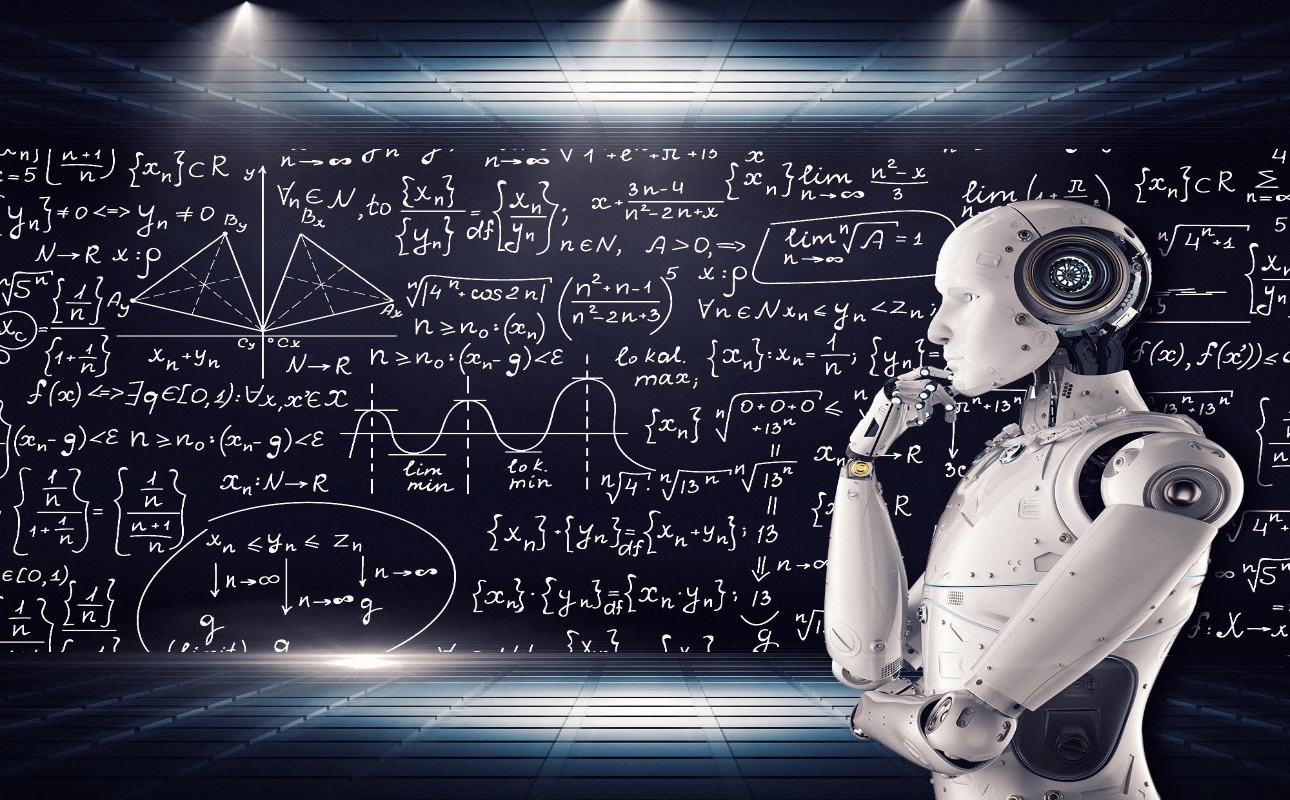
## P1#yIS1



AAI-511: [Music Genre and Composer Classification Using Deep Learning](https://ole.sandiego.edu/webapps/assignment/uploadAssignment?content_id=_3097238_1&course_id=_111306_1&group_id=&mode=view)

July 2023

Group 8: Christopher Teli, Adam Graves, Ikenna Opurum

Github Link to Model: <https://github.com/cteliStolenFocus/aai-511-team-8/tree/main>

