Improvising With Neural Networks: What We Did

Daniel Johnson*, Nick Weintraut†, Sam Goree‡, Mackenzie Kong-Sivert*, Wenbo Cao* and Robert Keller*
*Department of Computer Science
Harvey Mudd College, 340 E. Foothill Blvd., Claremont, CA 91711
Emails: ddjohnson@hmc.edu, mkongsivert@hmc.edu, wcao@hmc.edu, keller@cs.hmc.edu
†Department of Computer Science
Rowan University, 201 Mullica Hill Rd, Glassboro, NJ 08028
Email: weintraun8@students.rowan.edu
‡Department of Computer Science
Oberlin College, 173 W Lorain St, Oberlin, OH 44074
Email: sgoree@oberlin.edu

***Abstract*—Basically, I started working on this document to serve both as high-level mathematical documentation of what we are working on as well as potentially being a draft of part of a paper that we might submit at some time.**

I. Introduction

...

A. Long Short-Term Memory

Long Short-Term Memory Networks, introduced by Hochreiter and Schmidhuber [1997], have had great success at many sequence modelling tasks. They combat the vanishing gradient problem in standard recurrent networks by introducing memory cells, which can store state through multiple timesteps. LSTMs consist of a series of gates, which activate according to the 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 intput, and h_t is the hidden activations of the LSTM cell.

B. Product of Experts

Hinton [2002] proposed a system known as Product-of-Experts (PoE) for combining multiple models of the same 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 take the product of their distributions and renormalize, i.e.

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

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 a conditional probability: the probability that all experts choose x given that all experts choose the same vector.

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, and Hinton proposes using contrastive divergence to train such a model. However, if x is a discrete variable in a finite space, and each probability distribution f_i is a categorial distribution, the sum can be directly evaluated.

C. Autoencoders

Autoencoders are a neural network architecture consisting of two parts: the *encoder*, which takes the input x and produces a high-level feature representation h; and the *decoder*, which takes h and produces an approximation $\widetilde{x}$ of the original input. The autoencoder is trained to reproduce the input as accurately as possible.

One recent example of this that is relevant to our work is an autoencoder model developed by Bowman et al. [2015], which encodes an entire sentence in a single latent space and then decodes from it again.

D. Generative Adversarial Networks

Generative adversarial networks (GAN), introduced by Goodfellow et al. [2014], involve two parts: a generative model—which is tasked with creating new samples—and a discriminative model—which is tasked with distinguishing the training samples from the samples created by the generative model. The goal of the generative model is to fool the discriminative model by creating samples that the discriminative model mistakes for the training samples. The generative model never sees the training data; it only sees the likelihood—as determined by the discriminative model—that each of the samples it has generated is from the set of training samples. The discriminative model tries to maximize the probability of correctly determining which samples came from the training samples rather than from the generative model. The generative

model, by contrast, tries to maximize the likelihood that the discriminative model judges its output to be part of the training samples. Since these cost functions are necessarily opposed, it creates a minimax game which, according to Goodfellow et al., will only stabilize when the generated samples are indistinguishable from the training data.

E. Impro-Visor

Our efforts this summer have been part of a long, ongoing research project called Impro-Visor [Bosley et al., 2010]. The main goal of Impro-Visor is to create intelligent music software to help students practice jazz improvisation and especially to help them practice active trading without having to find other musicians to trade with them. Initially, Impro-Visor would generate four-bar solos that fit with the user's input from grammars. It also includes a critic that grades both its own solos and the user's solos using a neural network.

II. Generative Model

The task of generating interesting, convincing jazz solos can be broken down into a few general parts:

1) Rhythms: are the notes in correct places in time?
2) Contour: how does the melody rise and fall?
3) Chord: how well do the notes fit into the currently played chords?

A sequential model is designed to learn temporal relationships, such as rhythms, but contour and chord relationships are harder to model. To allow the network to learn about each type of relationship separately, we split the network into two experts, an *interval expert* and a *chord expert*.

A. Encodings

The interval expert is designed to learn relationships between consecutive notes. As such, at each timestep the interval network outputs a categorical probability distribution over possible interval jumps, ranging from -12 (down one octave) to +12 (up one octave). The distribution also includes an option of resting for this 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 bites and the chord. As such, the chord network outputs a categorical probability distribution over pitchclasses (in any octave) relative to the current chord bass note (i.e. if the current bass note is F, the network outputs probabilities for playing F+0=F, F+1=G♭, F+2=G, etc). Again, the distribution also includes an option of resting for this note, and one for sustaining the previous note, for a total of 14 possibilities.

