Fiddler: Irish Music Composer

Manthan Thakar, Rashmi Dwaraka,Tirthraj Parmar
College of Computer and Information Science, Northeastern University
{thakar.ma, dwaraka.r, parmar.t}@husky.neu.edu

Abstract

The goal of the project is to model music compositions by capturing temporal dependencies in classical music compositions and eventually generate novel music compositions from the learned model. We use approximately 24000 music compositions transcribed in ABC notation. We implement count-based n-gram language model, use its results as a baseline for recurrent neural network based methods and assess their ability to generate structurally coherent, human-pleasing music. We achieved test accuracy of 69.78% for char-RNN and 73% for seq2seq model with both methods generating valid music compositions.

Introduction

Music compositions can be viewed as sequential ordering of musical notes such that they collectively form pleasant music. Musical notes within a composition generally tend to depend on preceding notes in the same composition. Moreover, music compositions follow a well-defined structure through the combination of keys, time signatures and note lengths.

Due to the sequential and structural characteristics of music compositions, some of the methods applied on natural languages have been successfully applied to model music composition (Sturm et al. 2016). Therefore, we experiment with n-gram and recurrent neural network based methods on music composition modeling problem. Moreover, we propose and evaluate the efficacy of encoder-decoder based Sequence to Sequence learning method, which has been successfully applied on machine translation, speech recognition and text summarization (Sutskever, Vinyals, and Le 2014). Furthermore, we use these statistical models to generate music compositions and assess their validity as well as their ability to please human ears.

For the purposes of our implementations, we divide the music generation problem in two tasks:

1. Build a generative model for probability distributions of characters in the music compositions transcribed in ABC notation using:
   - Character-level N-Gram Language model
   - Recurrent Neural Network (RNN) based Language model
   - Encoder-Decoder based Sequence to Sequence (Seq2Seq) method
2. Sample text from the generative model to create novel compositions

Background

Language Models

Language models (LM) learn the joint probability function of a sequence of words or characters. Given the textual nature of ABC notated music transcripts, we can train a character-level language model that learns the probability of observing a musical note at a position i, given all its previous notes (1 to $i-1$). This kind of music transcription modeling can be used to generate new music notes. However, the degree of learning from the history depends on the choice of the model.

The intuition behind the N-gram LM is to assign probability to a sequence of events in order to predict the next possible event based on a given sequence. A simple character-level N-gram model derives the probability distribution with Markov assumption and maximum likelihood estimation based on character frequencies. Such model only takes into account the context of previous $N-1$ characters, ignoring any dependencies beyond that. This fails the model to understand the dynamic nature of sequential dependencies. Moreover, the model requires smoothing to predict based on unseen sequence.

Though N-gram LM is simpler in its implementation, the model size can grow exponentially as N increases. Moreover, N-gram LM also suffers from the sparsity problem as the context sequences get longer.

Recurrent Neural Network

Recurrent Neural Networks are an extension of feed-forward neural networks, whereby the weights in the hidden layers are reused. This architecture of RNNs can be viewed as sharing of weights across time steps. This cyclic graph structure of RNNs, make them an excellent candidate for capturing temporal dependencies of any length. In fact, Language modeling tasks have been significantly improved by using Recurrent Neural Networks [Chelba, Norouzi, and Bengio].

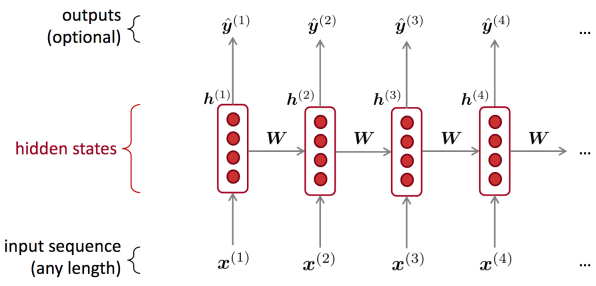

Figure 1: Recurrent Neural Network based Language Model

Figure 1 shows the topology for an RNN-based language model.

