Machine Learning Engineer Nanodegree
Capstone Project

Lillian Hartmetz Neff
May 9, 2018

I. Definition

Project Overview

This project falls under the domain of Automatic Music Transcription (AMT) and Music Information Retrieval (MIR). Existing problems in this domain include music recommendations, song identification, genre classification, AI songwriting, and transposition. Google, Spotify, and Ableton are among the companies researching these tasks. For the digital transcription of an audio file to its musical notation, one important stepping stone is Musical Instrument Digital Interface (MIDI). With this project, I propose that machine learning can be utilized to convert polyphonic audio files to MIDI.

Related research:
The projects listed below all utilize machine learning to classify or generate music information.

- https://arxiv.org/pdf/1508.01774.pdf: An end-to-end neural network for polyphonic piano music transcription
- https://musicinformationretrieval.com: Instrument classification, evaluation, and genre recognition
- https://github.com/jsleep/wav2mid: AMT with deep neural networks
- https://magenta.tensorflow.org: Music generation
- https://github.com/googlecreativelab/aiexperiments-ai-duet: Music generation
- https://deepjazz.io/: Music generation with an LSTM
- https://github.com/drscotthawley/audio-classifier-keras-cnn: Audio classification with CNN
- https://arxiv.org/pdf/1703.10847.pdf: A convolutional generative adversarial network for symbolic-domain music generation

Problem Statement

The problem to be solved here is a regression problem, the automatic conversion of an audio file to a MIDI file. I propose a model that can take as input an audio file (mono- or polyphonic) and output a MIDI file with the corresponding notes and note duration. For the purposes of this project, not all information from the audio file will be explicitly retained. Likewise, the model output will be a simplified representation of a MIDI file (which can then be formatted back to standard MIDI). Some

information loss is acceptable, such as dynamics (the volume of the notes being played) and the original instrument(s).

I propose to create a neural network which will learn to identify notes as on or off, and the midi number (or pitch) of the notes which are present. The input features are effectively frequency values (84 of them in this project) over time. The problem is quantifiable by comparing the original MIDI track with the MIDI track generated by the neural network, as a MIDI track consists of numeric values. As such, standard evaluation metrics can be applied and the problem is replicable.

Evaluation Metrics

Some common regression metrics, root mean squared error, mean absolute error, and R2 score, will be used to quantify and compare the performance of the benchmark and the neural network. RMSE effectively puts more emphasis on large errors, letting us gauge the presence of outliers in the error distribution. RMSE was also used as the loss function for this project. The measurement is essentially the same as mean squared error, but RMSE returns a number that is more interpretable than MSE.

$$\mathrm{RMSE} = \sqrt{\sum \frac{(y_{pred} - y_{ref})^2}{N}}$$

MAE measures the absolute errors. In this case, MAE was not appropriate as a loss function, but is a valuable metric as it returns values that are more intuitive than RMSE.

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n} |y_j - \hat{y}_j|$$

R2, the coefficient of determination, effectively gives a measure of how much variability in the prediction is explained by the model.

$$R^2 = 1 - \frac{SS_{Regression}}{SS_{Total}}$$

Sum Squared Regression Error

Sum Squared Total Error

With these metrics, we can measure the difference between the MIDI segments as they should be, versus the MIDI generated by the neural network.

Sources:
https://people.duke.edu/~rnau/compare.htm
https://stats.stackexchange.com/questions/239056/how-to-report-rmse-of-lasso-using-glmnet-in-r
https://stats.stackexchange.com/questions/183265/what-does-negative-r-squared-mean
https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

II. Analysis

Data Exploration

The dataset for this project is the Saarland Music Data. This dataset consists of 50 audio files and their temporally synchronized 50 MIDI files. The dataset also includes the composer and title of each song. Because 50 samples is not enough to train a neural network and because the output space for an entire song would be enormous, I decided to segment the songs. I used librosa's constant-Q transform (CQT) function to create a time-frequency representation of the audio. The CQT function outputs a 2D numpy array with shape (n bins, time), the constant-Q value for each frequency at each time. After slicing each song into 1 second segments, each segment has shape (84, 173).

MIDI, being a protocol designed for digital synthesizers, contains a good deal of information which I decided to set aside (for now) in order to keep the problem as simple as possible. This extraneous information includes pitch bend, velocity, control changes, MIDI file type, and metadata tracks. One general complication regarding audio/MIDI datasets is that MIDI is not perfectly standardized. For example, “A quirk of the MIDI standard (and one that some older Yamaha models had trouble dealing with), is that the standard allows notes to be released by sending a note on message with a velocity value of 0, instead of using a note off message.” Accounting for such inconsistencies makes MIDI parsing complicated. Regarding time in this dataset, the creators acknowledged that “there may be a global offset introduced by various reasons (decoding, MIDI parser, etc.)”

