Machine Learning Engineer Nanodegree

Capstone Project

Justin Todd
June 2, 2017

I. Definition

Project Overview

Automated musical transcription is the process of having a machine process a recorded piece of music and turn it into a digital format which encapsulates the piece in a more musical form. This more musical form should be enough to act as an instruction list which, when followed, is enough to reproduce the original piece of music.

Traditionally music is transcribed in a visual manor with symbols indicating pitch, key signature, and note duration along with any additional accents and notes which might be required by the human performer of the piece. For machines to understand a transcription of music there is another story. MIDI (short for Musical Instrument Digital Interface) is a technical standard for describing similar musical information, but which can more easily interpreted by machines.

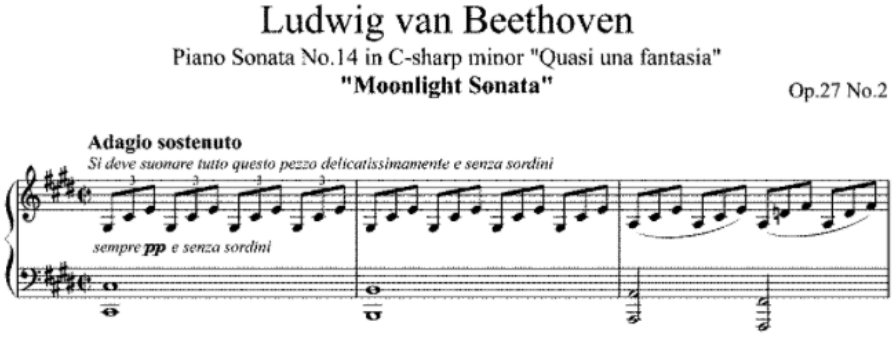

Figure 1: Traditional Notation

Instead of note durations and measures, MIDI (at least for the subset of the format which we are focused on today) involves a list of messages and the number of "ticks" at which those messages occur in the musical track. A single, typical message may encompass the following information: note on at tick 400, pitch 24, velocity 64. This translates to: Play a low C 0.92 seconds into the track at a medium volume.

Problem Statement

The goal is to be take a digital recording of a piece of monophonic piano music and produce a MIDI track which is able to reproduce the original music.

The tasks involved are like so:

1. Generate a midi track which simulates some measure of musicality which incorporating a lot of randomness.
2. Use the generated MIDI track and a virtual piano instrument to create a simulated digital recording of a piano to use as a digital signal for input.
3. Process the digital signal and the generated MIDI track into features and labels which can be used for training a model.
4. Build and train the classifier.
5. Have the trained classifier analyze a recorded piece of music, produce a MIDI track, and use a virtual piano to create a digital recording.

Metrics

Accuracy is a common metric for binary classification, but in this problem is not a good metric because most of the frames of music we are analyzing contain no events–they are merely space between note stops and starts. Accuracy is defined as the sum of true positives and true negatives divided by the size of the dataset. In this case a classifier which never chooses an even for a given frame of music tends to have a higher than 99% accuracy.

Therefore we will be looking at precision for the primary metric of this task and recall as an auxiliary metric.

Precision is the number of true positives divided by the sum of true positives and false positives.

Recall is the number of true positives divided by the sum of true positives and false negatives.

Precision is chosen as it will tend to favor making sure the notes chosen are correct. This may lead to the correct notes not being chosen at all, but false positives are more damaging to the result as wrong notes result in cacophonous music.

Recall is also being tracked as it can show if notes are being missed, but I'm greatly favoring precision in this case because I am saying it is better to stay in key than miss some notes.

II. Analysis

Data Exploration

The data set for this project was generated in whole by a script that I wrote. The method for generating the music is an attempt to get enough coverage of pitches and velocities while trying to imitate musicality. Music is typically composed within scales. A scale is usually comprised of 7 notes per octave. So I set off to generate random notes within scales and close to each other pitch-wise by "running" through the scales and randomly changing the direction of the run.

