Music generation with LSTM network

Edoardo Mortara id. 954198
Natural Interaction a.a. 2020/2021
University of Milan

1 State of the art

The problem of generating musical compositions has been addressed using many different methods.

The data-driven statistical methods typically use Markov-based approaches to model note-to-note transitions [1]. These models fail to take into account the higher-level structure and semantics central to music by concentrating only on certain local temporal dependencies. This type of problem was also found using RNN models [2].
Eck and Schmidhuber demonstrated that higher level temporal structure can be learned using LSTM [3]; in their experiment the harmonic form of the blues was learned by training a LSTM network with various improvisations over the standard blues progression.
Incredible results have recently been achieved by MuseNet (OpenAI) [4] and Magenta project [5]; they both use the Transformer model to learn complex music patterns, which could not be learned using LSTM.

2 Language models and Probabilistic classifiers

The problem of generating Music is a particular instance of Language processing and probabilistic classification.

A statistical language model is a probability distribution over sequences of words. As I will show in later sections, we can map the musical notes to words w_i of a finite size Vocabulary V. This way, given a sequence of length l, we can assign a probability $P(w_1, ..., w_l)$ to the whole sequence.

Neural language models are constructed and trained as probabilistic classifiers that learn to predict a probability distribution $P(w_k|w_{k-s}, ..., w_{k-1}), \forall k \in V$ from a feature vector representing the previous s words.
A probabilistic classifier is a classifier that predict a probability distribution over a set of classes, given an observation . Formally a probabilistic classifier use conditional distributions $P(Y|X)$ to assign probabilities to all $y \in Y$ given $x \in X$. In our case, given the last s notes, we want to predict the next most probable note to be played:

$$\hat{y} = argmax_y P(Y = y|X)$$

3 Prediction with RNN

Recurrent Neural Networks are a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. RNNs use an internal state h_t to process variable length sequences of inputs.
The problem of generating new notes given the last s notes can be modeled as a prediction problem with RNN[6]. An input vector sequence of notes $x = (x_1, ..., x_s)$ is passed through a RNN cell that compute the hidden vector sequence $h = (h_1, ..., h_s)$ and the output vector sequence $y = (y_1, ..., y_s)$. With the output vector sequence we can define a predictive distribution over the possible next note x_{s+1}: $P(x_{s+1}|y_s)$.
As I will explain in the next section, RNN cell suffers from vanishing / exploding gradient problem with Back-Propagation Through Time (BPTT); for this reason, I will use Long Short-Term Memory cell [7].

4 LSTM

In recurrent neural networks, layers that get a small gradient update stops learning; this mean RNN can forget what it seen in longer sequences, thus having a short-term memory.
LSTM overcomes this problem by using different cell architecture (Figure 1). The cell has a 'cell state' c_t that can carry relevant information throughout the processing of the sequence, and the hidden state h_t. Initially the input sequence x_t and the previous hidden state h_{t-1} are used to compute the forget gate:

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$

The result of this function is a vector of numbers between 0 and 1, and is multiplied with the previous cell state; this way we can forget previous information.
The next step is to decide what new information will be stored in the cell state. The cell uses these two function to compute the so called input gate:

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

$$\hat{C}_t = tanh(W_C * [ht-1, x_t] + b_C)$$

The new cell state will be:

$$C_t = f_t * C_{t-1} + i_t * \hat{C}_t$$

Finally the cell defines the output gate as:

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

Using $tanh$ and the sigmoid function we define an output based on a part of the cell state.

