Estimating High-Dimensional Temporal Distributions Application to Music & Language Generation

Anonymous Author(s)
Affiliation
Address
email

1 Introduction

Generating polyphonic music is a complex and high dimensional task. At each time step of a given song, many combinations of notes could potentially be played. Obviously, the distribution of potential notes evolves over time, depending on the notes that have actually been played. In other words, music has a fundamentally sequential nature - just like language: at each time step, there is a certain distribution of notes that could be played. This distribution is unknown - our goal is to estimate it! - and evolves over time.
The sequential structure of music and language makes music generation very analogous to letter or word generation.

Therefore, the recent advances in natural language generation - especially RNNs - should lead to significant improvement in music generation. Interestingly, not much work has been done on applying the latest architectures and optimizations of NLP to the task of music generation. Our goal is to implement these improvements - especially LSTMs, attention, ReLu, dropout - to outperform the music generation models presented in ICML 2012. Then, we apply this model to create fun demos - e.g. how Bach would complete Beethoven's song Finally, we tweak our music generation model to perform a language task - next word estimation in a sentence.

2 Previous work

Previous work on music generation include a ICML 2012 paper. It compares several music generation models. The most performant models combine neural networks and distribution estimators:
- the distribution estimator models the distribution of notes that could potentially be played at each time step
- the RNN determines the parameters of these distribution estimators. The RNN is able to model the evolution over time of the distributions - and captures information about the impact of a given note on the distributions of future notes.

However, this work was done before the recent improvements in deep learning. Complex RNN architectures (LSTMs, GRUs, attention...) and recent optimization techniques (dropout) should improve the existing baselines. Also, previous work has focused on relatively small datasets - a few hundred songs. Progress in computation power should enable us to run these models on much larger datasets - hundreds of thousands of songs - in order to make the most out of our models.

3 Data set

The ICML 2012 paper on music generation runs baselines on 4 music datasets. We decided to focus on one of them - JSBChorales - containing 382 piano interpretations of Bach's chorales.
After reproducing and improving the ICML results on the relatively small JSBChorales dataset, we build a fun demo that combines different musicians: we train the model on 1,200 classical piano songs from various musicians. Finally, since the model captures information on these datasets fairly quickly - it starts overfitting after a few thousand iterations - we train it on the largest music dataset we could find, 150,000 drum tracks.

All songs and tracks are initially MIDI files. We convert them to a Tensorflow readable format by keeping track of the notes played at each (discrete) time step.

4 Models

The music generation task consists in estimating the distribution of notes that should be played at a given time-step, based on what was played previously. To do so, we combine a RNN with distribution estimators - one per time step - as shown in Figure 1. The outputs of the RNN determine the parameters of the distribution. In other words, the RNN captures and carries information about the temporal evolution of the distribution of notes.

In the rest of the section, we use the same RNN architecture. Instead of using ICML's simple RNN, we use a multi-layer LSTM. Attention captures long time dependencies and dropout improves the learning. Gradient clipping and learning rate decay help our batch gradient descent converge.

We consider two different distribution estimators. We first consider the simple Bernoulli estimator, before analyzing the more complex Neural Autoregressive Distribution Estimator (NADE).

There are two distinct phases. At training time, we estimate the distribution of notes that should be played at time step t - and compare them to the ground-truth notes that are actually being played at that timestep. At sampling time, we estimate the distribution of notes that should be played at time step t and generate notes from these distribution.