The script worked like so:

```
let d be a direction (up or down) in which to run the scale
for each scale:
    for each tick duration:
        emit 100 notes like so:
            while cycling through velocities in a sine wave,
            stop the previous note (if there is one)
            start the next note of the scale in direction d
            randomly with probability 0.1 change direction of d
```

The following tick durations were used (a tick is 2.3ms in this project): 40, 60, 80, 150, 200

The following scales and arpeggios were used to generate notes: Chromatic (every note), major, major 3rd arpeggios, 7-chord arpeggios, major 7 arpeggios, major-minor 7 arpeggios, minor, minor harmonic scale, minor arpeggios, whole tone scale, octatonic diminshed scale, augmented, fifths, octaves, minor pentatonic, major pentatonic, and the blues scale.

The MIDI files were run through a piano virtual instrument to produce audio. VS Upright No. 1 is a free virtual instrument provided by Versilian Studios which was used for the audio. The audio was then downsampled using Sound eXchange to 16,000 samples per second.

Exploratory Visualization

The primary file used in this project is "Runs in A". The track contains 45,000 note-on events and 45,000 note-off events. The distribution plot below indicates that my method for generating the data while trying to imitate musicality as resulted in a fairly uniform distribution of pitches. This, we have an ample data set to train and test from.

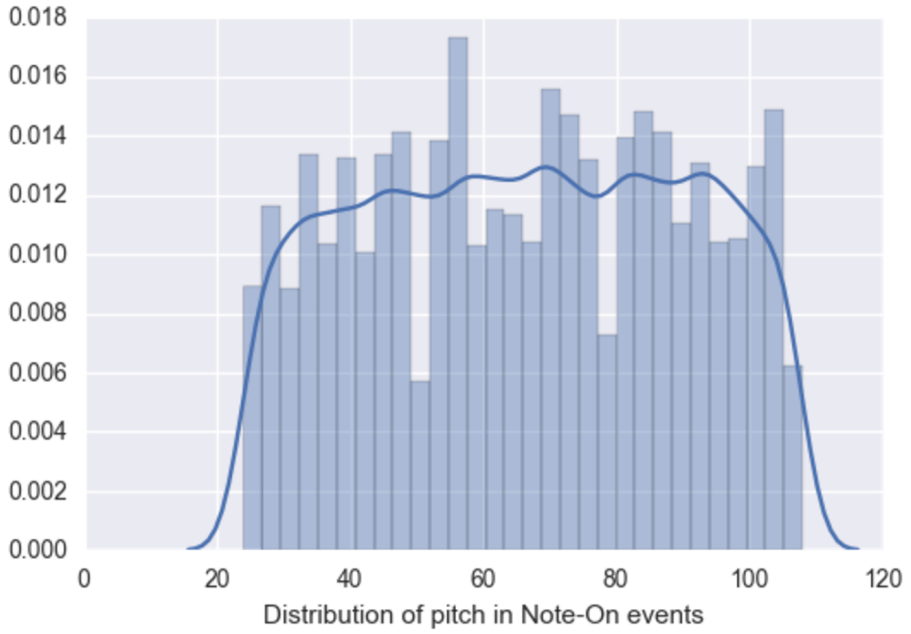

Figure 2:

Algorithms and Techniques

The data will be preprocessed by dividing the music into 40ms frames of frequency centers and magnitudes for use as features and note on/off events to use as labels. The frequency centers and magnitudes are obtained using a fast Fourier transform. See Data Preprocessing for more details.

The classifier being used is a simple Artificial Neural Network (ANN).

An artificial neural network is modelled after the brain. Similar to how neurons send and receive signals via synapses, a neuron in an ANN receives many different inputs simultaneously and sends a signal to other neurons depending on that value. Neurons in an ANN are grouped into layers. Each neuron in a layer sends its output to each neuron in the next layer. The output signal of a neuron is determined by taking all of the input values from the previous layer, mathematically combining them with a set of randomly initialized weights, and passing the single resulting scalar value through an "activation function." The activation function typically bounds the resulting signal between 0 and 1.

