Generating Polyphonic Music Using a Note Invariant RNN with LSTM

Trevor Gordon
M.S Electrical Engineering
Columbia University
tjg2148@columbia.edu

Jannik Wiedenhaupt
M.S Data Science
Columbia University
j.wiedenhaupt@columbia.edu

Abstract

We describe a neural network architecture that enables prediction and composition of polyphonic music and preserves the note-invariance of the dataset. Specifically, we demonstrate training a probabilistic model of polyphonic music using a set of parallel, tied-weight recurrent networks, inspired by the structure of convolutional neural networks. This approach is first introduced by the paper "Generating Polyphonic Music Using Tied Parallel Networks" by Daniel D. Johnson which we aim to replicate in this paper. The model we create is given minimal information on the musical domains, and tasked with discovering patterns present in the source dataset. We conduct multiple experiments to compare different architectures and levels of specification to provide insight into how changes affect the prediction quality. Link to our music output: https://youtu.be/CF0Wi2WEQOU

1. Introduction

Sequence learning using long short-term memory (LSTM) has become an increasingly common technique in a growing number of research fields, e.g., natural language understanding, image generation, text prediction, and also music generation. Music generation through the use of LSTM is regularly done by training an LSTM-based neural network (NN) on sequences of notes from songs and predicting the next note in the sequence Eck and Schmidhuber, 2002. Due to the existing patterns in music like chords, scales, pitch classes, and octaves, research can expect NNs to recognize these patterns and generate human-like music. These networks work similar to textual recurrent neural networks (RNNs) that predict the next character based on an input sequence of previous characters. They can be trained on a short sequence of notes and continually sample from the model's output distribution to generate the next note which is then fed back into the input sequence. This creates a potentially infinite prediction series, i.e. generation. However, these models also regularly repeat notes too many times or generate too long breaks between notes Johnson, 2017. Moreover, many models lack a consistent theme or structure (melodies, tempo, pitch, key, chord progression). This is particularly important to the perceived quality of the generated music. While high prediction accuracy could be achieved without accounting for music structure and for example holding notes, the generated music likely does not accomplish the goal of sounding like human-created music.

Different approaches to solve these problems have been tried in recent years. The basic Melody-RNN from Google for example uses a dual-layer LSTM model, with one-hot vectors as input that represent extracted melodies Waite et al., 2016. Another approach, evolving RNNs, attempts to find a NN that maximizes the chance of generating good melodies by implementing composition rules on tonality and rhythm in the fitness function Chen and Miikkulainen, 2001.

One paper and NN that stands out from the others, is the biaxial-RNN using tied parallel networks Johnson, 2017. It generates natural-sounding music by being carefully designed to understand time signature, be time and note-invariant, and understand chords.

We want to replicate this paper for three reasons. First, creative or pseudo-creative NNs are a very relevant research area that continues to advance and has the potential to shape the future of entertainment. Second, its custom architecture makes it an incredibly exciting project as it requires us to think carefully and deeply understand how shaping and adapting inputs and outputs can dramatically improve the performance of NNs. Third, the original code used for the paper does not use state-of-the-art deep learning libraries and therefore can be simplified and its performance can be improved.

2. Summary of the Original Paper

In his paper, Daniel Johnson implements his Biaxial-RNN using parallel, tied-weight recurrent networks, inspired by the structure of convolutional networks. His model is designed to be invariant to transpositions, but otherwise is intentionally given minimal information about the musical domain so that it is able to generalize where possible. It is tasked with discovering patterns like chords and musical transposition. This approach attains high performance at a musical prediction task and successfully creates note sequences which possess measure-level musical structure.

2.1. Methodology of the Original Paper

The model is designed to have the following properties.

- Understand time signature: Compose mostly strict to a fixed time signature

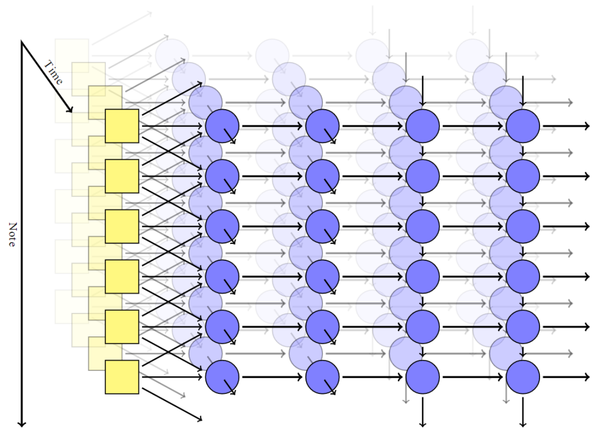

Figure 1. Simplified structure of the original RNN (Johnson, 2017)

- Time-invariant: Compose indefinitely and is identical for every time step
- Note-invariant: Transpose music up and down with identical network structure
- Chord progression: Play multiple notes at the same time in coherent chords
- Articulation: Hold notes over multiple beats or play them individually

He achieves these characteristics through his model architecture and by reshaping the inputs. Timesteps are split up across notes and every note is input as a vector of the following form.

