# Music Composition With Recurrent Neural Networks

Andrei Ciupan

Abstract. I produce a method for composing multi-track piano music using a prediction function trained from a dataset of existing songs. The prediction function uses a recurrent neural net architecture with LSTM cells. Composition examples and code are provided.

## 1. Results

Here are some sample compositions of the final trained model:1, 2, 3, 4, 5, 6, 7, 8, 9. For each example the first 8 seconds were given as input and the rest were produced by the composition program. The first 8 seconds were parts of songs drawn from a different dataset than the one the prediction function trained on. The code is available here.

## 2. Description

Let $S$ be a set of songs. In our case, $S$ is the set of all songs in MIDI format. For the task of music composition, I produce a method $M : S \mapsto S$ such that the input to $M$ is some song, and the output is a song of much longer length. In the sample compositions in this paper, the input songs are of length 8 seconds. Inputs of length 1 or 2 seconds give similar outputs (just wait a few seconds).

The method $M$ is defined by the following features:

(1) Encoding: A set $E$ of events and an encoding function $e : S \mapsto E^{\mathbb{N}}$, where $E^{\mathbb{N}}$ is the set of all sequences with elements in $E$. In words, the encoding $e$ transforms any song into a sequence of events. Let $E(e, S)$ be the set of all sequences of events which are produced by encoding a song in $S$ with $e$. The encoding function should be one-to-one, i.e. $e(s) \neq e(s')$ for $s \neq s'$.

(2) Composition: A composition function $c : E(e, S) \mapsto E(e, S)$ with the property that if $x = (e_1, e_2, \ldots, e_n)$ then $c(x)$ is a sequence of longer length than $x$, and the first $n$ elements of $c(x)$ are equal to $x$.

(3) Decoding: A function $d : E(e, S) \mapsto S$ such that $d(e(s)) = s$ for $s \in S$.

Let's describe each of these in detail.

## 3. Encoding

Each song in our dataset has two tracks playing at once. Let's call these tracks the *melody* and the *chords*. Each track is described by the set of notes playing at a given time. At any moment in time, a new note could be played, an old note could be stopped, or an old note could continue playing. This is just like playing at a piano: a note of G is played for 2 seconds if its corresponding key was pressed at time $t$ and held until time $t+2$, when it is released. In midi format, such a sound is labeled as a note playing for time interval $[t, t+2]$. Also, multiple notes could be played at once.

We introduce one quirky definition. We say that two notes in a song *irk* each other if there exist two intervals $I_1 = [a_1, b_1]$ and $I_2 = [a_2, b_2]$ which overlap, but none is a subset of the other, such that the first note is played for time interval $I_1$ and the second is played for time interval $I_2$. And two notes *sooth* each other if they play in overlaping time intervals and do not irk. Data inspection reveals two initial properties of our tracks (remember that a track is either the melody or the chords of a song, not the combination of the two).

(1) No notes irk each other
(2) If multiple notes are playing at once, then they all play for the same time interval.

Note that the second result also implies the first one. But in data inspection, of course I started with inspecting the first result and then moved to confirm the second.

The second result allows for the following definition: a *sound* corresponds to the set of notes playing at the same time, at some moment in the track. This sound can be a singleton, of course, or it can be the empty set, if no note is played.

With this definition, we can characterize a track: A track is a sequence of sounds playing successively, in non-overlapping intervals. Call the length of such an interval the *duration* of a sound.

Remember there are two types of tracks, the chord and the melody. For a track of a certain type, let $K$ be the set of all sounds which could play in a track of that type. If $t$ is a positive real number such that all durations are integer multiples of $t$, then any song $s$ can be fully described as a sequence $e(s) = \overline{e_1 e_2 \dots e_n}$, where:

(1) Each element $e_i$ is called an *event*, and is either an element of $K$ or a *continuation event*. If an event is an element of $K$, it is called a *sound event*. There is only one continuation event.
(2) The sequence can be split into consecutive and nonoverlapping subsequences with the property that each one begins with a sound event[1]. Each such a subsequence is called a *moment*. Each moment is characterized fully by the sound event at its beginning, and the length of the sequence. Let $K^+$ be the set of elements in $K$ and the continuation event.
(3) Remember that the song $s$ is a sequence of sounds playing successively for non-overlapping intervals. A sound of $e \in K$ for duration $d$ corresponds to the moment characterized by the same sound $e$, and of duration $\frac{d}{t}$[2].
(4) Let $s_1 \dots s_k$ be the list of consecutive sounds describing $s$, with each $s_i$ of duration $d_i$. The sequence $e(s) = \overline{e_1 e_2 \dots e_n}$ is the sequence of moments

[1]Since these subsequences are consecutive and nonoverlapping, there is a unique way to do this

[2]Remember that $d$ is divisible by $t$

$m_1 \dots m_k$, where each $m_i$ is a moment characterized by event $s_i$, and is of length $\frac{d_1}{t}$.