As input, each expert receives

1) the note chosen at the previous timestep, encoded with that network's output format (so the interval expert receives the previous interval jump, and the chord expert receives the previous relative pitchclass).
2) a *beat vector* giving the position of the current timestep 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 timestep is constructed using a set of reference note durations. Each reference note duration has a corresponding index into the beat vector, and the beat vector is 1 at that index if and only if the current timestep 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 be

$$\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{array}$$

The position vector has length two. 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 is of length 12. Each index of the chord vector corresponds to a pitchclass, and the chord vector has a 1 at that index if and only if that pitchclass is part of the current chord. These pitchclasses are relative to the expert encoding position: for the interval network, 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 network, since notes are chosen relative to the bass note of the chord, the chord vector is rotated so that the bass not of the chord is at index 0.

B. Using the Model

Each expert is implemented as a multi-layer LSTM network, followed by a fully-connected softmax activation layer.

Each expert gives probabilities relative to a particular position. To combine the distributions of the experts into a single distribution, we shift them to align with specific absolute note positions, clip the probability distributions to a specific note range (three octaves), take the product, and then normalize the resulting vector back into a categorical probability distribution. At sampling time, we sample a note from this distribution at each timestep and feed it back to the network.

To train the model, we evaluate the negative log-likelihood of the observed training data and adjust parameters to minimize this loss. We can express the probability of a whole piece

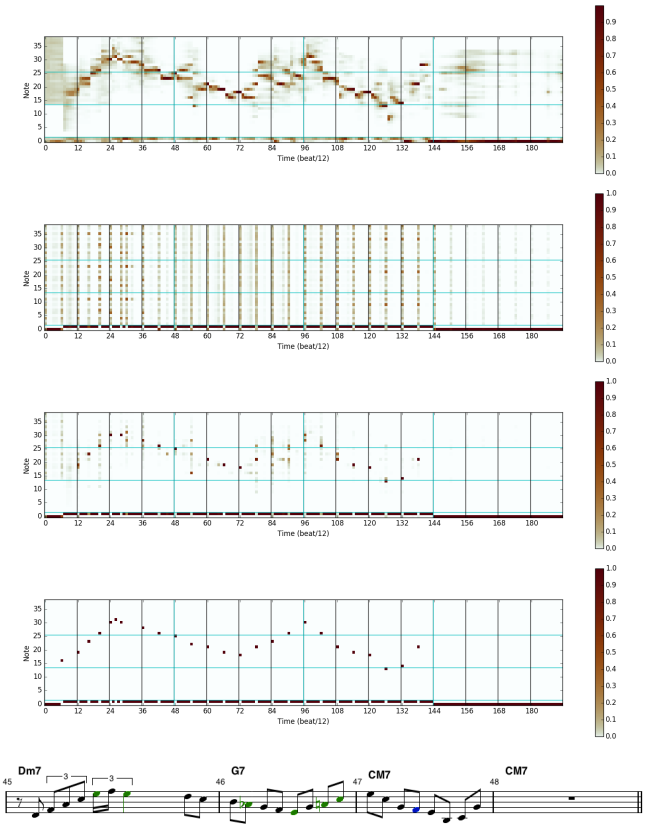

Fig. 1. A sample generated by the product-of-experts model. From top to bottom: interval expert probability distribution, chord expert distribution, product distribution, sampled notes, and generated output. In all plots, the bottom row represents rest, the second row represents sustain, and the rest of the rows represent articulating each pitch, with blue lines separating octaves.

as a product of conditional distributions at each timestep, i.e. the probability of generating a piece $\mathbf{x}$ from parameters Θ is

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

Notice that the output of an expert at some timestep t is conditioned on all previous timesteps 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 being fed as input timesteps $x_{t-1},x_{t-2},\ldots$. To evaluate the full probability of an input piece, we give the network the observed piece as input, and then accumulate the log-likelihood that the network assigns to the next notes of the observed piece.

III. Sequence Auto-Encoder

For active trading, we want to produce a variation of the input, which has some similarities but is not identical. An autoencoder has advantageous properties for this task: since it forms a high level representation of the input, and can reconstruct the input from it, modifications in the encoded feature space should produce meaningful variations to the reconstructed melody.

