Classifying Music with Deep Learning: Comparing RNN and CNN Methods

Shaun Friedman, Ali Azizi

Applied AI Program, University of San Diego

AAI 511: Neural Networks and Deep Learning

Prof. Kahila Mokhtari, Ph.D

August 11, 2025

Abstract

This study investigates the performance of two deep learning architectures—Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs)—for classifying classical music compositions by four composers: Bach, Beethoven, Chopin, and Mozart. Using a dataset of 1,628 MIDI files, we developed two parallel processing pipelines tailored to each model's strengths. For CNNs, MIDI data was transformed into spectrogram-based representations, including a lightweight Simple Array Representation (SRA) and more computationally intensive Mel spectrograms, with models trained on short temporal windows (3–5 seconds). For RNNs, we extracted symbolic musical features such as pitch intervals, rhythmic ratios, pitch classes, dynamics, and instrument programs, then modeled temporal dependencies using bidirectional LSTMs. Both architectures underwent extensive preprocessing, data balancing, and augmentation to address class imbalance and improve generalization. Results showed that the best CNN (SRA, 5-second windows, 16,000 samples) achieved 85.8% accuracy, while the optimized RNN reached 85% accuracy after rebalancing, both substantially outperforming the 25% baseline from random guessing. The CNN proved more efficient in feature extraction time, while the RNN demonstrated robustness in leveraging symbolic music features. These findings suggest that both architectures are viable for symbolic music classification, with trade-offs in computational cost, preprocessing complexity, and representational flexibility.

Keywords: music classification, deep learning, CNN, RNN, LSTM, spectrogram, symbolic music, MIDI, composer identification

Classifying Music with Deep Learning: Comparing RNN and CNN Methods

Deep Neural Networks (DNNs) have become a popular tool for solving many types of modeling problems and as compute resources have increased in availability and decreased in price, ever more sophisticated forms of these models have been developed. (Alzubaidi et al., 2021) Despite these advancements, it is not always clear which type of model is most suitable for a specific problem.

There are many types of DNNs that could be used for classification. In this exploration, the goal was to assess and compare a Recurrent Neural Network (RNN) architecture to a Convolutional Neural Network (CNN) architecture for the task of classifying classical music compositions written by four composers: Bach, Beethoven, Chopin, and Mozart.

Each DNN has a unique architecture which allows them to approach the same problem with its own strategy. A RNN is oriented more towards understanding sequential relationships and temporal patterns, and in this case how those patterns can reveal a composer. (Hochreiter & Schmidhuber, 1997) A CNN is more useful for understanding spatial relationships such as in a computer vision model. As such, the CNN model extracts features on the topological pattern of soundscape and then learns to recognize when a composition "looks" like it was written by one of the composers. (Alzubaidi et al., 2021)

In both cases, the models are trained on samples of data generated from MIDI files of the music. A MIDI file is structured as a list of "events" or "messages" that instruct music playback software how to play a song (2020). For example, when to start a note (note-on), when to stop it (note-off), or how loud to play the note (velocity changes). These events do not contain audio; rather, they are instructions that a synthesizer or instrument can follow to create audio.

After being trained, the models were evaluated by having them attempt to classify samples that they had not previously encountered. Both models performed well at this task but there were clear trade-offs in terms of how the data had to be generated and sampled, as well as in the form of the models themselves.

Exploratory Data Analysis

An analysis of note counts per piece was conducted for four composers: Bach, Beethoven, Chopin, and Mozart. The dataset contained 1,628 MIDI files, with Bach contributing 1,024 files, Beethoven 212, Chopin 136, and Mozart 256.

As shown in Table 1, Bach's pieces were generally much shorter, with an average of M = 444.4 notes (Mdn = 143.0) and most containing fewer than 1,000 notes. One exceptionally long composition reached 17,389 notes. Beethoven and Chopin both had longer works on average (M

= 1530.4, Mdn = 1060.0; M = 1466.6, Mdn = 928.0, respectively), while Mozart's works fell between these ranges (M = 1181.7, Mdn = 625.5).