The formula for computing the new cell state C_t is crucial for avoiding the vanishing / exploding gradient problem. If we derive C_t with respect to C_{t-1} we obtain:

$$\frac{\partial C_t}{\partial C_{t-1}} = C_{t-1}\sigma'(\cdot)W_f * o_{t-1}tanh(C_{t-1})'$$

$$+\hat{C}_t\sigma'(\cdot)W_i * o_{t-1}tanh'(C_{t-1})$$
$$+i_ttanh'(\cdot)W_C * o_{t-1}tanh'(C_{t-1}) + f_t$$

This term at any time step of the BPTT can take on either values that are greater than 1 or values in the range

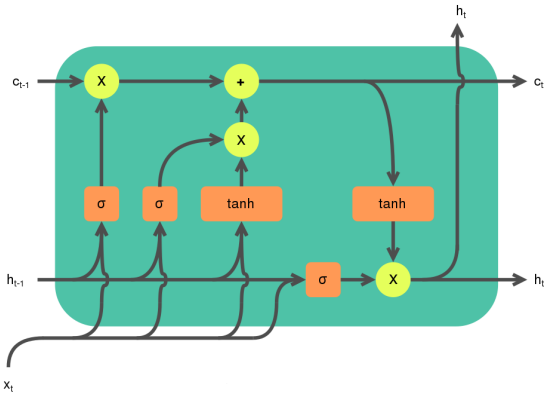

Figure 1: LSTM cell

$[0, 1]$. We can write this partial derivative as the sum of four elements:

$$\frac{\partial C_t}{\partial C_{t-1}} = A_t + B_t + C_t + D_t$$

We define the gradient of the error for some time step k as:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial C_k}(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}})\frac{\partial C_1}{\partial W}$$

If we plug the formula obtained before we get:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial C_k}(\prod_{t=2}^{k}[A_t + B_t + C_t + D_t])\frac{\partial C_1}{\partial W}$$

The cell state gradient is an additive function made up from the four elements denoted A(t), B(t), C(t), D(t). This additive property enables better balancing of gradient values during back propagation. The LSTM updates and balances the values of the four components making it more likely to avoid vanish gradient [7].
In vanilla RNN (Figure 2) the error term gradient is given by the formula:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial h_k}\frac{\partial h_k}{\partial C_k}(\prod_{t=2}^{k}\sigma'(W_{rec}C_{t-1}+W_{in}x_t)W_{rec})\frac{\partial C_1}{\partial W}$$

when W_{rec} is small we get vanishing gradient problem; when it's large we get exploding gradient problem.

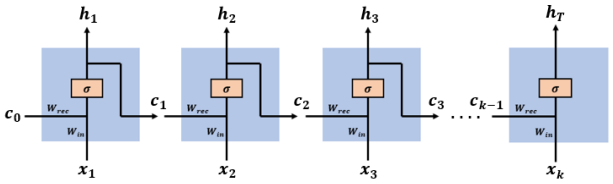

Figure 2: Unfolded RNN

5 Data sets

For training the network I will use sequences of notes and chords extracted from MIDI files consisting of a single instrument, like piano. I can easily extract notes and chords using Music21 toolkit (*web.mit.edu/music21/*); I transform a Music21's note in its pitch string notation, for example:

$$< music21.note.Note\ E >\rightarrow E$$

A Music21's chord is a list of notes, so I create a list of the normal order (see Music21 chords documentation) of each note separated by a dot:

$$[c, e, g] \rightarrow 0.4.7$$

The encoded notes and chords are put into a sequential list, that will serve as the input of the network. The output of an input sequence $[element_1, ..., element_s]$ is the next note or chord $element_{s+1}$.
The input sequence must be transformed from string-based categorical data, to integer-based numerical data. The mapping function $f : elem \rightarrow number$ uses a dictionary to map each note/chord to an integer. The domain of this mapping function will be the union of all the encoded notes and all the encoded chords. This mean that the network will not learn new chords, because a chord is encoded in a single number. Another very important aspect to underline is that this encoding does not take into account the dynamics and duration of the single note/chord. The input sequence must be normalized in the [0,1] range:

$$\frac{[el_1, .., el_s]}{|Vocabulary|}$$

where the denominator is the number of unique notes and chords in our Domain. The output is one-hot encoded using "np_utils.to_categorical()". The network will output a vector of prediction probabilities; I take the most probable prediction's index and transform it back to a note/chord using the inverse mapping function $f^{-1} : number \rightarrow elem$.

