Probabilistic and Deep Learning Methods for Sequential Music Generation

Anirudh Baddepudi*, Mayank Jain*, Arvind Mahankali*
Carnegie Mellon University
abaddepu@andrew.cmu.edu, mayankj@andrew.cmu.edu, amahanka@andrew.cmu.edu

Abstract

Our goal in this study is to build generative models to create complex music that simulates human composition. We focus on models which generate sequences of musical elements one at a time. In addition to learning singular notes/chords (which define a melody), we consider rhythm, tempo and duration to improve the authenticity of our pieces. Our baseline approach is a Markov Chain model which learns transition probabilities between note/rest objects. We then extend the current state of the art by proposing and testing 3 new architectures which make use of recurrent neural networks (RNNs) and autoencoders. Though evaluating music is inherently subjective, we find that these models provide high-quality musical results, as we evaluate the authenticity of our music by surveying CMU students.

1 Introduction

Generating aesthetically pleasing music is a fundamentally challenging task, due to its subjective nature and the extent of creativity required. It naturally follows that an interesting problem is to explore whether we can use machine learning methods to generate original music that sounds pleasing to the ear, simulating human-created music. Our primary goal in this study is to generate music that sounds good, but is not copied from human-generated music in a naive way. We aimed to build new deep learning techniques that have not been previously explored in the context of music generation.

A major difficulty in generating music is the consideration of properties such as rhythm, rests, note duration and melody. We hypothesized that creating models which are able to learn these properties would give us close approximations to human-constructed music. As a result, a large focus of our work is in embedding notions of rhythm, tempo, rests, melody and harmony into our algorithms. These required our models to have added complexity and learn sophisticated properties of a piece.

In this work, we focus on sequential models: we aim to predict a musical element, i.e. a note, a rest, or a chunk (which we define to be a fixed-length subsequence of notes) based upon a sequence of previous elements in a generated piece. In other words, all of our models generate music by sampling from a conditional probability distribution of the form

$$P(s_t \mid s_{t-1}, s_{t-2}, \ldots, s_{t-\ell})$$

where s_t denotes the musical object seen at time t, and ℓ represents the number of previous objects in the sequence which determine the next generated element.

In order to solve this problem, we take multiple approaches, which we introduce in the rest of this report. For our baseline, we use *FakeBach* [5], a Markov Chain approach which uses note pattern frequences to generate transition probabilities. Our other, more advanced, approaches, all make use of recurrent neural networks (RNNs), which have encountered great success in problems involving the generation of sequences ([9]). We first introduce an RNN architecture which generates notes of variable duration, by conditioning on the previous notes and their duration. We then introduce ***ChunkRNN***, which extends this architecture by taking in several previous *chunks* of C notes (for an integer $C > 1$) to generate a new chunk of C notes, rather than generating one note at a time. Along with ChunkRNN, we introduce a new pre-training approach for ChunkRNN based on previous work in pre-training neural networks ([2]), and pre-training using autoencoders ([8], [10]). Finally, we incorporate the notion of harmony into RNN-generated music through a generation method we call ***RhythmRNN***.

Note: All authors contributed equally to this work.

2 Background and Related Works

2.1 Markov Chains

We investigated the idea of using Markov Chains as a baseline to generate music. Among existing implementations, a common approach is to train a Markov chain (determine transition probabilities) to predict a note given a sequence of notes that came before [11]. While this works reasonably, it is unable to capture musical complexity and variation. More complex approaches involve predicting the octave [13] and rhythm (predicting whether the next note is a rest/note, as well as its duration) [5]. After considering these approaches, we decided to use an approach based on FakeBach as our baseline [5]. FakeBach also greatly informed our RhythmRNN approach, which can be considered to be a deep learning-based improvement upon FakeBach.

2.2 Deep Learning for Generation Problems

Deep generative models are a central topic in deep unsupervised learning. Some popular models are generative adversarial networks (GANs) and variational autoencoders (VAEs). For sequence generation problems, Recurrent Neural Networks are particularly well-suited, as demonstrated in [9]. One study using RNNs for music generation [7] generates notes and chords in fixed time steps. A key difference between [7] and our initial LSTM architecture is that we generate notes with variable durations i.e, our basic unit is a note (along with its duration), rather than simply the pitch of a note, since we considered variation in rhythm a key attribute of genuine music. One other relevant work is that of [17], which inspired our input strategy of feeding in a fixed number s of elements at a time to our RNNs.

