

Classifying Classical Composers Using a CNN-LSTM Deep Learning Model

Anitra Hernandez, Johnathan Kelsey, and Lakshmi Deepti Pamula

Shiley-Marcos School of Engineering, University of San Diego

AAI-511 Neural Networks and Deep Learning

Professor Rod Albuyeh, Ph.D

11 August 2025

Abstract

When speaking of “classical” music, one is most often referring to music created between 1750 and 1820. The music’s symmetry and order often set it apart from other genres of music. Classical music has inspired modern music in many ways and is still largely studied by people interested in creating music. The tonality of the piece can elicit an emotional response and alter the mood of the listener. However, connecting with a piece of music and desiring to hear more pieces by the same composer requires the listener to be able to identify which composer created the piece, a skill that takes time to develop. This may lead the novice listener to chase through the genre in hopes of finding more pieces that sound similar or that make them feel a certain way. The goal of this project is to develop a deep learning model that can predict the composer of a given musical score accurately. Such a model will help the novice listener who connects with a particular composition to identify the composer so that he or she can search out more pieces by that particular artist.

Classifying Classical Composers Using a CNN-LSTM Deep Learning Model

This project focuses on a combined architecture deep learning model that will be trained and used to predict which of four composers made a particular piece of music. The architecture will be the combination of a Convolutional Neural Network, CNN, and a Long Short-Term Memory, LSTM network (Sequence Classification Using CNN-LSTM Network - MATLAB & Simulink, n.d.). CNNs are generally very good at classifying images and are used heavily in computer vision problems. They consist of convolution layers and fully connected layers. The convolution layers learn to create meaningful feature maps from the image and then pass those feature maps onto the fully connected layers. The fully connected layers are the portion of the network that learns to recognize how the feature maps created by the convolution layers coincide with the various categories available to choose from.

LSTMs are a special type of Recurrent Neural Networks, specifically designed to forget context that provides little meaning while retaining the context that matters. What context is important enough to keep and what content can be safely discarded are learned by the network. Because of their ability to retain important context, LSTMs naturally lend themselves to time-series and sequencing problems. Because music is a series of notes over a series of time-steps, LSTMs may seem the obvious choice for use with music. However, because LSTMs excel at forecasting, the more appropriate use of an LSTM with music would be to learn from existing music and then produce new music through the repeated forecasting of the next musical token. This is not the goal of this project; the goal of this project is classification, a goal that is arguably more aligned with the nature of the CNN architecture. However, LSTM's strong ties to sequencing and time-series make it difficult to ignore when dealing with musical data.

To ensure use of both architectures, this project chose to combine the two, keeping the convolutional layers of the traditional CNN architecture and feeding the feature maps into an LSTM instead of the expected fully connected layers found in a straightforward CNN. In this way, the CNN processes the data by applying multiple kernel filters to the input, creating a stack of feature maps that are then fed into the LSTM, which processes each feature map as a sequence of data, learning long-term dependencies between the various feature maps. Understanding the internal workflow of the combination CNN-LSTM model is necessary to understand the data requirements needed to make such a model work.

In order for the CNN-LSTM model to work, the data used must be both visual and sequential. To accommodate this, this project converted the raw audio into a spectrogram, a visual representation of the music, and then took multiple time samples of the spectrogram and concatenated them side by side so that the visual of the music became something akin to a time series. This image was then transformed into a tensor and used to train and test the model. However, before any of this could happen, there was a whole different type of pre-processing that had to be done.

