CHAPTER 11

Music Generation

Chapter Goals

In this chapter you will:

- Understand how we can treat music generation as a sequence prediction problem, so we can apply autoregressive models such as Transformers.
- See how to parse and tokenize MIDI files using the `music21` package to create a training set.
- Learn how to use sine positional encoding.
- Train a music-generating Transformer, with multiple inputs and outputs to handle note and duration.
- Understand how to handle polyphonic music, including grid tokenization and event-based tokenization.
- Train a MuseGAN model to generate multitrack music.
- Use the MuseGAN to adjust different properties of the generated bars.

Musical composition is a complex and creative process that involves combining different musical elements such as melody, harmony, rhythm, and timbre. While this is traditionally seen as a uniquely human activity, recent advancements have made it possible to generate music that both is pleasing to the ear and has long-term structure.

One of the most popular techniques for music generation is the Transformer, as music can be thought of as a sequence prediction problem. These models have been adapted to generate music by treating musical notes as a sequence of tokens, similar to words in a sentence. The Transformer model learns to predict the next note in the sequence based on the previous notes, resulting in a generated piece of music.

MuseGAN takes a totally different approach to generating music. Unlike Transformers, which generate music note by note, MuseGAN generates entire musical tracks at once by treating music as an *image*, consisting of a pitch axis and a time axis. Moreover, MuseGAN separates out different musical components such as chords, style, melody, and groove so that they can be controlled independently.

In this chapter we will learn how to process music data and apply both a Transformer and MuseGAN to generate music that is stylistically similar to a given training set.

Introduction

For a machine to compose music that is pleasing to our ear, it must master many of the same technical challenges that we saw in Chapter 9 in relation to text. In particular, our model must be able to learn from and re-create the sequential structure of music and be able to choose from a discrete set of possibilities for subsequent notes.

However, music generation presents additional challenges that are not present for text generation, namely pitch and rhythm. Music is often polyphonic—that is, there are several streams of notes played simultaneously on different instruments, which combine to create harmonies that are either dissonant (clashing) or consonant (harmonious). Text generation only requires us to handle a single stream of text, in contrast to the parallel streams of chords that are present in music.

Also, text generation can be handled one word at a time. Unlike text data, music is a multipart, interwoven tapestry of sounds that are not necessarily delivered at the same time—much of the interest that stems from listening to music is in the interplay between different rhythms across the ensemble. For example, a guitarist might play a flurry of quicker notes while the pianist holds a longer sustained chord. Therefore, generating music note by note is complex, because we often do not want all the instruments to change notes simultaneously.

We will start this chapter by simplifying the problem to focus on music generation for a single (monophonic) line of music. Many of the techniques from Chapter 9 for text generation can also be used for music generation, as the two tasks share many common themes. We will start by training a Transformer to generate music in the style of the J.S. Bach cello suites and see how the attention mechanism allows the model to focus on previous notes in order to determine the most natural subsequent note. We'll then tackle the task of polyphonic music generation and explore how we can deploy an architecture based around GANs to create music for multiple voices.

Transformers for Music Generation

The model we will be building here is a decoder Transformer, taking inspiration from OpenAI's *MuseNet*, which also utilizes a decoder Transformer (similar to GPT-3) trained to predict the next note given a sequence of previous notes.

In music generation tasks, the length of the sequence N grows large as the music progresses, and this means that the $N \times N$ attention matrix for each head becomes expensive to store and compute. We ideally do not want to clip the input sequence to a short number of tokens, as we would like the model to construct the piece around a long-term structure and repeat motifs and phrases from several minutes ago, as a human composer would.

To tackle this problem, MuseNet utilizes a form of Transformer known as a *Sparse Transformer*. Each output position in the attention matrix only computes weights for a subset of input positions, thereby reducing the computational complexity and memory required to train the model. MuseNet can therefore operate with full attention over 4,096 tokens and can learn long-term structure and melodic structure across a range of styles. (See, for example, OpenAI's Chopin and Mozart recordings on SoundCloud.)

To see how the continuation of a musical phrase is often influenced by notes from several bars ago, take a look at the opening bars of the Prelude to Bach's Cello Suite No. 1 (Figure 11-1).

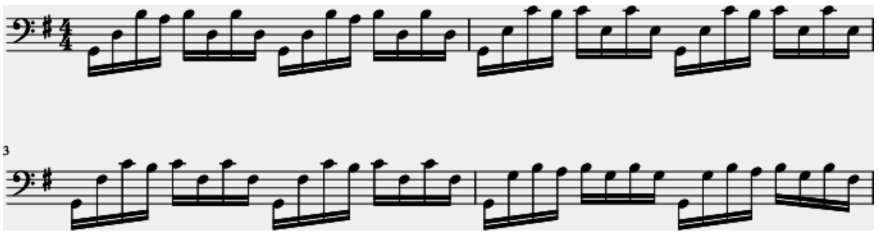


Figure 11-1. The opening of Bach's Cello Suite No. 1 (Prelude)

Bars

Bars (or *measures*) are small units of music that contain a fixed, small number of beats and are marked out by vertical lines that cross the staff. If you can count 1, 2, 1, 2 along to a piece of music, then there are two beats in each bar and you're probably listening to a march. If you can count 1, 2, 3, 1, 2, 3, then there are three beats to each bar and you may be listening to a waltz.

What note do you think comes next? Even if you have no musical training you may still be able to guess. If you said G (the same as the very first note of the piece), then you'd be correct. How did you know this? You may have been able to see that every bar and half bar starts with the same note and used this information to inform your decision. We want our model to be able to perform the same trick—in particular, we want it to pay attention to a particular note from the previous half bar, when the previous low G was registered. An attention-based model such as a Transformer will be able to incorporate this long-term look-back without having to maintain a hidden state across many bars, as is the case with a recurrent neural network.

Anyone tackling the task of music generation must first have a basic understanding of musical theory. In the next section we'll go through the essential knowledge required to read music and how we can represent this numerically, in order to transform music into the input data required to train our Transformer.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/11_music/01_transformer/transformer.ipynb* in the book repository.

The Bach Cello Suite Dataset

The raw dataset that we shall be using is a set of MIDI files for the Cello Suites by J.S. Bach. You can download the dataset by running the dataset downloader script in the book repository, as shown in Example 11-1. This will save the MIDI files locally to the */data* folder.

Example 11-1. Downloading the J.S. Bach Cello Suites dataset

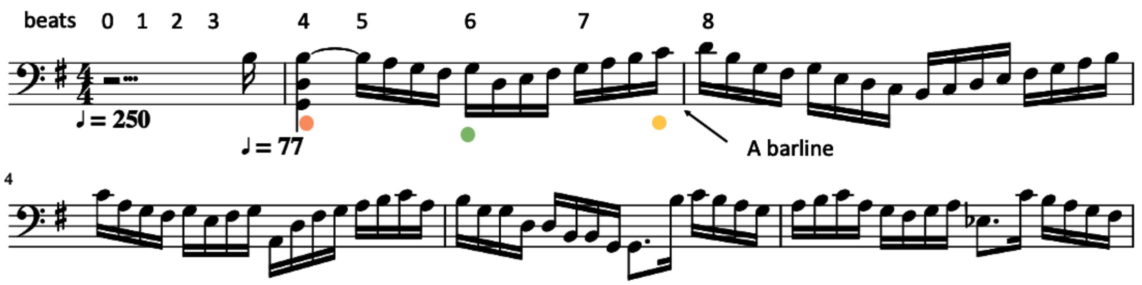
```
bash scripts/download_music_data.sh
```

To view and listen to the music generated by the model, you'll need some software that can produce musical notation. MuseScore is a great tool for this purpose and can be downloaded for free.

Parsing MIDI Files

We'll be using the Python library `music21` to load the MIDI files into Python for processing. Example 11-2 shows how to load a MIDI file and visualize it (Figure 11-2), both as a score and as structured data.

```
example_score.show()
```



```
example_score.show("text")
```

```
{0.0} <music21.metadata.Metadata object at 0xffff2b150b20>
{0.0} <music21.stream.Measure 1 offset=0.0>
    {0.0} <music21.instrument.Violoncello 'Solo Cello: Solo
    {0.0} <music21.instrument.Violoncello 'Violoncello'>
    {0.0} <music21.clef.BassClef>
    {0.0} <music21.tempo.MetronomeMark Quarter=250.0>
    {0.0} <music21.key.Key of G major>
    {0.0} <music21.meter.TimeSignature 4/4>
    {0.0} <music21.note.Rest 3.75ql>
    {3.5} <music21.tempo.MetronomeMark Quarter=77.0>
    {3.75} <music21.chord.Chord B3>
{4.0} <music21.stream.Measure 2 offset=4.0>
  ● {0.0} <music21.chord.Chord G2 D3 B3>
    {1.0} <music21.chord.Chord B3>
    {1.25} <music21.chord.Chord A3>
    {1.5} <music21.chord.Chord G3>
    {1.75} <music21.chord.Chord F#3>
  ● {2.0} <music21.chord.Chord G3>
    {2.25} <music21.chord.Chord D3>
    {2.5} <music21.chord.Chord E3>
    {2.75} <music21.chord.Chord F#3>
    {3.0} <music21.chord.Chord G3>
    {3.25} <music21.chord.Chord A3>
    {3.5} <music21.chord.Chord B3>
  ● {3.75} <music21.chord.Chord C4>
```

The midi file starts with some metadata around the instrumentation, tempo, key, and time signature of the piece.

