# Learning to Create Jazz Melodies Using a Product of Experts

**Daniel D. Johnson** and **Robert M. Keller**
Department of Computer Science
Harvey Mudd College
Claremont, CA 91711 USA
ddjohnson@hmc.edu
keller@hmc.edu

**Nicholas Weintraut**
Department of Computer Science
Rowan University
Glassboro, New Jersey 08028
weintraun8@students.rowan.edu

**Abstract**

We describe a neural network architecture designed to learn the musical structure of jazz melodies over chord progressions, then to create new melodies over arbitrary chord progressions from the resulting connectome (representation of neural network structure). Our architecture consists of two sub-networks, the interval expert and the chord expert, each being LSTM (long short-term memory) recurrent networks. These two sub-networks jointly learn to predict a probability distribution over future notes conditioned on past notes in the melody. We describe a training procedure for the network and an implementation as part of the open-source Impro-Visor (Improvisation Advisor) application, and demonstrate our method by providing improvised melodies based on a variety of training sets.

## 1 Introduction

Substantial work has been done on creation of music by computation, ranging from grammar-based methods (Keller and Morrison, 2007; Gillick, Tang, and Keller, 2010) or genetic algorithms (Biles, 1994) to neural network approaches, including recurrent models (Eck and Schmidhuber, 2002; Franklin, 2004) and deep belief networks (Bickerman et al., 2010). These approaches have had varying success in creating convincing jazz melodies over specific chord progressions.

In the current work, we focus on applying recurrent neural networks to the task of music improvisation. Recurrent models are particularly well suited for sequence prediction tasks, as they are structured to learn patterns across multiple time steps. In particular, LSTM networks (Hochreiter and Schmidhuber, 1997) have been shown to be able to infer complex patterns across many time steps based on data. Additionally, neural network models are interesting to explore because they do not require a large amount of domain knowledge to be explicitly encoded. When given only a limited amount of information about the musical domain, neural networks can use patterns in their training data to discover what makes something musical. This makes them interesting to study as creative systems.

One particularly important component of any neural-network generative model of music is the representation chosen for the task. We use a pair of encodings, motivated by the observation that good jazz melodies have both interesting contours as well as pitches that are sonorous with the chord progression. One encoding is based on intervals between adjacent notes and the other is based on harmonic relationships with the current chord in the chord progression. Each encoding is processed by a separate network component, and each component produces a candidate note probability distribution. These distributions are then combined using Product-of-Experts (Hinton, 2002) to produce a final distribution over notes that can be trained using the maximum-likelihood criterion.

## 2 Background

### 2.1 LSTM Recurrent Networks

Recurrent networks have been shown to be effective at a wide variety of sequence based tasks. In particular, they have been used in the past to model musical data. See Eck and Schmidhuber (2002) and Franklin (2004) for a few examples.

Long Short-Term Memory Networks (LSTM), introduced by Hochreiter and Schmidhuber (1997), have had great success at many sequence modeling tasks. They combat the "vanishing gradient" problem in standard recurrent networks by introducing memory cells that can store state through multiple time steps. An LSTM neuron consists of a series of gates that activate according to the update equations

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\widetilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t C_{t-1} + i_t \widetilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \tanh(C_t)
\end{aligned}
$$

where $C_t$ represents the contents of the memory cells at time $t$, $x_t$ is the input, and $h_t$ is the hidden activations of the LSTM cell. The network consists of one or more layers of LSTM neurons connected with feedback from output to input.

### 2.2 Product of Experts

Hinton (2002) proposed a system known as Product of Experts (PoE) for combining multiple models of the same