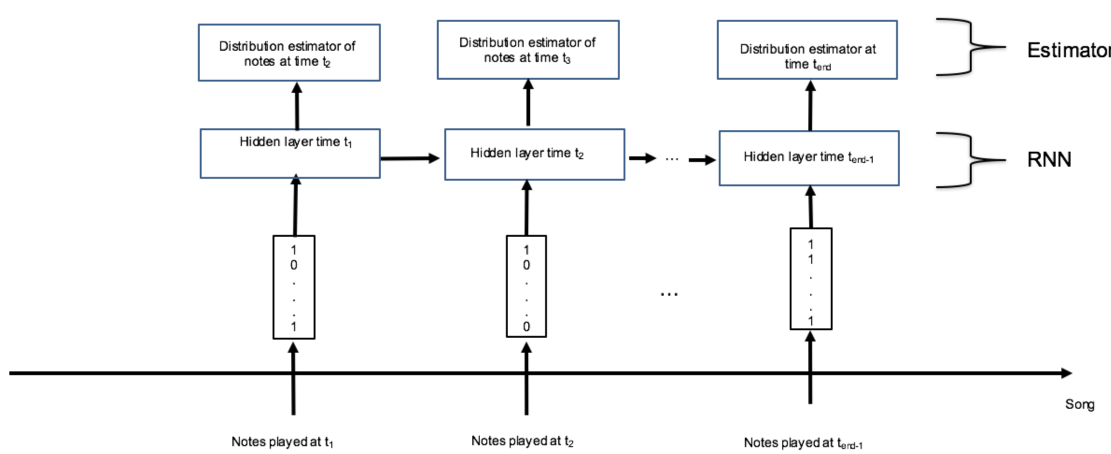

Figure 1: Baseline of our model

4.1 RNN-Bernoulli

In the RNN-Bernoulli model, the outputs of the RNN at time t represent the unnormalized probabilities of the candidate notes of time t+1. Each note is predicted independently from each other. Because of this simplistic assumption, we do not expect RNN-Bernoulli to reach breakthrough results. Instead, we wish to reproduce the results of ICML 2012 and focus on another type of model, which is much more promising: RNN-NADE.

4.2 RNN-NADE

NADE is a distribution estimator that is able to capture complex multi-modal distributions of notes at each time step. In this model, the vector x_0 is the sequence of notes played at time step t.

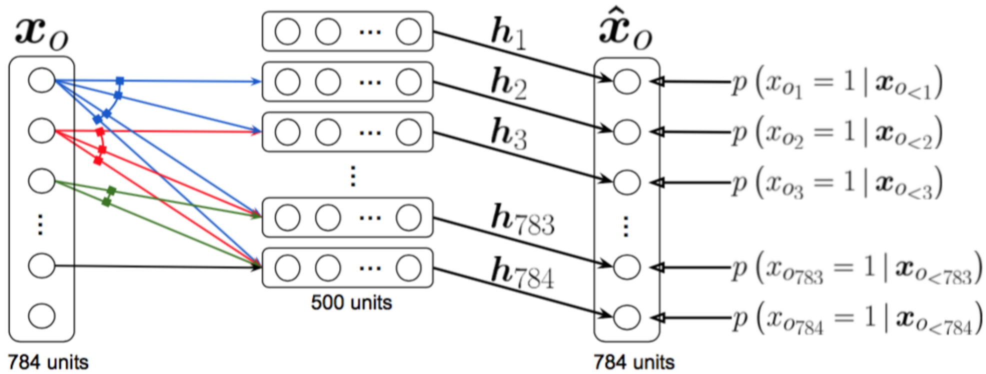

Figure 2: Illustration of a NADE model. In this example, in the input layer, units with value 0 are shown in black while units with value 1 are shown in white. The dashed border represents a layer pre-activation.The outputs x_O give predictive probabilities for each dimension of a vector x_O, given elements earlier in some ordering. There is no path of connections between an output and the value being predicted, or elements of x_O later in the ordering. Arrows connected together correspond to connections with shared (tied) parameters.