	Min	**Max**	**Median**	**Mean**
CQT Input	-14.42	14.50	0	-3.05e-6
MIDI Output	0	1	0	0.0252

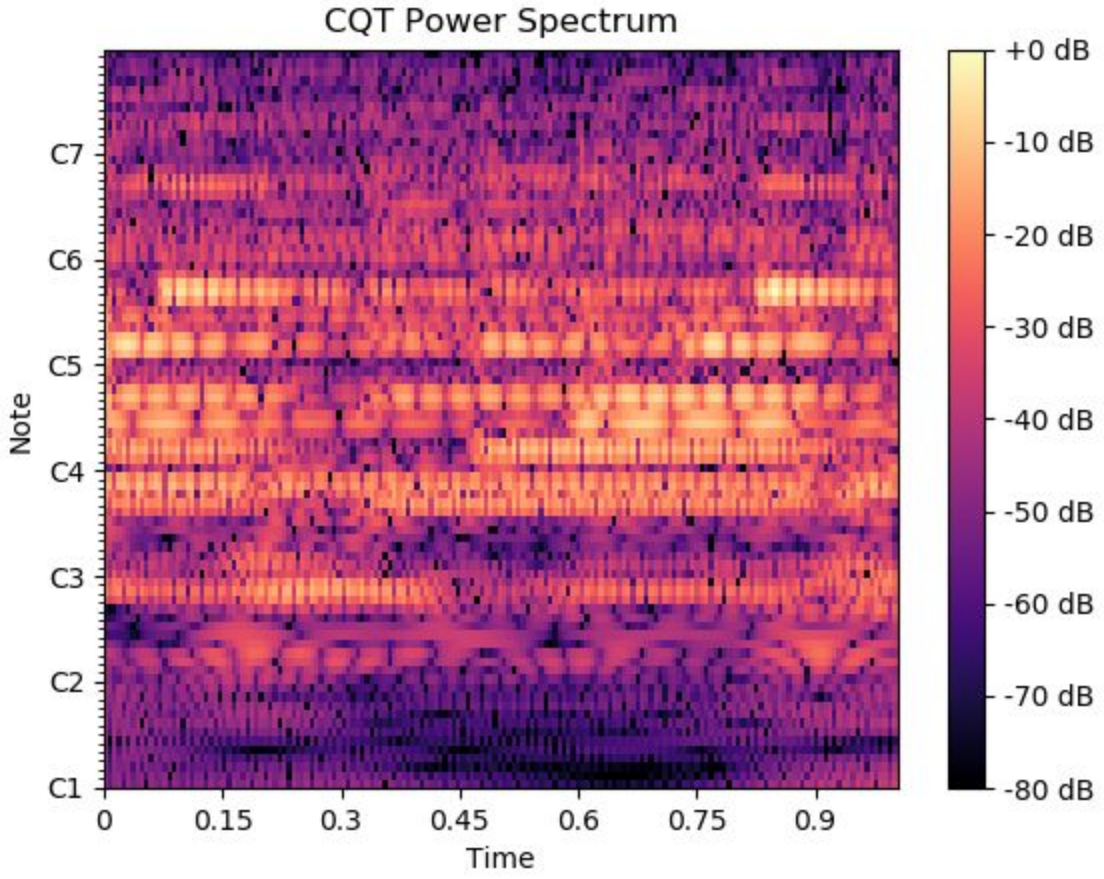
Ultimately, each input is a 2D numpy array of shape (84, 173). Each output is a 2D numpy array of shape (87, 6). (See Data Preprocessing section below for details)

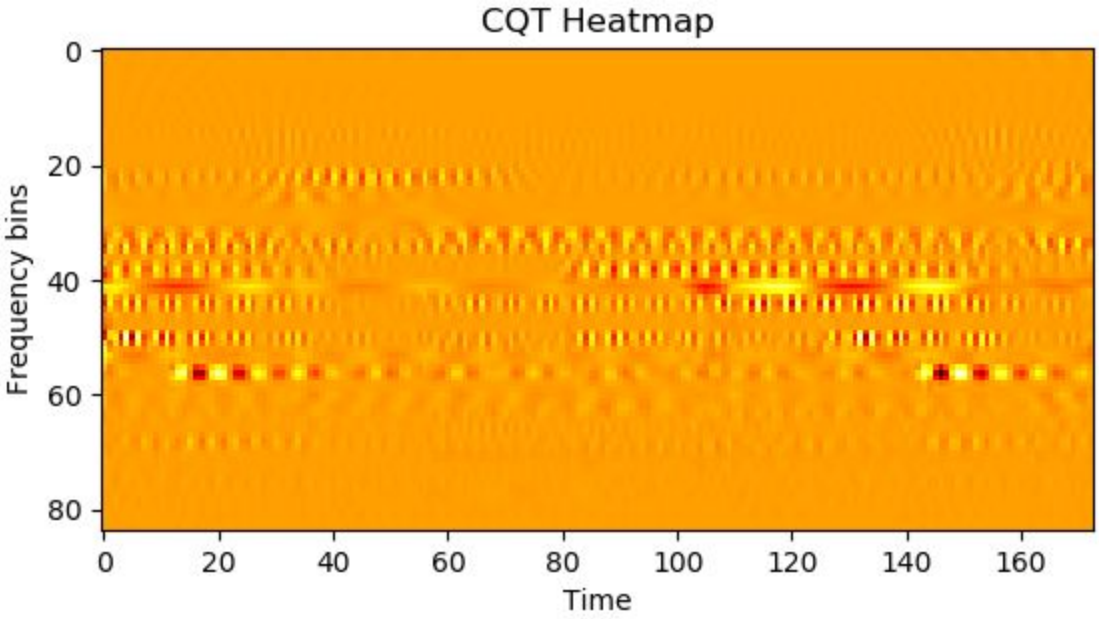
Saarland Music Data: https://www.audiolabs-erlangen.de/resources/MIR/SMD/midi Meinard Müller, Verena Konz, Wolfgang Bogler, Vlora Arifi-Müller: Saarland Music Data (SMD). In Late-Breaking and Demo Session of the 12th International Conference on Music Information Retrieval (ISMIR), 2011.

