MUSICAL AUDIO SYNTHESIS USING AUTOENCODING NEURAL NETS

Andy M. Sarroff and Michael Casey
Computer Science
Dartmouth College
Hanover, NH 03755, USA
sarroff@cs.dartmouth.edu

ABSTRACT

With an optimal network topology and tuning of hyperparameters, artificial neural networks (ANNs) may be trained to learn a mapping from low level audio features to one or more higher-level representations. Such artificial neural networks are commonly used in classification and regression settings to perform arbitrary tasks. In this work we suggest repurposing autoencoding neural networks as musical audio synthesizers. We offer an interactive musical audio synthesis system that uses feedforward artificial neural networks for musical audio synthesis, rather than discriminative or regression tasks. In our system an ANN is trained on frames of low-level features. A high level representation of the musical audio is learned though an autoencoding neural net. Our real-time synthesis system allows one to interact directly with the parameters of the model and generate musical audio in real time. This work therefore proposes the exploitation of neural networks for creative musical applications.

1. INTRODUCTION

Training advancements in backpropagation, nonlinear activation functions, and regularization have allowed the formulation of expressive artificial neural networks (ANNs) via deep architectures. Such networks are being applied to multitudinous domains. In music, there have been advances in instrument classification [1], genre classification [2–4], artist identification [2, 4], and key detection [4]. In each of these works a new representation of low level audio features is implicitly learned from the training data; in other works feature-learning for musical audio is explicit, e.g. [5–9]. Multimodal objectives are explored as well. Features are learned from data of multiple domains in [10]; a cross modal mapping between representations is subsequently learned.

An autoencoder is a feedforward ANN that is trained to approximately reconstruct its input. A simple autoencoder has one hidden layer of nodes connected to a visible input layer and to a visible output layer. Hence the output of the hidden layer is a transformation or encoding of the network's input. With suitable regularization the model may be trained so that the hidden layer produces a higher level representation of the input that "explains" dependencies of the input. Autoencoders are often stacked so that each hidden layer provides a successively more abstract representation of the input data space. A deep model consisting of several hidden layers is often further optimized for a regression or classification task.

Despite their increased popularity, neural networks have not been fully exploited to synthesize musical audio. If a model has been trained using stacked autoencoders trained on musical input, then the learned parameters may yield increasingly abstract representations of a musical data space. By direct manipulation of the parameters, we may use the model as an interactive musical audio processor and synthesizer.

In this work we suggest that the features learned by such networks may be directly modified to generate new musical audio. The time complexity for a feedforward (synthesis) operation is at most quadratic in the number of nodes of the largest layer. ANNs may therefore be used efficiently in musical synthesis tasks. ANNs are data-driven models; they learn characteristics of the space of the training data. They may be trained in an online fashion and further adapted using new data as it becomes available. With the use of nonlinear activation functions, ANNs may be highly expressive and capture characteristics of the data that linear models cannot. For these reasons, we believe that ANNs warrant investigation for musical synthesis.

As with any machine learning paradigm, there are also tradeoffs. The ANN is a highly general model and it may be designed in a number of ways. Some critical decisions that must be made include width of layers, depth of model, optimization objective, training algorithm, learning rates, types of nonlinearities, and types of regularizations. Thus the search space for discovering the best model for musical sound synthesis is large.

This paper evaluates a few simple models with the aim of musical sound synthesis. Some of our design choices are arbitrary, but they are rationalized in the appropriate sections. We also provide a real time system for musical sound synthesis based on shallow and deep autoencoders. Our models are trained using the Pylearn2 machine learning library [11] which wraps around Theano [12] for fast evaluation of mathematical expressions.

In the following section, we give some background on autoencoders. Section 3 describes how we have trained several shallow and deep autoencoders. It also addresses some of the challenges associated with learning meaningful mid-

level representation of the input features. We then describe our musical interface for "playing" an autoencoding neural net. Future directions are discussed in Section 5. All code is written in Python and provided at `https://github.com/woodshop/deepAutoController`. We hope that this paper encourages others to examine how this highly adaptable class of models may be used for creative musical tasks.