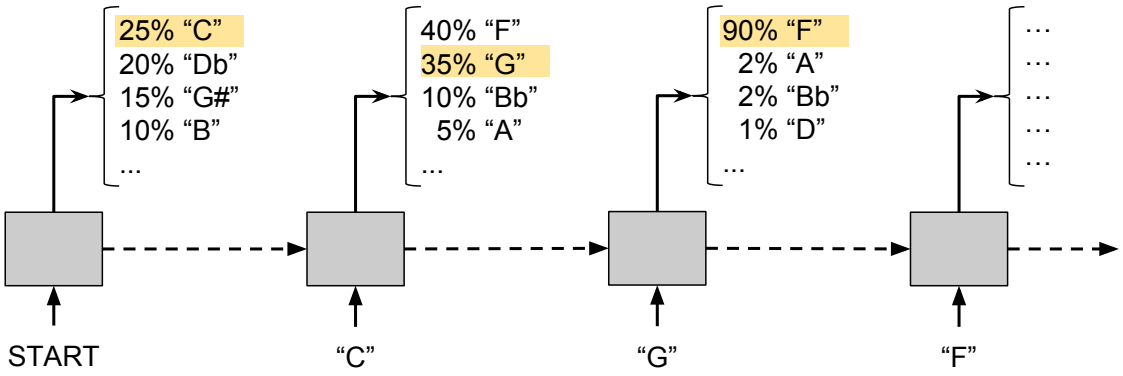


Figure 1: Overview of the network operation, unrolled over four time steps. Inputs are given one time step at a time, and the network outputs a probability distribution predicting the next input. Output is scored based on the probability of correct prediction (highlighted). Dashed arrows represent time-delayed recurrent connections. Although there is also chord input present, we do not show it to avoid clutter.

data. To combine a series of $N$ experts with parameters $\Theta_1, \Theta_2, \ldots, \Theta_N$ and associated probability distributions $f_1(\cdot|\Theta_1), f_2(\cdot|\Theta_2), \ldots, f_N(\cdot|\Theta_N)$, one can renormalize the product of their individual distributions, i.e.

$$p(x|\Theta_1, \Theta_2, \ldots) = \frac{\prod_{m=1}^{N} f_m(x|\Theta_m)}{\sum_c \prod_{m=1}^{N} f_m(c|\Theta_m)},$$

where $m$ indexes all experts, and $c$ indexes all possible vectors in the data space. Notice the similarity of this expression to the definition of conditional probability: $p(A|B) = \frac{p(A \cap B)}{p(B)}$. In fact, this product can be interpreted as the conditional probability that all experts choose $x$, given that all experts choose the same vector from the data space.

When $x$ is a continuous vector in some high-dimensional data space, computing the sum $\sum_c \prod_m f_m(c|\Theta_m)$ and its gradient are both intractable. As such, it is difficult to maximize the log-likelihood of observed data given the model, which led Hinton to propose using contrastive divergence to train such a model. However, if $x$ is a discrete variable in a finite space, as it is here, and each probability distribution $f_i$ is a categorical distribution, the sum can be directly evaluated.

## 3 Network Structure

Following previous work on LSTM-based models (Eck and Schmidhuber, 2002; Franklin, 2004; Boulanger-lewandowski, Bengio, and Vincent, 2012), we break each piece into a set of discrete time steps of a specific length (the length of one 32nd-note triplet). The network is designed to receive the previous note played as input at each step, producing as output a set of probabilities for which note to play at the next time step. The network is trained using the maximum-likelihood objective (i.e. choose the network model that assigns the highest probability to the true training data). Figure 1 gives an overview of the network's operation.

In order to fully develop a LSTM-based model for jazz melodies, we needed to choose a representation for the musical data for the network to understand. Furthermore, different representations might lead to different types of behavior. For instance, if we simply represented the notes by a bit-vector where each bit corresponded to a single note, the model would not be able to generalize well to different intervals and other patterns, as it would have to learn relationships between each note separately. Franklin (2004) found that representations that were motivated by musical relationships led to better performance than more naive approaches.

Two aspects of jazz melodies on which we decided the network should focus are the *contour* of the melody (i.e. how the notes rise and fall over time) and the *consonance* of the melody with the underlying chord progression (i.e. whether the notes sound good when played over chord or whether they are dissonant). (These aspects could be seen as “viewpoints” in the sense of Conklin and Witten (1995)). We also wanted our network to be *transposition invariant*, as used in Johnson (2017), so as to make similar predictions for inputs that are transposed by focusing on the position of notes relative to the roots of the chords.

To accomplish these goals, we decided to use two encodings, one that focuses on intervals between successive notes, and the other that focuses on intervals of notes relative to the chord progression. We thus split our network into two network modules. The *interval expert* module receives an interval-based encoding, while the *chord expert* receives the chord-based encoding. This enables the network to learn particular relationships relative to the current trajectory of the melody (as learned by the interval expert) and also relative to the current chord progression (as learned by the chord expert). Each of these modules produces a probability distribution over which note to choose at the next time step. This probability distribution represents the level of belief that those notes should be chosen to play next. The encodings used are depicted in Figure 2.

