6.867 Final Project: Learning Some Tonal Music (LSTM)

Tiancheng Qin
MIT
tcqin@mit.edu

Eshaan Nichani
MIT
eshnich@mit.edu

Michael Kural
MIT
mkural@mit.edu

1 Introduction

In this paper we utilize recurrent neural nets (RNNs) and long short-term memory (LSTM) models to compose classical music. Presently there has already been much work done on deep neural net music generation: Doug Eck has used LSTMs to improvise blues music [5], and researchers at Google have developed Magenta [1], a project using neural networks for the creation of art and music. The aim of our project is to develop our own implementation of an LSTM for the purpose of single melody music generation, and experiment with important musical qualities of pieces such as time signature, beat structure, phrasing, pitch, and duration.

2 RNNs and LSTMs

In recent years, the field of machine learning has seen tremendous successes in certain areas, largely due to the application of deep learning and neural networks to large data sets. In the domains of (possibly time-ordered) sequences of data such as language or music, a specific deep learning model known as a **recurrent neural network (RNN)** has been seen to be a particularly effective predictor. If we were to simply feed in all elements of an ordered sequence into the input layer of an ordinary neural network, this would cause multiple problems: first, the size of the dataset is too large, and second, this would ignore the temporal information given by our sequential data. A recurrent neural network solves this problem by summarizing all previous data at a given point of time in a hidden state.

2.1 RNN Details

We describe the structure of a shallow RNN which makes predictions based on an input vector sequence $(x^1, x^2, \cdots x^T)$. At each time-step t, the architecture computes a hidden state vector h^t based on some function of h^{t-1}, x^t, and weights θ to be trained:

$$h^t = f(h^{t-1}, x^t, \theta)$$

for some function f.

In a simple RNN, this will take the form

$$h^t = \tanh(W_h h^{t-1} + W_x x^t + b_h)$$

for some weight matrices W_h, W_x and bias vector b_h subsumed by θ (all to be trained).

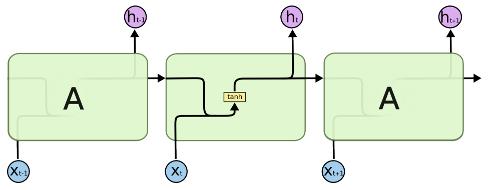


Figure 1: A diagram of an “unrolled” simple recurrent neural net (RNN) [7].

Furthermore, at each step t the RNN will output a value y^t based on the hidden state. This transformation is governed by further weight parameters W_y, b_y [6]:

$$y^t = s(W_y h^t + b_y)$$

where s denotes the softmax activation function. This specific setup is also known as the **Elman network**.

Note that the matrices and vectors W_h, W_x, b_h, W_y, b_y are shared across all time-steps; this is one of the most crucial features of the RNN that distinguishes it from a feedforward network. We can generalize this to a **deep RNN** in which again parameters are shared across time-steps t and apply it to the task of sequence prediction.

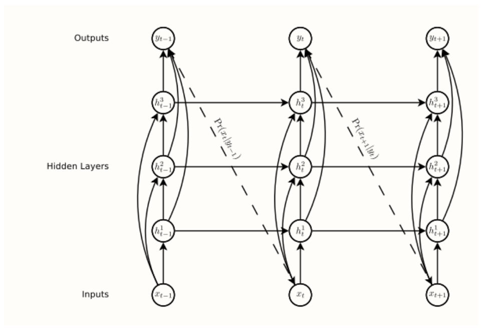


Figure 2: A deep recurrent neural network for for sequence prediction, given in [4].

In our general setup (we do not specify an implementation here because we will later describe an implementation using the LSTM), the time-shared parameters in θ are used to compute hidden states $h_1^1, h_1^2, \cdots h_1^T$ from $x^1, \cdots x^T$, followed by hidden states $h_2^1, \cdots h_2^T$, until finally hidden states $h_L^1, \cdots h_L^T$ and outputs $y^1, \cdots y^T$ are computed. We would like to train our parameters θ such that our recurrent neural network gives outputs y^t which yield predictive probability distributions $\mathbb{P}(x^{t+1}|y^t)$. More precisely, if the x^t are given **one-hot-encoded** inputs, we would like y^t to give a probability distribution over its indices such that y_i^t yields the probability of obtaining predicting x^{t+1} if x^{t+1} is the one-hot vector with index i. Our loss (negative log-likelihood) at each time-step t is the cross entropy $\sum_i -x_i^{t+1} \log(y_i^t)$. Then our total loss is given by

$$L = \sum_t \sum_i -x_i^{t+1} \log(y_i^t).$$

In order to train the parameters θ of an RNN we can generalize ordinary backpropagation to the idea of **Truncated Backpropagation Through Time (Truncated BPTT)** [9]. Briefly, this approach is nearly the same as ordinary backpropagation as applied to an RNN unrolled over some number T time-steps, noting that the gradient update for each time-shared parameter must include a summand corresponding to each time-step in which it is applied. (The truncation saves memory by only considering a fixed bounded number of steps backwards in time.) Finally, note that given a fully trained network, we may iteratively generate a predicted sequence from a starting stub $x^1, \cdots x^r$ by running our RNN on time steps $1, 2, \cdots r$ and sampling a value x^{r+1} from the probability distribution induced by y^r, then considering this to be a part of the sequence and continuing.

