Compose Chopin

Learn Chopin's style of composition through CNN, AutoEncoder, GANs and Pix2Pix.

Bethold Owusu, Hannah Do, Ethan Cobb

CSCI1910 / Spring 2021

Abstract

Our project aims to learn Chopin's style of composition through various DNN models such as CNN, LSTM, AutoEncoder, and GANs. Training a machine how to compose has been in demand for awhile, and many state of the art models such as Amper Music, AIVA, Jukedeck, and others have been developed with AI composers. However, no commercial AI system has been good enough by itself, and many composers use it as a mere reference. Therefore, we have taken a different approach in converting the wav files into spectrograms, and predicting the pieces as an image analysis process. In other words, our project focuses on predicting the next sequence of a song given a previous sequence of a song using spectrogram pixel values. We have developed different DNN models to produce machine-composed segments that resemble target music pieces with high similarity scores. Converting the midi and wav files into spectrograms unavoidably added some noises to the song, however complex CNN model with transpose layers and AutoEncoder model resulted in higher prediction accuracy than the Base models with low mse and mae scores when compared with the actual sequence. In addition, VAE, Transformer, DCGAN, and PIX2PIX models were tested and the last model, PIX2PIX resulted in highly accurate conversion in recreating the sequence of the songs. Having such a result, we hope to expand this project not only in Chopin style, but to different composing styles in order to allow composing songs as easy as clicking a button.

Introduction

Our research question was how to train the song sequences from a dataset to predict a new sequence, enabling a trained machine to compose songs by itself. To begin with, we selected a classical music midi file dataset from Kaggle. Compared to other datasets that contain different instrumentals and information about the file itself, our goal was to train the model with a clean midi file dataset that only contains piano tunes in refined forms. As we wanted to predict a composition style of a single composer, the composer with the most songs in the dataset - Chopin was selected. In researching different types of ways to pre-process the midi files, we found many state of the art models using spectrogram conversion for data analysis, using image analysis method to extract a new spectrogram to predict following sequences. As a spectrogram shows the power of various frequencies in the song over time in an image, we decided to move on with converting the midi files to spectrograms. The specific process included processing the midi into wav files first, then converting the wav files into spectrograms through Librosa library. And with the spectrogram arrays stacked together to form 3D numpy arrays, different DNN models such as CNN, Transformer, AutoEncoders, DCGAN, Pix2Pix were trained to predict the final sequence array. The procedure and detailed results are explained in the following sections.

Background

There has been significant progress in the generation of music using deep neural network architectures. GAN's have been shown to be successful at composing in the style of particular artists or genres, and LSTM-RNN networks have also shown to be effective at generating melodies, albeit their extremely long training times. While the majority of these network architectures have focused on producing music in the style of whatever is inputted into the model, not as much work has been done for the specific task of completing pieces fed as input to the model. This kind of problem - feeding half of a particular piece to a DNN and having it finish the rest of the piece - lends itself quite nicely to classical music applications. There have been many pieces composed throughout the Classical, Baroque and Romantic periods that were left unfinished - Schubert's 9th symphony is perhaps one of the most famous of these instances. This kind of DNN application could prove to be incredibly interesting to not only AI Audio Engineers but to those on the more qualitative end - musicologists, music historians, ethnomusicologists, etc. We could have chosen any number of pieces to train on, but we chose one composer - Chopin - to simplify the model and our understanding of our outputs. By only considering one composer, it is easier to develop an "ear" for what should sound like similar output. As stated above, although the majority of literature in AI music has focused on generative models to compose music in a particular style or genre, none has focused in particular on the task of completing a specific piece of music. In terms of the overall model and training architectures, there is not necessarily any novelty compared to the literature, however the problem formulation itself is unique. This unique approach lends itself to calculating various loss and accuracy scores since we know exactly what the results should be (for the other half).

Data

1. Dataset selection and File Format Conversion

Upon researching different types of datasets, we have decided to select midi files as our source as the format conserves piano tunes in a finer form compared to other instrumentals or audio files, which may add different types of noise into our data. And we decided to choose a dataset from a reliable source - Kaggle, and found a dataset called 'Classical Music MIDI' with a total 7 MB of data with over thousands of downloads history (https://www.kaggle.com/soumikrakshit/classical-music-midi (https://www.kaggle.com/soumikrakshit/classical-music-midi)). Considering we had to train the models with songs from one composer, we selected a composer with the most number of songs, which was Chopin. We initially started with the 48 midi files of Chopin's different piano pieces, however converted them to wav files in order to pre-process them into spectogram arrays and segmented the wav files into 15 seconds time intervals with 5 seconds of overlap, resulting in over 2700 wav files.

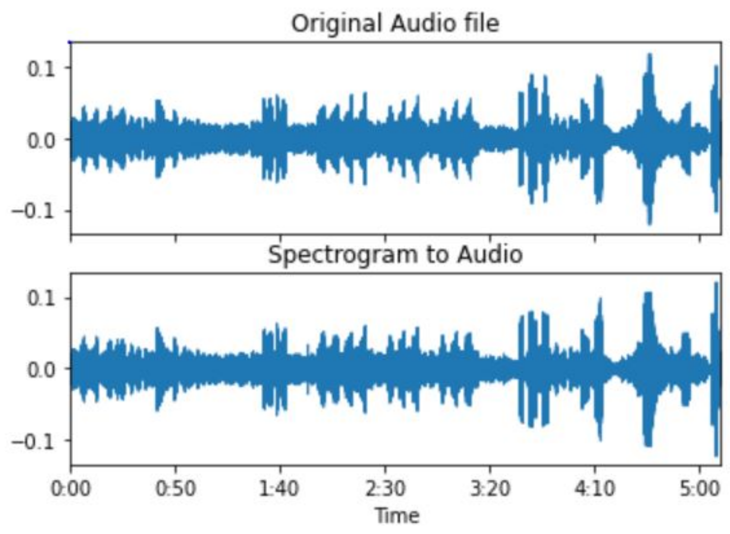
2. Dividing the data into Train-Test, Input-Output

We splitted the total dataset into train, test and input, output data before the conversion of wav files into arrays since splitting them after converting them to arrays resulted in loss in sound quality. After we have splitted the data into four parts, with the train and test data and input and output sets, we converted the wav files into spectogram arrays using Fluidsynth library. Since the spectogram in general shows the power of various frequencies in the song over time in an image, it allowed us to train the dataset with machine learning models such as CNN, Autoencoder, DCGAN, and PIX2PIX.

3. Cleaning and Pre-processing the Data

Before converting the wav files into spectograms, any files that were cut off under 15 seconds - the leftover segments were removed to reduce any outliers. And using Librosa, the waveplots and spectograms were visualized to make sure the spectograms were converted adequately. The segmenting process used AudioSegment function from Pydub library used to manipulate audio with high level interface in python, preserving the audio quality while dividing them into smaller pieces. In addition, with the Librosa library, the wav files were converted into a spectogram with the method of squaring the magnitude of the short term Fourier transform (STFT), then compressed to a mel-spectograms. This made sure all the data input and outputs were in range of appropriate spectogram scale. Each spectogram arrays were stacked together to In addition, Bethold converted the spectograms back to wav files to ensure the rhythm and tempo is preserved. Below shows the original audio and that of the spectrogram.

Figure 1. Spectograms converted back to Wav Files



4. Data Reshaping

Based on different models, the spectrogram arrays had to be reshaped in order to fit the models. For the baseline models, such as Random Forest and Linear Regression, the input and output shapes were flattened to 1D arrays with reshape(-1,1) function in order to be trained to evaluate mae and mse scores. And as the conversion between melspectogram and wav file required a 2D array, the dimension of the stacked 3D model was expanded to 4D array for regression prediction with np.expand_dims(data, -1). The final shape of train and test data for CNN models were (1275, 128, 646, 1) and (158, 128, 646, 1).

