PolyPop: Polyphonic Pop Music Generation Using a TCN Architecture

Matteo Berti
Università di Bologna
matteo.berti11@studio.unibo.it

Andrea Colledan
Università di Bologna
andrea.colledan@studio.unibo.it

I. Introduction

The application of machine learning techniques to the field of creative expression has been the source of surprising breakthroughs, as well as heated controversies, especially over the past decade. If on one hand many argue that creativity is a prerogative of the human intellect and that as a consequence it inherently transcends the capabilities of machines, on the other hand recent advancements in machine learning have shown that – if not the very essence of creativity – at least many aspects of the human creative processes can be effectively modelled by machines given a large enough dataset.

Music is among the forms of artistic expression which have most aroused the interest of machine learning researchers, along with visual arts and creative writing. Early attempts at generating music with the aid of machine learning models are almost as old as recurrent neural networks (RNNs) themselves [2, 9]. In fact, RNNs, as well as their variants, such as long short-term memory (LSTM) networks, have been the de-facto standard for automatic music generation since their introduction in the 80s. Despite their popularity for music modelling, in recent years some alternatives to RNNs are starting to emerge. Compared with RNNs, these novel architectures, which include temporal convolutional networks (TCNs) [7] and transformers [14], offer better performance, as well as higher parallelizability, which translates in drastically reduced training times.

For this project, we decided to take advantage of the novel TCN architecture to tackle the problem of music generation, and specifically pop music generation, as pop is a very popular music genre with which we are acquainted and about which we can make some interesting assumptions. We therefore present PolyPop, a polyphonic pop music generation model based on TCNs. PolyPop is inspired by JamBot [3], a polyphonic music generation model which generates music in two steps. First, a chord LSTM generates a sequence of chords, then a polyphonic LSTM generates a polyphonic melody starting from those chords. For our model we retain the same basic architecture of JamBot, but we replace the LSTMs of the chord and polyphonic models with a multi-layer perceptron (MLP) and a TCN, respectively. The resulting model is able to generate pleasant-sounding music, with realistic chord transitions and recognizable melodies which display long-term dependencies. We also propose two heuristics for the sampling phase of the generation process, which we believe favor more cohesiveness in the generated melodies, however at the expense of variety.

Lastly, we evaluate PolyPop against JamBot. We train both models on the same dataset, consisting of hundreds of pop songs in MIDI format, and we compare their performance using both generic and domain-specific evaluation metrics. We also compare training times and find that TCNs are significantly faster to train than LSTMs.

II. Background and Previous Work

A. Symbolic Music Generation

1) Overview: The task of music generation can be roughly split in two macro-varieties: symbolic and sub-symbolic. Sub-symbolic music generation works with raw audio and treats music as a waveform or some derived representation (such as a spectrogram). On the other hand, symbolic music generation works at a higher level of abstraction, and works with music in some form of symbolic representation. We will work exclusively on symbolic music generation, which from this point onward we will refer to as simply “music generation”. The symbolic approach has a number of advantages to it. For one, almost all symbolic representations of music carry with them some explicit information about concepts such as notes, durations, tempo, etc. which in the sub-symbolic approach need to be inferred from the raw audio. Note that even though we approach music in a *symbolic* way, the learning techniques that we study and employ are exclusively *sub-symbolic*, as they pertain to the field of deep learning.

2) Symbolic formats: Historically, the most widespread form of symbolic musical representation in the western world is the staff notation employed in modern scores. However, being graphical, this form of notation is unwieldy for computers to process. This is why nowadays most symbolic music generation efforts employ MIDI files and the MIDI notation to represent music. MIDI (Musical Instrument Digital Interface) is a technical standard that describes communication protocols and digital interfaces between electronic musical devices [5]. In this standard, notes are represented as numbers in the range 0–127 (60 being the middle C, or C4) and a piece of music is represented as a sequence of events spaced in time which either turn on a note, turn off a note, or change some global properties of the music being played, such as the tempo. Despite being already easy for a computer to process, MIDI files are oftentimes pre-processed into *piano rolls*. A piano roll represents a piece of music as a sequence of *timesteps* with fixed duration (expressed in terms of a fraction of a measure), each containing the notes that are currently playing in that time span. This format is not only easier to process (the subdivision in timesteps makes it a prime choice for machine learning models), but it is also more readily understandable by human readers. In fact, many modern music production environments prefer this notation to the traditional staff notation.

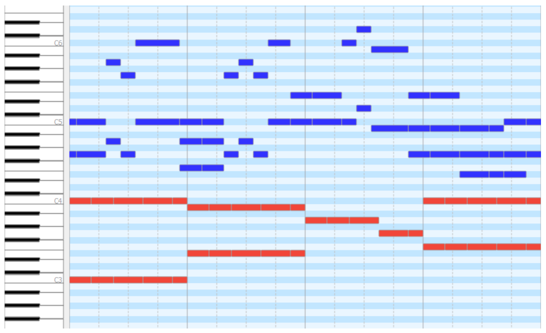
Fig. 1. An example of a graphical piano roll from the MIDI editing software *MidiEditor*.

B. JamBot

1) Overview: *JamBot* is a music generation architecture proposed by Brunner et al. in 2017 [3]. It generates music in two steps. Starting from a seed, first a chord LSTM generates a sequence of chords. Later, a polyphonic LSTM generates a polyphonic piece from the seed and the previously generated chords.