Having elaborated on the structure and training of recurrent neural networks of varying depths as applied to sequence generation, we finally discuss the cell which will form the basis of our RNNs: the **Long-Short Term Memory Cell (LSTM)**. [9]. The problem with basic RNN cells such as in Figure 1 stems from the vanishing/exploding gradient problem: as the products given by backpropagation tend to go to 0 or ∞ as the number of terms increase, it is very difficult for information to be effectively propagated over long distances in the input sequence. Since our sequences are much longer than the depths of traditional feedforward neural networks, this is a significant issue, especially because a predictive model for vectors x_j towards the end of the sequence must retain some selected information regarding x_is near the beginning of the sequence. The solution is to allow the cells themselves to train parameters which effectively record and erase memory over time.

Pictorially, an LSTM cell is given by

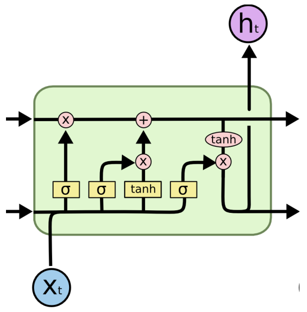


Figure 3: A single LSTM cell, represented pictorially. [7]

which contains a **forget gate, input gate, output gate,** and **cell**. This cell is recurrent in a similar way to our basic RNN cell, with update equations given by

$$\begin{aligned}
f^t &= \sigma(W_{fx}x^t + W_{fh}h^{t-1} + b_f) \\
i^t &= \sigma(W_{ix}x^t + W_{ih}h^{t-1} + b_i) \\
o^t &= \sigma(W_{ox}x^t + W_{oh}h^{t-1} + b_o) \\
c^t &= f^t \circ c^{t-1} + i^t \circ \tanh(W_{cx}x^t + W_{ch}h^{t-1} + b_c) \\
h^t &= o^t \circ \tanh(c^t)
\end{aligned}$$

where $\circ$ denotes the Hadamard product. This cell may be used in shallow or deep neural networks described earlier. In this paper, we will exclusively used LSTM-based recurrent neural networks.

3 Data and Musical Background

We have two main sources of data: online MIDI files of a wide variety of classical piano pieces written by Beethoven, and the Connolly collection of Irish violin pieces from Boston College.

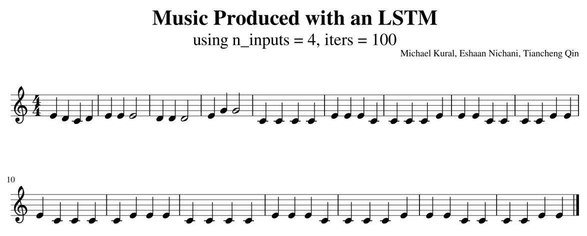
To process these MIDI files, we used music21, a toolkit for computer-aided musicology. Using music21, we could conveniently extract musical qualities from the MIDI files such as time signature, key signature, tempo, voices, and the notes themselves (pitch and duration).

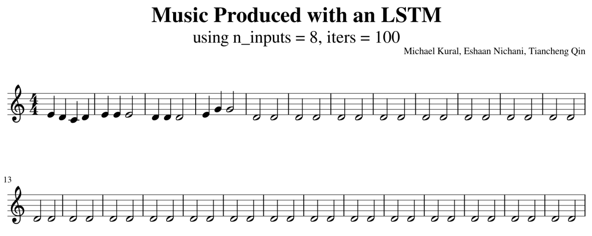
The Connolly music collection consists of 337 single melody violin pieces. These pieces are all around 20 measures in length and span various different time and key signatures. The set of Beethoven piano pieces consist of numerous melodies and harmonies, as they were written to be played by piano. We used music21 to extract the individual voices from each piece in order to train our model to produce a single melody output.

