Exploration of LSTM on Song Generation

Hargen Zheng
Halıcıoğlu Data Science Institute
University of California, San Diego
San Diego, CA 92092
yoz018@ucsd.edu

Nathaniel del Rosario
Halıcıoğlu Data Science Institute
University of California, San Diego
San Diego, CA 92092
nadelrosario@ucsd.edu

Chuong Nguyen
Computer Science and Engineering Dept.
University of California, San Diego
San Diego, CA 92092
chn021o@ucsd.edu

Ziyue Liu
Electrical and Computer Engineering Dept.
University of California, San Diego
San Diego, CA 92092
zil085@ucsd.edu

Adam Tran
Mathematics Department
University of California, San Diego
San Diego, CA 92092
ant010@ucsd.edu

Abstract

In our work we use a Long Short Term Memory network to predict sequences of characters, one at a time, for song generation. We work with data in the form of songs in ABC notation to output generated songs in the same format. Our baseline model consisted of a LSTM model with a single hidden layer, utilizing dropout, an embedding layer, and a final output linear layer. To train our model, we took random slices of notes from our song data and measured our loss with Cross Entropy Loss. Our baseline model had a training loss of 1.507 and a validation loss of 1.931. To generate songs with our character predicting model we draw from a distribution of next characters at each step. We introduce Temperature to our process to add randomness and make the song generating process less deterministic. Using a temperature value of 2 generated less robotic sounding songs with less repetitive notes compared to using temperature values of $.5$ and 1. In tuning our model, we compare our model to a regular Recurrent Neural Network, adjust the number of Neurons within our single hidden layer, and adjust the probability of dropout. As expected our baseline model out performs the RNN model with a greater training loss of 1.598 and greater validation loss of 2.062. While increasing our dropout from the initial $.1$ probability did not improve our model, we found increasing the number of hidden neurons to show improvements. By increasing the number of Neurons from 150 to 250 the training loss becomes 1.408 and the validation loss becomes 1.861.

1 Introduction

The generation of music using artificial intelligence has become a fascinating domain, merging the fields of machine learning and art. Among various approaches, the use of Long Short-Term Memory (LSTM) networks has shown significant promise due to their ability to remember long-term dependencies, a critical aspect in understanding musical composition. The task at hand involves creating an LSTM model capable of generating music by learning from a corpus of existing compositions. This is important not only as a technical challenge but also for its potential to innovate in music composition, offering tools for artists to explore new creative landscapes. Understanding the sequential nature of music, where each note or chord progression depends on the preceding elements, is crucial for appreciating the complexity of this task. In the scope of generative artificial intelligence, music generation, and more broadly any generative art is important because of the common criticism of AI not being sentient, or not being able to mimic human emotion and creativity without being super strict and one dimensional. For example, in writing music, there is more to the song than notes on a page, but additionally the phrases and chord progressions, as well as the emotions portrayed by different articulations and instrumentation / orchestration of instruments in larger ensembles, compared to a single instrument. Thus, this sparks the question of how diverse (and therefore more robust) these models can become, to more closely resemble human artists' techniques within their creative works.

2 Related Work

Several studies have laid the groundwork for using LSTM networks in music generation. The pioneering work by Eck and Schmidhuber (2002) [3] on a recurrent neural network for learning and generating melodies was among the first to demonstrate the potential of LSTMs in this field. Further, Hadjeres, Pachet, and Nielsen (2016) [5] introduced DeepBach, a model that generates music in the style of Bach, showcasing the LSTM's ability to capture the style of specific composers. These works, among others, provide a solid foundation for exploring character-level LSTM models in music generation. Building upon these, our approach focuses on the granularity of character-level data representation, aiming to capture finer musical nuances than previously addressed. This project also draws inspiration from advancements in text generation using LSTMs, applying similar principles to the domain of music.

Following the early works, more recent papers have introduced innovative approaches and techniques to enhance the quality and diversity of generated music. One notable advancement is the work by Huang et al. (2018) [6], who introduced the Music Transformer, a model that leverages self-attention mechanisms (very relevant to last week's lecture) to generate music with improved long-term coherence. This model addresses one of the key challenges in music generation, maintaining structure and thematic consistency over longer sequences, demonstrating the Transformer architecture's potential in capturing complex musical relationships.

Additionally, Roberts et al. (2019) [8] presented the Coconet model, which generates polyphonic music by iteratively refining incomplete pieces. This model, inspired by the structure and counterpoint of Bach's compositions, showcases the ability of neural networks to handle the intricate interdependencies of notes in polyphonic music. Furthermore, Donahue et al. (2019) [2] introduced the LakhNES model, which generates music in the style of 8-bit video games, using LSTM networks trained on a large dataset of video game music. This work highlights the versatility of LSTM models in adapting to various musical genres and styles, from classical to contemporary electronic music.

