Data Engineering Techniques And Designs With Music Generation Neural Networks

Noah Solomon, Wanchunzi Yu

Abstract

The generation of music artificially is an interesting concept to many and has received a lot of attention in recent years. The advancement of neural networks has allowed for the creation of models that can seemingly generate music creatively to mimic a specific genre or composer. This project delved deep into the many ways to construct these neural networks and compared different model architectures and data engineering techniques. Three main types of models were implemented and the resulting generated music was evaluated with respect to the melody, note agreeableness, and rhythm. These models used the Bach Chorales corpus as inspiration for music generation.

Key Words: Neural Networks, Data Engineering, Music Generation

1. Introduction

Up until recently, computers have never been conventionally thought to be capable of generating art up to the standards of a regular person. Unlike other existing natural phenomena such as weather patterns, CO_2 emissions, animal populations, creative human behavior is an extremely complicated idea that has been incredibly difficult to model. With advancement of artificial intelligence(in particular neural networks), now computers are able to model situations like these effectively. This project aimed to explore different implementations of neural networks that generate music and the techniques that are used.

1.1 History of Neural Networks and Music Generation

Neural networks, like many other complicated scientific ideas, went through varying ranges of societal interest and advancement through time. A simple neural network called the "Perceptron" was first created in 1958 by Frank Rosenburg. This perceptron was used to predict whether a fly's flee instinct would be activated with respect to what it was seeing. Due to its single layer architecture the perceptron was only capable of learning linear relationships, so it was not

applicable to any real world problems. Nonetheless it introduced neural networks to the world and led to future research. The next year, researchers Bernard Widrow and Martian Hoff from Stanford created a neural network that reduced noise from phone lines. This was the first neural network that fixed a non-trivial problem and in turn brought much hype to the field. Unfortunately researchers experienced many roadblocks and in the late 60s, research in neural networks ceased entirely. A phase called the "AI winter" had begun and it was not until the 1980s that research would continue for neural networks. At this point, an already existing concept called back propagation was applied to neural networks to bypass the roadblocks that held them back. Many important discoveries and advancements have been made since the 1980s to bring the field of deep learning to where it is today.

The history of music generation relative to neural networks is far more recent and comparable to the advancement of a similar field called natural language processing. Natural language processing is a field that explores ways for a computer to extract and express information from linguistic sources. The tasks that natural language processing explores are highly useful in society. Some natural language processing tasks that most people use commonly throughout their life are auto-complete, google translate, and scam detection. Although reading, writing, and speaking may seem natural for the average person, it is actually an extremely difficult task for a computer. The advancement of vanilla neural networks were somewhat successful in completing simple natural language processing tasks but it was the invention of *long-short-term-memory* (LSTMs) neural networks that made massive strides in the field. These networks, in addition to being able to extract complex relationships, worked well with temporal data which is the type of data at which most NLP tasks use. Due to music's similar structure to language, LSTMs were applied to music generation tasks and reached a high level of success compared to previous approaches. Within the past couple years a new type of neural network called the transformer has proved to be even more effective with most NLP and music generation tasks. Today, the industry leading music generation models use transformers.

1.2 Deep Neural Networks

As with any supervised learning task a neural network requires samples that consist of inputs and target outputs. Let $f(x)$ represent the target output associated with the input of x. The task at hand for the neural network is to approximate $f^*(x)$ such that total loss, or error, between $f(x)$ and $f^*(x)$ is minimized.

The approximation function $f^*(x)$ can be abstracted as a system of layers of densely connected nodes. An example is shown below:
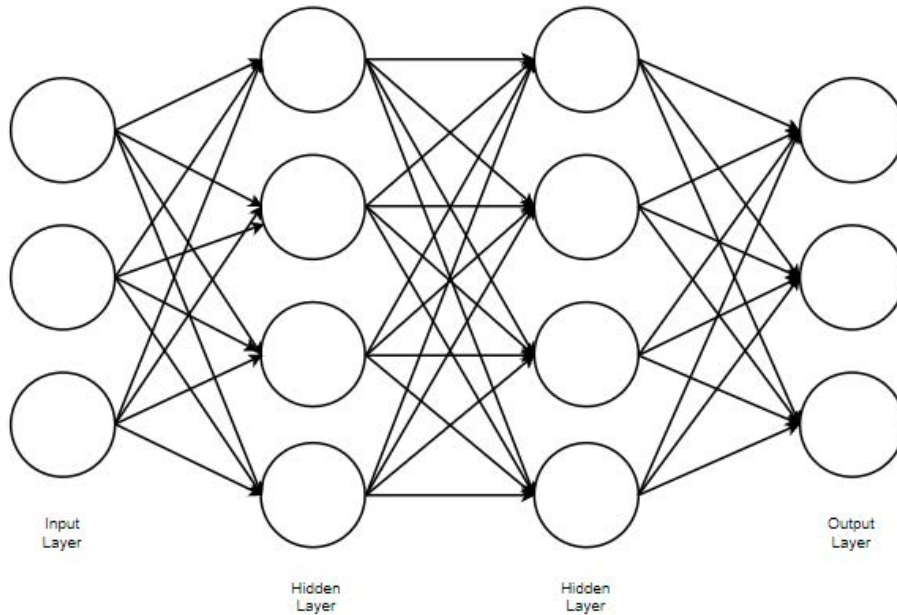
A bias parameter is associated with each node and a weight parameter is associated with each edge. The state of any given node can be calculated by taking the weighted outputs of the previous layer and adding the node's bias then applying an activation function. The weights that are used correspond to the edges coming into the node of interest. The activation function that is used depends on where the layer is positioned and the range of values that the next layer's nodes can take on. The activation that is typically used on intermediate layers is called *rectified linear unit* (RELU) and is defined as:

$$\sigma(x) = max(0, x)$$

Another common activation function is called the *sigmoid* and squeezes output values between 0 and 1. This activation is often used on nodes in the final layer in binary classification or multi-class classification problems. *Sigmoid* is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Softmax is the last activation that will be discussed. This activation function is used in multi-class classification tasks on nodes in the final layer of a neural

network. Applying the softmax function the final layer of a network creates a vector of probabilities that sum to 1. This allows the networks to "learn" a posterior probability distribution. The equation for *softmax* is the following:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Where i is the ith node in the last layer and K is the number of distinct classes. When this activation function is used $\hat{y}$ samples are to be one hot encoded so an accurate loss value can be calculated.

Unlike many other machine learning models, neural networks do not have a closed form solution to find the optimal weights and biases (defined as (**W**, **B**) or **θ**) such that the total loss is minimized. This is one of the major drawbacks of deep learning as models must iteratively be trained which can be extremely time consuming depending on the complexity of the training data and network architecture.

The algorithm that is used to train a neural network is called gradient descent. Let $\nabla_\theta L(\hat{y}, y)$ be the gradient of the loss function with respect to all parameters. Following this gradient would be going in the direction of higher loss relative to the training samples. Going the opposite direction would reduce total training error, tuning the parameters such that the predictions that are made are closer to $\hat{y}$, the target outputs. Typically the weights and biases are updated every n samples where n is the batch size. To update a weight w:

1. Calculate the partial derivative: $\frac{\partial L}{\partial w}$
2. $w_{new} = w_{old} - \gamma \cdot \frac{\partial L}{\partial w}$

Where γ is a constant called the learning rate.

There are a variety of loss functions that are used in different scenarios. A common loss function for regression tasks is squared error calculated as $L(\hat{y}, y) = (\hat{y} - y)^2$. Classification tasks instead use a loss function called *cross entropy*. This function is defined as :

$$L(\hat{y}, y) = -\sum_{i=1}^{K} y_i \cdot log(\hat{y})$$

where i is the ith output node.

1.3 Recurrent Neural Networks

While regular fully connected neural networks are effective at completing non-temporal tasks, they struggle working with time related data or data where previous outputs influence future outputs. This is where recurrent neural networks (RNNs) are useful.

The RNN performs similarly to regular deep neural networks but has an intermediate section called the hidden state which allows for previous predictions to be used for new predictions. The hidden state encapsulates all relevant information about the past outputs so an accurate prediction can be made relative to time. The RNN architecture is shown below:
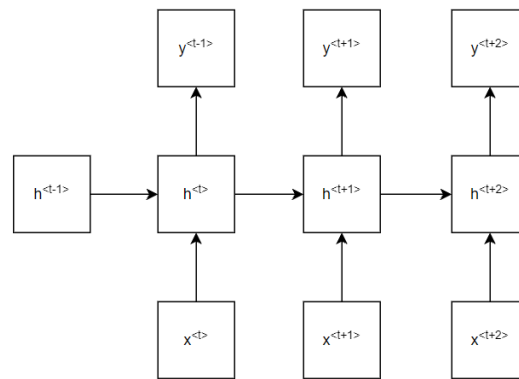
Recurrent Neural Network

Figure 1.3.1: An RNN's architecture where t represents the timestep.

Let $X = \{x^{<1>}, x^{<2>}, x^{<3>}, ..., x^{<T>}\}$ where $x^{<t>}$ means the tth element that occurs in a time-related sequence of data. The tth hidden state denoted as $a^{<t>}$ is computed as follows:

$$a^{<t>} = \sigma(W_a \cdot a^{<t-1>} + W_x \cdot x^{<t>} + b_{ax})$$

Where W_a is the weight vector associated with the hidden states and W_x is the weight vector associated with the inputs. Then:

$$y^{<t>} = \sigma(W_y \cdot a^{<t>} + b_{ay})$$

Notice that there is an optional output $y^{<t>}$ for every timestep t. Sometimes all the outputs are used while other times only a select few or even the last are used. This all depends on the task at hand. A task where an output is required at every

timestep uses a *many to many* RNN. An example of one of these types of tasks is predicting the next word of a sentence at every timestep of the input sentence. Another type of task requires the *many to one* RNN where a sequence of time related data is used to make a single prediction. An example of one of these tasks is determining the sentiment of a sentence (also known as sentiment analysis in the field of natural language processing).

In this project a more advanced RNN is used called a long-short term memory network (LSTM). LSTMs can more effectively learn what information to keep and what to forget, making them a more favorable choice than vanilla RNNs in most situations.