To combine the output of the two experts, we use the product-of-experts equation described in detail in section 2.2. This was motivated by the desire to create melodies

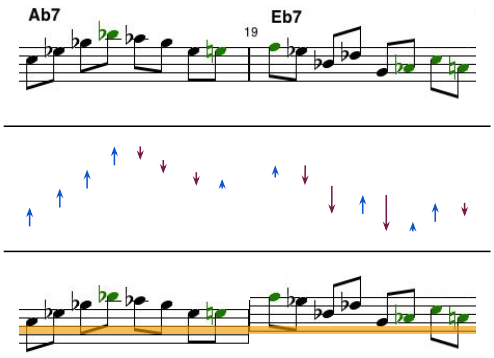


Figure 2: Visualization of the input encodings for the two networks. Top: a section of a training melody. Middle: Interval encoding of the contour. Bottom: Chord-relative encoding of note pitch, with the highlighted space or line representing the root of the chord.

based on both contour and harmonic relationship with the chords. Thus we want to lessen the probability that the combined networks choose a note deemed unlikely by either expert. The combination of the two expert networks is shown in Figure 3.

### 3.1 Encodings

The interval expert is designed to learn relationships between consecutive notes. Each index in the interval encoding of a note represents a possible interval jump, ranging from $-12$ (down one octave) to $+12$ (up one octave). This encoding also includes an option of resting, i.e. not playing a note, and one for sustaining the previous note, for a total of 27 possibilities.

The chord expert, on the other hand, is designed to learn relationships between pitches and the chord. This encoding consists of 12 pitch classes, relative to the current chord root (e.g. if the root of the current chord is F, then indices $0, 1, 2, \ldots$ correspond to notes $F + 0 = F, F + 1 = G\flat, F + 2 = G, \ldots$). Again, the encoding also includes an option of resting, and one for sustaining the previous note, for a total of 14 possibilities.

Each expert receives as input:

1. the note chosen at the previous time step, encoded using the corresponding expert's output format (so the interval expert receives the previous interval jump, and the chord expert receives the previous relative pitch class).
2. a *beat vector* giving the position of the current time step in a measure.
3. a *position vector* giving the position of the previous note relative to the upper and lower bounds.
4. a *chord vector* giving the notes of the current chord relative to the expert's current position.

The beat vector for each time step is constructed using a set of reference note durations. Each reference note duration has a corresponding index into the beat vector, and the beat

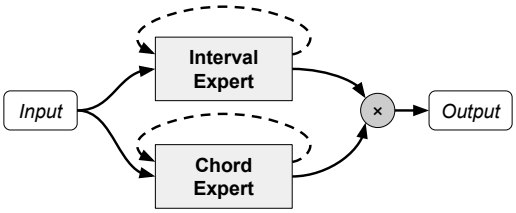


Figure 3: Diagram of the two expert subnetworks. Note that each expert is an independent recurrent network, with output distributions combined using product-of-experts (denoted with $\times$). Dashed arrows represent time-delayed recurrent connections.

vector is 1 at that index if and only if the current time step is a multiple of that note duration. For instance, using reference note durations of [whole note, half note, quarter note, eighth note], the values of the beat vector at each eighth note in a measure would consist of the following sequence of four-dimensional vectors.

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{matrix}$$

The beat vector we actually use is of length 9, consisting of whole, half, quarter, eighth, sixteenth, half-triplet, quarter-triplet, eighth-triplet, and sixteenth-triplet.

The position vector consists of two floating-point values. The first element of the vector is 1 at the lowest note of the network range and 0 at the highest note. The second is 1 at the highest note and 0 at the lowest. Each linearly interpolates its values between the two bounds. This allows the network to learn different behavior when playing high notes and low notes.

The chord vector has length 12. Each index of the chord vector corresponds to a pitch class, and the chord vector has a 1 at that index if and only if that pitch class is part of the current chord. These pitch classes are relative to the expert encoding position: for the interval expert, since notes are chosen as jumps relative to the previous note, the chord vector is rotated so that the previous note is at index 0; and for the chord expert, since notes are chosen relative to the root of the chord, the chord vector is rotated so that the root of the chord is at index 0.

Each expert gives probabilities relative to a particular position. To combine the distributions of the experts into a single distribution, we shift the relative distribution encodings to align them to absolute note positions, clip the probability distributions to a specific note range (three octaves), take the product of the distributions, and then normalize the resulting vector back into a categorical probability distribution (i.e. so that the probabilities sum to 1).