Another approach [12] generates rhythms for percussion pieces, which partly inspired our RhythmRNN architecture. The key difference between their architecture and RhythmRNN is that while they also make use of a feedforward network and an RNN, they merge these two to create a single architecture. We sample separately from our feedforward network (to create a rhythm) and our RNN (to get a melody).

2.3 Pre-Training and Music Generation with Autoencoders

An autoencoder is a neural network which performs a function analogous to PCA, as follows: given an unlabeled training dataset consisting of points $x_i \in \mathbb{R}^d$, for $i = 1, \ldots, N$, the goal of the autoencoder is to, given x_i in the training dataset, output a point x_i' such that $L(x_i, x_i')$ is minimized, for some appropriate loss function L. In words, the job of an autoencoder is to reconstruct points in the input dataset. Autoencoders typically consist of two fully connected neural networks joined end-to-end: an encoder network (which maps points in $\mathbb{R}^d$ to points in $\mathbb{R}^k$, for k much smaller than d) and a decoder network (which maps points in $\mathbb{R}^k$ to points in $\mathbb{R}^d$).

The typical application in deep learning of autoencoders is in pretraining of deep neural networks ([2], [8], [10]). This is an important step in the training process, since the initialization of the weights of a neural network can play an important role in the quality of the weights at the end of the process (since optimizing the weights of the neural network is a non-convex optimization problem, meaning local minima are not necessarily global minima). In addition to providing a good initialization to reduce training error, autoencoders can also reduce true/generalization error, as mentioned in [8] and [2]. One key difference between our approach and the others is that, unlike in other works involving pre-training with autoencoders (e.g. [10]), we also use the weights of the decoder in pre-training.

However, autoencoders can also be used as generative models, particularly in music generation. One work which applies such a technique is DeepHear ([16]) which trains a (stacked) encoder and decoder on entire pieces of ragtime music, and treats the decoder as a generative model by inputting random vectors into the decoder, the idea being that the decoder will be able to learn the factors of variation in different ragtime pieces. In a way, our ChunkRNN architecture can be viewed as an extension of DeepHear, since at several time steps, we use an autoencoder-like architecture to sample notes, together with an LSTM which (intuitively) successively generates compressed representations of chunks.

Another approach which is also related to our ChunkRNN model is unit selection [4], in that units of musical pieces are generated one at a time using an LSTM. The key difference between our approach and theirs is that they only use an autoencoder to obtain an encoder, which is used as an embedding for feature vectors (which are constructed from the musical units), while we also use the decoder to reconstruct chunks, and we make use of the encoder and decoder in pre-training. Moreover, our architecture outputs a probability distribution over

chunks (which allows us to independently sample notes in a chunk) while their architecture instead selects units from a training dataset.

3 Methods

As mentioned in Section 1, we used a Markov Chain approach as our baseline, followed by several deep learning approaches involving RNNs.

3.1 Markov Chain Models: Sampling Notes/Rests from a Rhythm Chain

For our baseline, we use an approach based on *FakeBach* [5]. Their approach consists of two Markov Chains - a **Note Chain** (where each state is a sequence of n previous notes, encoded as a number — n is a hyperparameter) for generating notes, and a **Rhythm Chain** (each state is a sequence of n placeholders — either notes or rests) to decide whether the next item in the piece is a note or a rest, as well as its duration.

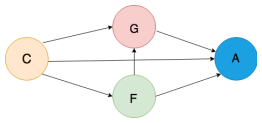

Figure 1: Note Chain Example

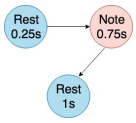

Figure 2: Rhythm Chain Example

Generation in this approach is based on two seed notes fed into the Markov Model. These are first used to create a sequence of notes and rests by sampling from the Rhythm Chain. This sequence is then filled with specific notes by sampling from the Note Chain. One can continue computing future notes by passing the latest sequence of generated notes as input.

