CS 224D Final Project
DeepRock

Ilan Goodman
Department of Computer Science
Stanford University
igoodman@stanford.edu

Sunil Pai
Department of Computer Science
Stanford University
sunilpai@stanford.edu

Abstract

We create a canonical encoding for multi-instrument MIDI songs into natural language, then use deep NLP techniques such as character LSTM variants to compose rock music that surpasses the prior state of the art and is competitive with certain pieces of music composed by human rock bands. We further define a neural network architecture for learning multi-instrument music generation in concert, but due to space and time constraints are unable to sufficiently train it.

1 Introduction, Background, and Problem Statement

How effectively and harmoniously can we compose rock-style music using neural networks? While there has been some prior research in algorithmic music generation, both using classical algorithms [1] and neural networks [3], these models have largely been trained exclusively on classical music. Rock music introduces some new challenges compared to classical music; in particular, drums are integral to rock music and must be handled carefully. Furthermore, these models do not perform competitively in a Turing-style test, as we demonstrate in our paper, and their compositions do not score competitively with human compositions in terms of average enjoyment. In our project, we seek to address both these problems: we write several variants of an RNN model to compose music that is both more Turing competitive with music written by rock stars, and then we propose a neural network architecture to targetedly solve the problem of multi-instrument music generation. We use rock music MIDI files for training, and our models output music in this style.

2 Character-LSTM Model

2.1 Model

While prior models focus exclusively on predicting notes atomically, our most competitive models involve first converting our MIDI inputs into a single string, then presenting the model with the task of predicting the next character in the string. This is in contrast to predicting the notes for the next time step. The benefit of using this format is that rather than dealing with one set of measures or timesteps at a time (as is done with state of the art models), we use a fixed number of preceding characters in our music string.

In constructing our model, we design several variants of the Tensorflow implementation of character-RNNs on GitHub (which is simply a stacked LSTM with an embedding variable) [4]. The variants include a two-layer 128 embed-size RNN-LSTM (`ShallowRockNet`), a three-layer 256 embed-size RNN-LSTM (`DeepRockNet`), and a three-layer 256 embed-size RNN-LSTM with peepholes activated (`PeepRockNet`). Our loss function is the sequence loss we implemented in Assignment 2. We modify and run these models on the GPU of the Amazon EC2 instances provided by CS224D.

We also design a new data format and music sampling scheme that applies the character-RNN intuition to musical composition, as discussed in section 2.2.

2.2 Format

At the beginning of every measure, we use a pipe character to prime the network about how measures should start and end. For every time step, the song plays note value(s) from 0 to 78 (the number of keys in the MIDI keyboard), which we represent as a list of notes. We may include multiple instruments by using a different delimiter for each instrument and iterating in the same order over all instruments at every time step. If no notes are played by an instrument at some time step, we use the token "X". Finally, if a note is articulated at a given time step, we denote that by following up the note with a ' character. A sample format generated by our peephole network looks like `11'*18' 21-33-45 X 19'-37'-40'-45' 11'*18' 19-37-40-45`. Here, it is apparent we have two instruments because we have two delimiters ("-" and "*" for non-percussion and percussion, respectively). We observe there is a rest of one time step in length for the percussion. Notice how 11 and 18 are common for the percussion since there are certain keys on the keyboard that correspond to percussion that is more often played in rock (such as the high hat, the snare drum, and the kick drum). Furthermore, as can be seen in the example, the notes are often repeated between time steps so that the decoder can generate longer notes than a sixteenth note. The decoder knows to extend a note for longer than a time step because the articulation character does not continue over multiple time steps (e.g. `19'-37'-40'-45'` is followed by `19-37-40-45` to extend the four notes over more than a time step). The decoder may also rearticulate an instrument at the very next time step as was done in our example (`11'*18'` is followed by `11'*18'` again). To augment our data, we always sort the notes in order (as in the format sample) as this allows for our network to more easily predict the next character in its generation. For example, we are very unlikely to follow up a note 18 with the note 9 in the same time step because in all of our training data the note 9 precedes the note 18.