# 4 Training

Our network is trained using the cross-entropy between the network predictions at each time step and the correct notes chosen in the training data. Equivalently, we want to maximize (the log-likelihood of) the probability that the network outputs the training data perfectly, since we want the network to output improvisations that are similar to the training data. We can express the probability of a whole melodic segment as a product of conditional distributions at each time step, i.e. the probability of generating a segment $\mathbf{x}$ from parameters $\Theta$ can be factored as

$$\begin{aligned} p(\mathbf{x}|\Theta) &= p(x_0|\Theta)p(x_1|x_0,\Theta)p(x_2|x_0,x_1,\Theta)\cdots \\ &= \prod_{t\in T} p(x_t|x_{t-1},x_{t-2},\ldots,\Theta). \end{aligned}$$

Notice that the output of the network at some time step $t$ is conditioned on all previous time steps, due to the recurrent nature of the network. Thus $p(x_t|x_{t-1},x_{t-2},\ldots,\Theta)$ is given by the output of the network at time $t$ after receiving $x_{t-1},x_{t-2},\ldots$ as input. To evaluate the full probability of the decoder outputting the input segment, we give the network the observed segment as input, and then accumulate the log-likelihood that the network assigns to the next notes of the observed segment. This gives us the loss

$$\begin{aligned} L_{\text{reconstruct}} &= -\log(p(\mathbf{x}|\Theta)) \\ &= -\sum_{t\in T} \log(p(x_t|x_{t-1},x_{t-2},\ldots,\Theta)). \end{aligned}$$

To train our network, we can then compute the gradient of the loss with respect to our parameters $\Theta$ by performing backpropagation through time (Werbos, 1990).

In our experiments, each expert subnetwork was implemented as two LSTM layers, each with 300 nodes. During training, dropout of 0.5 was applied to the non-recurrent connections between layers (Srivastava et al., 2014; Pham et al., 2014). We used the ADAM optimizer, introduced by Kingma and Ba (2014), to automatically set the learning rate for our training procedure.

## 4.1 Generation

After our model is fully trained, we can use it to create new melodies from the learned probability distribution. At each time step, starting from the beginning of the piece, we compute the probability distribution output by our model. We then choose the next note to play proportionally to the probability assigned to that note by our model. This note is then fed back into the model as the input for the next time step, and a probability distribution for the next time step is computed. This process repeats until we have created a full melody segment.

Optionally, we can modify the probability distribution before sampling from it, in order to modify or constrain the output of the network. The first way to modify the distribution is to vary the ratios between note probabilities. Specifically, if the probability of playing note $i$ is given by $p_i$, we can replace this with

$$p'_i = \frac{p_i^x}{\sum_j p_j^x}.$$

Small values of $x$ make the model more "adventurous" by causing the probabilities to become more similar to each other, and large values of $x$ will cause the model to be "conservative" by sampling high-probability notes more often and low-probability notes less often. Setting $x = 1$ results in the original probability distribution.

Another means of modifying the distribution is to modify the weights assigned to each expert. Instead of computing the probabilities by simply multiplying the probability distributions given by each expert, we can instead use exponents to change the weights assigned to each. If $a_i$ and $b_i$ represent the probabilities assigned by each expert, we obtain

$$p'_i = \frac{a_i^{1+x} b_i^{1-x}}{\sum_j a_j^{1+x} b_j^{1-x}}.$$

This allows the distribution to be biased toward the predictions of a single expert.

Finally, certain elements of the probability distribution can be set to zero in order to prevent sampling particular notes. This can be used either to limit which notes can be played (for instance by only allowing chord tones) or to restrict note durations.

# 5 Implementation

The software described here is implemented in two parts, one for training and the other for creation. The neural network training program was implemented using the Theano machine learning framework. The program reads and parses input pieces, such as jazz solo transcriptions, in Impro-Visor's leadsheet notation, trains a neural network model using those pieces as examples, then outputs a "connectome" file, containing the learned parameters of all of the network components for use in the second part.

To explore the use of our neural network model as an educational tool, we implemented the second part in Java as an extension to the open-source Impro-Visor tool (Keller; Johnson). This extension allows a trained connectome file to be loaded and new improvisations to be created in real time over any chord progression, including trading melodies with a human player.

The improvisation part also possesses a number of parameters, which can be used to modify the network output in various ways:

- **Risk Level**: This option determines how the network output probability distribution is scaled, allowing the user to produce either more original but potentially less consonant melodies or more consonant but possibly less interesting melodies.
- **Expert Weighting**: This option allows the user to put more emphasis on the chord expert or the interval expert, allowing the user to tune the relationship between contour and pitch.
- **Rest Limiting**: This option is designed to prevent the network from playing long rests (often due to the presence of long rests in the training set). If enabled, the probability distributions are modified so that rests cannot be played for too many consecutive time steps.