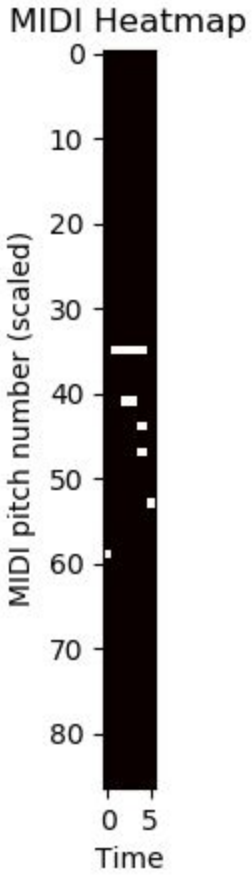
Electronic Music Wiki: http://electronicmusic.wikia.com/wiki/Velocity

Exploratory Visualization

This project includes a file named exploratory_visualization, which includes the code to generate the plots below. Below is an example of one audio segment CQT and its corresponding midi segment. A CQT transforms the audio data series to the frequency domain. The output of the transform is effectively amplitude/phase against frequency, a process which mirrors the human auditory system. Yellow pixels indicate the highest amplitude notes in this segment. By comparing the CQT power spectrum against the MIDI, we can see that we're dealing with very noisy data. I've included a heatmap of the CQT segment as well for ease of comparison.





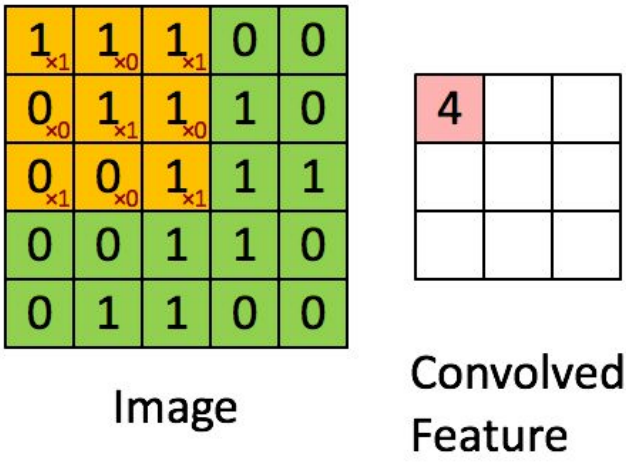


Algorithms and Techniques

Because of the way we can segment and shape audio data and because neural networks are well-suited for handling data of multiple dimensions, my most promising solution is a Convolutional Neural Network. CNNs are well-suited for problems for which the spatial information of the input (in

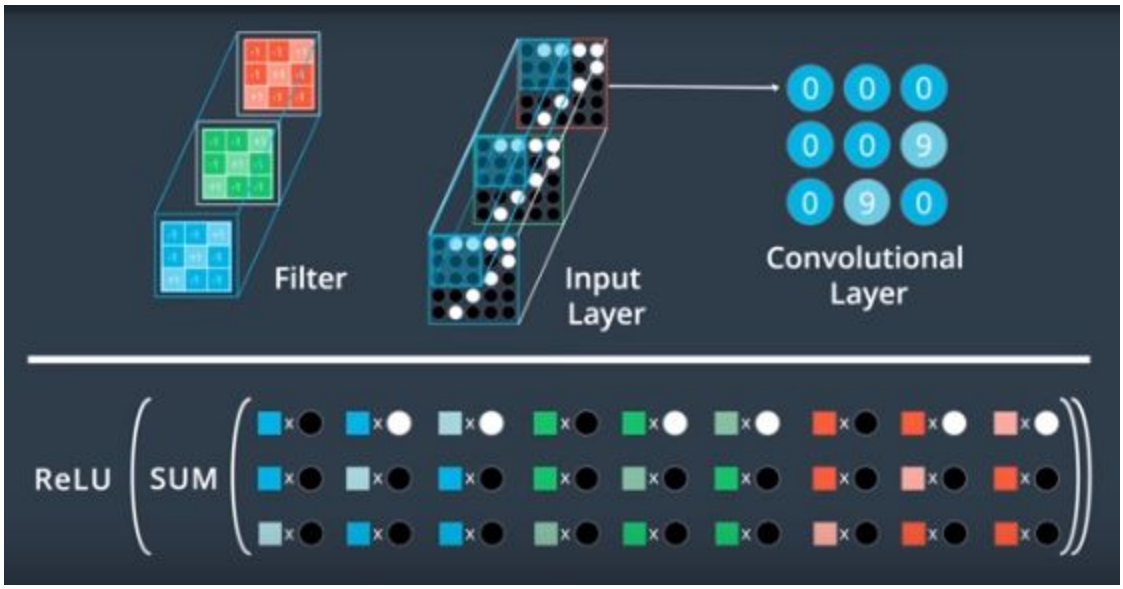
this case, the presence of notes over time) needs to be preserved. In previous automatic music transcription research, such as An End-to-End Neural Network for Polyphonic Piano Music Transcription, CNNs have proven to be relatively effective at determining pitches present within a given time frame. I began by reshaping my data for the CNN. I added a dimension (depth of 1) to the CQT segments. I also flattened the MIDI segments.

