Summary on the Auto Composer Module

Hanrui Zhang, Geek 30

November 12, 2014

1 Introduction

The Auto Composer (AM) is a module of the Scoreur Project, aimming at a completely automatic implementation to mimic existing melodies as well as to generate new ones. It bases on ideas from machine learning, uses Hidden Markov Model to simulate the logic of sequences of notes, which is quite efficient both theoretically and pratically.

2 Hidden Markov Model

Hidden Markov model (a.k.a HMM) is a method to model arbitrary random procedures. It assumes that the procedure to be modeled is generated by some hidden Markov chain, each state of the Markov chain induces a distribution on all the possible outputs. At each step, the Markov chain itself evolves, and produces an output recording to the distribution generated by its current state. It is obvious that HMM is an outstanding choice to handle problems related to sequences over time, e.g., rhythms.

The model could be trained by adjusting the model parameters, including parameters of the Markov chain, distributions induced by each state, and the initial distribution on the states of the Markov chain. Training is implemented by iteration, according to some formulas proved by the method of Lagrangian Multiplier, whose proof can be found in any related article. All the calculation can be done using basic recursional relationship between quantities.

Classification can also be done with HMM, by calculating and judging the probability of a string to appear as the output of the model among all possible strings of the same length. The probability can be calculated using dynamic programming.

3 Methods Used to Compose

The idea of the AM module is to model existing rhythms by using them to train an HMM, and then generate new rhythms from the trained HMM. We translate a piece of rhythm into a sequence of notes (with the same lasting time), use the sequences to train a discrete time HMM. The hidden states of the HMM are supposed to contain information of orders and lasting times of the notes, as well as details of chords and music bars.

Once the training is done, the idea to generate rhythms becomes quite simple: let the trained model evolve itself, the output of a particular length is then our desired rhythm. In fact, it is also natural to pick the sequence with the maximal possibility to be generated, which can be done using dynamic programming. It is, in some sense, the most similar rhythm to the original ones. However, there is only one piece of such rhythm. As we may need multiple different sequences, I chose to generate such sequences randomly.

Finally, the AM module communicates with a web server by *Xiaoqi Chen* to play the generated rhythms.

4 Parallelization and Scaling

As training on multiple strings are somehow independent, it can be done parallelly. In my implementation OpenMP was chosen, for its simplicity, and that I'm so lazy. Idealistically, using c cores simultaneously will reduce the time needed by c times.

For a sufficiently long string, the probability that it appears will be so small that exceeds precision limit of any possible machine. We scale the temporary variables here to avoid precision issues. In short, when we calculate the possibility of its prefix, we normalize this possibility and record the logarithm of the coefficient. When we are done, logarithm of the probability could be calculated by simply summing up the records at each step.

5 Efficiency of the AM

HMM, in my opinion, is a pretty fast model. Suppose that we have a HMM model with n hidden states and m possible outputs. A single iteration of training on a string of length t takes $O(tn^2)$ time. In practice, the parameters often converge after about 100 times of iteration. In my implementation, since we are to generate short pieces of rhythms, which does not contain much complicated structures, n is set to 100, and the length of the working part of any song is not likely to exceed 1000, which means that we can finish almost any training on a

single song in 10 seconds. When there are multiple songs to train on, we can use multiple threads to efficiently reduce the required time.

When we have done the training part, it seems quite easy to generate rhythms using the trained HMM. To get a piece of rhythm with a length of l, we need only $O(l(n+m))$ time, which is often much smaller than the time used to train the model. Still, we can use multiple threads to generate multiple pieces of rhythms simultaneously.