- Position [1]: The MIDI note value of the current note. Used to get a vague idea of how high or low a given note is, to allow for differences (like the concept that lower notes are typically chords, upper notes are typically melody).
- Pitchclass [12]: The pitchclass will be 1 at the position of the current note, starting at A for 0 and increasing by 1 per half-step, and 0 for all the others. Used to allow selection of more common chords (i.e. it's more common to have a C major chord than an E-flat major chord)
- Previous Vicinity [50]: Previous vicinity gives context for surrounding notes in the last timestep, one octave in each direction. The value at index 2(i+12) is 1 if the note at offset i from current note was played last timestep, and 0 if it was not. The value at 2(i+12) + 1 is 1 if that note was articulated last timestep, and 0 if it was not.
- Previous Context [12]: The value of the previous context at index i will be the number of times any note x where (x-i-pitchclass) mod 12 was played last timestep. Thus if current note is C and there were 2 E's last timestep, the value at index 4 (since E is 4 half steps above C) would be 2.
- Beat [4]: This is essentially a binary representation of position within the measure, assuming 4/4 time. With each row being one of the beat inputs, and each column being a time step.

Johnson uses a deep NN of LSTM-layers and a final dense layer with sigmoid activation. More specifically, the model is build on the RNN-NADE architecture that combines RNNs to capture temporal interactions, and a neural autoregressive distribution estimator (NADE), which calculates the joint probability of a vector of binary variables $\mathbf{v}$.

To overcome the difficulty of NADE to capture relative relationships between inputs which is crucial when considering notes, transferring the behavior of convolutional neural networks to RNNs. This achieves transposition invariance and captures some of the most important relationships between notes, but also prevents the network from learning any precise dependencies that extend past the size of the window. As an alternative, he replaces the NADE portion of the network with LSTM layers that have recurrent connections along the note axis. This combination of LSTMs along two different axes (first along the time axis, and then along the note axis) will be referred to as a "bi-axial" configuration.

2.2. Key Results of the Original Paper

Quantitative Analysis: The tied-parallel architectures (BALSTM and TP-LSTM-NADE) in the paper perform noticeably better on the test set prediction task than the simple RNN-NADE model from other papers and many architectures closely related to it. The BALSTM network performed the best across datasets. The TP-LSTM-NADE network, however, appears to be more stable, and converges reliably to a relatively consistent cost. Both tied-parallel network architectures perform comparably to or better than the non-parallel LSTM-NADE architecture. Additionally, the two tied-parallel architectures achieve the desired translation invariance. Both models perform comparably on the original and transposed datasets, while other architectures fail.

Model	JSB Chorales	MuseData	Nottingham	Piano-Midi.de
Random	−61.00	−61.00	−61.00	−61.00
RBM	−7.43	−9.56	−5.25	−10.17
NADE	−7.19	−10.06	−5.48	−10.28
RNN-RBM	−7.27	−9.31	−4.72	−9.89
RNN (HF)	−8.58	−7.19	−3.89	−7.66
RNN-RBM (HF)	−6.27	−6.01	−2.39	−7.09
RNN-DBN	−5.68	−6.28	−2.54	−7.15
RNN-NADE (HF)	−5.56	−5.60	−2.31	−7.05
DBN-LSTM	−3.47	−3.91	−1.32	−4.63
LSTM-NADE	−6.00, −6.10	−5.02, −5.03	−2.02, −2.06	−7.36, −7.39
TP-LSTM-NADE	−5.88, −5.92	−4.32, −4.34	−1.61, −1.64	−5.44, −5.49
BALSTM	−5.05, −5.86	−3.90, −4.41	−1.55, −1.62	−4.90, −5.00

Figure 2. Log-likelihood performance for the non-transposed prediction task. For LSTM-NADE, TP-LSTM-NADE, and BALSTM, the two values represent the best and median performance across 5 trials. Data for other models is reproduced from other papers.

Qualitative Analysis: The generated samples possess complexity and intricacy. They include rhythmic consistency, coherent chords, melody, as well as consistency across multiple measures, and exhibit smooth transitions. However, they change styles more frequently than one would expect from human composers.

3. Methodology

We attempt to follow a methodology as similar to the original paper as possible. However, as we use a different technical framework, TensorFlow compared to the original Theano, we approach the problem of music generation using multiple models and comparing their performance with each other. We implement six experiments to test how data pre-processing, model architecture, and hyper-parameter tuning affect the output accuracy and the quality of the generated music.

Like the original model, our final model has an RNN-NADE structure using LSTM to predict single notes based on the same note invariant data rearrangement, Johnson uses. We achieve this in our last experiment.

Ultimately, the goal is to implement the biaxial architecture of the original paper. Unfortunately, due to time constraints and the high complexity of the original implementation, we did not achieve exact replication. The original model is complex because it swaps the batch and time dimensions mid-training to achieve time invariance. This will be explained in more detail later.

Instead, we focused on analyzing the performance and model architectures of different experiments to understand the impact of note invariance on the quality of the generated music. We experiment with different approaches to pre-process data and limit connectivity between LSTM units. Particularly, we investigate the number of input beats, batch size, note vicinity, capacity of the model, and musical intelligence. While we will showcase the results of these experiments in chapter 5 and implement them in our final model, we will mainly describe our final model, that is, the one closest to replicating the original paper. We do this to clearly explain our model structure without listing different architectural decisions and their impact.

3.1. Objectives and Technical Challenges

The objective of our project is to generate polyphonic music that possesses the characteristics outlined in chapter 2. This requires the neural network (NN) to learn to accurately predict which notes are played in the next timestep. Additionally, this entails that the quality of the generated music. The generated music should sound as natural as possible, contain common chords, and be generated infinitely.