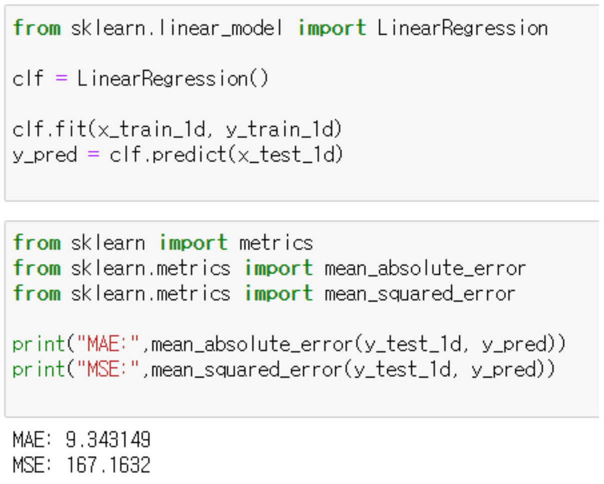
For Autoencoder, dimensions for the input were (1575, 256, 64, 1) and the same for the decoder, while the dimension of the latent space was 2. And for Transformer, GANs, and Pix2Pix models, an additional procedure of converting the grayscale spectogram to color spectogram was used to conserve more data, resulting in train and test data shapes of (2, 256, 300, 128, 1) and (2, 20, 300, 128, 1).

Methods

1. Baseline Models: Linear Regression and Random Forest

> For the baseline models, the input and output shapes were flattened to 1D arrays with reshape(-1,1) function in order to be trained to evaluate mae and mse scores. Simple logistic regression was performed without any regularization and following is the result, the mae score of 9.34 and mse score of 167.16.

Figure 2. Baseline Model 1 - Linear Regression and its mae, mse results



```
from sklearn.linear_model import LinearRegression

clf = LinearRegression()

clf.fit(x_train_1d, y_train_1d)
y_pred = clf.predict(x_test_1d)
```

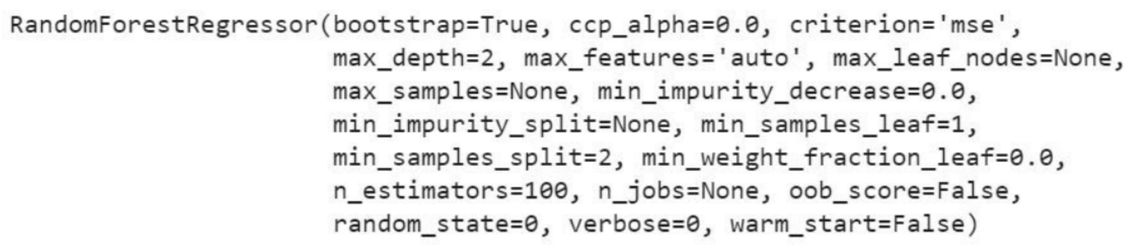
```
from sklearn import metrics
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

print("MAE:",mean_absolute_error(y_test_1d, y_pred))
print("MSE:",mean_squared_error(y_test_1d, y_pred))
```

```
MAE: 9.343149
MSE: 167.1632
```

> Meanwhile the Random Forest model was set up with different parameters after cross-validation as the following :

Figure 3. Baseline Model 2 - Random Forest model with different parameters



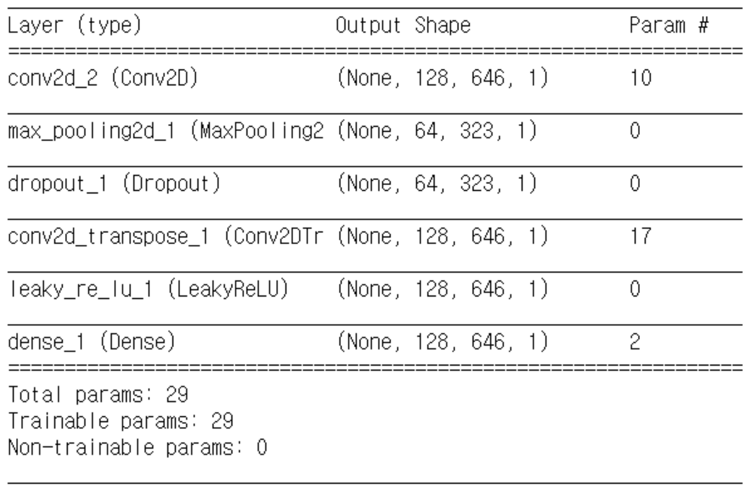
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=2, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=0, verbose=0, warm_start=False)
```

After training with the normalized train and target dataset, the Random Forest model was evaluated on the test dataset and as a result, an R square of 0.4 was obtained. Another metric of R square was utilized here as the baseline model is performing a regression prediction based on the train and its target dataset.

2. CNN : Hannah

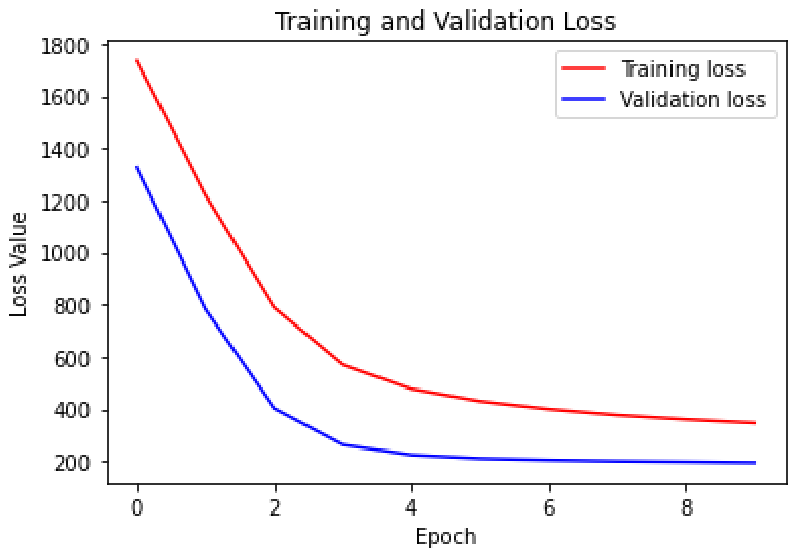
From the prepared spectogram arrays of 1275 train data and 158 test data for each input and output set, three different types of CNN models were run to predict the output array. The first model was a basic CNN model with minimum parameters with max pooling, drop out, conv 2d, transpose, and leaky relu layers embedded one time each. Total and trainable number of parameters were 29 and the model was compiled with adam optimizer to predict the mse and mae scores.

Figure 4. CNN Model One Summary



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 128, 646, 1)       10
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 323, 1)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64, 323, 1)        0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 128, 646, 1)       17
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 128, 646, 1)       0
_________________________________________________________________
dense_1 (Dense)              (None, 128, 646, 1)       2
=================================================================
Total params: 29
Trainable params: 29
Non-trainable params: 0
_________________________________________________________________
```

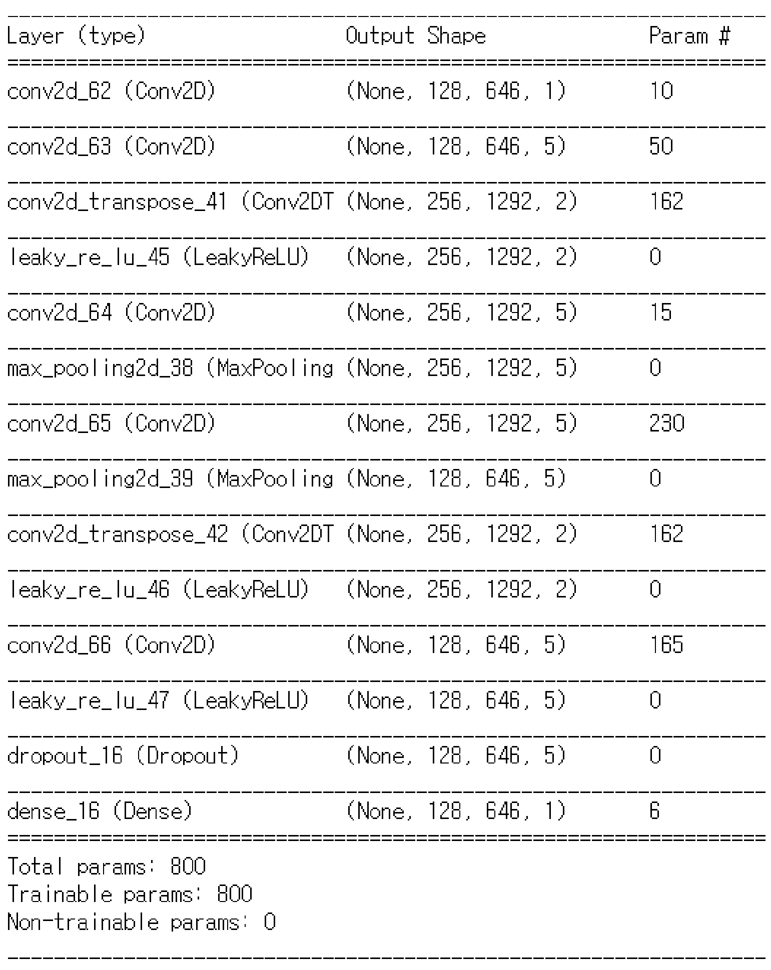
Figure 5. CNN Model One Training & Validation Result



Running 10 epochs of model one definitely showed steady decrease in loss value, however did not converge fast enough and could be run with more epochs for better fitting.