Table 1

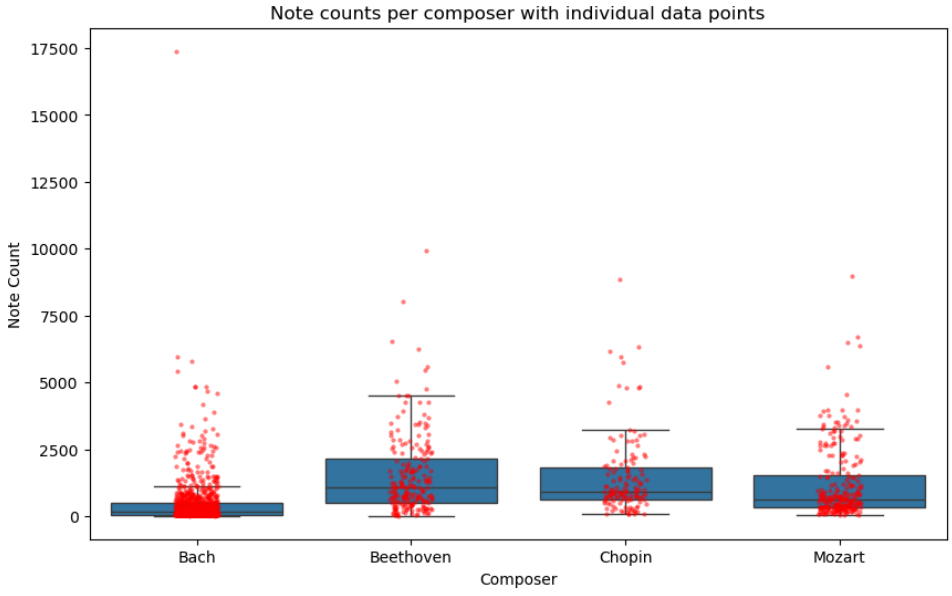
Data availability across composers

Composer	Files	Min Notes	Max Notes	Mean Notes	Median Notes
Bach	1,024	9	17,389	444.4	143.0
Beethoven	219	16	9,912	1,530.4	1,060.0
Chopin	136	61	8,865	1,466.6	928.0
Mozart	256	51	8,980	1,181.7	625.5

The boxplot with individual data points (Figure 1) clearly illustrates these differences: Bach's works cluster heavily at shorter lengths with a few extreme outliers, while the other composers' works show more variety.

Figure 1

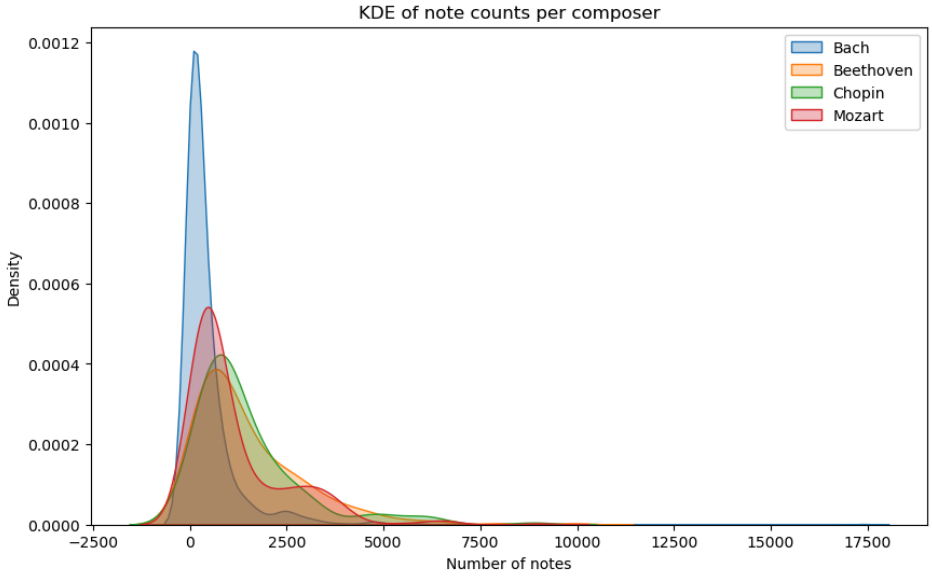
Distribution of Notes Counts by Composer



The KDE plot (Figure 2) further shows that Bach's note counts are tightly concentrated at the low end, whereas Beethoven, Chopin, and Mozart exhibit broader and more evenly spread distributions.

Figure 2

KDE plot of Note Count Distribution by Composer



Overall, these patterns illustrate both stylistic differences as well as their overall frequency in the dataset. Nonetheless, when analyzing the dataset in terms of the duration of music, there is sufficient data for training deep learning models after incorporating a sampling strategy.

Table 2

Total Music Data in Hours by Composer

Composer	Total Music Data (hours)
Bach	44.72
Beethoven	31.41
Chopin	8.36
Mozart	28.51

Note. Although Chopin is noticeably less prominent in the dataset, there is sufficient training data once a sampling strategy has been established given the over 8 hours of music.

Table 3

Summary of Composition Durations (seconds) by Composer

Composer	Count	Mean	Min	0.25	0.5	0.75	Max
Bach	1,024	157	18	43	77	191	5,209
Beethoven	219	516	22	226	415	676	5,032
Chopin	136	221	23	93	158	312	1,353
Mozart	256	401	26	207	348	530	1,478

Models

Convolutional Neural Network

Employing a Convolutional Neural Network (CNN) to solve the problem of predicting the composer of a digital audio file applies the notion of computer vision to a non-visual medium. Thus, leveraging this sort of methodology for training a neural network to perform this prediction first necessitates representing the audio as an image known as a spectrogram (Zeng et al., 2017).

Although a spectrogram as a general concept is a visual representation of audio, an engineer has the liberty of choosing how to generate this abstraction. This exercise follows two distinct processes for generating spectrograms and then provides a brief analysis of their strengths and modeling results.

The first method explored was a lightweight algorithm focused on the prominence of notes being played at a given time while the second, more complex method incorporated more information related to the timbre and frequencies of the instruments called a Mel spectrogram.