The final layer in the ANN will be a single vector which contains exactly as many classes as we have to predict. The value for each class is the probability that that class is true. A class in this case represents the pitch of the note and whether it is being turned on or off. We will use some threshold to determine if

we will actually count that as a prediction of that class or not (in this project, the threshold was 0.5).

At the end of a single iteration, the loss is calculated. Loss is a representation of by how far off the prediction is. The weights for each layer are then incrementally updated using some small learning rate in order to try to minimize the loss. This process is repeated through many, many iterations until the final loss is low. This should result in a high precision for our classifier.

See Implementation for more details on the neural network.

Parameters which can be tuned to optimize the classifier:

- Classification threshold (at what probability we choose to assign the class)
- Number of epochs to train
- Batch size (frames to look at per training step)
- Solver type (algorithm for learning)
- Learning rate

Benchmark

This project is very much an exploration of the space – using neural networks for transcription. As such, I don't have a solid threshold other than watching the precision of the classifier continue to increase. The other non-scientific metric is that I periodically use the classifier to turn a wave file into MIDI, re-render it to audio, and gauge my own satisfaction with the results. As someone who has been a musician for more than 17 years, I have a fairly well-trained ear.

If I had to put an actual benchmark, I would like to have a precision of 80%. That would mean a transcriber using this as a tool would be only need to correct 20%. It's completely arbitrary though.

III. Methodology

Data Preprocessing

Music is just sound. Sound is fluctuations in air pressure. The sounds in this project are stored in their base form as waveform audio files (waves). These waves are converted into a lists of features by slicing the wave into short segments then performing the fast fourier transform on them.

Frame:

I'm using this term to denote a short period which is used to generate a single feature set and class set. The time frame used in this project is 40ms as it is shorter than the shortest notes generated.

Wave files:

A wave file a discrete sampling of the amplitude of the audio signal. The wave files used in this project have 16,000 samples per second. Using a time frame of 40ms means each frame represents 640 samples of the audio signal.

Fast Fourier transform (FFT)

The fast Fourier is an algorithm for computing the discrete Fourier transform of a signal. In short, this turns the wave signal into a list of frequency centers and their amplitudes, known as a spectrum. Fourier analysis is used heavily in signal processing and is an excellent use case for feature generation as it gives good separation of lower and higher frequencies (pitches in this case). An FFT transformation for 640 samples of audio results in 321 values.

Since the resulting values of the FFT are complex, they were converted into tuples of floats, then flattened out. This created 2 times the number of features.

All these factors result in 642 features per 40ms "frame" of audio.

Below is a visualization of the spectrum of the first note in "Runs in A" which is a G in the third octave:

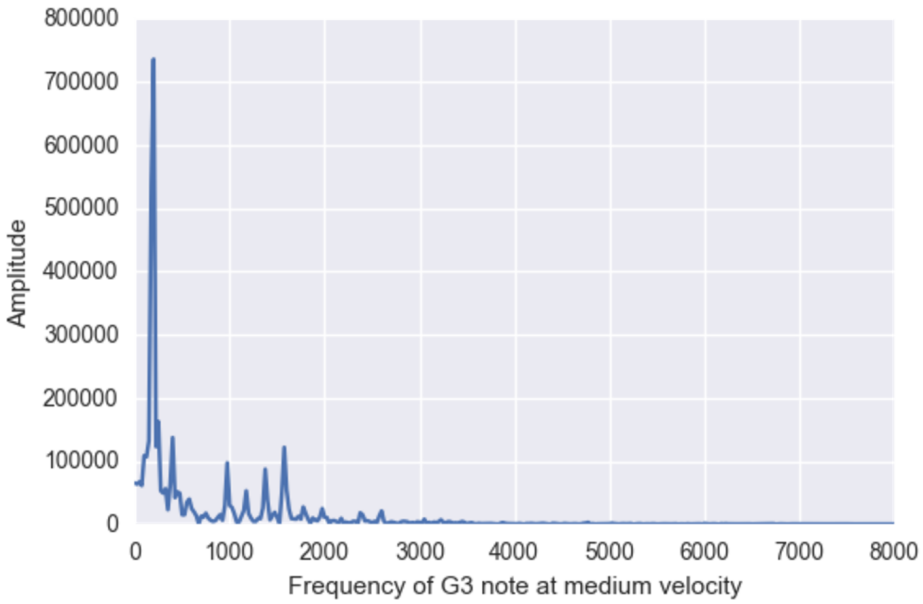

Figure 3: Visualization of the spectrum

MIDI

Midi events for each frame were turned into boolean values over 170 classes. The classes are all pitches from C2 to C9 inclusive and whether the note is turned on and whether the note is turned off.

Implementation

Data processing was implemented in the manor I described above using a custom script. The script uses the numpy and scipy libraries for array and wave form operations. The script usues the python-midi library for reading and creating midi files.

The classifier uses the TensorFlow library for building, running, and visualizing. I also make heavy use of the tensorflow.contrib.learn module for a bit simpler programming interface.

Below is an overly complex diagram of the network.

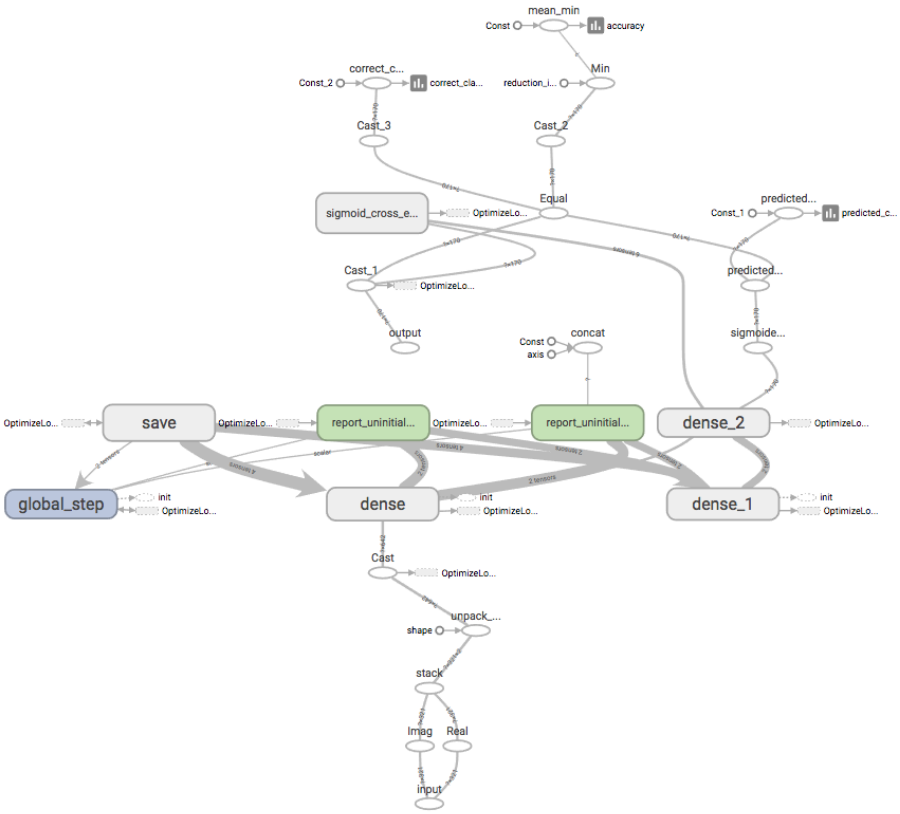

Figure 4: The Network

Input layer

The first step is unpacking the complex number. The pre-processing step stores the features as 128-bit complex numbers, so I unpack them into the 642-feature structure that I specified in the data processing section and cast them to 32-bit floats.