At each time step i, input x^i is fed to the network and the network generates probability distribution $\hat{y}^i$ for the next token, given context sequence of length i. x^i can either be a one-hot encoded vector with size $|V|$, where $|V|$ is the size of the vocabulary, or it can be word-embedding. h^i denotes hidden layer weights at each time step i. Note that, these weights are shared across time steps which lets the network keep historical context for long sequences. This gives RNN based LM greater flexibility since the context window is not fixed. Moreover, the model size doesn't increase with the increase in input sequence length.

The hidden states are obtained by using the following transformation:

$$h^t = W_h h^{t-1} + W_e e(x^t) + b$$

Here, h^t is the hidden layer state at time step t, W_h are weights associated with hidden layer, h^{t-1} denotes hidden state at time step $t-1$, W_e denotes weights associated with input, $e(x^t)$ denotes the transformation applied on input (i.e. one-hot encoding, word embedding) and b denotes bias.

In order to calculate probability distributions ($\hat{y}^i$) at time step i, Softmax layer is applied to the output of hidden state.

$$\hat{y}^t = Softmax(Uh^t + b_s)$$

Here, Softmax function can be given as:

$$Softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

U denotes the weights associated with last output from RNN hidden state h^t and b_s denotes bias associated with the softmax layer.

The above formal definition describes how RNN-LM generates probability distributions at each time step given previous tokens. In order to learn the correct distributions, we must back-propagate the error to previous time steps to update the shared weights W_h. Therefore, the loss function at time step t is defined as the cross-entropy between predicted $\hat{y}^t$ and the true distribution y^t and can be formalized as:

$$J(\theta) = -\sum_{j=1}^{|V|} y_j^t log(\hat{y}^t)$$

During training, we back-propagate the errors in time to update W_h by taking the derivative of loss function $J(\theta)$ with respect to hidden layer weights W_h:

$$\frac{\partial J^t}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^t}{\partial W_h}|i$$

It is evident from the above equation that the gradient with respect to W_h is summation of the gradient at each time step it appears. When the sequences are very long, this can cause exploding or vanishing gradients, which can affect the learning conversely.

Therefore, in our implementation we experiment with LSTM and GRU cells in the hidden layer to avoid RNN's gradient problem. Note that, apart from changing the cells in RNN, we use similar topology and loss function as described in this section.

Sequence to Sequence Learning

Recurrent Neural Networks have been able to achieve excellent performance with language modeling tasks, where we generate probability distribution for the current token given $n-1$ previous tokens. Despite its power, it is not straightforward to apply this method when the targets are sequences of varying lengths. This can be a significant limitation for Sequence modeling problems like Machine Translation.

(Sutskever, Vinyals, and Le 2014) proposed RNN based sequence learning method, which has produced state of the art results for sequence learning tasks like Neural Machine Translation.

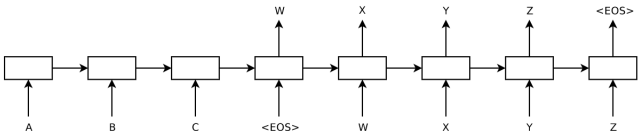

Figure 2: Sequence to Sequence Learning

The sequence to sequence architecture can be divided into two parts based on their contribution in sequence learning, encoder and decoder. Both Encoder and Decoder are LSTM cell based RNNs. Encoder is responsible to represent input vector as a thought vector by feeding the input sequence through Encoder LSTM network. The hidden state of Encoder is then used to initialize the hidden states of Decoder RNN. The Decoder RNN is then responsible for predicting the output sequence given the thought vector for input sequence.

Figure 2 shows the graphical representation of the architecture of a Sequence to Sequence model. Here, the input sequence "ABC" is fed through the encoder to obtain thought-vector. This thought-vector is then fed to the decoder to

initialize its hidden states. Consequently, the decoder attempts to generate the target sequence "WXYZ" using similar mechanism as described in RNN language models. Note that, the decoder input sequence is padded with a special symbol $< EOS >$ which marks the end of the encoder input sequence and signals the decoder to initiate decoding. Moreover, decoder target sequence (here, "WXYZ") is shifted to right by one time step and padded with $< EOS >$ to obtain decoder input sequence.