2. AUTOENCODERS

A classical autoencoder (also known as an autoassociator) is a deterministic feedforward ANN comprised of an input layer, a hidden layer, and an output layer (see Figure 1). Each layer of an autoencoder consists of one or more units. The input and output layers of an autoencoder have the same number of units. The autoencoder learns a mapping, or encoding, from an input vector $\mathbf{x} \in \mathbb{R}^d$ to a hidden representation $\mathbf{y} \in \mathbb{R}^e$. It also learns a mapping (decoding) from $\mathbf{y}$ to the output layer $\mathbf{z} \in \mathbb{R}^d$. The inputs to the units in the hidden and output layers are weighted sums of the activations of the layers immediately preceding before them, i.e.

$$\mathbf{y} = s(\mathbf{W}\mathbf{x} + \mathbf{b}_{\text{hid}}) \tag{1}$$

and

$$\mathbf{z} = s(\mathbf{W}_{\text{prime}}\mathbf{y} + \mathbf{b}_{\text{vis}}) \,, \tag{2}$$

where $s(\cdot)$ indicates an activation function (often nonlinear, such as a logistic function) and $\mathbf{W}$, $\mathbf{W}_{\text{prime}}$, $\mathbf{b}_{\text{hid}}$, and $\mathbf{b}_{\text{vis}}$ are the parameters that the model will learn. Activation function(s) for neural networks are a key component of design and continue to be a topic of active research.

When training an autoencoder we choose an objective function that minimizes the distance between the values at the input layer and the values at the output layer according to some metric. Popular metrics include squared error:

$$\sum_{k=1}^{d} (x_k - y_k)^2 \tag{3}$$

or if $\mathbf{x}, \mathbf{z} \in \{0, 1\}^d$, cross entropy:

$$-\sum_{k=1}^{d} \left[x_k \log y_k + (1 - x_k) \log(1 - y_k) \right] \,. \tag{4}$$

The architectures of autoencoders vary. The number of units in the hidden layer may be less than that of the input and output layers. In such cases the activations of the hidden layer are a compressed encoding of the input signal. Alternatively we may choose to give the hidden layer more units than the inner and outer layers. In such cases we will usually enforce sparsity or another regularization on the model so that the overcomplete set of weights may learn a meaningful representation of the data. One method of regularization is *denoising* [13], in which the model learns to reproduce the input from a corrupted version of the input. There are several types of corrupters used in practice, e.g. gaussian distributed noise, dropout, and salt and pepper corruption.

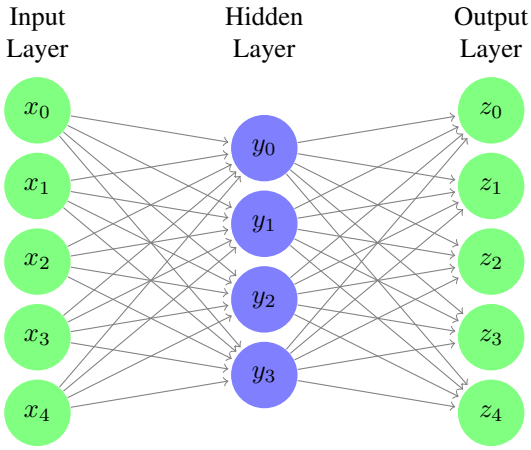

Figure 1. An autoencoder having 5-4-5 input, hidden, and output units, respectively. The autoencoder's objective function minimizes the difference between $\mathbf{x}$ and $\mathbf{z}$, as measured by some distance function.