So, starting from the set $K$ of all sounds which could play in a track [3], we found a set $K^+$ of events and a function $e$ which encodes a track as a sequence of events from $K^+$. This function is clearly one-to-one.

Let's introduce another notation. A *one-hot* vector of size $n$ is a vector of size $n$ with zero entries except for one entry which equals to 1. Every element in $K^+$ has a one-hot representation in the natural way: we pick a mapping from the set $K^+$ to the set $[K^+] = \{1, 2, \dots, |K^+|\}$, and then map each element in $[K^+]$ to its one-hot natural representation.

So, we first map a song $s$ into a sequence of events, then map every event to its one-hot representation and thus we are able to map a song $s$ into a sequence of one-hot vectors. This completes our encoding.

We have found one way to encode a track. Now we combine the two tracks of a song. Data inspection reveals a few more properties of the two tracks in a song:

(1) The MIDI encoding of the melody and chord tracks is consistent, in the sense that the MIDI description of songs containing just the melody, and just the chord, are identical to the MIDI description of the song containing both.
(2) Most songs have only sounds of duration divisible by 128 miliseconds. We choose $t = 128$ miliseconds as our timestep.

Take a song $s$ and consider $e_1(s) = \overline{a_1 a_2 \dots a_n}$ be its melody representation and $e_2(s) = \overline{b_1 b_2 \dots b_n}$ be its chords representation. The above findings show that indeed these two representations have equal length. Remember that each $a_i$ is a one-hot vector, and so is each $b_i$.

Now, construct $c_i = \begin{pmatrix} a_1 \\ b_1 \end{pmatrix}$ for each $i$ , by stacking the event representations for the two track types. Since each $a_i$ and $b_i$ is one-hot, the vector $c_i$ has two values of 1, and the rest zeros. We can call this representation *two-hot*.

So, this is our final encoding!

Before we move on to composition, let's introduce recurrent neural networks.

## 4. Recurrent neural networks

A neural network is a prediction function.

First consider simple neural networks. Consider a neural network $f$ such that $f$ takes as input a $k$-dimensional vector and outputs another $k$-dimensional vector. A neural network $f$ with $n$ hidden layers is characterized by functions $h_1, \dots h_n$ and $o$. On input $x$, $f$ makes the following computations in succession:

$$x \mapsto C^{(1)}(x) \mapsto C^{(2)}(x) \cdots \mapsto C^{(n)}(x) \mapsto W(x)$$

The computations are made according to the following rules:

(1) $C^{(1)}(x) = h_1(x)$.
(2) $C^{(i)}(x) = h_i\left(C^{(i-1)}(x)\right)$ for $1 < i \leq n$.
(3) $W(x) = o\left(C^{(n)}(x)\right)$.

[3]Remember, of type *melody* or *chord*

In this setup, an *observation* is a pair $(x, y)$, where both $x, y \in \mathbb{R}^k$. The purpose of the prediction function is to predict $y$ from $x$. A *dataset* is a collection of observations $(x^{(i)}, y^{(i)})$.

A given prediction function $f \equiv (h_1, \ldots, h_n, o)$ has defined a loss function $l(x, y, f) = \Delta(W(x), y)$ over the observation $(x, y)$. The function $\Delta$ is chosen by the user, and it is supposed to be a measure of the *difference* between the prediction $W(x)$ and the actual output $y$.

Then, given a dataset $D$ of observations $(x^{(i)}, y^{(i)})$, an optimization solver finds the function $f$ which minimizes the loss on the whole dataset,

$$L(D, f) = \sum_{\left(x^{(i)}, y^{(i)}\right) \in D} l\left(x^{(i)}, y^{(i)}, f\right).$$

In an observation $(x, y)$, the first element of the pair, $x$, is called the *input* value and the second element of the pair, $y$, is called the *outcome* value.