In a CNN, at each layer, a kernel (a matrix of weights) slides over the input matrix and produces a matrix (convolved feature) that gets passed on to the next layer.



We can use multiple filters to populate an additional collection of nodes, deepening the output. I used 10 Conv2D layers, each with 2 to 6 filters. So as to reduce dimensionality in the time dimension, but not the note dimension, I used a stride of 1 for half of the layers.

With each kernel, we use an activation function to define the value of the corresponding node in the output space (after the inputs and weights are multiplied and the bias is added).



Each convolutional layer in this project uses ReLU as its activation function. I flattened the output of the last convolutional layer in order to feed it to the final layer: a Dense layer. The Dense layer connects each node in the previous layer's output to 522 nodes (the same size as the MIDI output). In this way, every neuron of the last convolutional output is connected to every neuron in the Dense layer, narrowing the convolutional output to fit the shape of the desired MIDI output. An activation function (as it does with the convolutional layers) defines the output of each node in a dense layer. I chose sigmoid as the activation function for the Dense layer as it will return values between 0 and 1.

The optimizer iteratively updates the weights and biases, attempting to minimize the RMSE. For this project I chose the Adam optimizer as it is computationally efficient and requires little memory.

An End-to-End Neural Network for Polyphonic Piano Music Transcription: https://arxiv.org/abs/1508.01774

CNN Image:
https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/19273/1525533295/Convolution_schematic.gif

Udacity CNN Description and Visualization:
https://youtu.be/RnM1D-XI--8

Adam Optimizer:
https://arxiv.org/pdf/1412.6980.pdf

Benchmark Model

My benchmark model is chance: Given an audio file, the benchmark will return a random midi file from the dataset. The same evaluation metrics were applied to the benchmark and the final model.

	RMSE	**MAE**	**R2**
Benchmark	0.2189	0.0479	-0.9829

III. Methodology

Data Preprocessing

I reduced the information in the MIDI file by first looking at the MIDI track in each file with the most messages (the track containing note on/off messages, not metadata) and using the most relevant information in each message. That information is: whether a note is on or off, the pitch of each note turning on or off (represented as an integer ranging from 0 to 127), and the MIDI time (delta ticks: the number of MIDI ticks since the last message). Additionally, I found that some songs had

redundant on and/or off messages, which I had to remove in order to encode the MIDI segments. I encoded the MIDI data in such a way that a value of 1 indicates a note is on, and 0 indicates that a note is off.

Regarding time, I had initially attempted to segment the audio and MIDI based on beats using librosa's beat detector and converting MIDI ticks to beats. After several unsuccessful attempts, I decided to segment based on seconds. I segmented the 50 classical piano pieces into 33,898 1-second segments. Then I performed a CQT on each segment. This encoding based on seconds discretizes time for each MIDI segment.

To compensate for the offset, I segmented the data in such a way that for every MIDI segment, the corresponding notes are assuredly present in its corresponding audio segment. I accomplished this by segmenting the audio into wider (longer) segments than the MIDI. For example, a MIDI segment from timestamps 0.5 seconds to 1 second is represented by a audio segment from timestamps 0.25 seconds to 1.25 seconds. For the first and last MIDI segments, the audio is padded with silence at the beginning or end of the audio segment, respectively. For example, a MIDI segment from timestamps 0 seconds to 0.5 seconds is represented by an audio segment with 0.25 seconds of silence concatenated with audio from timestamps 0 to 0.75 seconds. This segmentation resulted in a dataset of 33,898 paired audio and MIDI segments.

For the MIDI messages, the pitch values get encoded. A pitch not being played will be represented with a 0. A pitch being played will be represented with a 1. The output size will be as follows:

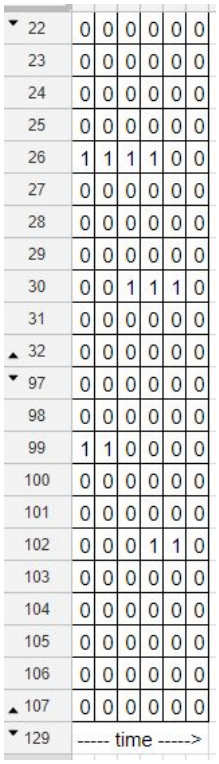
possible pitches * number of possible discrete time values

MIDI notation specifies 128 possible pitches. This dataset contains only songs with pitches between 21 and 107 (inclusively), so the output space is shaped for those 87 possible pitches. The number of possible discrete time values chosen for segments of length 0.5 seconds is 6. Therefore each discrete time value is 0.083 seconds.

