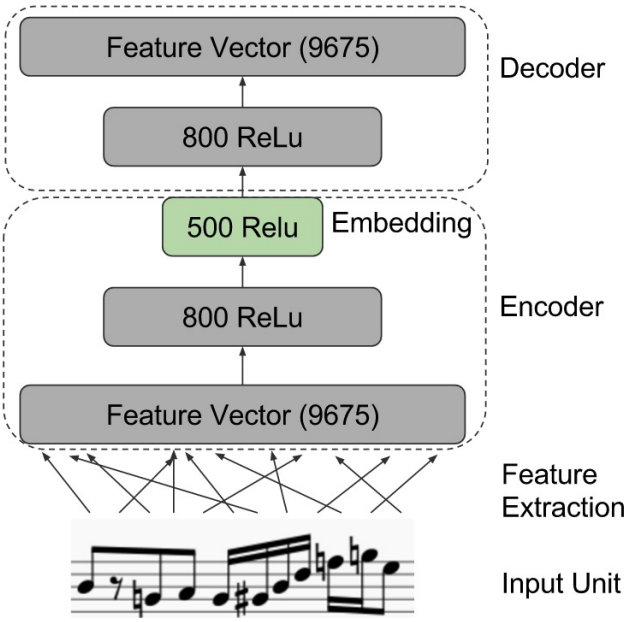

Figure 27: Stacked autoencoders architecture

9.8.1 Example: Unit Selection and Concatenation Melody Generation System

This strategy has been pioneered by Bretan *et al.* [BWH16]. The idea is to generate music from a concatenation of music units, queried from a database. The key process here is *unit selection*, which is based both on two criteria: *semantic relevance* and *concatenation cost*. The idea of unit selection to generate sequences is actually inspired by a technique commonly used in text-to-speech (TTS) systems.

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

The architecture includes one *autoencoder* and two LSTM *recurrent networks*. The first step is feature extraction. 10 features, manually handcrafted, are considered, following a *bag-of-words* (BOW) approach (see [GBC16, Section 12.4.3.3] or [BHP17, Section 4.4.4]), e.g., counts of a certain pitch class, counts of a certain pitch class rhythm tuple, if first note is tied to previous measure... (see details in [BWH16]). This results in 9,675 actual features. Note that all of them have integer values (`0` or `1` for the Boolean features and rests are represented using a negative pitch value). Therefore each unit is described (indexed) as a 9,675 size feature vector.

The autoencoder used is a double *stacked autoencoders* architecture, as described at Figure 27. Once trained, the usual self-taught way for autoencoders, on the set of feature vectors, it will become a new feature extractor encoding a 9,675 size feature vector into a 500 size vector *embedding*.

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

- *successor semantic relevance* – Based on a model of transition between units, as learnt by a LSTM recurrent network. In other words, that relevance is based on the distance to the (ideal) next unit as predicted by the model; This LSTM architecture has 2 hidden layers, each with 128 units. Input and output layers have 512 units (corresponding to the format of the embedding);
- *concatenation cost* – Based on another model of transition[34], this time between the last note of the unit and the first note of the next unit, as learnt by another LSTM recurrent network. That second LSTM architecture has hidden layers and its input and output layers have about 3,000 units, as this corresponds to the one-hot encoding of the characterization of an individual note (as defined by its pitch and its duration).

The combination of the two criteria (illustrated at Figure 28) is handled by a heuristic-based dynamic ranking process:

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

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

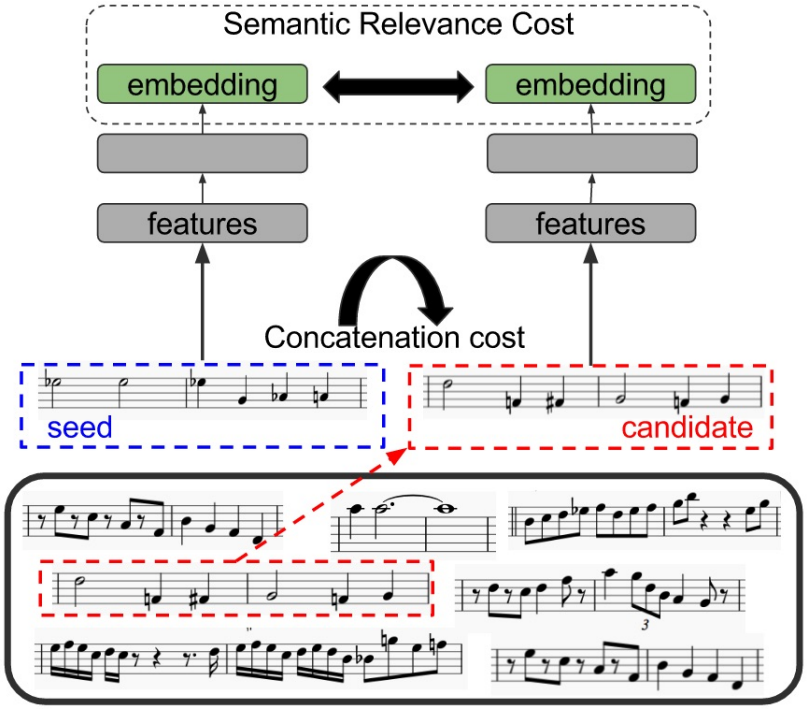

Figure 28: Unit selection based on semantic cost

4. select the unit with the highest combined rank.

The process is iterated in order to generate an arbitrary length melody. This iterative generation of a melody may at first look like an *iterative feedforward* generation from a recurrent network (as in Section 5.3.3), but with two important differences: 1) the embedding of the next musical unit item is computed through a multi-criteria ranking algorithm, 2) the actual unit is queried from a database with the embedding as the index.

Initial external human evaluation has been conducted by the authors. They found out that music generated using units of one or two measure durations tended to be ranked higher according to naturalness and likeability than units of four measures or note-level generation, with an ideal unit length appearing to be one measure.

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.