2) LSTM networks: LSTMs are a variety of RNNs purposefully designed to capture long-term dependencies. They achieve this goal by employing memory cells equipped with a gating architecture, which controls whether a cell should retain or forget its state at each timestep of the sequence. This allows the resulting network to learn dependencies which would be lost with a regular RNN. In order to briefly explain the functioning of a LSTM, suppose $\mathbf{x} = x_0, x_1, \ldots, x_n$ is an input sequence. For each cell, the following four values are computed:

$$\begin{aligned} i_t &= \tanh(W_i x_t + U_i h_{t-1} + b_i), \\ u_t &= \sigma(W_u x_t + U_u h_{t-1} + b_u), \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f), \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o), \end{aligned}$$

which are know as the *input, update, forget* and *output* gates respectively. Here, W_q, U_q and b_q (for $q = i, u, f, o$) are all learnable weight matrices and bias vectors. Next, the *state* c_t and the *hidden state* h_t of the cell are computed as follows:

$$\begin{aligned} c_t &= f_t \circ c_{t-1} + u_t \circ i_t, \\ h_t &= o_t \circ \tanh(c_t), \end{aligned}$$

where $x \circ y$ denotes the element-wise product of tensors x and y and $c_0 = h_0 = 0$. An LSTM network can comprise multiple cells, arranged in one or multiple layers. At timestep t, each cell in the first layer receives x_t as input, whereas in subsequent layers each cell receives all of the h_t of the previous layer's cells as input. For a more thorough explanation of LSTMs we refer the interested reader to [6].

3) Architecture: As mentioned earlier, JamBot employs two different LSTMs to generate music. The chord LSTM takes as input a sequence of index encoded chords. Chord indices are in the range 0–49 and denote the 49 most frequent chords in the dataset and an "unknown" chord. The chord network first embeds these indices in a higher-dimension space, then runs the resulting embeddings through the LSTM. The goal of the chord LSTM is to guess the next chord in the harmonic sequence.

The polyphonic LSTM, on the other hand, takes as

input a piano roll. This is encoded as a sequence of multi-hot vectors $x_0, \ldots, x_n$ where $x_t^i = 1$ if and only if note i (MIDI notation) is being played during the t-th eighth-note-long timestep of the song. Note how this format does not distinguish between notes that are repeatedly played and notes that are held. Each multi-hot vector is augmented with three additional features: the chord being played at the current timestep, the chord to be played at the next timestep (both in the form of embeddings produced by the chord model) and a binary counter which keeps track of the current position within the measure. The resulting vectors are run through the LSTM. The goal of the polyphonic LSTM is to guess *only* the notes playing at the next timestep (and not the chord or any additional feature).
For more details on the design of JamBot, we refer the interested reader to the original paper [3].

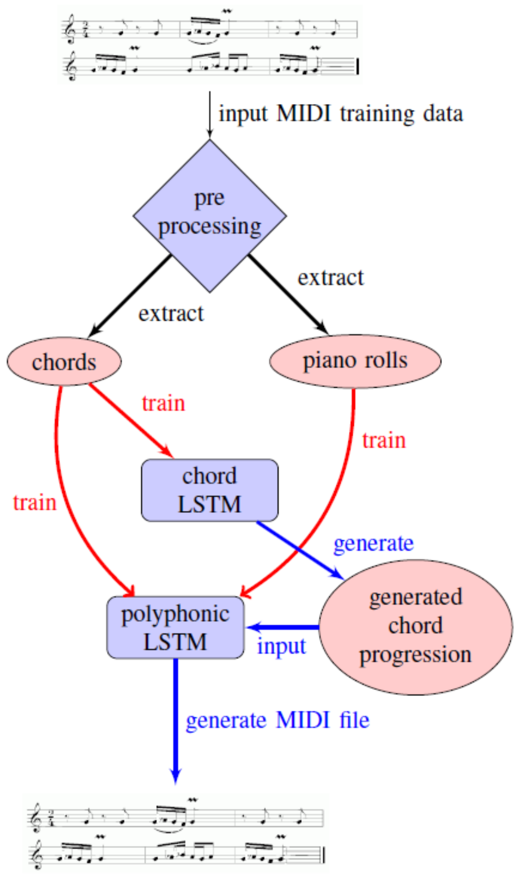

Fig. 2. The architecture of JamBot. The architecture generates midi files whose tempo, instrumentation and production can (or rather, must) be adjusted manually.

C. *Other Models*

Naturally, JamBot is not the only polyphonic music generation architecture available nowadays. Among the most promising music generation models at the time of writing, we mention the Google Magenta project [8], which includes a number of different LSTM models for the generation of polyphonic and monophonic music alike, and OpenAI's MuseNet [10], a transformer-based model capable of generating music complete with rich instrumentation and a notion of style. Transformers themselves are a very promising technology for the future of sequence (and therefore music) modelling. Despite that, they are beyond the scope of this document, so we refer the interested reader to [14].

D. *Temporal Convolutional Networks*

1) Overview: Temporal Convolutional Networks (TCNs) are a relatively novel deep learning architecture introduced by Lea et al. in 2016, originally for action segmentation in video [7]. TCNs combine the effectiveness and performance of convolutional neural networks (CNNs) with the ability to capture time-based relationships of RNNs. Like CNNs, their training is highly parallelizable and thus faster than that of RNNs, and like RNNs they are suitable for modelling sequences of variable-length. In fact, they have been shown to significantly outperform RNNs in many sequence modelling tasks [1].