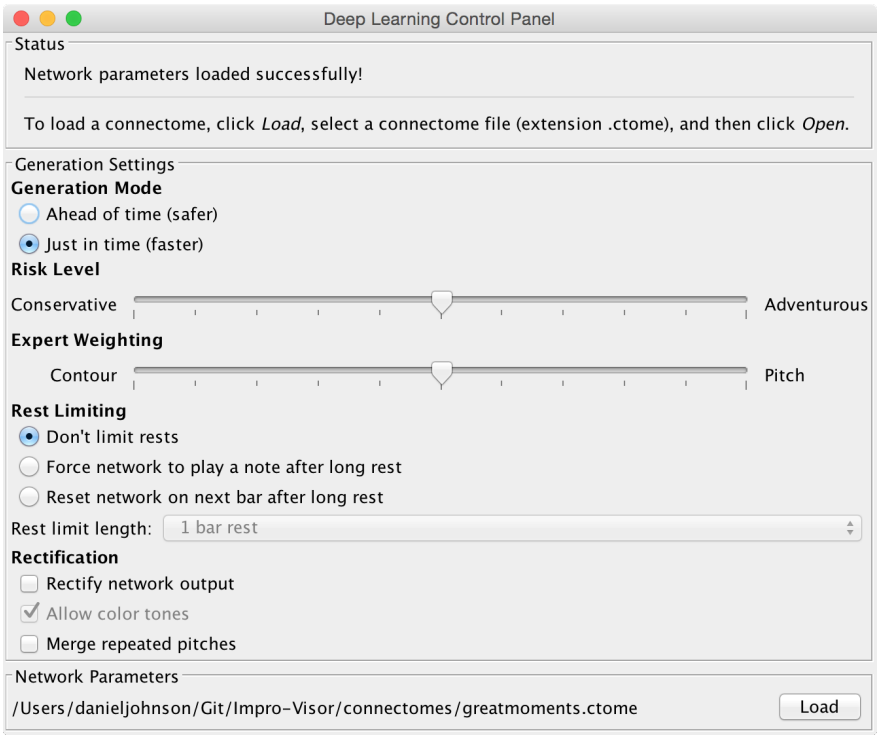


Figure 4: Screen capture of the Impro-Visor deep-learning model control panel.

- **Rectification**: If enabled, this option prevents the network from playing notes that would be dissonant based on the current chord progression. This can help prevent "mistakes" caused by the network randomly choosing dissonant notes with some low but nonzero probability. It can also automatically merge repeated pitches into a single held pitch.

Figure 4 shows the user interface for adjusting these parameters in Impro-Visor.

One advantage of using a recurrent neural network model is that arbitrarily long output can be produced simply by running the network for additional time steps while maintaining the internal state. However, since running the network requires many matrix multiplications, it may take a few seconds to create a few seconds of output. To enable quasi-real-time creation and playback, we implemented a "just-in-time" improvisation process that uses a background thread to produce network output for the next four bars while the previous four bars are being played by Impro-Visor. Then, once playback reaches the end of the existing region, the newly-created four bars are substituted in, and creation begins for the subsequent four bars.

As one application of real-time creation, we integrated the neural network into Impro-Visor's "passive trading" mode. In this mode, Impro-Visor alternates between playing back newly-improvised music and recording user input, allowing the user and the computer model to synthesize a combined piece. This was a natural fit for the just-in-time neural network improvisation procedure.

## 6 Results and Evaluation

Using the network model described herein, we successfully trained connectomes from a variety of corpora, ranging from licks in specific keys to full solo transcriptions from a variety of well-known jazz players. In all cases, we are able to create solos of arbitrary length, and in real-time, from these connectomes. Figure 5 compares a melody created with the current model to one created by a grammar trained from the same corpus. Subjectively evaluated by an experienced jazz player (co-author Keller), the improvised solos exhibited occasional shortcomings, such as:

- occasional "wrong" notes, such as a major 3rd that obviously should be minor, or a minor 9th over a minor or major chord, which is usually not advised, but which human players also play on occasion
- occasional rote learning of melody from the original corpus, albeit correctly transposed, also a characteristic of human players

Both of these shortcomings may be partially attributed to the size of our dataset; the first may represent a failure to learn musical rules of thumb, while the second may be indicative of over-fitting. A larger corpus could be used to improve the generalization ability of our model and prevent it from "memorizing" melodies in the training data.