1.4 Embeddings

Normally when the input data to a neural network is categorical it is one hot encoded to emphasize the existence of different classes. The problem with one hot encoding is that it can create massive sparse vectors that solely convey information about the classification of the event. Embeddings project categorical data points to a vector space such that spatial similarity is captured. Data points that are alike will be projected to vectors that are close together geometrically. In music terms, notes that are close together would most likely be projected to similar vectors while notes that are far apart would be projected to distant vectors.

Embedding data points before passing them through an LSTM is a common practice as it often reduces the dimension of the data and provides more information. Embeddings were used in all the models that were created in this project.

2. The Data

The quality and quantity of the data used is one the most influential factors of how well a neural network performs. The best results come from a network that is trained with a large amount of unbiased samples from the population of interest. In addition the data must be encoded in a way that best exploits relationships that must be learned (data engineering). In most scenarios as much data as possible should be used. As a general rule of the thumb, the higher the complexity of the training data, the more samples required to prevent *overfitting* from occurring. *Overfitting* is a concept in machine learning that occurs when a model "remembers" the training data instead of generalizing.
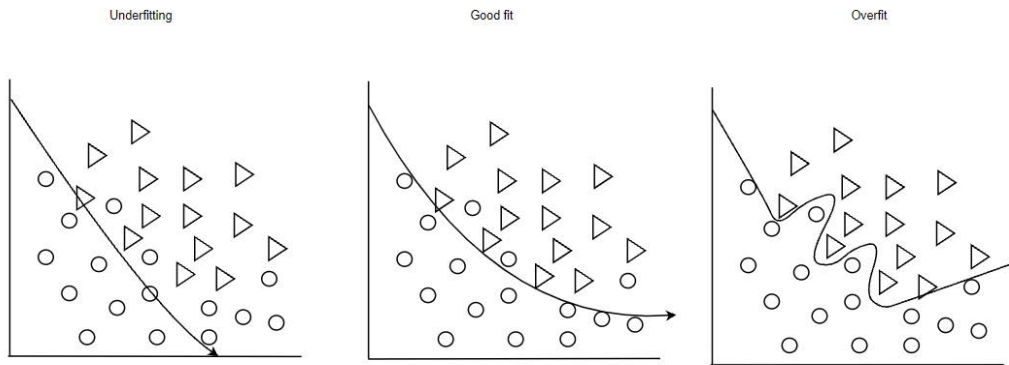
Figure 2.0.1: The difference between overfitting and underfitting with a binary classification task.

In cases of overfitting, the model will perform deceptively well on training data while performing poorly on unseen data points. Over training a neural network also causes overfitting.

Since neural networks are prone to overfitting it is essential to partition the dataset into a training set and testing set. Most data should be reserved for the training dataset but around 20% of the data can be used as the testing dataset. Then the neural network is trained using the training data and evaluated on the unseen test data to get an accurate idea on how well the model performs. The models created in this project used a *validation set* which allowed the models to be evaluated during training instead of after training as the testing set would be.

Data engineering is the process of taking raw data and manipulating it such that the transformed data can be used in a machine learning model. The choices that are made during this step will drastically change how a neural network performs. Even though two forms of data contain the same amount of information, one of the forms may expose important relationships more clearly. There are many techniques that can be used during the data engineering stage but there is no golden standard. Realistically the choices that are made come from a mixture of intuition/trial and error/domain knowledge. The choices made in this project are discussed in **section 4.**

Throughout this paper *events* will continuously be referenced and it is important to define it for consistency purposes. An *event* as mentioned in this paper is a piece of data that retains information about a musical action being performed through time. An event is represented as a vector of length n where n is the number of distinct pieces of information. The following is an event with 2 pieces of information:

$$event \;=\; [note\ on,\ c4]$$

This would signal for middle c to be pressed down. Information can be added as needed as well for example $[note\ on,\ c4,\ mf]$ would add a piece of information about the volume of the note. Ultimately the individual pieces of information in each event are integer encoded to be compatible with embedding layers which are used in all models in this project.

2.1 MIDI

MIDI (Musical Instrument Digital Interface) is the main type of raw data used in this project. It is the most common digital representation of music and data was found easily online. MIDI's consist of *messages* which possess the parameters of *type, note, velocity,* and *delta time*. *Type* can either be *note_on* or *note_off* to signal pressing down and releasing notes respectively. Other types of messages exist but they are not important to this project. *Note* is the midi representation of the pitch that is to be played or released. Middle C or C4 would be encoded to 60. *Velocity*, although not used in this project, is how loud the note is. *Delta time* is the number of *ticks* (An arbitrary unit of time determined by meta) that pass after the previous message is activated.

```
note_on channel=0 note=36 velocity=96 time=151
note_on channel=0 note=37 velocity=96 time=705
note_off channel=0 note=36 velocity=0 time=527
note_on channel=0 note=39 velocity=96 time=504
note_off channel=0 note=37 velocity=0 time=578
note_off channel=0 note=39 velocity=0 time=403
note_on channel=0 note=41 velocity=96 time=603
note_on channel=0 note=36 velocity=96 time=855
note_off channel=0 note=41 velocity=0 time=51
note_off channel=0 note=36 velocity=0 time=729
note_on channel=0 note=39 velocity=96 time=830
```

Figure 2.1.1: An example of MIDI encoded data.