The major technical challenges are related to data processing. Therefore, we test different ways of manipulating data and use parameters to determine how much of the input data, the model can process at any time in our experiments. For our final model, we achieve the data processing of the original paper. These experiments help to understand which features determine the prediction and generation performance of the model and shed more light of the exceptional quality of Johnson's model.

To achieve the note invariant shape of the original paper, we need to convert the data into beats, add articulation, and use a convolution-like approach to get the previous vicinity. This is challenging as it involves complex reshaping and transposing of large matrices. Additionally, to evaluate the model we recover the original shapes during training to measure the models' outputs against the labels. Next, our model generates music by predicting one time step at a time feeding each of the outputs back into the inputs for the next time step. Lastly, we convert the generated music back to a midi file to play it.

3.2. Problem Formulation and Design Description

We target the following problem: Feed a NN a number of classical piano midi files and convert the data to be note invariant. We train the model on the aggregated dataset to predict the next note. We need to solve the following subproblems:

- Convert midi files to a machine-readable dataset
- Transform the dataset to be note invariant
- Develop a model that learns to predict the next timestep
- Select appropriate model parameters
- Generate an infinite number of coherent notes

The architecture of our final model and all conducted experiments is mainly based on LSTM-layers. LSTM has been shown to be able to learn long-term temporal sequences Kaiser and Sutskever, 2016. The formulas of a single LSTM cell with inputs $\mathbf{x}_t$ and hidden recurrent activations $\mathbf{h}_t$ at timestep t are given as

$$\begin{aligned}
\mathbf{z}_t &= \tanh\left(W_{xz}\mathbf{x}_t + W_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z\right) && \text{block input}\\
\mathbf{i}_t &= \sigma\left(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i\right) && \text{input gate}\\
\mathbf{f}_t &= \sigma\left(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f\right) && \text{forget gate}\\
\mathbf{c}_t &= \mathbf{i}_t \odot \mathbf{z}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} && \text{cell state}\\
\mathbf{o}_t &= \sigma\left(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o\right) && \text{output gate}\\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{c}_t\right) && \text{output}
\end{aligned}$$

and are represented in Figure 3, the schematic of an LSTM cell.

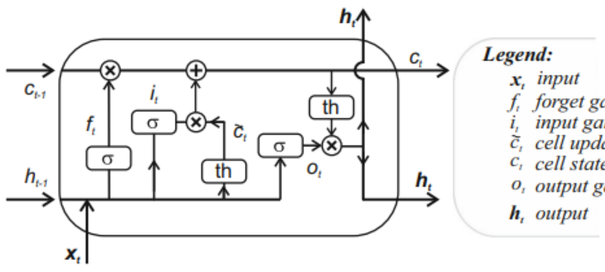

Figure 3. Schematic of an LSTM cell Hrnjica and Bonacci, 2019

Our final model, shown in Figure 5, is a deep NN of three LSTM-layers and a final dense layer with sigmoid activation. More specifically, the model is built on the RNN-NADE architecture that combines RNNs to capture temporal interactions, and a neural autoregressive distribution estimator (NADE), which calculates the joint probability of a vector of binary variables $\mathbf{v} = [v_1, v_2, ..., v_n]$ (the set of notes played at the same timestep) using a series of conditional distributions:

$$p(\mathbf{v}) = \prod_{i=1}^{|\mathbf{v}|} p(v_i|\mathbf{v}_{<i})$$

with each conditional distribution given by

$$\mathbf{h_i} = \sigma(\mathbf{b}_h + W_{:,<i}\mathbf{v}_{<i})$$
$$p(v_i = 1|\mathbf{v}_{<i}) = \sigma(\mathbf{b}_h + V_{i,:}\mathbf{h}_i)$$
$$p(v_i = 0|\mathbf{v}_{<i}) = 1 - p(v_i = 1|\mathbf{v}_{<i})$$

To overcome the difficulty of NADE to capture relative relationships between inputs which is crucial when considering notes, we transfer the behavior of convolutional neural networks to RNNs. Given a vector of notes $\mathbf{v}^{(t)}$ at timestep t and a candidate vector of notes $\hat{\mathbf{v}}^{(t+1)}$ for the next timestep, we construct shifted vectors $\mathbf{w}^{(t)}$ and $\hat{\mathbf{w}}^{(t+1)}$ such that $\mathbf{w}_i^{(t)} = \mathbf{v}_{i+\delta}^{(t)}$ and $\hat{\mathbf{w}}_i^{(t)} = \hat{\mathbf{v}}_{i+\delta}^{(t)}$ Then, the output of the model satisfies

$$p(\hat{\mathbf{w}}^{(t+1)}|\mathbf{w}^{(t)}) = p(\hat{\mathbf{v}}^{(t+1)}|\mathbf{v}^{(t)}).$$

This achieves transposition invariance. After the processed data is passed through the LSTM layers, we use a dense layer to compute the play and the articulation probability for the next note using a sigmoid activation function.

Our data pre-processing followed a three-step process: First, we transform the midi files into a data frame. Midi files are made up of notes that each possess timestamps to indicate when they are played. We use these timestamps to create a data frame where each column is a 128-long one-hot-encoded vector equal to one beat in a 16-beat scheme. The 128 elements represent the notes that are played or not played at the given beat.

Second, we add the articulation to each note to differentiate between holding and articulating a note. To do this, we sample the music files at a higher frequency than the one of the beats and extract gaps and velocity changes between beats. Velocity changes indicate articulating or pausing a note.