10 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*[35]. One approach is *a posteriori*, by ensuring that the generated music is not too similar (e.g., in not having recopied a significant amount of notes of a melody) to an existing piece of music. Therefore existing tools to detect similarities in texts may be used. Another approach, more systematic but even more challenging, is *a priori*, by ensuring that the music generated will not recopy a given portion of music from the training corpus[36]. A solution for music generation from Markov chains has been proposed [PRP14] but there is none yet solution for generation from deep architectures.

Let us now analyze some recent directions for favoring originality of the generated musical content.

10.1 Conditioning

10.1.1 Example: MidiNet Melody Generation System

In their description of MidiNet [YCY17] (see Section 9.3.3), the authors discuss two methods to control creativity:

- by restricting the conditioning by inserting the conditioning data only in the intermediate convolution layers of the generator architecture (see 10.2.1);
- by decreasing the values of the two control parameters of feature matching regularization, in order to less enforce the distributions of real and generated data to be close.

[35] On this issue, see a recent paper [Del17].

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

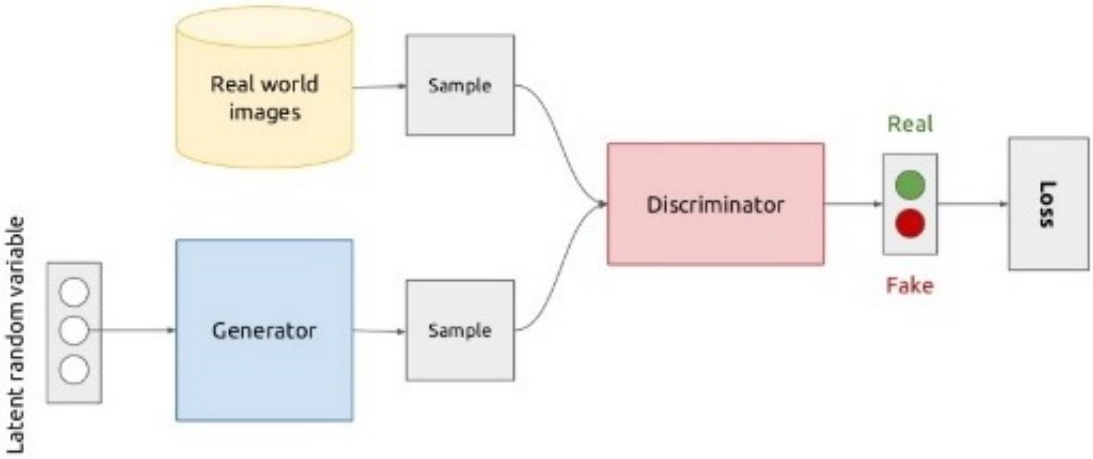

Figure 29: Generative adversarial networks (GAN) architecture – Reproduced from [Ahn17]

These experiments are interesting but they remain at the level of some *ad hoc* tuning of some hyper-parameters of the architecture.

10.2 Creative Adversarial Networks

Another more systematic and conceptual direction is the concept of *creative adversarial networks (CAN)* proposed by El Gammal *et al.* [ELEM17], as an extension of generative adversarial networks (GAN) architecture. Therefore, let us at first remind the principles of Generative adversarial networks.

10.2.1 Generative Adversarial Networks (GAN)

A significant conceptual and technical innovation was introduced in 2014 by Goodfellow *et al.* with the concept of *generative adversarial networks* (GAN) [GPAM+14]. The conceptual idea is to train simultaneously two networks[37] (see at Figure 29):

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

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

In the minimax equation (Equation 1):

$$\min_G \max_D V(G,D) = log(D(x)) + log(1 - D(G(z))) \quad (1)$$

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

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

One of the initial motivation for GAN is for classification tasks to better prevent adversaries to manipulate deep networks to force misclassification of inputs (this vulnerability is analyzed in detail in [SZS+14]). But it also improves

[37] In the original version, two feedforward networks are used. But other networks may be used, e.g., recurrent networks in the C-RNN-GAN architecture [Mog16] or convolutional feedforward networks in the MidiNet architecture (see Section 9.3.3).

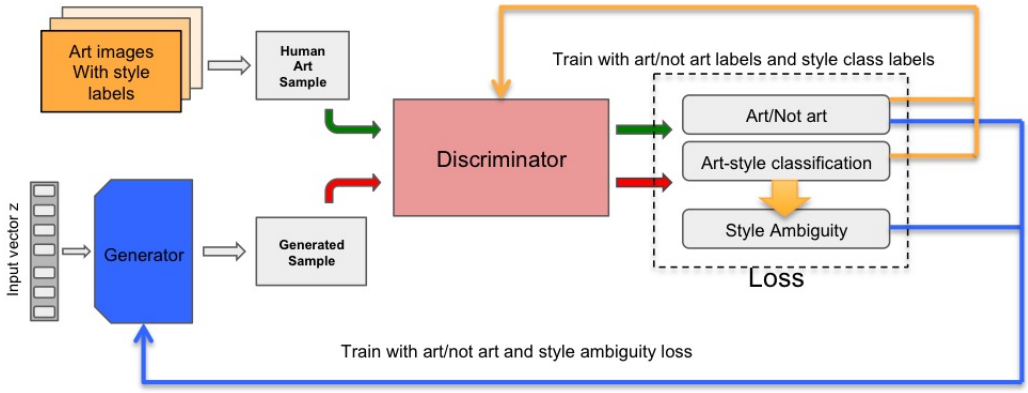

Figure 30: Creative adversarial networks (CAN) architecture

the generation of samples hard to distinguish from the actual corpus examples, thus addresses the generation task (which is our focus here).

Note that training based on a minimax objective is known to be challenging to optimize [YYSL16], with risks of non converging oscillations. Thus, careful selection of the model and its hyperparameters are important [GBC16, page 701]. There are also some improved techniques, such as *feature matching* and other ones, to improve training [SGZ+16].

To generate music, random noise is used as an input to the generator G, whose goal is to transform random noises into the objective, e.g., melodies. An example of the use of GAN for generating music (melodies) is the MidiNet system, described in Section 9.3.3.