2) Architecture: TCNs are one-dimensional causal dilated CNNs. The input and output of a TCN are always of the same length. This is achieved by enforcing that every hidden layer be the same size as the input layer, by the addition of a zero padding of length $k-1$, where k is the filter size. When dealing with CNNs, "causal" means that each output y_t of the network is exclusively a function of previous input timesteps $x_0, \ldots, x_t$, while "dilated" means that each hidden layer is given a *dilation factor* d which controls which timesteps from the previous layer the convolution is actually applied over. Given a sequence $\mathbf{x} = x_0, \ldots, x_n$ and a filter $f : \{0, \ldots, k-1\} \to \mathbb{R}$, the *dilated causal convolution* operation $\mathbf{x} *_d f$ on the t-th element of $\mathbf{x}$ is defined as follows:

$$(\mathbf{x} *_d f)(t) = \sum_{i=0}^{k-1} f(i) \cdot x_{t-di}.$$

Note how $t - di$ enforces the *causal* part of the convolution, as it can only point to sequence elements which are in the past. The dilation factor d usually increases the further we get into the network, often taking the values of increasing powers of two (i.e. $d = 2^i$ in the i-th hidden layer of the network). This ensures two things. First, that the receptive field of the network grows exponentially with the number of layers (whereas in an undilated CNN this would grow linearly), granting a longer history. Second, that the output y_t at time t is effectively a function of a contiguous number of steps in the input. A network with m layers has a receptive field of size $2^{m-1}k$.

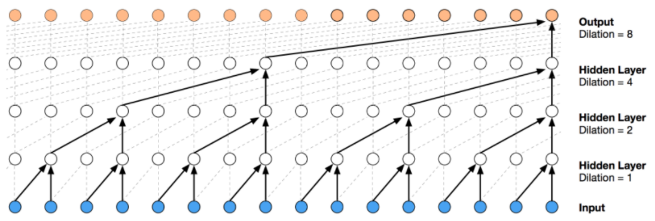

Fig. 3. A stack of 4 dilated causal convolution layers with dilation factors $d = 1, 2, 4, 8$ and filter size $k = 2$. The receptive field of the stack is $2^3 \cdot 2 = 16$. Note how the choice of $d = 2^i$ for layer i allows every output to be connected to 16 contiguous inputs.

III. Dataset

A. Description

All the songs used to train the model are drawn from *The Largest MIDI Collection on the Internet* [13], which comprises over 130,000 MIDI files ranging in genres from classical to rock and pop. Specifically, our dataset is a subset of the full collection, containing exclusively songs from pop music genres. This counts over 1000 songs by over 40 artists, mostly centered around a two-decade-long span of time, from the mid-90s to the early 2010s. Some titles appear multiple time in the dataset, but almost exclusively as different arrangements of the same song. This fact does not seem to compromise the training process.

B. Assumptions

The genre and historical distribution of our datasets allows us to make some assumptions about the kind of music we are generating which are somewhat stronger than the assumptions that are possible on more general datasets.

1) Complexity of Harmony: Of all music genres, pop tends to use harmony in a particularly simplistic way. Unlike classical pieces, which are based on functional harmony, pop songs usually gravitate around well-defined repetitions of few chords (usually 2 to 4). Furthermore, pop music makes little to no use of modulations, whereas classical musical forms depend heavily on it and other modern genres make extensive use of it (as is the case with metal music and some sub-genres of rock). We can therefore make some assumptions on the complexity of the harmonic sequences that we wish to model with our chord model. First, since these sequences can be assumed to be relatively uncomplicated, we can employ a simpler model than the LSTM used by JamBot to predict them. Secondly, since pop harmony is fundamentally cyclic, we do not need to concern ourselves with long-term dependencies and we can train out model with a fixed (and rather short) time window rather than on full songs. This is the reason why we chose an MLP for chord generation (see Section IV-B).

2) Separation of Chords and Melody: Oftentimes when composing pop music, artists tend to think of harmony and melody separately. That is, the choice of chords (harmony) and the design of the vocal lines (melody) are usually two distinct phases which are carried out sequentially and iteratively, often starting in this very order. This phenomenon is also reflected in the finished product, where most of the times the chosen harmony is encoded in very recognizable musical structures, such as block chords or strumming patterns. This marks a stark difference especially with classical music, whose composition requires a more holistic approach to harmony and melody and where the harmonic component is very rarely explicit, and is rather inferred from the melodic lines (a prime example of this are canons and fugues in baroque music). This is one of the reasons we chose JamBot, with its strong division between chord and melody generation, as the starting point for our model. We also apply this assumption directly to the generation of the output MIDI files. During this phase, in addition to writing the sampled output of the polyphonic model to the file, we also directly write the chords generated by the chord model, in the form of block chords (see Section IV-D).

C. Preprocessing

The preprocessing steps are exactly those carried out in the original JamBot paper, which we describe briefly. For further details, refer directly to [3]. At the end of this process we are left with a little over 700 of the songs that were originally in the dataset. Out of these, 20% (around 140 songs) are randomly taken and set aside for testing. The remaining songs are used for training.

1) Key Selection: First, all the songs in the dataset are categorized according to their key. To do so, a pitch class histogram is computed for each song. The 7 most frequent pitch classes are then matched against the notes of each of 48 possible scales (12 scales for each type: major/natural minor, harmonic minor, melodic minor and blues scale). Eventually, only the songs that are in a major or natural minor key are selected.

