Documentation
semester project – automatic composer

Marek Židek, 2nd grade on MFF UK

1. User doc

1.1. Project Info

1.1.1. Purpose

The project is meant to compose unique pieces of classical music automatically. However, the user can input any style of music in a midi format. The number of samples should be over 100. Input music can be polyphonic and can have more instruments, but the resulting music is for the piano.
The resulting music is in midi format with configurable length and is in 4/4 measure.

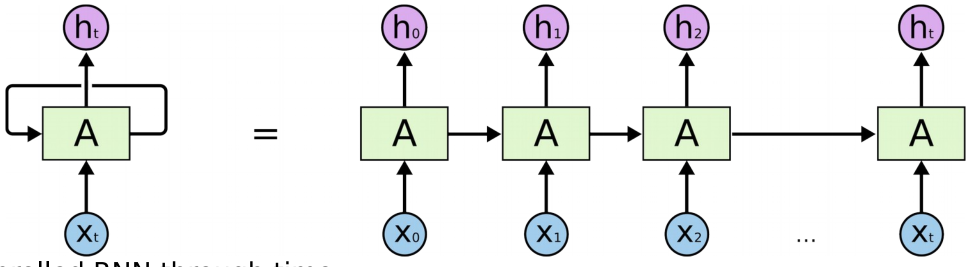
The topic interests me, because I always wanted to listen to some genre which I liked, but listening to the same songs started to be boring soon, so I had to look for others. This way, there can be an infinite source of new music parts. I also have some interests and knowledge in music, because I play the violin for about 15 years.

1.1.2. Idea

I wanted the resulting music parts to have some structure, to know when the part starts, when it ends, what main theme does it have or that it's good to sometimes play the same notes as played at the beginning of song throughout the song in different music keys(e.g. C major,...), variations or even disharmonies to not bore the listener.

Neural networks were also a reason to start this project as I was interested in them and this was a good use for them as for predicting the next notes to be played based on current notes played. However, the standard feedforward networks couldn't take the song structure as whole into account when predicting.

Program uses neural network called LSTM to learn itself how to compose. I suppose the reader has a basic knowledge about neural nets and knows about weights, activation functions, input, output and hidden layers, forward and backpropagation. LSTM stands for long-short term memory and it's a type of RNN – recurrent neural network. RNN neurons have recurrent connections with themselves from the previous time so the neurons in hidden layers get input not only from previous layer, but also from previous time.

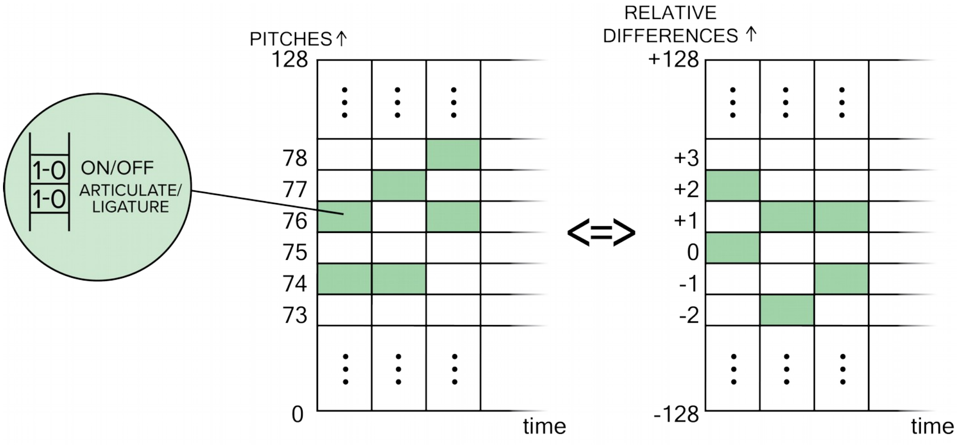


Unrolled RNN through time, (http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-unrolled.png)

LSTM works the same way, but keeps a “memory” the entire time, not only one timestep back and passes it along with the previous output. The memory is updated every timestep, some information is added and some is removed. These networks work really well in predicting long sequences, such as texts, speeches, videos or music, because they remember information for long time periods. More on LSTMs on colah's blog. Basically, the network predicts the probability for every note in a midi bounding to be played and the probability to be prolonged from the previous timestep or rearticulated all in one output vector.