From each conditional probability $p(\vec{v}|\vec{v}_{<i})$, where $\vec{v}_{<i}$ is the vector of probability of the i-1 first notes, we compute the probability of observing the sequence of notes x_0 by assuming that $p(x_0)$ can be written as:

$$p(x_0) = \prod_{i=1}^{D} p(x_i|x_{<i}) \tag{1}$$

4.3 Training procedure of the RNN-NADE

To train our model, we feed the RNN with the sequence of notes played at each time step and compute the output of the RNN. Then we use a fully connected layer to compute the parameters b(t) and c(t), for each time step - they determine the NADE estimator at time t. Given a sequence of notes, the estimator NADE will generate the probability vector corresponding to the input. In other words, the output of NADE will be a vector in which each component of the probability vector will represent the conditional probability of playing this note given what was already played.
The model is trained so that the likelihood of the ground-truth notes of time step t+1, under the estimator NADE of time t, is maximized. Indeed we want NADE of time step t to give a high likelihood to the next input vector. Thus, we train our parameters to minimize the following loss,

called negative log-likelihood:

$$loss(t) = -log(Pr[input_{t+1}] \tag{2}$$

Where $PR[input_{t+1}]$ is obtained by multiplying all the component of the output vector of NADE_t in order to construct $Pr[input_{t+1}]$ from equation 1.

Then we update our parameters by applying a mini-batch stochastic gradient descent, with 128 songs per batch.

$$\theta_{new} = \theta_{old} - \alpha \nabla_\theta loss \tag{3}$$

4.4 Sampling procedure from the RNN-NADE

During the sampling part, all the estimators of the distribution of notes have been trained. Now we would like these estimators to draw samples in order to generate time steps and thus, generate music. To do so, we sample a time step from the NADE and use this sample as the next input to our RNN. In doing so, we generate samples that follow the distribution of the notes estimated during the training phase.

To generate a sample from a NADE, we use the following procedure:

1. Start with a input vector of zeros. This is not unreasonable because NADE only computes conditional probabilities given the notes already played at the current time step, so the model is not influenced by elements whose index is larger than the note we are trying to predict.

2. Let's assume that we have already decided which notes were to be played among the first i-1 notes. we are now trying to compute the probability of playing the i^{th}, given what was already played. In other words, the input vector is already filled with i-1 notes. The input vector of i-1 notes is used to compute the (conditional) probability of playing the i^{th} note. To determine whether a note should be played, given this conditional probability, we sample from a Bernoulli distribution whose parameter is equal to the conditional distribution of playing the i^{th} note given the previous note. The result of this sample determines whether or not we play the i^{th} note.

3. We iterate the procedure until we have a full sample for the given time step (128 notes for a piano for example).

Note that performing this sampling many times could lead to many different samples.

5 Evaluating models

To evaluate the performance of our models, we consider the following qualitative metric: how well does the generated music sound? However, we would much rather have a quantitative metric. Previous work on music generation focused on two quantitative metrics: next-time step prediction accuracy and negative log-likelihood of the next time-step under the model.

1. Next time step accuracy: Given the distribution estimator of time step t, we draw multiple samples, compare each of them to the ground-truth and average their accuracy score. It is key to generate several samples (instead of 1) since the distribution of notes is likely to be multi-modal and a single sample could not reflect this.

2. Log-Likelihood: Since we want the estimator at time step t to estimate the distribution of notes of time step t+1, the next time step in our RNN should be a likely outcome of our distribution estimator. In other words, the likelihood of the notes of the next time step under our model should be high. Thus, we minimize the negative log-likelihood of the next-time step under our model.

6 Results

All models were trained to minimize the negative log-likelihood metric. We then computed the accuracy of our models, to compare it to the previous baselines. Demos for each model are available in this directory

6.1 Dataset JSBChorales

First, we trained our model on the JSBChorales dataset to compare the performance of our model to the 2012 ICML publication. The following graph shows the evolution of the loss for the training set and the validation set, for a given set of parameters.