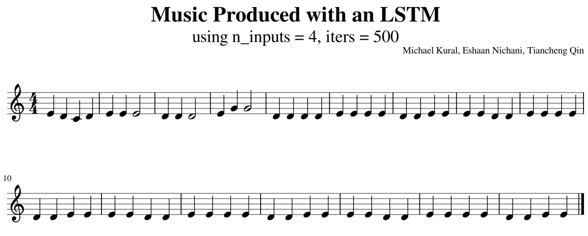
4 Our Approach

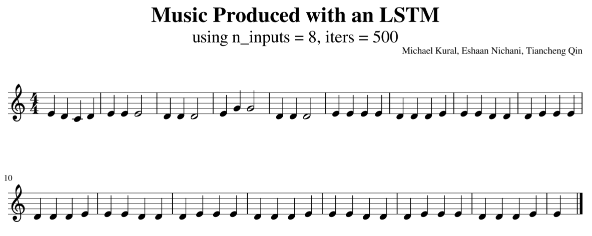
4.1 Initial Implementation

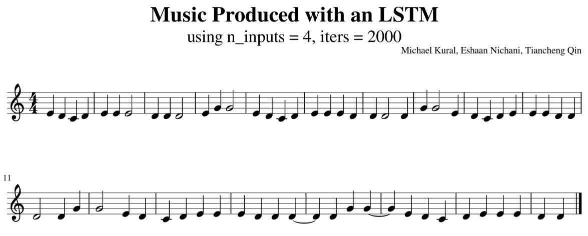
We used TensorFlow [10] and music21 to get an initial version of our final project running end-to-end and produce initial results. For our initial version, we decided to train our Long Short-Term Memory (LSTM) model on a very simple melody with only one voice: Mary Had a Little Lamb. Using the TensorFlow package that deals with Recurrent Neural Networks (RNNs), we varied hyperparameters such as number of previous inputs considered and number of training. We then fed the first four measures of Mary Had a Little Lamb into our LSTM model, producing the resulting music below for various parameters settings:


Music Produced with an LSTM
using n_inputs = 4, iters = 100
Michael Kural, Eshaan Nichani, Tiancheng Qin
10
(a) Previous inputs: 4, Iterations: 100


Music Produced with an LSTM
using n_inputs = 8, iters = 100
Michael Kural, Eshaan Nichani, Tiancheng Qin
13
(b) Previous inputs: 8, Iterations: 100


Music Produced with an LSTM
using n_inputs = 4, iters = 500
Michael Kural, Eshaan Nichani, Tiancheng Qin
10
(c) Previous inputs: 4, Iterations: 500


Music Produced with an LSTM
using n_inputs = 8, iters = 500
Michael Kural, Eshaan Nichani, Tiancheng Qin
10
(d) Previous inputs: 8, Iterations: 500


Music Produced with an LSTM
using n_inputs = 4, iters = 2000
Michael Kural, Eshaan Nichani, Tiancheng Qin
11
(e) Previous inputs: 8, Iterations: 2000

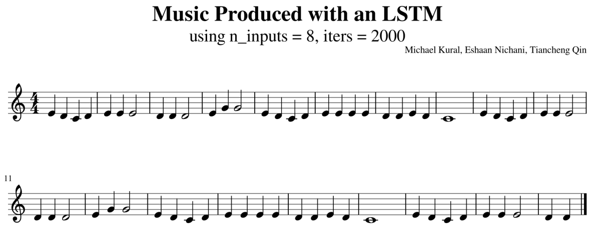

Music Produced with an LSTM
using n_inputs = 8, iters = 2000
Michael Kural, Eshaan Nichani, Tiancheng Qin
11
(f) Previous inputs: 8, Iterations: 2000

Figure 4: Initial LSTM trained on 'Mary Had a Little Lamb'

As you can see, the higher the number of previous inputs considered and the higher the number of training iterations, the closer the artificially produced music was to the training data. One problem with this is that training 8 previous inputs for 2000 iterations very clearly overfit to our data. Moving forward, we considered regularization and data augmentation to make our results more generalizable, as well as extended the the size of our training set in order to produce more unique and interesting melodies.

4.2 Data Preprocessing

The musical properties that we used to train our LSTM model were pitch (the note and octave played, such as F#3), note duration (quarter note, eight note, triplet, etc.), and, when activated, the beat with respect to the measure (represented by a float, and implicitly giving whether a note lies on a downbeat, upbeat, or in between). We expressed each pair of (pitch, duration) (or triple of (pitch, duration, beat)) as a one-hot vector, which allowed us to represent each voice uniquely and concretely.

We also included an option to transpose all of our music pieces to the key of C major. The reasoning behind this is that it may be challenging for the LSTM to learn the key of a piece, and remain in the same key when generating a new piece. By transposing our test set to the same key, we ideally hope that our model can focus on learning melodies and not get confused by variations in the key.