2.2 Music21

This project used a library called *music21* which had collections of music scores parsed out in an organized way. The *music21* scores were separated out by voice (soprano, alto, tenor, and bass) and time measured out in traditional musical time units (quarter notes, eighth notes etc). This structured form of data was easier to

work with and the separation out by voices allowed for an interesting and effective neural network architecture to be used.

3. General Approaches

Using the softmax activation function in the final layer of a neural network causes the output to be a vector of probabilities that sum to 1. In a sense, the neural network can learn to approximate a posterior distribution $p(Y|X,\theta)$. There are 2 general approaches that configured and used this posterior distribution to generate music creatively.

3.1 Sequential Predictions

The first method used involved training a neural network to use previous events to predict a future event. Generating new pieces using this approach is recursive and is done by using previous predictions as input to additional predictions. Let n be the number of previous events that the neural network will have access to when making predictions. Pieces can either be initialized using n start tokens or picking the first n events of a random Bach chorale. The piece before generation would be:

$$[event_1, \ event_2, \ ... \ , \ event_n]$$

Then for any $t > n$,

$$event_t \ = \ sample(p(Y|X,\theta))$$

Where $X = [event_{t-n}, \ event_{t-(n-1)}, \ ... \ , \ event_{t-1}]$

and $Y = \{event_1, \ event_2, \ ... \ , \ event_K\}$

Since new predictions use old predictions as input data, events must be generated in order. That means $event_{t-1}$ must calculated before $event_t$ because the prediction for $event_t$ uses $event_{t-1}$.

Each piece is able to generate $N - (n - 1)$ samples. Sample i would consist of:

$$(x_i, y_i) \ = \ ([event_i, \ event_{i+1}, \ ... \ , \ event_{i+n}], \ event_{i+n+1})$$

Sequential Predictions

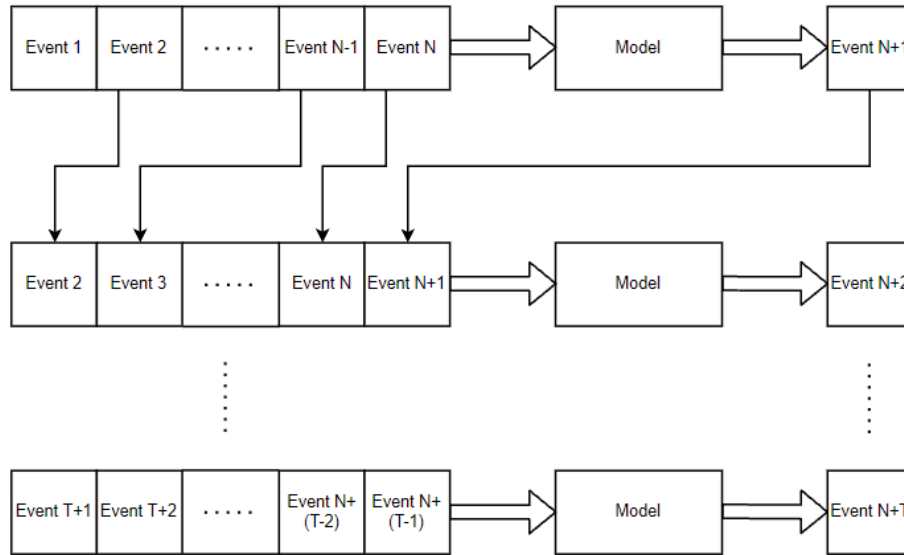For T timesteps using sequences of length N

Figure 3.1.1: Visual representation of sequential predictions generation approach.

3.2 Resampling

The second method of generating music that was used is a process of resampling from an existing or randomly initialized piece. The resampling process involves selecting an event at random and re-predicting it with the context of n previous events and n future notes. Let N be the number of events in an existing piece. Then the full piece before the re-sampling process would be:

$$[event_1, \ event_2, \ ... \ , \ event_N]$$

Then for any $n < t < N - n,$

$$event_t \ = \ sample(p(Y \,|\, X, \boldsymbol{\theta}))$$

Where $X = [event_{t-n}, \ event_{t-(n-1)}, \ ... \ , \ event_{t+n}] \setminus event_t$

and $Y = \{event_1, \ event_2, \ ... \ , \ event_K\}$

This process is described in Gaetan Hadjeres Deep Bach's research paper and is called pseudo Gibbs sampling.

4. Data Engineering/Model Design Choices

Throughout this project there were many choices that had to be made when parsing data and designing the models. It was discovered that most of these choices make a massive impact in the validation accuracy/loss of a model and the quality of its generated music.

4.1 Sequence Length

The first main choice that was made for all the models was selecting a *sequence length*. The sequence length is how many events a model has access to while making its prediction. A higher sequence length gives the model more context and allows more global trends to be learned. A smaller sequence size restricts how much context the neural network has and only allows for localized trends to be learned but training time is reduced. For each of the models a "small" sequence size was used due to the tendency for localized patterns to be present in Bach chorales. There was not a universal sequence length that was used for all the models because the training data differed in how much information was captured in one event. For example the *on-off* network required a higher number of events to capture the same amount of information relative to the *durational* net.