Table of Contents

[P1#yIS1 0](#_Toc142734043)

[Abstract 3](#_Toc142734044)

[Goals/Strategies 4](#_Toc142734045)

[Data Processing 7](#_Toc142734046)

[Extracting Musical Features from MIDI Files 7](#_Toc142734047)

[Processing Composer Data into a DataFrame 7](#_Toc142734048)

[Data Fields of the generated dataset 8](#_Toc142734049)

[Spectrograms Generation from Composer Midi files for CNN model 9](#_Toc142734050)

[Phase 1: MIDI to WAV Conversion 9](#_Toc142734051)

[Phase 2: Spectrogram Image Creation 9](#_Toc142734052)

[Model Architecture 11](#_Toc142734053)

[LSTM Models 11](#_Toc142734054)

[Model 1 11](#_Toc142734055)

[Model 2 11](#_Toc142734056)

[Model 3 12](#_Toc142734057)

[Convolutional Neural Network (CNN) Model for Composer Detection Using LSTM Data 12](#_Toc142734058)

[Model Architecture 13](#_Toc142734059)

[Convolutional Neural Network (CNN) Model for Composer Detection 14](#_Toc142734060)

[Model Architecture 15](#_Toc142734061)

[Results Analysis 17](#_Toc142734062)

[Summary 17](#_Toc142734063)

[Appendix A: 17](#_Toc142734064)

[Code Implementation Details 17](#_Toc142734065)

[LOADING THE DATASET 17](#_Toc142734066)

[**1.3.1 The process:** 18](#_Toc142734067)

[**1.4. INITIAL DATA VERIFICATION AND NORMALIZATION** 18](#_Toc142734068)

[**1.5. OPTIMIZING THE DATASET** 19](#_Toc142734069)

[**1.6. DATA CLEANING** 19](#_Toc142734070)

[**2.** **BUILDING THE LSTM MODEL (Long Short-Term Memory)** 20](#_Toc142734071)

[**2.1.** **LSTM Model Version 2** 22](#_Toc142734072)

[**2.2.** **LSTM Model Version 3** 24](#_Toc142734073)

[**3.** **CNN MODEL (Convolutional Neural Network)** 25](#_Toc142734074)

# Abstract

In computational musicology there is often a demand to answer conceptually a simple questions like ” Who composed this piece?” As an example, melodic lines, rhythmic pattern, chords and chord progressions, tonality, and cadenzas are used.

This project focuses on the analysis of MIDI music data and the development of machine learning models for composer prediction based on the extracted features from MIDI files. MIDI (Musical Instrument Digital Interface) files encode musical information, making them suitable for exploring musical patterns and characteristics. The primary objective of this project is to investigate the potential of utilizing MIDI data for composer prediction and to compare the performance of different machine learning algorithms.

The composers are:

**Composer #of records**

bach 42

bartok 41

byrd 42

chopin 41

handel 41

hummel 42

mendelssohn 41

mozart 41

schumann 38

MIDI files serve as digital representations of musical compositions, storing information about pitch, timing, and other musical attributes. This project explores the application of machine learning techniques to analyze MIDI data and predict the composer of a piece based on extracted features.

With a limited amount of records we are expected to have reduced scores of prediction success.

“In the past few years, several open-source libraries such as Keras, PyTorch Lightning, Hugging Face Transformers, and Ray Train have been attempting to make DL training more accessible, notably by reducing code verbosity, thereby simplifying how neural networks are programmed. Most of those libraries have focused on developer experience and code compactness.” (AWS Machine Learning Blog, taken from: <https://aws.amazon.com/blogs/machine-learning/reduce-deep-learning-training-time-and-cost-with-mosaicml-composer-on-aws/>)

The Keras libraries. (TensorFlow) is the chosen library to use in this code.

# Goals/Strategies

The primary objective of this project is to develop a deep learning model that can predict the composer of a given musical score accurately.