6 Implementation

The LSTM network used in this experiment is composed by 3 LSTM layers and two Dense layers. Figure 6 shows that the input sequence given to the network is of length 100. This mean that the network learn to predict the next note/chord using the last 100s. The output of the network is defined by the last Dense layer; its size must be equal to the number of unique notes and chords in our Vocabulary Domain. This assures that the output of the network will map directly to our classes.
Each LSTM layer uses the Keras default $tanh$ activation function and sigmoid recurrent activation function; each layer has a hidden state vector h_t of length 512, because the number of unique notes and chords in the MIDI files used in my experiments are less than 500; the length of the h_t vector influences the ability of the network of recognising and encoding long-term patterns. After the three LSTM layers I decided to add A Batch Normalization layer as suggested in [8]; this layer gives benefits to the network in terms of training time and model performance. I decided to add two Dropout layers [9], because LSTM can easily overfit training data, reducing the predictive skill.

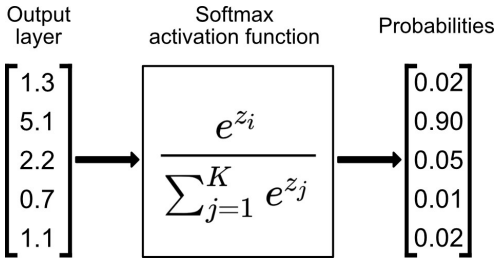

Figure 3: Softmax activation function

The first Dense layer uses a Rectified Linear Unit activation function to add non-linear boundaries, while the last Dense layer uses softmax. This function is crucial

for a LSTM network, because we want to predict the next more probable note/chord to be played. Softmax permits to take a vector of values and return a new normalized vector in [0,1] (that are probabilities) (see Figure 3). As explained in the last section, I take the most probable prediction's index of this normalized vector and transform it back to a note/chord using the inverse mapping function $f^{-1} : number \rightarrow elem$.

The network is trained using Stochastic Gradient Descent with batch size equal to 128. To calculate the loss for each iteration of the training I use the categorical cross entropy; this loss function is obtained by using the softmax activation followed by a normal cross-entropy loss:

$$softmax(S)_i = \frac{e^{S_i}}{\sum_j^C e^{S_j}}$$

for $i \in [1, C]$ and $S = (S_1, .., S_C) \in \mathbb{R}^C$. The cross-entropy loss is defined as:

$$CE = -\sum_i^C t_i \, log(softmax(S)_i)$$

In our case the C classes are the Vocabulary Domain of all notes and chords; when we compute this loss the t variable is the one-hot vector output described in the last section. This mean that there is only one element of the t vector which is not zero. We can rewrite the formula as follow:

$$CE = -log\left(\frac{e^{S_p}}{\sum_j^C e^{S_j}}\right)$$

where p is in the index of the t_p element equal to 1. The loss terms coming from the negative classes n ($t_j = 0$) are computed as:

$$\frac{\partial}{\partial S_n}\left(-log\left(\frac{e^{S_p}}{\sum_j^C e^{S_j}}\right)\right) = \left(\frac{e^{S_n}}{\sum_j^C e^{S_j}}\right)$$

while the positive class p is:

$$\frac{\partial}{\partial S_p}\left(-log\left(\frac{e^{S_p}}{\sum_j^C e^{S_j}}\right)\right) = \left(\frac{e^{S_p}}{\sum_j^C e^{S_j}} - 1\right)$$

The optimizer used for training is the Keras default *RMSprop*:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)\left(\frac{\partial C}{\partial w}\right)^2$$

$$w_t = w_{t-1} - \frac{\mu}{\sqrt{E[g^2]_t}}\left(\frac{\partial C}{\partial w}\right)$$

where $E[g^2]_t$ is the moving average of squared gradients for each weight at time t; $\frac{\partial C}{\partial w}$ is the gradient of the cost function with respect to the weight; μ is the learning rate; β is the moving average parameter set to 0.9.