3.2 Recurrent Neural Network Construction: Learning patterns with both notes and durations

To generate music with interesting rhythm and tempo, we designed our RNN architecture to take in two inputs, a one-hot encoding vector of pitches/chord pitch lists, and a scalar value corresponding to the note duration. The RNN was trained to predict the next 'element' (a note/chord and its duration), given a sequence of previous notes/durations. Our architecture then maps the note encoding vector and the duration each to vectors with same dimensionality by applying dense layers. We make this design choice because having the two inputs being of wildly different lengths can lead to a drop in performance, as shown in [18]. These two vectors are then concatenated and passed as input to a (bidirectional) LSTM layer, which incorporates information from previous notes.

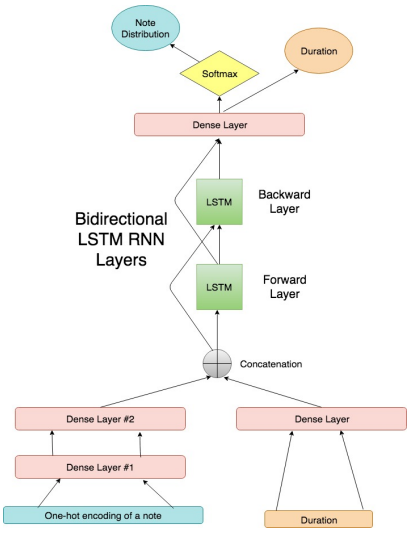

Figure 3: Our First RNN Architecture for Pitch and Duration Generation

The RNN learns to give a probability distribution over new pitches, as well as a scalar value for duration. The probability distribution over a new pitch at time t is

$$P(e_t|e_{t-1}, e_{t-2}, \ldots, e_{t-s})$$

where s is a hyperparameter used in all of our RNN-based methods — it is the number of previous items of a sequence we consider to generate a new element. Our training process is as follows: s pitches and durations are given as inputs to our RNN in succession, and used to predict the pitch of the following note and its duration. The loss function used for the pitch function is cross entropy, while the loss function used for duration is mean squared error. We then perform 'backpropagation through time,' where the RNN is unfolded into its timesteps (but where the weights of the RNN are the shared across time steps), and backpropagation is run through each operation.

Note that we use LSTM (Long Short Term Memory) layers in our RNN to avoid the 'vanishing gradient' problem that occurs due to repetitive computations in our recurrent layers [1]. This is where gradients computed

can either be too small or too large due to finite precision number calculations (leading to repeated rounding errors and therefore inaccurate computations). We use bidirectional layers as well — this has the effect of increasing the expressive power of our model. Note that it is not strictly necessary for us to use bidirectional layers in our case, since in the training process, we only train based on the prediction the model makes at the final time step, and in our sampling process, we only sample based on previous notes. Bidirectional layers, on the other hand, are typically used to incorporate information from future elements of a generated sequence. Regardless, including a bidirectional layer gave an improvement in the training error and the quality of music generated, and this is to be expected, since our model is capable of representing a strictly larger class of functions (since the parameters of the backward LSTM are independent from the parameters of the forward LSTM).

3.3 ChunkRNN: Generating Notes One Chunk at A Time and Pretraining Using Autoencoders

Among our original contributions is a different architecture that we developed which is designed to prevent discordant or false notes from playing: ChunkRNN (shown in Figure 5). The architecture takes in sequences of *chunks* of C notes (for some $C > 1$), and generates the next C notes. The motivation behind this architecture is that it allows future notes to influence past notes. Observe that with our previous RNN architecture and sampling strategy, one potential drawback is that once a note is generated, a future note cannot influence this note, even if these two notes are discordant together. With our new architecture, the goal is to ensure that, since the C notes are generated at the same time, they sound good together.

The architecture (shown in figure 4) consists of two dense layers (used to reduce the dimensionality of a chunk that is given as input) and a bidirectional LSTM layer (which incorporates information from previous chunks), whose outputs are concatenated and passed through two dense layers. To the final dense layer (whose dimensionality is equal to the size of a chunk), we apply a *multiple softmax* layer: we separate the output of the final dense layer into C vectors of equal size, and apply the softmax function to each of them. This has the effect of giving us C probability distributions, which can be sampled to give us the next C notes.