Third, we replicate the note invariant data structure of the paper. This is represented in Figure 4. Each time step in the original data is split up into separate notes. In this way, we ensure translation invariance: if we shift the inputs up by some amount δ, the output will also be shifted by δ. Therefore, instead of feeding the network the original time step vector $\mathbf{v}^{(t)}$, we input note-specific vectors $\mathbf{u}^{(t)}$. This vector contains the pitch value of the note p to differentiate between high and low notes. As an example, this alone reduces the range of predicted notes dramatically, as we will see later. Next, it contains the note's one-hot encoded pitch class $\mathbf{c}^{(t)}$, where $\mathbf{c}^{(t)} = 1$ at $p \mod 12$. Then, the previous vicinity $\mathbf{w}^{(t)}$ of each note is included. Considering, that the original note vectors include information on both playing and articulating a note, we chose a vicinity of $50 = 2(12*2+1)$ (one octave has 12 semitones), to collect information one octave above and below the current note. Therefore $\mathbf{w}_i^{(n,t)} = \mathbf{v}_{p+i}^{(t)}$ where $-26 \leq i \leq 24$. If the previous vicinity exceeds past the bounds of $\mathbf{v}^{(t)}$, it is padded with 0, so that no notes outside the initial pitch range are played. Additionally, the previous context $\mathbf{z}_i^{(p,t)}$ is built (see chapter 2.1). The content of each context, is the number of notes that are played at the offset i from the current note

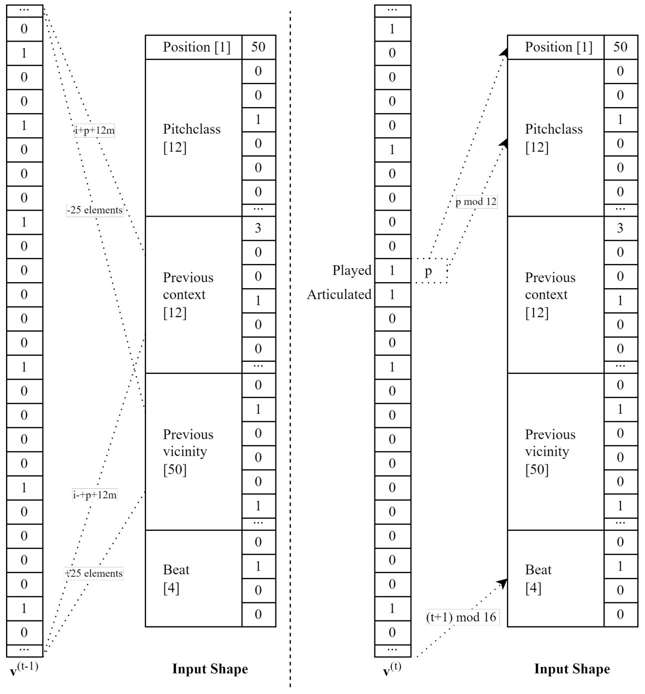

Figure 4. Illustration of the data pre-processing. On the left, data from the previous time step are collected for the input vector. Previous vicinity extracts a window around the current note and, while previous context extracts the counts of all played notes for each pitchclass. On the right, data from the current timestep, i.e. position, pitchclass, and beat is collected. Pitchclass is one-hot encoded while beats are encoded in binary.

across all octaves in the previous timesteps

$$\mathbf{z}_i^{(p,t)} = \sum_{m=-\infty}^{\infty} \mathbf{v}_{i+p+12m}^{(t-1)}$$

We train our network to model the conditional probability distribution of the notes played in a given timestep, conditioned on the notes played in previous timesteps just like the original paper. Letting $q^{(i,t)}$ represent our NN's estimate $p(v_i = 1|\mathbf{v}_{<i})$ Training our model minimizes the RMSE loss function

$$C_{rms} = \sqrt{\frac{\sum_i^N \sum_t^T (v_i^{(t)} - q^{(i,t)})^2}{n}}$$

where $v_i^t \in \{0,1\}$ is the label whether the note i is played at timestep t and $v_{i+1}^t \in \{0,1\}$ is whether the note i is articulated.

For music generation, we let the model predict a next timestep from a music sample in the dataset like in the original paper. Then this new timestep is added to the input sequence and the model predicts the next timestep now based on the initial sample and the previously predicted timestep. The LSTM time-axis layers are all advanced by one timestep. For each note. we choose which notes should be played and/or articulated $v_i^{(t)}$ from a Bernoulli distribution with probability $q^{(i,t)}$. All notes are processed for one timestep. Afterwards, the model starts predicting the next timestep in the same fashion.

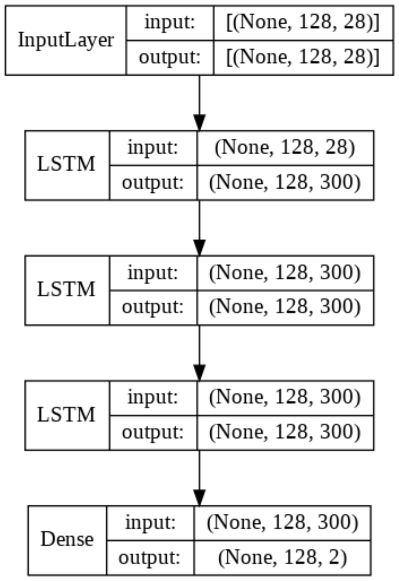

Figure 5. Architecture of the final RNN

4. Implementation

4.1. Data

We use the MAESTRO dataset (https://magenta.tensorflow.org/datasets/maestro), a dataset of midi files composed of about 200 hours of piano performances. The MIDI data includes key strike velocities. Audio and MIDI files are aligned with 3 ms accuracy and sliced to individual musical pieces, which are annotated with composer, title, and year of performance. Uncompressed audio is of CD quality or higher. We do not use the same dataset as the original papers as the MAESTRO dataset is more comprehensive and more easily accessible. However, we only consider classical piano pieces like the original paper.