2.3 Training Set

To construct our training set, we concatenate the music string output of 466 rock songs into one big string over which we select batches for training. To select a batch, we pick 50 consecutive training examples of 50 characters in length (total of 2500 characters in total) at a random place in the string. This means that there are song-crossover events that we are not handling, and those events may occur while we collect our 2500-character sequences over the full database for training. This issue sometimes carries over into training. When the percussion dies out in a generated sample (which is somewhat uncommon in rock), we may actually be simulating an event where we move from a song with percussion to a song without percussion. We will handle this in future runs using a "`)`" trailing character for each song.

2.4 Generation

For our generation, we sample differently depending on two cases:

1. If we move onto the next time step or the next instrument (i.e., if the previous character is a whitespace character), we sample using a weighting scheme to avoid playing the same note over and over again.
2. If we are in the process of completing a time step in note-space of our string, we simply pick the maximum probability based on the logits in our output layer (specified in section 2.2).

3 Visualizations and Qualitative Evaluation

To better understand the nature of our composition from the character RNN, we begin with a visualization and qualitative evaluation of our training. One can notice that the shallow network has a triplet structure in the composition, while the deep network outputs are more representative of a typical rock composition (see Figure 1). The peephole network shares the triplet structure with the shallow network, but even so, received a much more positive response (cf. section 4). This likely has

to do with the percussion from our peephole network modeling rock-style drums far more plausibly than our deep network does.

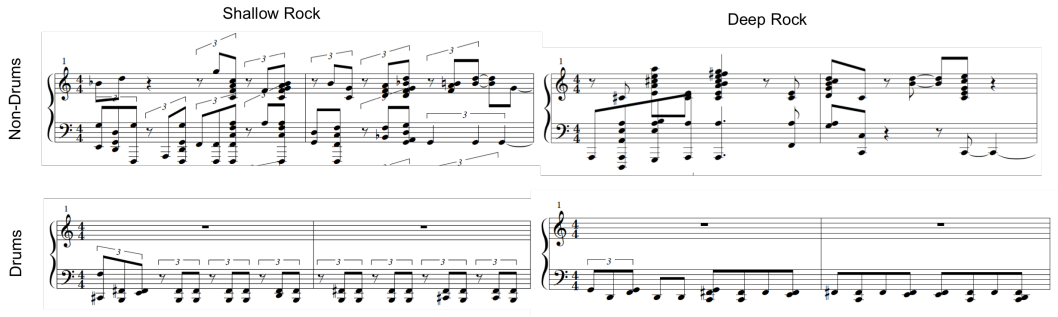

Figure 1: The generation shows overabundance of triplet structure in `ShallowRockNet` but not in `DeepRockNet`. Drum notes do not represent notes, but rather different instruments played at each beat.

In particular, the peephole connections allow input activations to be fed into the forget gate even when the output gate is closed allowing for long range communication within our network [2], likely leading to improved rhythmic performance.