In both cases, a secondary goal was to maximize the predictive power of the model while minimizing the complexity of the inputs. The following process demonstrates the efficacy of training a deep CNN on 3 or 5 seconds of input data.

Preprocessing & Feature Extraction

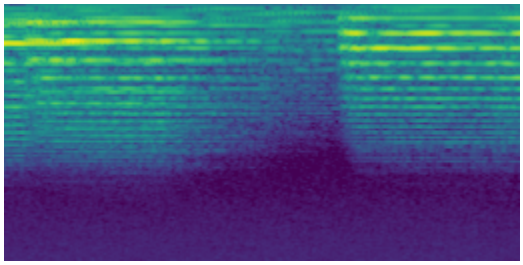
Two distinct processing pipelines were developed to evaluate the utility of multiple representations of the audio data. The first and simpler method was focused on establishing which notes were being played across multiple midi channels at a given tick and encoding that information in an array.

The second method leveraged third party audio decoding tools that converted the midi data into Mel spectrograms that captured more nuance related to the specific instruments being played at a given tick.

Figure 4

Visualizing Spectrogram Representation Strategies

Simple Array Representation



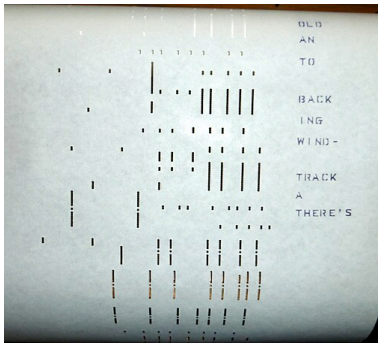
Mel Spectrogram

Note. The images above each represent three seconds from approximately the same portion of Mozart's

The **Simple Array Representation** was developed using only Numpy and was inspired by player piano rolls. The array is generated by reading midi messages sequentially across each channel and then indicating a 1 or a 0 for each tick in the midi stream where the note is active. Since each midi channel can play 128 notes, the resulting array is of the shape: 128 Notes x C channels x T ticks, where ticks are the base unit of time in the midi file.

Figure 5

Image of a Player Piano Roll



Note. The inspiration for the Simple Array Representation. Image courtesy of Wikimedia

In order to reduce dimensionality, elementwise addition was performed which collapses the channel dimension resulting in a final array of the shape 128 Notes x T ticks. In this final array, silent notes at a given tick have a value of 0 while active notes at a given tick will take the value of 1 x C active channels. Thus, this method provides a simple representation of which notes are played, held, and their concurrence over the orchestra through the course of the composition.

The **Mel Spectrogram** representations were developed using Librosa and Fluidsynth using the Fluid (R3) SoundFont. While the resulting images contain far greater detail than the Simple Array Representation, they also consumed far greater compute resources and considerably longer sampling times.

Sampling Strategies

Various sampling techniques were employed to develop training, testing, and validation datasets. In an effort to minimize size and dimensionality of the input data, as well as to maintain a balanced representation, the sampling algorithm randomly selected a window of seconds from random midi tracks in equal proportions, by composer. During this exploration, models were trained on either 3 or 5 second spectrograms for performance comparisons.

Heuristics were employed to prevent bad samples from being included in the datasets. Specifically in the Simple representation, any sample that had more than 20% silence was thrown out and a new random sample was drawn as an alternative.

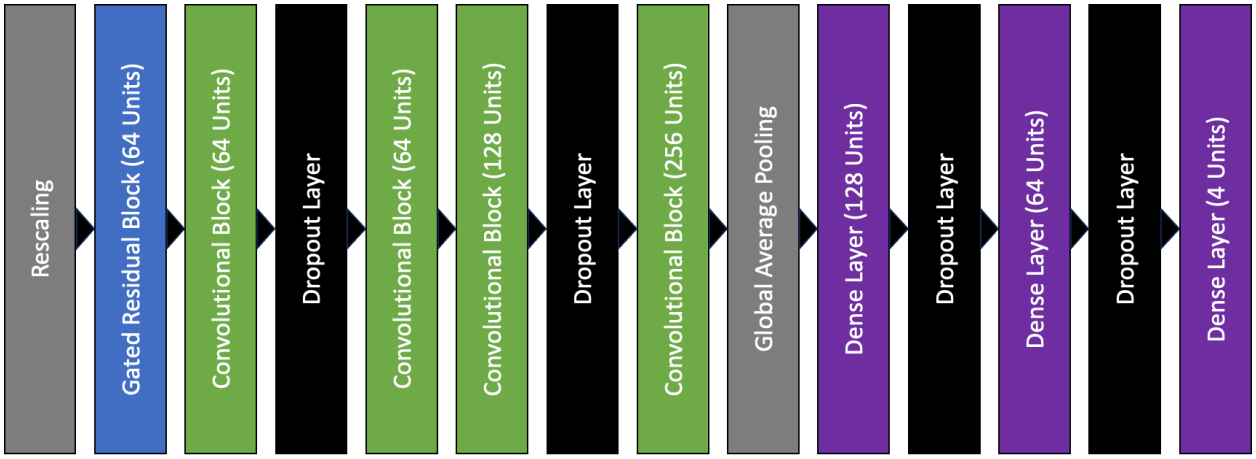
In addition to testing multiple sample lengths, models were also trained on datasets of various sizes including ~1000, ~2000, and ~4000 samples per each of the four composers in batches of 32 images.