4.3 Model Architecture

We then developed a more extensive model based on the dataset of Irish violin pieces. For this model we relied on the `LSTMCell` architecture [2] in Tensorflow, which allows us to make more specific changes to the parameters of the model. Additionally, we used Tensorflow's `dynamicRNN` wrapper, which allows our model to both train with and generate samples of sequences of variable length.

The general structure of our model is as follows. During an epoch of our training phase, we randomly select a piece from our set of training pieces, and then randomly select a sequence of `sample_length` notes. We then convert each note to a one hot vector with dimension m. If we are using the beat as one of the properties we train on, then $m = 328$, otherwise $m = 1703$. At time t, we feed the tth note into our LSTM, using the hidden state from the $t-1$th note. The output of our multilayered LSTM is a vector of dimension `lstm_size`. We then have a final fully connected layer, followed by a softmax activation function, to obtain a final vector of probabilities with dimension m. This vector represents the probability distribution over the next note in the sequence. We optimize the cross validation loss between this probability distribution at each time step and the one hot vector corresponding to the actual note being played.

The k-fold cross validation loss, as well as music samples from selected architecture, are reported in the results section for various model architecture combinations.

Below are various aspects of the model architecture which we modified.

Network Depth: We experimented with the effectiveness of a deep LSTM network using Tensorflow's `MultiRNNCell` wrapper. In a deep LSTM network with n layers, the output of the kth layer (which is an LSTM cell) is the input to the $(k+1)$th layer (also an LSTM cell). Furthermore, each layer has its own hidden state which it passes on to itself in the next time step - this increases the dimension of the hidden state by a factor of n.

Increasing the network depth gives our model the ability to learn more complicated structures and thus generate better music. However, the increase in the number of parameters can drastically increase the training time and lead to severe overfitting. Thus we rely on cross validation to select the optimal network depth.

Regularization and Dropout: Given a larger network, it is thus imperative we apply some form of regularization in order to prevent overfitting. One of the most popular forms of regularization within neural networks is dropout. During training time, we randomly set some fraction of the activations to 0, thus preventing overly complex relationships between the units and producing a form of boosting, or model averaging. We experimented with whether using dropout, with varying removal probabilities, decreased the cross validation loss.

Peephole: In addition, we modified the internal structure of an LSTM cell. Gers and Schmidhuber [3] introduced one such LSTM variant known as a "peephole LSTM". A peephole LSTM lets each of the gates use as an input the cell state in addition to the hidden state. This allows for greater variation in how information is transferred between cells in succesive timesteps. We experimented with whether the addition of peephole connections improved our model's performance.

Optimizer: We experimented with three optimizers for our model: the standard stochastic gradient descent optimizer, the `RMSProp` optimizer, and the Adaptive Moment Estimation (Adam) optimizer [8]. The standard SGD optimizer has a fixed learning rate as a hyperparameter, whereas the `RMSProp` optimizer divides the learning rate by an exponentially decaying average of squared gradients, which yields a decaying step size over time:

$$\mathbb{E}[g^2]_t = 0.9\mathbb{E}[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t$$

On top of this, the Adam optimizer keeps an exponentially decaying average of past gradients and corrects their biases so they don't decay to zero. In particular, if m_t and v_t are the first and second moments of past gradients, Adam adjusts them so that they are not pushed to zero:

$$\hat{m_t} = \frac{m_t}{1-\beta_1} \qquad \hat{v_t} = \frac{v_t}{1-\beta_2}$$

where β_1 and β_2 are hyper-parameterized constants.

When testing our model architecture, we discovered that the `RMSProp` optimizer did a much better job than the standard SGD optimizer of minimizing cross-validation loss, likely due to its ability to more accurately converge on the best possible parameters for our model. Further, the Adam optimizer out-performed `RMSProp` due to its bias-correction, allowing it to optimize gradients near the end as they become sparser.

4.4 Model Hyperparameters

In addition to modifying the general structure of the model, we tweaked various hyperparameters and analyzed their effect on model performance.

Learning Rate: We modified the hyperparameter η, which governs the the magnitude of the change in our paarameters each timestep. A large value of η can lead to quick convegence, but may overshoot our optimum. A small value of η is more likely to converge, but could converge far slower and possibly quite far from the desired solution.

Sample Length During model training, we would take in sequences of length `sample_length` and calculate the average training loss. Thus our model would look back at most sample_length notes in the past in order to predict the next note. In general, having a larger sample length allowed our model to better learn musical properties such as beat structure, key signature, and phrasing, and have our model produce music that reflected these qualities. The two downsides to increasing sample length are that it decreases the amount of training samples that we get to work with, and that it could cause our model to overfit to our training set.