Lastly, it would be unfair to leave out the recent the development of GANs (Generative Adversarial Networks) which also influenced music generation. These models have the advantage of capturing more diversity from the distributions compared to previous generative models. For example, Engel et al. (2020) [4] explored GAN-based models for generating realistic and stylistically varied music sequences, demonstrating an alternative approach to LSTMs that can capture the nuances of musical composition and style diversity.

3 Methods

3.1 Training using Teacher Forcing

For the baseline of the model, we chose to use the recurrent network with LSTM architecture. LSTM composes the input gate, output gate, cell state, and hidden state to enable the flexibility for the model to keep, forget, and read out the past information from the past input from cell state. The details of standard LSTM cell works as follows [7]:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{1}$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{2}$$
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{3}$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{5}$$
$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

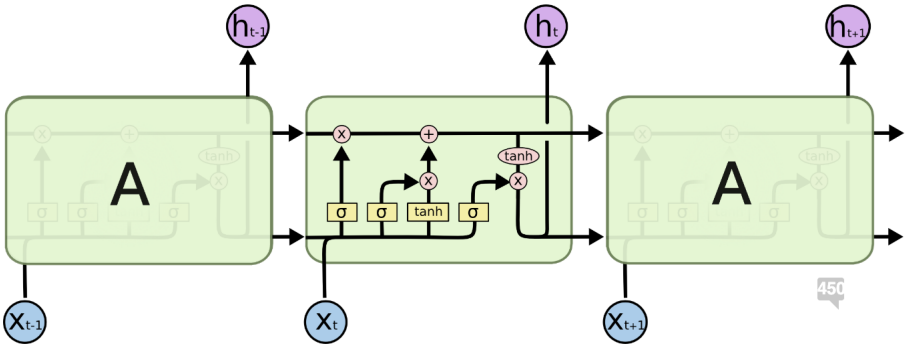


Figure 1: LSTM Architecture [1]

The LSTM cell has a key feature – a horizontal line that runs through the top of the diagram. This line represents the cell state, which runs straight down across time, with limited linear interactions. This allows information to easily move across time unchanged, similar to ResNet.

More importantly, the LSTM cell controls what information is stored in the cell state based on the previous cell state, the previous hidden state, and the current input. To do this, the LSTM cell performs three steps.

Firstly, the LSTM cell decides which information needs to be removed from the cell state. This is done using a sigmoid function in equation (2). The sigmoid function assigns a constant value between zero and one for each value in the cell. This determines the extent to which past information will be thrown out.

Secondly, the LSTM cell decides which information needs to be added to the cell state. The cell applies a tanh activation function to the sum of the input and hidden state as shown in equation (3). The result is then scaled by i_t in equation (1), which determines which new information needs to be added to the cell state.

Finally, LSTM need to run a sigmoid layer, which decides what parts of the cell state we're going to output (equation 4). Then, LSTM put the cell state through tanh and multiply it by the output of the sigmoid gate so that LSTM only outputs the parts it decided to, which will be hidden state for at the next time state.

To implement our model, we used the Pytorch library. Our baseline LSTM model included a single hidden layer with 150 neurons, along with an initial embedding layer and a final linear layer as our output. We included dropout within our recurrent layer. To train our model, we used Teacher Forcing, which takes the ground truth character as input to predict the next character in the sequence.

Following that, the next ground truth character is used to predict the following character. This differs from using the predicted output as the new input of the model. We used Stochastic Gradient Descent to optimize the weights of our model and measured the training and validation losses with Cross Entropy Loss, from the Pytorch Library.

The set of hyperparameters we use for the baseline model is given as follows:

Table 1: Hyperparameter Choices

Hyperparameter Name	Value
learning rate	1e-3
sequence size	30
number of layers	1
hidden size	150
dropout probability	0.1
model type	LSTM

Note that for the rest of the experiments in our report, we keep the hyperparameter values for {learning rate, sequence size, number of layers} while trying to tune other hyperparameter values for experiment purposes.

3.2 Song Generation

For song generation, our model runs the given prime string sequence. After that, the rest of the sequence would have maximum sequence length of $max\_length - len(prime\_str)$. We take in the last character in the prime string as our initial model input and generate the characters by running forward pass for the model until either of the following conditions is satisfied:

1. The last 5 characters in the generated sequence is "⟨end⟩" sequence, which means we have early stop in the generation process.
2. We have reached the maximum length of the song generation sequence.

For the details of song generation process, more specifically, we pass the output of the model's forward pass to a softmax layer, where we would obtain a set of probabilities, or likelihoods, of the next character, given our input. Then, we draw 1 character index from multinomial distribution with the probabilities given in the softmax. The character that corresponds to the randomly sampled index would be our next predicting character for the song generation process.

By dividing each individual element in the result of our forward pass, namely logits, by the temperature T, where our softmax probability distribution could be represented as follows:

$$Pr(z_i) = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_{j=1}^{N} \exp\left(\frac{z_j}{T}\right)},$$

where N is the dimension of the output given by model's forward pass.