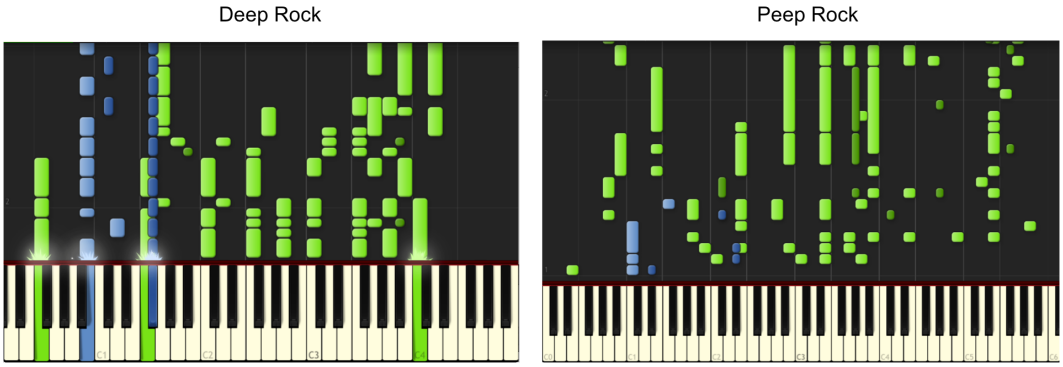

Figure 2: The `PeepRockNet` shows interesting rhymic patterns in the piano roll and `DeepRockNet` shows impressive chord performance, but poor synchronization of drums and other music.

4 Quantitative Evaluation

4.1 Model Loss

Figure 3 illustrates the loss of each of our models as we trained the respective neural nets.

We find that the `DeepRockNet` and `PeepRockNet` are both training very well even towards later iterations, and `DeepRockNet` seems to be training the best out of all the networks. The `ShallowRockNet` performs the worst out of all of the networks in terms of training loss. With more training (and perhaps a longer sequence length), we hope that the network can finally learn about how to synchronize percussion and non-percussion instead of learning independently. Overall, it is important to note that while this model loss evaluation is done in terms of purely model-specific terms (sequence loss is a log-likelihood for the model), our generation results may differ in terms of human evaluation (quality and Turing evaluations) as discussed in section 4.2.

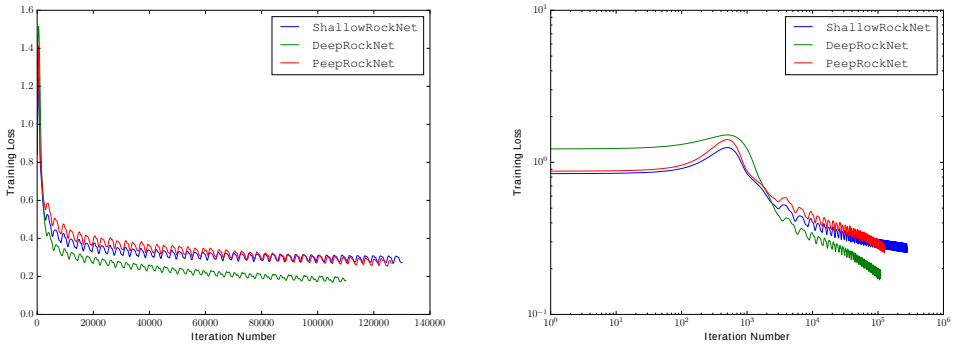

Figure 3: (Left) Training error plot for our neural network (sequence loss). (Right) Same training error plot, but in log scale to show how the different models are training over time.

4.2 Subjective Human Evaluation

We next compare the performance of our character-RNN models with a state-of-the-art biaxial LSTM-RNN architecture (SOTA) for music generation and an actual sample from our training set, "Karn Evil 9" (Emerson, Lake & Palmer), using a survey that we compiled.

In our study, we presented subjects with five songs in a random order: one song from the SOTA model, one song from our `PeepRockNet`, one song from our `ShallowRockNet`, one song from our `DeepRockNet`, and "Karn Evil 9." We told subjects that at least one song was written by a rock musician and at least one was computer generated, and we asked subjects to rank each song on a scale from 1 (worst song ever) to 10 (I love this song) and to guess whether each song was written by a human or by a computer. We chose "Karn Evil 9" for this test both because, while popular decades ago, its largely unknown today (so most people won't immediately recognize it), and also because our MIDI file for it was only piano, so it was already in the same style as our outputs. In order to eliminate bias, we simply took the first 10 seconds of the latest songs each model had produced. We had 132 distinct people take our survey, and each song had between 126 and 131 responses. This is a sufficient sample size to draw statistical meaningful conclusions about our results, although we would benefit from being able to perform a more controlled, in depth experiment.