10.2.2 Creative Adversarial Networks Painting Generation System

Elgammal *et al.* propose in [ELEM17] to address the issue of *creativity* by extending a generative adversarial networks architecture (GAN) architecture into a *Creative adversarial networks (CAN)* architecture to "generate art by learning about styles and deviating from style norms."

Their assumption is that in a standard GAN architecture, the generator objective is to generate images that fool the discriminator and, as a consequence, the generator is trained to be *emulative* but not *creative*. In the proposed creative adversarial networks (CAN) (illustrated at Figure 30), the generator receives from the discriminator not just one but two *signals*. The first signal is analog to the case of the standard GAN (see Equation 1) and specifies how the discriminator believes that the generated item comes from the training dataset of real art pieces. The second signal is about how easily the discriminator can *classify* the generated item into *established styles*. If there is some strong ambiguity (i.e., the various classes are *equiprobable*), this means that the generated item is difficult to fit within the existing art styles. These two signals are thus contradictory forces and push the generator to explore the space for generating items that are at the same time close to the distribution of existing art pieces and with some style originality.

Experiments have been done with a dataset of paintings from a WikiArt dataset 3 [Wik17]. This collection has images of 81,449 paintings from 1,119 artists ranging from the 15th century to 20th century. It has been tagged with 25 possible painting styles (e.g., cubism, fauvism, high-renaissance, impressionism, pop-art, realism...). Some examples of generated images are shown at Figure 31.

Note that, as the authors discuss, the generated images indeed are not recognized like traditional art, in terms of standard genres (portrait, landscapes, religious paintings, still life, etc.), as shown by a preliminary external evaluation and also a preliminary analysis of their approach. Their approach is actually relatively simple to implement and interesting to analyze further, but it assumes the existence of a prior style classification (it will be interesting to experiment with other types of classification) and it also 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 experiment with other types of classification.

As for the style transfer approach (described in Section 9.4.3), a musical style seems less easy to capture just as correlations of neuron activations or as music classification descriptors but, giving the large amount of techniques on classification and retrieval of music (see, e.g., the series of conferences of the International Society of Music Information Retrieval (ISMIR) [ISM17]), future experiments appear promising.

11 Incrementality

A straightforward use of deep architectures for generation leads to one-shot generation of a musical content as a whole, or as a temporal series, in the case of recurrent networks. This is a strong limitation, if we compare this to the way a human composer creates and generates music, in most cases very incremental, though successive refinements of various parts.