After training a shallow (single hidden layer) autoencoder, we may use the activations of its hidden layer as the input to a second autoencoder. In this way we may stack autoencoders. For each successive model, we may learn a more abstract mapping from the layer beneath it. The layer-wise pretraining of autoencoders and subsequent stacking and finetuning is a typical strategy for building deep neural networks.

Once a shallow or deep autoencoder has been successfully trained on musical audio, we may synthesize new musical audio by running feedforward passes through the model at an appropriate audio rate. By exposing the activations of the hidden units to a human operator, we may exercise control over the sound of the output of the model. In one case, we may stream audio through the model and modify the activations at one or more hidden layers. Alternatively, we may remove the encoding part of an autoencoder and replace a subset of the hidden units with our own streaming values, propagating them through the decoding half of the model.

3. MODELS

As a proof of concept for an autoencoder synthesizer, we trained several models and built an interface for manipulating the models. We chose simple model topologies and performed some minor grid searching across model size and quantity of corruption (for regularization). Our primary goal was to train several models having low reconstruction error that could be tested as music audio synthesizers using an interactive interface. We discuss the models in this section and discuss the interface in the next section.

We experimented with three model variants:

- Pretraining of a shallow autoencoder
- Deep pretraining of a second autoencoder
- Fine tuning of a deep composed autoencoder

	Noise						
HL1	0.00	0.01	0.02	0.05	0.10	0.25	0.50
8	0.0440						
16	0.0414						
64	0.0276						
256	**0.0187**						
512	0.0198				0.0664	0.0854	0.0921
1024	0.0352				0.0711	0.0927	0.0980
1500	0.0371	**0.0360**	0.0405	0.0547			
2048	0.0983				0.1798	0.2114	0.0972
2500	0.0951	0.0951	0.0951	0.0951			
3500	0.0951	0.0951	0.0951	0.0951			

Table 1. Mean squared validation error on for pretraining of shallow autoencoders. The input/output layers of each model had 1025 units. HL1 designates the number of hidden units. Noise designates the standard deviation of gaussian distributed noise used to corrupt the input signal.

The first model we train is a simple autoencoder like the one depicted in Figure 1. In the second variant we take the output of the hidden layer of an already-trained autoencoder and train a second autoencoder to reproduce the mapped input. Hence if the size of the hidden layer of the first encoder is N, then this is also the size of the input and output layers of the second autoencoder. There is no limit to how many stacked autoencoders we may train in this fashion. For our purposes, we have limited ourselves to stacked autoencoders of depth 2. In the final variant we build a deep composed autoencoder by taking the hidden layers of two pretrained autoencoders and finetuning the weights of the whole system to improve reconstruction of the original input.

We note that this is often the order of events for training a deep neural network. Each layer is pretrained in succession as a shallow model with the previous layer providing the input to the subsequent layer. When pretraining is finished the system is "finetuned".

3.1 Data

We used 70,000 frames of magnitude Fourier transforms randomly selected from a dataset of approximately eight thousand songs existing across unique artists. The dataset is roughly stratified across 10 musical genres. Of these audio frames 10,000 were held out as a validation set and 10,000 were held out as a test set. Each audio frame was computed from a 2,048-point FFT on audio having a sampling rate of 22,050 samples per second. The entire data set was normalized to the range $[0, 1]$. The magnitudes of the first 1,025 frequency bins were given to the models as the input vector of a shallow autoencoder.

We chose to use frames of magnitude FFTs for our models because they may be reconstructed exactly into the original time domain signal when the phase information is preserved, the Fourier coefficients are not altered, and appropriate windowing and overlap-add is applied. It was thus easier to subjectively evaluate the quality of reconstructions that had been processed by the autoencoding models. There are several disadvantages to using FFTs as the low level training data; these are discussed later.