LSTM Size: Another hyperparameter that we varied was the number of nodes for every hidden layer in our model. In general, having more nodes for every hidden layer allowed our model to achieve better training accuracy because our model would be able to capture more relevant features of the music. However, two downsides to increasing LSTM size are that it significantly increases training time and it could cause our model to overfit.

5 Results

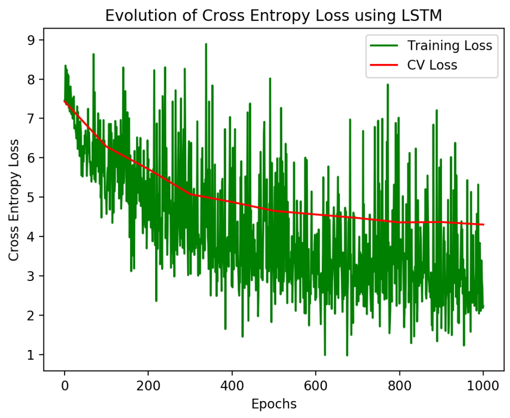
We trained our model on both the set of Beethoven pieces and the Connolly dataset. The model performed poorly on the Beethoven pieces, and generated results that did not seem very musical. This is likely due to the pre-processing step of separating a Beethoven piece into voices - most of the voices are harmonies and not melodies, and thus on their own don't sound very good. Our LSTM then combines both melodies and harmonies to produce a music sample which is neither. Thus we focus our results on training on the Connolly dataset, which contains pieces with a single voice, the melody line played by the violin, and produces much better results.

5.1 Cross Validation Loss:

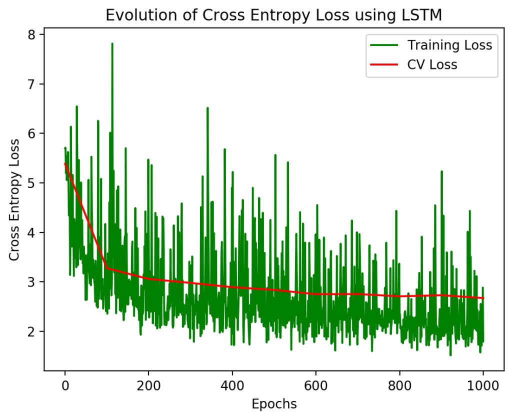
To compute the relative accuracies of each of our models, we computed the average k-fold cross validation loss. Specifically, we split our Connolly dataset of 337 pieces into k folds, and trained a model based on $k-1$ of these folds. Then, for each piece in the last fold, we ran our trained model to obtain a vector of probabilities $\hat{y}^t$ for each timestep t. Given the actual one hot vector encoding at time t, y^t, we can compute the cross entropy loss for the mth fold as $L_m = -\frac{1}{t}\sum_t \sum_i y_i^t \log(\hat{y}_i^t)$. Then, the average k-fold cross validation loss is given by $\frac{1}{k}\sum_{m=1}^{k} L_m$. Using $k = 4$, we got the following results for our various models:

Peephole	Dropout	Layers	Step Size	Length	LSTM Size	Beats	Transpose	CV Loss
FALSE	FALSE	1	0.01	48	128	FALSE	FALSE	2.84858
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
FALSE	FALSE	3	0.01	48	128	FALSE	FALSE	2.95909
FALSE	FALSE	2	0.01	48	32	FALSE	FALSE	2.89754
FALSE	FALSE	2	0.01	48	64	FALSE	FALSE	2.84535
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
FALSE	FALSE	2	0.01	48	256	FALSE	FALSE	3.1896
FALSE	FALSE	2	0.1	48	128	FALSE	FALSE	4.74789
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
FALSE	FALSE	2	0.001	48	128	FALSE	FALSE	2.80328
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
TRUE	FALSE	2	0.01	48	128	FALSE	FALSE	2.83888
FALSE	0.8	2	0.01	48	128	FALSE	FALSE	3.24388
FALSE	0.6	2	0.01	48	128	FALSE	FALSE	3.48164
FALSE	FALSE	2	0.01	32	128	FALSE	FALSE	2.89289
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
FALSE	FALSE	2	0.01	64	128	FALSE	FALSE	2.83842
FALSE	FALSE	2	0.01	96	128	FALSE	FALSE	2.83021
FALSE	FALSE	2	0.01	48	128	FALSE	FALSE	2.86406
FALSE	FALSE	2	0.01	48	128	FALSE	TRUE	2.84375
FALSE	FALSE	2	0.01	48	128	TRUE	FALSE	4.33437
FALSE	FALSE	2	0.01	48	128	TRUE	TRUE	3.98484
FALSE	0.8	3	0.1	48	256	TRUE	FALSE	4.17703
TRUE	FALSE	1	0.01	96	64	FALSE	TRUE	2.72566

Table 1: Cross-validation (CV) losses. The second-bottom row represents the parameters of a sub-optimal model, and the bottom row represents the parameters of our best model.