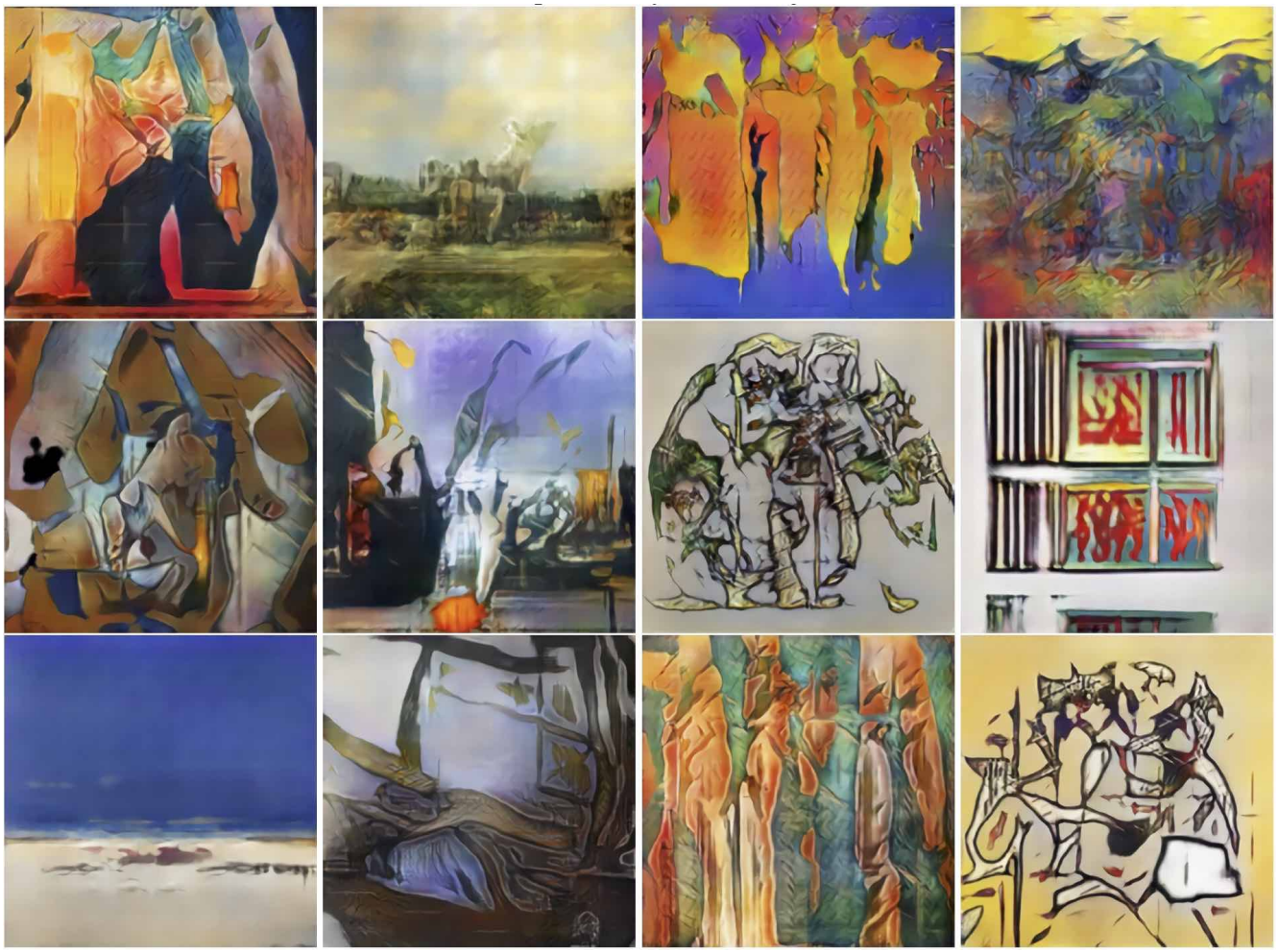

Figure 31: Examples of images generated by CAN

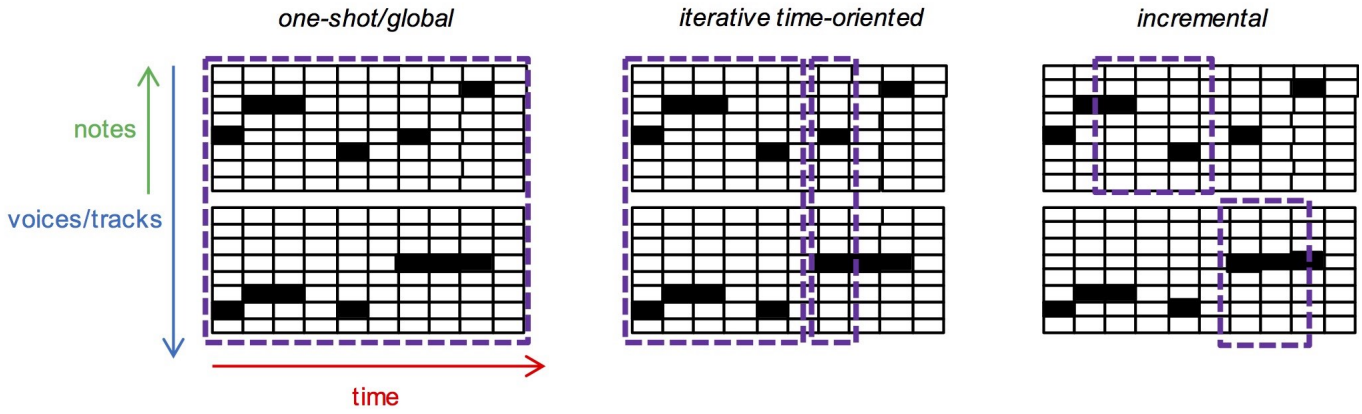

Figure 32: Generation – Time steps and voices – Three strategies

11.1 Strategies

Let us review how notes of a melody are instantiated during generation. There are three main strategies (illustrated at Figure 32):

- *One-shot global generation* – In this strategy, a *feedforward architecture* is used with a *global input* (and output) representation including *all time steps*. An example is DeepHear [Sun17], analyzed in Section 5.1.4;
- *Iterative time-oriented generation* – In this strategy, a *recurrent architecture* is used with a *time slice input* (and output) representation including a *single time step*. An example is in [ES02], analyzed in Section 4;
- *Incremental variable instantiation* – In this strategy, a *feedforward architecture* is used with a *global input* (and output) representation including *all time steps*. But, as opposed to *one-shot global generation* strategy, generation is not done in a global one step, but *incrementally* by instantiating and refining values of variables (the values corresponding to the output nodes, corresponding to the specification of the generated musical content, e.g., all pitch values for all notes of a melody). Thus, it is possible to generate or to *regenerate* only an *arbitrary part* (slice) of the intended musical content, for a specific *time interval* between two time steps or/and for a specific *subset of voices/tracks*, without the need for regenerating the whole content. An example is described in Section 11.2.

In the first strategy, using feedforward, one could imagine selecting only the desired slice from the regenerated content and to "copy/paste" it into the previous generated content. The limitation of such *ad hoc* approach is that there is no guarantee that the old and new parts will be consistent.

11.2 Example: DeepBach Chorale Symbolic Music Generation System

Hadjeres *et al.* have proposed the DeepBach architecture[38] [HPN17] for generation of Bach chorales. The architecture, shown at Figure 33, is *compound*, combining two *recurrent* (LSTM) and two *feedforward* networks. As opposed to standard use of recurrent networks, where a single time direction is considered, DeepBach architecture considers the two directions *forward* in time and *backwards* in time. Therefore, two recurrent networks (more precisely, LSTM, with 200 nodes) are used, one summing up past information and another summing up information coming from the future, together with a non recurrent network for notes occurring at the same time. Their three outputs are merged and passed as the input of a final feedforward neural network, with one hidden layer with 200 units. The final output activation function is *softmax*. The first 4 lines of the example data on top of the Figure 33 correspond to the 4 voices. The two bottom lines correspond to metadata (fermata and beat information). Actually this architecture is replicated 4 times, one for each voice (4 in a chorale).

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

The initial corpus is the set of J.S. Bach polyphonic chorales music [Bac85], where the composer chose various given melodies for a soprano and composed the three additional ones (for alto, tenor and bass) in a *counterpoint* manner. Contrary to other approaches, which transpose all chorales to the same key (usually in C major or A minor), the dataset is augmented by adding all chorale transpositions which fit within the vocal ranges defined by the initial corpus. This leads to a total corpus of 2,503 chorales. The vocal ranges contains less than 30 different pitches for each voice[39].

[38] The MicroBach architecture, described in Section 4.1.1, is actually a deterministic single-step feedforward major simplification of the DeepBach architecture.

