DeepSynth: Synthesizing A Musical Instrument With Video

Tal Stramer
Stanford University
Stanford, CA
tstramer@stanford.edu

Abstract

This paper introduces DeepSynth, an end-to-end neural network model for generating the sound of a musical instrument based on a silent video of it being played. We specifically focus on building a synthesizer for the piano, but the ideas proposed in this paper are applicable to a wide range of musical instruments. At a high level, the model consists of a convolutional neural network (CNN) to extract features from the raw video frames and two stacked neural autoregressive models, one to encode the spatiotemporal features of the video and the other to generate the raw audio waveform.

1. Introduction

Building a digital instrument like an electronic piano requires multiple tedious steps: 1) Creating an input interface for entering notes and note characteristics, such as a physical keyboard or a graphical user interface, and 2) building a digital synthesizer, which traditionally requires manually creating a large number of tables that map notes and note characteristics to wave-tables.

With the advent of virtual and augmented reality, an interesting question arises: Can we simplify the creation of realistic, playable virtual musical instruments? Doing so currently requires manually modeling the 3D geometry of the desired instrument as well as manually modeling how interactions with the instrument map to raw audio. Instead, it would be convenient if we could directly map a set of videos showing examples of the musical instrument being played to a playable 3D virtual model of it. We note that video is a natural input in this setting, as the input from a virtual reality headset (3D video frames) is similar to the input to a video (2D video frames).

There are two distinct problems here: 1) Mapping a set of videos depicting a musical instrument to a 3D model of it, and 2) mapping the interactions with a musical instrument to the outputted audio. In this paper, we tackle the latter problem, namely mapping a video of a musical instrument being played to the audio outputted by that musical instrument.

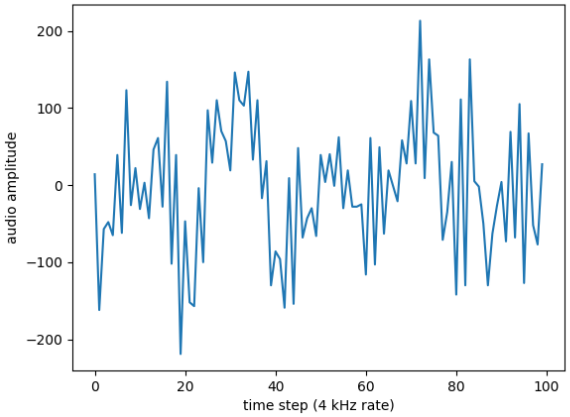
The ability to reconstruct the sounds outputted by a musical instrument based solely on a silent video of it being played is a challenging task. It requires one to extract the complex spatiotemporal features of a video and use this to model the raw audio waveform of the instrument. Furthermore, at a realistic video frame rate and audio sample rate, the inputs and outputs will be very large.

One can think of the task more concretely as a two-step process. The first step is to extract the notes being played from the video, and the second step is to map the notes to raw audio. Both of these tasks could be done in isolation, however, that would transform the problem into a supervised learning problem, as each video would have to be hand-labeled with the notes being played at each frame. Instead, we formulate this as an unsupervised learning problem by training an end-to-end neural network that attempts to learn how to generate audio directly from the pixels of the video frames, without having to hand label the notes being played. The hypothesis is that the features the model learns to extract from the video through training will be semantically similar to the underlying notes being played.

More specifically, our model consists of stacked autoregressive models to encode the spatiotemporal features of the video and to generate the raw audio waveform. By using autoregressive models for both the video and audio, we ensure that the model predicts the raw audio waveform using only features of the video up to and including the current time step as well as features of the audio waveform generated up to but not including the current time step. This matches the problem constraints one would expect to encounter in a real-word application, where future predictions can only rely on current and past information.

(a)

Figure 1: Single video frame of piano simulator



(a)

Figure 2: Audio sample from piano simulator

The training data is generated based on a piano simulator we developed that takes a MIDI file of a song as input and generates 1) a silent video of a fake piano playing the song, highlighting the keys being pressed and 2) the audio waveform of what the song would sound like if played on a grand piano. See figure 1 for an example video frame and figure 2 for an example snippet of the generated audio waveform. We note that the choice of grand piano is arbitrary - our model could be used to mimic the sound of a number of different musical instruments. The MIDI files are taken from an online dataset of classical piano songs [2].

