CS230

Neural Synthesis of Piano Performance

Patricia Lan[1], **Steven Leung**[1], **and Grant Yang**[2]
[1]Department of Bioengineering, [2]Department of Electrical Engineering
Stanford University
pslan@stanford.edu, leungsa@stanford.edu, granty@stanford.edu

Abstract

In this project, we designed and trained a deep learning model to "play" classical piano music by learning the relationship between notes from musical scores and recordings of virtuoso pianists. Our network consisted of a bidirectional RNN sandwiched between fully connected layers, and was trained with both time domain and log magnitude spectrogram MSE. From our results, we can see that the model learned to play the sound of each piano key. It also learned rudimentary musical phrasing via changes in dynamics (relative loudness of nearby phrases). Sample audio outputs can be found at https://goo.gl/gcgJbA.

1 Introduction

Performing music from musical scores is an ill-conditioned problem. The audio representation of a performance contains significantly more information than the original score. Musicians must make decisions about tone, tempo, and dynamics based upon complex non-causal relationships between the musical notes, phrases, and sections. In this project, we designed and trained a deep learning model to "play" classical piano music by learning the relationship between notes and rhythms from musical scores and recordings of virtuoso pianists.

Like a pianist, the model reads in a digital "score" consisting of a matrix encoding in which piano keys are pressed or held at a given time. The model uses a combination of fully connected layers and recurrent layers [1]. The fully connected layers help the model learn the relationship between the notes on the score and the sound produced by the piano (fundamental and harmonic frequencies). The recurrent layers help the model learn the temporal dynamics of the sound after the note is struck, as well as how loudly the notes should be played based on the surrounding notes (dynamics). The model outputs a spectrogram representation of the performance, which is converted to an audio performance using an inverse short time Fourier transform (iSTFT).

2 Related work

Digital piano synthesizers historically relied on looping samples of recorded piano sounds [2, 3]. However, the use of recorded sounds has large memory requirements, which limits the ability to reproduce the full range of piano sounds (for example, due to discretization in intensity levels). Recently, mathematical models have been used to model the continuous timbral changes from real acoustic pianos [4, 5]. Machine learning models have also been applied to sound generation [6, 7]. WaveNet [8] is Google DeepMind's deep generative model of raw audio waveforms. It takes raw signal as input and synthesizes one sample at a time, similar to a nonlinear infinite impulse response filter. It consists of a deep convolutional neural network, where each layer has varying dilation factors, allowing its receptive field to grow exponentially with depth and cover thousands of time steps. NSynth [9], a collaboration between Google Brain and DeepMind, is a novel approach to

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

music synthesis based on the WaveNet architecture. NSynth consists of a WaveNet-style autoencoder that conditions an autoregressive decoder to learn temporal embeddings. This allows the network to morphing between instruments by interpolating between timber and dynamics. NSynth and WaveNet have demonstrated the ability to generate realistic sounds for text to speech, as well as various instruments including the piano. The generative model has also been shown to generate random babbling or note sequences. However, all of these synthesizer generation methods require a performer to instruct them on how and when to play a specific musical piece. The problem of musical style was tackled by Malik et al. in their work named Neural Translation of Musical Style [10]. Malik's StyleNet used bidirectional recurrent neural networks (RNNs) with long short-term memory (LSTM) to learn the relationship between notes and velocities (how loud the note is played). StyleNet takes the notes and rhythms from MIDI recordings as inputs and attempts to learn the MIDI velocities. In our work, we implemented an end-to-end approach that attempts to synthesize the piano sound and the musical interpretation directly from the musical score.

3 Dataset and Features

We used the Piano Dataset curated by Malik et al. [10]. This dataset is comprised of 349 classical piano performances recorded in the MIDI file format. We inputted the MIDI files into GarageBand to create audio files of the performances. The data was then reformatted for the network model.