2) Transposition: Using the information computed in the previous step, each song is transposed a number of semitones up or down so that all songs are in the key of C major (or, equivalently, A natural minor). This step basically shifts every song into the same frame of reference, making it easier for any model to learn meaningful relationships between notes by ignoring the differences introduced by having multiple keys (which arguably do not contribute to the quality of a piece of music).

3) Range Selection: The MIDI file format is capable of representing 128 distinct notes. Because the extreme ends of this range are virtually unused (for reference, a full 88-key piano can only play the notes in the 21–108 MIDI range), only the notes in the 36–84 range are retained for each song, which translates to the 49 notes between C2 and C6.

4) Chord Extraction: In order to train the chord model, the chords of each song are extracted. To do so, a pitch class histogram is computed for each measure of each song. The chord corresponding to that measure is then defined as the three most frequent pitch classes. Note that this is but a crude approximation of what a chord really is: it excludes chords with more than three distinct notes (which also appear in pop, although less frequently than in other genres) and includes clusters of three notes that traditionally would not be considered chords. Nonetheless, this seems to be a satisfactory definition when extracting chords as an additional feature for learning.

5) Piano Roll Extraction: Lastly, each song is converted from a MIDI representation to a piano roll representation (see Section II-A2). The timestep is chosen as one eighth of a note, such that to each chord in the chord dataset described in the previous section (i.e. to each measure) there correspond 8 piano roll timesteps.

IV. Model

A. Overview

PolyPop's architecture follows closely that of JamBot. A chord model and a polyphonic model are trained to predict the next chord and the next piano roll, respectively, and then they are used to generate music starting from a seed. The main difference between the two architecture lies in the nature of the individual models.

B. Chord Model

1) Choice of Model: The chord model attempts to solve a chord prediction task. As we anticipated in the introduction and in Section III-B1, because the harmonic sequences that can be encountered in pop music are rather uncomplicated and do not exhibit significant long-term dependencies, the chord model consists of a simple MLP architecture (see Figure 4). This choice is also supported by our failed attempts to use a TCN for chord prediction too. When a TCN is used, we find that the model ends up either overfitting the training set, or predicting exclusively one chord, regardless of the input. This could indicate that our assumption on the complexity of pop harmony is indeed true.

2) Training Data: The chord index dataset that we extracted during preprocessing (see Section III-C4) is divided into a number of overlapping fixed-length sequences representing temporal windows over the chord sequences. In the current implementation, the window size is set to 10, as most of pop harmony is built around 4-measure-long chord cycles and this window size allows the model to take into consideration 2 and a half such cycles. Each temporal window $\mathbf{x}$ constitutes an input to the network. The target $\mathbf{y}$ for each window is then a one-hot encoded representation of the chord that immediately follows that window in the chord sequence. Our dataset yields around 50,000 such data points.

3) Description: The model receives as input a fixed-length sequence of chord indices. Each chord index in the input sequence is then embedded in a 10-dimensional vector space and the 10 resulting vectors are flattened into a single 100-dimensional vector. This single vector is then passed through two densely connected layers. In between these two layers, a dropout layer with a retention rate of 0.7 prevents overfitting. The output $\hat{\mathbf{y}}$ is expectedly a 50-dimensional vector in which y_i represents the probability of the i-th chord being the next chord in the harmonic sequence. Because chord predictions are mutually exclusive, the last activation function is a softmax.

4) Training: The model is trained to minimize the categorical cross-entropy between $\hat{\mathbf{y}}$ and $\mathbf{y}$. During training, we use the Adam optimizer with a learning rate of 10^{-3}. We train the model for 20 epochs.

C. Polyphonic Model

1) Choice of Model: As we anticipated in Section II-D, TCNs are an emerging architecture for variable-length sequence modelling which has been shown to outperform RNNs (including LSTMs) in a wide range of applications. Like Jambot's polyphonic model, our polyphonic model attempts to solve a note prediction task. Unlike Jambot, however, our polyphonic model is a TCN, not an LSTM (see Figure 4).

2) Training Data: For each piano roll in the piano roll dataset (see Section III-C5) we compute an augmented version of the piano roll itself. At each timestep, this version consists of a multi-hot encoded representation of the piano roll timestep, one-hot encoded representations for both the chord playing in the current measure and in the next measure, and a scalar counter representing the current position within the measure (which therefore goes from 0 to 7). A timestep s_t of one such sequence is therefore a 150-element vector which can be partitioned as follows:

$$s_t = \begin{pmatrix} r_t^0 \\ \vdots \\ r_t^{48} \\ h^0_{\lfloor \frac{t}{8} \rfloor} \\ \vdots \\ h^{49}_{\lfloor \frac{t}{8} \rfloor} \\ h^0_{\lfloor \frac{t}{8} \rfloor + 1} \\ \vdots \\ h^{49}_{\lfloor \frac{t}{8} \rfloor + 1} \\ c \end{pmatrix} \begin{matrix} \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{piano roll} \\ \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{current chord} \\ \left.\vphantom{\begin{matrix} a \\ b \\ c \end{matrix}}\right\} \text{next chord} \\ \text{counter} \end{matrix}$$

Here we highlight two differences with JamBot. First, in JamBot the current and next chords features are to be interpreted as the chords playing during the current and next *timesteps*, not *measures*. We went with the latter option because it allows more timesteps to "build up" to the next measure (which is the rationale behind the addition of such features in JamBot's paper) and because it still allows the model to infer the chord at the next timestep from the current chord, next chord and counter features (whereas the contrary is not possible in JamBot). Secondly, where we employ a *scalar* counter, JamBot employs a 3-element vector which implements a *binary* counter. In this case, our choice is supported by empirical results, as the model seems to perform significantly better with a scalar counter instead of a binary one.
Each augmented sequence constitutes a different data point for the polyphonic model. For each sequence $\mathbf{s}$ of length n, the input to the network is $\mathbf{x} = s_0, s_1, \ldots, s_{n-2}$, while the target is $\mathbf{y} = s_1, s_2, \ldots, s_{n-1}$.