However, it is unreasonable to expect the network to learn to accurately reconstruct the input from a single coded representation, due to the large variety between musical samples. Instead, we allow the encoder to output a list of feature vectors, each which represents a piece of the input. The decoder reads one of these feature vectors at a time, and decodes the input piece.

Conceptually, we can think of the features as being in a queue: the encoder enqueues features onto the queue, and the decoder dequeues features one at a time. Importantly, the decoder dequeues in sync with the encoder: if the encoder pushes a feature $\mathbf{h}_{t:t+F}$ at timestep $t+F$ representing timesteps t through $t+F$, the decoder receives $\mathbf{h}_{t:t+F}$ as input for all timesteps between t and $t+F$, and then dequeues the next feature for timestep $t+F+1$.

A. Queue Operation

We experimented with both fixed-size and variable-size features. For fixed size features, the temporal locations of the features are fixed ahead of time: the network pushes a feature to the queue at regular intervals, and we read one feature at a time. Specifically, for a fixed feature size of T_{feat}, the encoder pushes a feature at (one-indexed) timestep $T_{feat}, 2T_{feat}, 3T_{feat}$, etc., and the decoder receives the first feature as input for timesteps 1 to T_{feat}, the second feature for timesteps $T_{feat}+1$ to $2T_{feat}$, etc.

For variable-sized features, however, the temporal locations of the features are given by the network itself, and we need to be able to differentiate through the locations of these features. To do this, we use a restricted version of a Neural Queue [Grefenstette et al., 2015].

At each timestep, the encoder outputs a candidate feature $\mathbf{v}_t$, as well as an *enqueue strength* s_t representing the degree of belief that a feature should be enqueued at this timestep. Using this, we can express the feature read by the decoder at timestep t as

$$r_t = \sum_{i=t}^{T} \min\left(s_i, \max\left(0, 1 - \sum_{j=t+1}^{i} s_j\right)\right) \cdot \mathbf{v}_i$$

where T is the length of the input piece.

B. Encoder and Decoder

The encoder and decoder networks can have a variety of structures, from single LSTM layers to product of experts. The encoder takes as input the observed note for the current timestep, a beat vector, and a chord vector. Its final hidden activation layer is processed by sigmoid activation to obtain a push strength and a feature vector. The decoder takes the previous decoded output, a beat vector, a chord vector, and the current top feature vector, and produces a categorical distribution over notes.

If we are using a product of experts model to decode, for encoder-decoder symmetry we also use two networks in the encoder. However, since there are no probability distributions, we cannot take a product of distributions. Instead, we add the

final activations of each network before applything the sigmoid activation function.

C. Loss function

Our autoencoder is trained using the cross-entropy between the network predictions at each timestep and the correct reconstructions of the input. Equivalently, we want to maximize (the log-likelihood of) the probability that the network reconstructs the input perfectly. This loss is the same as in the generative product-of-experts model.

For variable-size features, we also want to encourage the network to learn a few features with large strengths, instead of a large number of weak features. To encourage sparsity, we penalize the network for each feature it creates, and to encourage strong features, we penalize using a nonlinear function p. For example, since we want to prioritize a single feature of strength 1 over two features of strength 0.5 (for instance), we want $p(1) < 2p(0.5)$, but to preserve border behaviour, we want $p(0) = 0$, $p(1) = 1$. For our experiments we used $p(x) = \log_{100}(1 + 99x)$. Then the full loss is given by

$$L = L_{reconstruct} + \alpha \sum_{t=1}^{T} p(s_t).$$

where α is a scaling parameter. In practice, we found that starting with $\alpha = 0$ and then interpolating up to $\alpha = 1$ once $L_{reconstruct}$ was small enough produced the best results.

D. Variational Autoencoder

Variational autoencoders [Kingma and Welling, 2013] have been shown to be beneficial in learning a meaningful latent coding space. In particular, Bowman et al. [2015] use a variational architecture in their autoencoder model.

Integrating variational inference into our autoencoder requires a few changes. Firstly, instead of outputting a candidate feature vector $\mathbf{v}_t$, the encoder outputs a vector of means $\boldsymbol{\mu}_t$ and a vector of standard deviations $\boldsymbol{\sigma}_t$. We then sample values from a Gaussian distribution with these means and standard deviations as our feature vector $\mathbf{v}_t$. Thus we interpret $\mathbf{v}_t$ as a random variate chosen from our latent feature distribution.