4.2 Transpositions/Converting To C

Another choice that was made was choosing to transpose samples to generate more training data or convert all pieces to the constant key signature of C to reduce the complexity of trends existing in the data.

Transposing was effective if done in a way such that new events were not introduced into the sample space. Since the y vectors were one hot encoded with each index representing an event (or part of an event), introducing new notes to the sample space would increase the number of dimensions in the output. As the number of options increases, the harder it is to make an acceptable choice. Therefore each sample had a minimum and maximum number of half steps at which it could be transposed by. Transposing the samples increased the number of samples by a magnitude which had a positive effect on all models but dramatically increased the amount of time it took to train a model. It was also

necessary to partition the largest datasets and load each into memory one at a time to prevent from overloading ram.

Converting pieces to the constant key signature of C was a challenge with midi data because oftentimes the midis do not have key signature data. However the music21 corpus of Bach Corpus did have key signature data for each chorale. As a side project a neural network was trained to predict key signature based on a vector of note proportions using the music21 data. This model performed well and achieved an accuracy of ~90%. It was used to predict key signatures of the Bach chorales that were in MIDI format which the *On-Off* and *Durational* networks both used. Then all the midis were converted to the key signature of C. So far the *On-Off* network was the only model that was trained using the pieces that were transposed to C and the results are surprisingly good. The models train about 10 times quicker and reach loss/accuracy values that are almost as good as the models trained with transpositions. It seems like this is a good option if losing a point or 2 of accuracy is not a problem and speed is the main selling point.

4.3 Information That The Network Has Access To

The information that a model has access to in the training data/ how the information is encoded was shown to make an impact into the resulting generated music heavily. The main 2 pieces of information that all the models included in some form were the notes being played and some sense of timing. The models differ in how they encode this data as the *On-Off* network records time relative to previous events while the *Durational* network records time by duration of a note. In *Deep Bach* there is a constant amount of time between each event in the input sequences. There is also additional information that is encoded into the events of *Deep Bach* such as the key signature, subdivisions of a beat, and existence of fermatas. Oftentimes, the more relevant information that a network has access to the better it performs its task. That being said it is not always possible to find additional information that is relevant to the task at hand and if it is, training time will increase.

4.4 Hyper-Parameters

Tuning hyper-parameters for a neural network is a difficult and tedious task especially when training one model takes a long time. The models trained in this project were complex and used a large amount of data so training was a lengthy process. For this reason hyper-parameter optimization was not a huge focal point. The only form of hyper-parameter tuning was for each model type creating a handful of models with varying levels of complexity. Surprisingly it was discovered that selecting a small value for the embedding output dimension created a bottle neck and decreased accuracy/increased loss a fair amount. For

example with the transposition version of *On-Off* Network the validation accuracy increased nearly 3% when increasing the embedding dimension from 15 to 75.

5. Models

There were 3 main types of models with varying designs that were trained until validation loss stopped increasing. In this section, the architecture will be explained then the model will be evaluated based on the following qualities:

1. Training time
2. Data complexity
3. Melody
4. Note agreeableness
5. Rhythm

5.1 Model 1: *On-Off* Network

The first model that was designed used the sequential predictions approach and events consisted of a vector of length 1. The events used, played off the note on - note off trend already existing in MIDI allowing for a convenient data parsing process. There are 3 main types of events that can be classified as listed below.

1. [note_on_<*pitch*>]: Note with corresponding *pitch* to be pressed down
2. [note_off_<*pitch*>]: Note with corresponding *pitch* to be released
3. [rest_<*time*>]: Rest *time* amount of ticks

Since the events have 1 piece of information, the neural network will have 1 input being a sequence of integer encoded events. The sequence of integers then goes through an embedding layer which maps each integer to a vector. At this point the sequence of vectors passes through an LSTM which extracts the temporal relationships between events. The output is sent through a fully connected dense layer activated with the softmax function to output a vector of probabilities. The length of the output vector is equal to:

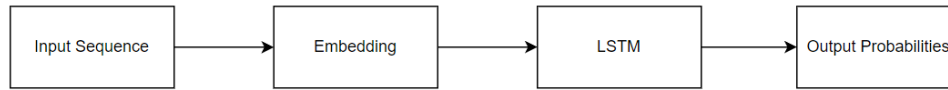$$2 \cdot |\{x_i : x_i \text{ is distinct note}\}| + |\{x_i : x_i \text{ is distinct time}\}|$$

Figure 5.1.1: A diagram of the *On-Off Network's* architecture.