By adding temperature to our song generation process, we are driving the model to draw the next character from multinomial distribution towards uniform distribution, as the differences among probabilities would even out as the value of temperature T grows. By increasing the value of temperature T, we are introducing more randomness in our song generation process and thus the resulting sequence would be less deterministic given our trained model.

3.3 Hyperparameter Tuning

Firstly, we used the same set of hyperparameters in Table 1 but changed the recurrent unit to normal RNN unit, instead of LSTM RNN unit. At each time step, the RNN unit takes in an input and the current hidden state and then produces an output and a new hidden state, which are used as inputs for the next time step. This process continues until the entire sequence has been processed. More specifically, we apply a multi-layer with $\tanh$ as the nonlinearity transformation computed as follows:

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh}), \tag{7}$$

where h_t is the hidden state at time t, x_t is the input at time t, and h_{t-1} is the hidden state of the previous layer at time $t-1$ and W_{ih}, W_{hh} are weights for each layer [7].

Besides comparing the loss values between the RNN and LSTM units, we also experimented with different choices of hyperparameters with LSTM unit to see how much more the LSTM unit could do to generate music sequences. For tuning the number of hidden neurons we have, our baseline model already used 150 hidden neurons. Therefore, we experimented with 2 more values – 200 and 250. For the dropout probability, our baseline model result was obtained with dropout probability of 0.1. Therefore, we tried to experiment with 2 more dropout probabilities – 0.2 and 0.3.

For all these fine-tuning experiments, we used the same SGD optimizer and Cross Entropy loss function to make sure everything besides the value of hyperparameter itself stay unchanged, so we can make meaningful observations and insights from the results we obtain from these fine-tuning experiments.

In summary, our experiments include tuning our model for $dropout \in \{0.1, 0.2, 0.3\}$ and tuning our model for $hidden\_size \in \{150, 200, 250\}$, where only change one hyperparameter at a time. The result will be presented in the results section later.

3.4 Feature Evaluation

In the context of RNNs, the input data often consists of sequential information, such as time series data, natural language text, or any other sequential data. When it comes to RNN architectures, it would be important for us to perform feature evaluation. One point would be making the model more interpretable. In the case of the baseline model, where we have 150 hidden neurons, performing feature evaluation on individual neurons helps us identify the hidden representation that the given neuron has learned through the training process. With the neuron's activation values – reflected by the heatmap – we can interpret the model better. Moreover, some of the generated songs look pretty random. By performing feature evaluation, we can understand the underlying pattern within the generated songs and evaluate on how our model's performance. This not only helps us to understand why some seemingly random outputs make sense but also enable us to interpret our model to a greater extend. With the above reasons, the feature evaluation is a crucial part.

Now, we explain the way we did feature evaluation and plot the corresponding heatmap. Firstly, we initialize an empty Tensor to store the activations of our hidden states. After running a forward pass for our model given the previous character in the sequence, we would be able to obtain a softmax distribution of the next likely character, where we then draw the predicted character from multinomial distribution – as discussed in the previous subsection. In the SongRNN class's forward pass function, we return the hidden activations along with the decoder output. Therefore, once we call the forward pass for the model, we concatenate the resulting activation to the Tensor we initialized. After the song generation process is finished, we convert the Tensor to operate on CPU, so further NumPy operations for heatmap plotting could work properly. As we call $generate\_heatmap$ function with different values of neuron index, we were able to leverage the Matplotlib package to plot the heatmap as desired.

4 Results

4.1 Baseline LSTM Model

The following graph contains the train/validation loss plot for the baseline model, which we plot the train/validation loss against the number of epochs.

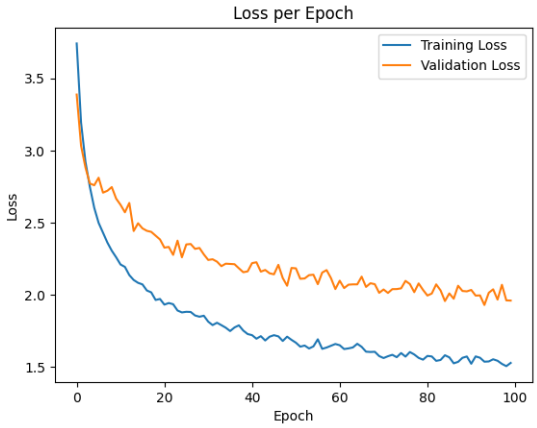


Figure 2: Baseline Plot

4.2 Song Generation with Various Temperatures

We tried using different temperature values: 0.5, 1, and 2 to experiment with our music generator.