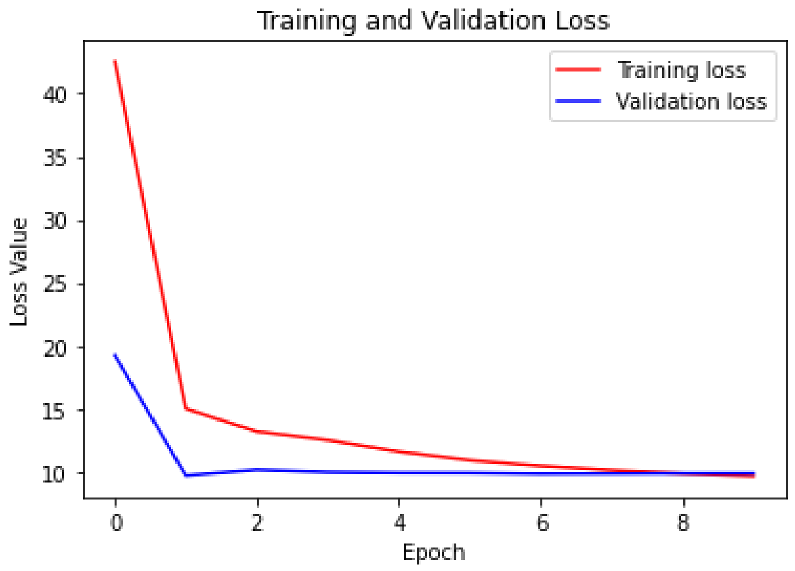
The second model was developed with more deeper convolution layers, with narrower layers with only few nodes in each layer, a total number of parameters of 800.

Figure 6. CNN Model Two Summary



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_62 (Conv2D)           (None, 128, 646, 1)       10
_________________________________________________________________
conv2d_63 (Conv2D)           (None, 128, 646, 5)       50
_________________________________________________________________
conv2d_transpose_41 (Conv2DT (None, 256, 1292, 2)      162
_________________________________________________________________
leaky_re_lu_45 (LeakyReLU)   (None, 256, 1292, 2)      0
_________________________________________________________________
conv2d_64 (Conv2D)           (None, 256, 1292, 5)      15
_________________________________________________________________
max_pooling2d_38 (MaxPooling (None, 256, 1292, 5)      0
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 256, 1292, 5)      230
_________________________________________________________________
max_pooling2d_39 (MaxPooling (None, 128, 646, 5)       0
_________________________________________________________________
conv2d_transpose_42 (Conv2DT (None, 256, 1292, 2)      162
_________________________________________________________________
leaky_re_lu_46 (LeakyReLU)   (None, 256, 1292, 2)      0
_________________________________________________________________
conv2d_66 (Conv2D)           (None, 128, 646, 5)       165
_________________________________________________________________
leaky_re_lu_47 (LeakyReLU)   (None, 128, 646, 5)       0
_________________________________________________________________
dropout_16 (Dropout)         (None, 128, 646, 5)       0
_________________________________________________________________
dense_16 (Dense)             (None, 128, 646, 1)       6
=================================================================
Total params: 800
Trainable params: 800
Non-trainable params: 0
_________________________________________________________________
```

Figure 7. CNN Model Two Training & Validation Result



The training and validation loss overlapped around epoch eight for model two. The training loss may continue to steadily decrease, but the model training seems to be stopped early for generalization trade-off. The loss scores were much lower than the first model.

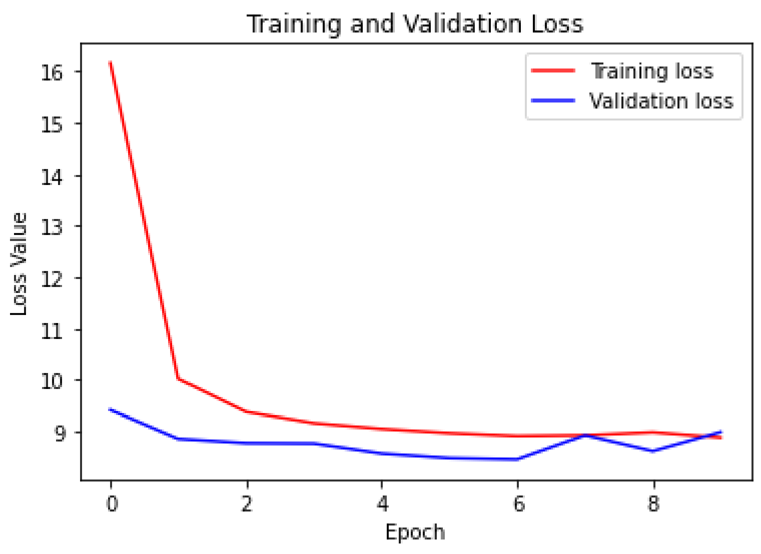
The third model was developed with more deep convolution layers, along with additional nodes on each layers, the total number of parameters of 38,611.

Figure 8. CNN Model Three Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_18 (Conv2D)           (None, 128, 646, 25)      250       
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 128, 646, 40)      9040      
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 64, 323, 40)       0         
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 128, 646, 20)      12820     
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 128, 646, 20)      0         
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 128, 646, 20)      3620      
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 64, 323, 20)       0         
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 128, 646, 40)      12840     
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 128, 646, 40)      0         
_________________________________________________________________
dropout_4 (Dropout)          (None, 128, 646, 40)      0         
_________________________________________________________________
dense_4 (Dense)              (None, 128, 646, 1)       41        
=================================================================
Total params: 38,611
Trainable params: 38,611
Non-trainable params: 0
_________________________________________________________________
```

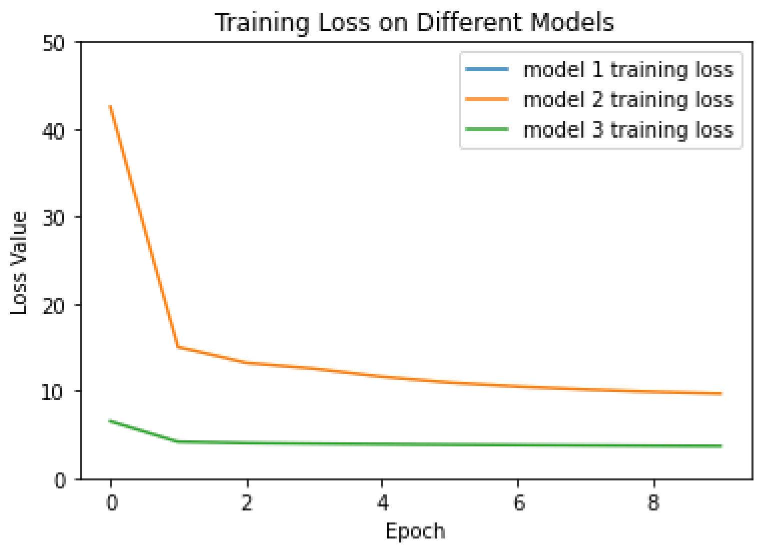
Figure 9. CNN Model Three Training & Validation Result

Figure 9. CNN Model Three Training & Validation Result



The validation loss started to increase around epoch seven for model three. And following is a graph summarizing performance of all three models.

Figure 10. Performances of different CNN Models



Here, we can see that model one is out of bounds since the convergence is too slow and model three is shown to perform the best. Observing this result, model three was chosen to train the dataset and the detailed evaluation is continued in the next section.