Model Architecture

The most effective model architecture discovered in this exploration was a Deep CNN with a Gated Residual Block (Figure 6).

Figure 6

Deep Gated Residual CNN Architecture

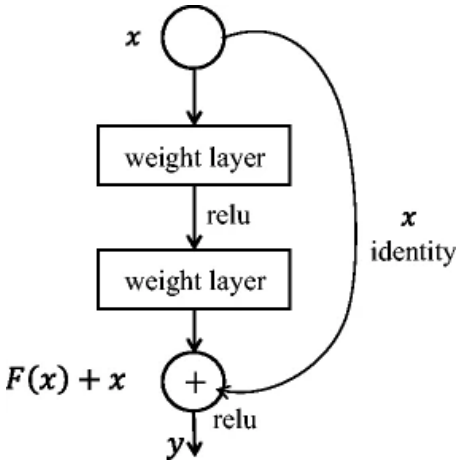


Note. The model above was leveraged to develop benchmarks for the CNN models in this study.

The Gated Residual Block followed the schema shown in Figure 7.

Figure 7

Residual Block Structure



Convolutional blocks contained Convolutional units with ReLu activations, followed by Batch Normalization, then a Max Pooling layer.

The Gated Residual block seemed to help decrease overfitting and allow the optimizer to better discover the optimal network weights. Dropout layers further helped to provide regularization in the fitting process.

Training Process

The training process incorporated an 80:10:10 split between training, testing, and validation datasets, respectively, regardless of the total size of the training dataset. The following analysis demonstrates the results of the various training sizes: 4,000 samples, 8,000 samples, and 16,000 samples for the Simple Array Representation spectrograms and 4,000 samples for the Mel spectrograms.

The model leveraged the Adam optimizer across up to 70 epochs, but with early stopping implemented after 30 epochs to prevent overfitting. The model also leveraged an adaptive learning rate such that the learning rate would be decreased for several rounds of training if the validation loss did not improve. The adaptive learning rate looked for an improvement factor of 0.5 and had a patience of 3 training rounds.

Results

Overall, the model was highly effective at distinguishing between classes with both representation styles. The best fitting model was the Simple Representation Array (SRA) model with 16,000, 5-second samples. Although this model had the lengthiest training times, the time spent developing the spectrogram samples was significantly shorter than the amount of time expended developing the 3-second Mel spectrograms. Therefore, it was still far more efficient to train the model on a greater volume of simpler data.

With 4,000 samples, the Mel spectrogram model performed better than the SRA model with a similar number of 5-second samples but not as well as the SRA with 8000 5-second samples. Although not an exact comparison, this evaluation provides evidence that the simpler representation can perform nearly as well with a similar volume of training data but with the benefit of significantly shorter sampling times.

Table 4

Evaluation Metrics by Representation Type and Sample Size

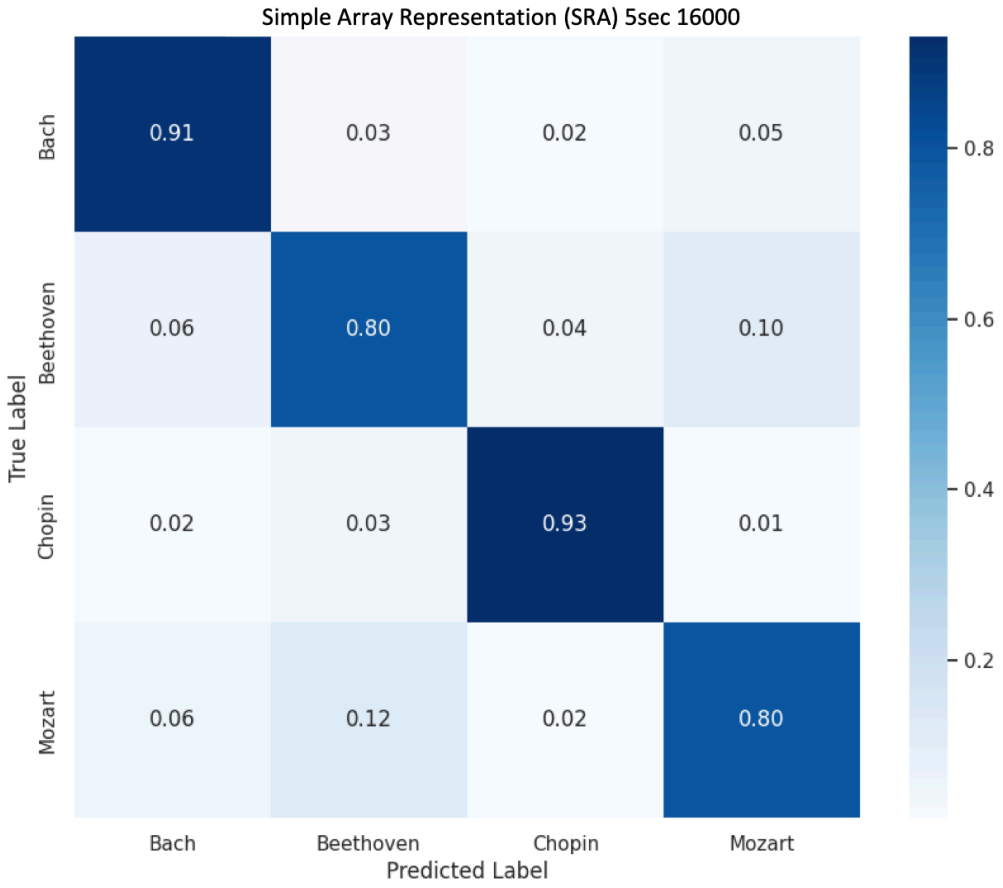
	Mel 4000 3sec	SRA 16000 5sec	SRA 8000 5sec	SRA 4000 5sec
Accuracy	0.7402	0.8578	0.7875	0.6693
Precision	0.7438	0.8571	0.7899	0.6786
Recall	0.7402	0.8578	0.7875	0.6693