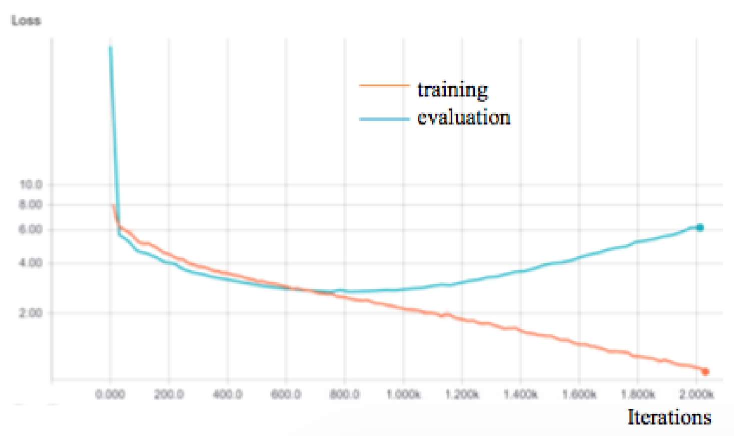

Figure 3: Evolution of the loss on the training set and the validation set

Figure 5 points out that our model overfits after 800 iterations. The loss on the training set keeps decreasing with the number of iterations, when the loss on the evaluation set starts increasing. For the rest of this section, all trainings were interrupted when the models start overfitting - ie when the evaluation loss increases for more than 50 iterations.

Our model presents many tunable hyper parameters. For each set of hyperparameters, we determine the optimal number of iterations and measure the performance. Figure 4 displays the different learning curves obtained by changing the size of the hidden layer, the dropout rate, the attention window length, the initial learning rate and the learning rate decay. Unsurprisingly, the model with most parameters (purple curve) reaches the best evaluation loss.
Once we have chosen the optimal set of hyper parameters, we evaluate the model on test set and compare the results to the ICML publication.

In Figure 5, we compare the result of our models to the ICML models on two different metrics: log-likelihood and accuracy. Since we trained and tuned our model to minimize the log-likelihood we will primarily focus our attention on this one. First, we can note that our RNN-Bernoulli model achieves a slightly lower performance than the ICML model. Two reasons might explain this result. First, the RNN-Bernoulli has a fairly simplistic distribution estimator, that might level the performance of the global model in spite of the improvements made to the RNN (e.g.: attention, dropout). Also, the RNN-Bernoulli model was not fine-tuned. Instead, our goal was achieve similar results to the ICML publication, and then focus our attention and computation power on more elaborate models like RNN-NADE.

Samples generated by this model are available here.

As expected, the RNN-NADE with attention and LSTM model outperforms all ICML models on the log-likelihood metric. This model was fine-tuned as shown on Figure 4. It achieves a log-likelihood of -2.7, compared to the ICML score of -5.56.

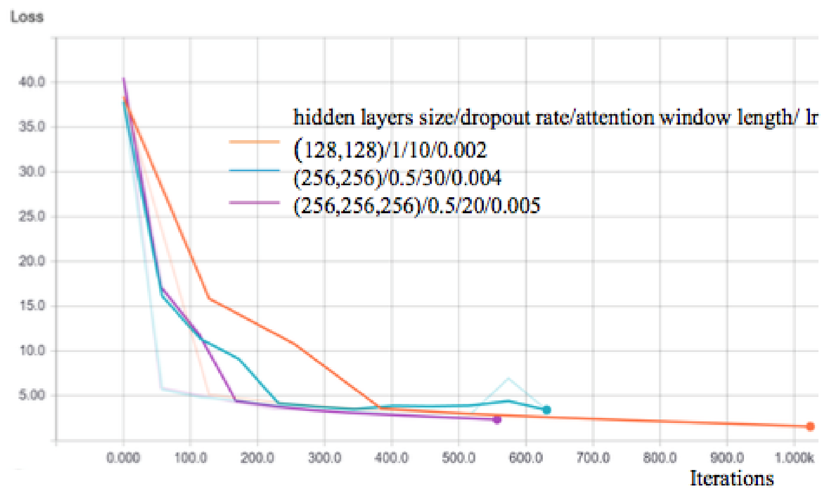

Figure 4: Fine tuning: Evolution of the loss on the training set and the validation set, for three sets of parameters