The Impro-Visor website (Keller) gives these examples, along with the Impro-Visor leadsheet files, MIDI files, and the common training corpus used for both the network connectome and the grammar. It should be kept in mind that

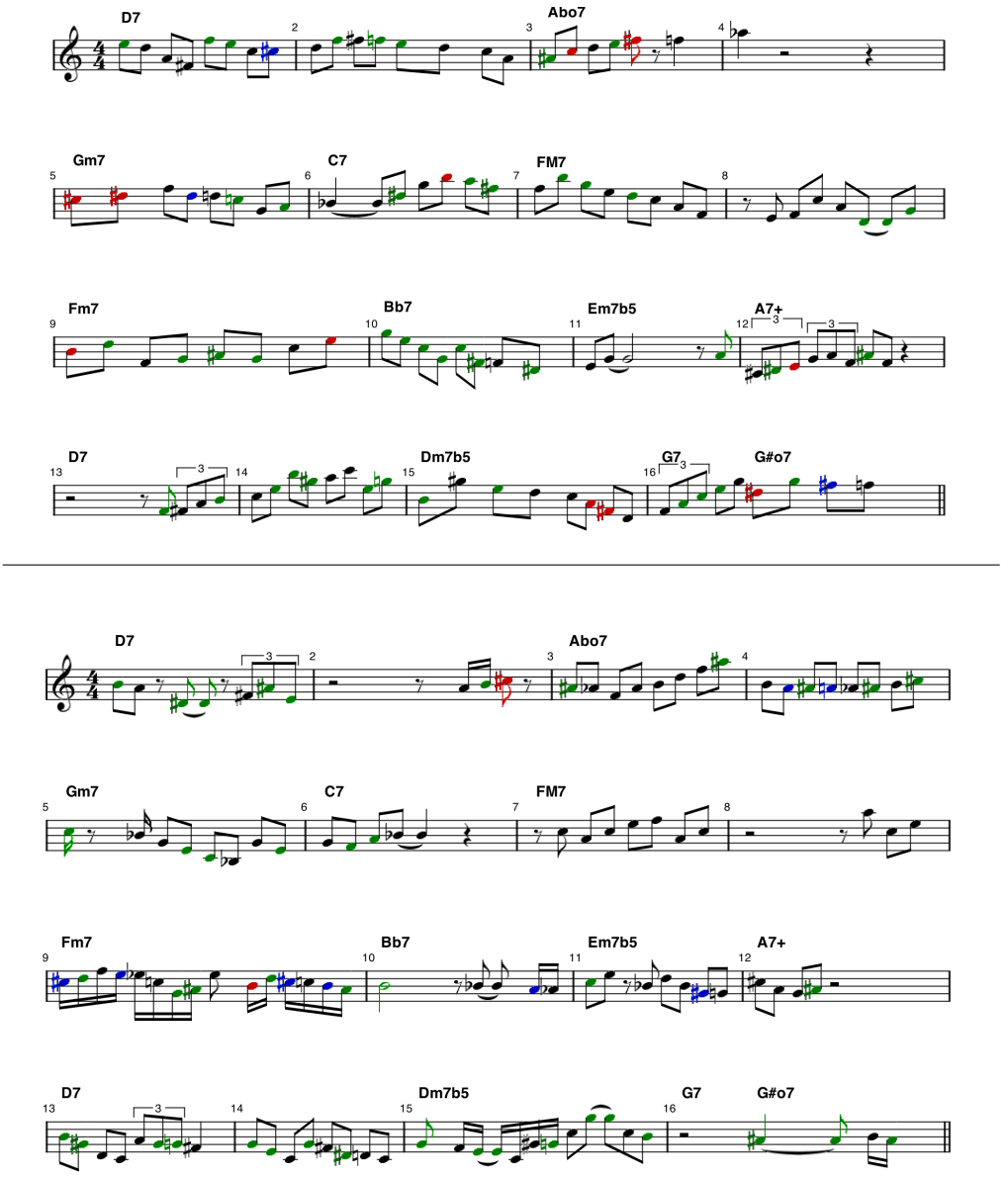


Figure 5: Samples of output created over the chord progression to "Corcovado" by Antonio Carlos Jobim from a connectome trained on a large corpus of licks and solo transcriptions. The corpus did not include a solo for this tune. Top: Version created by our neural network model, with default settings midway between Risky vs. Adventurous, and Contour vs. Pitch preference. Bottom: Version created by Impro-Visor's grammar-based methods trained on the same corpus. Black notes are notes in the current chord, green notes are "color tones", i.e. sonorous with the current chord, blue notes are "approach tones" that transition to chord or color tones, and red notes are "outside" notes that do not fall into these categories. To make the comparison fair, rectification, which would generally eliminate red notes, was not used in either sample.

this represents only one connectome and grammar, from one corpus. Several corpora and corresponding connectomes are available with the release, as are many grammars. Each grammar or connectome can be applied to any tune or chord progression.