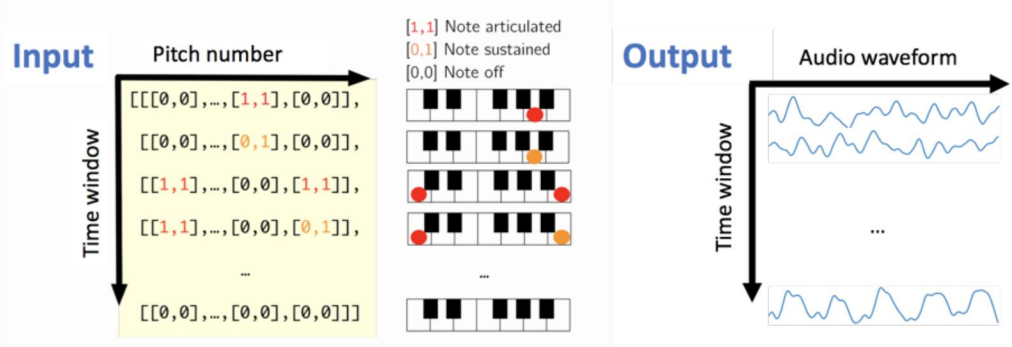


Figure 1: Visual representation of the input and output data.

Inputs: pitch encoding matrix [examples, time windows, 176] The MIDI files contain timing, pitch (frequency), and velocity (loudness) of played notes. They were reformatted into a pitch encoding matrix, which has dimensions (T, 176). The first dimension is the time window dimension. T is the number of 50 ms time windows in the performance. Our base model had T = 80, so that each example consisted of 50ms * 80 = 4s worth of data; we also experimented with other time windows. The second dimension is the pitch encoding. We encoded 88 pitches using two numbers: 1) whether the note was struck, or articulated, in a particular time window and 2) whether the note was sustained. This pitch encoding is described by Malik et al. (Figure 1).

Outputs: audio matrix [examples, time windows, audio samples] The audio signals were sampled at 8 kHz and split into 50 ms windows. The formatted audio matrix has dimensions (T, 400). The first dimension is the time window dimension where T is the number of 50 ms time windows, and the second dimension is the time dimension (Figure 1).

Each performance was split into multiple examples, each containing 4 seconds worth of audio data (T = 80). We used a total of 30 performances and used a train/dev/test split of 80/10/10, with 24 performances in the train, 3 in dev, and 3 in test.

4 Methods

Our final model consisted of a bidirectional RNN architecture sandwiched between two time-distributed fully connected feed forward networks (Figure 2). The first fully connected network learned to take pitch encoding inputs and output spectrograms. The RNN took the spectrograms as

inputs and added temporal dependencies to them (note decay and other dynamics). The last fully connected layer output the final log magnitude spectrograms.

We used mini-batch (8 examples) gradient descent to increase learning speed, while still allowing for vectorization. For the optimization algorithm, we decided on using Adam optimization, which combines momentum and RMSprop.

We implemented our model using the Keras API [11] with the TensorFlow backend [12]. NumPy and Pandas from the SciPy library [13] were used to perform matrix operations and manage the data. Our github repository can be accessed at https://github.com/stevenaleung/CS230-final-project.git.

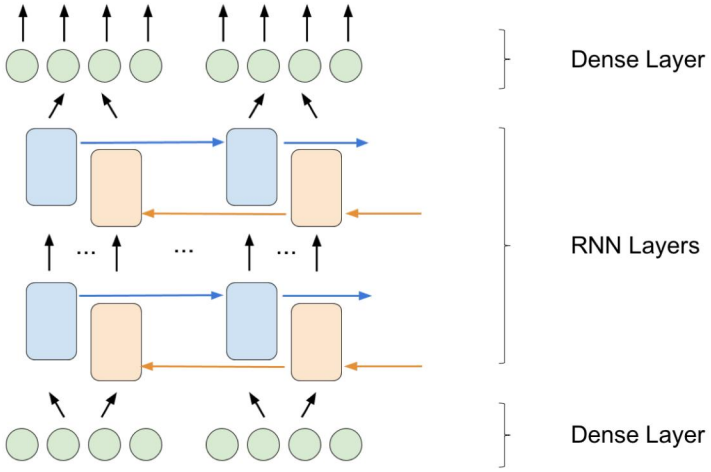


Figure 2: Schematic of our model architecture.