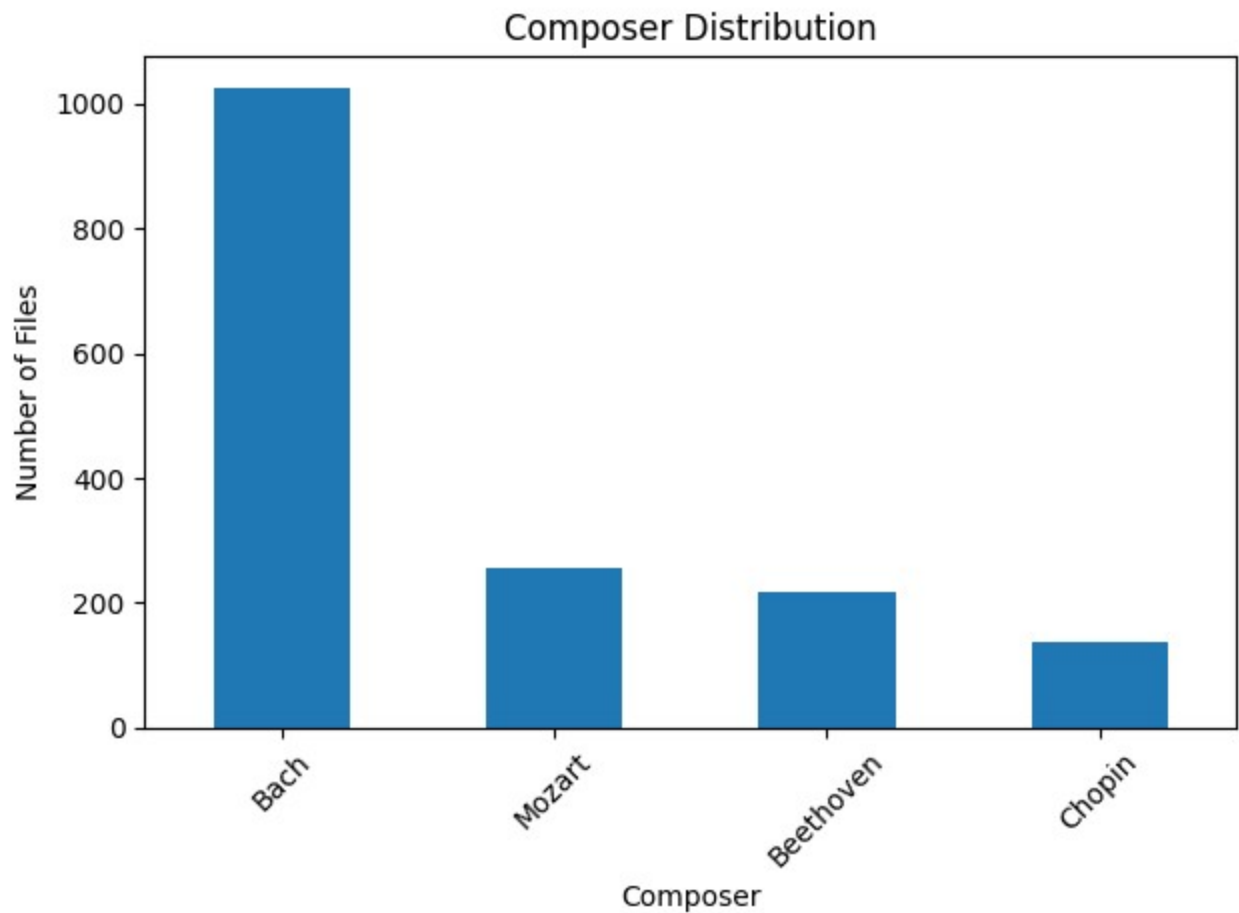
Data Pre-processing

The dataset used for this project is the 'midi_classic_music' dataset found at [kaggle.com](https://www.kaggle.com/bohdaan) (Bohdan Fedorak, 2019). As one may surmise from the title of the dataset, this is MIDI data from multiple classical composers. It is important to note that MIDI data is not audio data. Instead, MIDI data is more akin to metadata about the music. It is comprised of a series of data points that indicate when a note starts, when a note ends, how hard it is played, and by what instrument, among others. This project focused on the music of only four composers: Bach, Beethoven, Chopin, and Mozart. When looking at the dataset, it became

apparent that there was a high discrepancy between the volume of pieces created by each composer. Figure 1 shows the raw number of works for each composer.

Figure 1

Raw File Count for Each Composer of Interest



However, this is only part of the story. By tokenizing the data, the disparity between the composers became even greater. This is because there are multiple tokens associated with each note, and thus, the note density of the piece drastically influences the total token count for any particular piece of work. In this way, tokenization acts as a multiplier so that the raw token count is proportionally greater based on the number of tokens per note. Ironically,

Chopin, the composer with the fewest raw examples, also had the least dense compositions when viewed from a number of notes per bar standpoint.