In case when the dataset of observations has the property that the outcome value of any observation is a one-hot vector, the following loss function $\Delta(w, y)$ is used:

(1) Remember that by construction, the values $w$ and $y$ are $k$-dimensional, and $y$ has exactly one nonzero entry, say at dimension $p$.
(2) Transform $w = (w_1, \ldots, w_k)$ into a vector of probabilities $w' = (w'_1, \ldots, w'_k)$, where $w'_i = \dfrac{e^{w_i}}{\sum_{1 \le i \le k} e^{w_i}}$, for all $i$.
(3) Remember that $p$ is the dimension where the vector $y$ is nonzero. The loss $\Delta(w, y)$ is defined as $-\log(w'_p)$

This function is called a *softmax cross-entropy* loss function. This loss function makes sense. It wants to teach the function $f$ to output a large value for $W(x)$ in the dimension where the target $y$ is actually zero. We also saw above how to normalize the vector $W(x)$ into a vector of probabilities.

Now let's mode on to recurrent neural networks.

A recurrent neural network(RNN) is also a prediction function. An RNN $f$ takes as input a sequence of size $t$, $(x_1, \ldots, x_t)$ of $k$-dimensional vectors and outputs another sequence, also of size $t$, of $k$-dimensional vectors.

An RNN with $n$ hidden layers is characterized by functions $h_1, \ldots, h_n, r_1, \ldots, r_n$ and a function $o$. On input $(x_1, \ldots, x_t)$, $f$ makes the following computations in succession:

$$x_1 \mapsto C^{(1)}(x_1) \mapsto C^{(2)}(x_1) \mapsto \cdots \mapsto C^{(n)}(x_1) \mapsto W(x_1)$$

$$x_2 \mapsto C^{(1)}(x_1, x_2) \mapsto C^{(2)}(x_1, x_2) \mapsto \cdots \mapsto C^{(n)}(x_1, x_2) \mapsto W(x_1, x_2)$$

$$\vdots$$

$$x_t \mapsto C^{(1)}(x_1, x_2, \ldots, x_t) \mapsto C^{(2)}(x_1, x_2, \ldots, x_t) \mapsto \cdots \mapsto C^{(n)}(x_1, x_2 \ldots, x_t) \mapsto W(x_1, \ldots, x_t).$$

The computations are made according to the following rules:

(1) $C^{(1)}(x_1) = h_1(x_1)$.

(2) $C^{(i)}(x_1) = h_i\left(C^{(i-1)}(x_1)\right)$ for $1 < i \leq n$.
(3) $C^{(1)}(x_1, \ldots, x_j) = r_1\left(x_j, C^{(1)}(x_1, \ldots, x_{j-1})\right)$ for $1 < j \leq t$.
(4) $C^{(i)}(x_1, \ldots, x_j) = r_i\left(C^{(i-1)}(x_1, \ldots, x_j), C^{(i)}(x_1, \ldots, x_{j-1})\right)$, for $1 < j \leq t$ and $1 < i \leq n$.
(5) $W(x_1, \ldots, x_j) = o\left(C^{(n)}(x_1, \ldots, x_j)\right)$, for $1 \leq j \leq t$.

We observe that the RNN is a generalization of a simple neural network. The *link functions* $r_i$ take input not just from the current element of a sequence, $x_j$, but also from the calculation made at the same layer $i$ for the previous element of a sequence $x_{j-1}$.

Keep the same notation as in the case of the simple neural network. An observation is a pair $(x, y)$. A dataset is a collection of observations. A prediction function $f \equiv (h_1, \ldots, h_n, r_1, \ldots, r_n, o)$ has defined a loss function $l(x, y, f) = \Delta\left(W(x), y\right)$ over the observation $(x, y)$. $\Delta$ is chosen by the user, and an optimization solver finds the function which minimizes the loss on the whole dataset.

The difference is that now an observation $(x, y)$ consists of a pair of sequences of the same length. The loss $\Delta(W(x), y)$ consists of the sum of losses between each dimension of $W(x) = (w_1(x), \ldots, w_t(x))$ and each dimension of $y = (y_1, \ldots, y_t)$, $\Delta(W(x), y) = \sum_{1 \leq j \leq t} \delta(w_j(x), y_j)$. The user needs to choose $\delta$ now. It is a difference function between two vectors of the same dimension $k$.

Recall the softmax cross-entropy loss function described earlier in the paper. Call this function *crentloss*.

Recall the two-hot representation we constructed in the section describing encoding, where a vector $v$ is represented as $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$, where each $v_i$ is a one-hot representation from dictionaries of potentially different sizes, say the dictionary containing $v_1$ is of size $n_1$ and the one containing $v_2$ is of size $n_2$.