F1	0.7302	0.8572	0.7875	0.6669
ROC-AUC	0.9286	0.9666	0.9394	0.8826

Note. The column headers indicate Representation Type Dataset Set, Seconds per Sample. Mel refers to Mel spectrogram and SRA refers to Simple Array Representation.

Figure 8

Confusion Matrix of SRA 5sec 16000 Model



Note. The high degree of correct classifications signify the high quality fit of the model.

Recurrent Neural Network

The Long Short-Term Memory (LSTM) training system is an effective way of predicting the composer, since it is designed to model the time-varying data. Whereas in the CNN approach, the audio would be converted to images and work with those images, this is done with the sequence of notes in a MIDI file and would retain the timing and order of the notes.

Each note has information such as pitch, length of time they should be played, the intensity with which they should be hit and the instrument playing them. These were arranged into a series of fixed-sized chunks to ensure that the model receives consistent input size, while preserving the flow of the music.

The wide variety of data presented a challenge as each piece had different lengths and active channels; selected chunks needed sufficient size to establish context without making the dataset excessively small. Each composer had varying total file counts. These problems were resolved by chunking the information, enforcing class balancing, and incorporating masking to enable the model to exclude silence.

Preprocessing & Feature Extraction For LSTM Model

The MIDI files presented a highly abstracted form of these classical compositions and it was not sufficient to merely ingest the information. In order to be useful, it was key to understand some basic principles of musical notation, the structure of the MIDI-type, and develop a method to encode this information in a useful way.

The Python library pretty_midi (Raffel & Ellis, 2014) was employed to convert raw MIDI messages into data objects. Musical pitch was prioritized by excluding drums:

```
if inst.is_drum: continue
```

Then for each non-percussion note,Table 5 outlines how we pulled raw features initially.

Table 5

Handling of Messages for Non-precussion (pitched) Instruments

Message Type	**Definition**
Start time (note_on)	When the note begins (in seconds)
End time (note_off or velocity = 0)	When the note ends (in seconds)
Pitch	The note's number (e.g. 60 = middle C)
Velocity	How hard a note is struck (attack)
Program number	the instrument type in the General MIDI standard(which makes 128 possible values in total, 0–127 range)

These details gave us a clean, structured way to represent each piece before doing feature engineering.

Noise Removal

Note durations were calculated by taking the difference between end and start times. Very short notes, less than 0.0001 seconds were treated as conversion artifacts and scrubbed from the dataset. The goal of this denoising step was to concentrate the focus of the dataset on musically meaningful events.

```
keep = dur > 1e-4
```

To continue feature engineering and preparation for model training, additional filtering was applied to remove uninformative chunks (e.g., those with very few notes) and refine the dataset further. These later steps are discussed in subsequent sections, as the cleaning, feature engineering, and preparation stages are interconnected rather than strictly sequential.

Feature Engineering

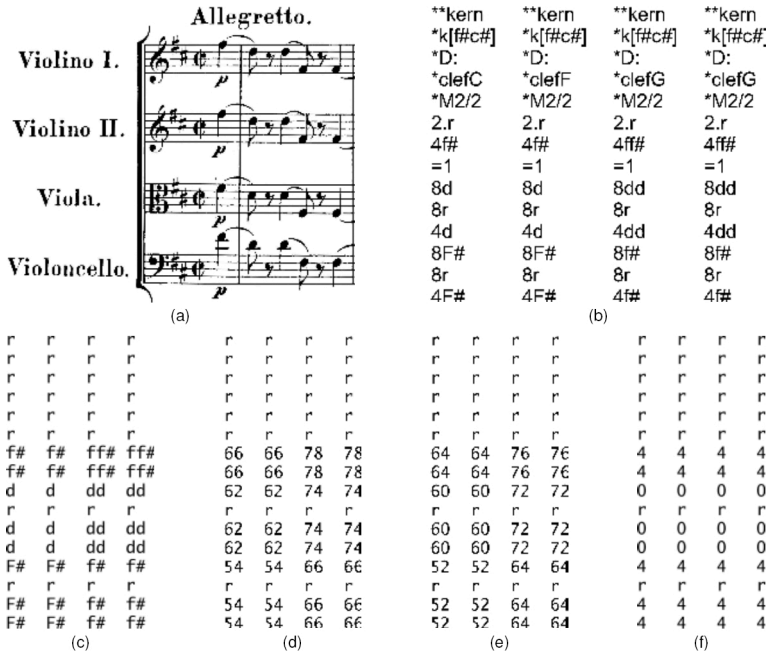
Having obtained our clean note-level dataset, the next step was to extract features that the machine learning model could learn from with significant value. Rather than relying on raw attributes such as absolute start times or pitches—which may introduce extraneous variation from key or tempo—prior studies have shown that features capturing relative musical relationships (e.g., pitch intervals, rhythmic ratios) provide more stylistically informative signals for composer classification.