Related work

(Bengio et al. 2003) proposes a neural language model (NLM) that leverages the learned distributed representation of each word to fight the curse of dimensionality in language models. The same model can be used at character level as well. However, since our corpus has only 111 unique tokens, it doesn't suffer from the curse of dimensionality. Moreover, our model doesn't require to understand similarities between characters as they are unique music notes. To improve on the simple N-gram model, we need a better modeling of the musical sequences of dynamic length. Therefor, we do not use a character-level version of the NLM proposed by (Bengio et al. 2003) for our solution.

(Kim et al. 2016) propose a character-level convolution neural network (CNN), whose output is used as an input to a long short-term memory (LSTM) recurrent neural network language model (RNN-LM). The use CNN helped the model to differentiate character n-grams between prefixes, suffixes and others. However, ABC notation does not posses any such properties to detect. Therefore, we do not use CNN and directly implement LSTM for modeling.

Dataset

ABC notation is a shorthand form of musical notation. Our dataset includes approximately 24000 tunes transcribed in ABC notation. Each character in the notation is used to represent the notes A-G and various characters to capture the length of the note, key and title of the tune. ABC notation helps us apply the same language model that is used in text generation to generate music.

The collated dataset includes 111 unique vocabulary of tokens forming a corpus of 14913399 number of tokens. The average length of the tune in the dataset is 607 characters.

The language model requires sequence of encoded characters as input to train. Given, the batch size and step size of the cell, we need to encode the music notation to input sequence (X) and corresponding output sequence (Y). We perform the input encoding at character level and sequence level. Consider the below sample tune in ABC notations:

```
<s>
T: Bardes \n
M:6/8 \n
K:Cmin \n
G F G A 2 G | F 2 E 2 \< D 2 | B, C D E D C |
</s>
```

Figure 3: Sample Tune

We use the text-based ABC notation format as our dataset, since it is readily compiled and preprocessed. Moreover, it allows us to leverage some of the advanced natural language processing methodologies.

Character level encoding

We add a start symbol $< s >$ and an end symbol $< \backslash s >$ to each tune. This enables the network to learn how the tunes are structured given sequence of characters as input.

For a given step size of the input which is set as text window for encoding purposes. For each s characters (0 to s) in input sequence, the output sequence is formed by (1 to $s+1$) characters. The text window is shifted 1 character at a time. For batch size 10 and step size 5, character level encoding of the above sample tune can be processed as below.

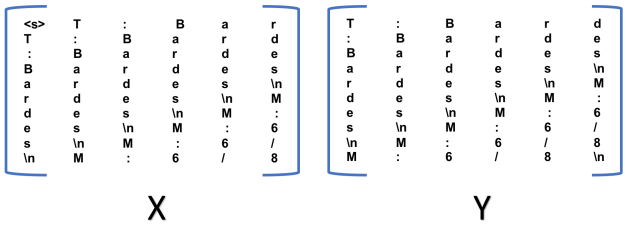

Figure 4: Character level encoding

Sequence level encoding

Similar to Character level encoding, we add start and end symbol. Based on the intuition that the title of the tune do not contribute to the quality of the music, we eliminated the title line from the input ABC notations from the dataset. Unlike character level encoding, here we shift the text window by sequence of characters per encoding. For each s characters (0 to s) in input sequence, the output sequence is formed by (k to $s+k$) characters. Here, k is the length of the shift. After various trails, we set the shift size as 3 characters. For batch size 10 and step size 5, we encode the sample tune as below.

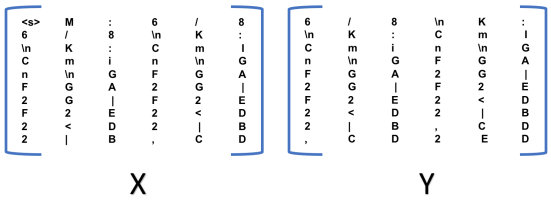

Figure 5: Transposed Sequence encoding

Project Description

In this section, we describe the methodologies used to train statistical models for music transcriptions.