	Log-Likelihood	**Accuracy(%)**
RNN-Bernoulli ICML 2012	-8.71	28.46
RNN-Bernoulli	-9.16	
RNN-NADE ICML 2012	-5.56	32.5
RNN-NADE	**-2.7**	21

Figure 5: Performance comparison

Now that we tuned our model to perform well on the JSBChorale dataset (Bach dataset), we wanted to generate of fun demo. We trained the model on a diverse set of classical piano songs and fine-tuned it on Bach piano songs. Then, we asked the to complete the first nodes of a Mozart's song, by sampling sequences of notes from our estimators - to continue the song "like" Bach would do it. From figure 6, we see that our model doesn't perform well on this task. We analyzed our error by decomposing it in several contributions. As shown on Figure 8, the number of missed notes penalizes our model. From a qualitative point of view, the results are definitely not great to listen to, although some sequences of notes actually respect the melody. Several explanations might explain this behaviour: the JSBChorales is a relatively small dataset, if we had a bigger dataset for piano, we could have improved the learning stage and trained estimators that better represent Bach style.

Samples are available here.

6.2 Drum Dataset

As mentioned above, our model suffers from the small size of the JSBChorales dataset. To overcome this problem, we increased the number of training examples by working with the largest dataset we could find - a drum dataset. From figure 8, we see that our model is less likely to over-fit on the drum dataset. We trained our model on this dataset and generated drum samples out of it. The results are actually quite pleasant to listen to! The model captures well the notion of rhythm.

Link to samples.

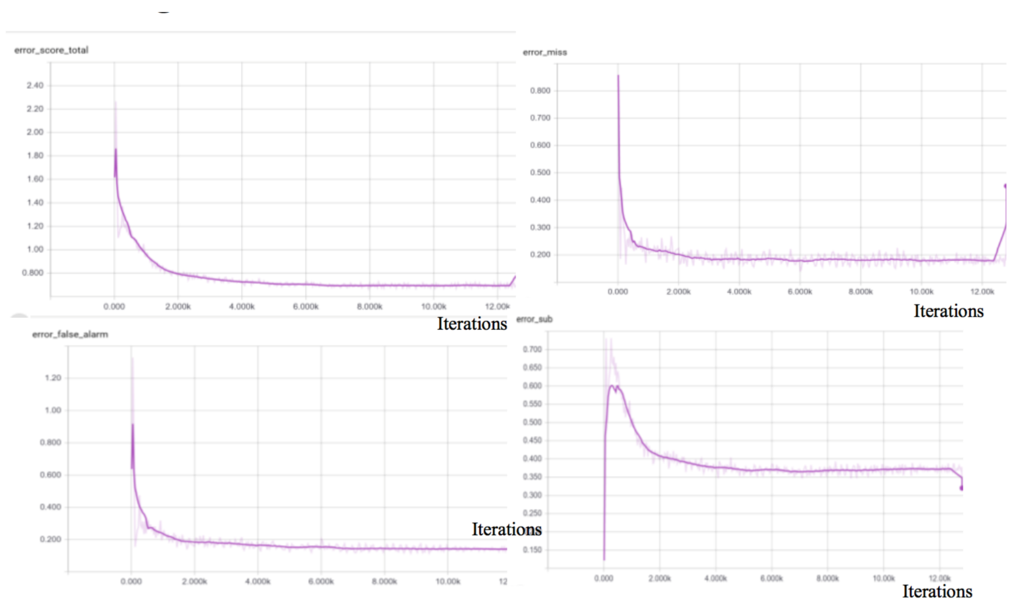

Figure 6: Evaluation of the accuracy of the prediction of our model to complete a song from a unseen author. Global error (top-left), average of missed notes (top-right), false positive(bottom-left) and misplaced note(bottom-right)