By nearly every metric, `DeepRockNet` performs the poorest of the five models. Its average enjoyment score was 4.84, and only 23% of participants believed that it was composed by a human. The SOTA model does not fare much better, as predicted: its average score was 4.90, although it did manage to convince 31% of the participants that it was written by humans. Our `ShallowRockNet` fooled 35% of participants into believing that it was written by humans, illustrating the power of even a simple model in our framework.

The most surprising and exciting result is that `PeepRockNet` seems to produce music that participants thought likely to be written by a human. While it had a lower average score (5.52) than `ShallowRockNet` (5.72), it fooled 43% of participants into thinking that it was written by a human: this is a larger proportion than participants who thought "Karn Evil 9" was written by a computer (42%). Finally, "Karn Evil 9" scored the highest of all the songs, with an average value of 6.21, but it does not run away by any metric.

With so many participants, these differences are virtually all statistically significant. We can confidently say that our `PeepRockNet` and `ShallowRockNet` each outperform the previous state of the art model, and `PeepRockNet` even is competitive with certain actual rock songs. While a more detailed study is needed, our models clearly produce reasonable output that can legitimately be considered, to the best of our knowledge, a new state of the art for rock music generation.

5 Biaxial Quad-Directional RNN Model Structure

5.1 Overview

In the interest of theoretical continuations of our character-RNN success, we define a structure called a deep biaxial quad-directional RNN. Consider a MIDI file in the following manner: for each instrument, we have a matrix of notes by times, where at each time step and each note we have feature describing if the note is played and if it is newly articulated. After splitting up instruments, we consider this input as layer 0 in that instrument's section of our computation graph. Then at each hidden layer, we have an affine transformation from the previous layer, and we use LSTM-RNN equations in each of the four directions of the layer (i.e., both note directions and both time directions). At the end of this set of layers, the instruments have only learned independently (it's entirely possible for the melody to optimize in one direction and the percussion to optimize in another), so we add a mixing layer that involves an affine transformation between the final hidden layers of all the instruments. We then use a sigmoid to calculate each note's probability of being played at each time step, and then use these probabilities to evaluate loss and generate music.

Our new neural network architecture (shown in Figure 4) is structured a bit differently than the state of the art biaxial RNN to generalize the generation task to any written song composition with a variety of instruments. To summarize, our neural network is a deep bidirectional, biaxial LSTM (d-BBLSTM) with an affine mixture layer that takes into account the relationships between the instruments.

We will now detail the d-BBLSTM structure and equations that make up f [1].

Input Layer: The input layer has dimensions $\mathsf{R}^{B \times T \times N \times D_c \times M}$, where B is the size of the batch. Consider one of the input layers for instrument class c up until the instrument mixture layer. Let a single training example ${}^{c}\boldsymbol{x} \in \mathsf{R}^{T \times N \times D_c}$ be the input to this layer.

d-BBLSTM Layers: The largest layers are the d-BBLSTM layers. The d-BBLSTM has a three-dimensional prismatic network structure of $T \times N \times L$ hidden nodes with bidirectional connections as shown in Figure 4. We have a fixed dimension for the hidden node $\boldsymbol{h}_{tn}^{(\ell)} \in \mathsf{R}^P$, with $(t, n, \ell) \in [T] \times [N] \times [L]$. The equations for the forward propagation are:

$$
\begin{aligned}
{}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)} &= \mathrm{ReLU}\left(\boldsymbol{W}_c{}^{c}\boldsymbol{x}_{tn} + \overleftarrow{\boldsymbol{U}}_c^{(\ell)c}\,\overleftarrow{\boldsymbol{h}}_{t(n+1)}^{(\ell)} + \overleftarrow{\boldsymbol{V}}_c^{(\ell)c}\,\overleftarrow{\boldsymbol{h}}_{(t-1)n}^{(\ell)} + \overleftarrow{\boldsymbol{Z}}_c^{(\ell)c}\boldsymbol{h}_{tn}^{(\ell-1)} + \overleftarrow{\boldsymbol{b}}_c^{(\ell)}\right) \\
{}^{c}\overrightarrow{\boldsymbol{h}}_{tn}^{(\ell)} &= \mathrm{ReLU}\left(\boldsymbol{W}_c{}^{c}\boldsymbol{x}_{tn} + \overrightarrow{\boldsymbol{U}}_c^{(\ell)c}\,\overrightarrow{\boldsymbol{h}}_{t(n+1)}^{(\ell)} + \overrightarrow{\boldsymbol{V}}_c^{(\ell)c}\,\overrightarrow{\boldsymbol{h}}_{(t-1)n}^{(\ell)} + \overrightarrow{\boldsymbol{Z}}_c^{(\ell)c}\boldsymbol{h}_{tn}^{(\ell-1)} + \overrightarrow{\boldsymbol{b}}_c^{(\ell)}\right) \\
{}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)} &= \mathrm{ReLU}\left(\boldsymbol{W}_c{}^{c}\boldsymbol{x}_{tn} + \overleftarrow{\boldsymbol{U}}_c^{(\ell)c}\,\overleftarrow{\boldsymbol{h}}_{t(n-1)}^{(\ell)} + \overleftarrow{\boldsymbol{V}}_c^{(\ell)c}\,\overleftarrow{\boldsymbol{h}}_{(t+1)n}^{(\ell)} + \overleftarrow{\boldsymbol{Z}}_c^{(\ell)c}\boldsymbol{h}_{tn}^{(\ell-1)} + \overleftarrow{\boldsymbol{b}}_c^{(\ell)}\right) \\
{}^{c}\overrightarrow{\boldsymbol{h}}_{tn}^{(\ell)} &= \mathrm{ReLU}\left(\boldsymbol{W}_c{}^{c}\boldsymbol{x}_{tn} + \overrightarrow{\boldsymbol{U}}_c^{(\ell)c}\,\overrightarrow{\boldsymbol{h}}_{t(n-1)}^{(\ell)} + \overrightarrow{\boldsymbol{V}}_c^{(\ell)c}\,\overrightarrow{\boldsymbol{h}}_{(t+1)n}^{(\ell)} + \overrightarrow{\boldsymbol{Z}}_c^{(\ell)c}\boldsymbol{h}_{tn}^{(\ell-1)} + \overrightarrow{\boldsymbol{b}}_c^{(\ell)}\right)
\end{aligned}
$$

where ${}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)}, {}^{c}\overrightarrow{\boldsymbol{h}}_{tn}^{(\ell)}, {}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)}, {}^{c}\boldsymbol{h}_{tn}^{(\ell)}$ all have dimension P and represent hidden layer activations trained in note-space and time-space for instrument class c, and the input $\boldsymbol{x}$ only feeds into the first layer (set this term to $\mathbf{0}$ for all other layers). We use the notation ${}^{c}\boldsymbol{h}_{tn}^{(\ell)} = \left[{}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)}; {}^{c}\overrightarrow{\boldsymbol{h}}_{tn}^{(\ell)}; {}^{c}\overleftarrow{\boldsymbol{h}}_{tn}^{(\ell)}; {}^{c}\boldsymbol{h}_{tn}^{(\ell)}\right]$ to represent the concatenation of these hidden states for a given c, t, n, and ℓ. We choose to use ReLU nonlinearities in order to combat the vanishing gradient problem. Using this formulation, we have all bias terms $\boldsymbol{b}_c^{(\ell)} \in \mathsf{R}^P$ (for each direction), $\boldsymbol{W}_c \in \mathsf{R}^{P \times D_c}$, $\boldsymbol{Z}_c^{(\ell)} \in \mathsf{R}^{P \times 4P}$ (for each direction), and all other weights $\boldsymbol{U}_c^{(\ell)}, \boldsymbol{V}_c^{(\ell)} \in \mathsf{R}^{P \times P}$ (for each direction). This gives us a total of $\left(\sum_c D_c P\right) + 4CP^2L + 4CP^2L + 4CPL = \left(\sum_c D_c P\right) + 4CPL(2P+1)$ parameters (depending on hyperparameters, this typically comes out to somewhere between $0.5 - 1.5$ million parameters when we have two instrument classes: percussion and other instruments). We have the following initial states: $\boldsymbol{h}_{tn}^{(0)} = \boldsymbol{h}_{0n}^{(\ell)} = \boldsymbol{h}_{(T+1)n}^{(\ell)} = \boldsymbol{h}_{t0}^{(\ell)} = \boldsymbol{h}_{t(N+1)}^{(\ell)} = \mathbf{0}$. The output of this layer is ${}^{c}\boldsymbol{h}_{tn}^{(L)}$, which we will group into $\boldsymbol{H}_c \in \mathsf{R}^{T \times N \times P}$. Note that each of