Character-level N-Gram Language Model

An N-gram is a sequence of N characters in ABC notation. The model training consists of calculating probability of the

last character of an N-gram given previous $N-1$ characters for all N-grams:

$$P(c_n|c_1^{n-1}) \approx P(c_n|c_{n-N+1}^{n-1})$$

The dependence on only $N-1$ previous characters comes as a result of N^{th} order Markov property. With MLE N-gram parameter estimation, the conditional probability can be computed based on counts from the corpus as:

$$P(c_n|c_{n-N+1}^{n-1}) = \frac{C(c_{n-N+1}^{n-1}c_n)}{C(c_{n-N+1}^{n-1})}$$

To separate tunes in the data, for each tune we prepend $N-1$ start tokens ('$'), and append a single end token ('#'). The model is then trained for all N-gram blocks derived from the entire dataset of tunes. Our trained model can be evaluated based on Perplexity and Accuracy for the test data. The Perplexity of an N-gram model for test set T containing K characters is calculated as:

$$PP(T) = P(c_1c_2...c_K)^{-\frac{1}{N}}$$

A sequence in test set which is not seen during training gives zero probability. We need to take into account smoothing to avoid this problem. We have implemented add-k smoothing which modifies conditional probabilities used to calculate perplexity as follows:

$$P^*_{Add-k}(c_n|c_{n-N+1}^{n-1}) = \frac{C(c_{n-N+1}^{n-1}c_n) + k}{C(c_{n-N+1}^{n-1}) + kV}$$

Character-level RNN Language Model

In order to capture long dependencies in music compositions, we implement character-level RNN language model with both LSTM and GRU cells.

The architecture of the network, shown in Figure 6, resembles RNN LM (described in background). At each time step, we input ABC notation data encoded with integer ids, which is then converted into embedding. Next, the output distribution for next notes is produced from the given input sequence using the final Softmax layer. Cross-entropy loss function is used to calculate the errors in the output $\hat{y}^t$ using which W_h is updated. To optimize the learning we use Adam Optimizer, which updates the hidden layer weights W_h such that the cross-entropy loss function $J(\theta)$ is minimized.

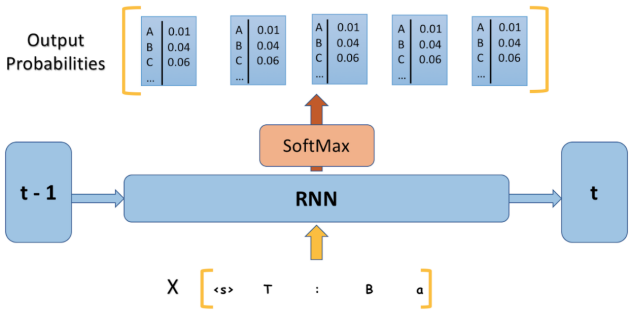

Figure 6: char-RNN Architecture

As mentioned, we train character-level RNN LM with two configurations of cell types: LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit). The architecture of LSTM and GRU is shown in Figure 7. Due to the gated structure of these cells, the network assigns weights to previous context and selectively forgets some of the previous states. Therefore, the risk of vanishing gradients is significantly minimized. As can be seen, GRU has fewer transformations on hidden states and is hence computationally more efficient than LSTM.

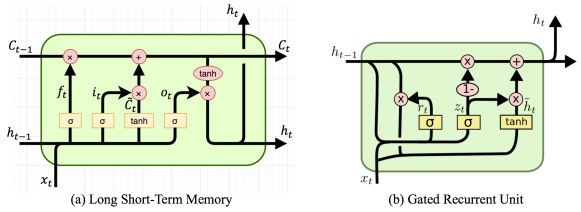

Figure 7: LSTM and GRU Cells

Candidate models with different configuration of layers are also trained. We also vary other hyper parameters like learning rate, number of time steps to unroll and dropout.

Sequence to Sequence method

Both of the previous methods focus on building a language model over music transcriptions, which produces probability distribution for n^{th} character, given historical context of $n-1$ characters. But since music compositions can also be thought of as sequence of notes that depend on previous sequences (bars), we hypothesized that Sequence to Sequence model is a better fit for the task.

Therefore, we apply Sequence to Sequence method on music transcriptions. Similar to the language translation problem, the input sequence is a sequence of $n1$ music notes, while the target output sequence is $n2$ music notes. Both the Encoder and Decoder consist of LSTM cells (figure 7).