This note starts on beat 4 of the piece (zero-indexed). It has duration of 1 beat (as the next note starts on beat 5) and consists of a chord of low G, D, and B.

This note starts on beat 6 of the piece. It has duration of a quarter of a beat (as the next note starts on beat 6.25) and consists of a single note – G.

This note starts a quarter of a beat before beat 8 of the piece. It has duration of a quarter of a beat and consists of a single note – high C.

Figure 11-2. Musical notation

Example 11-2. Importing a MIDI file

```
import music21

file = "/app/data/bach-cello/cs1-2all.mid"
example_score = music21.converter.parse(file).chordify()
```

Octaves

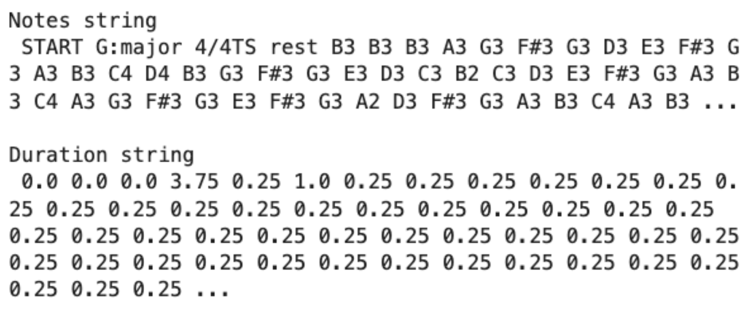
The number after each note name indicates the *octave* that the note is in—since the note names (A to G) repeat, this is needed to uniquely identify the pitch of the note. For example, `G2` is an octave below `G3`.

Now it's time to convert the scores into something that looks more like text! We start by looping over each score and extracting the note and duration of each element in the piece into two separate text strings, with elements separated by spaces. We encode the key and time signature of the piece as special symbols, with zero duration.

Monophonic Versus Polyphonic Music

In this first example, we will treat the music as *monophonic* (one single line), taking just the top note of any chords. Sometimes we may wish to keep the parts separate to generate music that is *polyphonic* in nature. This presents additional challenges that we shall tackle later on in this chapter.

The output from this process is shown in Figure 11-3—compare this to Figure 11-2 so that you can see how the raw music data has been transformed into the two strings.



```
Notes string
 START G:major 4/4TS rest B3 B3 B3 A3 G3 F#3 G3 D3 E3 F#3 G
3 A3 B3 C4 D4 B3 G3 F#3 G3 E3 D3 C3 B2 C3 D3 E3 F#3 G3 A3 B
3 C4 A3 G3 F#3 G3 E3 F#3 G3 A2 D3 F#3 G3 A3 B3 C4 A3 B3 ...

Duration string
 0.0 0.0 0.0 3.75 0.25 1.0 0.25 0.25 0.25 0.25 0.25 0.25 0.
25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.25 0.25 0.25 ...
```

Figure 11-3. Samples of the notes text string and the duration text string, corresponding to Figure 11-2

This looks a lot more like the text data that we have dealt with previously. The *words* are the note–duration combinations, and we should try to build a model that predicts the next note and duration, given a sequence of previous notes and durations. A key difference between music and text generation is that we need to build a model that can handle the note and duration prediction simultaneously—i.e., there are two streams of information that we need to handle, compared to the single streams of text that we saw in Chapter 9.

Tokenization

To create the dataset that will train the model, we first need to tokenize each note and duration, exactly as we did previously for each word in a text corpus. We can achieve this by using a `TextVectorization` layer, applied to the notes and durations separately, as shown in Example 11-3.

Example 11-3. Tokenizing the notes and durations

```
def create_dataset(elements):
    ds = (
        tf.data.Dataset.from_tensor_slices(elements)
        .batch(BATCH_SIZE, drop_remainder = True)
        .shuffle(1000)
    )
    vectorize_layer = layers.TextVectorization(
        standardize = None, output_mode="int"
    )
    vectorize_layer.adapt(ds)
    vocab = vectorize_layer.get_vocabulary()
    return ds, vectorize_layer, vocab

notes_seq_ds, notes_vectorize_layer, notes_vocab = create_dataset(notes)
durations_seq_ds, durations_vectorize_layer, durations_vocab = create_dataset(
    durations
)
seq_ds = tf.data.Dataset.zip((notes_seq_ds, durations_seq_ds))
```

The full parsing and tokenization process is shown in Figure 11-4.

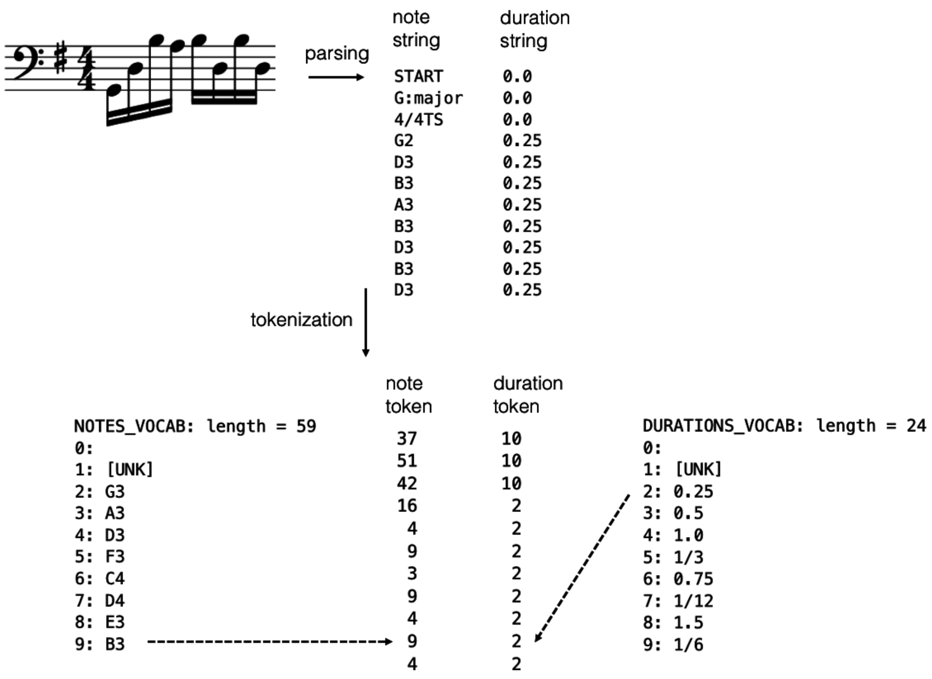


Figure 11-4. Parsing the MIDI files and tokenizing the notes and durations

Creating the Training Set

The final step of preprocessing is to create the training set that we will feed to our Transformer.

We do this by splitting both the note and duration strings into chunks of 50 elements, using a sliding window technique. The output is simply the input window shifted by one note, so that the Transformer is trained to predict the note and duration of the element one timestep into the future, given previous elements in the window. An example of this (using a sliding window of only four elements for demonstration purposes) is shown in Figure 11-5.

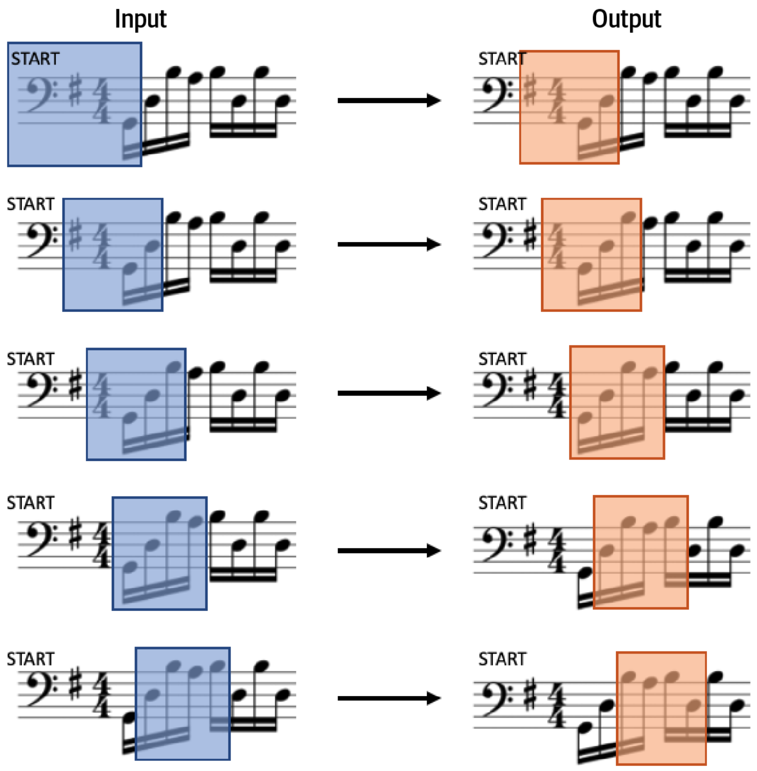


Figure 11-5. The inputs and outputs for the musical Transformer model—in this example, a sliding window of width 4 is used to create input chunks, which are then shifted by one element to create the target output

The architecture we will be using for our Transformer is the same as we used for text generation in Chapter 9, with a few key differences.

Sine Position Encoding

Firstly, we will be introducing a different type of encoding for the token positions. In Chapter 9 we used a simple `Embedding` layer to encode the position of each token, effectively mapping each integer position to a distinct vector that was learned by the model. We therefore needed to define a maximum length (*N*) that the sequence could be and train on this length of sequence. The downside to this approach is that it is then impossible to extrapolate to sequences that are longer than this maximum length. You would have to clip the input to the last *N* tokens, which isn't ideal if you are trying to generate long-form content.