Now, observe that the first two dense layers of the ChunkRNN architecture, together with the last two, can be regarded as a form of autoencoder. This is because the task of the lower two layers is two obtain a lower dimensional representation of a chunk of size Cd (where d is the number of distinct notes/chords which appear in the training set) while the task of the last two layers is to use the output of the LSTM to reconstruct a chunk.

This motivates our idea of pre-training our ChunkRNN model by first training the first two and last two dense layers in the way that autoencoders are usually used for pretraining. As mentioned in section 2, several previous works in supervised learning and image classification have used autoencoders to obtain good initializations of the weights in a neural network. Unlike in convex optimization problems (such as linear regression), where convergence to a global minimum is guaranteed, the objective/loss function used in training neural networks is typically nonconvex (since the neural network is itself typically not a convex or concave function — if some weights are positive and others are negative, then the linear combination of the nodes in the previous layer using these weights may not be convex, since some terms are convex functions while others are concave).

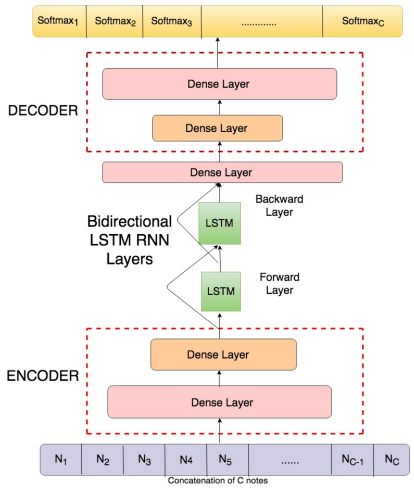

Figure 4: ChunkRNN Architecture

As mentioned in Section 2, one key difference between our work and previous works is that after our pre-training process, we do not discard our decoder — rather, we reuse its weights as well, for the last two dense layers of our ChunkRNN network. This is because, as explained above, we might expect the last two layers of the ChunkRNN network to learn weights which reconstruct a chunk vector from a compressed representation. Note that in order to make this approach sound, we apply our multiple softmax layer to the top layer of the decoder in the pre-training process, so that the weights of the decoder are appropriate for ChunkRNN. The rest of our training process is analogous to that of our original RNN model (described in Section 3.2) — we train our network to predict a chunk based on the previous s chunks, where s is the hyperparameter mentioned in Section 3.2. Note that we again apply the cross-entropy loss to the output of our network and the chunk which

occurs next in the training dataset — this effectively applies the cross-entropy loss to each of the probability distributions over pitches (for each note in the chunk that is to be generated).

3.4 Rhythm RNN

Finally, we designed a model which incorporates the ideas of both rhythm and harmony, as follows. We take inspiration from the approach of FakeBach [5], and we attempt to obtain better results using more powerful function approximators. The goal is to generate two tracks which will be overlaid on one another. We first use a fully connected neural network to generate two sequences of 0s and 1s (0 representing the presence of a rest, and 1 representing the presence of a note), as well as the corresponding durations for each element. The neural network is trained by taking the previous s 0/1 elements and durations (where s is the hyperparameter mentioned in Section 3.2). The fully connected neural network is trained using a method similar to the one used in Section 3.2, although ordinary backpropagation is used here, rather than backpropagation in time.

In these two sequences, the placeholders for notes are then filled in with melody (a sequence of pitches) generated by a separate RNN (whose architecture is shown in Figure 5 — observe that this is a simplified version of our architecture discussed in Section 3.2 (there is no need to consider duration, since that is taken care of by the fully connected network). Finally, one of the tracks is transposed down by an octave.

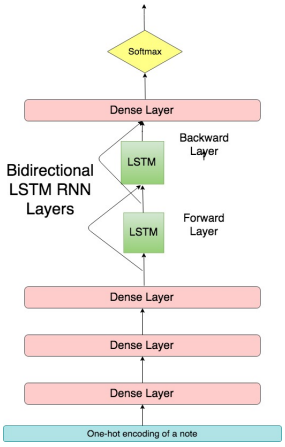

Figure 5: Architecture for Rhythm RNN

4 Datasets

4.1 Preprocessing and Training

We use a dataset containing 92 MIDI files from pop music and various video games for all our training [14]. Pop music is considered to be one of the easiest genres to learn to replicate, and this was the reasoning behind the choice of this dataset.