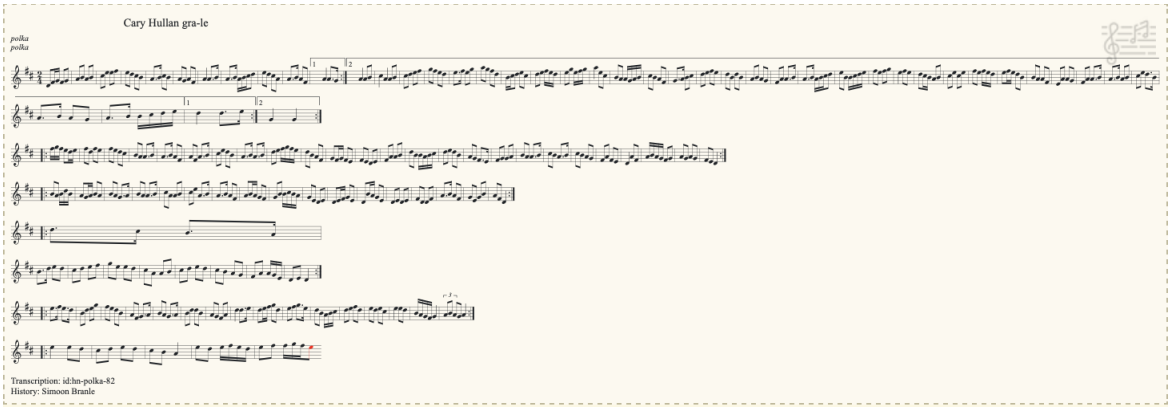


Figure 3: Song Generated Using Temp 0.5

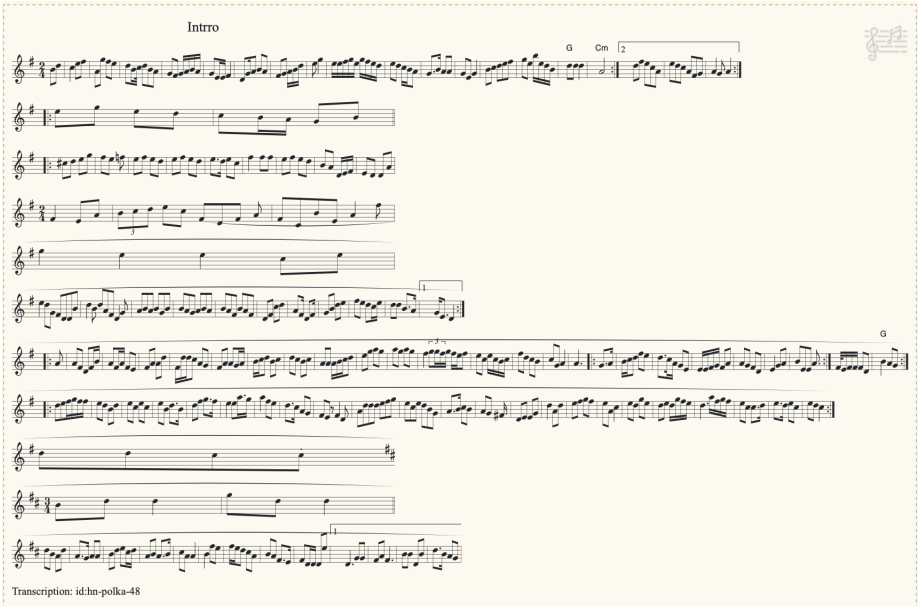


Figure 4: Song Generated Using Temp 1

We noticed that there is a lot of repetition notes in the generated song using temperature 0.5. When playing the song in the abc notation converter, often times, the same keys would be played multiple times in row, making the song sound robotic like. When using a temperature value of 1, there was a lot more diversity in the notes. However, there was still some repetitive keys within the song. Overall, the song improved in sounding less robotic. Lastly, we generated songs using a temperature of 2. The song was a lot more diverse in notes and sounded the most natural out of all of the previous songs we've generated using temperature 0.5 and 1. We also tested a temperature value of 2 with an increased max length of 5000. The result was the song sounding less robotic, and certain parts in the song where multiple keys were played at the same time compared to just a single key being played at a time. We tried using temperature 3, but the result was as good or worse compared to temperature 2. Thus, we were pretty convinced temperature 2 is a good threshold to test the model generator.