3. AutoEncoder : Ethan

Borrowing from Valerio Velardo, a notable figure in the AI Audio community, I implemented a preprocessing pipeline that involves the following steps:

- Load the midi files
- Convert the midi files to wav files
- Left or right pad the signal (if necessary)
- Extract log spectrogram of signal
- Save normalized spectrogram as .npy file

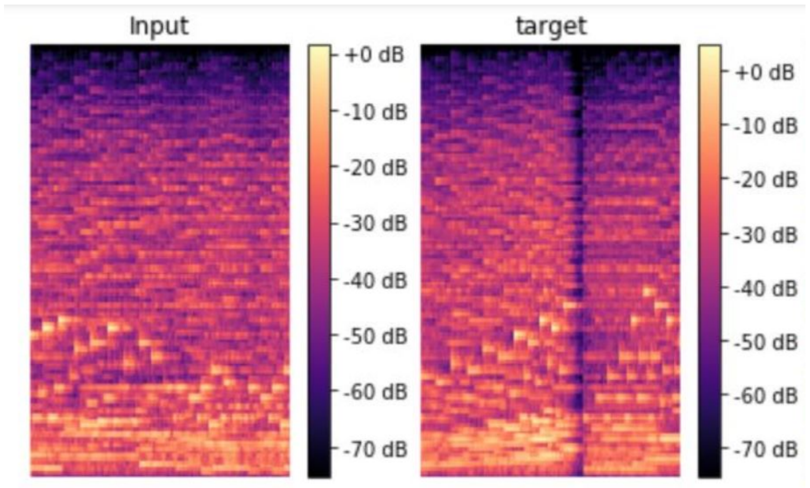
> At the end of this preprocessing pipeline, you end up with a directory that contains all of the normalized log spectrograms of the samples, as well as a pickle file file that contains the min and max values of the spectrograms, to be used for normalization (before being fed into the model) and eventually for denormalization (to go back to normal audio after conversion from spectrogram to audio). Log spectrograms were extracted using librosa's short-time Fourier transform function with log scaling.

> As for the Variational Autoencoder, I used a 5 convolutional layer architecture with mirrored encoder and decoder components and a two-dimensional latent space. I trained it for 100 epochs across all 1575 samples (extracted from original full MIDI files of Chopin pieces).

4. Transformer, GANs, Pix2Pix : Bethold

> I converted each of the spectrograms to a numpy array of (512,128), then I splitted into input data and target data of (256, 128) array respectively. Thus given an input audio, the goal is to predict the rest of the audio song. So the 15 seconds audio files were split into 5 seconds input and 5 seconds target. Below shows sample spectrogram for input and target 5 seconds audio files.

Figure 11. Sample spectrograms of input and target of 5 seconds respectively



> Total input and target data were (1431, 256, 128) numpy arrays respectively. From here I normalized the data to be between 0 and 1 by subtracting each value from the max and dividing the entire values by max. In this work there was no need to explore univariate and multivariate feature selection or dimension reduction because the wav files transformed into mel-spectrograms had the important features preserved.

> Next, I split the data into training and test data. The train consisted of 0.9 and test data consisted of 0.1 of the total input and target data. This gave us a training dataset of (1287, 256, 128) and test set of (144, 256, 128).

> Base models I utilized are Random Forest and tenor Regressor. I implemented 4 different models in addition to the base models. These models include Auto Encoders, Variational AutoEncoders, Deep Convolutional Generative Adversarial Network (DCGAN) and PIX2PIX. I utilized these models because we were dealing with image spectrograms and predicting pixel values.

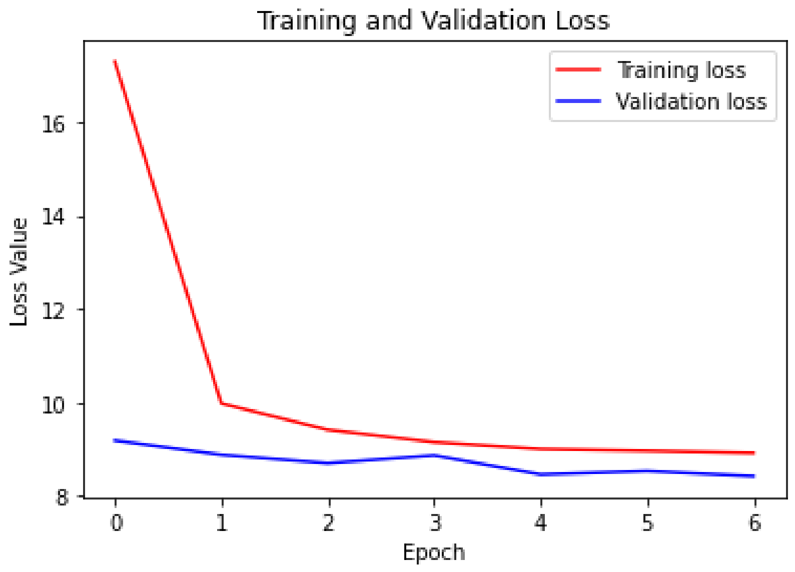
> Hyperparameters I utilized include the number of hidden layers, filters and activation functions in each model.

Evaluation

1. CNN : Hannah

Based on the previous results, model three was trained with the optimal epochs of seven and following is the training and validation loss measured as the following. Final mae score resulted in 8.9 for training and 8.4 for test data, much better than the baseline models which had an average mae score of 9.34 as can be referred in the method section.

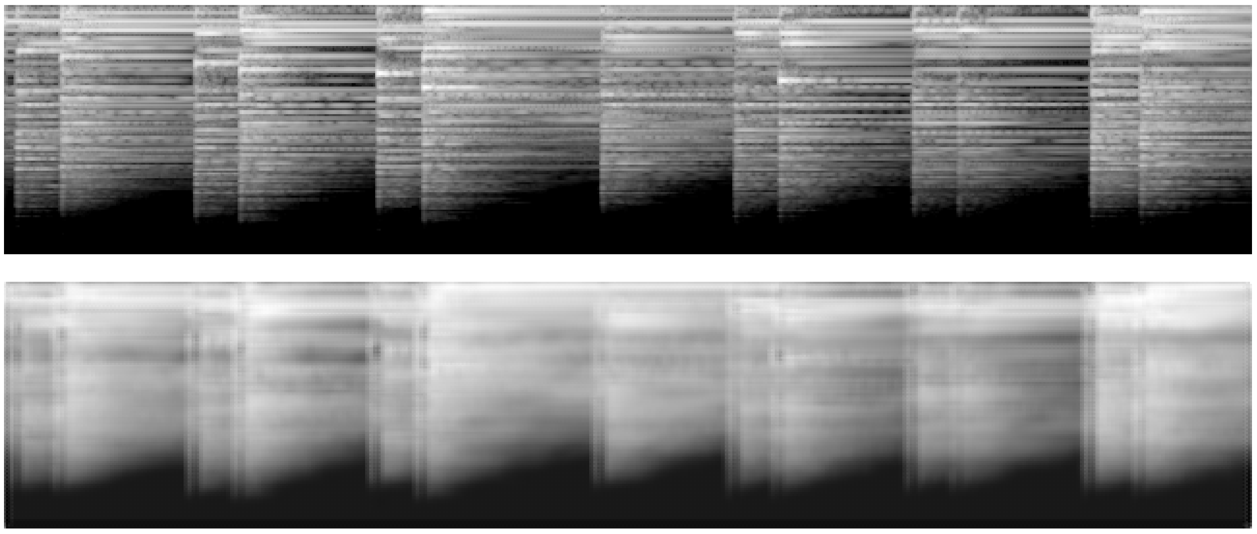
Figure 12. Model Three run with 7 epochs - early stop at optimal convergence



```
Epoch 1/7
40/40 [==============================] - 194s 5s/step - loss: 1139.8225 - mae: 26.2944 - val_loss: 172.5020 - val_mae: 9.1900
Epoch 2/7
40/40 [==============================] - 194s 5s/step - loss: 175.0071 - mae: 10.1181 - val_loss: 154.2593 - val_mae: 8.8856
Epoch 3/7
40/40 [==============================] - 194s 5s/step - loss: 152.9021 - mae: 9.5242 - val_loss: 149.8727 - val_mae: 8.7059
Epoch 4/7
40/40 [==============================] - 195s 5s/step - loss: 147.3443 - mae: 9.2866 - val_loss: 151.8076 - val_mae: 8.8730
Epoch 5/7
40/40 [==============================] - 195s 5s/step - loss: 136.5246 - mae: 8.9639 - val_loss: 146.4015 - val_mae: 8.4636
Epoch 6/7
40/40 [==============================] - 195s 5s/step - loss: 134.4252 - mae: 8.8763 - val_loss: 146.4263 - val_mae: 8.5363
Epoch 7/7
40/40 [==============================] - 195s 5s/step - loss: 135.7883 - mae: 8.9157 - val_loss: 146.1080 - val_mae: 8.4267
```

With the trained model, the test data was used to predict the output sequence array and each of the spectogram arrays were converted back to melspectograms to be converted to wav files. The spectograms showed similarities in rhythm and gradient variations, however the predicted output seemed to have a faded filter on the actual spectogram image with slight noise. The wav files in fact captured the rhythm of the sequence, however the keys or melodies were difficult to distinguish.