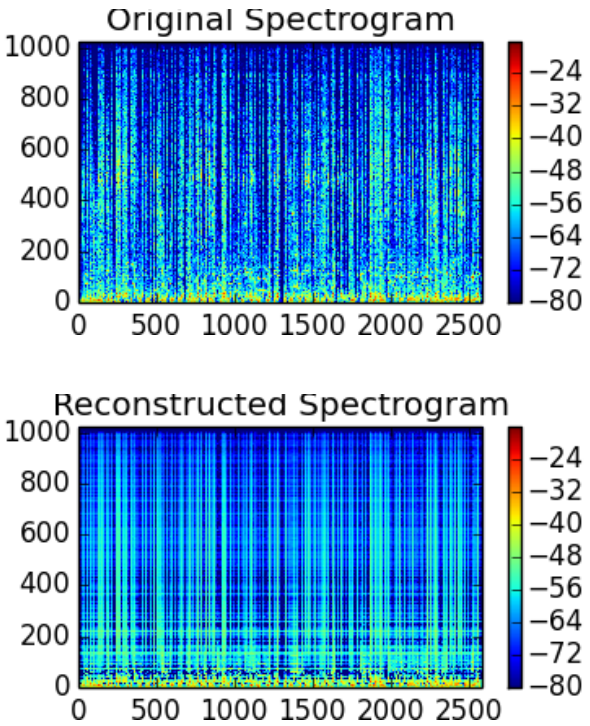

Figure 2. Top: STFT of original audio file. Bottom: STFT of reconstructed audio file.

3.2 Training

Training was performed using stochastic gradient descent on mini-batches of 100 frames. The learning rate was set at 0.005 and a learning momentum 0f 0.5 was used. In all training, the mean squared error was used as the cost function. On pretraining of shallow networks, a sigmoid activation function was used only on the hidden layer, with linear activation on the output layer. When a second autoencoder was employed for a deep model, the sigmoid activation function was used on both the hidden and output layers of the second autoencoder. On some models we additionally used gaussian noise as a network corruptor for regularization. The encoding and decoding weights were untied in our models.

		Noise		
HL1	HL2	0.00	0.10	0.25
256	8	**0.0737**	0.0737	0.0737
256	16	0.0737	0.0737	0.0737
256	32	0.0737	0.0737	0.0737
1500	8	0.0331	0.0331	**0.0329**
1500	16	0.0348	0.0346	0.0344
1500	32	0.0356	0.0369	0.0384

Table 2. *Adjusted* mean squared validation error for second-layer pretraining of deep autoencoders. The input/output layers of each model is designated by HL1. The models printed in boldface in Table 1 were used to provide the inputs to the models in this table. HL2 designates the number of hidden units. Noise designates the standard deviation of gaussian distributed noise used to corrupt the input signal. The mean squared error has been adjusted for easier comparison with other tables. For each model, the mean squared error was scaled by $\frac{1025}{\text{HB1}}$.

3.3 Training Results

Table 1 shows the mean squared error on the validation set for each model that was trained. Smaller networks that employed no denoising perform the best. The optimal number of hidden units given the chosen hyperparameters and model topologies appears to be 256, a feature size reduction of approximately 25%. Increasing the hidden layers to yield overcomplete filters does not appear to improve the models' performance. This is expected behavior for overcomplete models lacking regularization. Adding some corruption to the model with 1500 hidden units appears to improve results slightly. The largest models each show the same reconstruction error. This result may mean that one or more of the values of the hyperparameters (such as learning rate, momentum, initial weights) were inappropriate.

Figure 2 shows the original spectrogram and a reconstructed spectrogram using a 256-8-256 unit autoencoder trained without denoising. We observe that much of the fine-grained detail is lost by the autoencoder, especially above the lowest frequency bins. The figure does not depict desirable behavior for an optimal autoencoder but nonetheless some of the detail in the lower frequency bins is approximately reconstructed.

Table 2 shows the *adjusted* validation performance of a second autoencoder trained using the activations of the hidden units of a first autoencoder as input. The mean squared error of each model has been scaled by $\frac{1025}{\text{HB1}}$ so that it may be directly compared with results shown in other tables. Once again smaller networks perform better than large ones and denoising does not appear to help much. Interestingly we find that the deep 1500-8 model has a better per-visible-unit performance than the 8 unit shallow autoencoder.