For our model, we consider 20 random files from this dataset and convert them as discussed in chapter 3.2.

4.2. Deep Learning Network

As mentioned before, we use deep RNN with LSTM, and a dense layer with sigmoid activation, that is shown in Figure 5. We implement our model using TensorFlow (https://www.tensorflow.org/), an end-to-end open source platform for machine learning.

4.3. Software Design

Our software takes in a specified number of midi files and outputs a model that can predict, based on a given sequence of music, which notes are played in the next timestep. Our full software process is shown in Figure 6 and will be explained in detail in the following paragraphs.

(1+2) We convert midi files into a matrix of timesteps using the function `midi_obj_to_play_articulate()` represented by algorithm 1. We use the library prettyMIDI to convert each midi file to a python object. Then our function samples from the prettyMIDI object at a sampling frequency of 100 Hz. This is much higher than the frequency of the beats but is used to also detect velocity changes between the beats to extract the articulation. Github-Midi Support

Algorithm 1 Convert midi object to beat matrix with articulations

```
1:  Require: $midi$
2:  function $midiObjToPlayArticulate(midi)$
3:      $sf \leftarrow 100$
4:      $beats \leftarrow$ timestamps of all beats using $midi$.getBeats()
5:      $sample \leftarrow$ matrix from midi.getPianoRoll($sf$)
6:      $articulations \leftarrow$ locations of velocity changes
7:      $beatMatrix \leftarrow$ songToBeats($sample, beats$)
8:      $song \leftarrow$ intermix $beatMatrix$ and $articulations$
9:      return $song$
10: end function
```

(3) Next, we extract the previous vicinity of each note and reshape the data such that columns no longer represent timesteps but notes using the function `windowed_data_across_notes_time` represented by getWindows in algorithm 2. The relative of all timesteps remains the same and each note still has the information about its respective beat. This operation makes the data time invariant and implements the earlier mentioned convolution. The input is padded to play no notes outside the initial range of $(0, 128)$. It is the reshaped using the stride of 2 (because there are both articulation and played for each note) and the indexers. X is the output matrix of the shape $(vic, n * steps)$. $elements$ is the number of elements per time step, i.e. 128 notes, mainly used for indexing. Github-Midi Support

Algorithm 2 Convert song matrix to time invariant form

```
1:  Require: $song$: matrix with articulation, $vic$; size of
    previous vicinity
2:  function $getWindows(song, vic = 50)$
3:      $n, steps \leftarrow song.shape$
4:      $padding \leftarrow vic//2$
5:      $padded \leftarrow$ addPadding($song, padding$)
6:      $elements \leftarrow ((n + 2 * padding) + vic)//stride$
7:      $stride \leftarrow 2$
8:      $index_h \leftarrow$ horiz. index list for columns in new shape
9:      $index_v \leftarrow$ vert. index list for rows in new shape
10:     $indexer \leftarrow index_h + index_v$
11:     $X \leftarrow padded.flatten()[indexer]$
12:     return $X, elements$
13: end function
```

(4) Fourth, we extract the position, previous pitchclass, and context of each note using the functions `add_midi_value`, `calculate_pitchclass`, and `build_context` represented by $addMidiValue$, $calcPitchClass$, and $buildContext$ in algorithm 3, respectively. Notice that in $addMidiValue$ the 1 is added to every element. This is because the $song$ array is shifted by one along the note time axis so that vicinity and context can be calculated for each

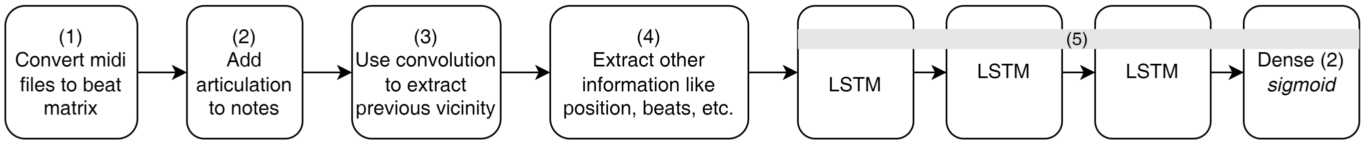
(1)
Convert midi files to beat matrix
(2)
Add articulation to notes
(3)
Use convolution to extract previous vicinity
(4)
Extract other information like position, beats, etc.
(5)
LSTM
LSTM
LSTM
Dense (2)
sigmoid