The encoder takes $n1$ music notes and feeds them forward through the Encoder LSTM cells which has a shared state similar to an RNN. In the original implementation of Sequence to Sequence model, the authors (Sutskever, Vinyals, and Le 2014) suggest that feeding the input sequence improves the performance, we reverse the input sequence before feeding it to the encoder. This state h^t_{enc} is updated at each time step t similar to a recurrent neural network. At the end of $n1$ time steps, the decoder RNN is initialized with the final encoder hidden state. The final hidden state of the encoder can be thought of as a thought-vector, which represents the previous $n1$ music notes as a vector. The decoder can be thought of as a character-level RNN language model described above. The decoder receives a special $< GO >$ token in its first time step and subsequently receives target sequence of length $n2$.

We train Sequence to Sequence models with different configurations of size of hidden states, directionality of RNN cells and embedding sizes.

Generation

In this section, we describe the methods used to sample text from generative models of music transcriptions.

Models based on LM

With language model based models (character-level N-gram and character-level RNN), we model the joint distribution over characters in ABC notations. A tune can be generated using these probabilities as follows.

context ← beginning context of the tune
repeat till the end of tune character:
  token ← random character from $P(character|context)$
  append *token* to *output*
  update *context* based on token

Models based on Seq2Seq

With Seq2Seq models, we generate music by feeding the encoder warm up characters. A thought vector thus produced is used to initialize decoder hidden states. After which, the music is sampled from the decoder using the same algorithm described above.

Experiments

N-Gram Language model

For N-gram LM, we used perplexity and accuracy as evaluation metrics. The perplexity is calculated using add-k smoothing with $k = 0.1$. We use these evaluation matrices to determine N by evaluating N-gram models for $N = 2$ to 20.

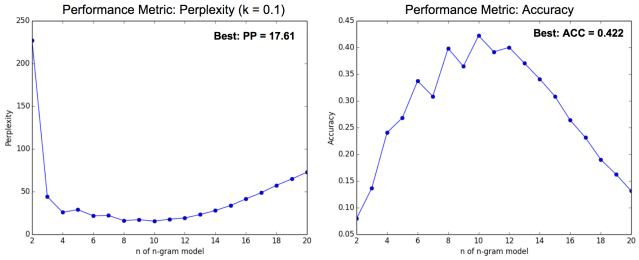

Figure 8: Performance of N-gram model for varying N

Figure 8 shows the perplexity and accuracy results for different N-gram models. As it can be observed, both short and long context models give comparatively poor performance. The best model (10-gram) gives an accuracy of 42.2%, which does not seem good enough. For small N, the model lacks contextual data. For large N, the model suffers from the sparsity problem. This explains the performance of 10-gram model as it captures longer context than smaller N and doesn't suffer heavily from the sparsity problem as opposed to larger N.

Based on this experiment, we chose 10-gram model for our final Model. Overall, the N-gram LM does not give promising accuracy due to its limitations in capturing the temporal dependencies. However, it is a good candidate as a baseline for the subsequent models we have implemented.

Char-RNN Language model

Using both variants of the LSTM and GRU cells for the RNN, we analyzed the effects of the hyper parameters of RNN. In each experiment, the RNN was trained for 50 epochs for 80% of the dataset and 20% of the dataset was used for testing the trained model. Also, we integrated generate module to our code base, which can be used to generate tunes from the trained model. We generated various tunes and evaluated the music quality for any unstructured notes or errors in the tune generated.

Note that, Char-RNN is able to generate valid tunes with properly delimited bars. Moreover, it also picks correct notes based on the key of the song. However, it fails to generate tunes that have recurring themes in them.

Effects of Context Length

The analysis of the effects of context length suggests that larger-context language model improves the language model by capturing the theme of the tune better. Figure 9 shows the accuracy of the training model with varied context length. We can notice that that the accuracy increases with the increase in context length.

Based on this experiment, we set the context length as 20 for our final model.