7 Results

Figure 4: Model Loss

I trained the network with contemporary classical music (see *www.classicalarchives.com/midi.html*). Figure 4 shows the model loss with respect to the training epochs. In general this network reaches ≈ 0 loss with 150 epochs.

If the network is able to predict the next note/chord given the last 100s, it can also create new notes sequences; first of all, I generate the element x_{s+1} by giving to the network the sequence $[x_1, ..., x_s]$; now if I give to the network the sequence $[x_2, ..., x_{s+1}]$, it will generate the element x_{s+2}. If I repeat this routine k times I will obtain the following sequence:

$$[x_1, ..., x_{s+1}, ..., x_{s+k}]$$

This way I can generate new musical compositions of k notes/chords and save them as MIDI files.
Figure 5 shows the score of a sequence generated by the neural network. From a musical point of view the melody is pleasant to listen and the harmony is consistent. The chords progression does not present dissonant elements, indeed very structured.

References

[1] Parag Chordia, Avinash Sastry, and Sertan Şentürk. Predictive tabla modelling using variable-length markov and hidden markov models. Journal of New Music Research, 40(2):105–118, 2011.

[2] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. arXiv preprint arXiv:1206.6392, 2012.

[3] Douglas Eck and Jurgen Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks. In Neural Networks for Signal Processing, 2002. Proceedings of the 2002 12th IEEE Workshop on, pages 747–756. IEEE, 2002.

[4] Payne, Christine. "MuseNet." OpenAI, 25 Apr. 2019, openai.com/blog/musenet

[5] Music Transformer: Generating Music with Long-Term Structure,Huang, Cheng-Zhi Anna and Vaswani, Ashish and Uszkoreit, Jakob and Shazeer, Noam and Hawthorne, Curtis and Dai, Andrew M and Hoffman, Matthew D and Eck, Douglas,arXiv preprint arXiv:1809.04281,2018

[6] Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

[7] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation, 9(8):1735–1780, 1997.

[8] Whang, Jay, and A. Matsukawa. "Exploring Batch Normalization in Recurrent Neural Networks." Stanford Center for Professional Development. Available online: https://jaywhang. com/assets/batchnorm_rnn. pdf (accessed on 28 September 2020).

[9] Cheng, Gaofeng, et al. "An Exploration of Dropout with LSTMs." Interspeech. 2017.