The project aims to accomplish this objective by using two deep learning techniques: Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN).

The strategy includes:

1. Feature Engineering of the Dataset

* Preprocess and normalize the input data, such as image or audio files.
* Extract relevant features that are necessary for the training of both LSTM and CNN models.
* If handling MIDI files, convert them into an appropriate format, like WAV, for further processing.

1. Determination of the Final Dataset

* Select and organize the final set of features for extraction.
* Load the processed data into a suitable structure, such as a dataframe, for efficient handling.

1. Building an LSTM Model

* Research and identify the best architecture for the LSTM model.
* Select the most suitable optimizer and evaluation metrics for the model.
* Train the LSTM model with the processed data, tuning hyperparameters as needed.

1. Building a CNN Model for Character-Based Data

* Design and construct a Convolutional Neural Network that is suitable for character-based data.
* Optimize the architecture to achieve the desired performance.

1. Generating a Training Dataset from Composer MIDI Files by Converting to Spectrogram Images

* Convert MIDI files into WAV format.
* Transform the WAV files into PNG graphical files, preserving relevant information for the task.
* Prepare the PNG dataset for training with a CNN.

1. Building a CNN Model for the composer spectrogram Graphical-Based Dataset

* Design a CNN architecture that is tailored to the graphical representation of the data (PNG files).
* Train the CNN model with the PNG dataset, adjusting the structure and parameters as necessary.

1. Evaluation and Comparison

* Evaluate the performance of the LSTM and CNN models using suitable metrics.
* Compare the results and identify the strengths and weaknesses of each approach.
* Determine the best model(s) based on the project's specific requirements and goals.

The final outcome of the project will be the development and evaluation of three distinct models:

1. MODEL-1: Utilizing Long Short-Term Memory (LSTM) Networks for Composer Detection
2. MODEL-2: Implementing a Convolutional Neural Network (CNN) with Time Series Data for Composer Detection
3. MODEL-3: Developing a CNN Based on Spectrograms for Composer Detection from Musical Recordings

The final analysis will reveal the most suitable deep learning model for composer detection within this specific dataset.

# Data Processing

## Extracting Musical Features from MIDI Files

The function calculate\_features takes a MIDI file as input and computes various musical characteristics, such as pitch, note density, volume, rhythmic complexity, and tempo.

* Pitch: The average pitch of the notes playing at a given time.
* Note Density: The number of notes per second.
* Volume: The average volume of the notes playing at a given time.
* Rhythmic Complexity: The variance in the intervals between note onsets.
* Tempo: The musical tempo at each point in time, interpolated from tempo changes within the MIDI file.

These features are computed at one-second intervals over the duration of the MIDI file, resulting in time series data for each feature.

## Processing Composer Data into a DataFrame

The function process\_composer\_data iterates over a set of composer directories, each containing MIDI files for a particular composer. It leverages the calculate\_features function to compute the aforementioned features for each MIDI file, appending the results to a DataFrame along with the corresponding composer's name.

Composer: The name of the composer.

Times: The timestamps for the extracted features.

Pitch: Time series data for pitch.

Note Density: Time series data for note density.

Volume: Time series data for volume.

Rhythmic Complexity: A single value representing the rhythmic complexity.

Tempo: Time series data for tempo.

Once all the files have been processed, the DataFrame is serialized into a pickle file. This file serves as a compact and efficient way to share the preprocessed data across different parts of the project or with other team members.

The ETL process embodied in this code not only facilitates the creation of a structured dataset tailored for training deep learning models but also promotes collaboration by allowing for the seamless sharing of preprocessed musical features extracted from the raw MIDI files of different composers. It decouples the often computationally expensive feature extraction step from model building, thus enhancing the overall efficiency and flexibility of the modeling process."