To circumvent this problem, we can switch to using a different type of embedding called a *sine position embedding*. This is similar to the embedding that we used in Chapter 8 to encode the noise variances of the diffusion model. Specifically, the following function is used to convert the position of the word (*pos*) in the input sequence into a unique vector of length *d*:

$$PE_{pos,2i} = \sin\left(\frac{pos}{10,000^{2i/d}}\right)$$

$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10,000^{(2i+1)/d}}\right)$$

For small *i*, the wavelength of this function is short and therefore the function value changes rapidly along the position axis. Larger values of *i* create a longer wavelength. Each position thus has its own unique encoding, which is a specific combination of the different wavelengths.

Notice that this embedding is defined for all possible position values. It is a deterministic function (i.e., it isn't learned by the model) that uses trigonometric functions to define a unique encoding for each possible position.

The *Keras NLP* module has a built-in layer that implements this embedding for us—we can therefore define our `TokenAndPositionEmbedding` layer as shown in Example 11-4.

Example 11-4. Tokenizing the notes and durations

```
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.pos_emb = keras_nlp.layers.SinePositionEncoding()

    def call(self, x):
        embedding = self.token_emb(x)
        positions = self.pos_emb(embedding)
        return embedding + positions
```

Figure 11-6 shows how the two embeddings (token and position) are added to produce the overall embedding for the sequence.

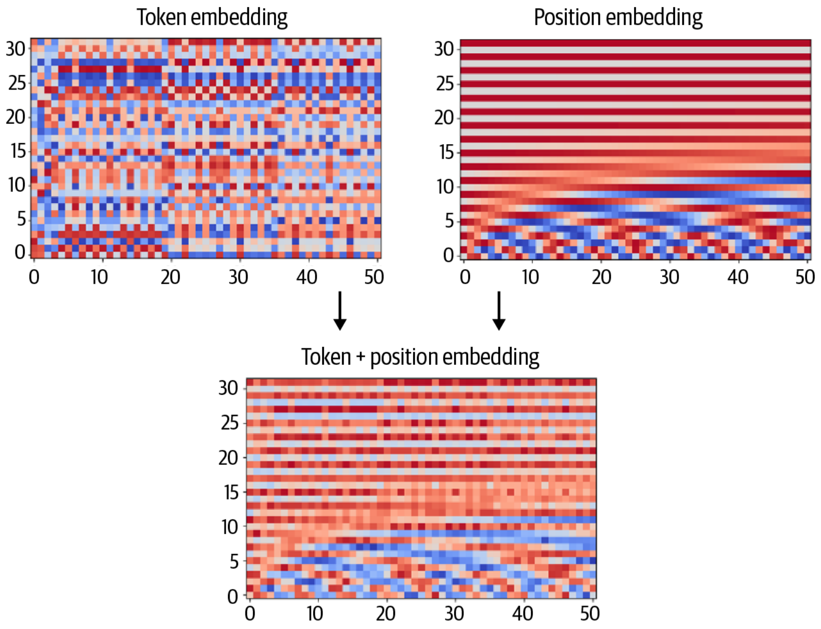


Figure 11-6. The `TokenAndPositionEmbedding` layer adds the token embeddings to the sinusoidal position embeddings to produce the overall embedding for the sequence

Multiple Inputs and Outputs

We now have two input streams (notes and durations) and two output streams (predicted notes and durations). We therefore need to adapt the architecture of our Transformer to cater for this.

There are many ways of handling the dual stream of inputs. We could create tokens that represent each note–duration pair and then treat the sequence as a single stream of tokens. However, this has the downside of not being able to represent note–duration pairs that have not been seen in the training set (for example, we may have seen a `G#2` note and a `1/3` duration independently, but never together, so there would be no token for `G#2:1/3`.

Instead, we choose to embed the note and duration tokens separately and then use a concatenation layer to create a single representation of the input that can be used by the downstream Transformer block. Similarly, the output from the Transformer block is passed to two separate dense layers, which represent the predicted note and duration probabilities. The overall architecture is shown in Figure 11-7. Layer output shapes are shown with batch size `b` and sequence length `l`.

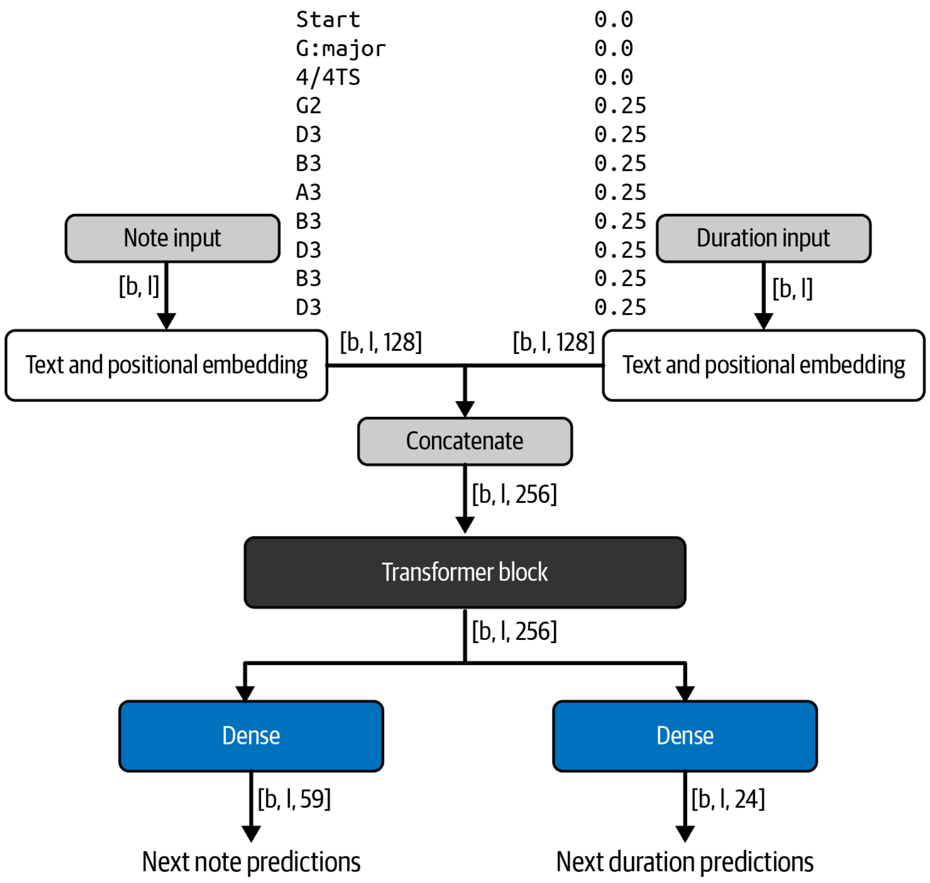


Figure 11-7. The architecture of the music-generating Transformer

An alternative approach would be to interleave the note and duration tokens into a single stream of input and let the model learn that the output should be a single stream where the note and duration tokens alternate. This comes with the added complexity of ensuring that the output can still be parsed when the model has not yet learned how to interleave the tokens correctly.

There is no *right* or *wrong* way to design your model—part of the fun is experimenting with different setups and seeing which works best for you!

Analysis of the Music-Generating Transformer

We'll start by generating some music from scratch, by seeding the network with a `START` note token and `0.0` duration token (i.e., we are telling the model to assume it is starting from the beginning of the piece). Then we can generate a musical passage using the same iterative technique we used in Chapter 9 for generating text sequences, as follows:

1. Given the current sequence (of notes and durations), the model predicts two distributions, one for the next note and one for the next duration.
2. We sample from both of these distributions, using a `temperature` parameter to control how much variation we would like in the sampling process.
3. The chosen note and duration are appended to the respective input sequences.
4. The process repeats with the new input sequences for as many elements as we wish to generate.

Figure 11-8 shows examples of music generated from scratch by the model at various epochs of the training process. We use a temperature of 0.5 for the notes and durations.

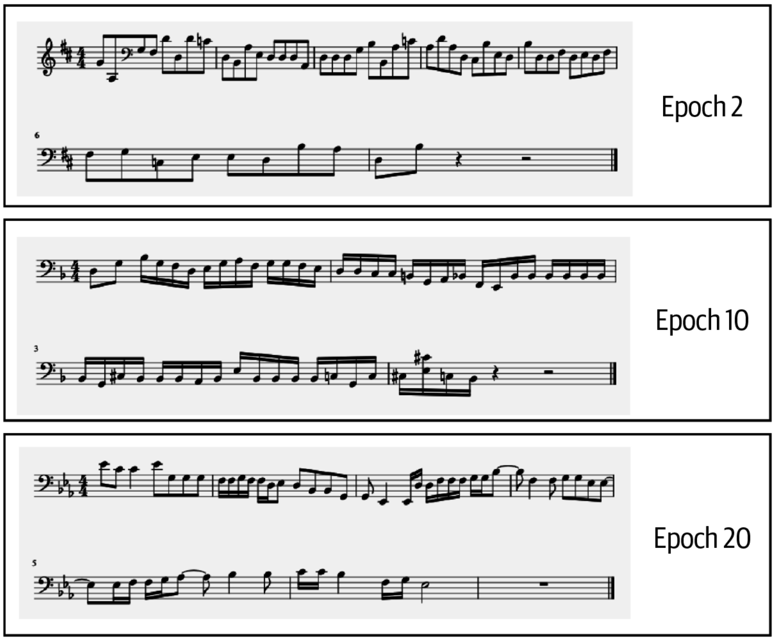


Figure 11-8. Some examples of passages generated by the model when seeded only with a `START` note token and `0.0` duration token

Most of our analysis in this section will focus on the note predictions, rather than durations, as for Bach's Cello Suites the harmonic intricacies are more difficult to capture and therefore more worthy of investigation. However, you can also apply the same analysis to the rhythmic predictions of the model, which may be particularly relevant for other styles of music that you could use to train this model (such as a drum track).