2. Related Work

There is a large body of work on video and audio processing using deep learning. Most notably, the WaveNet paper [1] showed it is possible to build a neural autoregressive model to generate raw audio with high temporal resolution, on the order of 16 kHz, which achieves close to human-level speech. There work is inspired by PixelCNN [12], a generative model for images using CNNs. The WaveNet paper presented both un-conditional and conditional versions of the model. They provide a few examples of possible contexts to condition on: 1) a global context, i.e. a speaker or 2) a local context, i.e. text to translate to speech. The biggest down-side with WaveNet is that at prediction time the audio waveform is generated sequentially, which can prove very slow given a realistic temporal resolution in the 10-45 kHz range.

More recently, Arik et al. [3] introduced DeepVoice, a end-to-end text-to-speech system using neural networks. DeepVoice uses a variant of WaveNet to generate the raw audio waveform, as well as a highly-optimized version of the WaveNet inference kernel that achieves up to a 400x speedup in prediction speed from previous baselines. This allowed them to output audio at 16 kHz in real-time. To achieve the speedup, they took great care to ensure optimal processor caching and utilization of computational units.

Another related work by Chung et al. [4] attempted to build a lip-reading system to recognize phrases and sentences being spoken by a talking face, both with and without audio. Their model consists of several components: a video encoder based on the VGG-M model [5] to generate image features for every input time step, fed into an LSTM network; a audio encoder consisting of 13-dimensional MFCC feature vector for each audio sample, fed into a separate LSTM network; and a character-level decoder also based on LSTMs which includes an attention mechanism. Since their model uses audio as input only, they are able to use much lower temporal resolution then is necessary to generate realistic sounding audio as output.

There has also been recent work to build digital synthesizers based on neural networks. For example, Engel et al. [7] built a Wavenet-style auto-encoder to generate raw audio based on a large dataset of notes across thousands of instruments and note characteristics like timbre and pitch. The auto-encoder is able to learn a semantically meaningful hidden representation that can be used to control various characteristics of notes in an intuitive way. Their model is also able to combine instruments and characteristics to generate new types of sounds not previously seen in the training set.

3. Methods

In section 3.1 we describe our model at a high-level. In section 3.2 we describe a major building block of our model, the dilated causal convolutional network. In section 3.3 and section 3.4 we describe the specifics of the video encoder and audio decoder. In section 3.5 we describe the output and loss function. In section 3.6 we discuss other features of the model used to help with performance and to speed up training. Finally, in section 3.7 we discuss other model variations that we experimented with that ultimately did not yield satisfactory results.

3.1. Model

The input into our model is a video represented by image frames $x = \{x_1, \ldots, x_n\}, x_i \in R^{wxh}$, where w

and h are the width and height of the gray-scale image. The output of our model is the raw audio waveform of the video, represented by samples $y = \{y_1, \ldots, y_m\}, y_i \in R$, where each sample is a 16-bit number representing the amplitude of the raw audio waveform at that time step.

Using the chain rule, the probability of our audio waveform can be factored into conditional probabilities as follows:

$$p(y|x) = \prod_{i=1}^{m} p(y_i|y_1, \ldots, y_{i-1}, x_1, \ldots, x_i)$$

Our goal is to find the most likely audio sample y given video x.

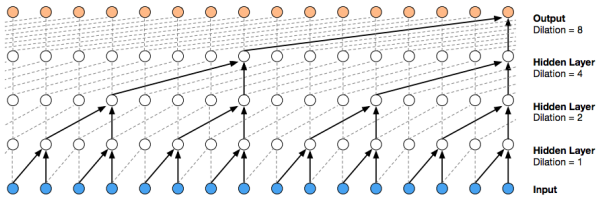
3.2. Dilated Causal Convolution Network

The main building block of our model is the dilated causal convolutional network [11]. This network is a type of autoregressive model that attempts to predict some time series based only on inputs from previous time steps. The network consists of a stack of convolutional layers with two additional behaviors:

1. Each convolutional filter is causal, which means every pre-activation value z_i can only be computed based on inputs $x_{<i}$ from the previous layer. This invariant ensures that the final output of the network at each time step will only be computed based on inputs at previous time steps.

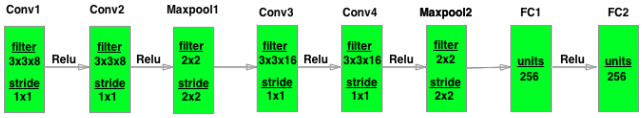
2. The dilation factors of the convolution filters at each layer are increased exponentially from the previous layer (1, 2, 4, 8, etc. up to 256 and then repeated). For context, a dilated convolution [6] is akin to a normal convolution except the filter is expanded based on the dilation factor by inserting zeros in the empty positions. The reason for doing this is to increase the receptive field at each layer. For a stack of standard convolution layers, the receptive field grows linearly with the number of layers, while for a stack of dilated convolution layers with exponentially increasing size, the receptive field grows exponentially. This has the advantage of increasing the context used to make a prediction at each time step without decreasing the resolution or the number of parameters in the network. Also, by using dilation instead of striding or pooling, the input, hidden, and output layers all have the same size.