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

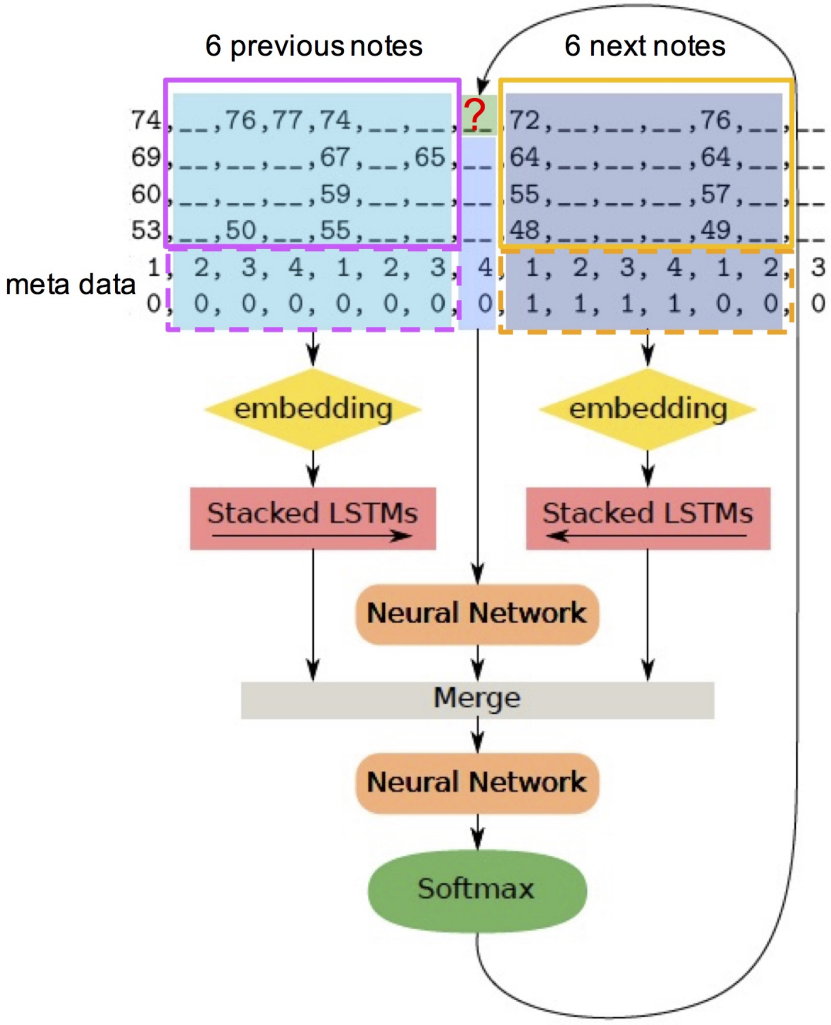

Figure 33: DeepBach architecture

The choice of the representation in DeepBach has some specificities. Rhythm is modeled by simply adding a *hold symbol* (_) encoding whether or not the preceding note is held to the list of existing notes. This representation is unambiguous, compact and well-suited to the sampling method[40]. Another specifity of DeepBach is that the representation consists in encoding notes using their *real names* and not their MIDI pitches (e.g., $F\#$ is different from a Gb)[41]. Also the fermata symbol for Bach Chorales is explicitly considered as it helps producing structure and coherent phrases. Details about the representation used can be found in [HPN17].

Training, as well as generation, is not done in the conventional way for neural networks. The objective is to predict the value of current note for a a given voice (shown with a red "?", on top center of Figure 33), using as information surrounding contextual notes (and their associated metadata), more precisely: the current notes for the 3 other voices (the thin rectangle in light blue, in the center), the 6 previous notes (the rectangle in light turquoise blue, on the left) for all voices, the 6 future notes (the rectangle in light magenta, on the right) for all voices. The training set is formed on-line by repeatedly randomly selecting a note in a voice from an example of the corpus and its surrounding context (as defined above).

Generation is done (the algorithm is shown at Figure 34) by sampling, using a *pseudo-Gibbs sampling* process, analog but computationally simpler than *Gibbs sampling* procedure[42], to produce a set of values (each note) of a polyphony, following the distribution that the network has learnt.

An example of chorale generated is shown at Figure 35. As opposed to many experiments, a systematic evaluation has been conducted (with more than 1,200 human subjects, from experts to novices, via a questionnaire on the Web) and results are very positive, with a significant difficulty to discriminate between chorales composed by Bach and generated by DeepBach.

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

[41]This means, as opposed to most of systems, considering *enharmony*.

[42]The difference, based on the non assumption of compatibility of conditional distributions, as well as the algorithm, are detailed and discussed in [HPN17].

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 $pi(V_i^t|V_{\setminus i,t}, \theta_i)$
end for

Figure 34: DeepBach incremental generation/sampling algorithm

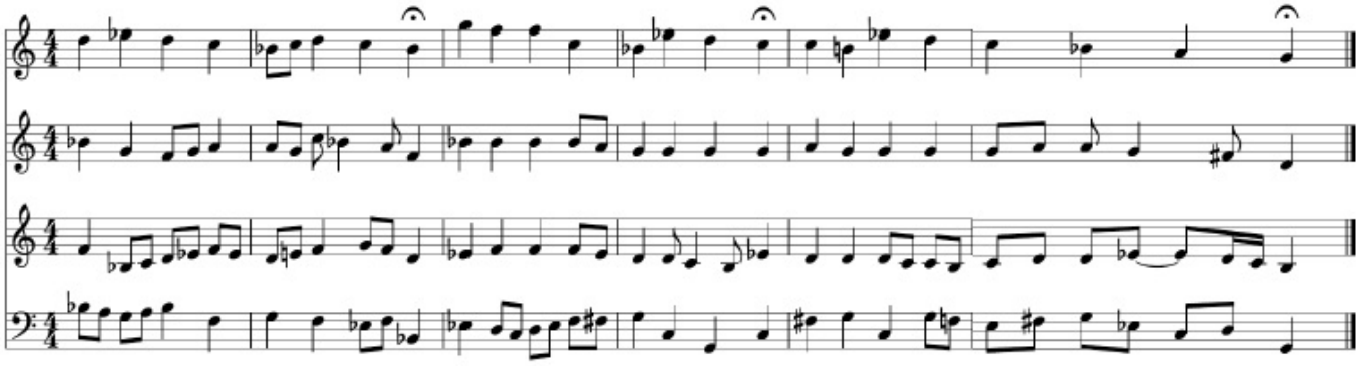

Figure 35: Example of chorale generated by DeepBach

12 Interactivity

An important issue is that for most of current systems, generation of musical content is an *automated* and *autonomous* process. *Interactivity* with the *human user* is fundamental to have a companion system and not an automated system, to help humans in their musical tasks (composition, counterpoint, harmonization, arranging, etc.) in an incremental and interactive manner (such as, e.g., showed by the FlowComposer prototype [PRP16] for Markov chain based generation). Some examples of such partially interactive incremental systems based on deep architectures are deepAutoController (Section 5.1.5) and DeepBach (see Section 11.2).