This corresponds to the following formula in terms of musical notation:

max_bps * (1 / (beats_per_measure * shortest_note)) * midi_segment_length_in_seconds

Max_bps refers to the maximum bps the output is intended to handle. Shortest_note refers to the briefest note the output is intended to handle (eg, ⅛ for an eighth note). For this project, I've set max_bps to 3 (180 bpm), beats_per_measure to 4, and the shortest_note to a sixteenth note, making the number of possible discrete time values 6. Below is a visual representation of the output space.



In short, each input is a 2D numpy array of shape (84, 173). Each output is a 2D numpy array of shape (87, 6).

Implementation

In this project, I'm using Keras with a Tensorflow backend.

Libraries used:

- collections
- keras
- librosa
- math
- matplotlib
- mido
- ntpath
- numpy
- os
- pickle
- random
- scikit-learn
- shutil
- tensorflow
- time

For reproducibility I set the numpy and random seeds to 21 throughout the project. Preprocessing the audio and MIDI takes hours to run, so I used pickle to save the data to disk and save time. To build my model, I began by shuffling and splitting my data. I set aside 20% for testing, then 20% of the remaining samples for validation.

I used an 11-layer CNN: 10 convolutional layers with ReLU as the activation function and 1 Dense layer with sigmoid as the activation function. I used RMSE for the loss function. Metrics include RMSE, MAE, and R2.

Notable complications include NaN loss and unwanted bottlenecking. When running my first model architectures, the loss would turn to NaN after only a few epochs. This NaN loss issue was corrected by decreasing the learning rate from 0.001 to 0.0001.

To eliminate the bottlenecking issue I had in earlier architectures, I realized I needed to carefully consider the kernel and padding specifications. Regarding kernel specifications, I discovered that if the kernel height for a layer was greater than 1, the stride must be 1 in order to avoid throwing out the information in every other frequency bin. Similarly, if valid padding was used in certain layers, I was throwing out information. Below is the model summary of the final model. The stride increases from 1 to (1, 2) after the fourth convolutional layer. In the model summary below, we can see that once the stride is increased after the fourth layer, the output quickly loses size in the time dimension.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 84, 173, 2)	6
conv2d_2 (Conv2D)	(None, 84, 173, 2)	30
conv2d_3 (Conv2D)	(None, 84, 173, 3)	15
conv2d_4 (Conv2D)	(None, 84, 173, 3)	66
conv2d_5 (Conv2D)	(None, 84, 87, 4)	28
conv2d_6 (Conv2D)	(None, 84, 44, 4)	36
conv2d_7 (Conv2D)	(None, 84, 22, 5)	45
conv2d_8 (Conv2D)	(None, 84, 11, 5)	55
conv2d_9 (Conv2D)	(None, 84, 6, 5)	55
conv2d_10 (Conv2D)	(None, 84, 6, 6)	66
flatten_1 (Flatten)	(None, 3024)	0
dense_1 (Dense)	(None, 522)	1579050

I used a handy built-in Keras callback called ModelCheckpoint to save the “best so far” models based on the validation loss.

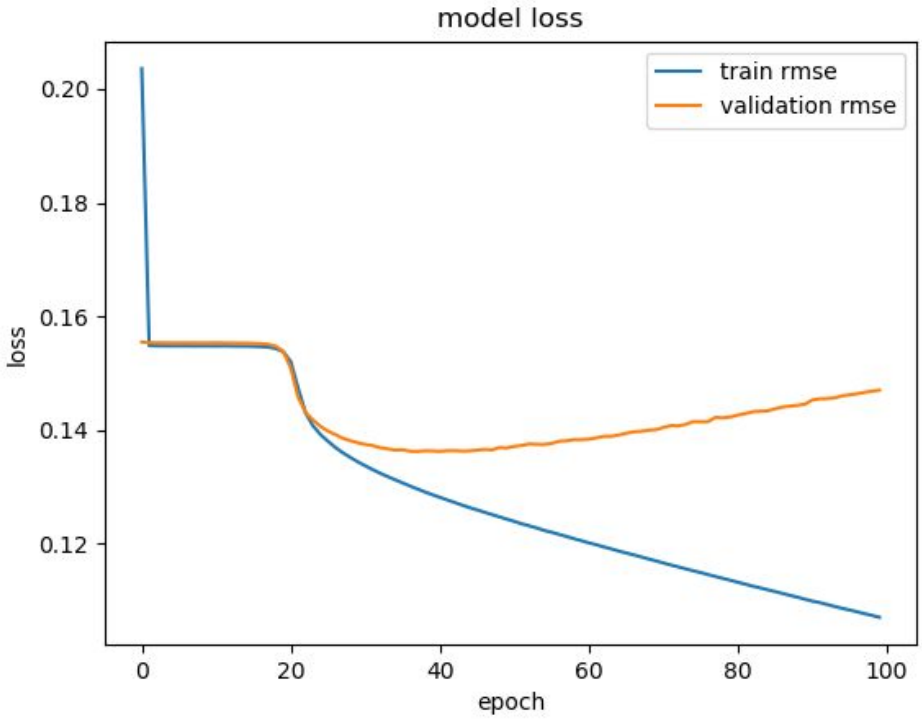
I found recording and validly comparing model architectures to be challenging because their are an infinite number of parameter combinations and because each run is fairly time consuming (ranging from 35 minutes to 3 hours on a GTX 1080 Ti). The table in the section below shows the last six architectures. Ultimately I had the most success using a combination of the ModelCheckpoint and a Google spreadsheet to track parameter changes and their effect on performance.