Results

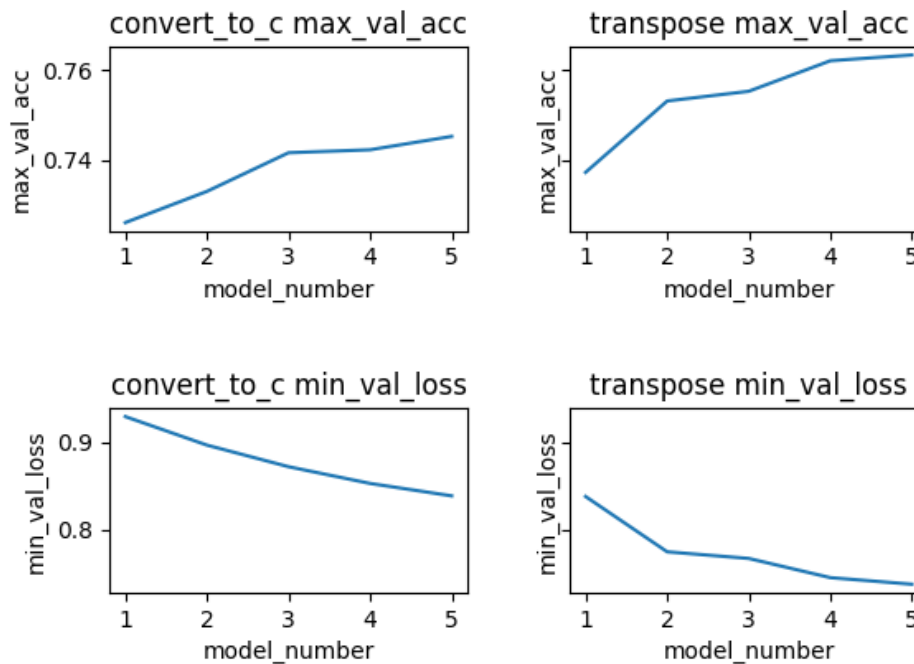Training each *On-Off Network* model took about an hour

Figure 5.1.1: These plots compare validation loss and accuracy between the transpose and convert to c models. As the model number increases so does the complexity of the model with respect to its hyper-parameters.

1. Training time - C/A

Training the transposition model took a long time while training the convert to c models took far less time.

2. Melody - D

The melodies generated do resemble the Bach chorales to an extent, but they are more chaotic and have no structure.

3. Note agreeableness - C

Most notes that are generated to go well together but there are definitely a handful of predictions that are unnatural. Also sometimes notes are signaled to be pressed but never released which results in muddiness.

4. Rhythm - A

Rhythm is decently modeled and sound close to the original Bach chorales.

5.2 Model 2: *Durational Network*

The second type of model that was designed aimed to transform the data from the previous network into a more usable and direct form. The previous data captured note on, note off, and rest occurrences while the *duration* network solely used note ons and corresponding durations. An event consisted of 2 pieces of information: the signal, and the duration. The signal could either be the note that was to be pressed down or a "rest". The duration was the amount of ticks that the event would be activated for.

Events could be the following form:

1. [note_on_<*pitch*>, <*duration*>]: Note with corresponding *pitch* to be pressed down for *duration* ticks
2. [rest, <*duration*>]: Signal to rest for *duration* ticks

This network is multi-input and multi-output since the events have two components each. The input of a sample consists of two input arrays where one is a sequence of the signals and the other is a sequence of the durations. All the elements in the signal sequence and duration sequence are integer encoded to allow for compatibility with embeddings. Both the sequences go through different embedding blocks and the outputs are combined. Then the sequence of combined embeddings go through an LSTM. The LSTM output is first sent through a fully connected layer activated by softmax to represent the signal prediction. Then the LSTM output is combined with the signal prediction probabilities and sent through another fully connected layer activated with softmax to represent the duration predictions. The duration prediction output uses the signal prediction because the signal and duration are heavily dependent. Whether a specific note or

rest predicted is chosen changes the range of acceptable predictions for the duration.
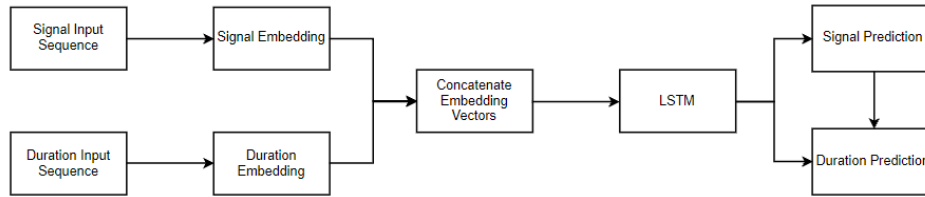