these nodes are actually LSTM nodes in order to improve the long-term dependencies of our model.

Mixing Layer: The mixing layer consists of a concatenation of the final hidden outputs from the previous layer along the hidden dimension to construct $\boldsymbol{H} \in \mathbb{R}^{T \times N \times CP}$ followed by an affine transformation and sigmoid nonlinearity to the output layer. In the mixing layer, we do not mix along the time and note dimensions but rather along the concatenated hidden dimension. Ultimately, we'd like to output the tensor $\hat{\boldsymbol{Y}} \in \mathbb{R}^{T \times N \times 2 \times C}$, an output layer for each of the instruments. The equation for this transformation is as follows:

$$\hat{\boldsymbol{y}}_{tn} = \sigma(\boldsymbol{W}_M \boldsymbol{h}_{tn} + \boldsymbol{b}_M)$$

where $\boldsymbol{h}_{tn} \in \mathbb{R}^{CP}$ lies along the hidden dimension of $\boldsymbol{H}$. Here, we have $\boldsymbol{W}_M \in \mathbb{R}^{2 \times C \times CP}$ and $\boldsymbol{b}_M \in \mathbb{R}^{2 \times C}$. This layer has a total of $2C^2P + 2C$ parameters.

Output Layer: The output layer consists of the final cross-entropy calculation. Since at this point we have calculated the binary target labels $\boldsymbol{Y}$ and the probabilities at the output $\hat{\boldsymbol{Y}}$, we end up with the elementwise sum:

$$CE(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = \|\boldsymbol{Y} \odot \log \hat{\boldsymbol{Y}}\|_{1,1,1,1}$$

where $\|\boldsymbol{k}\|_{1,1,1,1}$ indicates a sum over the absolute value of the 4-tensor (note that all entries in the tensor will be positive so the absolute value does not matter in this case).

5.2 Initial Results

In Figure 5, we have some results which show the slow training ability of our network as is currently structured.

6 Conclusion

Overall, we had a lot of success with generating enjoyable rock music with character RNNs and we would like to extend this project in the future by implementing some sparse version of our dBQL-STM. To our knowledge, our results and feedback from our model demonstrate that character RNNs are powerful tools that can not only create interesting classical compositions but can be extended to multiple instruments of different styles of music.

References

[1] Gérard Assayag and Shlomo Dubnov. Using factor oracles for machine improvisation. *Soft Computing*, 8(9):604–610, 2004.

[2] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 3:115–143, 2003.

[3] Daniel Johnson. Biaxial recurrent neural network for music composition. `https://github.com/hexahedria/biaxial-rnn-music-composition`, 2015.

[4] Sherjil Ozair. Multi-layer recurrent neural networks (lstm, rnn) for character-level language models in python using tensorflow. `https://github.com/sherjilozair/char-rnn-tensorflow`, 2015.