Figure 6. Flowchart of the training process

input vector without having to look back in time. Github-Midi Support

Algorithm 3 Convert song matrix to time invariant form

```
1: Require: $song$: matrix with articulation, $n$: number of notes
2: function $addMidiValue(song, n)$
3:     return $midiRow \leftarrow range(song.shape[1] + 1)\%n$
4: end function
5:
6: function $calcPitchClass(song, vic = 50, midiRow)$
7:     $pc = [[...]]$
8:     for $i$ in range(12) do
9:         for $j$ in $midiRow$ do
10:            if $i$ mod $12 == 0$ then
11:                $pc[i, j] == 1$
12:            end if
13:        end for
14:    end for
15:    return $pc$
16: end function
17:
18: function $buildContext(song)$
19:    Append column with pitchclass value to $song$
20:    $context$ ← grouped data by pitchclass and sum of individual counts
21:    return $context$
22: end function
```

(5) The reshaped input is passed sequentially into the TensorFlow model, that is presented in Figure 5. As optimizer, we use the TensorFlow-native Adam. Adam is particularly suited because the input data, being one-hot encoded and many notes never being played, is sparse. As loss function, we use RSME. During some experiments, where we predict whole timesteps at once, we use Categorical Crossentropy. While this works, we have found that transforming the input so that each note of a timespace is handled independently both achieves the desired translation invariance as well as a better performance. Github-Basic RNNs

5. Results

In this chapter, we will explain the results achieved both from our different experiments as well as our final model. These results mirror the experimentation discussed in the main jupyter notebook in the corresponding repo.

5.1. Non note-invariant network

Before directly implementing all details of the original author's paper, we started by implementing a model that did not achieve note invariance. To accomplish this we started by taking all piano keys as input into the model and tried to predict all outputs. This proved challenging and we were not able to train the model to achieve anything meaningful.

We experimented slightly further with this non note-invariant model to achieve the note vicinity described by the original author. We accomplished this by implementing a custom kernel regularizer which penalized LSTM kernel weights between note that were far apart so that only notes close together saw eachother's input. The loss for training these functions is included in the final table below.

5.2. Number of Beats

We ran an experiment on the number of piano time steps or "beats" used to predict the next note. Figure 7 shows these results while predicting using 1 beat, 3 beats, 15 beats, and 31 beats in that order. It can be seen that the single beat appears to be the most random. This makes sense as the neural network is not able to retain useful musical memory in it's internal state. The network using 31 beats to predict the 32 shows the best performance. It should be noted that in each of these cases the number of input sequences to the network is 128 times the number of beats described her. 1 beat in time is input over 128 sequences to represent all of the notes.

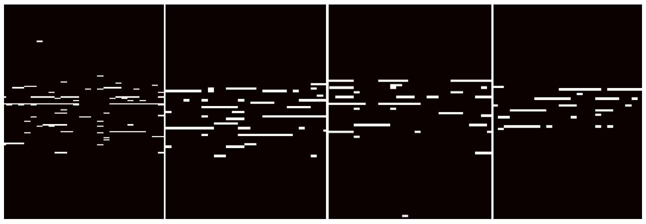

Figure 7. Close-up of the notes played by a model with input of 1 beat, 3 beats, 15 beats, and 31 (from left to right).

5.3. Batch Size

As shown in Figure 8, we found that training with a larger batch size results in the model not properly learning. This could be because we are inputting different piano notes across the batch dimension. If we group many notes together the patterns disappear and get entangled. For example, training on information from C and the adjacent C# note would suggest that C and C# would be played together. However, we know from music theory that playing to directly adjacent notes are not complementary. In fact this is referred to dissonant notes and results in bad sounding songs.

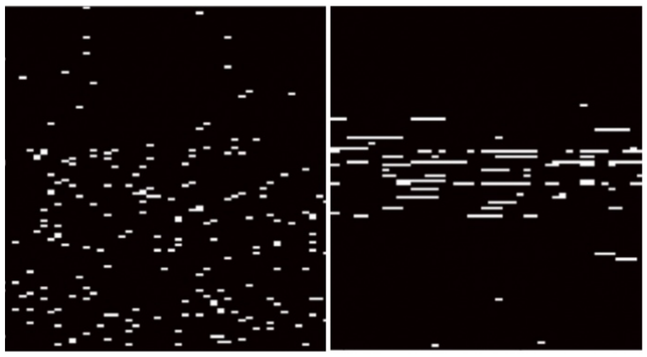
Figure 8. Close-up of the notes played by a model with a batch size 50 (left) and batch size 1 (right).

5.4. Note Vicinity

Note vicinity refers to how many adjacent notes are used to predict the following note. For example, with a single octave vicinity, middle C note uses one octave above and one octave below to predict whether in the next time step, middle C should be played or not. In his paper, Johnson uses 2 octaves above and 2 octaves below. We wanted to experiment with this values to understand how it affects the model. In Figure 9, we can see noticeable differences between the models. The model using only 1 octave above and below tends to play notes closer together on the piano keys. This makes sense because when the highest keys on the piano are determining whether they should be played or not, only the models with a larger vicinity will have a view into the notes on the middle of the piano roll. The models with a small vicinity do not see any notes being played and so they never predict more notes being played.

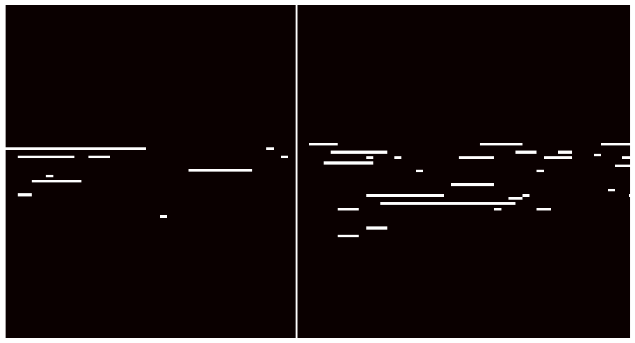
Figure 9. Close-up of using a one-octave vicinity (left) and a two-octave vicinity (right).