As in the original formulation of a variational autoencoder, we add an additional regularization term to our loss, based on the KL divergence between the feature distribution and a zero-centered unit Gaussian. This term is of the form

$$L_{var} = -\frac{1}{2} \sum 1 + \ln(\boldsymbol{\sigma}_t^2) - \boldsymbol{\mu}_t^2 - \boldsymbol{\sigma}_t^2.$$

When training a variational version of the autoencoder model, we add this loss to $L_{reconstruct}$ to attain the full loss, which is backpropagated normally. Note that, in order to allow backpropagation through the sampling of the features, we use the reparameterization trick described in Kingma and Welling [2013]: we first sample a vector $\mathbf{z}_t$ from a zero-mean unit Gaussian distribution, and then compute

$$\mathbf{v}_t = \boldsymbol{\mu}_t + \mathbf{z}_t \odot \boldsymbol{\sigma}_t,$$

where $\odot$ indicates elementwise multiplication.

IV. Generative Adversarial Network

In addition to the compressing auto-encoder model, we also worked on generating music using a GAN. The model takes in arrays of both the melodies and their corresponding chords. The notes within the models are represented using the Circle of Thirds representation.

The model composes the melody by setting out the pitches and other attributes at each of a set number of time steps, each of which is equivalent to the duration of a thirty-second note triplet. Each time step is assigned a pitch, which, using circle of thirds encoding, takes seven bits; three bits determining whether the note is a rest, a continuation of the previous note (sustain), or the beginning of a new note (articulate); and another set of three bits which determines the octave from a set consisting of the octave starting on middle C, the octave above it, and the octave below it.

A. Note Representation

The Generative Adversarial network uses the Circle of Thirds representation introduced by Franklin [2004]. In Circle of Thirds representation, all of the notes in the chromatic scale are arranged on two sets of circles: one with each note separated from the ones next to it by major thirds and the other with each note separated from the ones next to it by minor thirds. Each note is represented according to the two circles on which it lies, as shown in figure 2. For example, C is on the first circle of major thirds and the first circle of minor thirds, so it would be represented as 1000100, and A is on the second circle of major thirds and the first circle of minor thirds, so it would be represented as 0100100.

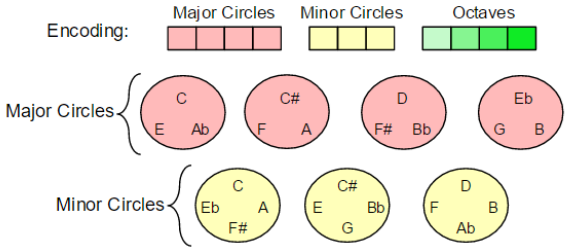

Fig. 2. Circle of Thirds encoding

We also include three bits to specify the octave using one-hot encoding and three bits to specify whether the timestep is an articulation, sustain or rest, respectively, bringing our total note encoding to 13 bits.

B. Structure

The generative model is a neural network with three hidden layers of LSTM nodes, using a $\tanh$ activation function. It takes 63 inputs: 13 bits in Circle of Thirds encoding to specify the root of the chord, 12 to specify the scale degrees above the root in one-hot encoding, 25 random values to add variance (25 was chosen somewhat arbitrarily to bring the non-recurrent inputs to an even 50), and 13 bits in Circle of Thirds encoding to specify the previous timestep's output. It outputs 13 real

values between zero and one where each category (major third, minor third, octave and articulation) sums to one, representing the probability of each bit being one once sampled.

The discriminative model has a similar internal structure to the generative model. It takes 38 inputs: 25 bits for the same chord encoding inputted to the generator and 13 for the Circle of Thirds encoding for the melody timestep it is evaluating. It outputs two values that sum to one, the probability that it classifies the timestep as from the generator and the probability that it classifies the timestep as from the training data.

C. Loss Function

Applying GANs to this sort of musical data is challenging because, when working with discrete data, loss is measured at the distribution level: specifically, based on how likely the network is to output the training data. However, to allow the discriminator to compare the generator output with the training data, we need to sample from that distribution and produce timesteps of music, which means we can't backpropagate error from the output of the discriminator to the weights of the generator, since the space of possible musical timesteps is not continuous.