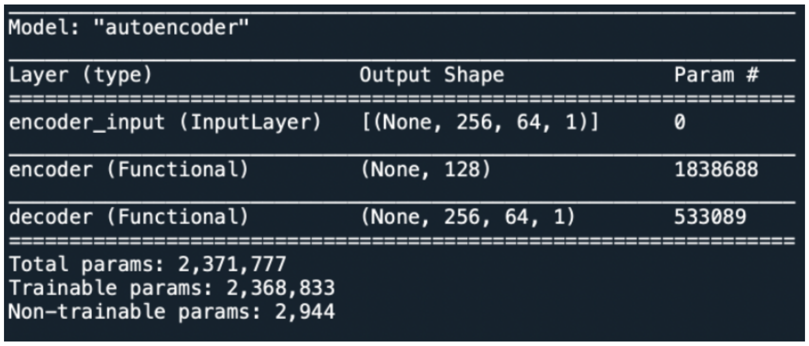
Figure 13. Spectograms of actual sequence (top) and predicted sequence (bottom)



2. AutoEncoder : Ethan

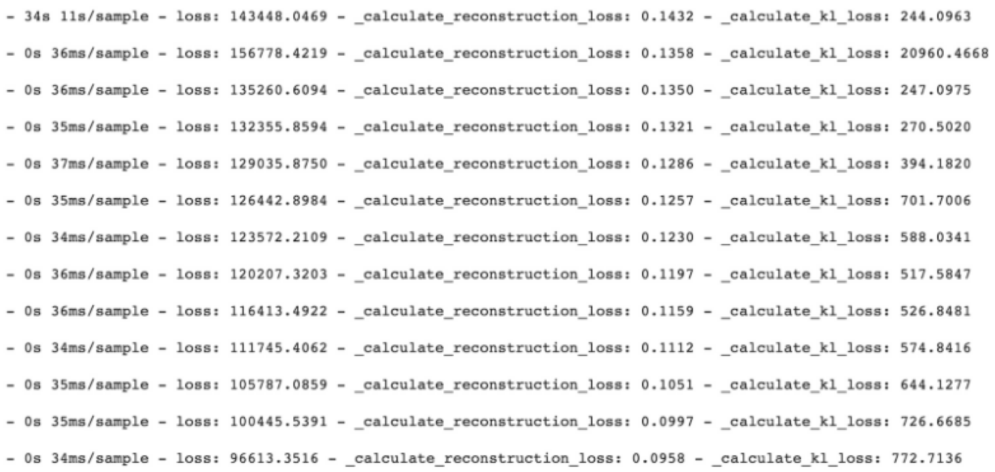
The final architecture for the variational autoencoder was as follows:

Figure 14a. Model Summary of Variational Autoencoder (VAE)



```
_________________________________________________________________
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 256, 64, 1)]      0
_________________________________________________________________
encoder (Functional)         (None, 128)               1838688
_________________________________________________________________
decoder (Functional)         (None, 256, 64, 1)        533089
=================================================================
Total params: 2,371,777
Trainable params: 2,368,833
Non-trainable params: 2,944
```

Figure 14b. Evaluation of Variational Autoencoder (VAE)



```
- 34s 11s/sample - loss: 143448.0469 - _calculate_reconstruction_loss: 0.1432 - _calculate_kl_loss: 244.0963

- 0s 36ms/sample - loss: 156778.4219 - _calculate_reconstruction_loss: 0.1358 - _calculate_kl_loss: 20960.4668

- 0s 36ms/sample - loss: 135260.6094 - _calculate_reconstruction_loss: 0.1350 - _calculate_kl_loss: 247.0975

- 0s 35ms/sample - loss: 132355.8594 - _calculate_reconstruction_loss: 0.1321 - _calculate_kl_loss: 270.5020

- 0s 37ms/sample - loss: 129035.8750 - _calculate_reconstruction_loss: 0.1286 - _calculate_kl_loss: 394.1820

- 0s 35ms/sample - loss: 126442.8984 - _calculate_reconstruction_loss: 0.1257 - _calculate_kl_loss: 701.7006

- 0s 34ms/sample - loss: 123572.2109 - _calculate_reconstruction_loss: 0.1230 - _calculate_kl_loss: 588.0341

- 0s 36ms/sample - loss: 120207.3203 - _calculate_reconstruction_loss: 0.1197 - _calculate_kl_loss: 517.5847

- 0s 36ms/sample - loss: 116413.4922 - _calculate_reconstruction_loss: 0.1159 - _calculate_kl_loss: 526.8481

- 0s 34ms/sample - loss: 111745.4062 - _calculate_reconstruction_loss: 0.1112 - _calculate_kl_loss: 574.8416

- 0s 35ms/sample - loss: 105787.0859 - _calculate_reconstruction_loss: 0.1051 - _calculate_kl_loss: 644.1277

- 0s 35ms/sample - loss: 100445.5391 - _calculate_reconstruction_loss: 0.0997 - _calculate_kl_loss: 726.6685

- 0s 34ms/sample - loss: 96613.3516 - _calculate_reconstruction_loss: 0.0958 - _calculate_kl_loss: 772.7136
```

Each layer in the encoder consists of a convolutional layer followed by a Relu activation and Batch normalization. Finally, they are flattened and mirrored in the decoder. Three measures of loss were calculated and an example of training with only 3 samples is shown below: The overall loss is provided by the reconstruction error multiplied by a weight plus the KL Divergence loss, the difference between a normal distribution from the standard normal distribution. All three measures of loss decreased throughout the range of epochs and showed no signs of overfitting. Due to the size of the dataset, hyperparameters had to be chosen to end up with a number of temporal frames equal to a power of 2 - this forced the duration to be fed into the stft librosa function to be .74 seconds. As a result, the predictions of the VAE were limited to very short snippets. The predictions themselves were made by first reshaping the log spectrogram (to omit the 3rd dimension reserved for the number of color channels), applying denormalization (using the saved max/min values) and finally applying the Griffin-Lim algorithm to convert the spectrogram to audio (what's under the hood in the librosa implementation).

3. Transformer, GANs, Pix2Pix : Bethold

1. Performance of First Auto Encoder Model

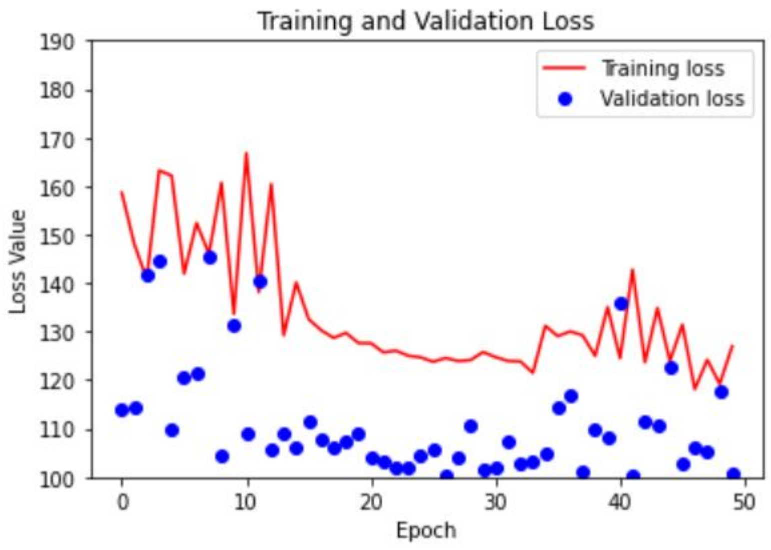
This model had 4 hidden layers with filters of 16, 32, 32, 16 filters respectively. The relu activation was utilized here. Below shows the model summary :

Figure 15. Model Summary of First Autoencoder

```
_________________________________________________________________
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 440, 128, 1)]     0
_________________________________________________________________
encoder (Functional)         (None, 16)                18672
_________________________________________________________________
decoder (Functional)         (None, 440, 128, 1)       31233
=================================================================
Total params: 49,905
Trainable params: 49,905
Non-trainable params: 0
```