## Data Fields of the generated dataset

Columns **=** ["Composer","Times",

"Pitch",

"Note\_Density",

"Volume",

"Rhythmic\_Complexity",

"Tempo" ]

## Spectrograms Generation from Composer Midi files for CNN model

The process of generating spectrograms from composer MIDI files for the Convolutional Neural Network (CNN) model is a critical step in feature extraction for our project. Spectrograms are visual representations of the frequency spectrum of audio signals, which contain valuable information about the characteristics of sound. In our context, these spectrograms serve as the primary input data for the CNN model to detect different composers. The entire procedure can be divided into two main phases: MIDI to WAV conversion and spectrogram image creation.

### Phase 1: MIDI to WAV Conversion

1. MIDI to WAV Transformation: Utilizing FluidSynth and sound font files, the MIDI files corresponding to different composers' works were converted into WAV audio format. The choice of WAV files stems from their lossless nature, preserving all the musical information without compression.
2. Automating the Conversion: To efficiently handle multiple files across various directories, a shell script was created to iterate through the directories and perform the conversion. This streamlined the process, allowing for batch processing of files.

### Phase 2: Spectrogram Image Creation

1. Loading WAV Files: Using Librosa, a popular library for analyzing and processing audio files, the WAV files were loaded into the Python environment. Librosa's load function was employed to read the audio data.
2. Short Time Fourier Transform (STFT): The Short Time Fourier Transform was applied to the audio data using Librosa's stft function. This method breaks down the audio signal into small windows and computes the Fourier Transform for each, allowing for the frequency analysis of the signal.
3. Amplitude and Image Resizing: The amplitude of the STFT was extracted and resized to a consistent 224x224 dimension using SciPy's ndimage module. This ensured uniform input size for the CNN model.
4. Spectrogram Visualization: Librosa's specshow function was used to visualize the resized amplitude as a spectrogram. The visualization parameters included both time and logarithmically-scaled frequency axes.
5. Image Saving: Finally, the spectrogram images were saved as PNG files in corresponding directories using Matplotlib's savefig function. PNG was chosen as the format for its lossless compression and wide support in image processing.

A blue and orange grid

Description automatically generated

The conversion from MIDI files to spectrogram images was a critical preprocessing step for our project. It transformed complex musical information into a visual form that could be directly fed into the CNN model. By automating the process through scripting and leveraging specialized libraries like Librosa, this phase successfully prepared the data for machine learning analysis, focusing on the unique attributes of each composer's works. The resulting spectrograms provide a rich and consistent dataset that plays a vital role in the subsequent modeling and classification tasks.

# Model Architecture

## LSTM Models

Three LSTM Models were evaluated.

### Model 1

Architecture:

* LSTM layer with 50 units, ReLU activation function.
* Dense layer with num\_classes units, softmax activation function.

Input Shape: (num\_steps, num\_features).

Optimizer: Adam.

Special Features: A simple, single LSTM layer model.

### Model 2

Architecture:

LSTM layer with 128 units, ReLU activation function, returns sequences.

LSTM layer with 64 units, ReLU activation function, returns sequences.

LSTM layer with 32 units, ReLU activation function.

Dense layer with num\_classes units, softmax activation function.

Input Shape: (num\_steps, num\_features).

Optimizer: Adam.

Special Features: A deeper model with 3 LSTM layers.

### Model 3

Architecture: Same as Model 2.

Input Shape: (num\_steps, num\_features).

Optimizer: Adamax.

Special Features: 3 LSTM layers, similar to Model 2, but specifically using the Adamax optimizer and Early Stopping.

## Convolutional Neural Network (CNN) Model for Composer Detection Using LSTM Data

The task of detecting composers is accomplished using a Convolutional Neural Network (CNN) designed to recognize patterns in the data processed by Long Short-Term Memory (LSTM) networks. The combination of LSTM and CNN architectures allows the model to capture both sequential dependencies and local patterns in the musical data, making it a powerful tool for this specific application.

### Model Architecture

Input Layer: The input to the model consists of sequences generated by previous LSTM processing. The shape of this input (num\_steps, X\_train.shape[2]) reflects the temporal structure and feature dimensions of the LSTM-processed data.