5 Hyperparameter tuning

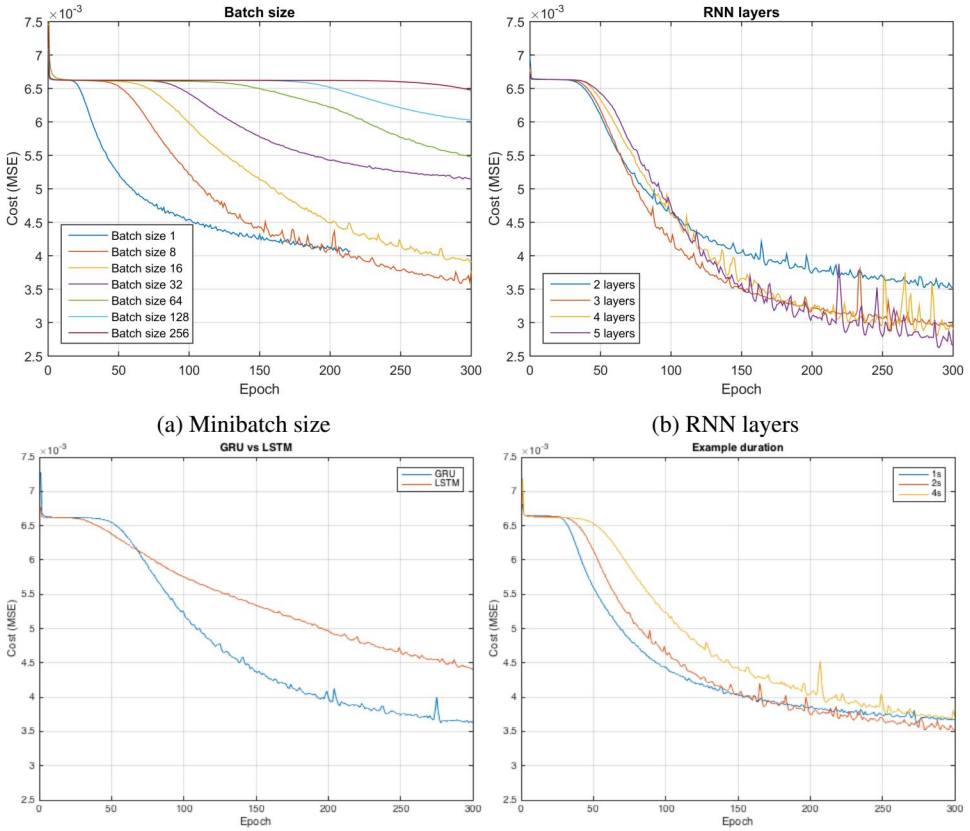
We used the mean squared error (MSE) cost function to test the performance of multiple hyperparameters, architectures, and data formats. Our initial default model consisted of 2 bidirectional GRU layers with 128 hidden units and a fully connected output layer. The options included mini-batch size, number of RNN layers, RNN cell type, and example duration.

Figure 3a shows the effect of mini-batch size on training efficacy. A mini-batch size of 8 appeared to perform the best; across 300 epochs it resulted in the lowest MSE. Because weight and bias gradients are averaged across examples within a mini-batch, larger mini-batch sizes could result in diminished gradients. Our mini-batch size of 8 was smaller than typical RNN mini-batch sizes of 32 to 128 - this may be due to the formatting and representation of our data.

Figure 3b shows the effect of RNN layers on training efficacy. Five layers resulted in the lowest MSE over 300 epochs, though the improvement over three and four layers was minimal (MSE 0.002). We therefore chose to use two layers as a compromise between training performance and computation time.

Figure 3c shows the effect of RNN cell type on training efficacy. Across 300 epochs, we found GRU cells to perform better than LSTM cells. We believe this is because the GRUs were robust enough to capture long distance relationships in the music audio. The simpler construction of GRUs allowed them to train faster than LSTMs.

Figure 3d shows the effect of example duration on training efficacy. There did not appear to be much difference in MSE between different durations, though there were some differences in training speed in earlier epochs. We decided to use example durations of 4 seconds in the hopes that the network could capture longer distance relationships.



(a) Minibatch size

(b) RNN layers

(c) RNN cell type

(d) Example duration

Figure 3: Hyperparameter tuning

6 New loss function: log magnitude spectrogram MSE

As an extension, we decided to test whether the log magnitude spectrogram (LMS) MSE loss would improve training performance. We found that the network trained faster with LMS MSE compared to time domain MSE (it required only 100 epochs to achieve similar results to the 1000 epochs of time domain MSE loss). After further tuning on the log magnitude loss spectrogram, we determined that placing 4 time distributed dense layers allowed us to reduce the number of recurrent layers while improving model performance. Our final network architecture used a 1 layer bidirectional RNN with GRU cells, 200 hidden units, sandwiched between four fully connected layers and 1 fully connected output layer. For training we used a mini-batch size of 8, and an example duration of 4 seconds.