We use MIT's Music21 Python library to parse MIDI files into its constituent components [6]. This allows us to parse each note into its corresponding pitch, duration, and also consider rest objects at the preprocessing stage. All chords that appear in the input stream are considered as elements of our vocabulary vector — Music21 allows us to obtain the pitch classes of the chord and obtain a string representation of the chord. We also make use of code provided in [17] and [15] for boilerplate constructions involving Music21.

5 Experiments and Results

5.1 Training and Hyperparameter Tuning

5.1.1 Markov Chain

After trying different values of N and different values M for the memory of the Note Chain, we found that $N = 8$ and $M = 2$ was giving us consistently good results on music generation. We therefore used these hyperparameters for our model after running these experiments. The sample we generated and used in our survey is: `http://bit.ly/markovmusic`

5.1.2 Original RNN Architecture

For all of our RNN architectures, we tested a variety of hyperparameters, such as input and output dimensions of the layers, and the number of previous sequence elements (i.e. notes, chords, and rests). Existing implementations tend to use a sequence length of 10, but as we created our own architectures, we performed our own experiments to determine the best choices of hyperparameters. Moreover, we experimented with

different sampling strategies for the RNNs and decided on the following strategy: sample (Sequence Length)-many notes from the training examples, and continue sampling afterwards from the distribution defined by the RNNs. This strategy is used with our original RNN architecture and our RhythmRNN.

We considered the following hyperparameter settings for our original RNN architecture:

Config Number	Sequence Length	Dense Layer 1	Dense Layer 2	Dense Layer Duration	LSTM Output Dimension
1	10	50	30	30	128
2	20	50	30	30	128
3	10	200	100	100	150
4	20	200	100	100	150
5	20	256	128	128	128

The "Dense Layer 1" and "Dense Layer 2" hyperparameters refer to the output dimensions of the two dense layers through which the one-hot encoding of the current note is passed, while "Dense Layer Duration" refers to the size of the dense layer through which the scalar-valued current duration is passed. Note that as mentioned in Section 3.2, we match these dimensions for numerical stability.

In order to choose between these hyperparameter settings, we trained our original RNN architecture (with all of these configurations) on our dataset, mentioned in Section 4 — we ended up training the first and fifth configurations for 300 epochs, and the rest for 100 epochs (we trained till accuracy flatlines). To ultimately choose the best configuration, we sampled a single piece from each trained model and chose the model which produced the most pleasing and realistic music.

Qualitatively, we ended up liking the pieces produced by the second and fifth models the best, and chose the second configuration. However, the second model had only 59% training accuracy, while models 3 and 4 had training accuracy 74% and 69% respectively. While this is not surprising (models 3 and 4 have more parameters) it does show that there is not a direct correlation between training accuracy and the quality of music generated, since we preferred the piece generated by setting 2.

The training plots for the second configuration are shown in Figure 6. The sample this model generated is at: `http://bit.ly/ogrnn`

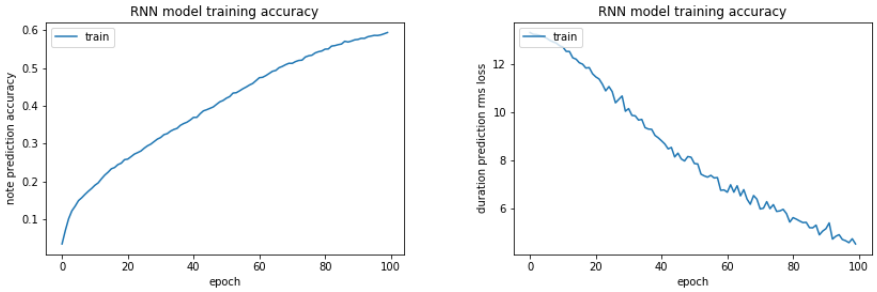

Figure 6: The first plot shows the gradual increase in training accuracy for the softmax output of our original RNN, as it is trained with the second configuration. The second shows the decrease in the mean squared error loss for the duration predictions.