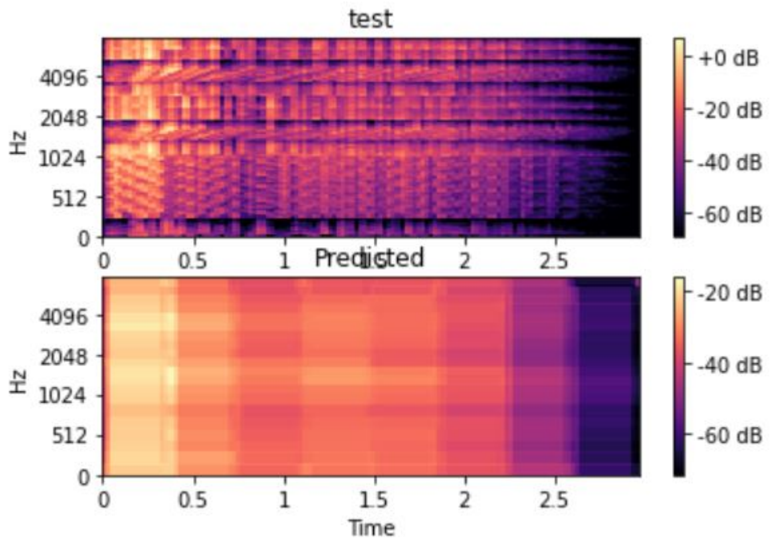
I utilized the loss function of mean square error (mse) and Adam optimizer as optimization function with a learning rate of 0.001. After training with the normalized train dataset and its target, I obtained the lowest loss to be 130 after 50 epochs.

Figure 16. Training and Validation Loss of First Autoencoder



I evaluated the test dataset and I obtained figure below :

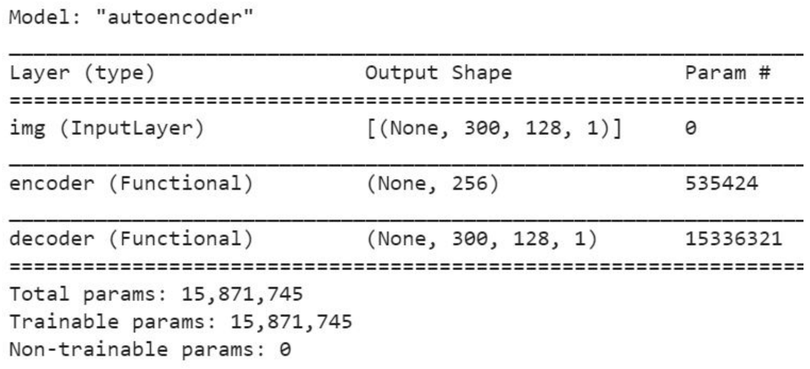
Figure 17. Evaluation of First Autoencoder



2. Performance of Second Auto Encoder Model

> This model had 5 hidden layers with filters of 32, 64, 128, 128, 256filters respectively. The relu activation was utilized here. Below shows the model summary :

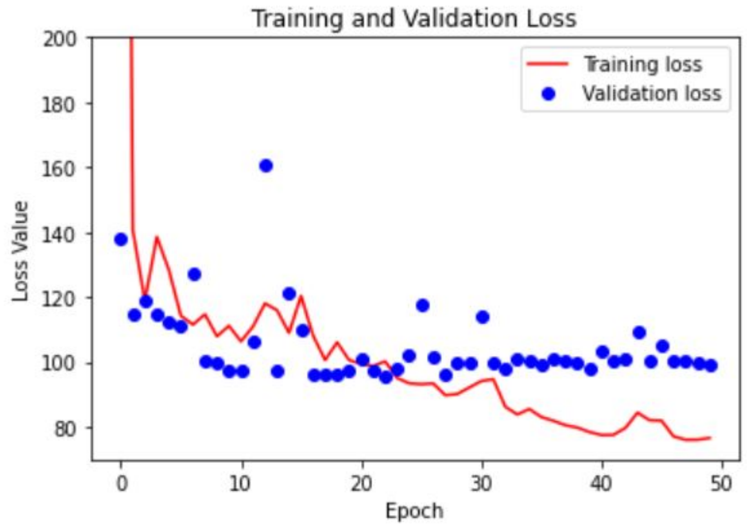
Figure 18. Model Summary of Second Autoencoder



```
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 300, 128, 1)]     0
_________________________________________________________________
encoder (Functional)         (None, 256)               535424
_________________________________________________________________
decoder (Functional)         (None, 300, 128, 1)       15336321
=================================================================
Total params: 15,871,745
Trainable params: 15,871,745
Non-trainable params: 0
```

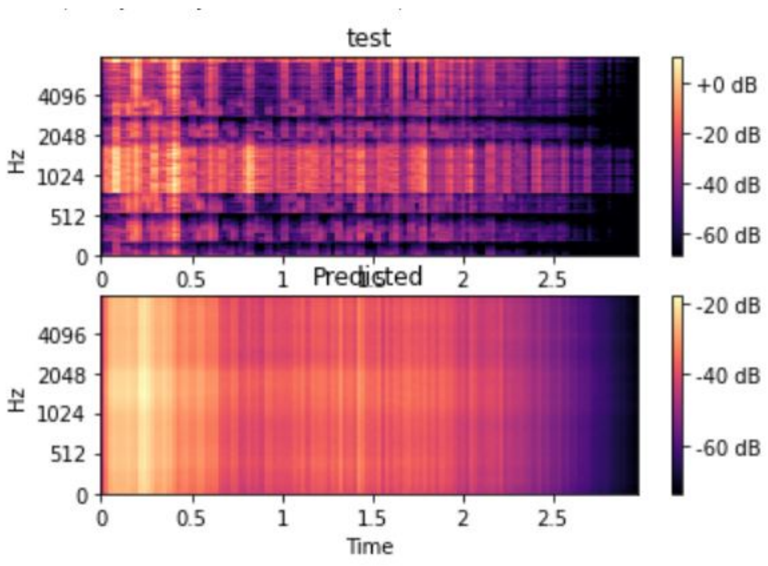
> I again utilized the loss function of mean square error (mse) and Adam optimizer as optimization function with a learning rate of 0.001. After training with the normalized train dataset and its target, I obtained the lowest loss to be 80 after 50 epochs.

Figure 19. Training and Validation Loss of Second Autoencoder - 5 hidden layers



I evaluated the test dataset and I obtained figure below.

Figure 20. Evaluation of Second Autoencoder - 5 hidden layers on sample test data



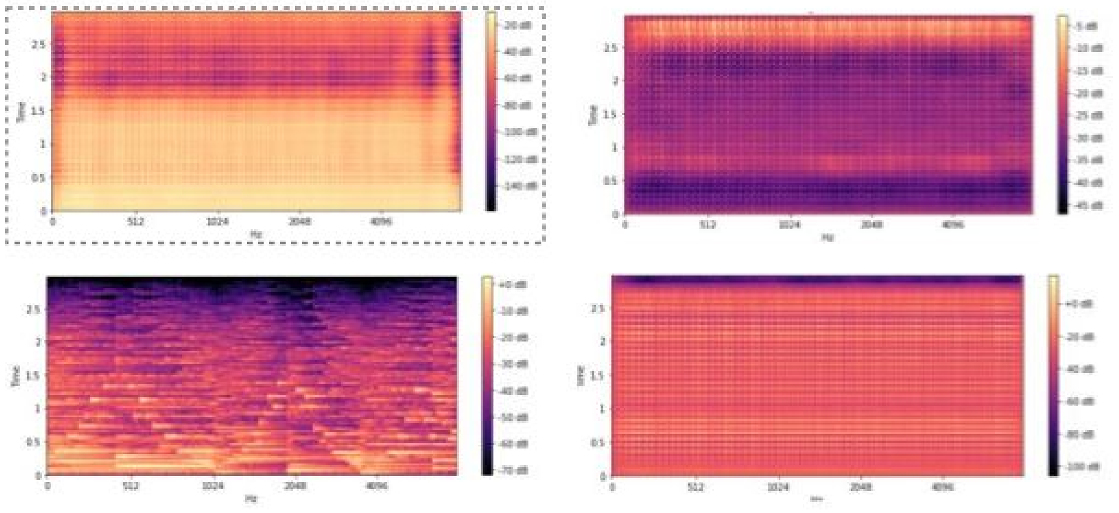
3. Performance of DCGAN (Third Model)

The generative model had 6 hidden layers with filters of 256, 128, 128, 64, 32 filters respectively. The activation utilized here was leaky relu. The discriminator had 2 hidden layers of 64 and 128 filters respectively.

Both the generator and discriminator had binary cross entropy as loss function. Adam optimizer was utilized as an optimization function with a learning rate of 0.0001. I trained for about 20 epochs.