Compared with the grammar-based creation in Impro-Visor, the network-created solos are sometimes less coherent. Our network model was built intentionally to have minimal musical knowledge. Since the grammar methods can utilize more prior musical knowledge, and also use Markov chaining of representative melody and abstract melody fragments, the grammar-created samples possess more structure than the network-created samples. Although the current corpus is rather large, an even larger corpus with longer training time might allow our model to learn this latent musical knowledge more effectively, improving the resulting output.

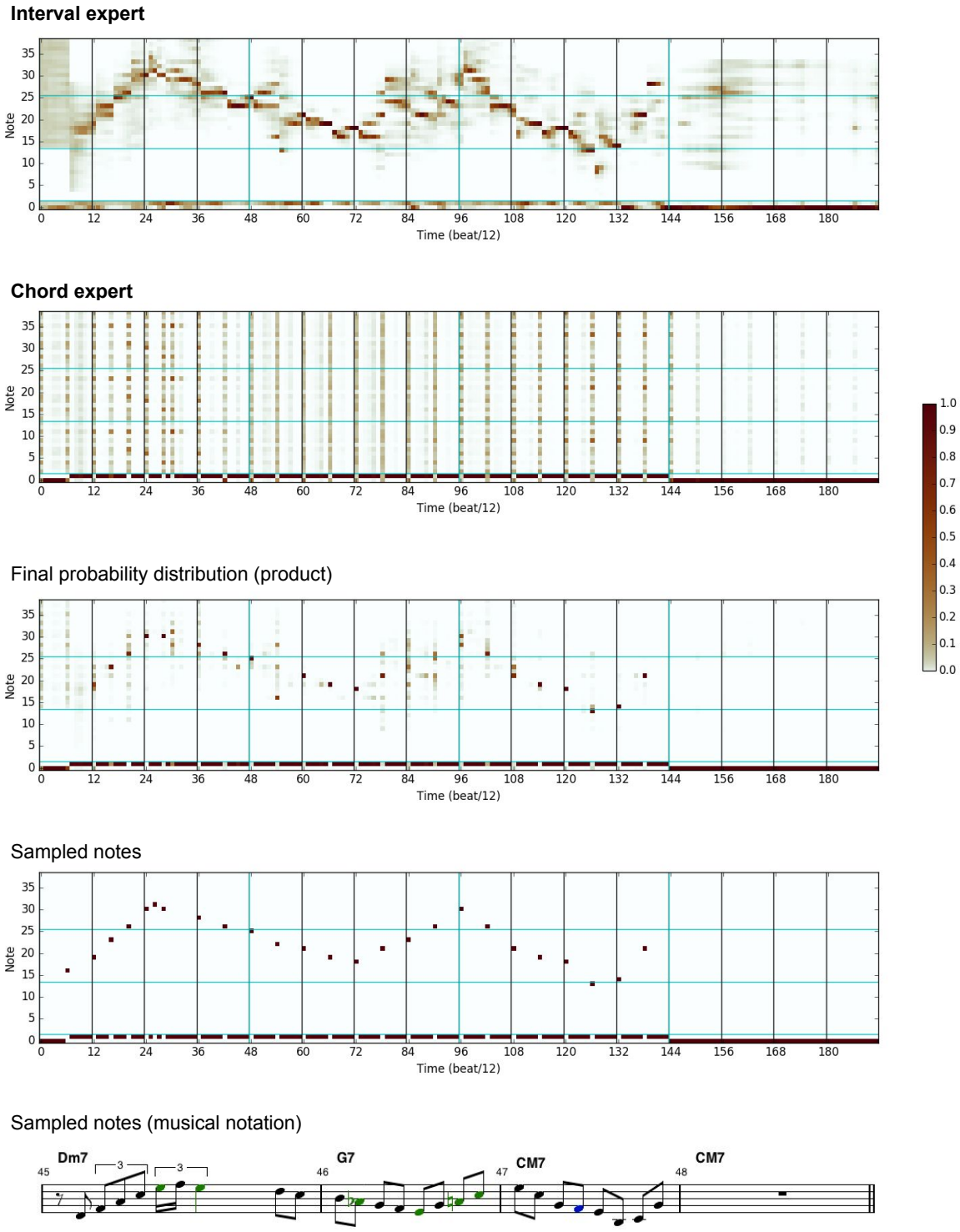


Figure 6: Output of different components of the trained network while creating a new improvisation. Horizontal lines denote octaves. Vertical lines denote quarter-note durations. The bottom two rows show sustain and rest probabilities. Note that, as desired, the interval expert appears to focus on the direction of movement, the chord expert picks out particular notes that are appropriate, and the combined distribution features note choices that are reasonable to both experts.

Compared with the second author's earlier work based on deep-belief networks Bickerman et al. (2010), melody creation is much faster with the current model. One reason for this is that deep-belief networks are slow due to internal probabilistic operation. Unfortunately, the learning time for either form of network is on the order of hours, versus minutes for learning grammars.

To attempt to understand what it is that the model is learning, we visually examined the probability distributions generated by each of the experts as well as by the full network. These distributions for a particular improvised four-bar phrase are shown in Figure 6. As predicted, the interval expert seems to learn to make a smooth contour, demonstrated by the focus of the probability distribution around the locations of previously played notes. The chord expert, on the other hand, identifies particular pitch classes as more likely than others. Interestingly, the chord network also seems to have learned to keep track of note durations.

## 7 Conclusions

We have presented a product-of-experts network approach to learning to create improvised melodies over chord progressions. The network can learn from an arbitrary corpus of existing solos, coded in the form of Impro-Visor's lead-sheet notation. The learning is then saved in the form of a connectome, which can be loaded into Impro-Visor by the user. Melodies can then be created over arbitrary chord progressions. Comparison with the grammar approach previously available in Impro-Visor indicates a slight subjective superiority of grammars, suggesting that future work might focus on improving the network model to understand larger structural units of melodies.