There are several points to note about the generated passages in Figure 11-8. First, see how the music is becoming more sophisticated as training progresses. To begin with, the model plays it safe by sticking to the same group of notes and rhythms. By epoch 10, the model has begun to generate small runs of notes, and by epoch 20 it is producing interesting rhythms and is firmly established in a set key (E ♭ major).

Second, we can analyze the distribution of notes over time by plotting the predicted distribution at each timestep as a heatmap. Figure 11-9 shows this heatmap for the example from epoch 20 in Figure 11-8.

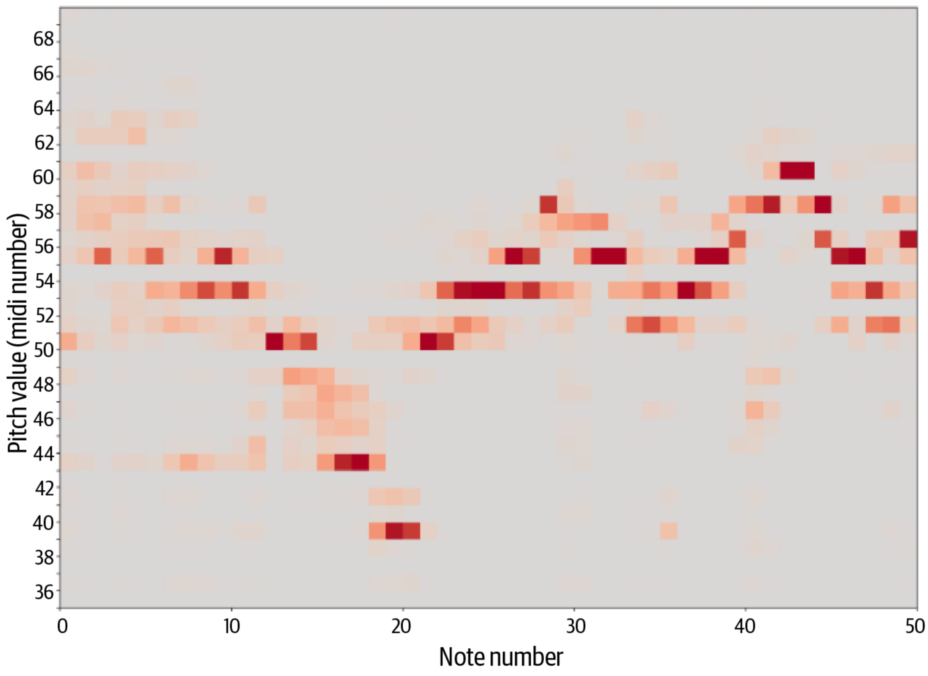


Figure 11-9. The distribution of possible next notes over time (at epoch 20): the darker the square, the more certain the model is that the next note is at this pitch

An interesting point to note here is that the model has clearly learned which notes belong to particular *keys*, as there are gaps in the distribution at notes that do not belong to the key. For example, there is a gray gap along the row for note 54 (corresponding to G ♭/F ♯). This note is highly unlikely to appear in a piece of music in the key of E ♭ major. The model establishes the key early on in the generation process, and as the piece progresses, the model chooses notes that are more likely to feature in that key by attending to the token that represents it.

It is also worth pointing out that the model has learned Bach's characteristic style of dropping to a low note on the cello to end a phrase and bouncing back up again to start the next. See how around note 20, the phrase ends on a low E ♭—it is common in the Bach Cello Suites to then return to a higher, more sonorous range of the instrument for the start of next phrase, which is exactly what the model predicts. There is a large gray gap between the low E ♭ (pitch number 39) and the next note, which is predicted to be around pitch number 50, rather than continuing to rumble around the depths of the instrument.

Lastly, we should check to see if our attention mechanism is working as expected. The horizontal axis in Figure 11-10 shows the generated sequence of notes; the vertical axis shows where the attention of the network was aimed when predicting each note along the horizontal axis. The color of each square shows the maximum attention weight across all heads at each point in the generated sequence. The darker the square, the more attention is being applied to this position in the sequence. For simplicity, we only show the notes in this diagram, but the durations of each note are also being attended to by the network.

We can see that for the initial key signature, time signature, and rest, the network chose to place almost all of its attention on the `START` token. This makes sense, as these artifacts always appear at the start of a piece of music—once the notes start flowing the `START` token essentially stops being attended to.

As we move beyond the initial few notes, we can see that the network places most attention on approximately the last two to four notes and rarely places significant weight on notes more than four notes ago. Again, this makes sense; there is probably enough information contained in the previous four notes to understand how the phrase might continue. Additionally, some notes attend more strongly back to the key signature of D minor—for example, the `E3` (7th note of the piece) and `B-2` (B ♭–14th note of the piece). This is fascinating, because these are the exact notes that rely on the key of D minor to relieve any ambiguity. The network must *look back* at the key signature in order to tell that there is a B ♭ in the key signature (rather than a B natural) but there isn't an E ♭ in the key signature (E natural must be used instead).

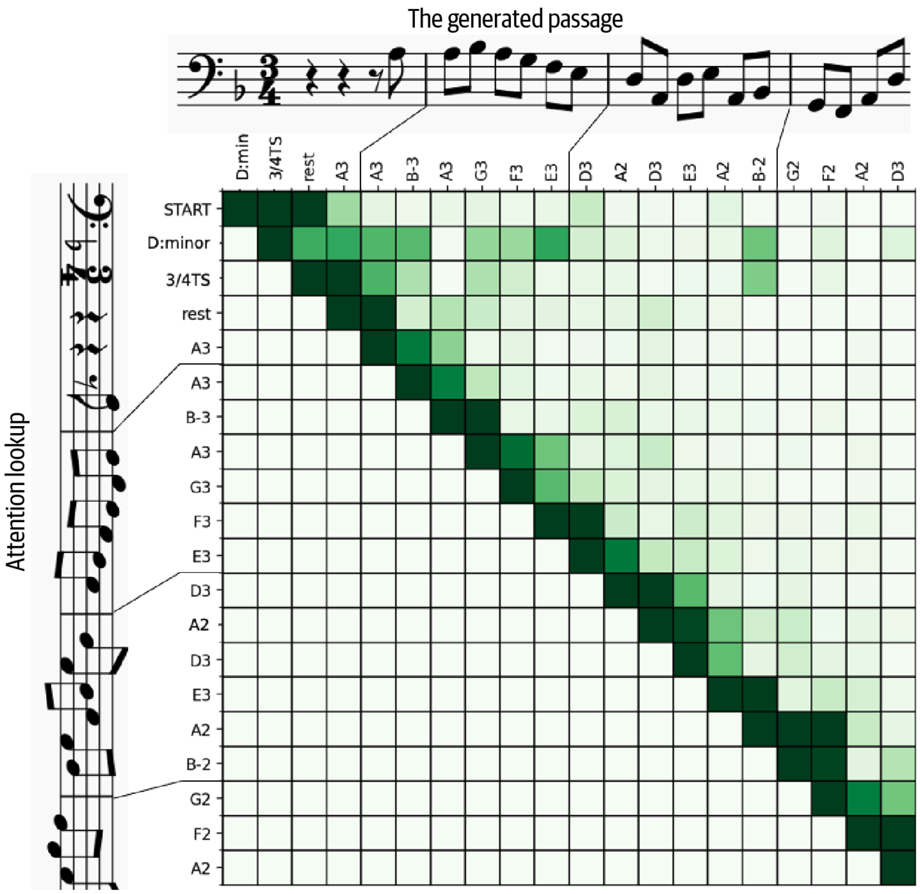


Figure 11-10. The color of each square in the matrix indicates the amount of attention given to each position on the vertical axis, at the point of predicting the note on the horizontal axis

There are also examples of where the network has chosen to ignore a certain note or rest nearby, as it doesn't add any additional information to its understanding of the phrase. For example, the penultimate note (`A2`) is not particularly attentive to the `B-2` three notes back, but is slightly more attentive to the `A2` four notes back. It is more interesting for the model to look at the `A2` that falls on the beat, rather than the `B-2` off the beat, which is just a passing note.

Remember we haven't told the model anything about which notes are related or which notes belong to which key signatures—it has worked this out for itself just by studying the music of J.S. Bach.

Tokenization of Polyphonic Music

The Transformer we've been exploring in this section works well for single-line (monophonic) music, but could it be adapted to multiline (polyphonic) music?

The challenge lies in how to represent the different lines of music as a single sequence of tokens. In the previous section we decided to split the notes and durations of the notes into two distinct inputs and outputs of the network, but we also saw that we could have interleaved these tokens into a single stream. We can use the same idea to handle polyphonic music. Two different approaches will be introduced here: *grid tokenization* and *event-based tokenization*, as discussed in the 2018 paper "Music Transformer: Generating Music with Long-Term Structure."[1]

Grid tokenization

Consider the two bars of music from a J.S. Bach chorale in Figure 11-11. There are four distinct parts (soprano [S], alto [A], tenor [T], bass [B]), written on different staffs.