Dense layer

The next (hidden) layer is a 512-neuron, fully connected layer. This operates in the classic nerual-netowrk style of multiplying randomly initialized weights by the input features, adding a bias vector, and passing the result through an activation function.

The activation function in this case is a rectifier (ReLU for "rectified linear unit"). A rectifier is ideally linear (f(x) = x) for all x > 0, but 0 otherwise.

Rectifier approximation:

$$f(x) = ln(1 + e^x)$$

Output layer

At this point the outputs from the dense hidden layer is fed into another dense layer which operates the same as the hidden layer except:

1. It has exactly as many neurons as there are classes to classify (170).
2. It uses a simple linear activation function.

Loss

Since this is a model for a multi-label classifier, softmax is ineffective. So instead I use a sigmoid cross-entropy loss function on the result of the output layer, crossed with the true labels for the feature.

Optimization is handled by `tf.contrib.layers.optimize_loss` which uses stochastic gradient descent and a constant learning rate of 0.001 to minimize loss.

Evaluation

Every 50,000 iterations training is stopped to evaluate against testing data. Precision and recall for the validation set are recorded at this point. At the end of an epoch, the metrics are manually evaluated and the classifier is restarted.

Periodically a wave file is processed and passed through the classifier to visually and audibly inspect the resulting MIDI.

Refinement

The model described above is the final model. I had some frustrating experience while learning to use tensorflow and had some experiments which purely did not work. I think they were due to faulty implementations on my part, which is why I came back to a simpler model. My recurrent neural net wouldn't optimize. Attempting to run a convolutional net on series of frames wouldn't even run.

One experiment in the same line as the final model was to use a deeper network. I opted for 200 x 100 x 100-neuron hidden layers instead of the one 512-neuron layer I have. It had lower precision and recall scores and did not converge on a low loss nearly as fast as the current model.

IV. Results

Model Evaluation and Validation

This model is terrible, but I simply ran out of time to continue improving it and/or learning TensorFlow. The model can understand low notes vs high notes pretty well, but that's about it.

I predicted a live rendition of Beethoven's Moonlight Sonata... Shameful.

Here's the prediction of "Runs in A" on the right channel with the original on the left channel: https://soundcloud.com/bobisme/final-training A visualization of this can be seen in the conclusion section of this paper.

Here are the final graphs of the metrics used.

Justification

This super simple neural net did learn some basic things like how to only predict one or two notes at a time. It learned high notes and low notes.

This problem is not solved by this model.

V. Conclusion

Free-Form Visualization

Below is a screenshot of the virtual instrument used to generate the audio for this project above the MIDI result from using a poor classifier. The MIDI file is very cluttered and noisy. This is the result of near-randomly predicted notes.