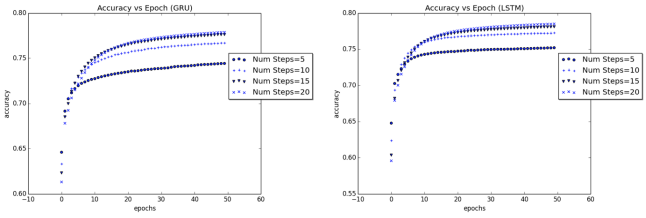

Figure 9: Effects of varying Context length

Effects of Learning rate

The deep learning models rely on stochastic gradient descent optimizer. If the learning rate is low, the training is more reliable, but will take lot of time to optimize. If the learning rate is high, the training may not converge or it might diverge. Weights changes overshoots the minimum and increases the loss. This effect can be noticed in the figure 10. It shows the effects of learning rate on the accuracy of the training model.

Based on this experiment, we set the learning rate as 0.001 for our final model.

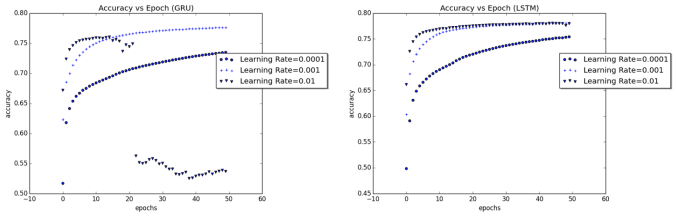

Figure 10: Effects of varying Learning rate

Effects of the number of layers in the network
The number of layers of the Recurrent network does not has much effect on the accuracy of the training model as shown in the figure 11. We trained the model with 3 layered neural network, but did not achieve good accuracy for the model. Hence the experiment was restricted to 2 layers. Based on the analysis, we set the number of layers as 2 for our final model.

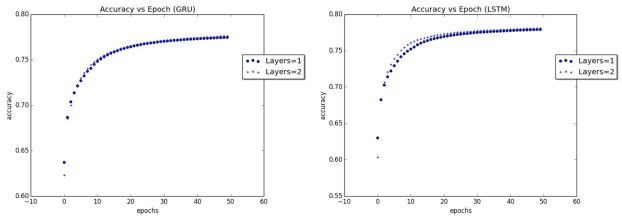

Figure 11: Effects of varying number of layers

Effects of Dropout layer
In order to reduce the effects of any over-fitting of the model, we included a dropout wrapper to the network. The dropout value is a decimal between 0 (no dropout) and 1. Based on the dropout parameter, certain input and recurrent connections to the RNN cells are excluded with a probability of $(1 - d)$ from activation and weight updates while training the network. We conducted few experiments by varying the probability by which the regularization is applied to the model. Based on the results, we analyzed that the lower dropout (d) decreased the training accuracy of the model, but achieved good test accuracy. We set the dropout parameter as 0.8 for final model.

Seq2Seq method

Figure 12 shows that by increasing the number of hidden states we achieve higher accuracy. By setting the number of hidden states as 150, we can see the effects of directionality on the training accuracy from the figure 12. With bi-directional Seq2Seq method using 150 hidden states, we can achieve good accuracy.

Figure 13 shows a music composition generated using Seq2Seq model. Note that, not only does the Seq2Seq Model learn to compose a valid ABC notation for the tune it also creates bars that are aligned with the tune's time signature. Moreover, the tune contains some of the themes that are repeated throughout the song which is closer to how a human would compose a tune.

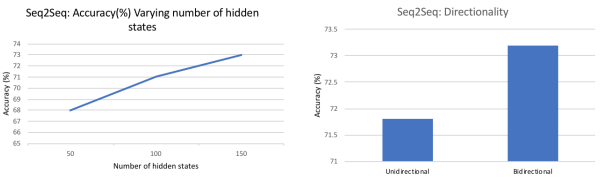

Figure 12: Seq2Seq evaluation

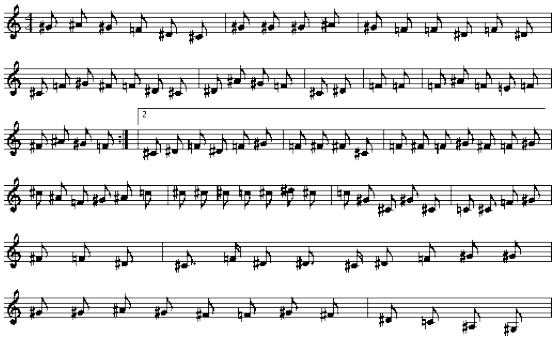

Figure 13: Composition generated using Seq2Seq Model

Test Accuracy:

Based on our experiments, we set the hyper parameters to the optimal values and achieved the highest test accuracy of 73.4% for Seq2Seq method. The character RNN Language model performed with 69.7% accuracy and whereas with N-gram model, we were able to achieve 42.2% accuracy.