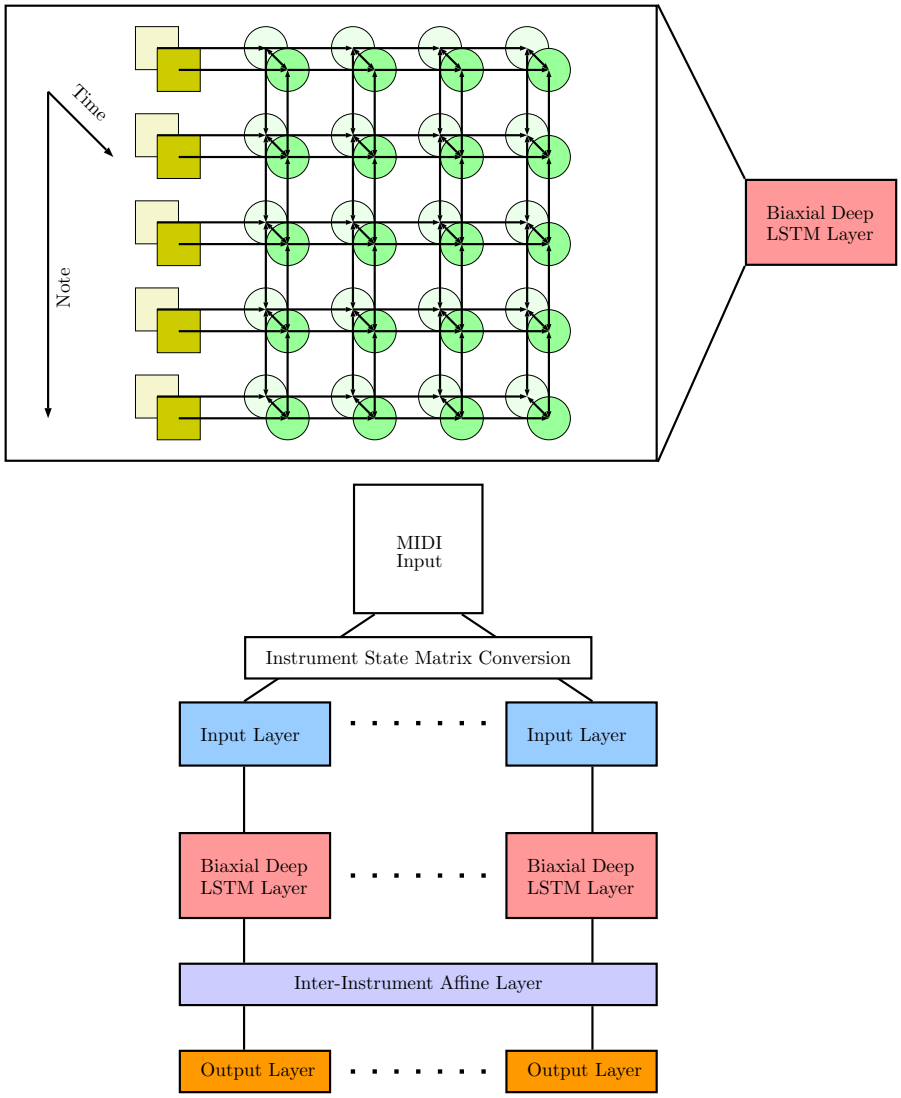

Figure 4: Our deep LSTM architecture

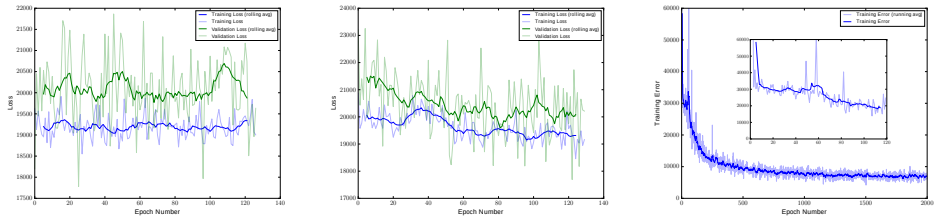

Figure 5: From top to bottom: baseline loss, dBQLSTM loss, SOTA loss. Inset shows dBQLSTM mirrors early SOTA.