3) Description: The model receives as input a variable-length sequence of timesteps. This sequence is run through a single TCN stack consisting of 6 dilated causal convolution layers with power-of-two dilations. For each layer we employ 150 filters with kernel size $k = 8$. As in the chord model, each layer of the TCN is followed by a dropout layer with a retention rate of 0.7. The TCN stack is configured to return sequences, so for an input sequence of length n, the output of the TCN stack is a sequence of n 150-element vectors. Each of these vectors is passed through a dense layer to obtain the final sequence of n 49-element vectors $\hat{\mathbf{y}}$ corresponding to the piano roll prediction. Note that unlike chord prediction, piano roll prediction is not a multi-class prediction task, but rather a multi-label prediction task (in principle, the fact that note i is playing does not prevent note j from playing). Because of this, the last activation function is a sigmoid rather than a softmax.

4) Training: The model is trained to minimize the categorical cross-entropy between each timestep $\hat{y}_t$ of $\hat{\mathbf{y}}$ and the corresponding y_t of $\mathbf{y}$. The use of binary cross-entropy as the loss function would seem more appropriate, as $\hat{y}_t$ is more intuitively interpreted as a collection of 49 distinct binary predictions (each dictating whether an individual note should be playing or not). However, we find that the model performs significantly better when trained with categorical cross-entropy rather than binary cross-entropy. During training, we use the Adam optimizer with a learning rate of 10^{-3}. We train the model for 50 epochs.

D. Generation

1) Process: Once a chord model and a polyphonic model have been trained, they can be used together to generate music. First, a seed is chosen as a portion of a song from which piano rolls and chords have been extracted. The generation process is the typical one for predictive models: the model produces a prediction for the next timestep based on the seed. An outcome is then sampled from such prediction and appended to the seed, to be used in the generation of the following timestep. The chord model is the first of the two to run and generates the chords for $n + 1$ measures, starting from the seed.
Next comes the turn of the polyphonic model. At each timestep, the polyphonic model generates a prediction for the piano roll. For each note, we independently sample whether it is actually playing or not based on the probabilities assigned by the model. The resulting

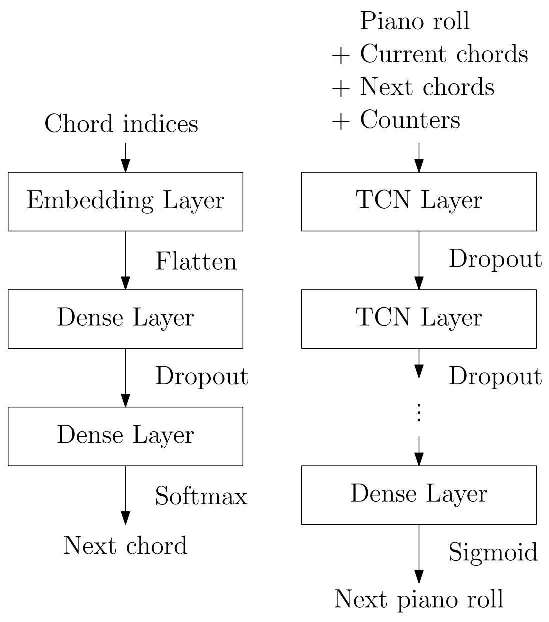

Fig. 4. A graphical representation of the proposed architecture. The chord model is on the left, while the polyphonic model is on the right.

timestep is augmented with the current chord and next chord features (which are drawn from the previously generated chord sequence), as well as with the counter feature, and is then appended to the seed. The polyphonic model generates $8n$ timesteps like so (the chord model generates $n+1$ measures so that the next chord feature is always well-defined).
At the end of the generation process, the original seed is removed from the beginning of the resulting sequence. The remaining, entirely machine-generated sequence is then stripped of the additional features described in Section IV-C2 and re-converted to a midi representation, which is then written to file. During this last step, we also write the chords predicted by the chord model directly to the midi file, in the form of block chords.

2) Sampling Heuristics: During the sampling phase of the generation process we experiment with hard-coded heuristic to refine the independent sampling described in the previous section. The polyphonic model has no real concept of melodic lines: it sees every piano roll as a sequence of “bag of notes” timesteps and not as a collection of parallel monophonic melodies (each corresponding to a melodic line), which is the more “human” way of interpreting a piano roll and music in general. Because of this, the independent sampling of notes falls short of being satisfactory. Suppose, for example, that because note i is playing at timestep t, the model predicts that both notes i and $i+1$ are very likely to play at timestep $t+1$. This is in fact a very plausible prediction, as it reflects the common scenarios of a note being held and of step-wise motion, respectively. If a real composer were to mentally compute the same probabilities and sample them, they would interpret note i as the last note in a monophonic melodic line and choose either i or $i+1$, but never *both*, to continue it. By sampling notes independently this intuition does not hold and the risk is that both notes are played together in the following timestep, resulting in an unrealistically dissonant harmonic interval (for every i, notes i and $i+1$ form a minor second).
To circumvent these undesirable scenarios, we adjust the probability of each note being sampled dynamically while sampling. Specifically, if note i gets to play, then the probability of playing notes in immediate dissonant intervals (which currently include minor and major seconds) drops by half.
In addition to that, to further promote the sampling of notes in a way that resembles a melodic line, we boost the probability of step-wise motions and smaller skips being generated. This means that whenever note i is playing at timestep t, then the probability p of the notes in the vicinity of i (where “vicinity” is to be interpreted as “up to a major third away”) playing at timestep $t+1$ is boosted to $p' = p \dot{+} p = p + p - p^2$.