Convolutional Layer 1: The first convolutional layer, containing 64 filters of size 3 and using the ReLU activation function, is designed to detect local patterns within the temporal sequence. These filters work on segments of the LSTM-processed data to identify relevant local features.

Max Pooling Layer 1: The max pooling layer of size 2 helps reduce the dimensionality of the data, preserving the most salient features. It aids in reducing computation and focuses on the dominant patterns within the local segments.

Convolutional Layer 2: A second convolutional layer with 128 filters of size 3 continues to build on the extracted local patterns, further refining the features that characterize the composers.

Max Pooling Layer 2: Another max pooling layer further compresses the spatial representation, emphasizing the most significant local features.

Flatten Layer: The flatten layer transforms the spatially structured data into a flat vector, preparing it for the dense layers. It maintains all the spatial relationships identified by the previous layers.

Dense Layer: A dense layer with 128 neurons and ReLU activation builds higher-level abstractions from the flattened features. It integrates the local patterns into a global understanding of the data.

Dropout Layer: A dropout layer with a rate of 0.3 is used to mitigate overfitting, ensuring that the model generalizes well to unseen data.

Output Layer: The final dense layer maps the integrated features to the classes representing different composers, using a softmax activation to provide probability scores for each class.

Model Compilation and Summary: The model is compiled with the Adam optimizer and uses categorical cross-entropy loss, reflecting the multi-class nature of the classification task. Accuracy is chosen as the evaluation metric.

The integration of LSTM-processed data with a CNN model presents a novel approach to composer detection. By leveraging the sequential understanding provided by the LSTM and the pattern recognition capabilities of the CNN, the model offers a nuanced and robust means of analyzing complex musical data. The architecture is tailored to exploit both the temporal and spatial dimensions of the data, offering a sophisticated solution to a challenging problem.

### Convolutional Neural Network (CNN) Model for Composer Detection

The Convolutional Neural Network (CNN) designed to recognize patterns in the spectrogram images generated from the composer's MIDI files. CNNs are particularly well-suited for image classification tasks, as they can learn spatial hierarchies of features directly from the data. The architecture of the CNN model for our project is detailed below:

### Model Architecture

Input Layer: The input to the model consists of a sequence of spectrogram images, each represented as a fixed-size matrix. The input\_shape argument defines the dimensions of these matrices, corresponding to the number of time steps (num\_steps) and the number of features in the training set (X\_train.shape[2]).

Convolutional Layer 1: The first convolutional layer consists of 64 filters, each of size 3. These filters slide over the input data to detect local patterns, such as edges and textures. The activation function used is the Rectified Linear Unit (ReLU), a popular choice for introducing non-linearity into the model.

Max Pooling Layer 1: Following the first convolutional layer is a max pooling layer with a pool size of 2. Max pooling helps in reducing the spatial dimensions of the input, retaining only the most important information and thereby reducing computation.

Convolutional Layer 2: The second convolutional layer has 128 filters, each of size 3, and also uses the ReLU activation function. This layer continues the process of feature extraction, learning more complex and abstract patterns.

Max Pooling Layer 2: A second max pooling layer further reduces the spatial dimensions and emphasizes the dominant features.

Flatten Layer: The flatten layer reshapes the pooled feature maps into a single continuous vector, making it suitable for input to the dense layers.

Dense Layer: A fully connected dense layer with 128 neurons and ReLU activation is used to perform higher-level reasoning on the extracted features.

Dropout Layer: To prevent overfitting, a dropout layer is introduced with a rate of 0.3. This layer randomly sets a fraction of the input units to 0 during training, which helps in achieving a more robust model.

Output Layer: The final layer is a dense layer with as many neurons as there are classes (num\_classes, representing different composers). The softmax activation function is used to transform the raw output into probabilities, indicating the likelihood of each class.