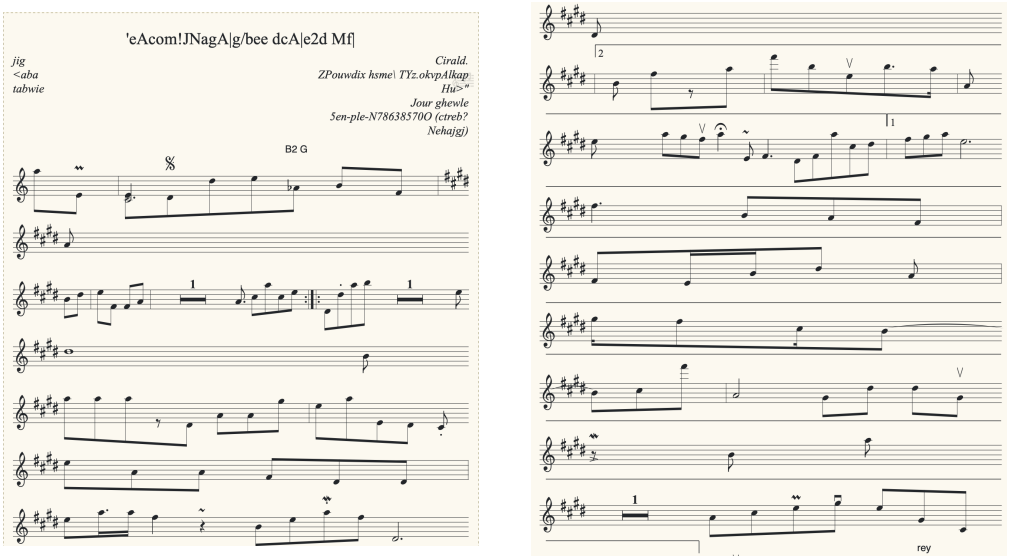


Figure 5: fig:Song Generated Using Temp 2

4.3 Fine-Tuning the Baseline Model

In this subsection, we present the training/validation plots for all fine-tuning experiments we have done throughout the process. The figures contain captions to state the hyperparameter value we changed while keeping other hyperparameter values the same as the baseline model.

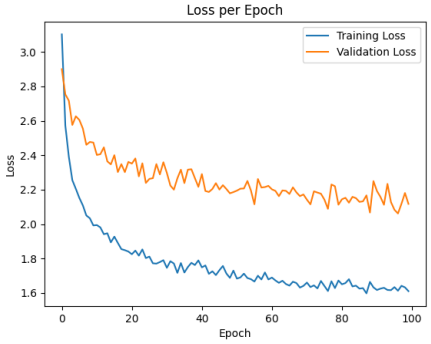
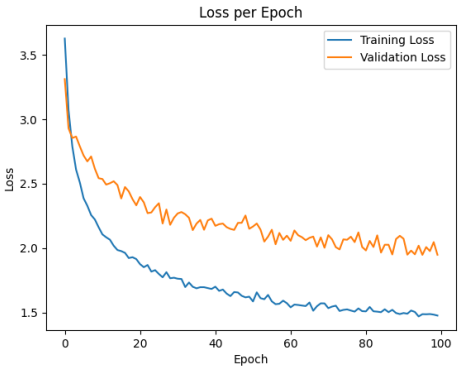
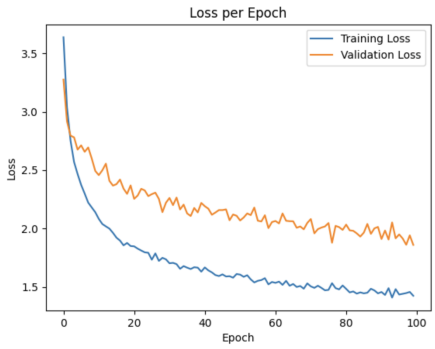


Figure 6: Baseline with RNN Unit Plot



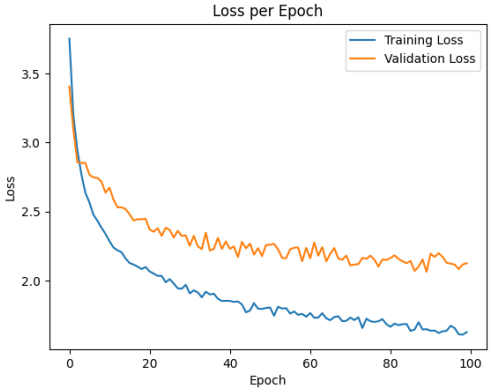
(a) Baseline with 200 Hidden Neurons Plot



(b) Baseline with 250 Hidden Neurons Plot

Figure 7: Fine-Tuning with Size of Hidden Neuron

(a) Baseline with Dropout Probability of 0.2 Plot



(b) Baseline with Dropout Probability of 0.3 Plot

Figure 8: Fine-Tuning with Dropout Probability

For the ease of comparison, we summarize all the best train/validation loss values for all our experiments in the following table, where we obtain the lowest possible loss within the 100-epoch window.

Table 2: Models and Their Performances

Model Name	Best Train Loss	Best Validation Loss
Baseline	1.5068130493164062	1.9306975603103638
Baseline with normal RNN Unit	1.5976272821426392	2.061519145965576
Baseline with Hidden Size 200	1.46933114528656	1.9474481344223022
Baseline with Hidden Size 250	1.407871961593628	1.8607800006866455
Baseline with Dropout 0.2	1.5800480842590332	2.010080575942993
Baseline with Dropout 0.3	1.6120514869689941	2.064985990524292

5 Discussion

5.1 RNN and LSTM Comparison

Based off the results from Table 2, we can see that the baseline model with LSTM unit performs better than the normal RNN unit, with a train loss of around 1.51 and the best validation loss of around 1.93. On the other hand, the network with normal RNN unit has roughly 1.60 best train loss and 2.06 best validation loss across 100 training epochs. In both training and validation losses, the LSTM RNN unit out-performs the normal RNN unit.