A major advantage of this type of autoregressive model over for example an RNN [14] is that at training time we can compute the output predictions for each time step in parallel, which is considerably more efficient when dealing with a large time series. At test time, however, the time series must be generated sequentially,



(a)

Figure 3: Illustration of a dilated causal convolution network copied from [1]



(a)

Figure 4: Illustration of the frame encoder CNN

as we assume we do not have access to the input time series.

As in the WaveNet paper [1], we use a gated activation unit after each convolutional layer, defined as follows:

$$z_k = tanh(W_{f,k} * x_k) \odot \sigma(W_{g,k} * x_k)$$

where i is the time step, k is the layer index, * is a convolution operator, and $\odot$ is an element-wise multiplication.

Figure 3 shows an example of a dilated causal convolution network.

3.3. Video Encoder

We first encode each video frame into a feature vector. Specifically, for each video frame $x_i \in R^{wxh}$, we generate a feature vector $f_i \in R^l$ via a VGG-like [15] network consisting of a series of conv3 - conv3 - maxpool layers followed by a series of fully-connected layers, as illustrated in figure 4.

The image features $\{f_1, \ldots, f_n\}$ are then fed as time-series input into a dilated causal convolution network, as described in section 3.2. The output of this network is a time-series $f' = \{f'_1, \ldots, f'_n\}$, where each f'_i is a spatiotemporal feature representation of the video up to that time step. Note that the input video frames to the network are up-sampled to the same resolution as the audio, so that the network generates spatiotemporal feature vectors for the video at the same resolution as the audio time series.

3.4. Audio Decoder

To decode the raw audio waveform, we use a second dilated causal convolution network. In this case the network is estimating a conditional probability distribution $p(y|f')$. To achieve this, we use the same setup as described in section 3.2 with one modification to our activation unit:

$$z_k = tanh(W_{f,k} * x_k + V_{f,k} * f') \odot \sigma(W_{g,k} * x_k + V_{g,k} * f')$$

where i is the time step, k is the layer index, * is a convolution operator, and $\odot$ is an element-wise multiplication.

3.5. Classification and Loss

As mentioned previously, the audio waveform is a series of 16-bit integers representing the amplitude of the wave at each time step, for a total of 65,536 possible values. In order to reduce the complexity of our final prediction, we discretize the waveform into 8-bit values, for a total of 256 possible values. To do this, we first apply a μ-law transformation [9] with $\mu = 256$, which is a non-linear transformation of the audio waveform that has been shown to provide better reconstruction accuracy for human speech. Then, we discretize the transformed values into 256 values linearly. The final output from the audio decoder at each time step is a 256-dimensional vector representing a predicted probability distribution over the discretized audio waveform at that time step.

We use cross-entropy loss to minimize the divergence between our predicted probability distribution over the discretized audio waveform and the true distribution at each time step as follows:

$$L_{i,t} = -log(\frac{e^{f_{i,t,y_{i,t}}}}{\sum_j e^{f_{i,t,j}}})$$

where i is the training example, t is the time step, $f_{i,t,k}$ is the predicted probability of the discretized audio waveform taking on value k at time step t for example i, and $y_{i,t}$ is the true value of the discretized waveform at time step t for example i.

3.6. Additional Features

We add skip connections in both the video encoder and audio decoder networks. Skip connections work by using each layer of the network for prediction instead of only the last layer of the network. Specifically, we sum the outputs at each layer and then add a ReLU activation followed by two 1D convolutions to produce the final predicted probability distribution for that time step.

Residual connections [11] are also used at each layer of the video encoder and audio decoder. Residual connections calculate the output of each layer as the sum of the input to that layer plus a residual. More specifically, instead of calculating the layer output as $y = f(x)$, a residual network calculates the output as $y = f(x) + x$. This allows the network to learn more incremental representations.

We use l2 regularization to prevent overfitting the data and find that this works better for audio generation then dropout [13]. Additionally, we use learning rate annealing to speed up training.

3.7. Model Variations

In this section we describe a few model variations that ultimately did not yield a useful representation for learning how to map video to audio.