Table 3 shows the final validation and test error for two models. The test error is significantly worse than the validation error—a sign of possible overfitting. The final finetuned models perform worse than the deep architectures

HL1	HL2	Validation	Test
256	8	0.0723	0.1006
1500	8	0.0953	0.1333

Table 3. Mean squared error on validation and test set for deep composed autoencoders. The models printed in bold in Tables 1 and 2 were connected and finetuned.

presented in Table 2, suggesting that the learning rate may have been inappropriate. Overall more complex models perform worse than the simpler topologies.

3.4 Discussion

We conducted informal listening tests in conjunction with the synthesis interface described in the next section. The reconstructions sounded similar to the originals, but with a "grainy" noise mixed in. The detail in the high end was missing but much of the harmonic material was preserved.

The optimal parameters of the models were mostly inhibitory. Therefore the deactivation of a unit in a hidden layer yields a denser mixture of sounds at the output. Learning to play such an interface may prove difficult for new users, as one typically expects the opposite behavior from a musical synthesizer.

Neural networks have lots of hyperparameters over which to search, including learning rate, regularization, layer width, and model depth. The training results presented indicate that the hyperparameters chosen for the models were probably inadequate. Further work is needed to examine how these models might be improved.

It is notable that the low level input features, which were magnitude FFT coefficients, exhibit decreased average amplitude as the frequency bin increases (cf. Figure 2. If many of the input features are close to zero, then training may require more epochs and a steeper learning rate to adjust. We choose to work with FFT feature frames because the reconstruction will be cleaner than if we use band-limited features, given an optimal autoencoder.

Future experiments will consider alternative low-level representations with which to train the models. It might be more appropriate for musical sound synthesis if the models are trained using log frequency features rather than linear frequency. Log frequency representations more closely match our auditory systems.

The input data was globally normalized to the range $[0, 1]$. As noted above the distribution of the data was highly sparse. The sparsity of the input data may have affected the ability of the models to optimize their parameters. In future models we will standardize the features to unit variance. Future models will also explore log amplitude representations.

The shallow autoencoders in this paper use a sigmoid encoder and linear decoder with untied weights. It has been shown that these model topologies learn a transformation that spans the same subspace as Principal Components Analysis [14, 15]. Future models will explore alternate models having tied weights.

We have noted that the inhibitory nature of the weights makes it difficult for a musician to learn how to play the synthesizer. We explored models having nonnegative weights

by using an asymmetrical weight decay as shown in [16]. The results are not presented here as they are preliminary. Reconstruction error in such models is worse than without nonnegativity constraints. But we find informally that the models are somewhat more intuitive to play as synthesizers.

We also continue to explore other types of activation functions. In particular, rectified linear units have been shown to work well in the audio domain [17]. We have investigated shallow and deep autoencoders with rectified linear units with and without nonnegativity constraints. We have found that one of our shallow models with 16 hidden inits, unconstrained weights, rectified linear encoding units, linear decoding units, and tied wights achieves relatively good performance, having a 0.0310 mean squared validation error. The filters for this particular model sound more "musical" to us and the model is more intuitive to play as a musical sound synthesizer. However all current results on new developments are subjective and performance results are too preliminary to include in this paper.

4. INTERFACE

We programmed a real-time interface for interaction with the hidden units of deep or shallow autoencoders. Whichever type of model is given to the program, the innermost hidden layer is exposed to the user for interaction. The interface is designed to work with models that have been trained using the Pylearn2 library, but generalizing the program to accept lists of parameters rather than class instances of models is trivial. The code is available at `https://github.com/woodshop/deepAutoController`; it will be actively improved/updated. The current version is written in Python, but another version which is written in Objective-C++ may be deployed soon.