Table 1: Test Accuracy of the models

N-Gram	42.2%
char-RNN	69.7%
Seq2Seq	73.4%

Evaluation

In order to evaluate the quality of the music generated, we conducted a user survey which involved varied audience. The audience was requested to evaluate whether a particular tune is perceived as human generated or computer generated. The user can evaluate this by measuring the pleasantness of the tune and look for some well-defined structures in the notes.

To reduce selection bias that can be introduced by relative placement of tunes and other reasons, we 2 surveys with different orderings were designed. The tunes generated using models described above were used along with real human generated tunes. The total tune duration for the survey summed to 3.9 minutes.

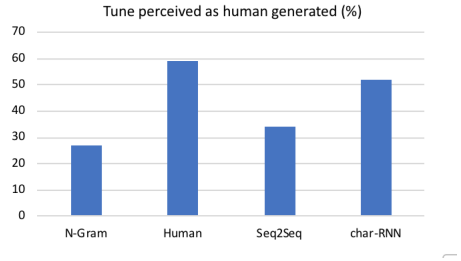

Figure 14: User Survey Evaluation Summary

The figure 14, represents the summary of the results obtained from the survey taken by 134 people. Each user typically spent 3.15 minutes on the survey, which is very close to

the total duration of tunes included in the survey. It indicates the percentage of users who perceived the tune generated by the model as human generated. It can be seen that 59% of the users were able to correctly identify human generated tune.

Among the computer generated tunes, character-level RNN was the most successful in fooling users to believe that the tune is human generated with 52% of surveyors thinking the tune is created by a human.

Although, we hypothesized that Seq2Seq is better suited for producing tunes with recurrent structures or more coherence, unfortunately that is not evident in the survey results. This may be because we only ran seq2seq for 3 epochs due to computational restrictions.

Conclusion

With the results obtained with various methods, we showed that the natural language methods applied to music transcriptions produce promising compositions. Although, language model based methods learn to produce valid compositions, those compositions still lack recurring themes which are prevalent in classical music.

In order to model recurring themes within a composition, we implement Sequence to Sequence model. Although, Sequence to Sequence model is successful in producing recurrent themes that are either reused or improvised on, it is still far from human created tunes as indicated by the survey results. We also believe that the survey results obtained with character RNN are reasonable.

In the future, we would like to experiment with attention based Sequence to Sequence method to obtain a better model for recurrent themes within classical compositions. We would also be interested in experimenting with MIDI files and apply recurrent neural network based methods on them.

Appendix

Below are the sample tunes generated using various methodologies and used in the user survey for evaluation.

- **N-Gram model**
  https://soundcloud.com/manthan-thakar/tune-1
- **Human Generated (Part of the dataset)**
  https://soundcloud.com/manthan-thakar/tune-2
- **Seq2Seq method**
  https://soundcloud.com/manthan-thakar/tune-3
- **LSTM based char-RNN LM**
  https://soundcloud.com/manthan-thakar/tune-4
- **GRU based char-RNN LM**
  https://soundcloud.com/manthan-thakar/tune-5

References

Bengio, Y.; Ducharme, R.; Vincent, P.; and Jauvin, C. 2003. A neural probabilistic language model. *Journal of machine learning research* 3(Feb):1137–1155.

Chelba, C.; Norouzi, M.; and Bengio, S. 2017. N-gram language modeling using recurrent neural network estimation. *arXiv preprint arXiv:1703.10724*.

Kim, Y.; Jernite, Y.; Sontag, D.; and Rush, A. M. 2016. Character-aware neural language models. In *AAAI*, 2741–2749.

Sturm, B. L.; Santos, J. F.; Ben-Tal, O.; and Korshunova, I. 2016. Music transcription modelling and composition using deep learning. *arXiv preprint arXiv:1604.08723*.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.