For example, Gelbukh, Loya, Calvo, and Gómez-Adorno (2024) demonstrated that symbolic encodings emphasizing intervallic and rhythmic patterns improved classification accuracy, while Deepaisarn, Nakamura, and Morita (2023) applied NLP-based sequence models that abstract from absolute note values, focusing instead on relational structures within the music.

As shown in Figure 9 (Gelbukh et al., 2024) musical score can be converted from traditional notation into symbolic encodings (e.g., **kern format) and finally into numerical values such as pitch classes, intervals, and durations, which can then be used as machine learning features. This visual mapping from symbolic music to numerical representation reflects the same process we applied to our dataset.

Figure 9

Converting Traditional Notation into Symbolic Encodings



Note. From *Multi-Instrument Based N-Grams for Composer Classification Task* (Figure 1), by A. Gelbukh, D. A. Pérez Álvarez, O. Kolesnikova, L. Chanona-Hernández, & G. Sidorov, 2024, *Computación y Sistemas, 28*(1), 85–98, p. 87. https://doi.org/10.13053/CyS-28-1-4903. Licensed under CC BY-NC 4.0.

We used six final features for each note:

1. Time gap (dt) - A measure for the number of seconds since we last played a note derived using np.diff (starts, prepend = starts[0]). Log transformation was used to minimize extreme pauses, just as symbolic music modeling does in dealing with extreme variance in timing distributions.

2. Pitch interval (interval) - The melodic leap between one note to the next in semitones, trimmed to +/- 24 to keep down infrequent extreme jumps. Interval representations are generally used in symbolic composer classification because they maintain transposition-invariant patterns of stylistic characteristics (Deepaisarn et al., 2023).

3. Pitch class (pc) The pitch class represents the location of a note within the octave ignoring the octave number and focusing solely on the note name. This feature has been recommended in composer classification tasks because it reduces octave-specific noise

while preserving harmonic information. In our implementation, pitch class was extracted from the raw MIDI pitch values using:

```
pc = np.mod(pitch, 12)
```

This transformation converts each absolute pitch into its equivalent position within the chromatic scale, regardless of octave. Our experiments confirmed the value of this approach. When using raw pitch as a feature (dataset: balanced_chunks_seq70_7features.pkl), the model achieved a test accuracy of 0.7996 (loss = 0.5194).

While Bach and Chopin maintained relatively high scores, Beethoven and Mozart showed weaker performance, likely due to increased key-specific variance introduced by absolute pitch values. This supported the decision to replace raw pitch with pitch class in the final feature set, which improved consistency and reduced noise in classification.

4. Duration ratio (dur_ratio) – We calculated duration ratio as the note's length divided by the time gap to the next note:

```
dur_ratio = dur / time_to_next
```

This measure helps capture articulation styles—values near 1 reflect smoother legato playing, while smaller values suggest more detached staccato patterns. Before log transforming, distributions were not normal and extreme numbers showed that there were long holds and followed by silence.

To handle this, a smoothing transformation was applied:

```
dur_ratio = np.log1p(dur / time_to_next)
```

This technique compresses outliers while retaining relative differences. Support for the relevance of such timing ratio features comes from performance modeling research, where ratios of onset-to-onset timing have been successfully used to distinguish stylistic traits in performers (e.g., Mahmud Rafee, Fazekas, & Wiggins, 2021).

5. Velocity as a raw feature – The dynamic level of the note, taken directly from the MIDI velocity field (1–127). Dynamics are known to reflect expressive performance styles and have been incorporated in prior composer identification systems (Raffel & Ellis, 2014).

6. Program number (program another raw feature from MIDI library) – The General MIDI instrument program (0–127). Including this information gives the model a sense of timbral context without requiring audio, which can be useful in modeling stylistic instrumentation choices (Raffel & Ellis, 2014).

Sampling Strategies

The dataset was imbalanced — Bach and Chopin had more varied training examples, while Beethoven and Mozart had fewer or less diverse ones. As a result, the model often confused Beethoven and Mozart with other composers. This problem was addressed with two strategies:

1. Targeted Data Augmentation

Certain training chunks were transposed up or down by a few semitones (±2 or ±4). This preserves the melodic pattern but alters the key, giving the model more varied examples without changing the relative musical structure. Only Beethoven and Mozart pieces received augmentation as they had less variety. Small shifts prevent unrealistic note ranges, and using both upward and downward shifts avoids bias toward higher or lower registers.