(a) Model A: A suboptimal model (the second-bottom row above).



(b) Model B: Our optimal model (the bottom row above).

Figure 5: Evolution of cross-validation losses

Looking at the table, we notice a couple of trends:

Number of layers: Having more layers gives higher cross-validation loss, possibly due to there being a greater number of parameters to train in addition to the model over-fitting to the training data.

LSTM Size: Having too many nodes per layer caused our model to overfit to the training data, whereas having too few made our model incapable of capturing enough relevant information. Therefore having a medium number (64) of nodes per layer gave us the best results.

Step Size: Our cross-validation loss decreased as we decreased our step-size because this allowed our model to get better fine-tuned to our data.

Peephole: Making our model actually a peephole LSTM decreased the crossvalidation loss. One possible reason is that having access to both the cell state and the hidden state allows for our model to more easily find the desired relations

Dropout: Enabling dropout regularization increased our cross-validation loss because we only used 2 layers in our recurrent neural net, making our model incapable of correcting errors associated with dropout. Furthermore, our dataset of music samples may be too small for dropout to work effectively, and that our central issue has to do with underfitting, instead of overfitting.

Sample Length: Cross-validation loss decreased with increasing sample length because having more history input into our LSTM allowed our model to understand more of the music as a whole, resulting in better predictions. However, we cannot increase the sample length arbitrarily, as this will cause our training set to shrink too much. Thus we seek to choose a sample length which is large enough but still gives us a sufficiently large training set.

Beat detection: Enabling beat detection expanded our input vector from pairs (pitch, duration) to triples (pitch, duration, beat), allowing our LSTM would be able to learn and generate music consistent with the rhythm of its measures. The beat was given by the `note.beat` property in `music21`, which is a decimal representing the number of beats into a measure a note is played. Note that adding beat detection increases the cross-validation loss significantly, but it is important to note that changing the data type increases the dimension of the one-hot vector, so this cross-validation cross-entropy loss cannot be directly compared to other configurations. However, a subjective analysis led us to conclude that beat detection was unsuccessful in increasing our compositions' rhythmic or melodic viability.

Transposition: Transposition to C major caused our cross-validation loss to decrease because it provided a way to standardize the data, take key signature into account, and stabilize our results.

We also plotted the evolution of the k-fold cross validation loss, as well as the training loss, as a function of epoch. The graphs for a sub-optimal model (model A) and one of our better models (model B) are shown in **Figure 5** above. Notice how Model A converges at a far higher training loss and validation loss than Model B. The green spikes in our graph are due to the stochastic nature of our optimization process, as we are essentially considering mini-batches of size 1.

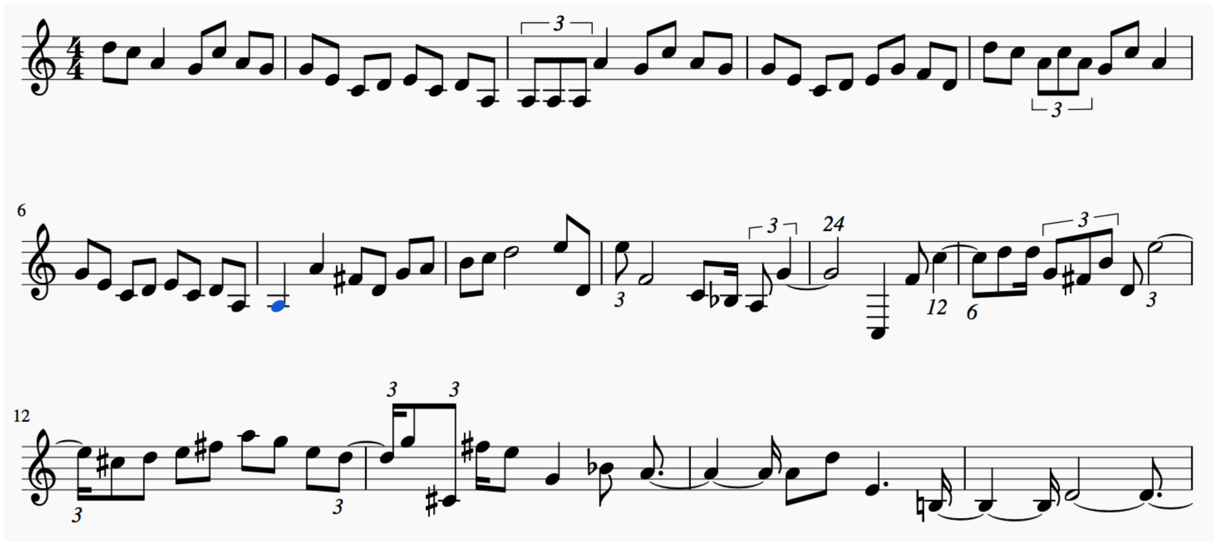


Figure 6: A sample of music generated by a poor model (peephole = FALSE, dropout = 0.8, num_layers = 3, beat = TRUE, transpose_toC = FALSE, learning_rate = 0.1, sample_length = 48, lstm_size = 256)