E. *Implementation Notes*

For the implementation of the model we used Keras with the TensorFlow backend. For the TCN layers, we relied on the Keras TCN implementation by Philippe Rémy, available on GitHub [12]. The conversion from MIDI files to piano rolls and the accompanying preprocessing operations were made possible by the Pretty_MIDI [11] library for Python, whereas the conversion from piano rolls to MIDI files was made possible by the Music21 [4] library.

V. Results

A. *Qualitative Observations*

The music generated by PolyPop is overall very pleasant. Even without the heuristics described in Section IV-D2, the majority of the notes that are played together are consonant, with only some occasional dissonance, and more importantly they are consistent with the underlying harmony generated by the chord model.
Speaking of the chords, they transition into each other frequently (the cases of chords being held indefinitely are rare) and in a realistic way, which seems to mimic some of the common chord sequences that can be found

in pop music. Chord transitions are never jarring. This is likely due to the the fact that the training set is all in one key, and thus chords of different keys rarely appear in the dataset, and to the complexity assumption, which we therefore believe to be at least somewhat reasonable (see Section III-B1). The explicit addition of harmony to the generated music in the form of block chords (see Section IV-D) makes the resulting piece feel more cohesive and adds to the overall realism of the generated music.

Speaking of melody, on the upside the model is clearly able to generate recognizable melodies, which also exhibit long-term dependencies. On the downside, these melodies often jump back and forth across octaves, with large skips, which makes them somewhat sparse in the vertical (harmonic) dimension. Because they are dense in the horizontal (temporal) dimension, a human listener is nonetheless able to perceive and extrapolate more distinct and cohesive melodic lines from them, but the size and frequency of the skips is still rather unrealistic. This "sparsity" is perhaps the most significant shortcoming of PolyPop's music and is likely due to the fact that the model has no explicit concept of vocal or melodic lines, as we discussed in Section IV-D2, and thus the likelihood of cohesive melodic lines forming on their own by means of random sampling is low. By turning on the heuristics described in the same section, the generated melodies become in fact less prone to skips. However, melodies sampled this way are also less varied, which is to be expected since the heuristics strongly favor consonance and step-wise motion.

At the time of writing, some examples of the music generated by PolyPop (with and without heuristics) can be found at the following URL: https://rb.gy/tau5wh.

B. Quantitative Evaluation

1) Overview: For our objective quantitative evaluation, we evaluate both our model and JamBot as predictors. We do not evaluate the performance of the respective chord models directly, but rather indirectly as part of the wider generation process. Also, during the evaluation phase we do *not* employ any of the heuristics that we described in Section IV-D2. As metrics, we employ those described in Section V-B2. We present and compare the results of the evaluation in Section V-B3.

2) Evaluation Metrics: The evaluation metrics that we employ include the sequence F-score and cross-entropy, as well as two of the domain-specific metrics described by Ycart and Benetos in [15].

Let $\mathbf{y}$ be a ground truth and $\hat{\mathbf{y}}$ the non-binary output prediction of a model. Let $\tilde{\mathbf{y}}$ be the binary form of $\hat{\mathbf{y}}$, obtained from $\hat{\mathbf{y}}$ by means of sampling. At timestep t, the number of true positives TP_t is defined as the number of indices i for which $\tilde{y}_t^i = y_t^i = 1$. False positives FP_t and false negatives FN_t are also defined in the intuitive way from $\tilde{y}_t$ and y_t. Precision P_t and recall R_t are then defined in the usual way from TP_t, FP_t and FN_t, and the F-score of a single timestep F_t is thus defined as

$$F_t = 2\frac{P_t \cdot R_t}{P_t + R_t}.$$

The cross-entropy between $\mathbf{y}$ and $\hat{\mathbf{y}}$ at timestep t is defined as

$$H(y_t, \hat{y}_t) = \sum_{i=0}^{48} y_t^i \log(\hat{y}_t^i) + (1 - y_t^i)\log(1 - \hat{y}_t^i),$$

where 0–48 is the note range that we used to train the polyphonic model in the first place. The sequence F-score and cross-entropy of a sequence $\mathbf{s}$ are then defined as the average of F_t and $H(y_t, \hat{y}_t)$, respectively, over the timesteps of $\mathbf{s}$.