In general, it is a good practice to feed balanced data into a Machine Learning model if one hopes to get viable and accurate results. To that end, there were several options for balancing the data between the various composers. The first step was to tokenize everything and then try to make sure that each composer was represented by a similar number of tokens. After splitting the files based on token count, the options for balancing the representation were to either under-sample the overrepresented classes, over-sample the underrepresented classes, or add a weighting system that gave greater weight to the underrepresented classes and less weight to the overrepresented ones. As with all decisions, it is important to balance the pros and cons of each approach. One important aspect of models that generalize well is data augmentation. This is a process that systematically alters the original data in ways that keep it similar to the original data without it being an exact copy. To utilize data augmentation, this project would require increasing the amount of data used. However, upon doing so, the volume of data very quickly got out of hand, making it so that training was unfeasible on anything less than top-tier hardware to which this project had no access. To remediate this issue, just under one percent of the original data was selected through the use of integer division on the original composer file count. This brought the total file count down to just fifteen files that were then broken up by token count and augmented. After augmentation, the composer files were unbalanced, so a random sample of files from each composer was selected, and the rest were discarded. This brought the total data being worked with down to just 400 files, 100 for each composer.

Feature Extraction Techniques

Once the data was augmented and balanced out, the data was transformed from the MIDI data, as shown in Figure 2, to the audio wave, as shown in Figure 3.

Figure 2

Midi Data Representation of File

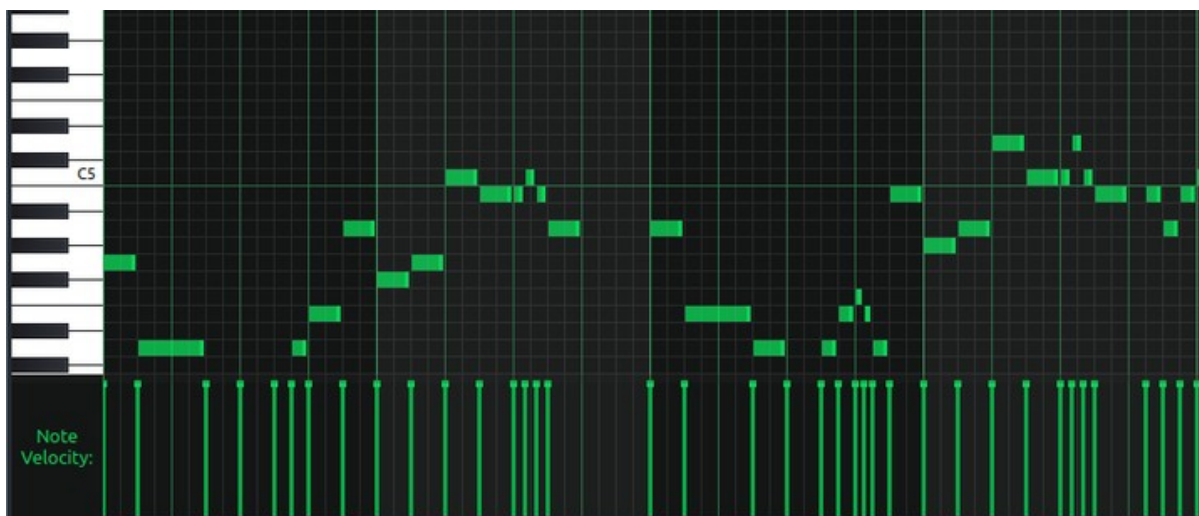
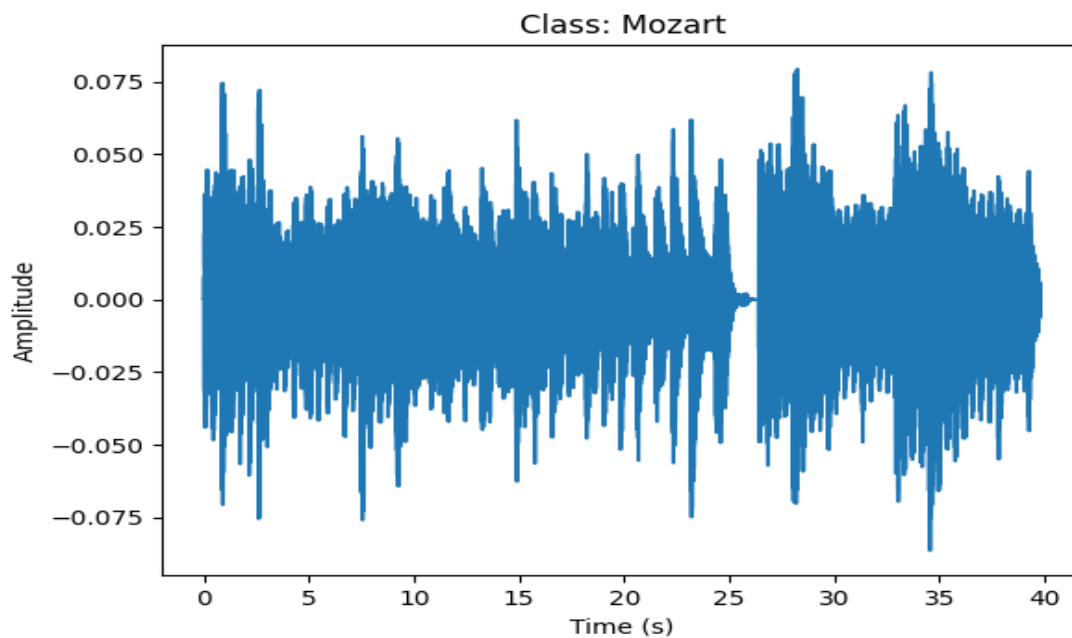