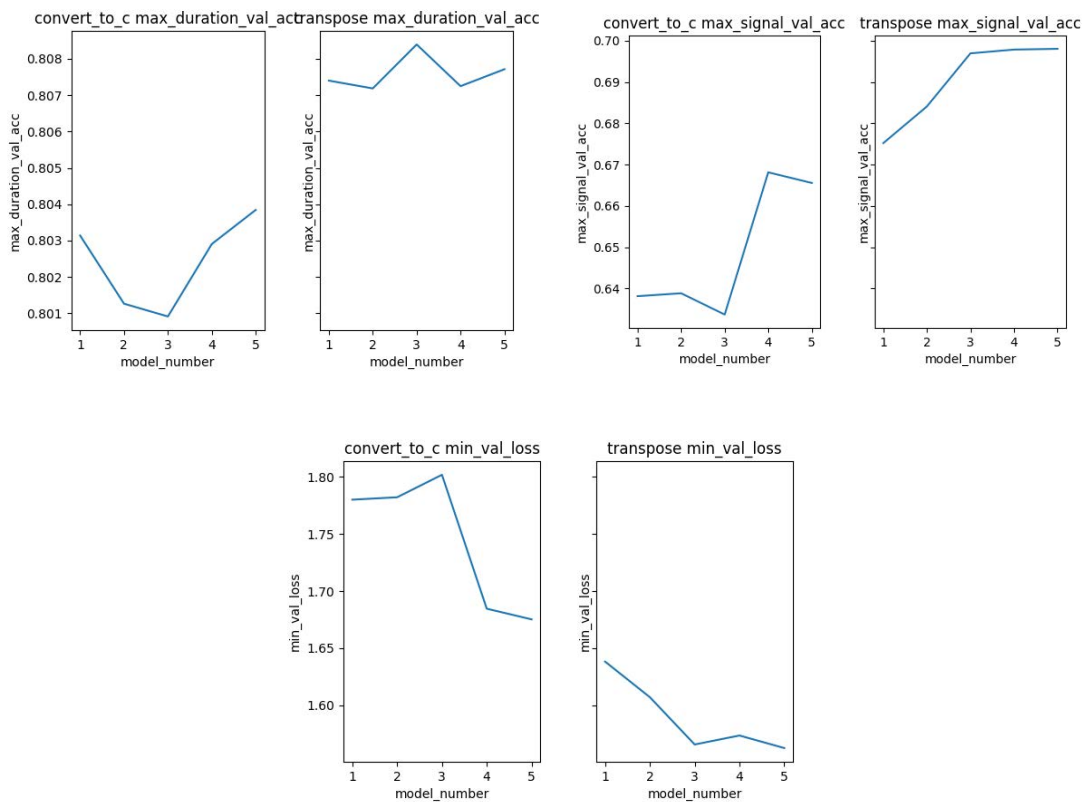**Figure 5.2.1:** Network design of the *Duration Network*

Results

Figure 5.2.1: These plots compare validation accuracy and loss between the transpose and convert to c models.

The model number once again corresponded with the dimension of the embeddings and dimension of the LSTM. Increased complexity resulted in better performance. Compared to the *on-off* network, the *durational* network has far more validation loss (at best 1.55 compared to 0.75) but this is due to the multiple outputs. The validation loss is the total loss between both the signal and duration outputs so there is naturally more net error. There are two validation accuracy metrics: one for the signal and one for the durations. The signal accuracy peaks at 68% while the duration peaks at 80% suggesting that the rhythm is a more extractable trend than the note choice. Although it is difficult to compare the accuracy metrics from a multi-output model to a single output model, based on the metrics it seems as if the *Durational* network performs similarly or maybe even worse than the *On-Off* network. But this is far from the truth as the music generated from the *Durational Network* is noticeably improved.

1. Training time - B

Training the transposition model took far less time than the transposed *on-off* network. Models on average took about 15 minutes to train completely.

2. Melody - C+

The melody is still chaotic and has no clear direction. It is easily distinguishable between the actual Bach Chorales.

3. Note agreeableness - B

The majority of notes locally go well together. Note agreeableness is greatly improved compared to the *on-off* network.

4. Rhythm - B+

Rhythm is slightly improved from the *on-off* network.

5.3 Model 3: *Deep Bach*

Deep Bach is a model designed and implemented by Gaetan Hadjeres. This model consists of four neural networks - one for each voice (soprano, alto, tenor, bass) and uses the resampling method to generate music. The data that is used comes from the music21 library in python and an event consist of seven pieces of information:

1. Soprano note: midi representation of note in soprano voice
2. Alto note: midi representation of note in alto voice
3. Tenor note: midi representation of note in tenor voice
4. Bass note: midi representation of note in bass voice
5. Subdivision: what quarter beat is in - can be 0,1,2,3
6. Key signature: key signature of piece that sample came from

7. Fermata: 1 if fermata is present in, 0 otherwise

The overall process of music generation involves many iterations of a random voice and note within the voice being selected to be resampled. The previous N events, future N events, as well as current notes being played in the other voices are used as input data. This context is used to predict the random note that was selected to be resampled. The prediction comes from the model that was trained to predict the voice of the note being resampled.