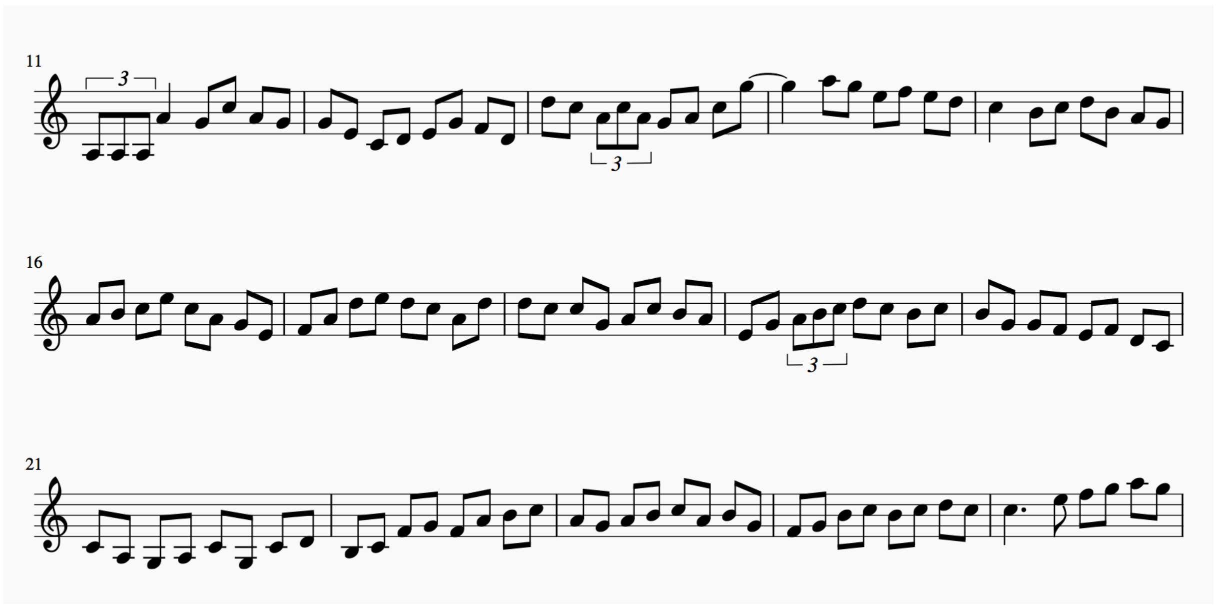


Figure 7: A sample of music generated by our one of our better models (peephole = TRUE, dropout = FALSE, num_layers = 1, beat = FALSE, transpose_toC = TRUE, learning_rate = 0.01, sample_length = 96, lstm_size = 64)

5.2 Qualitative Analysis of Produced Music:

We can also use our LSTM to construct a generative model in order to produce our own music. To do this, we start out with a sample of music with T notes from our training set, and pass these in as our x values over various timesteps to obtain a probability distribution for the $(T+1)$th note. We sample a note from this distribution to append to our initial music sample. Then, we consider the one-hot vector corresponding to that note, as well as the hidden state from the previous timestep, to obtain the probability distribution for the $(T+2)$nd note. By repeating such a process, we can obtain a short sample of music generated by our LSTM. **Figure 6** and **Figure 7** above are samples of music generated by LSTM models using recurrent neural nets.

The first 6 measures of **Figure 6** were used to initially generate this piece, and the rest was produced by the model from the corresponding softmax probability distribution. As you can see, our model does a poor job of detecting the key signature and phrase structure. The piece starts in C Major, but slowly morphs into G Major, followed by D Major, with accidentals scattered throughout. Further, the piece begins with an 8th note structure, which slowly disappears later on, with no conceivable phrasing. This is potentially due to not using peephole regularization, a large step size (0.1), small sample length (48), and over-fitting due to a large number of nodes (256) per hidden layer. To improve on this model, we tried adding peephole connections, decreasing step size, increasing sample length, and decreasing the number of nodes per hidden layer. **Figure 7** displays a segment of music generated by such a model.