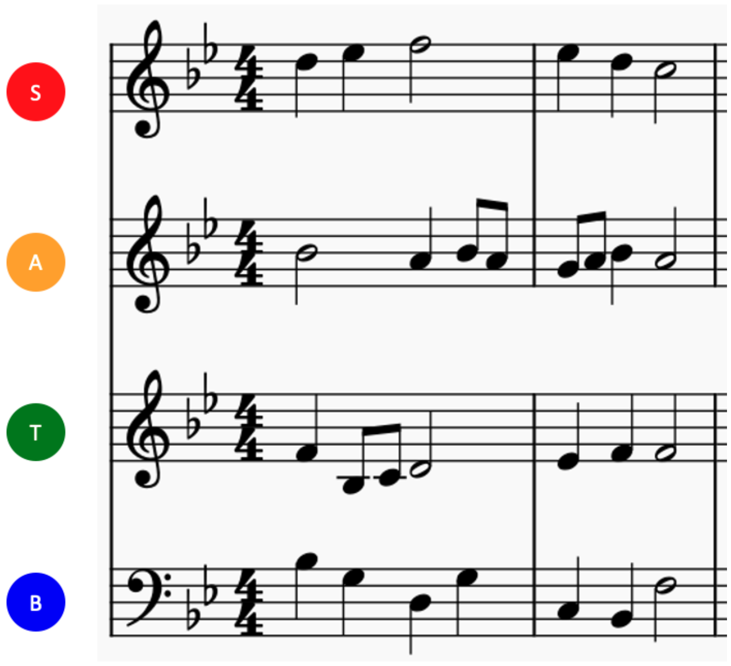
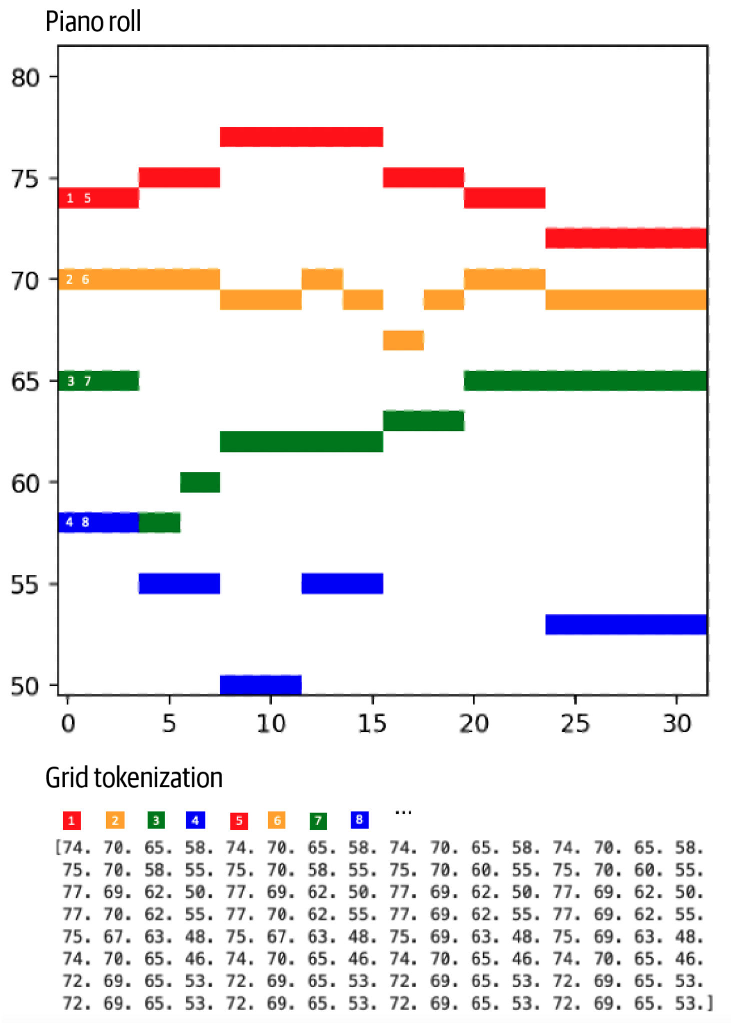


Figure 11-11. The first two bars of a J.S. Bach chorale

We can imagine drawing this music on a grid, where the y-axis represents the pitch of the note and the x-axis represents the number of 16th-notes (semiquavers) that have passed since the start of the piece. If the grid square is filled, then there is a note playing at that point in time. All four parts are drawn on the same grid. This grid is known as a *piano roll* because it resembles a physical roll of paper with holes punched into it, which was used as a recording mechanism before digital systems were invented.

We can serialize the grid into a stream of tokens by moving first through the four voices, then along the timesteps in sequence. This produces a sequence of tokens $S_1, A_1, T_1, B_1, S_2, A_2, T_2, B_2, \ldots$, where the subscript denotes the timestep, as shown in Figure 11-12.



```
[74. 70. 65. 58. 74. 70. 65. 58. 74. 70. 65. 58. 74. 70. 65. 58.
 75. 70. 58. 55. 75. 70. 58. 55. 75. 70. 60. 55. 75. 70. 60. 55.
 77. 69. 62. 50. 77. 69. 62. 50. 77. 69. 62. 50. 77. 69. 62. 50.
 77. 70. 62. 55. 77. 70. 62. 55. 77. 69. 62. 55. 77. 69. 62. 55.
 75. 67. 63. 48. 75. 67. 63. 48. 75. 69. 63. 48. 75. 69. 63. 48.
 74. 70. 65. 46. 74. 70. 65. 46. 74. 70. 65. 46. 74. 70. 65. 46.
 72. 69. 65. 53. 72. 69. 65. 53. 72. 69. 65. 53. 72. 69. 65. 53.
 72. 69. 65. 53. 72. 69. 65. 53. 72. 69. 65. 53. 72. 69. 65. 53.]
```

Figure 11-12. Creating the grid tokenization for the first two bars of the Bach chorale

We would then train our Transformer on this sequence of tokens, to predict the next token given the previous tokens. We can decode the generated sequence back into a grid structure by rolling the sequence back out over time in groups of four notes (one for each voice). This technique works surprisingly well, despite the same note often being split across multiple tokens with tokens from other voices in between.

However, there are some disadvantages. Firstly, notice that there is no way for the model to tell the difference between one long note and two shorter adjacent notes of the same pitch. This is because the tokenization does not explicitly encode the duration of notes, only whether a note is present at each timestep.

Secondly, this method requires the music to have a regular beat that is divisible into reasonably sized chunks. For example, using the current system, we cannot encode triplets (a group of three notes played across a single beat). We could divide the music into 12 steps per quarter-note (crotchet) instead of 4, that would triple the number of tokens required to represent the same passage of music, adding overhead on the training process and affecting the lookback capacity of the model.

Lastly, it is not obvious how we might add other components to the tokenization, such as dynamics (how loud or quiet the music is in each part) or tempo changes. We are locked into the two-dimensional grid structure of the piano roll, which provides a convenient way to represent pitch and timing, but not necessarily an easy way to incorporate other components that make music interesting to listen to.

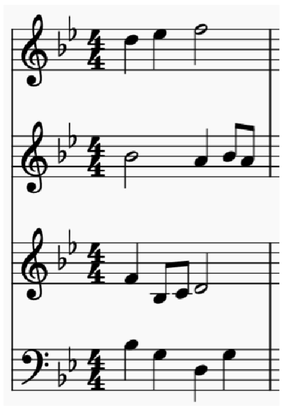
Event-based tokenization

A more flexible approach is to use event-based tokenization. This can be thought of as a vocabulary that literally describes how the music is created as a sequence of events, using a rich set of tokens.

For example in Figure 11-13, we use three types of tokens:

- `NOTE_ON<`*`pitch`*`>` (start playing a note of a given pitch)
- `NOTE_OFF<`*`pitch`*`>` (stop playing a note of a given pitch)
- `TIME_SHIFT<`*`step`*`>` (shift forward in time by a given step)

This vocabulary can be used to create a sequence that describes the construction of the music as a set of instructions.



Event tokenization

```
[NOTE_ON<74>, NOTE_ON<70>, NOTE_ON<65>, NOTE_ON<58>
, TIME_SHIFT<1.0>
, NOTE_OFF<74>, NOTE_OFF<65>, NOTE_OFF<58>
, NOTE_ON<75>, NOTE_ON<58>, NOTE_ON<55>
, TIME_SHIFT<0.5>
, NOTE_OFF<58>, NOTE_ON<60>
, TIME_SHIFT<0.5>
, NOTE_OFF<75>, NOTE_OFF<70>, NOTE_OFF<60>, NOTE_OFF<55>
, NOTE_ON<77>, NOTE_ON<69>, NOTE_ON<62>, NOTE_ON<50>
, TIME_SHIFT<1.0>
, NOTE_OFF<69>, NOTE_OFF<50>, NOTE_ON<70>, NOTE_ON<55>
, TIME_SHIFT<0.5>
, NOTE_OFF<70>, NOTE_ON<69>
, TIME_SHIFT<0.5>
, NOTE_OFF<77>, NOTE_OFF<69>, NOTE_OFF<62>, NOTE_OFF<55>
, ...
```

Figure 11-13. An event tokenization for the first bar of the Bach chorale

We could easily incorporate other types of tokens into this vocabulary, to represent dynamic and tempo changes for subsequent notes. This method also provides a way to generate triplets against a backdrop of quarter-notes, by separating the notes of the triplets with `TIME_SHIFT<0.33>` tokens. Overall, it is a more expressive framework for tokenization, though it is also potentially more complex for the Transformer to learn inherent patterns in the training set music, as it is by definition less structured than the grid method.