As a result of the inherent structure of LSTM unit discussed before, it captures long term dependencies within the music generation sequences, thus making our model more powerful to incorporate longer context of the music pieces. This explains why the LSTM unit works better than the normal RNN unit in terms of the best train and validation loss values, given the same number of epochs.

Moreover, we see that the difference between the best train loss and the best validation loss for the normal RNN unit is larger than the model with LSTM unit – 0.4639 compared with LSTM model's 0.4239. This means the model with RNN unit is not generalize as well as the model with LSTM unit. One interpretation would be that music generation is a relatively complicated task for our model to perform. Therefore, since the complexity of the vanilla RNN unit is not as good as LSTM unit, the model with RNN unit fails to capture nuances in predicting the next most likely character and thus less capable of fitting the model well and generalize to unseen sequences.

5.2 Tuning Size of Hidden Neurons

From the result Table 2, we can see our best train loss decreases as we increased the size of hidden neurons. The best train loss from roughly 1.507 to 1.408 when we increase the number of hidden neurons from 150 to 250, where the best train loss is roughly 1.469 when we have 200 hidden neurons. It makes sense for the best train loss to show a decreasing trend because with more hidden neurons, our model can capture more hidden representations of the input sequence and thus learn better and better as we have more number of neurons. However, sometimes our model tends to overfit the training sequence, thus showing a worse generalization compared with the model that has smaller number of hidden neurons. This explains why our best validation loss increases from roughly 1.931 to roughly 1.947 when we increase the number of hidden neurons from 150 to 200. Since our model is trained in randomly clipped sequences, it is possible that our model has more power to learn the input sequence thanks to the increased number of hidden neurons, but unfortunately learns much noise in the training data so that the generalization result is not decreasing as one would expect.

Similarly, when we increase the size of hidden neurons from 200 to 250, we can see that the best train loss decreases. In this case, the best validation loss also decreases to roughly 1.861. This is an improvement over the baseline model. One possible reason would be that with even more hidden neurons than 200, our model is powerful enough to learn hidden representations of the training sequences, where the similar patterns, or representations, are also shown in the validation data. Therefore, our model fits the training data better while generalizing well on the validation set.

5.3 Tuning Dropout Probability

Dropout is an important technique that helps to prevent over-fitting, particularly for complex models with large parameters and layers. When using dropout, the model randomly sets some neurons to zero with a probability of r_d (dropout rate), thereby reducing the size of the input parameter and the complexity of the model. In theory, dropout can add noise to our model and reduce its dependency on specific neurons. This is also known as regularization. Nevertheless, we have not seen the effect of dropout regularization. When we applied the dropout rate of 0.2 and 0.3, the best validation loss decreased from 1.93 (the baseline LSTM model with a dropout ratio of 0.1) to 2.01 and 2.06, respectively. We also have not seen the effect of dropout regularization in reducing generalization, which is evidenced by the increasing differences between the best train loss and the best validation loss, as shown in Table 2. We suspect that this may also be related to the training process. We used random clipping to generate the train sequences for the training datasets, which means the structural dependency between the next character might not be as robust if we train our model from the start. With the dropout technique, our model further reduces the long-term dependency between each time state. Thus, we observed a similar performance between LSTM (with a dropout rate of 2.06) and the RNN model, which cannot effectively leverage long-term memory and has little long-term dependency due to its backpropagation problem of exploding and vanishing gradient descent.

5.4 Heatmap Insights

In this section, we analyze the heapmaps of different hidden neurons on a 282-character sequence of song, generated with our tuned baseline model, which has the best performance. The song is produced starting with "⟨start⟩" and the Sheet Music of the generated song is given as follows:

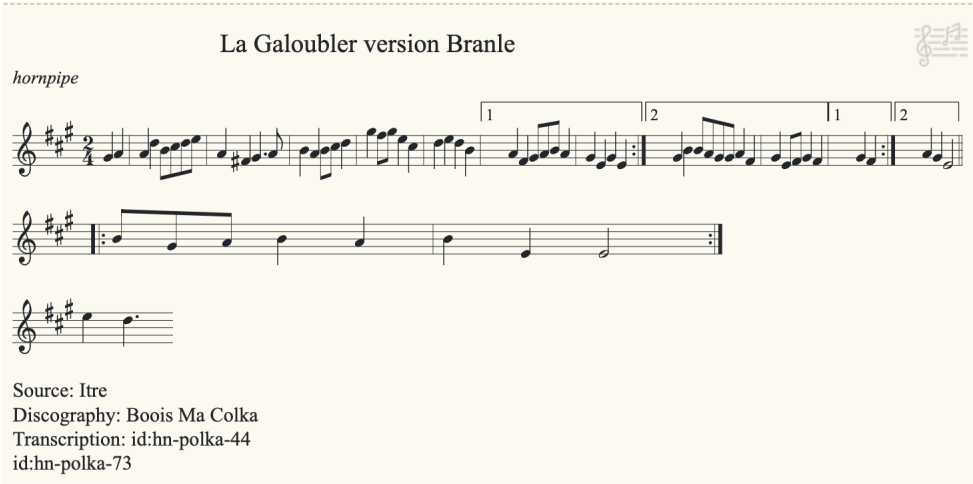


Figure 9: 282-Character Sequence Song Generated with Baseline

Generally, we notice that the first quarter of the sequence has high pitch, whereas the latter part would have lower pitch. We believe the 10th hidden neuron in our model would pick up the low pitch character in the sequence.

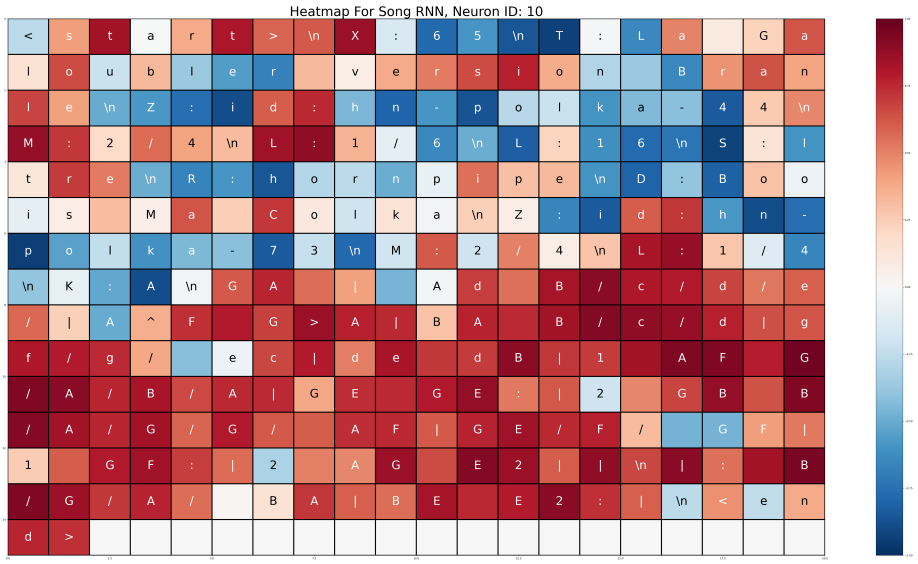


Figure 10: Heatmap From the 10th Neuron

With the above heatmap, we can see that the 10th hidden neuron is pretty activated for the latter half of the generated sequence, which, we believe, captures the low pitch that is inherent in the generated song sequence.

Another interesting heapmap is shown below, which comes from the 30th hidden neuron:

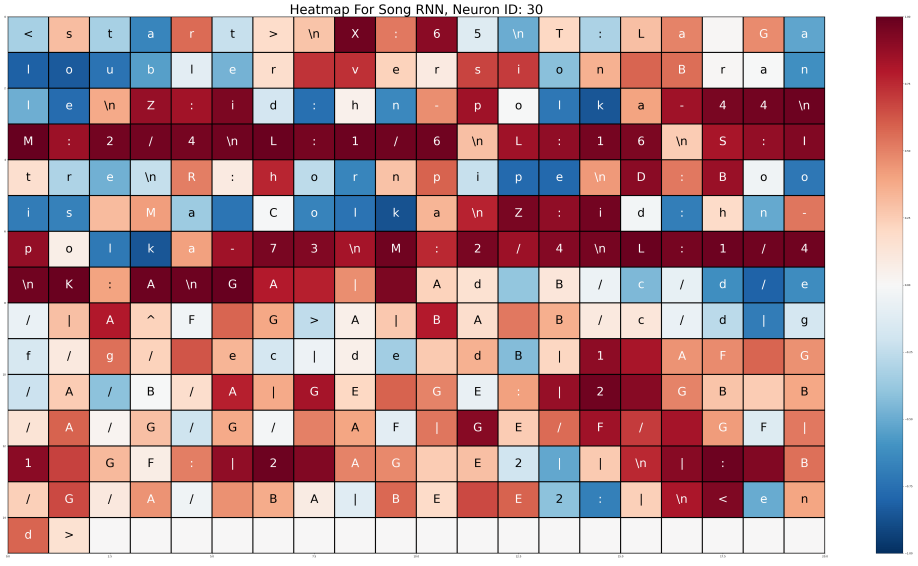


Figure 11: Heatmap From the 30th Neuron

We can see that the neuron would pick up the header of the generated song sequence. More specifically, sequences like **M: 2/4** and **L: 1/6** have pretty high activation values, whereas the body of the music would have lower activation values, besides some frequently appearing characters in the header of the music.