The current code consists of two classes, one for the interface, and one for the audio streaming and processing. The program is executed with three mandatory input arguments: the path to a pickled Pylearn2 model; a file indicating the parameters for low-level feature extraction; and a file designating what preprocessing to apply to the features. At the initialization of the application a Python Queue is instantiated for message-passing between the Autocontrol class and the PlayStreaming class. The two classes are briefly described below.

4.1 Autocontrol Class

The interface is designed to work with the Korg nanoKontrol2, a MIDI controller having 8 fader channel controls and a transport. Although the code has been written for this controller, it is easy to rewrite the mappings for another MIDI controller. The Autocontrol class instantiates a MIDI connection and uses the Pygame package to poll for MIDI events and produce informational output in a separate window (Figure 3). The interface receives and several defined MIDI events form the nanoKontrol2:

- Track: Cyclically moves the view of hidden layer units backward or forward by 8 units.

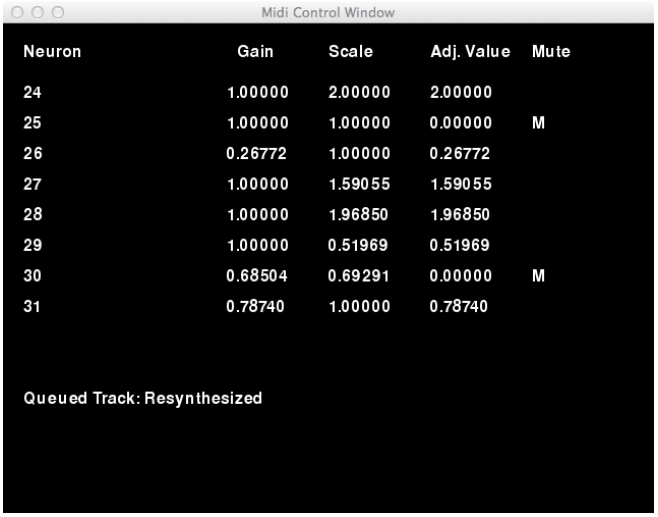

Figure 3. A snapshot of the information window showing which hidden units are in view and what their scaling settings are.

- Cycle: Shuts down the application
- Set: Sets the output of all units to 0.
- Rewind and Fast Forward: Switches between original and synthesized audio
- Stop: Stops the audio and rewinds
- Play: Plays/pauses the audio output
- Record: Resets all hidden units to original activation values
- Pan pot and fader: Control a scaling factor which is multiplied against a particular unit's activity, thus suppressing or augmenting the activity at that unit.

4.2 PlayStreaming Class

This class is instantiated as a separate process. It loads the parameters of the Pylearn2 model an an optional audio file. It polls for messages from a queue instance. When a user interacts with the midi controller this class instance receives a message from the Autocontrol class instance. Audio frames are read directly from an open audio file or the computer's default input and transformed to feature frames. If the user has designated that the original audio stream should be monitored, the audio frame is immediately transformed back to the time domain and sent to the output. Otherwise it is encoded (Eq. 1). The activation of the hidden units are scaled or muted by the user's settings. The output is decoded (Eq. 2 back to an low level feature frame, converted to the time domain, and sent to the output.

If an audio file is not provided, sound from the computer's default input device is used. The interface can also operate in a no input mode. In this case, the class "fires" the hidden layer at an audio rate while the user maintains control over the scale of the hidden layer units. A channel vocoder provides phase continuity for the inverse FFT if the user

decides to ignore the phase information from the input signal.

5. FUTURE WORK AN CONCLUSIONS

This paper presents a first step toward extending the typical use patterns of neural networks beyond classification and regression to audio synthesis. Training an autoencoder so that it captures a meaningful mid-level or higher-level representation of the input is difficult. As has been shown in Section 3 it may be difficult to optimize a model. Simply adding extra layers to create a deep model does not automatically yield a richer instance. There are lots of model hyperparameters to finetune in ANNs, including learning rate, weight decay, momentum, and other forms of regularization. In the future additional effort will be placed toward building more robust models.