I encourage you to try implementing these polyphonic techniques and train a Transformer on the new tokenized dataset using all the knowledge you have built up so far in this book. I would also recommend checking our Dr. Tristan Behrens's guide to music generation research, available on GitHub, which provides a comprehensive overview of different papers on the topic of music generation using deep learning.

In the next section we will take a completely different approach to music generation, using GANs.

MuseGAN

You may have thought that the piano roll shown in Figure 11-12 looks a bit like a piece of modern art. This begs the question—could we in fact treat this piano roll as a *picture* and utilize image generation methods instead of sequence generation techniques?

As we shall see, the answer to this question is yes, we can treat music generation directly as an image generation problem. This means that instead of using Transformers, we can apply the same convolutional-based techniques that work so well for image generation problems—in particular, GANs.

MuseGAN was introduced in the 2017 paper "MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment."[2] The authors show how it is possible to train a model to generate polyphonic, multitrack, multibar music through a novel GAN framework. Moreover, they show how, by dividing up the responsibilities of the noise vectors that feed the generator, they are able to maintain fine-grained control over the high-level temporal and track-based features of the music.

Let's start by introducing the the J.S. Bach chorale dataset.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/11_music/02_musegan/musegan.ipynb* in the book repository.

The Bach Chorale Dataset

To begin this project, you'll first need to download the MIDI files that we'll be using to train the MuseGAN. We'll use a dataset of 229 J.S. Bach chorales for four voices.

You can download the dataset by running the Bach chorale dataset downloader script in the book repository, as shown in Example 11-5. This will save the MIDI files locally to the */data* folder.

Example 11-5. Downloading the Bach chorale dataset

```
bash scripts/download_bach_chorale_data.sh
```

The dataset consists of an array of four numbers for each timestep: the MIDI note pitches of each of the four voices. A timestep in this dataset is equal to a 16th note (a semiquaver). So, for example, in a single bar of 4 quarter (crotchet) beats, there are 16 timesteps. The dataset is automatically split into *train*, *validation*, and *test* sets. We will be using the *train* dataset to train the MuseGAN.

To start, we need to get the data into the correct shape to feed the GAN. In this example we'll generate two bars of music, so we'll extract only the first two bars of each chorale. Each bar consists of 16 timesteps and there are a potential 84 pitches across the 4 voices.

Voices will be referred to as *tracks* from here on, to keep the terminology in line with the original paper.

Therefore, the transformed data will have the following shape:

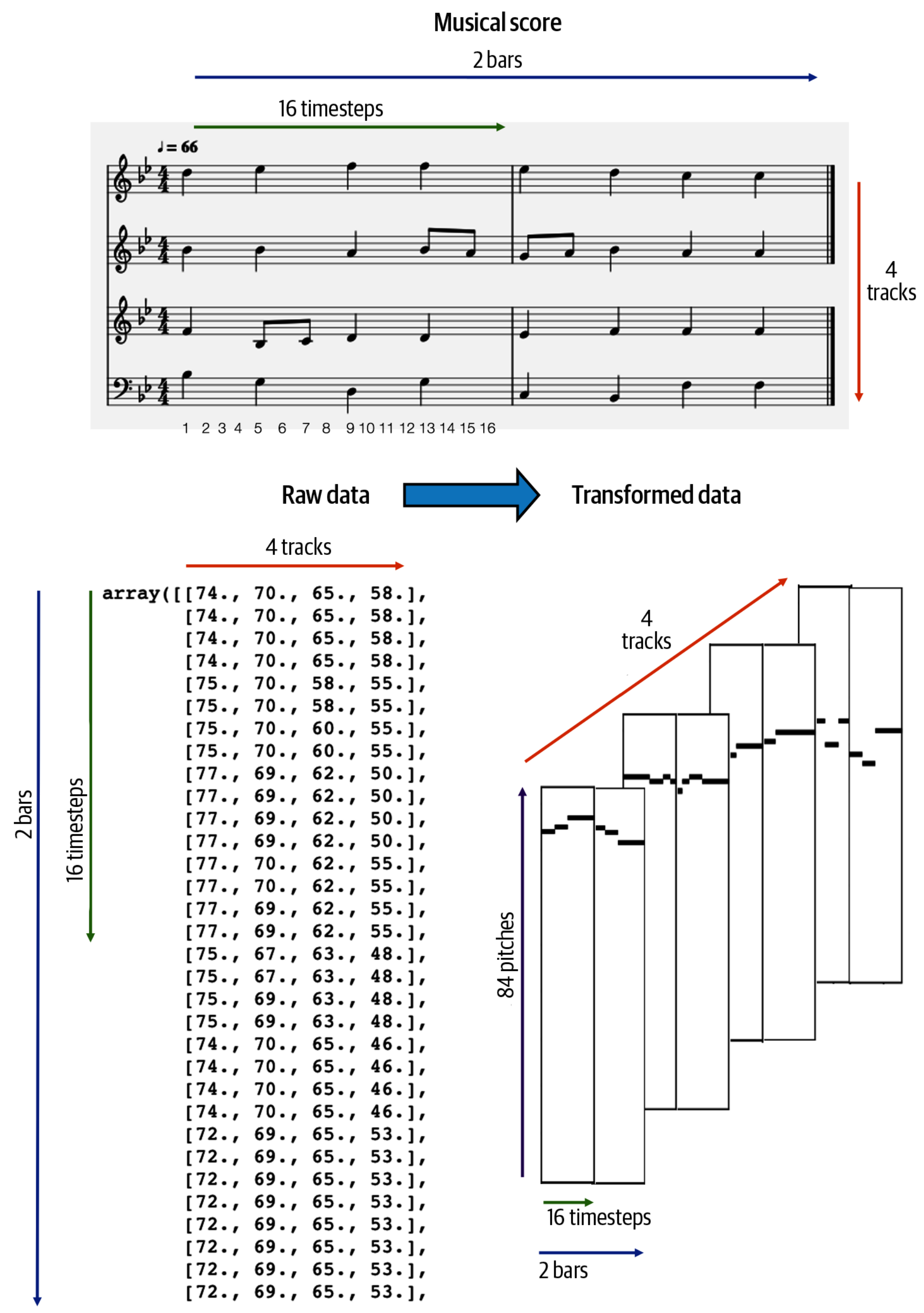
```
[BATCH_SIZE, N_BARS, N_STEPS_PER_BAR, N_PITCHES, N_TRACKS]
```

where:

```
BATCH_SIZE = 64
N_BARS = 2
N_STEPS_PER_BAR = 16
N_PITCHES = 84
N_TRACKS = 4
```

To get the data into this shape, we one-hot encode the pitch numbers into a vector of length 84 and split each sequence of notes into two bars of 16 timesteps each. We are making the assumption here that each chorale in the dataset has four beats in each bar, which is reasonable, and even if this were not the case it would not adversely affect the training of the model.

Figure 11-14 shows how two bars of raw data are converted into the transformed piano roll dataset that we will use to train the GAN.



```
array([[74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [74., 70., 65., 58.],
       [75., 70., 58., 55.],
       [75., 70., 58., 55.],
       [75., 70., 60., 55.],
       [75., 70., 60., 55.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 69., 62., 50.],
       [77., 70., 62., 55.],
       [77., 70., 62., 55.],
       [77., 69., 62., 55.],
       [77., 69., 62., 55.],
       [75., 67., 63., 48.],
       [75., 67., 63., 48.],
       [75., 69., 63., 48.],
       [75., 69., 63., 48.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [74., 70., 65., 46.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
       [72., 69., 65., 53.],
```

Figure 11-14. Processing two bars of raw data into piano roll data that we can use to train the GAN

The MuseGAN Generator

Like all GANs, MuseGAN consists of a generator and a critic. The generator tries to fool the critic with its musical creations, and the critic tries to prevent this from happening by ensuring it is able to tell the difference between the generator's forged Bach chorales and the real thing.

Where MuseGAN differs is in the fact that the generator doesn't just accept a single noise vector as input, but instead has four separate inputs, which correspond to four different characteristics of the music: chords, style, melody, and groove. By manipulating each of these inputs independently we can change high-level properties of the generated music.

A high-level view of the generator is shown in Figure 11-15.

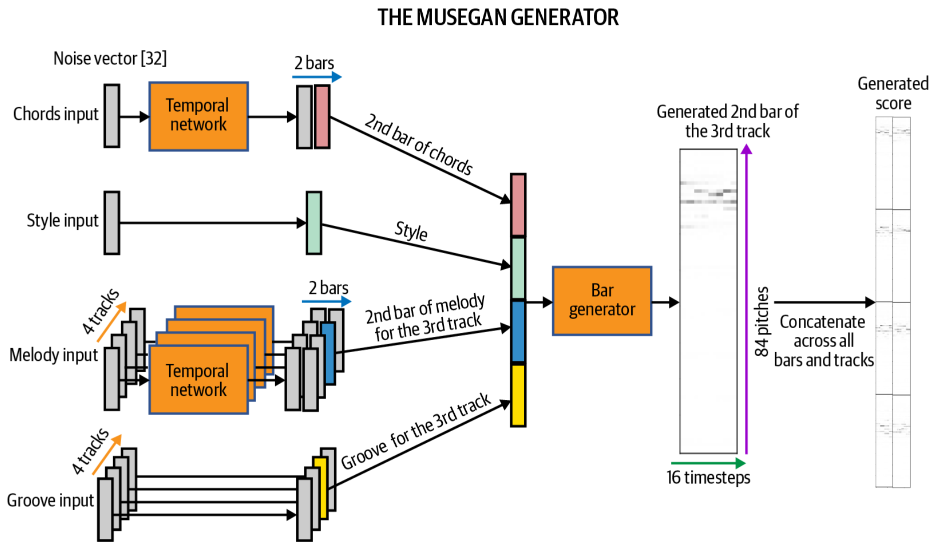


Figure 11-15. High-level diagram of the MuseGAN generator

The diagram shows how the chords and melody inputs are first passed through a *temporal network* that outputs a tensor with one of the dimensions equal to the number of bars to be generated. The style and groove inputs are not stretched temporally in this way, as they remain constant through the piece.