# Results Analysis

# Summary

# Appendix A:

### Code Implementation Details

#### IMPORTING LIBRARIES

Importing the necessary libraries to read the dataset: Most are standard to the work done in this class regarding deep machine learning. The new ones that are loading deal with the MIDI file formatting: they are:

**import** os

**import** glob

**import** pretty\_midi

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

*# Ignore warnings*

**import** warnings

warnings**.**filterwarnings('ignore')

**from** sklearn.metrics **import** accuracy\_score, precision\_score, recall\_score, f1\_score

**import** tensorflow **as** tf

**from** tensorflow.keras **import** layers, models, optimizers

**from** tensorflow.keras.preprocessing.image **import** ImageDataGenerator

**from** sklearn.metrics **import** classification\_report

**from** keras **import** models, layers

*# Check if pickle file exists and use the file for dataset*

**import** pickle

### LOADING THE DATASET

Load the pickle file:

* 'team8\_composer\_dataset.pkl'

## **1.3.1 The process:**

A few steps in creating the pickle data file for import of the MIDI data.

Feature Extraction: First, extract the relevant features from each MIDI file. Using the pretty\_midi library to parse the MIDI files and extract information like times, pitch, note density, volume, etc. Create a function to perform this feature extraction for each MIDI file.

There are 9 composers, each with 40-42 files.

## **1.4. INITIAL DATA VERIFICATION AND NORMALIZATION**

Review data field attributes and data types, ensure proper import and verify the field types, check the dataframe df as in 2.4

Code sample: 2.4

*#@title 2.4: Data Pre-processing - verifications*

print(df**.**head())

print(df**.**info())

Verify the list of Composers and record count as in 2.6

Code sample 2.6:

*#@title 2.6: Check composer names and Count the occurrences of each 'quality' value*

composer\_counts **=** df['Composer']**.**value\_counts()

*# Sort the count by 'quality' values*

sorted\_composer\_counts **=** composer\_counts**.**sort\_index()

*# Print the count for each 'quality' value*

print(sorted\_composer\_counts)

Confirming the results:

bach 42

bartok 41

byrd 42

chopin 41

handel 41

hummel 42

mendelssohn 41

mozart 41

schumann 38

## **1.5. OPTIMIZING THE DATASET**

Normalization of the data by removing unnecessary columns and replacing missing values.

## **1.6. DATA CLEANING**

Data cleaning and normalization by removing unnecessary columns and replacing missing values. Check for an NaN values in the data fields, or other values that can be an obstacle in the building of a DL model.

**Summary**:

* The code defines a function called “calculate\_features(midi\_file)” that loads a MIDI file, extracts various musical features like pitch, note density, volume, rhythmic complexity, and tempo. The extracted features are returned as numpy arrays.
* The code defines another function called “process\_composer\_data()” that iterates over directories containing MIDI files for different composers. It uses the “calculate\_features()” function to extract features from each MIDI file and appends the data into a pandas DataFrame (df)
* The DataFrame (df) is then saved to a pickle file for future use. If the pickle file already exists, the code loads the data from the pickle file instead of reprocessing the MIDI files.
* The code checks if the pickle file ('team8\_composer\_dataset.pkl')

exists before processing the data. If the pickle file exists, it loads the data from the file, and if not, it calls the process\_composer\_data() function to create the dataset.

## **BUILDING THE LSTM MODEL (Long Short-Term Memory)**

The project will evaluate a few versions of the DL models. This will determine the best tuning of the model to fit this project.

Starting with version 1:

Load of the related libraries, which are:

* from tensorflow.keras.models import Sequential
* from tensorflow.keras.layers import LSTM, Dropout, Dense

The code we are providing is for creating a sequential neural network model using Keras, a high-level deep learning library.