Figure 3

Audio Wave Representation



The raw audio wave was then converted to a spectrogram, as shown in Figure 4, and sampled to produce the final image representation of the file shown in Figure 5.

Figure 4

Spectrogram Representation

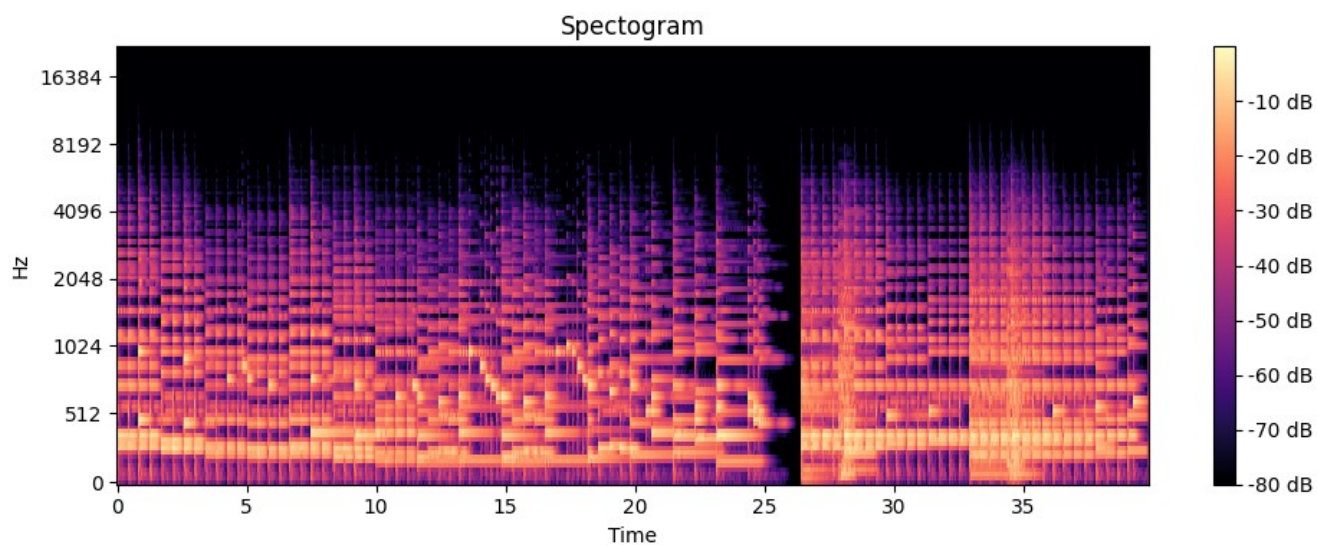
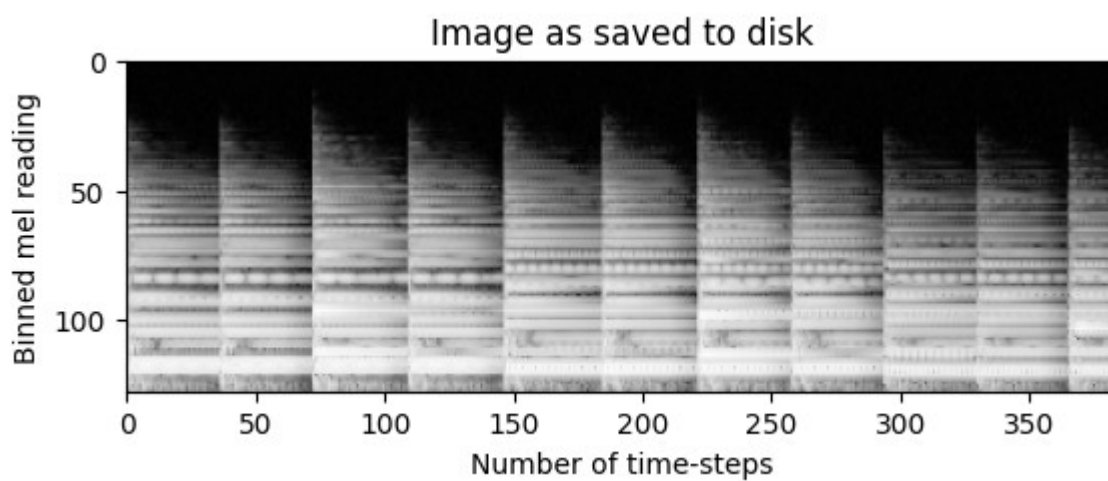