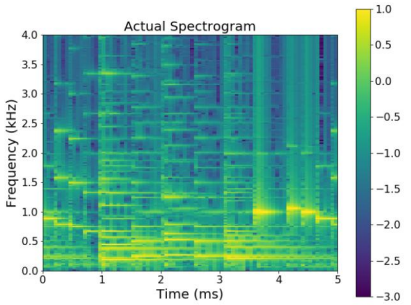
7 Results

The model outputted log magnitude spectrograms, which were then transformed to the time domain using iSTFT. In Figure 4b, we can see that the model learned to represent note decay over time and temporal dynamics (relative loudness of nearby phrases).

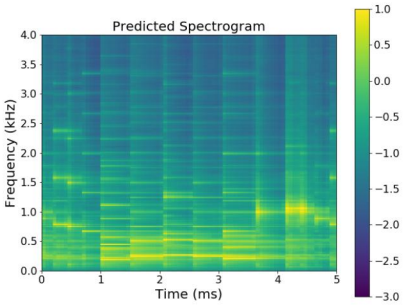
Training error went down to 0.0031 on average and test error was 0.0062 on average. For reference, white noise with a standard deviation 0.078 will result in a MSE of 0.006. Sample audio outputs can be found at https://goo.gl/gcgJbA.

8 Discussion

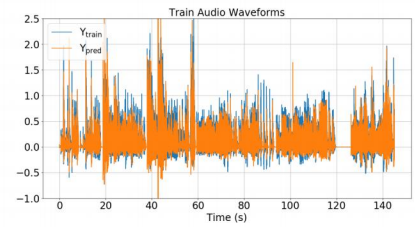
Using the log magnitude spectrogram (LMS) loss caused the model to train drastically faster than when using the time domain MSE loss. In fact, using the new cost function made the largest difference - it was more influential than tuning the hyperparameters and network architecture.



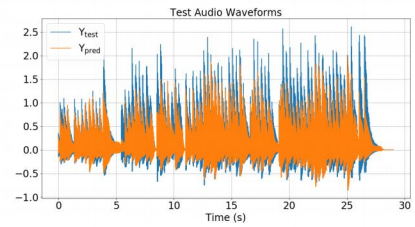
(a) Actual spectrogram



(b) Predicted spectrogram



(c) Prediction of train example



(d) Prediction of test example

Figure 4: Results

One downside of using LMS is that we lost all phase information. Taking the magnitude of the spectrum effectively causes all frequencies to be in phase, therefore transforming back to the time domain resulted in larger signal amplitudes than the original audio waveform (Figures 4c, 4d).

Interestingly, many of our training runs started out with an approximately 40 epoch long plateau before errors started to noticeably decrease. Hitting a saddle point this early in training was unexpected behavior and led us to believe our models were not working. However, we learned to let the training run for many more epochs before concluding whether a certain network architecture and set of hyperparameters was working.

9 Conclusion/Future Work

Using a bidirectional RNN sandwiched between fully connected layers allowed us to successfully generate recognizable audio directly from our digital 'scores'. The choice of loss function had the largest effect on training convergence and the quality of the resulting model. Training on a mean square loss in the time domain resulted in slow convergence and over-fitting, because there was very little structure in the time domain waveform. Training on the spectrogram loss resulted in much faster convergence, and better performing models.

The loss of phase information using the log magnitude spectrogram was a major impediment to achieving natural sounding audio. In the future, we would like to explore ways to include phase information in our loss function such as using the real and imaginary spectrograms in the loss function. There are also more advanced reconstruction methods for STFTs such as the Griffin-Lim algorithm [14] which approximates the missing phase information.

We would also like to include additional information such as dynamic and tempo markings as inputs to our model. One important inclusion would be pedal markings. The sostenuto pedal on the piano allows notes to continue after the keys are released and can have a dramatic effect on the timbre and temporal evolution of the piano sound. Finally, we would like to train our model on different instruments. Current synthesizers have great difficulty generating realistic performances for stringed instruments due to the complex array of ways that a note can be played. Deep learning using recurrent networks should have the ability to better model the performers choices and result in a more natural synthesized performance.