2. Class Balancing via Downsampling

All composers' chunks were downsampled to match the smallest class size. This ensured that the model was trained on an equal number of examples per composer, and eliminated class bias. Chunking rules were carefully designed to maximize useful structure while avoiding redundancy. Sequence lengths of 70 notes ensured that chunks overlapped enough to uncover important motifs without being overly repetitive. Minimum real notes per chunk (= 50) implies that fewer than 50 non-padded notes were discarded to avoid learning from mostly empty data. Enforcing a maximum chunk size per piece (= 20) prevented a single long work from dominating the dataset and skewing the training distribution. Before balancing, the model performed unevenly.

Model Architecture

Inputs and Representations

After balancing, each composer class contributed the same number of training sequences: Bach = 2,538, Beethoven = 2,538, Chopin = 2,538, and Mozart = 2,538. Each sequence consisted of 70 notes and included two parallel feature sets:

- **Note features** (X_notes): Five per-note continuous or ordinal variables—dt, interval, pitch class (pc), dur_ratio, and velocity—shaped (N, 70, 5).
- **Instrument programs** (X_prog): One General MIDI program ID per note, as integers in the range [0, 127], shaped (N, 70).

Program IDs were treated as categorical and embedded rather than scaled. An embedding layer with input_dim = 128 and output_dim = 10 produced a tensor of shape (N, 70, 10). This embedding was concatenated with the note features along the last axis, resulting in a fused sequence of shape (N, 70, 15).

Normalization

In order to stabilize the training of the models, only z-score normalization of continuous variables (dt, interval, dur_ratio and velocity) was done. This was fitted on the training set and applied to the test set. Pitch class (pc) was not treated as scaling since it is cyclic (0 11) and scaling would basically skew its natural circle-of-fifths connections— a way musicians arrange the 12 pitch classes (C, C♯/D♭, D, etc.). Scaling of program ID was also avoided, they were passed through the embedding layer.

Sequence Encoder

The concatenated sequence was processed through the following layers: masking (mask_value = 0.0), a bidirectional LSTM (128 units per direction, return_sequences = True) to capture forward and backward temporal context, dropout (0.25) for regularization, and a second LSTM (96 units) to compress the sequence into a single vector. The 96-unit size reflects a trade-off in architecture design: while many recurrent networks follow a halving pattern (e.g., 128 → 64), our experiments showed that using 96 units yielded a small but repeatable improvement in validation accuracy. The output then passed through a dense layer (64 units, ReLU activation) with dropout (0.25), followed by a softmax layer over the four composer classes.

Training Process

The dataset was split into training and test sets using an 80/20 stratified split to preserve class balance, with labels encoded as integers. Feature scaling was performed only on continuous variables, as detailed in the Normalization subsection of the Model Architecture section.

The model was trained with the Adam optimizer and sparse categorical cross-entropy loss, using accuracy as the primary evaluation metric. The batch size was set to 64 and training was run over up to 100 epochs though early termination (EarlyStopping (patience = 10, restore_best_weights = True)) stopped after the validation loss had not improved in a given number (patience) of epochs to avoid overfitting and complex computing.

Other callbacks were ModelCheckpoint to store the best model and ReduceLROnPlateau to reduce the learning rate when model validation performance ceased to improve in order to aid in improved convergence. Major layers were followed by dropout layers (rate = 0.25) as a way to enhance generalization.

The choice of these strategies aimed at achieving efficiency in training while maintaining robustness in the model to make sure that the network should be converging successfully without overfitting to the training set.

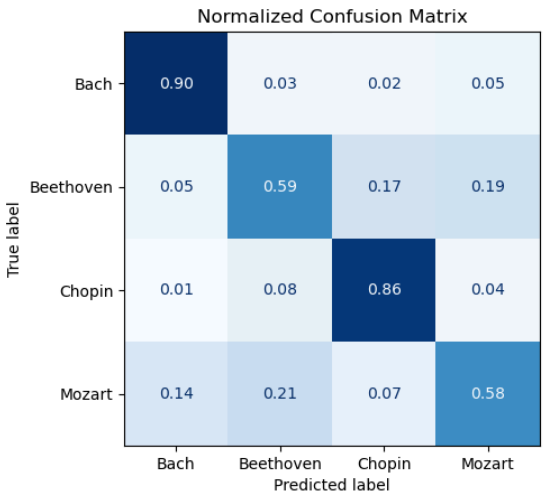
Results

The confusion matrix in Figure 10 shows this imbalance — Beethoven and Mozart had high confusion rates, with up to 20% misclassification into other composers.