Out of the domain-specific metrics mentioned earlier, we employ the *transition cross-entropy* and *steady-state cross-entropy*. Intuitively, the former computes a cross-entropy over those timesteps in which the melody changes, whereas the latter computes the cross-entropy over those timesteps in which the melody is held. More formally, let Tr be the set of transitioning timesteps, defined as follows:

$$Tr = \{t \mid y_t \neq y_{t-1}\}.$$

Furthermore, let d_t be the number of notes by which timesteps y_t and y_{t-1} differ, defined as

$$d_t = |\{i \mid y_t^i \neq y_{t-1}^i\}|$$

The transition cross-entropy between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is then defined as

$$H_{tr}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{|Tr|}\sum_{t\in Tr}\frac{H(y_t, \hat{y}_t)}{d_t},$$

where we divide $H(y_t, \hat{y}_t)$ by d_t because the former is observed to be proportional to the latter [15].

Now let Ss be the set of steady timesteps, defined as follows:

$$Ss = \{t \mid y_t = y_{t-1}\}.$$

Note that if $\mathbf{y}$ has length n, then $Tr \cup Ss = \{1, 2, \ldots, n-1\}$. The steady-state cross-entropy between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is then defined as

$$H_{ss}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{|Ss|}\sum_{t\in Ss} H(y_t, \hat{y}_t).$$

3) *Evaluation Results:*

a) Performance: From Table I and Figure 5 it seems that PolyPop outperforms JamBot significantly as a predictor. It is worth noting that after epoch 10, JamBot's performance over both the test set (shown) and the training set (not shown) starts decreasing. This is likely caused by too high of a learning rate, which prevents the model from reaching a local minimum (although our choice for learning rate is the same as the original paper). Even though this prevents us from affirming the true extent by which PolyPop outperforms JamBot, we do not attempt to train the latter again with a lower learning rate, mainly because of the prohibitive amount of time that it would require to do so (see next paragraph). We therefore limit ourselves to pointing out that our model's performance surpasses JamBot's best performance (epoch 10) as early as epoch 5, if not earlier, and that the general trend of the evaluation metrics does not suggest that JamBot would be able to surpass our model with more training. Looking at the domain-specific metrics, we find that, with more training time, the ability of our model to predict steady states improves more than its ability to predict transitioning states. Intuitively, it is easy to be convinced that it is much simpler for any model to predict a repeated state rather than a transitioning one, and that therefore such a difference between H_{tr} and H_{ss} is to be expected. This intuition is also supported in [15]. What is more curious is that JamBot seems to be better at predicting transitioning states, rather than steady states. However interesting, we do not investigate this phenomenon further.

b) Training Time: The most surprising difference between PolyPop and Jambot lies certainly in the amount of time that it takes to train either model. Both models were trained on the same training dataset, with the same batch size and on the same setup (whose specifications are found in Table II). However, as can be evinced by Table I, the TCN architecture of our model took orders of magnitude less time to train than JamBot's LSTM architecture. For reference, we where able to train the entire 50 epochs on our model (complete with tests every 5 epochs) in less than half the time that it took JamBot to complete a single epoch. It is because of this that we limited JamBot's training to just 20 epochs instead of 50 and we decided not to train JamBot again with a lower learning rate.

VI. Conclusions and Future Work

A. *Conclusions*

We presented PolyPop, a pop music generation model based on the novel TCN architecture. PolyPop generates music in two phases, first by generating the chords for a song, and then by generating a polyphonic melody from those chords. This setup is inspired by JamBot, a general music generation model based on the LSTM architecture. We compared our model's and JamBot's performance on a pop music dataset with different evaluation metrics (including domain-specific ones) and found that our model performed significantly better than JamBot according to all metrics. Not only that, but we also showed that our model's TCN architecture was significantly faster to train than JamBot's LSTM. We believe that the ability to quickly train an effective model for sequence modelling, even on a consumer-grade machine such as a laptop, will lead to interesting applications of TCNs in the future.
Objective evaluations apart, we found that our model was capable of generating music of good quality, with consistent chord transitions and recognizable melodies, which exhibit long-term dependencies. We proposed two heuristics to use during sampling to favor the generation of realistic melodic lines, which we found to help improve the cohesiveness of the generated melodic lines, although at the expense of their variety.

B. *Future Work*

Our sampling heuristics are but a crude workaround to the fact that PolyPop (like JamBot) does not have any internal concept of melodic line. We therefore believe that two directions would be worth researching. The first one would be the incorporation of the concept of melodic line within the model itself, perhaps by means of multiple TCNs generating each a single line. The second one would be the design of a more advanced sampling strategy. This could be done by devising more complex, musically informed heuristics, or by relying on a third machine learning model, which could learn which notes are likely to play together from the dataset and guide the sampling process accordingly.
Furthermore, our model cannot distinguish between multiple repeated notes and notes that are held in time. This shortcoming is inherent to the piano roll representation, and is also experienced by JamBot. Our model currently resolves this form of ambiguity in favor of the multiple-repeated-notes interpretation, but we think it would be beneficial to the quality of the final output if a different representation for melodies could be used that allowed

TABLE I
COMPARATIVE RESULTS.

Architecture	F-score[a]	Cross-entropy[a]	H_{tr}[b]	H_{ss}[b]	Training time (epoch)	Training time (song)
JamBot (10 epochs)	0.51	8.59	3.68	4.91	183 min	19 s
Our model (10 epochs)	0.58	6.12	2.97	2.94	51 s	88 ms
Our model (50 epochs)	0.67	5.30	2.56	1.62	45 s	79 ms

[a]Average across all timesteps of all songs.
[b]Average across all songs.