Our solution is based on a paper by Schulman et al. [2015]. Instead of directly backpropagating the error in the discriminator output to modify the weights of the generator, we calculate the loss for a training example with chords c and melody m of length l:

$$L_g = \sum_{t=0}^{l} -log\left(\sum_{a \in A} p(D(a, c_t) = 1)p(G(m_{t-1}, c_t) = a)\right)$$

Where A is the set of possible values for a timestep of music, D is the discriminator network and G is the generator network, both of which are functions of their network weights and previous hiddens, as well as the values given (left out for clarity).

In other words, we evaluate the discriminator's output on every possible output the generator could have for the next timestep, then compute the sum, weighted by how likely it is that the generator produces that output, and use that as the loss function for the generator. This method allows us to minimize the expected value of the loss, which is a differentiable approximation of the actual loss.

The discriminator's loss function is much more typical, at time t with chord c and melody m:

$$L_d = \sum_{t=0}^{l} -log\left(p(D(a) = r\right)$$

where r is 1 if the sample is from the training data and 0 if it is sampled from the generator. When training on generated data, an entire melody of length l is generated by sampling from the generator's probability distribution.

V. Experiments

We ran a lot of experiments. Describe describe describe etc etc etc.

Lots of datasets. Some jazz licks hand-created, some transcribed pieces from fake books

We used the Adam optimizer [CITE] (with training rate X?) and dropout 50% [CITE]

1) Circles of Thirds generative model: Some success. Didn't really work with this much. Probably not worth discussing in any detail.
2) Product of experts generative model: Good results. Model successfully learns convincing measure-level structure (arguably better than RBMprovisor did) and can generalize to different transpositions / longer pieces. Does not exhibit much long-term structure. Training runs:
   a) Trained on RBMprovisor data: some success, but some wrong notes still
   b) Trained on a large collection of data: much better generalization ability, but occasionally switches styles oddly
   c) Trained on only transcriptions, some good results but "a little outside"
   d) Trained on Bopland progressions: good results, but signs of overfitting.
   e) Trained on Barry Harris: learns general form, but not fully correct, some wrong notes. Still working on this one.
3) Circles of Thirds Compressing Autoencoder with Variable Features: Reasonably successful, but uses queue a lot. Degrades poorly with bad jumps. Probably not worth discussing in any detail
4) Product of Experts Compressing Autoencoder with Variable Features: Successfully learns to reconstruct well. But outputs one feature per note, which is not ideal. Variants:
   a) With variational features: fails to use queue at all, queue loss goes away, acts like a generative model
   b) With forgettable queue: not very good results so far
5) Product of Experts Compressing Autoencoder with Fixed Features: Successfully learns to reconstruct well, using the fixed size features. Not perfect, but generally preserves intervals and most notes are in chord.
   a) Adding noise: Adds variety, but also produces red notes. Post-training the decoder to deal with the noise improves quality of generated notes but reduces reconstruction accuracy.
   b) Interpolation: Can smoothly interpolate between two sets of features. Some odd notes every once and a while. Difficulty interpolating between notes and rests. Intermediate result quality improved with noise post-training
   c) Variational: Still in progress, shows promising signs. Hopefully successful.

6) Generative Adversarial Circles of Thirds:
    a) Expected Value GAN on ii-V-I corpus: Once the expected value method was put in place, the training converged on optima very quickly, however, since the discriminator learned to output zero very quickly (within five timesteps) on suspected generator samples, the generator learned that no note was better than another at any timestep later in the piece and did not learn that most timesteps were sustains or rests in the training data.
    b) Expected Value GAN with reduced piece length on ii-V-I corpus: Since it was clear that the generator was not getting anything out of timesteps after the first few, we reduced the length of training samples from 192 timesteps to 12, which resulted in much faster training and less overwhelming advantages for the discriminator (see fig 3). Future research should be done investigating different thresholds for generator loss before expanding the number of timesteps gradually.

A. Principal Component Analysis

We took the principal component analysis of the feature matrix from the autoencoder model and plotted the first three dimensions, color-coded according to the percentage of time steps that are resting, articulating, or sustaining, according to the graph. As demonstrated in figure 4, three distinct clusters appeared: a cluster mostly dedicated to rests on the far right; a cluster mostly dedicated to long, sustained notes on the far left; and a larger, more scattered general cluster in the center for everything else.

