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

Artificial Neural Networks (ANNs) have recently experienced renewed interest. New training paradigms that include backpropagation, nonlinear activation functions, and regularization schemes have allowed the formulation of expressive models via deep architectures. Such deep 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 work feature-learning for musical audio is explicit, e.g. [5–9].

Despite their increased popularity for classification and regression, attributes of neural networks have not been fully exploited to synthesize musical audio. The generative properties of Restricted Boltzmann Machines, which are the building blocks of Deep Belief Networks, are not explicitly used for musical sound synthesis in the literature. Autoencoders, which are feedforward building blocks for deep

architectures have also not been studied as musical audio synthesizers.

In this work we suggest that the features learned by such networks may be directly modified to generate new musical audio. We 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 [10] which wraps around Theano [11] 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 freely on github at https://github.com/woodshop/deepAutoController. We hope that this paper encourages other researches to examine how this highly adaptable class of models may be used for creative musical tasks.

2. AUTOENCODERS

A classical autoencoder [1] is a deterministic feedforward ANN comprised of an input layer, a hidden layer, and an output layer (see Figure 1). We note that some authors place Restricted Boltzmann Machines (RBMs) in the same class as autoencoders. We follow the convention in [12] and use therm term "autoencoder" when speaking of the deterministic feedforward networks and reserve RBMs for the stochastic generative variant.

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 $\mathbf{x} \in [0, 1]^d$ to a hidden representation $\mathbf{y} \in [0, 1]^e$. It also learns a mapping (decoding) from $\mathbf{y}$ to the output layer $\mathbf{z} \in [0, 1]^d$. 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

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

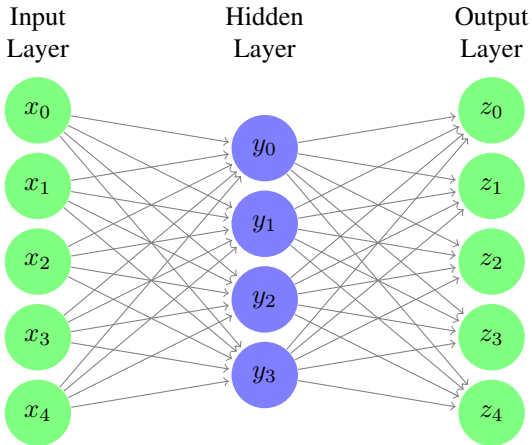[1] Autoencoders are also known as autoassociators.

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

where $s(\cdot)$ indicates an activation function (often a logistic function such as the sigmoid or hyperbolic tangent). Activation function(s) for neural networks 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:

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

and cross entropy:

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

The architecture 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 some sparsity or 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* [12], in which the model learns to reproduce the input from a corrupted version of the input, There are a number of corrupters used in practice, e.g. gaussian distributed noise, dropout, and salt and pepper corruption.

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 predominant strategy for building deep neural networks.

Classical autoencoders are not inherently generative. They are feedforward deterministic graphical models. Yet once a shallow or deep autoencoder has been successfully trained, we may expose the activations of the hidden units to a human operator. 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 entirely and replace a subset of the hidden units with our own streaming values, propagating them through the decoding half at an appropriate audio rate.

3. MODELS

As a proof of concept for an autoencoder synthesizer, we trained several models and built an interface for manipulating the models. 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

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 size 2. In the final variant we build a deep composed autoencoder by taking the hidden layers of our two pretrained autoencoders and finetuning the weights of the whole system to improve reconstruction of the original input.

We note that this is the order of events usually employed 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", sometimes by additionally attaching one last layer for softmax classification.

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.

	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.

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 as a denoising regularizer.

3.3 Discussion of Training Results

Table 1 shows the best mean squared error on the validation set for each model that was trained. It is clear that the smaller networks that employed no denoising perform the best. The optimal number of hidden units 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 (such as no denoising). Adding some corruption to the model with 1500 hidden units appears to improve results slightly.

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 after the first few frequency bins. The figure does not emulate depict desirable behavior for an optimal autoencoder.

Table 2 shows the validation performance of a second autoencoder trained on the hidden units of a first autoencoder. Once again smaller networks perform better than large ones and denoising does not appear to help much.