12.1 #1 Example: DeepAutoController Audio Music Generation System

The deepAutoController system [SC14], by Sarroff and Casey, uses stacked autoencoders to generate audio described in [SC14] (see Section 5.1.5) (see some analysis in [BHP17, Section 7.1.3.2]). It provides a user interface to interactively control the generation by, e.g., selecting a given input (to be feedforwarded into the decoders), or generate a random input, controlling (by scaling or muting) the activation of a given unit.

12.2 #2 Example: DeepBach Chorale Symbolic Music Generation System

The user interface of DeepBach (see Section 11.2) is made possible by the incremental nature of the generation (see Section 12). It is implemented as a plugin for the MuseScore music editor (see Figure 36). 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 11).

Let us finally mention, at the crossing between *control* and *interactivity*, the interesting discussion by Morris *et al.* in [MSB08], on the issue of what *control parameters* (for music generation by a Markov-chain trained model) are to be *exposed* at the human user level. Some examples of user level control parameters they have experimented with are: major vs minor, follow melody vs follow chords, locking a feature (e.g., a chord).

13 Adaptability

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

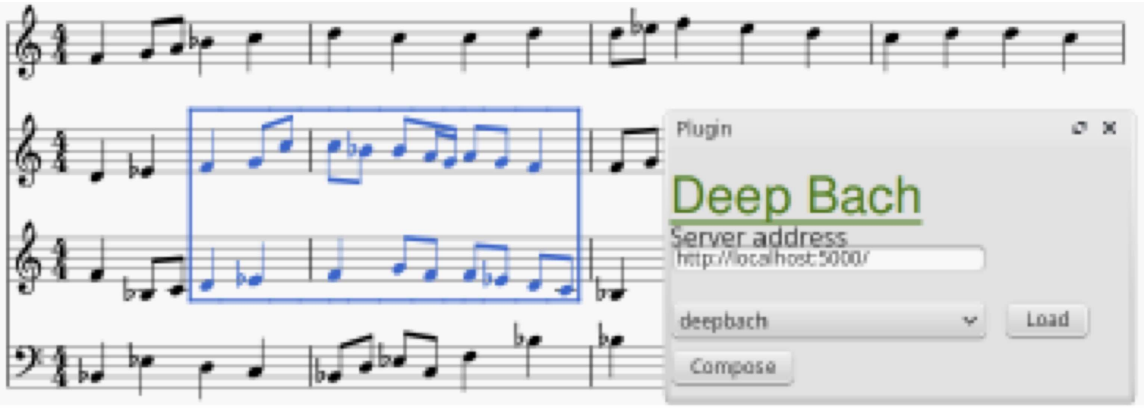

Figure 36: DeepBach user interface

Several approaches are possible. The more straightforward approach, considering the nature of neural networks and supervised learning, would be to add the new generated musical piece to the training set and eventually[43] retrain the network. (This could be done in the background). This would reinforce the amount of positive examples and gradually update the learnt model and as a consequence future generations. However, there is no guarantee that the overall generation quality would improve. This could also lead for the model to *overfit* and loose some generalization. Moreover, there is no direct possibility of *negative feedback*, as one cannot remove from the dataset a bad example generated, because there is almost no chance that it was already present in the dataset.

At the crossing between *adaptability* and *interactivity*, an interesting approach is that of *interactive machine learning* for music generation, as discussed by Fiebrink and Caramiaux [FC16]. They report on an experience with a toolkit they designed, named Wekinator, to allow users interactively modify the training examples. They for instance argue 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 performance (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 to go back to the issue of control, via, e.g., constrained sampling strategy, input manipulation strategy and obviously reinforcement strategy, as the control objective and parameters should be *adaptive*. The RL-Tuner framework (see Section 9.7.1) is an interesting step towards this. Although, the initial motivation for RL-Tuner is to introduce musical constraints onto the generation, by encapsulating them into an additional reward, this approach could also be used to introduce user feedback as an additional reward (e.g., as in the following RL-based generation system [GV10]). Although such experiments are preliminary, they show the way for future exploration and research.

14 Explainability

A common criticism of *sub-symbolic* approaches of Artificial Intelligence (AI), such as neural networks and deep learning, often considered as *black boxes* is the issue of its explainability [Cas16]. This is indeed a real issue, to be able to explain what and how a deep learning system has learned and how and why it generates a specific content. Being able to better understand this issue also indirectly addresses the issue of how to control the generation of a system.

14.1 Example: BachBot Chorale Symbolic Music Generation System

Although preliminary, some interesting study conducted with the BachBot system [Lia16] (and analyzed in [BHP17, Section 7.3.1.3]) is about the analysis of the specialization of some of the units (neurons) of the network, through some correlation analysis with some specific motives and progressions, see Figure 37 (see more details in [Lia16, Chapter 5]).

Some findings are, e.g.:

- Layer 1/Neuron 64 and Layer 1/Neuron 138 seem to detect (specifically) perfect cadence with root position chords in the tonic key;
- Layer 1/Neuron 151 seems to detect a minor cadences that end phrases 2 and 4.

One may imagine using techniques like *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, e.g., [RS04]), and apply and adapt them to the study of activation correlation in deep architectures.

[43] immediately or after some time or some amount of new feedbacks, as for a *minibatch* (see [GBC16, Section 8.1.3]).