5.1.3 ChunkRNN

We performed a similar procedure for the other two RNN architectures. Our sampling strategy with ChunkRNN was slightly different — we first sample C notes (where C is the size of a chunk), and then generate a series of chunks. We considered the hyperparameter configurations shown in Table 5.1.3.

Config No.	Chunk Size	Sequence Length	Dense Encoder 1	Dense Encoder 2	LSTM	Dense Decoder 1	Epochs
1	5	30	200	150	150	200	2000
1	10	30	300	200	200	300	3000
3	5	60	200	150	150	200	2000
4	10	60	300	200	200	300	3000

Here, "Chunk Size" refers to the number of notes that are generated at a time by the RNN, while the Dense Encoder values are defined in 5.1.2. Observe that the output dimension of the dense layer right after the LSTM, as well as that of the dense layer right before our multiple softmax layer are fixed, since the input dimension of the first layer of the decoder must match the output of the second encoder layer.

One thing we observed is the large number of epochs required to effectively train the Autoencoder and ChunkRNN. When we performed preliminary experiments with 100 epochs, the models had very low training accuracies and did not produce pleasing music. However, training for a larger number of epochs caused their training accuracies to steadily increase. We hypothesize that this is due to the comparatively larger number of parameters present in this model (particularly due to the larger input size). Out of all the pieces sampled, we preferred the one generated by the second hyperparameter configuration shown in the table above.

The training accuracies for this configuration are shown in Figure 7. Observe that a few thousand epochs are required for the accuracy to increase appreciably. Another subtle point is that the accuracy of these two networks is measured using Keras's cross-entropy loss function applied to our multiple-softmax layer — however, cross-entropy is typically only used for probability distributions, while our output is a vector which forms multiple probability distributions. Therefore, the accuracy metric is less meaningful in this case, but still gives us a proxy for how well the models train.

Finally, we observed that the piece we sampled from this model (using the second hyperparameter setting) satisfied several key properties of musical pieces which we did not explicitly train the model for — for example, this piece seemed to have a fixed key. It is difficult to explicitly constrain a piece generated by a deep learning model to have a certain key. The piece generated by this model is at: `http://bit.ly/chunkrnn`

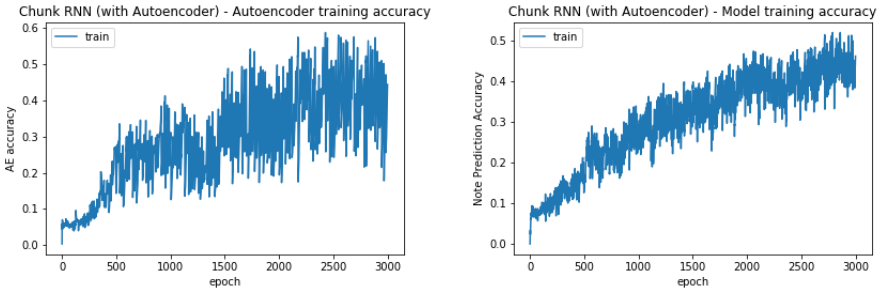

Figure 7: The first plot shows the gradual increase in training accuracy for the multiple softmax output of the autoencoder, while the second shows the training accuracy for the multiple softmax output of ChunkRNN.

5.1.4 RhythmRNN

The hyperparameter settings we considered for RhythmRNN are:

Sequence Length	Dense 1	Dense 2	Dense 3	Epochs
10	20	20	20	100
20	20	20	20	100
10	40	30	20	100
20	40	30	20	100

We do not tune the parameters of the LSTM (such as layer dimensions or learning rate) — instead, we use layer dimensions from our best hyperparameter setting for our original RNN architecture. We do this because the RNN used here is essentially our original RNN, but without any duration inputs or outputs. In our experiments with RhythmRNN, we used a fully-connected neural network with three dense layers — to the last dense layer, we apply another dense layer (with 1 output) to find the probability of the next item in our piece being a note (versus being a rest), and we apply another dense layer to the third dense layer to find the duration of the next item in our piece. Figure 8 shows the prediction accuracy for the fully connected network (for rhythm prediction and generation) and the RNN (for melody prediction and generation). We chose the fourth hyperparameter configuration — the sample it generated is at: `http://bit.ly/rhythmrnn`.