Refinement

As I developed this model, I used only the validation samples to inform my parameter tuning, and did not evaluate model performance on the test data until the final run of my final model.

Model architecture	**Batch size**	**Validation RMSE**	**Run time per epoch**
6 Conv layers with kernel sizes (1, n)	32	0.1381	35 seconds
8 Conv layers with kernel sizes (1, 2)	32	0.1369	92 seconds
10 Conv layers kernel sizes (1, 2) and (7,1)	32	0.1363	108 seconds
10 Conv layers kernel sizes (1, 2) and (7,1)	1000	0.1555	45 seconds
10 Conv layers kernel sizes (1, 2) and (7,1)	500	0.1555	46 seconds
10 Conv layers kernel sizes (1, 2) and (7,1)	250	0.1546	69 seconds

Loss plot for the final model:



After running the model with validation RMSE 0.1363 (ultimately the final model), I experimented with batch size. Up to this point, I had used a batch size of 32. I ran the same model with batch sizes of 1000, 500, and 250, expecting that would help keep the model from overfitting. Interestingly, the loss for all runs with increased batch sizes was higher.

IV. Results

Model Evaluation and Validation

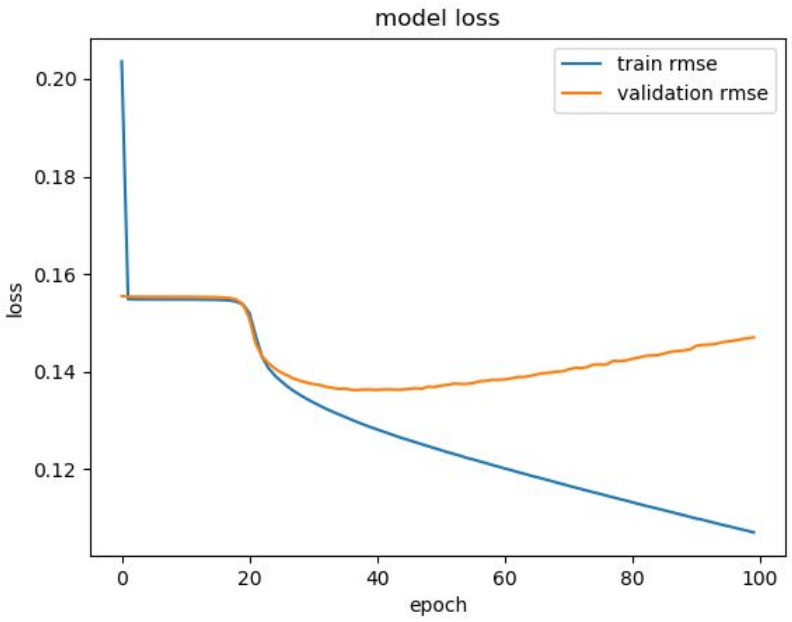
During development, I used a validation set to inform my decisions about how to improve the model. Below are the parameter details of the final model.

Layer	Filters	Kernel size	Stride	Activation function
Conv2D	2	(1, 2)	1	ReLU
Conv2D	2	(7, 1)	1	ReLU
Conv2D	3	(1, 2)	1	ReLU
Conv2D	3	(1, 2)	1	ReLU

Conv2D	4	(1, 2)	(1, 2)	ReLU
Conv2D	4	(1, 2)	(1, 2)	ReLU
Conv2D	5	(1, 2)	(1, 2)	ReLU
Conv2D	5	(1, 2)	(1, 2)	ReLU
Conv2D	5	(1, 2)	(1, 2)	ReLU
Conv2D	6	(1, 2)	1	ReLU
Dense	NA	NA	NA	sigmoid

In order to validate the robustness of the model, I used K-Fold Cross Validation to explore different folds scores. Training the model on varying subsets revealed that the results of the final model (0.1363) were better than average (0.1416).

	Random Seed	**RMSE**	**MAE**	**R2**
Benchmark	21	0.2189	0.0479	-0.9829
Final Model Validation	21	0.1363	0.0175	0.6
Final Model Test	21	**0.1347**	**0.0361**	**0.2531**
K-Fold 1	None	0.1445	0.0291	0.1125
K-Fold 2	None	0.1395	0.0269	0.1638
K-Fold 3	None	0.1419	0.0270	0.1388
K-Fold 4	None	0.1423	0.0268	0.2196
K-Fold 5	None	0.1398	0.0275	0.1681
K-Fold Mean Test Result	None	**0.1416**	**0.0275**	**0.1601**



Looking again at the loss plot for the final model, we can see that the model starts to overfit just before the 40th epoch: the training error continues to improve, while the validation error increases. The model's RMSE on the test data was 0.1347, slightly better than the best validation RMSE.

It should be noted that all songs in this dataset are classical and played on piano. It is difficult to guess how the results might vary with audio input of another genre, another instrument, multiple different instruments playing, vocals, percussion, etc.