5.5. Capacity of the Model

Given we are trying to learn complex sequences it is natural to experiment with the number of hidden layers in our LSTM model. First, we notice that increasing the number of hidden layers in the LSTM makes the model harder to train. We compensate for this by decreasing the learning rate and increasing the number of training epochs. There is not a clear result as to whether more or less capacity is needed. In this experiment we are only training on a single song. This might be the reason the result is not as pronounced here. As we train on more songs, it is possible that the number of hidden nodes becomes more important as we need the neural network to recognize complex relationships that appear across many songs.

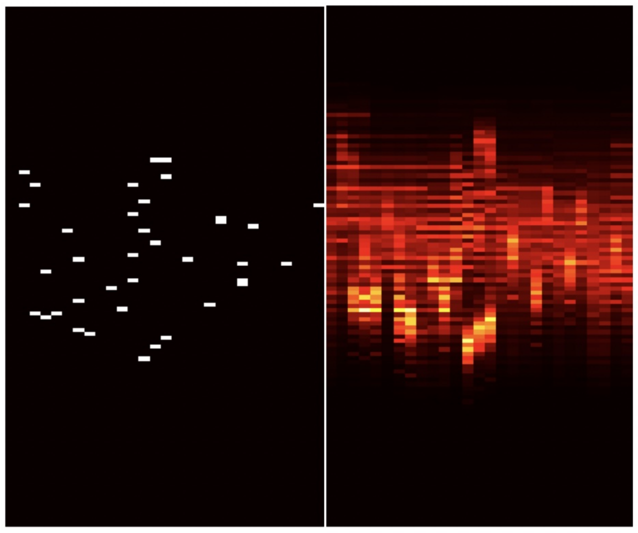
Figure 10. Close up of predicted notes from a model (left) and the associated note prediction probability (right).

5.6. Final Model

Figure 10 shows a portion of a predicted song. This model was trained using 10 songs. It has 4 layers of 300 hidden LSTM nodes. Prediction was done using a sequence of 63 notes. The color intensity shows which notes have the highest probability of being predicted next. An interesting result is shown in the center lower part of the graph where there is a sequence of 3 increasing notes. The trained model know that the next following notes should be close and likely increasing. Furthermore, we can see that the model is likely predicting the following notes in the same key as it increases by whole steps rather than half steps.

A recording output from our best results can be found at this YouTube-Link (https://youtu.be/CF0Wi2WEQOU).

5.7. Comparison of the Results Between the Original Paper and Our Project

In this experimentation we closely follow the original author's ideas but with some noticeable differences. The most significant difference is our model is not designed to be time invariant. To achieve time invariance, the original author transposed the piano note and piano time dimensions for two of their LSTM layers. In doing the the time dimension of piano was trained in the batch dimension of the LSTM. Doing this allowed the network to learn all possibilities of what the next notes should be regardless of the point in the prediction sequence. Besides this aspect, our model implemented many aspects of the original author's.

Our most complete model is the closest to the original authors paper. In this model we implemented the 2-octave vicinity, previous context, pitch class, midi value, and beat. We trained on 10 classical piano pieces for one epoch with a batch size of 1. To predict notes, we gave the model 63 previous beats in time and had the model predict the next beat. This model showed promising results and output piano pieces that song sound like real songs. We recognize the quality of the piano pieces are not of the same quality that an expert pianist would output.

As we use different loss functions and have a wider pitch range than Johnson, we cannot sensibly compare the loss minimization performance of his model with ours. However, we report our loss results in Figure 11. Additionally, as Johnson highlights himself in his paper for the task of music generation it is more important to evaluate difference by listening to the music instead of judging it by some form of numeric value. Therefore, we provide some of his generated music pieces and one of ours at the following links.
Our model: https://youtu.be/CF0Wi2WEQOU
Original model: https://www.danieldjohnson.com/2015/08/03/composing-music-with-recurrent-neural-networks/

Johnson's music extracts more patterns, chords, and logical musical progressions like scales. Additionally, his model plays fewer off-beats and notes that sound inharmonious with what has been played before. Nevertheless, his model falls into unnatural, long breaks in the middle of a piece or repeats the same notes unusually many times. Our model, in comparison makes more mistakes and sometimes plays notes off-beat. We can hear that some common musical patterns are extracted, especially upward-sloping fast note progressions. These differences are expected and we believe that we could achieve great improvement by implementing the time invariant property of the original model. This would make it easier for our model to learn to play common chords.

6. Future Work

6.1. Improving the Quality of Our Network

Given the time constraints of this project we were only able to train our models on a small number of songs. It is likely that training on more music pieces would give improved performance. Furthermore, it is likely that training on simple piano pieces in the beginning could help the model learn simple relationships before tackling more complicated songs. The quality of our generated music should be improved in the future. The ultimate goal for generated music should be to be as good as or better than human-created music. Johnson's paper and this work have shown by considering data characteristics specific to our chosen domain, music, the output quality can be improved. Therefore, the new music-specific NNs should be developed. This includes using more training data, further tweaking the model's parameters and architecture as well as employing other methods of data processing. As it was not possible for us to implement the time invariance of the original paper given the short time frame of this project, future research should also investigate how the original architecture can be fully replicated and potentially improved using modern frameworks like TensorFlow. This will enable researchers to understand how technical differences between the original code and new technologies affect the model's performance and capability to generate music.