## 8 Acknowledgements

We thank the National Science Foundation for providing funding under CISE REU award number 1359170 and Harvey Mudd College for providing the equipment and facilities for this project. We also thank the developers of Theano, which we used to construct all of our models. We appreciate the work of Joshua Zhao in transcribing many examples into the training corpus.

## References

Bickerman, G.; Bosley, S.; Swire, P.; and Keller, R. M. 2010. Learning to create jazz melodies using deep belief nets. In *ICCC-X, First International Conference on Computational Creativity*, 228–237.

Biles, J. A. 1994. GenJam: A genetic algorithm for generating jazz solos. In *ICMC*, volume 94, 131–137.

Boulanger-lewandowski, N.; Bengio, Y.; and Vincent, P. 2012. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, 1159–1166.

Conklin, D., and Witten, I. H. 1995. Multiple viewpoint systems for music prediction. *Journal of New Music Research* 24(1):51–73.

Eck, D., and Schmidhuber, J. 2002. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale* 103.

Franklin, J. A. 2004. Recurrent neural networks and pitch representations for music tasks. In *FLAIRS Conference*, 33–37.

Gillick, J.; Tang, K.; and Keller, R. M. 2010. Machine learning of jazz grammars. *Computer Music Journal* 34(3):56–66.

Hinton, G. E. 2002. Training products of experts by minimizing contrastive divergence. *Neural computation* 14(8):1771–1800.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Johnson, D. D. LSTMprovisor training program source code. `https://github.com/Impro-Visor/lstmprovisor-python`.

Johnson, D. D. 2017. Generating polyphonic music using tied parallel networks. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, 128–143. Springer.

Keller, R. M., and Morrison, D. R. 2007. A grammatical approach to automatic improvisation. In *Proceedings, Fourth Sound and Music Conference, Lefkada, Greece, July.*, 330–337.

Keller, R. M. Impro-Visor. `https://www.cs.hmc.edu/~keller/jazz/improvisor/`. Files specific to this paper: `https://www.cs.hmc.edu/~keller/jazz/improvisor/iccc2017/`. Source: `https://github.com/Impro-Visor/`.

Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Pham, V.; Bluche, T.; Kermorvant, C.; and Louradour, J. 2014. Dropout improves recurrent neural networks for handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, 285–290. IEEE.

Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15(1):1929–1958.

Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688.

Werbos, P. J. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78(10):1550–1560.