One model variation used a regression loss instead of a softmax loss. The use of a regression loss is intuitively appealing as the numerical difference between amplitudes seems like a natural way to interpret prediction error. With a softmax loss, on the other hand, the model is free to learn a more general probability distribution. We found that both losses work similarly well on the training set, however regression loss did significantly worse on the validation set. We hypothesize that this is due to the more limited representational power of regression loss which does not generalize as well.

Another model variation we tried introduced an auto-encoder style component to predict the feature vectors of each video frame based on the audio and then performing a softmax loss between the generated feature vectors for each frame and the predicted ones. This variation did not work well. We hypothesize that this is because the audio is sampled at a much higher rate then the video frames, so it is difficult to map the audio back to specific video frames / time steps.

4. Dataset

As mentioned previously, the dataset is generated based on a piano simulator that takes a MIDI file of a song as input and generates a video as output. Figure 1 shows an example of a single video frame of the simulator and figure 2 shows a snippet of the generated audio waveform after discretization.

We convert each video frame into a gray-scale image by extracting the y-channel from the full RGB image. Each image is of size 780x57. We use a sparse representation of the video by capturing the frames where the notes change as well as the timestamp of each frame.

The audio waveform is generated based on a digital synthesizer of a grand piano sampled at 4 kHz. Each audio sample is a signed 16-bit integer representing the amplitude of the wave at that time step. As noted previously, we transform each audio waveform into a 8-bit value using a μ-law transform.

To generate the data, we feed around 100 classical songs taken from [2] into the simulator. Each song is around 2-3 minutes. In order to break this up into manageable size chunks for our model, we split each song into 5 second segments with 2 second overlapping window between segments. In total we end up with $\approx$ 3,800 segments.

We use 60% of the data for training, 20% for validation, and 20% for testing.

5. Training

We train each model using stochastic gradient descent and the Adam optimizer [8]. The models are coded in Tensorflow [16] with a few code snippets taken from a Wavenet implementation by Google [17]. We use a learning rate schedule starting at 1e-3 and decreasing in schedules every 20,000 batches down to 1e-6. We train with a batch size of 3 segments due to memory constraints. We use gradient clipping with a max gradient norm of 10. We train each model for 200 epochs. We calculate the average loss after each epoch over 100 randomly selected segments from the validation set, and choose the model with the best validation performance.

During training and validation, we generate each audio sample in parallel. To achieve this, each audio sample is generated based on the true audio samples from the previous time steps, rather than the generated ones. At test time, we generate audio samples sequentially, sampling the audio amplitude at each time step based on the probability distribution outputted by the model at that time step.

6. Experiments and Results

We experiment with various hyper-parameters, including:

1. L2 regularization strength: 1e-2, 1e-4, 1e-5
2. Number of dilation layers: 15, 20, 27. Note that the dilation factors per layer are as follows up to the maximum number of layers: $[1, 2, 4, ..., 128, 256, 1, 2, 4, ..., 128, 256, ...]$. Each number of dilation layers yields a different receptive field size, namely: 143ms for 15 layers, 256ms for 20 layers, and 383ms for 27 layers (based on a 4 kHz audio sampling rate).
3. Number of convolution channels/features for each neuron in the dilation networks: 128, 256, 512

The results are summarized in table 1. Note that results for other model variation presented in section 3.7 are not included as they fail to achieve reasonable performance.

Reg strength	Layers	Channels	Train Loss	Val Loss
1e-2	15	128	2.23	2.87
1e-4	15	128	1.87	2.35
1e-5	15	128	1.65	2.46
1e-4	20	128	1.51	2.26
1e-4	27	128	1.21	2.18
1e-4	**27**	**256**	**1.16**	**2.15**
1e-4	27	512	1.14	2.16

Table 1: Experiment results.

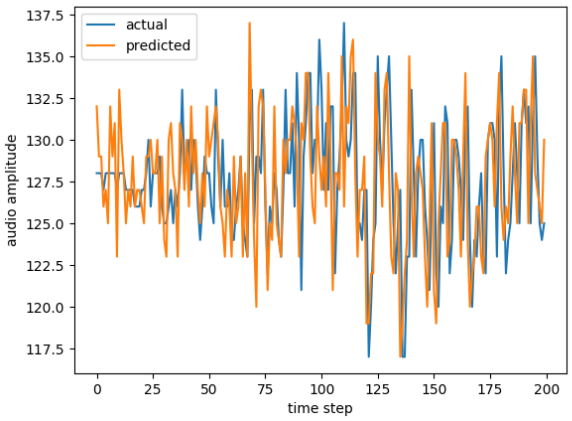
As can be seen in the table, the best performance on the validation set is achieved with a l2 regularization strength of 1e-4 and 27 layers / 256 channels per convolution in each of the video encoder network and audio decoder network. In general, too low of a l2 regularization strength led to overfitting, and too high led to the model under fitting, as well as generating complete noise at test time. The number of layers has the biggest impact on train and validation performance. The number of channels makes the least difference, and in fact after 256 does not seem to improve performance. We note that more hyper-parameter combinations were tested, but are left out here for brevity.