Figure 5

Final Image Used as Input to the CNN Portion of the CNN-LSTM Model



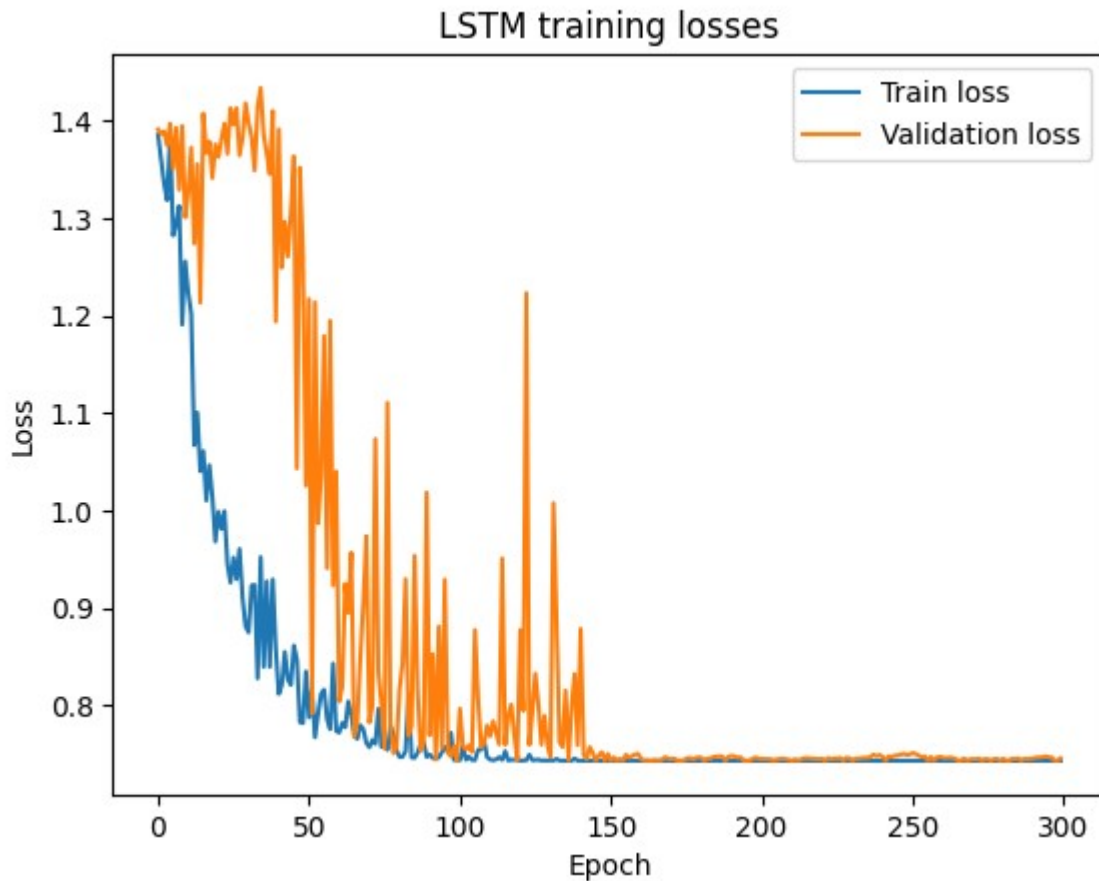
From here, the model itself will learn what features of the image it should pay attention to as it creates various feature maps by running the image through its convolution layers.