Besides picking up the differences between sound pitches and header/body of the music sheet, the activations from another hidden neurons detects some interesting patterns inherent in the rhythm of the generated song sequence. The heatmap is shown as follows:

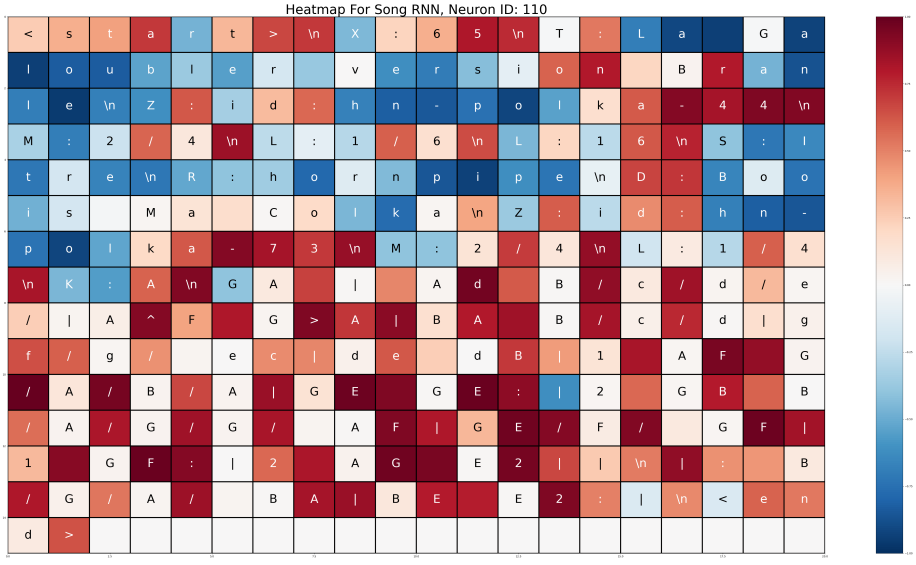


Figure 12: Heatmap From the 110th Neuron

As we can observe from the heatmap above, characters, especially **/**, are strongly activated for this neuron, which are characters that denotes separates the subsequences in the song.

6 Author's Contributions

6.1 Hargen Zheng

During the initial phase where we set up the training loop for our RNN models, I mainly helped with debugging the training loop and also working on the basic RNN model in 5a. After that, I mainly worked on fine-tuning the parameters for the size of hidden neurons and debugging code for our song generation process and heapmap plotting.

6.2 Nathaniel del Rosario

I helped with the model design by pulling ideas from related work such as RNNs versus LSTMs, with LSTM's performing better due to the idea of attention. Additionally I implemented the heatmap-generator with Chuong. The majority of my other contribution lied in formatting the report, writing the introduction paragraphs as well as researching Related Work, and helping with the parameter choices as well as writing them down in the tables.

6.3 Chuong Nguyen

I helped implement the song-generator and heatmap-generator. Additionally, I was very interested with the song generator, so I helped with experimenting on using different drop-out rates and hidden layer sizes especially on the LSTM model. I also played around with experimenting different temperature and the song's max-length to see which parameter would yield the most optimal/natural like song. Furthermore, I also played with slicing and using part of a generated song, which sounds the best, and feed it back into the generator to generate a new song. The result was interesting, as we got some pretty and some unnatural generated songs.

6.4 Adam Tran

I helped implement the initial baseline LSTM model, and worked with Ziyue Liu to get the model running. In addition, I helped with writing the Abstract, Baseline portion of the writeup and other miscellaneous editing. I helped with finalizing various parts of the report to be ready for submission.

6.5 Ziyue Liu

I implemented the baseline LSTM model with Adam Tran and worked with Hargen Zheng to debug the code in train, song generation, and heatmap generation. Finally, contribute to writing the baseline model in the method section and drop out in the discussion section.

References

[1] Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2024-02-20.

[2] Chris Donahue, Huanru Henry Mao, Yiting Ethan Li, Garrison W. Cottrell, and Julian McAuley. Lakhnes: Improving multi-instrumental music generation with cross-domain pre-training. In *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019*. 2019.

[3] Douglas Eck and Jürgen Schmidhuber. Learning the long-term structure of the blues. 2002.

[4] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. Gansynth: Adversarial neural audio synthesis. In *Proceedings of the 7th International Conference on Learning Representations, ICLR 2020*. 2020.

[5] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning*. 2016.

[6] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer: Generating music with long-term structure. In *ICLR 2018 Conference*. 2018.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[8] Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music. In *Proceedings of the 36th International Conference on Machine Learning*. 2019.