In addition to evaluating loss on the training and validation set, we generated some samples sequentially on the test dataset and evaluated them qualitatively. For all but the last three hyper-parameter configurations in table 1, the model generates mostly noise. On the last three configurations, the model is able to generate audio that clearly mimics the underlying sample, but also includes what sounds like static noise in the background and is in general quieter than the original sample.

Figure 5 shows a plot of a snippet of an audio waveform from the validation set overlaid with the predicted audio waveform from our best model. Figure 6 shows the loss rate for the first 26,000 iterations ($\approx$ 150,000 total iterations).

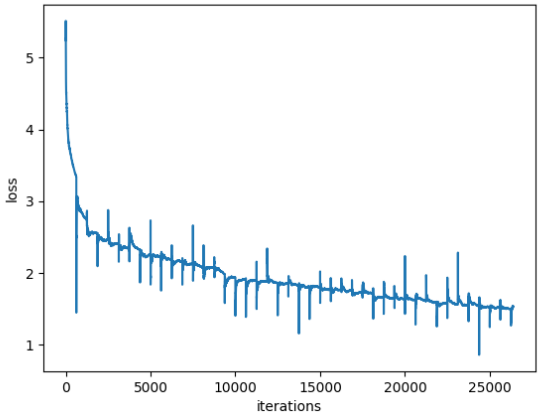
7. Conclusion

In this paper, we presented DeepSynth, a new neural network model for generating the raw audio waveform outputted by a musical instrument based on a video of the musical instrument being played. The model uses a series of convolutional neural networks: one to encode each



(a)

Figure 5: Actual and predicted audio waveform



(a)

Figure 6: Loss per iteration

video frame, one to encode the spatiotemporal features of the video, and one to decode the raw audio. We used exponentially increasing dilation factors to be able to capture long-range dependencies in the audio, and showed that our model can generate audio at a high accuracy and quality on a validation and test set.

The model requires no manual feature engineering or labeling, which is a huge benefit. In the future, we would like to use real videos instead of simulated ones, like piano videos from the Youtube 8M dataset. In addition, we would like to invest time into building an efficient generator by using low-level optimizations, so that the system is useful in real applications.

References

[1] Van den Oord, Aron, et al. "Wavenet: A generative model for raw audio." CoRR abs/1609.03499 (2016).

[2] Krueger, Bernd. "MIDI Files." Classical Piano Midi Page. http://www.piano-midi.de/beeth.htm

[3] Arik, Sercan O., et al. "Deep Voice: Real-time neural text-to-speech." arXiv preprint arXiv:1702.07825 (2017).

[4] Chung, Joon Son, and Andrew Zisserman. "Lip reading in the wild." Asian Conference on Computer Vision. Springer, Cham, 2016.

[5] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutionalnets. In Proc. BMVC., 2014.

[6] Yu, Fisher, and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions." arXiv preprint arXiv:1511.07122 (2015).

[7] Engel, Jesse, et al. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders." arXiv preprint arXiv:1704.01279 (2017).

[8] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[9] Recommendation, C. C. I. T. T. "Pulse code modulation (PCM) of voice frequencies." ITU (1988).

[10] He, Kaiming, et al. "Identity mappings in deep residual networks." European Conference on Computer Vision. Springer International Publishing, 2016.

[11] Yu, Fisher, and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions." arXiv preprint arXiv:1511.07122 (2015).

[12] Oord, Aaron van den, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel recurrent neural networks." arXiv preprint arXiv:1601.06759 (2016).

[13] Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[14] Medsker, L. R., and L. C. Jain. "Recurrent neural networks." Design and Applications 5 (2001).

[15] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[16] Abadi, Martn, et al. ”Tensorflow: Large-scale machine learning on heterogeneous distributed systems.” arXiv preprint arXiv:1603.04467 (2016).

[17] Babuschkin, Igor, ”A TensorFlow implementation of DeepMind’s WaveNet paper.” (2016), GitHub repository, https://github.com/ibab/tensorflow-wavenet