The first 3 measures of **Figure 7** are part of the initial sample we use to generate the piece, and the rest of the notes were produced by sampling from the corresponding probability distribution. We observe that the the model does a pretty good job of generating a sequence of notes. The entire piece is in C major, which is due to the fact that we transposed all of our training data into C major before running. Additionally, there are no large jumps - the largest jump between consecutive notes is a 5th - also following a common rule in music theory. There appear to be ascending and descending lines, mimicing scales, as well. However, the piece still lacks overall structure when analyzing it from a broader perspective than just a measure, as measures in the piece seem to be constructed entirely independently than others.

6 Further Directions

As far as our data set goes, most of our work dealt with the Irish violin pieces, which only had one voice. Moving forward, we could look to train our LSTM model on polyphonic pieces, such as those written by J.S. Bach. We would then have to modify our model to train and generate music laterally as opposed to one voice at a time, since this way we would be able understand the music more as a whole.

Furthermore, one critical component of musical works that our model fails to captures is repetition, specifically between different sections of the piece. It would be interesting to analyze the output of a model that tried to use learn some implicit repetition structure, and then generate music according to a generated repetition. More specifically, we could include two separate LSTMs - one to learn a rhythm, and one which takes as input this rhythm and outputs an attached melody. This may help to alleviate some of the issues the model has in learning rhythm, such as playing only one triplet note instead of a sequence of 3.

Finally, we could look into modifying the cost function we optimize. Our cost function treats each (pitch, duration, beat) tuple as a distinct coordinate in the one hot vector, and we could incorporate into our cross entropy loss a better distance metric than the discrete one to less harshly penalize outputs that get one of the three quantities correct.

7 Division of Labor

For the most part, all three group members played a hand in all steps of the process, including planning, finding data, contributing code to our collective repository, analyzing data, and writing the final paper. Our group members did focus a bit more on different areas, but our contributions in the end turned out to be equal.

Tiancheng spearheaded much of the first implementations of our data parser to incorporate classical MIDI files into our dataset. He also contributed to our first static RNN implementation in TensorFlow and refactored much of our conflicting code in a third iteration. Eshaan also contributed to the first static RNN implementation and rewrote another implementation using a dynamic RNN as well as mini-batches to speed up the program. He also implemented the calculation of k-fold cross validation loss, and experimented with changes in network architecture such as number of layers, in the final iteration. Michael rewrote a second implementation of the dynamic RNN implementation and also implemented the incorporations of several of the model changes, including the peephole option, the dropout option, including beats, and transposing to C (and implemented tensorboard summary files for debugging). Michael also researched the computational music software music21 and gained access to an Irish violin repository for melodic analysis, both thanks to his 21M.302 professor, and focused on writing up the paper.

8 Acknowledgments

We would like to thank Professor Michael Scott Cuthbert (Michael Kural's professor for 21M.302, Harmony and Counterpoint II) for his direct and indirect help, including: the creation of the invaluable `music21` package, the supplying of the Irish solo violin repository, and advice regarding transposition for training. We would also like to thank the professors and TAs for making 6.867 such a valuable and intellectually stimulating course this entire semester.

9 Miscellaneous

This project is not being used as a final project for any of our other classes.

10 References

[1] Abolafia, Dan. “A Recurrent Neural Network Music Generation Tutorial.” 10 June 2016. Web.

https://magenta.tensorflow.org/2016/06/10/recurrent-neural-network-generation-tutorial

[2] Atienza, Rowel. ”LSTM By Example Using TensorFlow”. 17 March 2017. https://medium.com/towards-data-science/lstm-by-example-using-tensorflow-feb0c1968537

[3] Gers, Felix & Schmidhuber, Jurgen. (2000). Recurrent nets that time and count. Proceedings of the International Joint Conference on Neural Networks. 3. 189 - 194 vol.3. 10.1109/IJCNN.2000.861302. ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf

[4] Graves, Alex. ”Generating Sequences with Recurrent Neural Networks.” 5 June 2014. Web. https://arxiv.org/pdf/1308.0850.pdf

[5] Johnson, Daniel. ”Composing Music with Recurrent Neural Networks”. 2 August 2015. Web. http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/

[6] Karpathy, Andrej. ”The Unreasonable Effectiveness of Recurrent Neural Networks.” Andrej Karpathy Blog. 21 May 2015. Web. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[7] Olah, Chris. ”Understanding LSTM Networks.” Colah’s blog. 27 August 2015. Web. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[8] Ruder, Sebastian. An Overview of Gradient Descent Optimization Algorithms. 15 June 2017. Web. ruder.io/optimizing-gradient-descent/index.html#rmsprop.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation, 9:17351780, 1997. http://www.bioinf.jku.at/publications/older/2604.pdf

[10] TensorFlow. ”TensorFlow API”. 2 November 2017. Web. https://www.tensorflow.org/api_docs/python/