Each voice model is separated into four parts: left LSTM, center fully connected layer, right LSTM, fully connected output. A voice model starts with 3 main types of input: previous events, the current event, and future events. The previous events form a sequence of the previous events and each individual component is passed through a distinct embedding layer. The sequence of combined embedded vectors then goes through the left LSTM. A similar process is done to the future events, but as an additional step, the order of the events are reversed. The current event input goes through the embeddings and the output is inputted into a fully connected layer. The right LSTM, center fully connected, and left LSTM outputs are concatenated and sent through one last fully connected layer which outputs the note probabilities for the note being resampled.
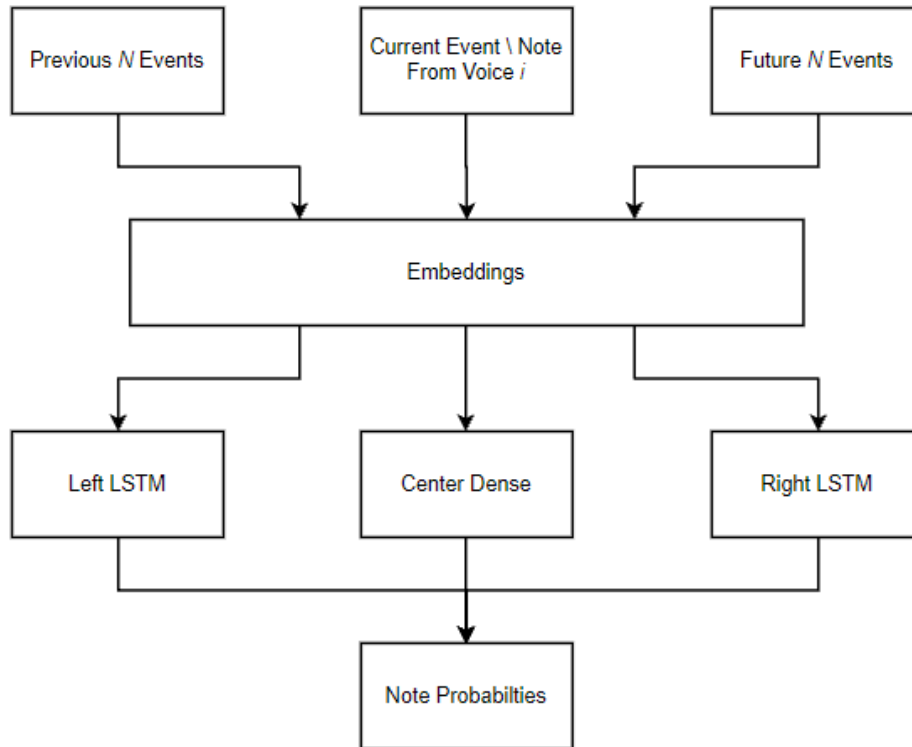
Neural Network For Voice *i*

| Previous *N* Events | Current Event \ Note From Voice *i* | Future *N* Events |

Embeddings

Left LSTM | Center Dense | Right LSTM

Note Probabilties

Figure 5.3.1: The network design of the *Deep Bach*

Results:

Compared to the 2 other models *Deep Bach* took an incredibly long time to train. This is due to the high complexity of the data as one sample's input consists of two sequences of seven dimension vectors as well as one six dimensional vector for the current event. Each voice had to be modeled which also added to the training time heavily. With a sacrifice of training time however, came a model that performed far better than the previous two networks. Each voice model was highly specialized and all of them combined formed a highly effective model. Only one complete model was trained due to the lengthy training process but the metrics were extremely impressive.

Voice Model	Min Loss	Max Accuracy

Soprano	0.171	0.954
Alto	0.221	0.932
Tenor	0.264	0.921
Bass	0.178	0.946

Table 5.3.2: The validation loss and accuracy values for each voice model.

Once again, it is difficult to compare models that have completely different structures, but *Deep Bach* clearly performs the best on paper compared to the *Durational Network* and *On-Off Network*. The generated music is also far better.

1. Training time - D
Training all the voice models took hours which makes hyper-tuning and experimentation nearly impossible.
2. Melody - B+
The melody is much more controlled and tame compared to the other two networks.
3. Note agreeableness - B+
Most notes go well together locally.
4. Rhythm - A
Rhythm closely mimics the Bach Chorales.

6. Conclusion

Based on the results of the three models trained in this project, it is clear that data engineering, and model design choices heavily impact performance. The *On-Off Network* conveniently took in data that had a structure directly derived from MIDI and the resulting music was low quality. From here the data was manipulated such that the *note on-note off* trend was eliminated. This network objectively performed far better than the *On-Off Network* but it still struggled to generate coherent melodies that wouldn't spiral out in chaotic directions. This led to the implementation of an already existing model called *Deep Bach* which separated out the music by voice and used the resampling method to generate music. This model, by far, most effectively learned the existing trends in the Bach Chorales corpus and was able to generate the best music. This drastic difference in outcomes among each of the models shows that data engineering and model design choices hugely determine the success that any given neural network reaches.

Acknowledgements

References

Amidi, Afshine, and Shervine Amidi. "VIP Cheatsheet: Recurrent Neural Networks." (2018).

Briot, Jean-Pierre, Gaëtan Hadjeres, and François-David Pachet. "Deep learning techniques for music generation--a survey." *arXiv preprint arXiv:1709.01620* (2017).

Chollet, Francois. *Deep learning with Python*. Simon and Schuster, 2017.

Hadjeres, Gaëtan, François Pachet, and Frank Nielsen. "Deepbach: a steerable model for bach chorales generation." *International Conference on Machine Learning*. PMLR, 2017.

Rothman, Denis. "Transformers for Natural Language Processing Build Innovative Deep Neural Network Architectures for NLP with Python, Pytorch, TensorFlow, BERT, RoBERTa, and More." (2021).