1.1.3. First steps

I searched for some related works, D.Johnson's and D.Eck's and found them both working with network that gets a note as input and some other relevant data and outputs the probability to be played. I wanted the network to catch the "essence of music" which I believe are the intervals between notes at current time (chords) and between past and now (melody), because then the music can be transposed along notes and still sound good. So I designed the network to take a vector of relative differences between previous note and the current (counted from the middle of the vector) and because the kind of music I wanted to work with was polyphonic, I had to choose what would be the relative value to which then interpret the next predicted output. The idea of taking a mean was clearly bad, because in music, even semitone out causes disharmony. So I chose the highest pitch note to be it as it often makes the melody part of the composition.



with initial relative value of 74

Then I implemented functions to convert midi to that relative difference matrix and tested it. Afterwards came the part where I needed to fully understand how the LSTM networks work and how to implement them so I researched I lot of github code and asked the gitters questions about it. And then first test were made.

1.1.4. First tests

When thinking about the idea, I was worried that the generated music would get out of the scope of midi pitch boundaries, meaning that the next predicted note should be like +5 semitones many times in a row. However, I thought that the recurrent neural network would remember how many times the melody usually goes higher in a row and how many times it goes lower. Over two nights of training on classical music, this was not achieved. The generated music was never really out of tone or disharmonic, but jumped across the pitches and went too high or too low to be played in a matter of seconds.

So I reimplemented the network to take the direct vectors representing the pitches of notes and predicting the next and added binary representation of beat to the input because the compositions didn't really get the feeling for when to play chords. I thought this way it could learn all the common chords as the input and output vectors hold all possible notes so the neurons will take them into account.

1.1.5. Final configurations

The final network gets input of length 78*2 + 4, 78 is span of notes(cutted off of too high or too low pitches from 128 midi representation), times 2, because we need binary indicator not only for note play, but also for holding the previous note or articulate it and + 4 for beat representation.
Output vector is of length 78*2 (without the beat) and is activated using sigmoid function and the values at note positions are probabilities that the note is played or articulated.

The cost function is cross-entropy, which negative logs the values of how much close were the probabilities to the actual output on scale of 0 to 1 and then sums them.

The Number of hidden layers is 3 and the number of neurons is
[300, 300, 250] all of LSTM type.

The network uses dropout to counter overfitting the training data and its value is 0.5, meaning that half of the weights are zeroed out in every training epoch and in multiplied by 0.5 in generation.

My training data were about 150 classical music parts which I found free on the internet in midi format.

I used online training method that updates the weights after every input, but later I also implemented batch training, which is possible to switch to(uncomment in code). However, both should converge to the same configurations. I found the batch training results more consistent as they don't get premature about updating the weights towards some song-dependent fragile features. Both methods outputs are in the project folder.

1.1.6. Fist results

The resulting music sounds interesting and can be found in “output” folder. What I think is that it really got the sense of rhythm and the chords sound pleasant. Nevertheless, there are some issues with silent moments and then disharmonies, which I cause with my treatment to silent moments. I think the network learned correctly, that sometimes there's an empty bar or bars in every music part, but it didn't learned the dynamics(***p, f***) of how to prepare the listener for it and how to start again, so these transitions sound very sudden and not what the listener would expect. The problem is that every note in the predicted music has the same velocity of 64-mezzoforte. I tried to fiddle a little bit with output vectors and power every probability with a number little less than 1. That way I force the probabilities for note plays to be higher to counter those silent moments and make the music more lively, but it also causes that notes which don't fit in are played.