We also plotted all the PCA values according to the components they came from and color-coded them according to the percentage of articulating time steps, resting time steps, and sustaining time steps, respectively, shown in figure 5 in the appendix. We noticed a very sharp drop off in width between the sixth and seventh components, possibly suggesting that only about the first six components have a significant effect on the resulting melody. The color-coding shows the same general clustering that the three-dimensional graphs demonstrated for resting and sustaining, but, where previously we could only see the clusters where the model was least likely to choose to articulate, whereas now, in the second component, we see a clear progression from least likely to most likely to articulate.

VI. Conclusion

VII. Acknowledgements

We would like to thank the developers of the the Python library Theano [Theano Development Team, 2016], which we used to train all all of our models. We would also like to thank the National Science Foundation for providing the REU funding and Harvey Mudd College for providing the equipment and facilities for this project.

References

Sam Bosley, Peter Swire, Robert M Keller, et al. Learning to create jazz melodies using deep belief nets. 2010.

Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

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

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

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

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

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

Appendix

The graphs of our results were getting in the way, so we made an appendix for them.

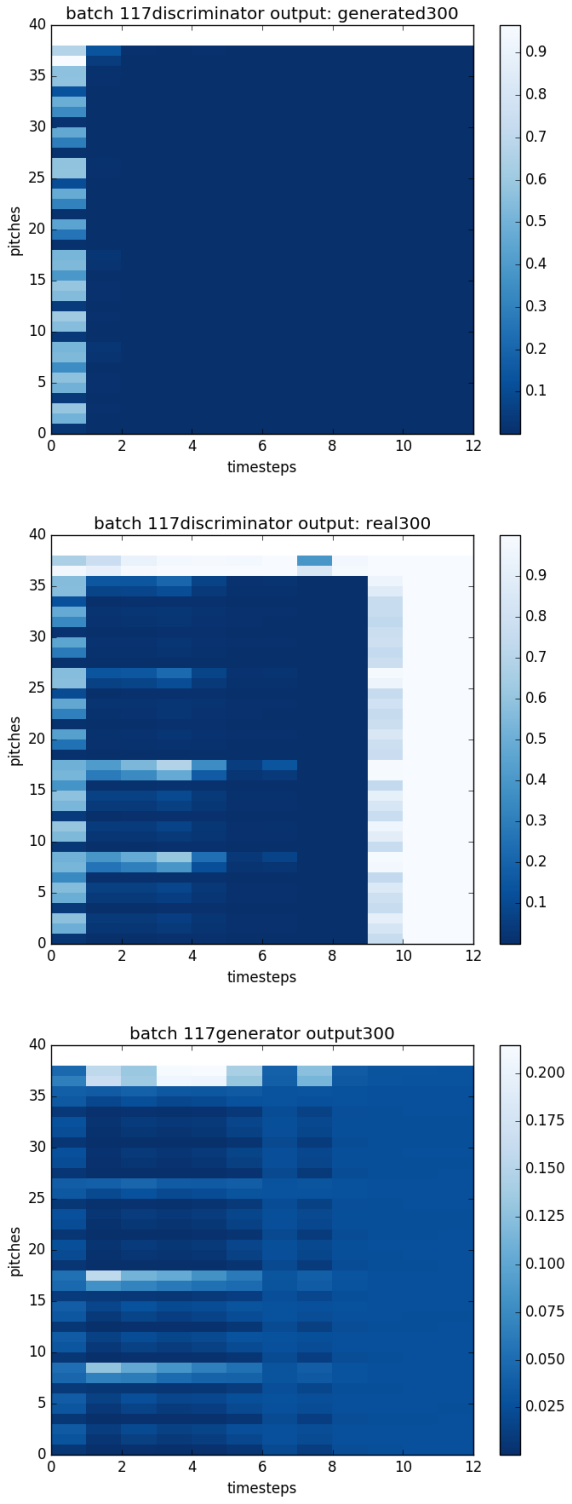

Fig. 3. Probabilities of each possible next pitch plotted over timesteps where color corresponds to probability of a pitch being labeled real (for the discriminator outputs) and probability of being chosen (for the generator output). The top line (pitch 37) is rest, and the second-to-top (pitch 36) is sustain.