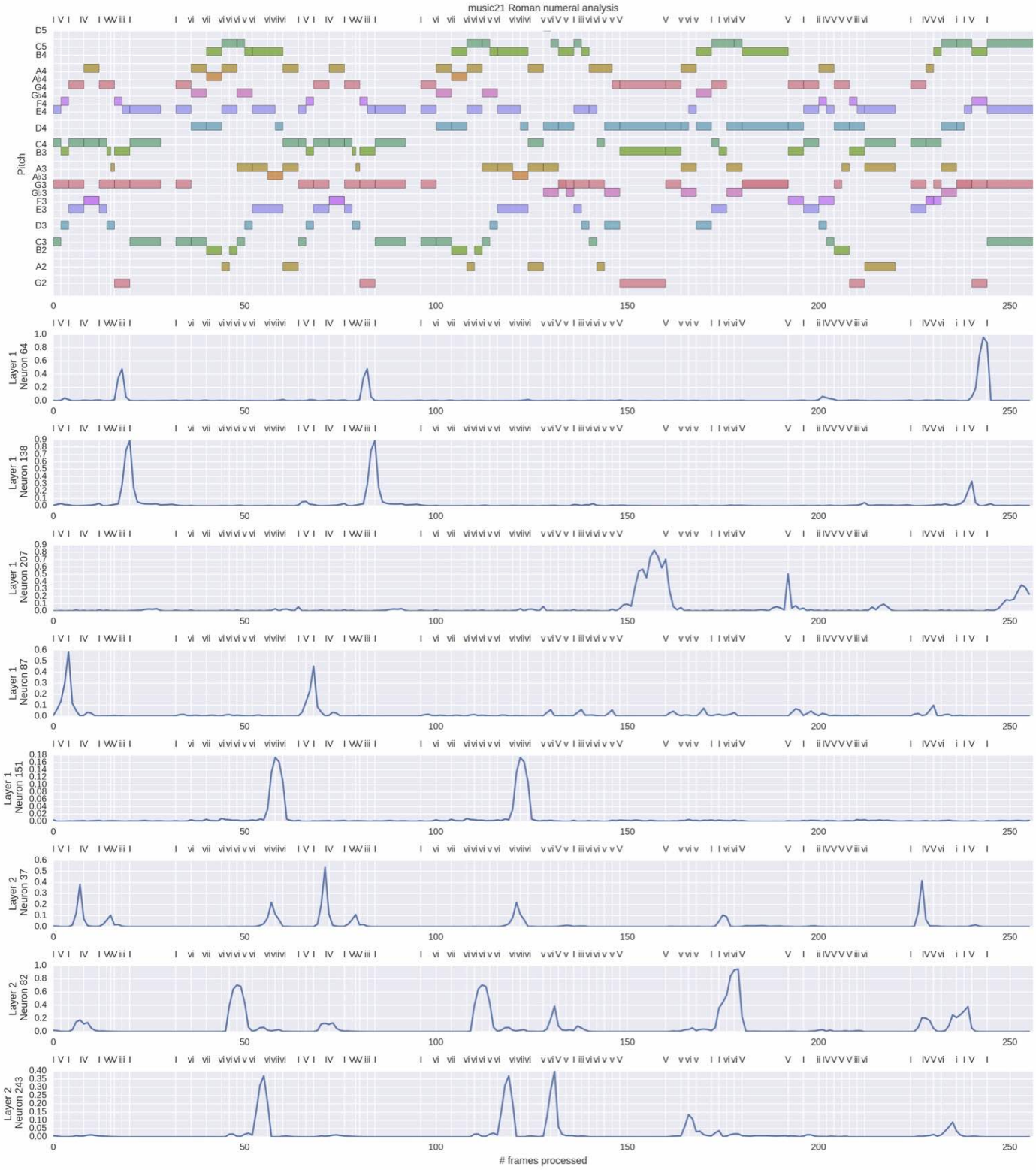

Figure 37: Correlation analysis of BachBot layer/unit activation

15 Discussion

We can observe that the various limitations and challenges that we have analyzed may be partially dependent and furthermore conflicting, i.e., resolving one may damper another one. For instance, using the *incremental variable instantiation* in order to achieve *incrementality* implies a *fixed length output*, as opposed to the *iterative time-oriented generation* on a recurrent network, which has as one of its advantage the generation of sequences (of notes) of *arbitrary length* [BHP17, Section 6.1.2]. Another example is the fact that the *iterative feedforward* strategy (see Section 5.3) resolves the limitation of *variable length generation* (see Section 6), but at the cost of impeding solutions for the *incrementality* limitation (see Section 11). Thus, concerning multi-criteria decision and optimization, it is difficult to win for all respects and selection of architectures and strategies depends on preferences among issues.

We will later see (in Section 11) that the *iterative feedforward* strategy also hampers the possibility of *incremental generation*. In other words, there is no complete solution for all challenges as it is difficult to win in all aspects.

16 Conclusion

The use of deep learning architectures and techniques for the generation of music (as well as other artistic content) is a growing area of research. Objectives like control, originality and interactivity lead us to challenge some of the principles of deep learning. As there are no easy networks internals entry points (hooks) on which attach control, strategies have to rely on externals (input, output, or complete encapsulation) of the architecture. In order to make progress, we need to better understand the merits and limits of current and future proposals. We hope that this analysis may be a contributing input in that direction.

Acknowledgements

This paper is based on the first half of the "Machine-Learning for Symbolic Music Generation" tutorial given at the 18th International Society for Music Information Retrieval Conference (ISMIR'2017). We thank the audience for their feedback. We thank Gaëtan Hadjeres and Pierre Roy for discussions on this topic.

References

[Ahn17] Nam Hyuk Ahn. Generative adversarial network, January 2017. https://www.slideshare.net/nmhkahn/generative-adversarial-network-laplacian-pyramid-gan.

[Bac85] Johann Sebastian Bach. *389 Chorales (Choral-Gesange)*. Alfred Publishing Company, 1985.

[BHP17] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation – A survey, September 2017. arXiv:1709.01620.

[BL15] Nicolas Boulanger-Lewandowski. Chapter 14th – Modeling and generating sequences of polyphonic music with the RNN-RBM. In *DeepLearning Tutorial – Release 0.1*, pages 149–158. LISA lab, University of Montréal, September 2015. http://deeplearning.net/tutorial/deeplearning.pdf.

[BLBV12] 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.

[BWH16] Mason Bretan, Gil Weinberg, and Larry Heck. A unit selection methodology for music generation using deep neural networks, December 2016. arXiv:1612.03789v1.

[Cas16] Davide Castelvecchi. The black box of AI. *Nature*, 538:20–23, October 2016.

[CB98] Yann Le Cun and Yoshua Bengio. Convolutional networks for images, speech, and time-series. In *The handbook of brain theory and neural networks*, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.