Modeling Building

The model architecture consists of four convolution layers, each consisting of four separate actions: convolve, normalize, rectified linear unit activation, and pooling. The kernel size for the convolve action is the same across all four layers, a 3 by 3 matrix, but the kernel size for the pooling action varies across the layers, ranging from a 2 by 2 matrix to a 4 by 2 matrix. After running through all the convolution layers, the tensors are flattened out and fed into the LSTM portion of the model. At this point, there are 3,072 components in the vector. The LSTM learns what is important and ultimately passes a representation of what it has learned in the form of a 256-component vector to a linear activation layer that filters it down to only 4 components, one for each composer. This is then run through the softmax function, effectively converting the final vector into a probability distribution of the four composer options.

Model Training

Sixty percent of the image files were used to train the model. The remaining forty percent was split in half to form one dataset used for validation and another used for testing. This means that there were 240 images used to train the model. The model ran through the training data 300 times and used cross-entropy as the loss function for back propagation. The learning rate was held static at 0.001. No dropout or early stopping was used. After each cycle of training, the model ran through the validation data to gather a feel for how it was progressing. Figure 6 shows the graph of the loss for both the training dataset and the validation dataset. The validation loss, especially, was very volatile over the first 150 cycles, but then settled down as the model got closer to converging on a minimum loss.