Images generated from DCGAN along with a sample real image is shown below :

Figure 21. Sample test data and images generated from DCGAN



4. Performance of PIX2PIX (Fourth Model)

> The generative model had both upsample and downsample concatenated together with 4 hidden layers which had filters of 32, 64, 128 and 128 respectively. The downsample had an activation of leaky Relu while upsample had an activation relu. The discriminator had downsample of 3 hidden layers of 64, 128 and 256 filters respectively.

> Both the generator and discriminator had binary cross entropy as loss function. Adam optimizer was utilized as an optimization function with a learning rate of 0.0002 and beta of 0.5. I trained for about 30 epochs.

Figure 22. Loss graph of the Discriminator

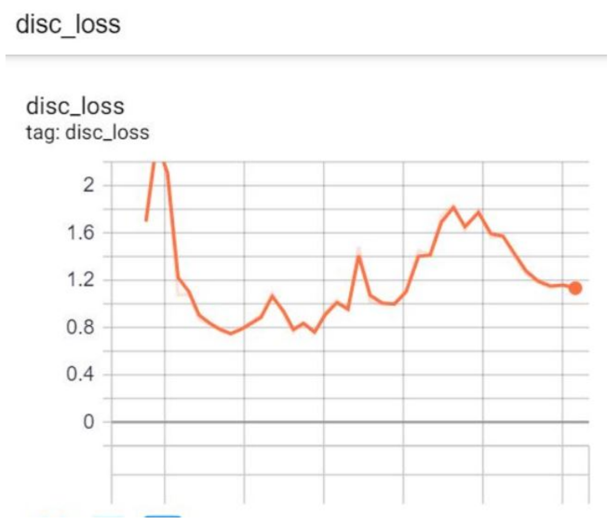
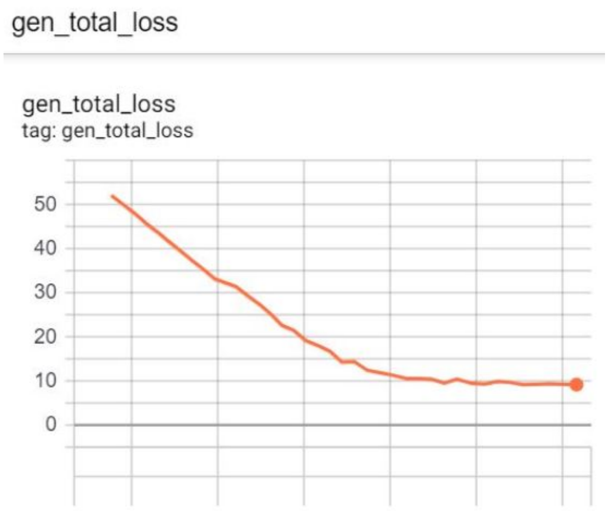
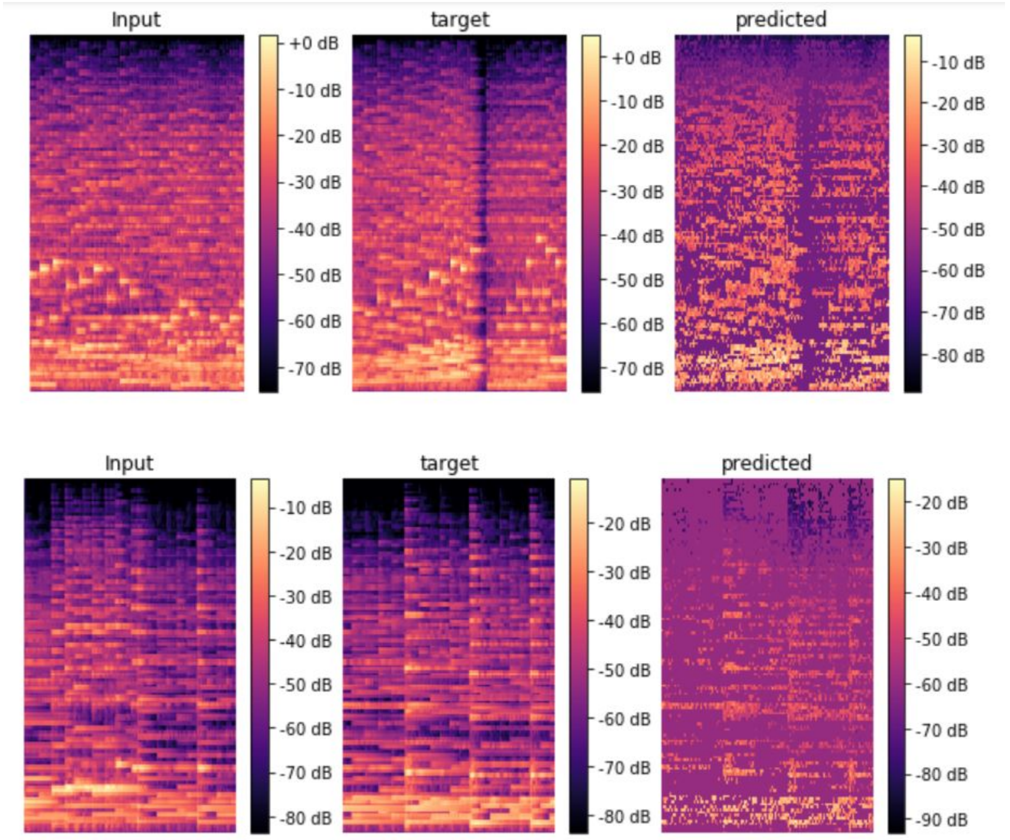


Figure 23. Loss graph of the Generator



I evaluated the PIX2PIX model on the test dataset and I obtained figures below :

Figure 24. Input, target and predicted results from PIX2PIX model



Conclusion

In this project, we have tried out different DNN models on spectogram prediction, starting from CNN, commonly used for image analysis, to AutoEncoders, Transformer, GANs, to Pix2Pix. Although CNN was a rather simple approach, regularizing the parameters and creating different shaped models resulted in creating a final model with fairly low mse scores, and produced a spectogram that preserves the rhythm of a sequence. Autoencoders, transformers and GANs on the other hand did not produce high performance compared to its complexity. And our final model - Pix2Pix model produced most accurate soundtrack, considering it uses a conditional generative adversarial network (cGAN) to learn mapping an output image from an input image.

The soundtrack produced by converting the spectograms from different models are visualized as the following :

Figure 25. Audio Visualization of spectograms from CNN, target waveplot (top) and predicted waveplot (bottom)