Justification

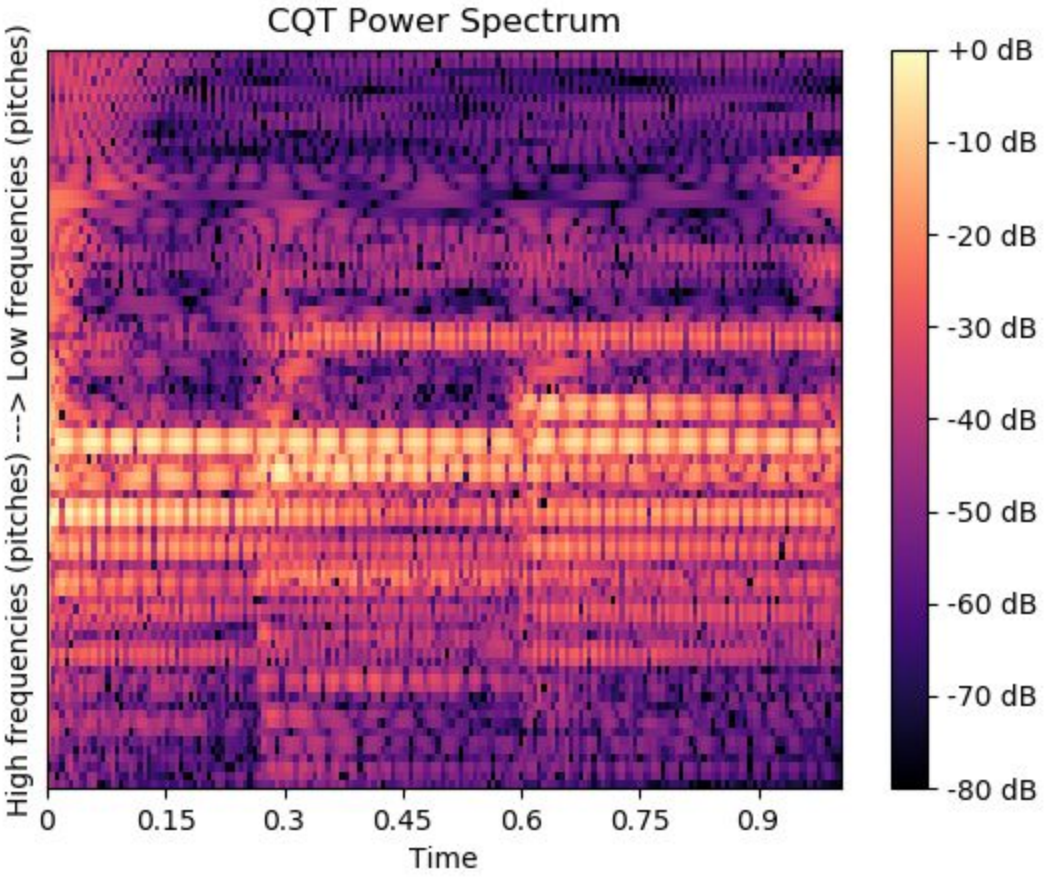
The final solution is certainly stronger than the benchmark. Though the final model does not entirely solve the problem at hand, the results are pretty exciting. The model can predict most MIDI messages correctly. For many digital audio workstations (DAWs), it's much easier to delete or change the duration of a MIDI message than it is to create a message. My code, as is, rounds the predictions to the nearest integer. If I were to give someone this model to use "out of the box", the user could change the rounding threshold to suit their needs, for example, rounding every float greater than 0.35 to 1. By setting a lower threshold, they could conceivably generate a MIDI message for all notes the models "suspects" are on, and simply select and delete the false positives.