Figure 6*Graph of Loss by Cycle Count*

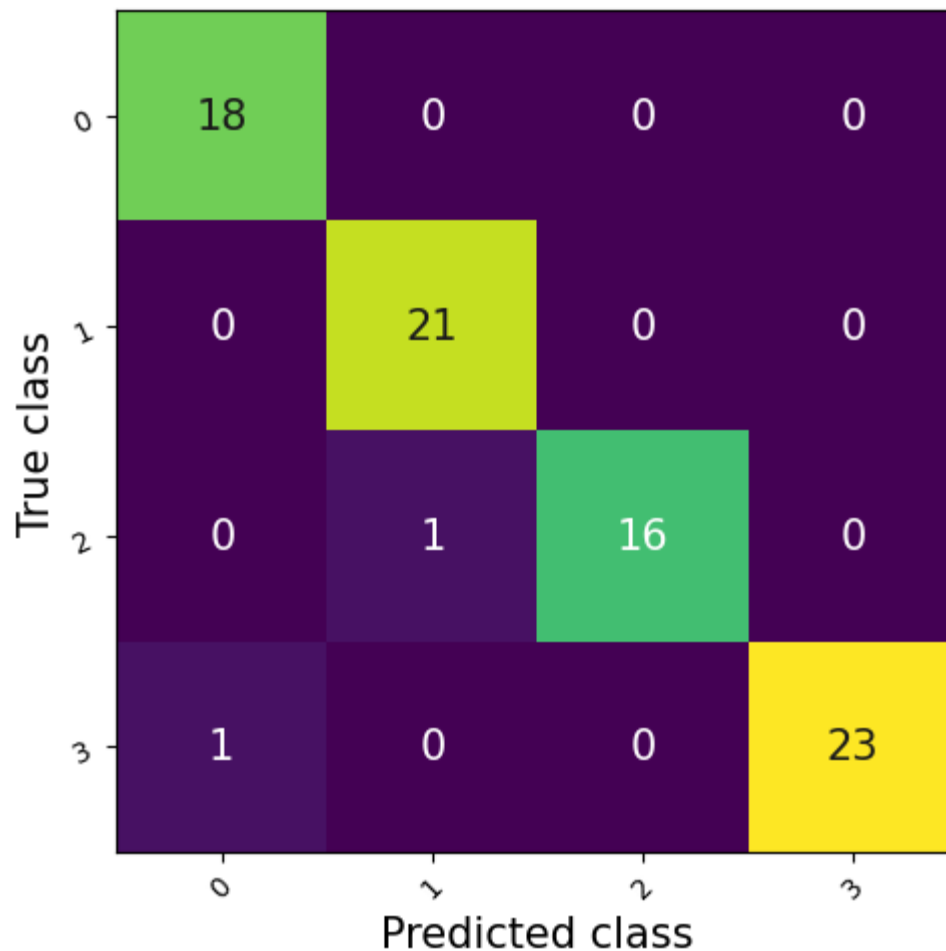
Model Evaluation

To evaluate the performance of this model, the test dataset, comprised of 80 unseen images, was used to test the trained model. The model took each image as an input and output the composer that the model thought most likely to have composed the music used to make the image. Raw accuracy, precision, recall, and an F1 score were all used as metrics to determine the efficacy of the model's classification powers. Further, a multi-class confusion matrix was used to visualize this data. As can be calculated from the confusion

matrix shown in Figure 7, the accuracy, precision, recall, and F1 are all above 97%, with only two files being inaccurately predicted.

Figure 7

Multi-class Confusion Matrix of Trained Model Predictions on Test Dataset



Model Optimization

The model's performance didn't leave much room for optimization. As such, this project chose to create two additional models: a standalone CNN and a standalone LSTM model. The CNN was able to use the same images as the dual-architect model and was relatively simple to train and test against. Much like the dual architecture, the baseline CNN

performed very well, achieving ninety percent or greater across all four performance metrics. After optimization, all four metrics rose, but not enough to justify the efforts it took to optimize the model's hyperparameters.

The LSTM worked a little differently. It did not operate on images and so needed to use the raw token sequence originally extracted from the MIDI files. Upon running the training cycle on the baseline LSTM, it was quickly noted how slow the model was training. This was likely due to manual padding being used to ensure all sequences were of the same length. The speeds improved substantially once manual padding was removed and replaced by a packing sequence library function called 'pack_padded_sequence'. The number of training cycles was also reduced from 300 to only 100. This reduced training time enough to allow for more testing and greater architectural optimization.

A second LSTM layer was added to provide better capacity to capture more complex context, increasing the representational learning. The number of hidden nodes was decreased from 256 to 128, and gradient clipping was introduced to guard against exploding and vanishing gradients. The LSTM was changed to a bidirectional LSTM, allowing the model to capture context from both past and future tokens. This helped in understanding musical structure more holistically, leading to better generalization and higher accuracy. Overall, the baseline model, set to mimic the single-layer LSTM used in the dual architecture, achieved only 38% accuracy. After employing the optimization techniques outlined above, the optimized model achieved 90% accuracy.

Conclusion

While each standalone model, CNN and LSTM, was able to be optimized to make fairly decent classifications, neither was as accurate as the dual architecture that combined the two into a single CNN-LSTM model. The CNN-LSTM model required heavier pre-processing than

either standalone model, but proved to be worth that effort by providing such powerful results. The world of AI will likely see more and more combinatorial models in the future as AI practitioners continue to realize new ways to capture the most powerful aspects of different models.

References

Bohdan Fedorak. (2019). midi_classic_music. Kaggle.com.

<https://www.kaggle.com/datasets/blanderbuss/midi-classic-music>

Sequence Classification Using CNN-LSTM Network - MATLAB & Simulink. (n.d.).

Www.mathworks.com. <https://www.mathworks.com/help/deeplearning/ug/sequence-classification-using-cnn-lstm-network.html>