[Cop00] David Cope. *The Algorithmic Composer*. A-R Editions, 2000.

[Del17] Jean-Marc Deltorn. Deep creations: Intellectual property and the automata. *Frontiers in Digital Humanities*, 4, February 2017. Article 3.

[DU05] Kenji Doya and Eiji Uchibe. The Cyber Rodent project: Exploration of adaptive mechanisms for self-preservation and self-reproduction. *Adaptive Behavior*, (2):149–160, 2005.

[Ebc88] Kemal Ebcioglu. An expert system for harmonizing four-part chorales. *Computer Music Journal*, 12(3):43–51, 1988.

[EBC+10] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, and Pascal Vincent. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, (11):625–660, 2010.

[ELEM17] 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.

[ES02] Douglas Eck and Jürgen Schmidhuber. A first look at music composition using LSTM recurrent neural networks. Technical report, IDSIA/USI-SUPSI, Manno, Switzerland, 2002. Technical Report No. IDSIA-07-02.

[FC16] Rebecca Fiebrink and Baptiste Caramiaux. The machine learning algorithm as creative musical tool, November 2016. arXiv:1611.00379v1.

[FYR16] Davis Foote, Daylen Yang, and Mostafa Rohaninejad. Audio style transfer – Do androids dream of electric beats?, December 2016. https://audiostyletransfer.wordpress.com.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, September 2015. arXiv:1508.06576v2.

[GPAM+14] 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.

[Gra14] Alex Graves. Generating sequences with recurrent neural networks, June 2014. arXiv:1308.0850v5.

[GV10] Sylvain Le Groux and Paul F.M.J. Verschure. Adaptive music generation by reinforcement learning of musical tension. In *Proceedings of the 7th Sound and Music Computing Conference (SMC'2010)*, Barcelona, Spain, 2010.

[HCC17] Dorien Herremans, Ching-Hua Chuan, and Elaine Chew. A functional taxonomy of music generation systems. *ACM Computing Surveys*, 50(5), September 2017.

[HD92] Christopher J. C. H. and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, August 2002.

[HOT06] 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.

[HPN17] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. DeepBach: a steerable model for Bach chorales generation, June 2017. arXiv:1612.01010v2.

[HS86] 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.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[HS06] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[ISM17] ISMIR. International Society for Music Information Retrieval Conference(s) (Proceedings), Accessed on 23/08/2017. http://dblp.uni-trier.de/db/conf/ismir/.

[JGTE16] Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning, November 2016. arXiv:1611.02796.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, (4):237–285, 1996.

[Lam16] 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.

[LGW16] Stefan Lattner, Maarten Grachten, and Gerhard Widmer. Imposing higher-level structure in polyphonic music generation using convolutional restricted Boltzmann machines and constraints, December 2016. arXiv:1612.04742v2.

[Lia16] 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.

[LRM+12] 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.

[MB17] Brian McFee and Juan Pablo Bello. Structured training for large-vocabulary chord recognition. In *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR'2017)*. ISMIR, October 2017.

[MKPKK17] Dimos Makris, Maximos Kaliakatsos-Papakostas, Ioannis Karydis, and Katia Lida Kermanidis. Combining LSTM and feed forward neural networks for conditional rhythm composition. In Giacomo Boracchi, Lazaros Iliadis, Chrisina Jayne, and Aristidis Likas, editors, *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings*, pages 570–582. Springer, 2017.

[MKS+13] 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.

[Mog16] Olof Mogren. C-RNN-GAN: Continuous recurrent neural networks with adversarial training, November 2016. arXiv:1611.09904v1.

[MOT15] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Deep Dream, 2015. https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

[MSB08] 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. AAAI Press, July 2008.

[Nie09] Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer-Verlag, 2009.

[PPR17] 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.

[PR11] François Pachet and Pierre Roy. Markov constraints: Steerable generation of Markov sequences. *Constraints*, 16(2):148–172, 2011.

[PRP14] Alexandre Papadopoulos, Pierre Roy, and François Pachet. Avoiding plagiarism in Markov sequence generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2731–2737, Québec, PQ, Canada, July 2014.

[PRP16] Alexandre Papadopoulos, Pierre Roy, and François Pachet. Assisted lead sheet composition using FlowComposer. In Michel Rueher, editor, *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 769–785. Springer, 2016.

[PW99] George Papadopoulos and Geraint Wiggins. Ai methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB'1999 Symposium on Musical Creativity*, pages 110–117, April 1999.

[RS04] Dario Ringach and Robert Shapley. Reverse correlation in neurophysiology. *Cognitive Science*, 28:147–166, 2004.

[SC14] Andy M. Sarroff and Michael Casey. Musical audio synthesis using autoencoding neural nets, 2014. http://www.cs.dartmouth.edu/ sarroff/papers/sarroff2014a.pdf.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs, June 2016. arXiv:1606.03498v1.

[SSBTK16] Bob L. Sturm, João Felipe Santos, Oded Ben-Tal, and Iryna Korshunova. Music transcription modelling and composition using deep learning, April 2016. arXiv:1604.08723v1.

[Ste84] Mark Steedman. A generative grammar for Jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.

[Sun17] Felix Sun. DeepHear – Composing and harmonizing music with neural networks, Accessed on 01/09/2017. https://fephsun.github.io/2015/09/01/neural-music.html.

[SZS+14] 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.

[Tem11] David Temperley. *The Cognition of Basic Musical Structures*. MIT Press, Cambridge, MA, USA, 2011.

[UL16] Dmitry Ulyanov and Vadim Lebedev. Audio texture synthesis and style transfer, December 2016. https://dmitryulyanov.github.io/audio-texture-synthesis-and-style-transfer/.

[vdODZ+16] 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.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning, December 2015. arXiv:1509.06461v3.

[Wik17] WikiArt.org. WikiArt, Accessed on 22/08/2017. https://www.wikiart.org.

[YCY17] 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.

[YYSL16] Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2Image: Conditional image generation from visual attributes, October 2016. arXiv:1512.00570v2.