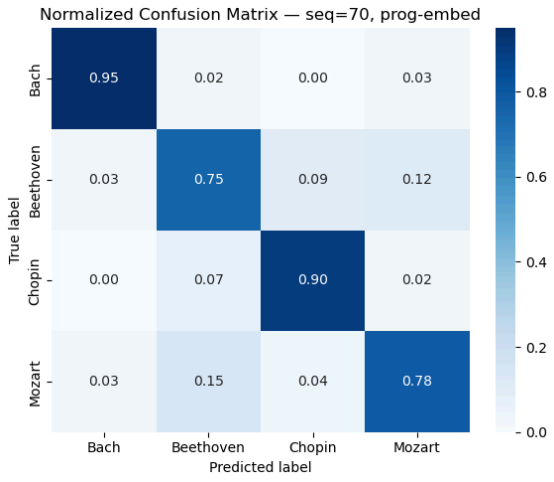
Figure 10

Results of RNN Before and After Rebalancing Dataset

Before Rebalancing



After Rebalancing



Note. After augmentation and balancing, the results improved substantially.

Table 6

Comparison Results of RNN Before and After Rebalancing Dataset

	Before Rebalancing	**After Rebalancing**
Bach	0.82 precision / 0.90 recall	0.94 precision / 0.95 recall
Beethoven	0.64 precision / 0.59 recall	0.75 precision / 0.75 recall
Chopin	0.77 precision / 0.86 recall	0.87 precision / 0.90 recall
Mozart	0.67 precision / 0.58 recall	0.82 precision / 0.78 recall
Overall	Overall accuracy: ~0.73	Overall accuracy: 0.85

Note. Beethoven misclassified as Chopin dropped from 17% → 9%. Mozart misclassified as Beethoven dropped from 21% → 15%.

These results show that while augmentation slightly helped, handling class imbalance and carefully controlling chunk parameters produced the biggest jump in performance, raising accuracy from ~0.73 to 0.85.

Conclusions

The exercise demonstrated above all, the flexibility of deep learning frameworks for uncovering meaningful patterns in data while employing a variety of methods for modeling complex systems. Although the RNN and the CNN frameworks each approach the classification problem with a fundamentally different methodology, they both achieved high levels of accuracy in the composer classification task.

When analyzing the results data, it is clear that both of the selected models could effectively distinguish between classes. Both the selected RNN and CNN models achieved a classification accuracy of 85% on a balanced dataset. This is significantly better than a random choice algorithm which would yield 25% accuracy, on average.

The CNN, trained on spectrograms, benefitted from efficient feature extraction especially when leveraging the lighter weight Simple Array Representation (SRA) encoding. This method proved to be highly scalable and allowed for training on much larger datasets. While the Mel spectrograms also suited the task and had good results with less data. The Mel spectrogram dataset was a quarter of the samples of the SRA dataset but required many hours longer to generate.

Conversely, the RNN which leveraged the input values more directly, demanded a far more intensive preprocessing pipeline in order to capture sufficient musical information. The model could then learn the temporal relationships between relative pitch, rhythms, and other dynamics inherent in the musical composition.

Both models had the most difficulty distinguishing between Mozart and Beethoven. This may be because, of the four composers, their lives had the most overlap. Some music critics have made note of the influence of Mozart on Beethoven. (2023, December 9) Although subjective, it seems reasonable to infer that Mozart and Beethoven had the most stylistic overlap. By extension, one might introduce an input sample from a composer unknown to the model and understand the output as the models' opinion on which composer they are most like of the four in the set.

More generally, these results imply that this multi-class classification problem can be approached from a variety of perspectives and benefit from a hybridized approach that combines aspects of both models. Further work might lead to a combination of these methods or test the benefits of transfer learning by way of models trained on large musical corpora.

References

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of Deep Learning: Concepts, CNN Architectures, challenges, applications, Future Directions. *Journal of Big Data*, *8*(1). https://doi.org/10.1186/s40537-021-00444-8

Deepaisarn, S., Nakamura, E., & Morita, H. (2023). *NLP-based music processing for composer classification.* Scientific Reports, 13, 13736. https://doi.org/10.1038/s41598-023-40332-0

Gelbukh, A., Pérez Álvarez, D. A., Kolesnikova, O., Chanona-Hernández, L., & Sidorov, G. (2024). *Multi-instrument based n-grams for composer classification task*. *Computación y Sistemas, 28*(1), 85–98. https://doi.org/10.13053/CyS-28-1-4903

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

MIDI 1.0 detailed specification: Document version 4.2.1. (2020). . The MIDI Manufacturers Association.

Music History and Facts·2 min read, Facts, T. H. and, FactsTrivia, M. H. and, & Facts, M. H. and. (2023, December 9). *Mozart vs. Beethoven: The intriguing similarities & legacies of classical greats*. mordents.com. https://mordents.com/similarities-between-mozart-and-beethoven

Raffel, C., & Ellis, D. P. W. (2014). Intuitive analysis, creation and manipulation of MIDI data with pretty_midi. *In Proceedings of the 15th International Society for Music Information Retrieval Conference Late-Breaking and Demo Papers.* International Society for Music Information Retrieval. https://github.com/craffel/pretty-midi

Wikimedia Foundation. (2025, August 3). *Piano Roll*. Wikipedia. https://en.wikipedia.org/wiki/Piano_roll

Zeng, Y., Mao, H., Peng, D., & Yi, Z. (2017). Spectrogram based multi-task audio classification. *Multimedia Tools and Applications*, *78*(3), 3705–3722. https://doi.org/10.1007/s11042-017-5539-3