Then, to generate a particular bar for a particular track, the relevant outputs from the chords, style, melody, and groove parts of the network are concatenated to form a longer vector. This is then passed to a bar generator, which ultimately outputs the specified bar for the specified track.

By concatenating the generated bars for all tracks, we create a score that can be compared with real scores by the critic.

Let's first take a look at how to build a temporal network.

The temporal network

The job of a temporal network—a neural network consisting of convolutional transpose layers—is to transform a single input noise vector of length `Z_DIM = 32` into a different noise vector for every bar (also of length 32). The Keras code to build this is shown in Example 11-6.

Example 11-6. Building the temporal network

```
def conv_t(x, f, k, s, a, p, bn):
    x = layers.Conv2DTranspose(
                filters = f
                , kernel_size = k
                , padding = p
                , strides = s
                , kernel_initializer = initializer
                )(x)
    if bn:
        x = layers.BatchNormalization(momentum = 0.9)(x)

    x = layers.Activation(a)(x)
    return x

def TemporalNetwork():
    input_layer = layers.Input(shape=(Z_DIM,), name='temporal_input') ❶
    x = layers.Reshape([1,1,Z_DIM])(input_layer) ❷
    x = conv_t(
        x, f=1024, k=(2,1), s=(1,1), a = 'relu', p = 'valid', bn = True
    ) ❸
    x = conv_t(
        x, f=Z_DIM, k=(N_BARS - 1,1), s=(1,1), a = 'relu', p = 'valid', bn = True
    )
    output_layer = layers.Reshape([N_BARS, Z_DIM])(x) ❹
    return models.Model(input_layer, output_layer)
```

❶ The input to the temporal network is a vector of length 32 (`Z_DIM`).

❷ We reshape this vector to a 1 × 1 tensor with 32 channels, so that we can apply convolutional 2D transpose operations to it.

❸ We apply `Conv2DTranspose` layers to expand the size of the tensor along one axis, so that it is the same length as `N_BARS`.

❹ We remove the unnecessary extra dimension with a `Reshape` layer.

The reason we use convolutional operations rather than requiring two independent vectors into the network is because we would like the network to learn how one bar should follow on from another in a consistent way. Using a neural network to expand the input vector along the time axis means the model has a chance to learn how music flows across bars, rather than treating each bar as completely independent of the last.

Chords, style, melody, and groove

Let's now take a closer look at the four different inputs that feed the generator:

Chords
: The chords input is a single noise vector of length `Z_DIM`. This vector's job is to control the general progression of the music over time, shared across tracks, so we use a `TemporalNetwork` to transform this single vector into a different latent vector for every bar. Note that while we call this input chords, it really could control anything about the music that changes per bar, such as general rhythmic style, without being specific to any particular track.

Style
: The style input is also a vector of length `Z_DIM`. This is carried forward without transformation, so it is the same across all bars and tracks. It can be thought of as the vector that controls the overall style of the piece (i.e., it affects all bars and tracks consistently).

Melody
: The melody input is an array of shape `[N_TRACKS, Z_DIM]`—that is, we provide the model with a random noise vector of length `Z_DIM` for each track.

  Each of these vectors is passed through a track-specific `TemporalNetwork`, where the weights are not shared between tracks. The output is a vector of length `Z_DIM` for every bar of every track. The model can therefore use these input vectors to fine-tune the content of every single bar and track independently.

Groove
: The groove input is also an array of shape `[N_TRACKS, Z_DIM]`—a random noise vector of length `Z_DIM` for each track. Unlike the melody input, these vectors are not passed through the temporal network but instead are fed straight through, just like the style vector. Therefore, each groove vector will affect the overall properties of a track, across all bars.

We can summarize the responsibilities of each component of the MuseGAN generator as shown in Table 11-1.

Table 11-1. Components of the MuseGAN generator

	Output differs across bars?	Output differs across parts?
Style	X	X
Groove	X	✓
Chords	✓	X
Melody	✓	✓

The final piece of the MuseGAN generator is the *bar generator*—let's see how we can use this to glue together the outputs from the chord, style, melody, and groove components.

The bar generator

The bar generator receives four latent vectors—one from each of the chord, style, melody, and groove components. These are concatenated to produce a vector of length `4 * Z_DIM` as input. The output is a piano roll representation of a single bar for a single track—i.e., a tensor of shape `[1, n_steps_per_bar, n_pitches, 1]`.

The bar generator is just a neural network that uses convolutional transpose layers to expand the time and pitch dimensions of the input vector. We create one bar generator for every track, and weights are not shared between tracks. The Keras code to build a `BarGenerator` is given in Example 11-7.

Example 11-7. Building the `BarGenerator`

```
def BarGenerator():

    input_layer = layers.Input(shape=(Z_DIM * 4,), name='bar_generator_input') ❶

    x = layers.Dense(1024)(input_layer) ❷
    x = layers.BatchNormalization(momentum = 0.9)(x)
    x = layers.Activation('relu')(x)
    x = layers.Reshape([2,1,512])(x)

    x = conv_t(x, f=512, k=(2,1), s=(2,1), a= 'relu',  p = 'same', bn = True) ❸
    x = conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)
    x = conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)
    x = conv_t(x, f=256, k=(1,7), s=(1,7), a= 'relu', p = 'same', bn = True) ❹
    x = conv_t(x, f=1, k=(1,12), s=(1,12), a= 'tanh', p = 'same', bn = False) ❺

    output_layer = layers.Reshape([1, N_STEPS_PER_BAR , N_PITCHES ,1])(x) ❻

    return models.Model(input_layer, output_layer)
```

❶ The input to the bar generator is a vector of length `4 * Z_DIM`.

❷ After passing it through a `Dense` layer, we reshape the tensor to prepare it for the convolutional transpose operations.

❸ First we expand the tensor along the timestep axis…

❹ …then along the pitch axis.

❺ The final layer has a tanh activation applied, as we will be using a WGAN-GP (which requires tanh output activation) to train the network.

❻ The tensor is reshaped to add two extra dimensions of size 1, to prepare it for concatenation with other bars and tracks.

Putting it all together

Ultimately, the MuseGAN generator takes the four input noise tensors (chords, style, melody, and groove) and converts them into a multitrack, multibar score. The Keras code to build the MuseGAN generator is provided in Example 11-8.

Example 11-8. Building the MuseGAN generator

```
def Generator():
    chords_input = layers.Input(shape=(Z_DIM,), name='chords_input') ❶
    style_input = layers.Input(shape=(Z_DIM,), name='style_input')
    melody_input = layers.Input(shape=(N_TRACKS, Z_DIM), name='melody_input')
    groove_input = layers.Input(shape=(N_TRACKS, Z_DIM), name='groove_input')

    chords_tempNetwork = TemporalNetwork() ❷
    chords_over_time = chords_tempNetwork(chords_input)

    melody_over_time = [None] * N_TRACKS
    melody_tempNetwork = [None] * N_TRACKS
    for track in range(N_TRACKS):
        melody_tempNetwork[track] = TemporalNetwork() ❸
        melody_track = layers.Lambda(lambda x, track = track: x[:,track,:])(
            melody_input
        )
        melody_over_time[track] = melody_tempNetwork[track](melody_track)

    barGen = [None] * N_TRACKS
    for track in range(N_TRACKS):
        barGen[track] = BarGenerator() ❹

    bars_output = [None] * N_BARS
    c = [None] * N_BARS
    for bar in range(N_BARS): ❺
```

```
        track_output = [None] * N_TRACKS

        c[bar] = layers.Lambda(lambda x, bar = bar: x[:,bar,:])(chords_over_time)
        s = style_input

        for track in range(N_TRACKS):

            m = layers.Lambda(lambda x, bar = bar: x[:,bar,:])(
                melody_over_time[track]
            )
            g = layers.Lambda(lambda x, track = track: x[:,track,:])(
                groove_input
            )

            z_input = layers.Concatenate(
                axis = 1, name = 'total_input_bar_{}_track_{}'.format(bar, track)
            )([c[bar],s,m,g])

            track_output[track] = barGen[track](z_input)

        bars_output[bar] = layers.Concatenate(axis = -1)(track_output)

    generator_output = layers.Concatenate(axis = 1, name = 'concat_bars')(
        bars_output
    ) ❻

    return models.Model(
        [chords_input, style_input, melody_input, groove_input], generator_output
    ) ❼

generator = Generator()
```

❶ Define the inputs to the generator.

❷ Pass the chords input through the temporal network.

❸ Pass the melody input through the temporal network.

❹ Create an independent bar generator network for every track.

❺ Loop over the tracks and bars, creating a generated bar for each combination.

❻ Concatenate everything together to form a single output tensor.

❼ The MuseGAN model takes four distinct noise tensors as input and outputs a generated multitrack, multibar score.

The MuseGAN Critic

In comparison to the generator, the critic architecture is much more straightforward (as is often the case with GANs).