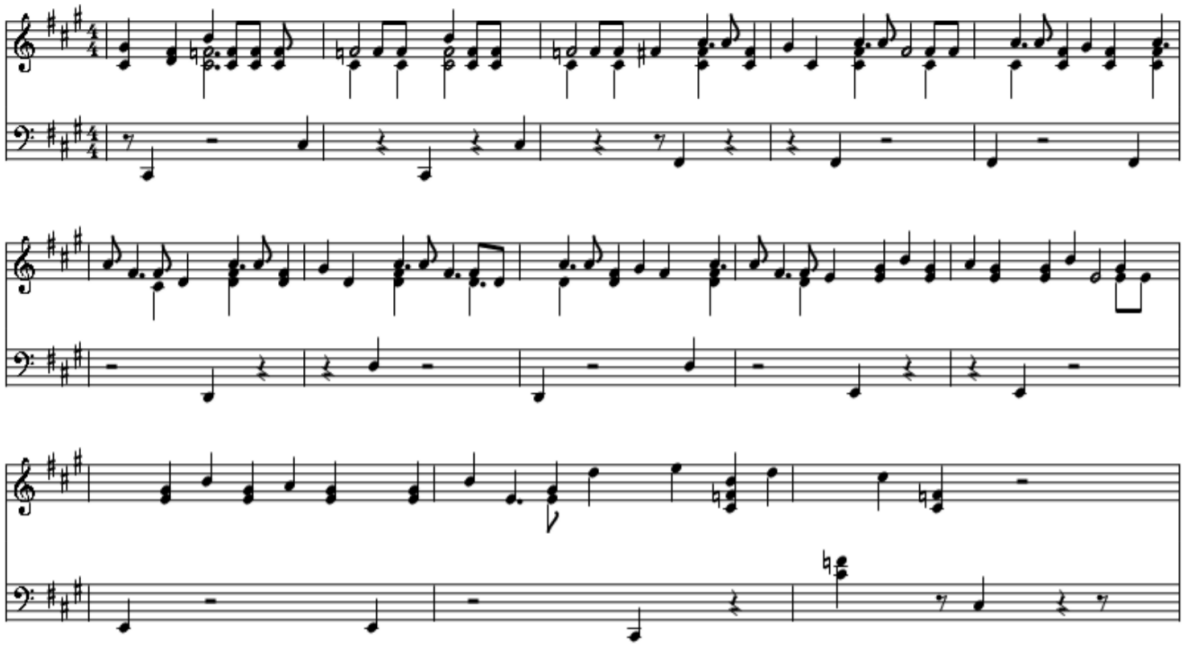

Figure 5: Score of a sequence generated by the neural network

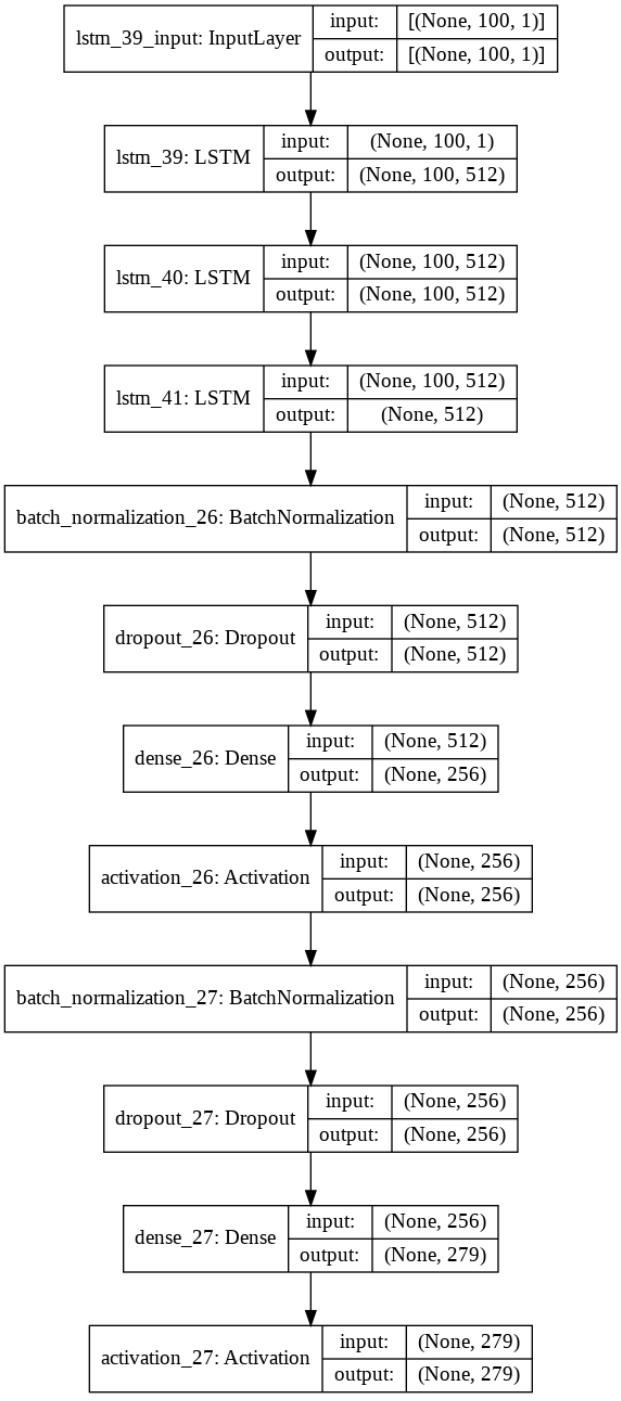

Figure 6: LSTM Network architecture