The other configurations gave samples that consisted of extremely slow notes. It may be the case that the fully connected network sometimes gives a disproportionately high probability to rests, due to the fact that

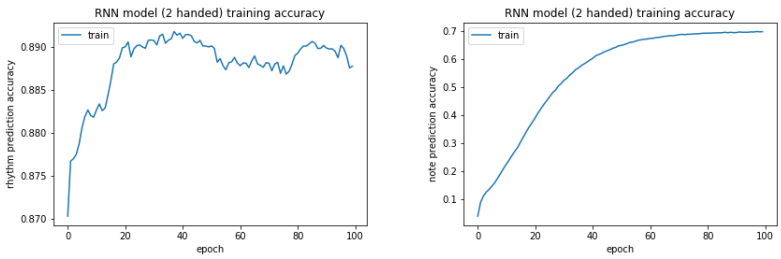

Figure 8: The first plot shows the gradual increase in training accuracy for the multiple softmax output of the autoencoder, while the second shows the training accuracy for the multiple softmax output of ChunkRNN.

the outputs are 0 and 1. A potential solution in the future is to geometrically decay the probability of a rest occurring.

5.2 Evaluation

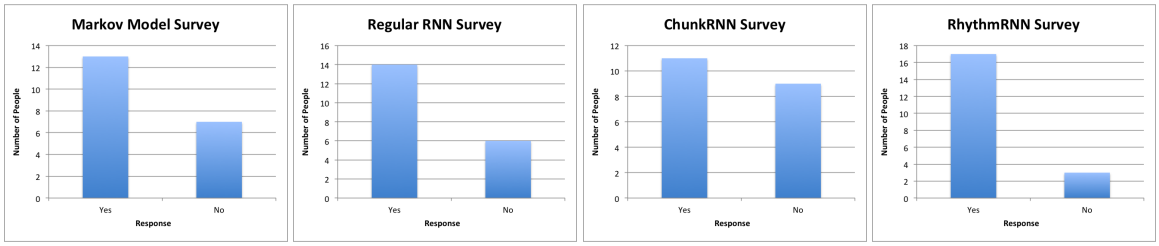

Figure 9: Charts show number of people on y-axis and yes/no answers on x-axis. (1) Survey for Markov (Baseline) Model, (2) Survey for Regular RNN, (3) Survey for ChunkRNN, (4) Survey for RhythmRNN.

As an evaluation metric, we performed a survey to evaluate how pleasing these pieces are to the human ear, and whether they would qualify as 'credible' music. We surveyed 20 CMU students who were given 1 sample from each model and asked to determine whether they thought the sample qualified as a real piece of music (for the RNN models, we used the samples we took from the best hyperparameter configurations shown above). We found that RhythmRNN performed the best (considering the percentage of respondents who voted 'Yes', which was 85%), and significantly outperformed the baseline. Meanwhile, our other two RNN methods underperformed the baseline. In our opinion, getting a positive response larger than 50% for all our approaches is a great sign, and makes it seem possible that machine music generation will become an everyday reality.

6 Conclusions, Limitations and Future Work

Crucially, we note that evaluating music is **highly** subjective, and by asking for music credibility, we attempt to achieve our objective of evaluating closeness to human-composed music. In fact, we (authors) thought that the sample generated by the original RNN was the most pleasing. Therefore, not being able to evaluate our models in an objective manner is an inherent limitation of this study. We believe we succeeded in our goal of finding original methods to create realistic sounding music, at least through the success of RhythmRNN in the survey.

One potential direction for future work is to explore reasons for ChunkRNN's relative lack of success. One approach to resolve this may be to replace one-hot encodings with many-hot encodings (where an input vector has dimensionality equal to the number of *notes*, and a chord is instead represented by having 1s in multiple locations). Learning distributions over many-hot vectors is a nontrivial task, however — one approach using RBMs is given in [3].

References

[1] Nir Arbel. *How LSTM networks solve the problem of vanishing gradients*. URL: https://medium.com/datadriveninvestor/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577.

[2] Yoshua Bengio et al. "Greedy Layer-Wise Training of Deep Networks". In: *Advances in Neural Information Processing Systems 19*. Ed. by B. Schölkopf, J. C. Platt, and T. Hoffman. MIT Press, 2007, pp. 153–160. URL: http://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf.