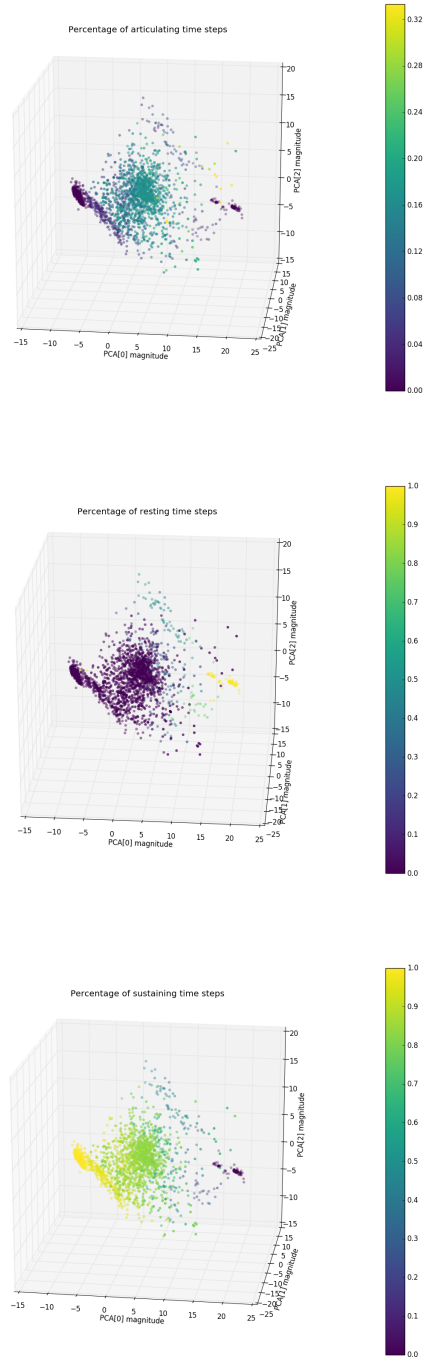

Fig. 4. A graph of the first three dimensions of the principal component analysis, color-coded according to articulating time steps in the first plot, resting timesteps in the second, and sustaining timesteps in the third.

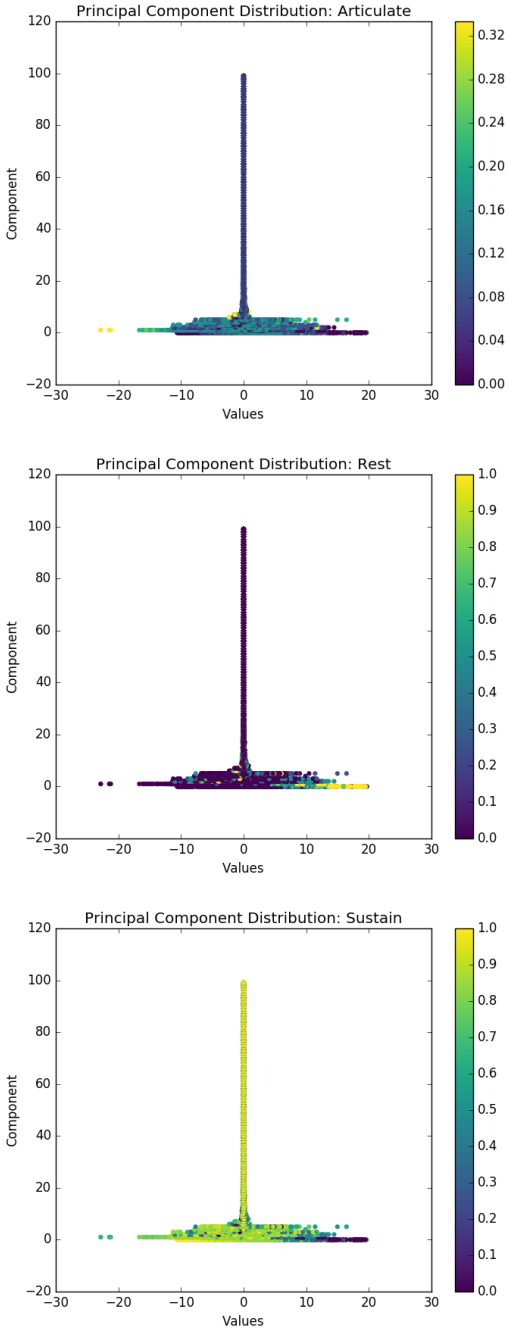

Fig. 5. A graph of all of the values of the principal component analysis, laid out according to their components and color-coded according to percentages of articulating time steps, resting time steps, and sustaining time steps, respectively.