Recall that in our case, an observation has both input and outcome of the same dimension $k$. Consider the case where the outcome $y$ always can be represented in the one-hot representation as above, $y = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$. Consider the following value of the loss function $\delta(w, y)$:

(1) Write $y$ as $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$, where $y_1$ and $y_2$ represent the two components of the two-hot representation.
(2) Divide $w$ into $w = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$, with $w_1$ and $w_2$ of sizes corresponding to the sizes of $y_1$ and $y_2$.
(3) Define $\delta(w, y) = crentloss(w_1, y_1) + crentloss(w_2, y_2)$.

By minimizing this loss function, we (hopefully) train a model which tries to minimize both losses, and can predict both positions of the two-hot representation, not just one. This is a neat application of the classical principles of machine learning from simple neural networks. A recurrent neural network trained with this loss function it trained to be able to predict the next elements $y_t$ from a sequence of inputs $(x_1, x_2, \ldots, x_t)$.

Now we can move on to composition.

## 5. Composition

Recall the goal we have in mind: find a prediction function which, starting from a sequence $(e_1, e_2, \ldots, e_t)$ can add a new event $e_{t+1}$ to the sequence in a meaningful way. We already defined our dictionary of events such that any added event $e_{t+1}$ produces a new sequence $(e_1, e_2, \ldots, e_t, e_{t+1})$ which can be encoded as a song. This is great. Now we just want to make this meaningful. With the recurrent neural infrastructure defined in the previous section, we are almost done. We do the following:

(1) Encode songs as sequences of events as described above. Split each song into consecutive subsequences of the same fixed size. In the code I call these *segments*.
(2) From each segment $(e_1, \ldots, e_t)$, construct an observation $(x, y)$ in the following way: $x = (e_1, \ldots, e_{t-1})$ and $y = (e_2, \ldots, e_t)$.
(3) Train the recurrent neural network described in the previous section, on this dataset. A design note: training (i.e. updating the function in order to continue minimizing the loss) is done on batches of segments, and segments are assigned to batches in a random way, at the beginning of the program. This ensures that we train on various parts of various songs. And we use batch training for efficiency reasons too.
(4) The trained RNN takes as input a sequence $(e_1, \ldots, e_t)$ and outputs a sequence $(e_1, \ldots, e_t, e_{t+1})$. Let this $e_{t+1}$ be the added event to an existing sequence. We then iterate this process and continue adding sound events.

Because the input and targets in an observation are shifted by 1 from actual parts of a song, we are making the RNN learn the next sound event from an actual song.

In order to compose a new song, I take a song which has not been used in the function training, split it into subsequences, pick a random sequence and feed it to the composition function, which then produces the next events.

Decoding is next.

## 6. Decoding

Decoding is trivial from the description of encoding. You can check out the code to see how I do it. Next, specific parameters.

## 7. Specifics

In my trained model, I make the following choices:

(1) Consider only frequent sounds for the melody and chords track of a song. I use a set of 43 sound events for the melody and 38 sound events for the track.
(2) For the RNN training, I split songs into segments of length 256.
(3) For the RNN infrastructure, I use 2 hidden layers, each with 256 LSTM cells.
(4) For composing sounds, I start with sequences of length 64.

That's it. Here's something extra:

## 8. Bonus: a second model

I also build a second model! Here I only train for one track, for the melody, where there are no consecutive notes playing at once. Therefore, here each song can be expressed as a sequence of notes, followed by its duration, followed by the next note etc. I construct the set $S_1$ of note events, the set $S_2$ of duration events. These sets are made disjoint. I consider an event to be either a note event or a duration event. I encode an event in one-hot notation, split the data in the same manner as above, and train the RNN (which now needs to predict just one next event: the note or the duration) according to the standard softmax cross entropy loss. I pray that the RNN is able to understand that a note is supposed to be followed by a duration, and vice versa. It does! The code is available here.

## References

[1] https://www.csie.ntu.edu.tw/~r92092/ref/midi/
[2] https://github.com/jukedeck/nottingham-dataset
[3] http://abc.sourceforge.net/NMD/
[4] http://colah.github.io/posts/2015-08-Understanding-LSTMs
[5] https://github.com/tensorflow/tensorflow
[6] https://arxiv.org/pdf/1308.0850v5.pdf
[7] http://yoavz.com/music_rnn/
[8] http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/]
[9] http://cs224d.stanford.edu/lecture_notes/LectureNotes4.pdf]
[10] https://github.com/kjw0612/awesome-rnn]
[11] http://karpathy.github.io/2015/05/21/rnn-effectiveness/]