The critic tries to distinguish full multitrack, multibar scores created by the generator from real excerpts from the Bach chorales. It is a convolutional neural network, consisting mostly of `Conv3D` layers that collapse the score into a single output prediction.

Conv3D Layers

So far in this book, we have only worked with `Conv2D` layers, applicable to three-dimensional input images (width, height, channels). Here we have to use `Conv3D` layers, which are analogous to `Conv2D` layers but accept four-dimensional input tensors (`n_bars`, `n_steps_per_bar`, `n_pitches`, `n_tracks`).

We do not use batch normalization layers in the critic as we will be using the WGAN-GP framework for training the GAN, which forbids this.

The Keras code to build the critic is given in Example 11-9.

Example 11-9. Building the MuseGAN critic

```
def conv(x, f, k, s, p):
    x = layers.Conv3D(filters = f
                , kernel_size = k
                , padding = p
                , strides = s
                , kernel_initializer = initializer
                )(x)
    x = layers.LeakyReLU()(x)
    return x

def Critic():
    critic_input = layers.Input(
        shape=(N_BARS, N_STEPS_PER_BAR, N_PITCHES, N_TRACKS),
        name='critic_input'
    ) ❶

    x = critic_input
    x = conv(x, f=128, k = (2,1,1), s = (1,1,1), p = 'valid') ❷
    x = conv(x, f=128, k = (N_BARS - 1,1,1), s = (1,1,1), p = 'valid')

    x = conv(x, f=128, k = (1,1,12), s = (1,1,12), p = 'same') ❸
    x = conv(x, f=128, k = (1,1,7), s = (1,1,7), p = 'same')

    x = conv(x, f=128, k = (1,2,1), s = (1,2,1), p = 'same') ❹
    x = conv(x, f=128, k = (1,2,1), s = (1,2,1), p = 'same')
    x = conv(x, f=256, k = (1,4,1), s = (1,2,1), p = 'same')
    x = conv(x, f=512, k = (1,3,1), s = (1,2,1), p = 'same')

    x = layers.Flatten()(x)

    x = layers.Dense(1024, kernel_initializer = initializer)(x)
    x = layers.LeakyReLU()(x)

    critic_output = layers.Dense(
        1, activation=None, kernel_initializer = initializer
    )(x) ❺

    return models.Model(critic_input, critic_output)

critic = Critic()
```

❶ The input to the critic is an array of multitrack, multibar scores, each of shape `[N_BARS, N_STEPS_PER_BAR, N_PITCHES, N_TRACKS]`.

❷ First, we collapse the tensor along the bar axis. We apply `Conv3D` layers throughout the critic as we are working with 4D tensors.

❸ Next, we collapse the tensor along the pitch axis.

❹ Finally, we collapse the tensor along the timesteps axis.

❺ The output is a `Dense` layer with a single unit and no activation function, as required by the WGAN-GP framework.

Analysis of the MuseGAN

We can perform some experiments with our MuseGAN by generating a score, then tweaking some of the input noise parameters to see the effect on the output.

The output from the generator is an array of values in the range [–1, 1] (due to the tanh activation function of the final layer). To convert this to a single note for each track, we choose the note with the maximum value over all 84 pitches for each timestep. In the original MuseGAN paper the authors use a threshold of 0, as each track can contain multiple notes; however, in this setting we can simply take the maximum to guarantee exactly one note per timestep per track, as is the case for the Bach chorales.

Figure 11-16 shows a score that has been generated by the model from random normally distributed noise vectors (top left). We can find the closest score in the dataset (by Euclidean distance) and check that our generated score isn't a copy of a piece of music that already exists in the dataset—the closest score is shown just below it, and we can see that it does not resemble our generated score.

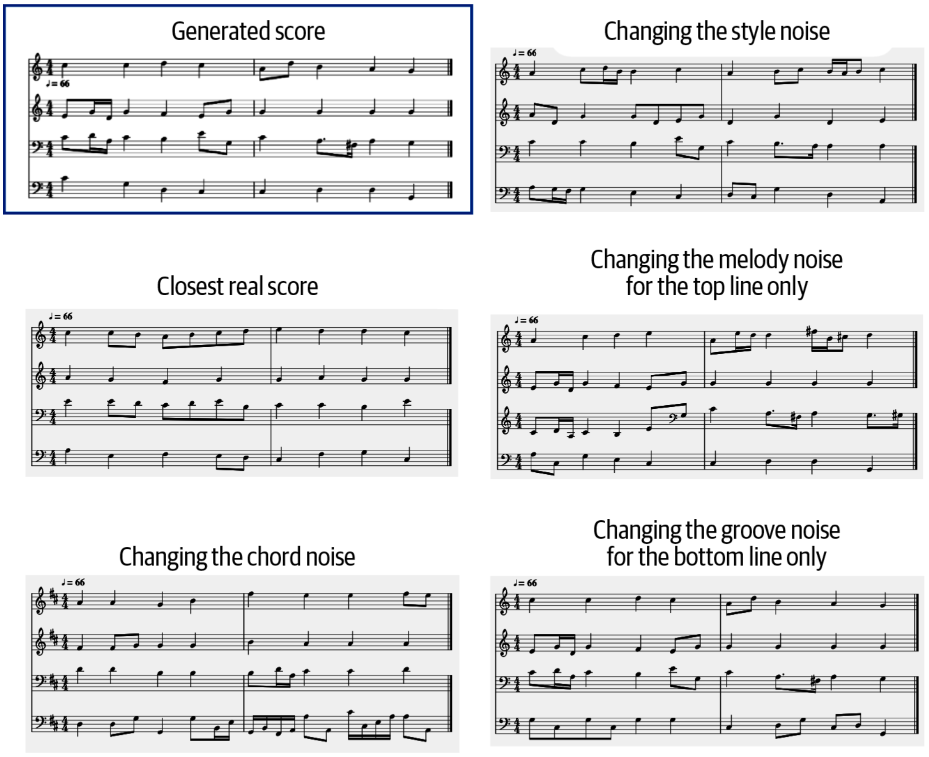

Generated score
Changing the style noise
Closest real score
Changing the melody noise for the top line only
Changing the chord noise
Changing the groove noise for the bottom line only

Figure 11-16. Example of a MuseGAN predicted score, showing the closest real score in the training data and how the generated score is affected by changing the input noise

Let's now play around with the input noise to tweak our generated score. First, we can try changing the chord noise vector—the bottom-left score in Figure 11-16 shows the result. We can see that every track has changed, as expected, and also that the two bars exhibit different properties. In the second bar, the baseline is more dynamic and the top line is higher in pitch than in the first bar. This is because the latent vectors that affect the two bars are different, as the input chord vector was passed through a temporal network.

When we change the style vector (top right), both bars change in a similar way. The whole passage has changed style from the original generated score, in a consistent way (i.e., the same latent vector is being used to adjust all tracks and bars).

We can also alter tracks individually, through the melody and groove inputs. In the center-right score in Figure 11-16 we can see the effect of changing just the melody noise input for the top line of music. All other parts remain unaffected, but the top-line notes change significantly. Also, we can see a rhythmic change between the two bars in the top line: the second bar is more dynamic, containing faster notes than the first bar.

Lastly, the bottom-right score in the diagram shows the predicted score when we alter the groove input parameter for only the baseline. Again, all other parts remain unaffected, but the baseline is different. Moreover, the overall pattern of the baseline remains similar between bars, as we would expect.

This shows how each of the input parameters can be used to directly influence high-level features of the generated musical sequence, in much the same way as we were able to adjust the latent vectors of VAEs and GANs in previous chapters to alter the appearance of a generated image. One drawback to the model is that the number of bars to generate must be specified up front. To tackle this, the authors show an extension to the model that allows previous bars to be fed in as input, allowing the model to generate long-form scores by continually feeding the most recent predicted bars back in as additional input.

Summary

In this chapter we have explored two different kinds of models for music generation: a Transformer and a MuseGAN.

The Transformer is similar in design to the networks we saw in Chapter 9 for text generation. Music and text generation share a lot of features in common, and often similar techniques can be used for both. We extended the Transformer architecture by incorporating two input and output streams, for note and duration. We saw how the model was able to learn about concepts such as keys and scales, simply by learning to accurately generate the music of Bach.

We also explored how we can adapt the tokenization process to handle polyphonic (multitrack) music generation. Grid tokenization serializes a piano roll representation of the score, allowing us to train a Transformer on a single stream of tokens that describe which note is present in each voice, at discrete, equally spaced timestep intervals. Event-based tokenization produces a *recipe* that describes how to create the multiple lines of music in a sequential fashion, through a single stream of instructions. Both methods have advantages and disadvantages—the success or failure of a Transformer-based approach to music generation is often heavily dependent on the choice of tokenization method.

We also saw that generating music does not always require a sequential approach—MuseGAN uses convolutions to generate polyphonic musical scores with multiple tracks, by treating the score as an image where the tracks are individual channels of the image. The novelty of MuseGAN lies in the way the four input noise vectors (chords, style, melody, and groove) are organized so that it is possible to maintain full control over high-level features of the music. While the underlying harmonization is still not as perfect or varied as Bach's, it is a good attempt at what is an extremely difficult problem to master and highlights the power of GANs to tackle a wide variety of problems.

References

1. Cheng-Zhi Anna Huang et al., "Music Transformer: Generating Music with Long-Term Structure," September 12, 2018, *https://arxiv.org/abs/1809.04281*.

2. Hao-Wen Dong et al., "MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," September 19, 2017, *https://arxiv.org/abs/1709.06298*.