One drawback of using neural networks for musical audio synthesis is that the learned weights may be negative. Since weights may be subtractive as well as additive, it is difficult to understand how they contribute to the model. Future work will include investigating models that are trained using nonnegative weight regularization, as well additional sparsity constraints and activation functions. It is the authors' belief that neural networks having overcomplete, sparse nonnegative weights will be easier to musically control.

The currently investigated models do not consider temporal dependency. In the future we would like to apply musical synthesis using temporally inclusive architectures such as recurrent neural networks.

There are many other extensions to consider. For instance we envision pretraining a deep autoencoder for optimal reconstruction, followed by supervised finetuning using instrument classes. If the model learns to respond well to specific instruments (or other acoustic events), we may use autoencoder synthesizers to remix music.

6. REFERENCES

[1] P. Hamel, S. Wood, and D. Eck, "Automatic identification of instrument classes in polyphonic and poly-instrument audio." in *Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR-09)*, Kobe, Japan, October 2009, pp. 399–404.

[2] H. Lee, P. T. Pham, Y. Largman, and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks." in *Proceedings of the Neural Information Processing Systems Foundation (NIPS-14)*, vol. 9, Vancouver, Canada, December 2009, pp. 1096–1104.

[3] P. Hamel and D. Eck, "Learning features from music audio with deep belief networks." in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR-10)*, Utrecht, Netherlands, September 2010, pp. 339–344.

[4] S. Dieleman, P. Brakel, and B. Schrauwen, "Audio-based music classification with a pretrained convolutional network," in *Proceedings of the 12th international society for music information retrieval conference (ISMIR-11)*, Miami, USA, October 2011, pp. 669–674.

[5] E. Humphrey, A. Glennon, and J. Bello, "Non-linear semantic embedding for organizing large instrument sample libraries," in *Proceedings of the IEEE International Conference on Machine Learning and Applications (ICMLA-11)*, Honolulu, HI., December 2011.

[6] E. M. Schmidt, J. J. Scott, and Y. E. Kim, "Feature learning in dynamic environments: Modeling the acoustic structure of musical emotion." in *Proceedings of the 13th international society for music information retrieval conference (ISMIR-12)*, Porto, Portugal, October 2012, pp. 325–330.

[7] E. Humphrey and J. Bello, "Rethinking automatic chord recognition with convolutional neural networks," in *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA-12)*, Boca Raton, USA, December 2012.

[8] E. Humphrey, T. Cho, and J. Bello, "Learning a robust tonnetz-space transform for automatic chord recognition," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-12)*, Kyoto, Japan, May 2012, pp. 453–456.

[9] E. M. Schmidt and Y. E. Kim, "Learning rhythm and melody features with deep belief networks," in *Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR-13)*, Curitiba, Brazil, November 2013.

[10] O. Fried and R. Fiebrink, "Cross-modal sound mapping using deep learning," in *New Interfaces for Musical Expression (NIME'13)*, 2013.

[11] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio, "Pylearn2: a machine learning research library," *arXiv preprint arXiv:1308.4214*, 2013. [Online]. Available: http://arxiv.org/abs/1308.4214

[12] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[13] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[14] H. Bourlard and Y. Kamp, "Auto-association by multilayer perceptrons and singular value decomposition," *Biological Cybernetics*, vol. 59, no. 4–5, pp. 291–294, 1988.

[15] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1798–1828, Aug 2013.

[16] A. Lemme, R. F. Reinhart, and J. J. Steil, "Online learning and generalization of parts-based image representations by non-negative sparse autoencoders," *Neural Networks*, vol. 33, no. 0, pp. 194–203, 2012.

[17] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. Hinton, "On rectified linear units for speech processing," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 3517–3521.