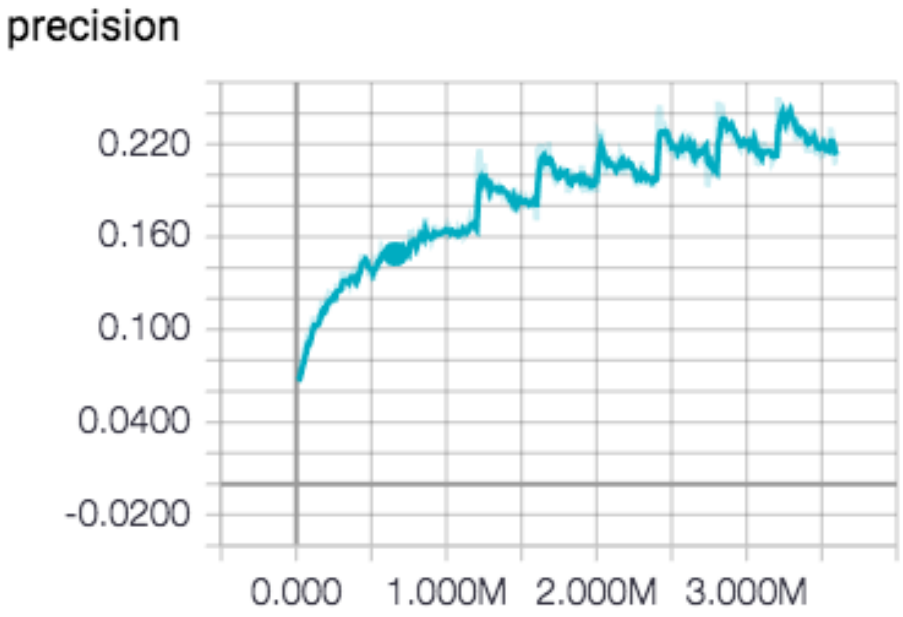

Figure 5: Precision

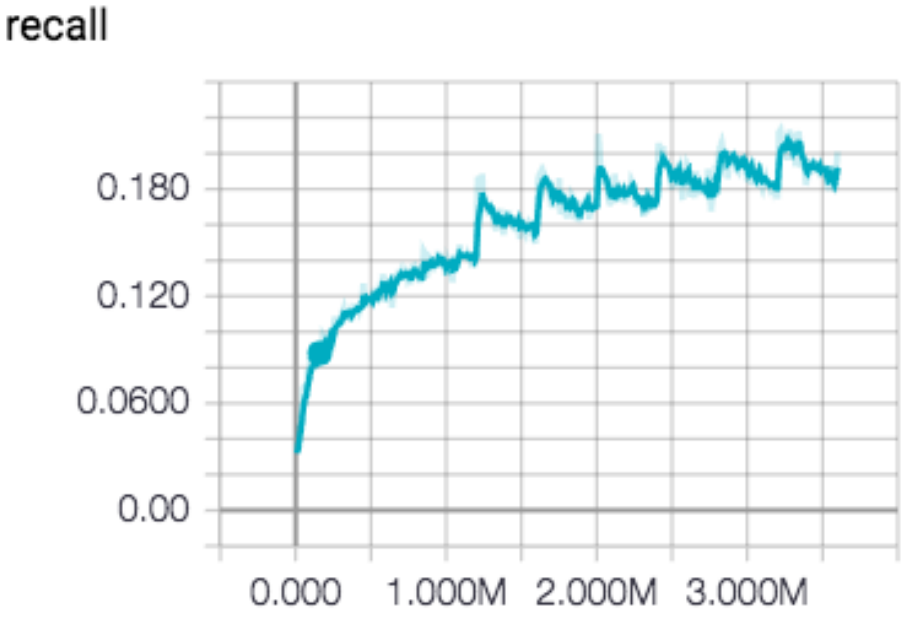

Figure 6: Recall

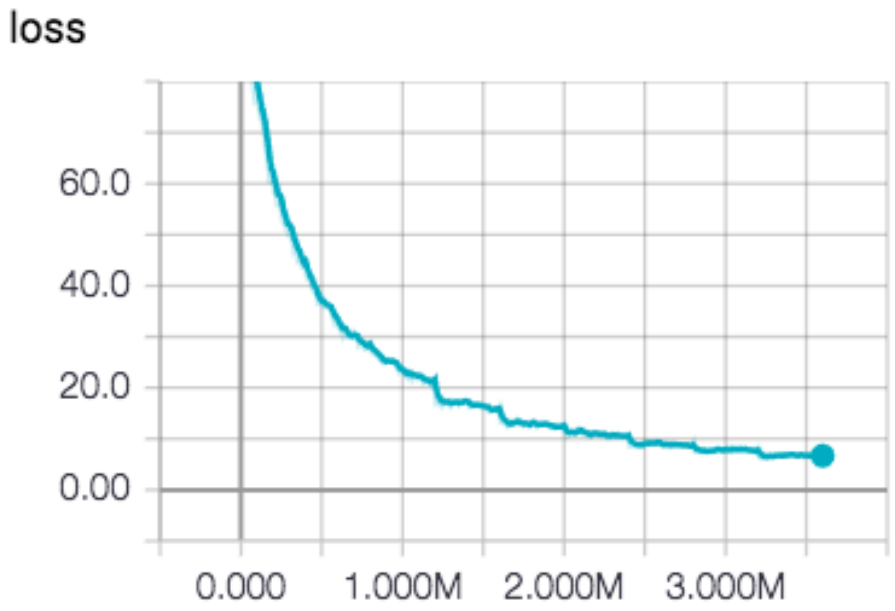

Figure 7: Validation Loss

Figure 8: Really bad model

Below is a screenshot of using the classifier. You can see many incorrectly sustained notes (long horizontal lines). But you can see a noisy shape which mimics the curve of the actual notes:

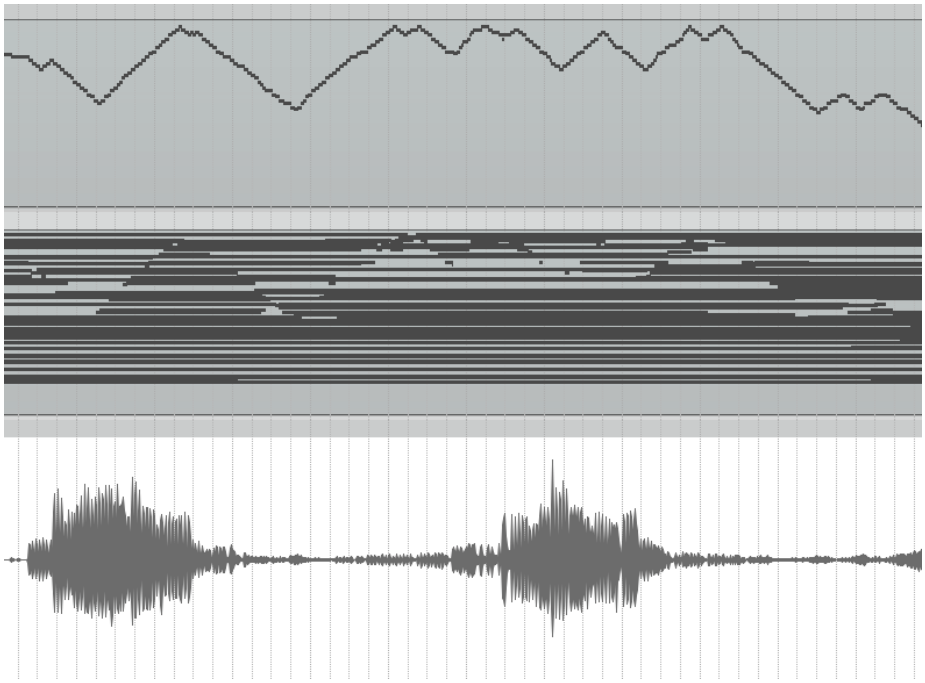

Figure 9: Original (top). Predicted (middle). Original Wave (bottom)

Reflection

The process for addressing this problem involved:

1. Generating and rendering MIDI renditions of musical passages and their WAV-file recording counterparts.
2. Processing that data into 40ms frames of note on/off events to use as labels and a Fourier transform of the sound wave into a frequency-space representation of the data to use as a feature set.
3. Splitting the data into randomized training/testing sets.
4. Running that data through a simple artificial neural network with a single hidden layer and minimizing the loss.
5. Testing the result by using the predictions of the classifier against the true values in the testing set.
6. Manually inspecting the result by rending the predicted MIDI to an audio file.

This project was the first step in a longer term project of training a network to understand features of music generally.

This has been a fantastic project. Music has been a large part of my life for more than half of it. I finally got to build a neural network that dose something I wanted to try, and I finally got to learn TensorFlow. There was a lot of frustration there too due to some under-documented contrib modules.

This final solution is not great, but I think the idea of using neural nets for this task has promise. I think the right kind of neural net would do much better.

Improvement

I think a more complex and more modern network would do much better. A recurrent net or a convolutional net (or a combination of the two) could be astounding. One thing I should have tried was feature scaling. I think that caused some strange errors in other models I abandoned.