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 fine-tuned models perform worse than the deep architectures presented in Table 2, suggesting that the learning rate may
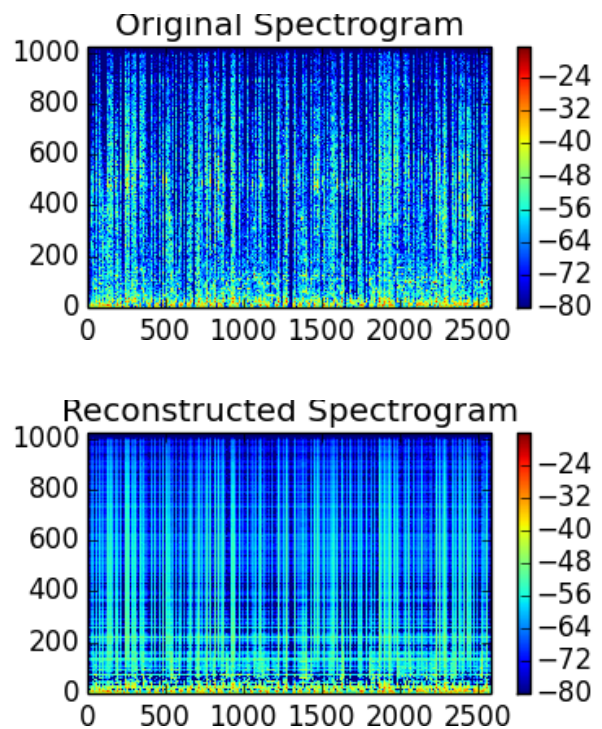

Figure 2. Top: STFT of original audio file. Bottom: STFT of reconstructed audio file.

		Noise		
HL1	HL2	0.00	0.10	0.25
256	8	**0.0184**	0.0184	0.0184
256	16	0.0184	0.0184	0.0184
256	32	0.0184	0.0184	0.0184
1500	8	0.0485	0.0484	**0.0482**
1500	16	0.0509	0.0506	0.0504
1500	32	0.0521	0.0540	0.0562

Table 2. Mean squared validation error for second-layer pretraining of deep autoencoders. The input/output layers of each model is designate 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.

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 to reduce overall reconstruction error.

have been inappropriate. Overall more complex models perform much worse than the simpler topologies.

Neural networks have lots of hyperparameters over which to search, including learning rate, regularization, layer width, and model depth. The training results presented here indicate that the hyperparameters chosen for the models were probably inappropriate. Further work is needed to examine how these models might be improved. It is also notable that the input features—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 given an optimal autoencoder the reconstruction will be cleaner than if we use band-limited features. Future models may employ preemphasis and deemphasis stages to move average amplitudes of higher frequency bins closer to the middle of the models' operating range.

4. INTERFACE

We programmed a real-time interface for interaction with the hidden units of deep or shallow autoencoders. Which ever 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 is trivial. The code is freely available on github (`https://github.com/woodshop/deepAutoController`) and 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 two input arguments: the path to a pickled Pylearn2 model and the path to an audio file.

At the initialization of the application a Python Queue is instantiated for message-passing between the Autocontrol class and the PlayStreaming classes. 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 package Pygame to poll for
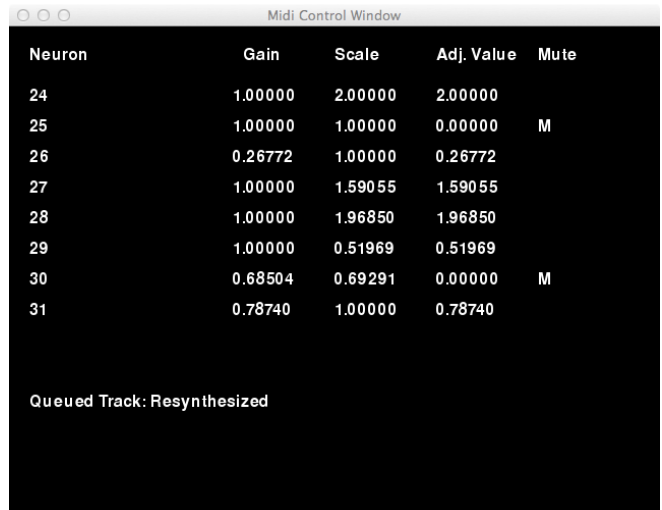

Neuron	Gain	Scale	Adj. Value	Mute
24	1.00000	2.00000	2.00000	
25	1.00000	1.00000	0.00000	M
26	0.26772	1.00000	0.26772	
27	1.00000	1.59055	1.59055	
28	1.00000	1.96850	1.96850	
29	1.00000	0.51969	0.51969	
30	0.68504	0.69291	0.00000	M
31	0.78740	1.00000	0.78740	

Queued Track: Resynthesized

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

MIDI events and produce informational output in a separate window 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.

- 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 and an 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 and transformed to FFT 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 by the user's settings, and the output is decoded (Eq. 2 back to an FFT frame, converted to the time domain, and sent to the output.

The current version of the class only works with audio files. But the class will be extended in the near future so that it can also take sound from the computer's audio input, or operate in a no input mode. In the latter case, the class will "fire" the hidden layer at an appropriate audio rate while the user maintains full control over the scale of the hidden layer units. A channel vocoder will provide phase continuity for the inverse FFT.

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 immediate future additional effort will be placed toward building more robust models. This may require preprocessing the input data so that it is more appropriate for neural networks.

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. 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] 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

[11] 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.

[12] 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.*,