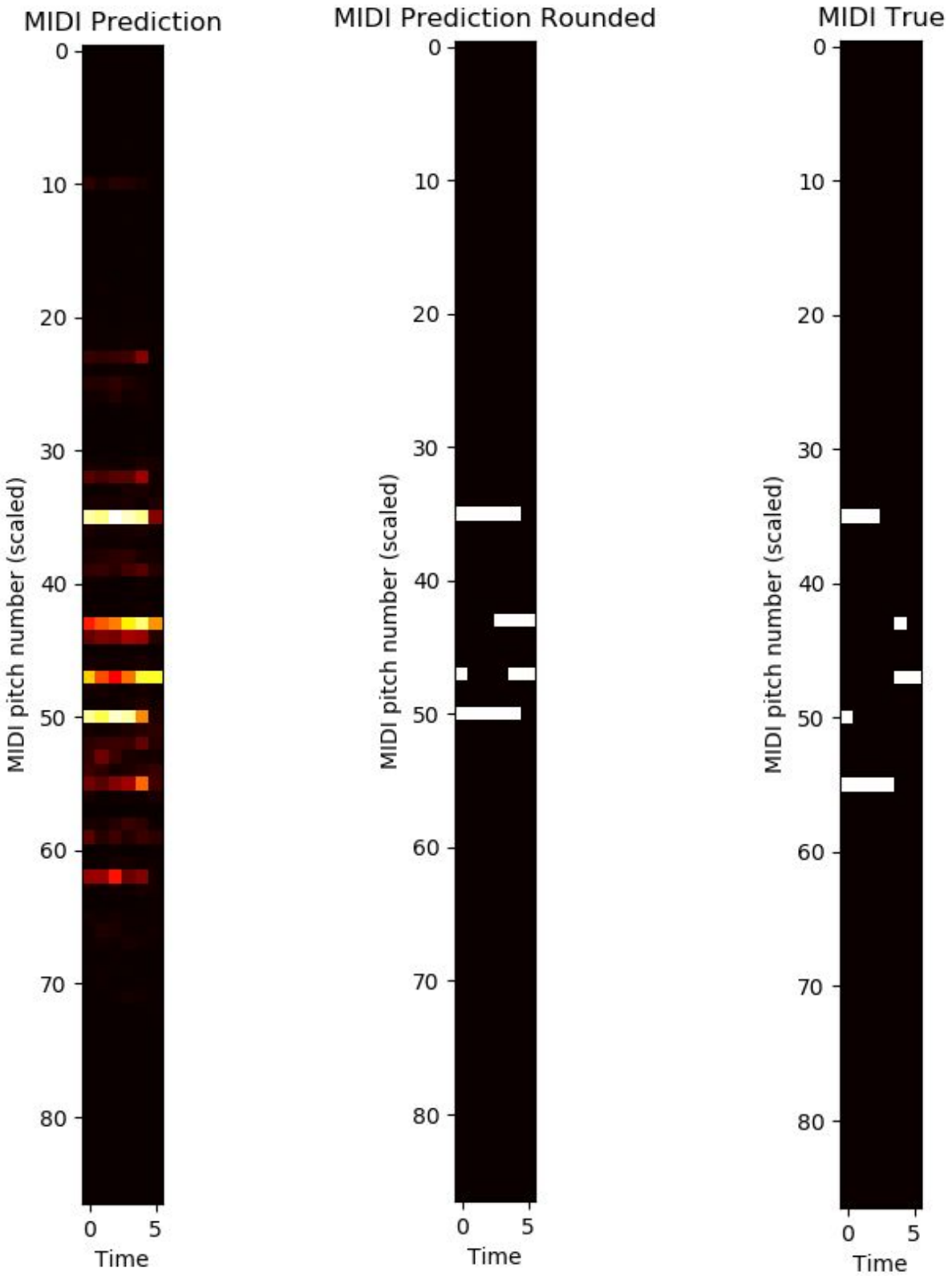
	RMSE	**MAE**	**R2**
Benchmark	0.2189	0.0479	-0.9829
Final Model Validation	0.1363	0.0175	0.6
Final Model Test	0.1347	0.0361	0.2531

V. Conclusion

Free-form Visualization

The CQT power spectrum shows some elements of how challenging audio conversion is. Below is one test sample of the model's output. The RMSE for this prediction is 0.1372 (slightly worse than average for the model). (The CQT below was flipped upside-down so that, as with the MIDI y-axis, the low pitches are represented at the top and the high pitches are represented at the bottom of the image.)





When comparing the CQT with the true MIDI, we can see that we're interpreting very noisy data. This is largely due to the presence of overtones. In the MIDI true heatmap, we can see five fundamental pitches. The CQT reflects that, from the audio, the neural network must interpret harmonic overtones as silence and not assume that all higher frequencies are overtones.

There are a number of hyperparameters that especially helped the performance of the final model. When choosing kernel and stride sizes, I had to take care to not reduce the frequency dimension and or reduce the time dimension below the chosen number of discrete time values in each MIDI segment (6). As with most neural networks, increasing the number of layers improved performance. Additionally using kernel sizes of (1, n) or (n, 1) improved performance. I strongly suspect that the kernel sizes of (n, 1) allowed the model to better learn which notes were fundamental notes and which were overtones.

Reflection

To reiterate the problem at hand, what we want to do is convert audio to MIDI automatically. In my opinion, the most interesting and difficult challenge in this project is preprocessing the data. As I mentioned, I initially wanted the element of time to be based on musical notation (ie beats), but ultimately I decided finding and segmenting based on beats was out of scope for this project. Even after segmenting based on seconds, it was very difficult to gauge whether or not my audio/MIDI segments were aligned. Of course it is always better to check code with a unit test, but auditory tests were sometimes faster, equally informative, and very interesting.

As a more general lesson, I learned a lot about the benefit of focusing on a minimum viable product and failing fast. This project was a reach for me and, in hindsight, I wish that I had let go of my more difficult sub-goals early on. For example, I spent a lot of time segmenting based on beats and trying to get the alignment perfect so that I wouldn't have to pad the audio. If I wrote more unit tests throughout, I would have saved time and also kept my code more organized the project.

Improvement

There's lots of opportunity in this project for me to return to the preprocessing stage and (hopefully) improve performance. In retrospect, it may have been more efficient to perform the CQT in such a way that n_bins is equal to the number of possible MIDI pitches in the output. I originally performed the CQT on my data with the default n_bins value of 84. I suspect that the reason for some of my error is that my model is having to map 84 bins to 87 possible MIDI pitch values. Additionally, I'd like to see how feature standardization affects the model's training and performance. I understand that feature standardization is usually recommended. I don't believe it's necessary for this data, but it's certainly worth exploring.

Another change I want to explore is to use fewer discrete time values in the output space. A smaller output space generally makes predictions more computationally feasible. This would also allow for more layer options, since I wouldn't have to worry about bottlenecking as much as I did with the current input and output shapes. My understanding is that I must keep the dataset exactly as I described it in my proposal, so I plan to explore these possibilities after I complete this course.