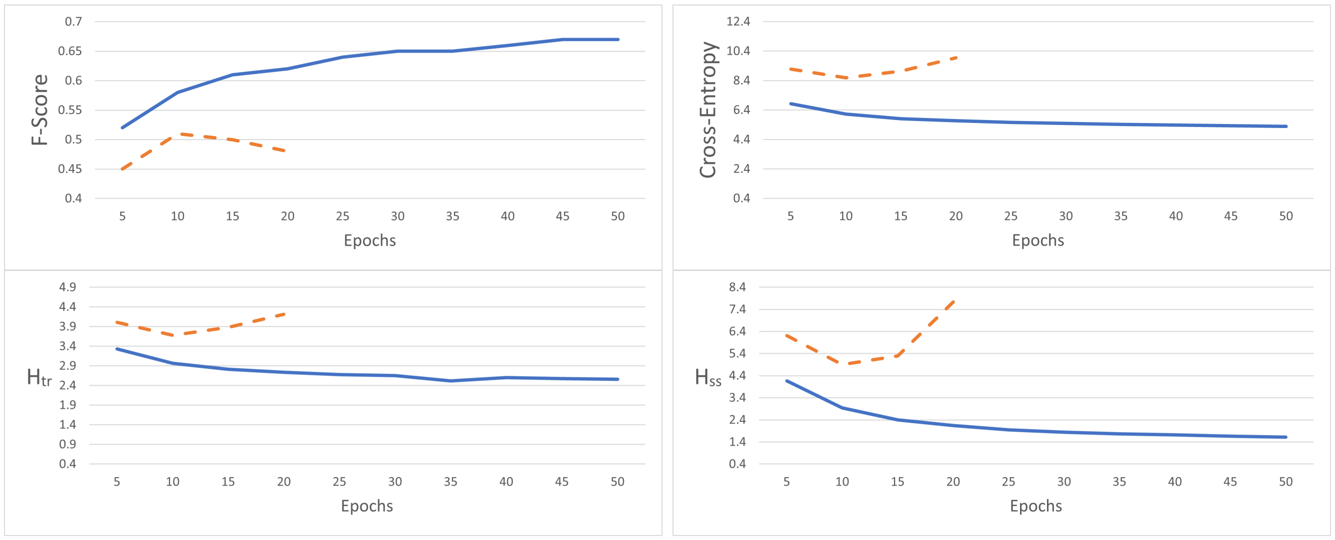

Fig. 5. Evaluation metrics comparison between JamBot (dashed orange line) and our model (solid blue line). From top to bottom, left to right: sequence F-score, sequence cross-entropy, transition cross-entropy and steady-state cross-entropy.

TABLE II
TRAINING SETUP

CPU	Intel Core i5-8250 CPU, 1.60GHz
RAM	8GB
GPU	NVIDIA GeForce 940MX, 2GB VRAM
CUDA	Versions 10 (JamBot), 10.1

the model to distinguish between the repeated and held cases.
Lastly, it would also be interesting to see if our model performs as well on others genres of music as it does on pop music. In particular, if this were not the case, it would mean that the assumptions that we made on the dataset and that we used to guide the design our model are in fact correct.

REFERENCES

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*. 2018. eprint: arXiv:1803.01271.

[2] Jamshed J. Bharucha and Peter M. Todd. "Modeling the Perception of Tonal Structure with Neural Nets". In: *Computer Music Journal* 13.4 (1989), pp. 44–53. ISSN: 01489267, 15315169. URL: http://www.jstor.org/stable/3679552.

[3] Gino Brunner et al. *JamBot: Music Theory Aware Chord Based Generation of Polyphonic Music with LSTMs*. 2017. arXiv: 1711.07682.

[4] Michael Scott Cuthbert and Christopher Ariza. "Music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data." In: *ISMIR*. Ed. by J. Stephen Downie and Remco C. Veltkamp. International Society for Music Information Retrieval, 2010, pp. 637–642. ISBN: 978-90-393-53813. URL: http://dblp.uni-trier.de/db/conf/ismir/ismir2010.html#CuthbertA10.

[5] Amanda Ghassaei. *What Is MIDI?* 2016. URL: https://www.instructables.com/What-is-MIDI/.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Colin Lea et al. *Temporal Convolutional Networks for Action Segmentation and Detection*. 2016. eprint: arXiv:1611.05267.

[8] *Magenta: Music and Art Generation with Machine Intelligence*. 2020. URL: https://github.com/magenta/magenta.

[9] Michael C. Mozer. "Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Multi-scale Processing". In: *Connection Science* 6.2-3 (1994), pp. 247–280. DOI: 10.1080/09540099408915726.

[10] Christine Payne. *MuseNet*. 2019. URL: openai.com/blog/musenet.

[11] Colin Raffel and Daniel P.W.Ellis. "Intuitive analysis, creation and manipulation of midi data with pretty midi". In: *15th International Conference on Music Information Retrieval Late Breaking and Demo Papers*. 2014.

[12] Philippe Rémy. *Keras TCN*. 2020. URL: https://github.com/philipperemy/keras-tcn.

[13] *The Largest MIDI Collection on the Internet, collected and sorted diligently by yours truly*. 2015. URL: https://www.reddit.com/r/WeAreTheMusicMakers/comments/3ajwe4/the_largest_midi_collection_on_the_internet/.

[14] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[15] Adrien Ycart and Emmanouil Benetos. "Learning and Evaluation Methodologies for Polyphonic Music Sequence Prediction with LSTMs". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* PP (Apr. 2020), pp. 1–1. DOI: 10.1109/TASLP.2020.2987130.