1.1.7. Future work

I would definitely like to implement dynamics to generated music, because the same velocity for all the notes makes even a human musician playing a midi controller sound unnatural. I think that maybe adding a mean of all velocities from given time into input vector and then predicting it for output and distribute it with slight random bias to to all predicted notes might work well. I also think that this could be done algorithmically analyzing the empty bars and adjusting the velocities.

Another thing might be going back to the relative difference idea and improving the network adding some data to input or make some classes for each octave and increase the cost if the music tends to jump around or go out of midi scope. That would require updating the input so it takes maybe like one measure back note octave classes into consideration.

1.2. How to use

1.2.1. What do you need

To start to the program, user needs to have python 2.7 interpreter installed along with some dependencies which are specified in **2.1.** programmer documentation. The script to install is there for all debian derived linux distributions and the program is meant to run and is tested on linux. However python is multiplatform and I hope those libraries and dependecies can be found on the internet for other platforms.

The user will also need a large collection of data in midi format and put it in a folder called “music” in the project folder.

The output data are generated in the number of five parts with configurable length named example0-4.mid if the user wants to generate, or with name afterXepochs.mid, if the user is training the network, both stored in “output” folder. The learned configurations are stored in “params” folder.

1.2.2. How to start

The program is started by navigating to the project folder and executing `python main.py t|g 80` where the first argument is `t` for training and `g` for generating and the second argument is the number of 4/4 bars, for example `80` bars is a minute and half.

Even for generating, there must be some data in music folder, because the first timestep input to network is from this data.

If the user wants to use his/her own trained params and not that I provided, add the path in `main.py` – line 16

```
model.config = pickle.load(open("PATH", "rb"))
```

2. Programmer doc

2.1. Dependencies

I used a python library theano for working with multi-dimenstional arrays and better speed & stability optimizations along with adadelta optimizer.
And theano-lstm for easier working with LSTM related operations.

There is a script in project folder to install everything needed called `dependencies.sh`:

```
sudo apt-get install python-dev python-pip python-nose python-numpy python-scipy
sudo pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
sudo pip install python-midi theano-lstm
```

2.2. Project files and their purposes

Project is divided into files: `main.py, lstm.py, midi_to_[difference]_matrix.py, train.py`

2.2.1. main.py

Starts the program, gets user arguments and prepares the model with given configurations, then loads the music data and starts to either train or generate.

Also contains `generate` function, which takes the first played notes from music data and starts predicting with function `genFunction` defined in `lstm.py`.

2.2.2. lstm.py

Initiates the model with given configuration and creates a symbolic computational graph for theano to work with for functions `trainingFunction` and `genFunction`.

`setTrainingFunction` symbolically defines all the structures needed and operations to be done by theano: input and output matrixes as inputs for the training function, step function to execute on the input matrix, scan function as a form of looping in theano, initial states from the -1 time in recurrent relations.
Then compares the resulting matrix with the real output matrix using the cross-entropy function to determine cost, update the weights and returns cost.

`setGenFunction` defines the starting vector from which will be predicting, length of song and keeps track of current time for beat generation which is an easy function creating binary representation of 8 timesteps (as my smallest note is eighth note) in a measure. Step function for forward propagation also choses the notes to be played by generating random numbers and comparing to note probabilities and then passing this output as an input to the network in time + 1, so the scan function doesn't work over sequence but over it's own output.

2.2.3. midi_to_matrix.py

There are two of these files, however as stated in user documentation the difference matrix idea didn't go quite well, so the midi_to_matrix.py is used and it just converts midi file into matrix with vertical vectors representing note pitches and holding notes from previous time with granulation of eighth notes. It also does the reverse operation, recreating the midi file, but with fixed values of velocities.

2.2.4. train.py

Contains methods for loading and adjusting the data for the network and a function `train` for operating with configurations, saving them and generating sample parts.