6.2. Extending the Model

Based on our work on generating classical piano music using LSTM, we would like to see deep learning based music generation for different genres and instruments. Particularly, the architecture we present to generate as harmonic piano music as possible works best on music with a 4/4 beat and cannot predict for example different instruments or volumes. Therefore, we believe that trying to further generalize music generation to encompass the wide variety of possibilities holds great potential to understand computational art.

7. Conclusion

In this paper we outlined the generation of our music generating recursive neural network based on the paper "Generating Polyphonic Music Using Tied Parallel Networks" by Daniel D. Johnson Johnson, 2017. We modify the original architecture by removing the bi-axial characteristic and instead focus on time invariance, different types of input data, and tuning of the different parameters.

The modified architecture still divides the the music generation and prediciton task such that each network instance is responsible for only a single note and receives input relative to each note, a structure inspired by convolutional networks.

Our experiments have shown numerous insights. First, feeding a network more beats enhances it predictive performance and generation of musical patterns. Second, smaller batch sizes, contrary to common practice, can support the generalization process in our model as they avoid weight interactions. Third, including note vicinity as suggested by the original paper has a dramatic impact on the note selection during prediction. With higher vicinity comes more interaction between the notes and the range of played notes is effectively limited. Fourth, the capacity of the model is not crucial in our experiments and LSTM can achieve good results without the need to very deep or wide neural networks.

Even though we could identify interesting characteristics of deep learning in music generation, our model performs worse than the original. This is likely because our model misses the time invariance of the original model. Future work will explore further generalizations of music generation such as different beat structures, or different music styles.

8. Acknowledgement

We would like to acknowledge the author of the original paper, Daniel Johson, for the detailed information on model implemendation. We acknowledge the TensorFlow Tutorial on generating music using an RNN which helped us to initially understand how to structure our model.

References

Chen, C.-C., & Miikkulainen, R. (2001). Creating melodies with evolving recurrent neural networks. *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, *3*, 2241–2246 vol.3.

Eck, D., & Schmidhuber, J. (2002). A first look at music composition using lstm recurrent neural networks.

Hrnjica, B., & Bonacci, O. (2019). Lake level prediction using feed forward and recurrent neural networks. *Water Resources Management*, 1–14. https://doi.org/10.1007/s11269-019-02255-2

Johnson, D. D. (2017). Generating polyphonic music using tied parallel networks. In J. Correia, V. Ciesielski, & A. Liapis (Eds.), *Computational intelligence in music, sound, art and design* (pp. 128–143). Springer International Publishing.

Kaiser, Ł., & Sutskever, I. (2016). Neural gpus learn algorithms.

Waite, E., Eck, D., Roberts, A., & Abolafia, D. (2016). Project magenta: Generating long-term structure in songs and stories.

9. Appendix

9.1. Individual Student Contributions in Fractions

	tjg2148	jjw2196
Last Name	Gordon	Wiedenhaupt
Fraction of (useful) total contribution	1/2	1/2
What I did 1	Setup major architecture and experiment structure	Preprocessing of data into invariant form
What I did 2	Development of experiments 1-4	Development of experiment 5
What I did 3	Wrote chapter 5	Wrote chapters 1-4, 6-8

9.2. Experiment Report

Experiment	**Learning rate**	**Epochs**	**Batch size**	**Files**	**Vicinity**	**Training sequence length**	**Prediction sequence length**	**Number of hidden nodes**	**Loss**	**Loss Function**
Multi Input Multi Output	0.0001	3	64	2	128	15	15	256	17.2644	Crossentropy
Limited Connectivity Multi Input Multi Output	0.001	5	1	1	8	15	15	50	37.4899	Crossentropy
Note invariant LSTM	0.01	3	1	1	24	1 * 128	31 * 128	50	0.0256	RMSError
Analysis of number of beats used for prediction	0.01	3	1	1	24	1 * 128	31 * 128	50	0.0390	RMSError
Analysis of number of beats used for prediction	0.01	3	1	1	24	1 * 128	15 * 128	50	0.0390	RMSError
Analysis of number of beats used for prediction	0.01	3	1	1	24	1 * 128	3 * 128	50	0.0390	RMSError
Study on Batch Size	0.01	3	1	1	24	1 * 128	31 * 128	50	0.0395	RMSError
Study on Batch Size	0.01	3	50	1	24	1 * 128	31 * 128	50	0.0850	RMSError
Study on Note Vicinity	0.01	3	1	1	24	1 * 128	31 * 128	50	0.0397	RMSError
Study on Note Vicinity	0.01	3	1	1	48	1 * 128	31 * 128	50	0.0391	RMSError
Study on increasing the capacity of the model	0.01	5	1	1	48	1 * 128	31 * 128	50	0.0404	RMSError
Study on increasing the capacity of the model	0.01	8	1	1	48	1 * 128	31 * 128	100	0.0411	RMSError
Study on increasing the capacity of the model	0.005	20	1	1	48	1 * 128	31 * 128	300	0.0733	RMSError
Final	**0.0001**	**20**	**1**	**20**	**48**	**1 * 128**	**63 * 128**	**300**	**0.0550**	**RMSError**