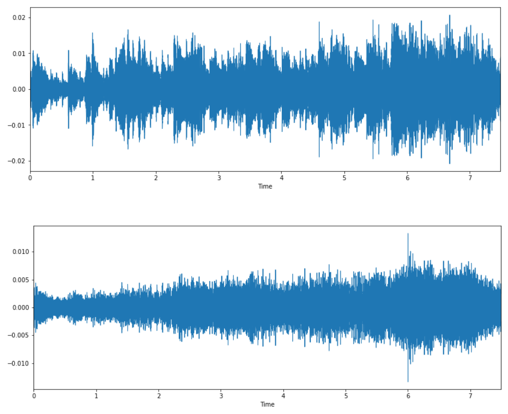


Figure 26. Audio Visualization of spectograms from Auto Encoder One

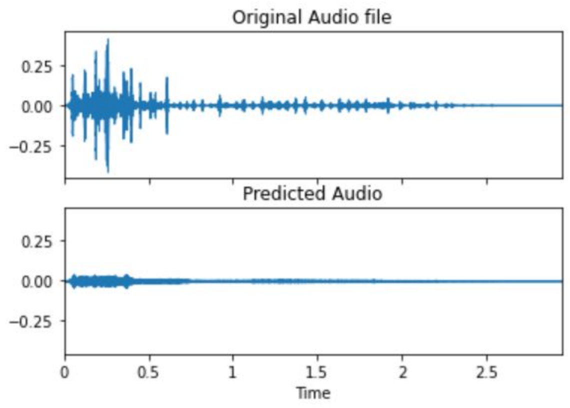


Figure 27. Audio Visualization of spectograms from Auto Encoder Two

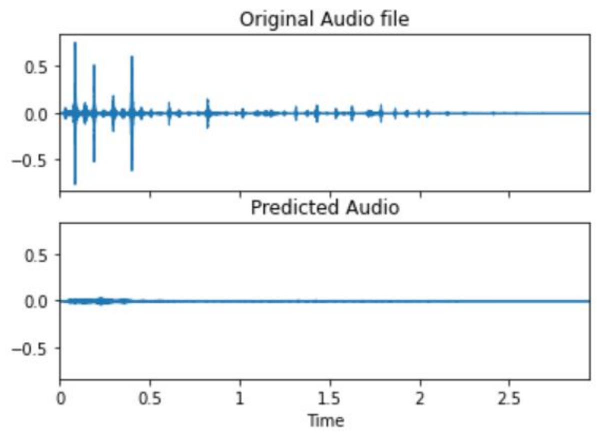


Figure 28. Audio Visualization of spectograms from DCGAN

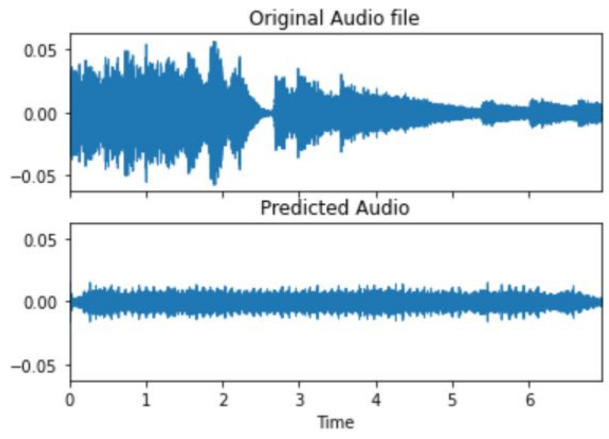
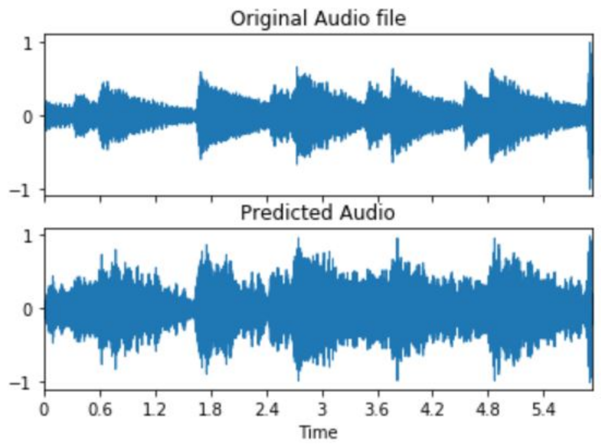


Figure 29. Audio Visualization of spectograms from Pix2Pix

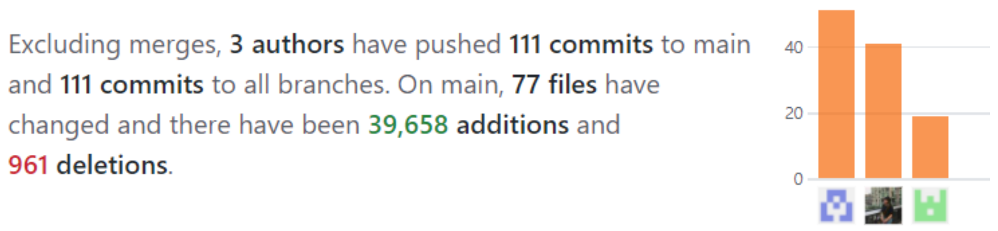


As seen in the figures above, CNN spectogram produces some sort of similarity in the magnitude of the sound, hoewever does not show high accuracy in following the target waveplot. And the two Auto Encoders and DCGAN do not seem to produce any meaningful audio sound, they were hashed with noises.

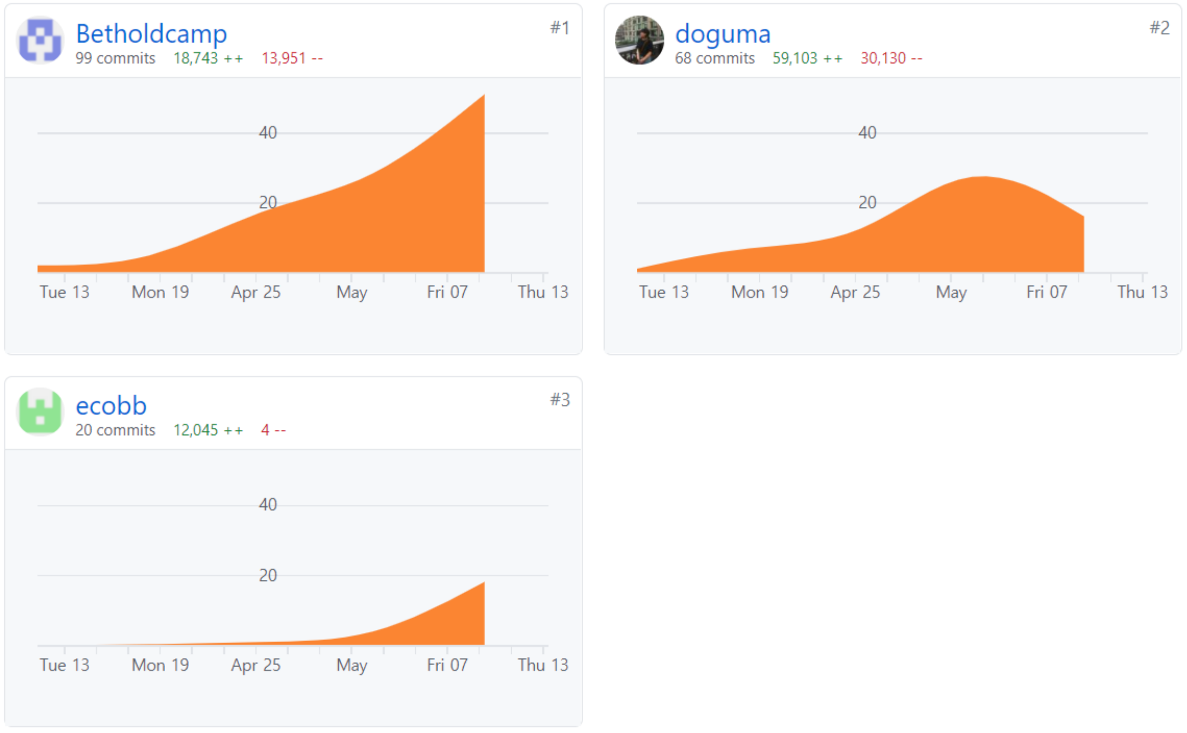
On the other hand, the **PIX2PIX model generated the best spectrograms and wav audio** as shown in figure 29, as the predicted wav audio shows close similarilty to the target audio. Compared to other methods that work well for the classification problems, the PIX2PIX method (cGAN) seem to excel in image-to-image translation.

Attribution

Figure 29. Github Summary and Insights



Excluding merges, **3 authors** have pushed **111 commits** to main and **111 commits** to all branches. On main, **77 files** have changed and there have been **39,658 additions** and **961 deletions**.



Betholdcamp #1
99 commits 18,743 ++ 13,951 --

doguma #2
68 commits 59,103 ++ 30,130 --

ecobb #3
20 commits 12,045 ++ 4 --

Betholdcamp : Bethold Owusu

doguma : Hannah Do

ecobb : Ethan Cobb

- Link to Github : https://github.com/CSCCNY/final-project-recomposeclassics (https://github.com/CSCCNY/final-project-recomposeclassics)

Bibliography

Briot, Jean-Pierre, et al. “Deep Learning Techniques for Music Generation -- A Survey.” ArXiv:1709.01620 [Cs], Aug. 2019. arXiv.org, http://arxiv.org/abs/1709.01620 (http://arxiv.org/abs/1709.01620).

Rivero, Daniel, Iván Ramírez-Morales, Enrique Fernandez-Blanco, Norberto Ezquerra, and Alejandro Pazos. "Classical Music Prediction and Composition by Means of Variational Autoencoders." Applied Sciences 10, no. 9 (2020): 3053. doi:10.3390/app10093053.

Szelogowski, Daniel. “Generative Deep Learning for Virtuosic Classical Music: Generative Adversarial Networks as Renowned Composers.” ArXiv:2101.00169 [Cs, Eess], Apr. 2021. arXiv.org, http://arxiv.org/abs/2101.00169 (http://arxiv.org/abs/2101.00169)

Rakshit, Soumik, "Classical Music Midi" Classical Music MIDI files for compositions of various composers, 2019. Kaggle, https://www.kaggle.com/soumikrakshit/classical-music-midi (https://www.kaggle.com/soumikrakshit/classical-music-midi)