Figure 7: Evolution of the loss on the drum set

7 From music to language generation

There is a major difference between the task of generating language and music because in the case of music. In the case of music, input vectors are binary multi hot-vector extracted from midi files. But in the case of language, the input vectors are continuous vectors. Our RNN-NADE model is designed to work with binary vector representation and not with continuous vectors ... Nevertheless, the RNADE [3]has been proposed as an extension of the NADE model. The RNADE relies on a multi Gaussian distribution to estimate the notes distribution. Each parameters of the Gaussian (mean, standard deviation), as in the NADE, is computed from the output of the RNN. We obtain the following learning curve by training our model on the Wikipedia 2 dataset and using GloVes word vectors. unfortunately, because of time constraint reasons, we were not able to decode the sequence of word vectors generated into human interpretable words. These results in an updated version of this paper.

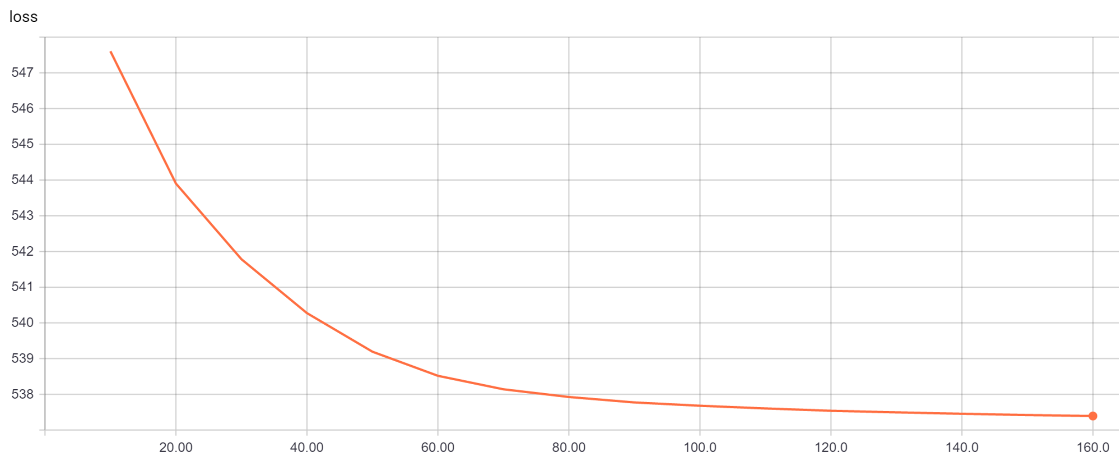

Figure 8: Evolution of the training loss with the number of iterations, on the language generation task

8 Next steps

In addition to the (R)NADE estimators, we are looking to implement RBM estimators. They allegedly obtain better results but are intractable... Nevertheless, advanced sampling algorithm, such as Gibbs sampling, can counter the intractability. To improve sentence generation, one might also consider converting the word vectors into sparse binary vectors and thus reuse our NADE model.

9 Acknowledgments & Conclusion

Overall, the RNN-NADE model for music generation outperforms previous baselines, on a reference piano dataset. We then trained this model on a larger piano dataset and created a demo of style transcription. The model was also trained on a very large drums dataset in order to get the most out of it. Finally, we adapted the RNN-NADE to a RNN-RNADE model in order to perform language generation.

We would like to thanks Daniel Richtie and Richard Socher for their amazing help on this project!

References

[1] Hugo Larochelle & Iain Murray *The Neural Autoregressive Distribution Estimator.*

[2] Benigno Uria & Marc-Alexandre Cote & Karol Gregor & Iain Murray & Hugo Larochelle (2016) *Neural Autoregressive Distribution Estimation.*

[3] Nicolas Boulanger-Lewandowski & Yoshua Bengio & Pascal Vincent *Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription.*