[3] 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 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. 2012. URL: http://icml.cc/2012/papers/590.pdf.

[4] Mason Bretan, Gil Weinberg, and Larry P. Heck. "A Unit Selection Methodology for Music Generation Using Deep Neural Networks". In: *CoRR* abs/1612.03789 (2016). arXiv: 1612.03789. URL: http://arxiv.org/abs/1612.03789.

[5] Bhairav Chidambaram. *FakeBach*. URL: https://github.com/bchidamb/FakeBach.

[6] Michael Scott Cuthbert and Christopher Ariza. "music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data". In: *11th International Society for Music Information Retrieval Conference (ismir 2010)2* (2010), pp. 637–642. URL: http://ismir2010.ismir.net/proceedings/ismir2010-108.pdf.

[7] Douglas Eck and Juergen Schmidhuber. *A First Look at Music Composition Using LSTM Recurrent Neural Networks*. Tech. rep. 2002.

[8] Dumitru Erhan et al. "Why Does Unsupervised Pre-training Help Deep Learning?" In: *J. Mach. Learn. Res.* 11 (Mar. 2010), pp. 625–660. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1756025.

[9] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *arXiv e-prints*, arXiv:1308.0850 (Aug. 2013), arXiv:1308.0850. arXiv: 1308.0850 [cs.NE].

[10] Maximilian Kohlbrenner et al. "Pre-Training CNNs Using Convolutional Autoencoders". In: (). URL: https://pdfs.semanticscholar.org/e1ab/3b9dee2da20078464f4ad8deb523b5b1792e.pdf.

[11] Alvin Lin. *Generating Music Using Markov Chains*. Nov. 2016. URL: https://hackernoon.com/generating-music-using-markov-chains-40c3f3f46405.

[12] Dimos Makris et al. "Combining LSTM and Feed Forward Neural Networks for Conditional Rhythm Composition". In: *Engineering Applications of Neural Networks*. Ed. by Giacomo Boracchi et al. Cham: Springer International Publishing, 2017, pp. 570–582. ISBN: 978-3-319-65172-9.

[13] Yosua Giovanni Pratama Polii, Tito Waluyo Purboyo, and Randy Erfa Saputra. "An Experiment of Markov Chain and N-Gram Methods for Reconstructing a Music". In: *Journal of Engineering and Applied Sciences* 15.3 (2020), pp. 876–880.

[14] Skúli Sigurgeirsson. *Classical-Piano-Composer [Dataset]*. URL: https://github.com/Skuldur/Classical-Piano-Composer/tree/master/midi_songs.

[15] Sigurour Skuli. *How to Generate Music using a LSTM Neural Network in Keras*. URL: https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5.

[16] Felix Sun. *DeepHear - Composing and harmonizing music with neural networks*. URL: https://fephsun.github.io/2015/09/01/neural-music.html.

[17] will108. *Keras-Music Generation*. URL: https://github.com/will108/Keras-Music-Generation/blob/master/Music20Composition.ipynb?fbclid=IwAR0xcoehqrOrLQpqVOKumy_baHkNrfrJ018Kx0Sr7jZ0oLqF8vd0d61XVYY.

[18] Pengfei Zhu, Xin Li, and Pascal Poupart. "On Improving Deep Reinforcement Learning for POMDPs". In: *CoRR* abs/1704.07978 (2017). arXiv: 1704.07978. URL: http://arxiv.org/abs/1704.07978.

7 Use of Found Code

We made use of code from the following repositories:

- FakeBach (https://github.com/bchidamb/FakeBach) — we used the code in this repository as a baseline, with which to compare the models we created.
- Keras-Music-Generation (https://github.com/will108/Keras-Music-Generation/blob/master/Music%20Composition.ipynb) We use the code in this repository for processing of MIDI files using Music21, and we also based our general training strategy (that of using a fixed number of notes at a time as input to the RNN) on the training strategy used here.
- How to Generate Music using a LSTM Neural Network in Keras (https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5) We used this source's suggestion of

partitioning MIDI files by instruments in order to recover the main track — otherwise, the training examples will not form a good melody.