Definig the LSTM as:

LSTM(50, activation='relu', #utilizing the ReLU activation function

input\_shape=(num\_steps, num\_features)): This line adds an LSTM (Long Short-Term Memory) layer to the model. LSTM is a type of recurrent neural network (RNN) that is particularly well-suited for sequence data. Here, 50 is the number of LSTM units or cells in the layer, activation='relu' specifies the activation function as Rectified Linear Unit (ReLU), and input\_shape=(num\_steps, num\_features) defines the shape of the input data. num\_steps represents the number of time steps in the sequence, and num\_features represents the number of features at each time step.

Dense (num\_classes, activation='softmax'): This line adds a dense (fully connected) layer to the model. The Dense layer is typically used for the final layer of a classification model. num\_classes represents the number of classes in your classification problem, and activation='softmax' applies the softmax activation function to the output of this layer, which converts the output values into probabilities for each class.

Using optimizer = Adam

Item 7: LSTM Model

*# Build the LSTM model*

model **=** Sequential([

LSTM(128, input\_shape**=**(num\_steps, X\_train**.**shape[2]), return\_sequences**=True**),

Dropout(0.2),

LSTM(64),

Dropout(0.2),

Dense(num\_classes, activation**=**'softmax')

The results:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

lstm (LSTM) (None, 50) 11200

dense (Dense) (None, 9) 459

Total params: 11,659

Trainable params: 11,659

Non-trainable params: 0

In the data there are under 11,700 parameters which is not a large dataset.

Model evaluation:

Breakdown Score Results for the Version 1 of the LSTM Model:

Classification Report - LSTM Model:

precision recall f1-score support

bach 0.62 0.89 0.73 9

bartok 0.20 0.12 0.15 8

byrd 0.71 0.62 0.67 8

chopin 0.47 0.88 0.61 8

handel 0.43 0.38 0.40 8

hummel 0.67 0.44 0.53 9

mendelssohn 0.00 0.00 0.00 8

mozart 0.43 0.38 0.40 8

schumann 0.38 0.38 0.38 8

accuracy 0.46 74

macro avg 0.43 0.45 0.43 74

weighted avg 0.44 0.46 0.43 74

Top performer is the composer Bach with a f1 score of: 0.73, and a poor performance of composer Mendelssohn with a f1 score of: 0.

Overall f1 score of: 0.46 being at 46%.

## **LSTM Model Version 2**

Due to an average level scoring of version 1, the next version is about introducing a few feature changes: (see Item 11)

* + - 1. Add another hidden layer (L2)
      2. Increase the epochs to 200 (from 50)
      3. Use optimizer = Adamax

Item 11: LSTM Model with two layers and changes Version 2

*# Adding layers*

model**.**add(LSTM(512, input\_shape**=**(num\_steps, X\_train**.**shape[2]), return\_sequences**=True**))

model**.**add(Dropout(0.1))

model**.**add(LSTM(256))

model**.**add(Dense(256))

model**.**add(Dropout(0.1))

model**.**add(Dense(num\_classes, activation**=**'softmax'))

*# Compiling the model for training*

opt **=** Adamax(learning\_rate**=**0.01)

model**.**compile(loss**=**'categorical\_crossentropy', optimizer**=**opt, metrics**=**['accuracy'])

Results:

Modifying the model did reduce the losses, which is good, and it did increase the training accuracy. However, the difference between the validating accuracy and the training one is too much. Again, not having enough validation data is the cause of this overfitting.

This can be noticed in the accuracy convergence of the train and validation data as seen in image 13

A graph of a train performance

Description automatically generatedImage 13: Train vs. Validation Accuracy (v.2)

## **LSTM Model Version 3**

In efforts to stop a situation of overfitting of the data, a new code feature is introduced to the model this is an early stopping of the fitting of the data. This is accomplished using the library:

**from** tensorflow.keras.callbacks **import** EarlyStopping

and settin the number of passes to 10 in this case.

early\_stopping **=** EarlyStopping(patience**=**10)

Results:

Introducing the early stopping criteria helped reduce overfitting and computational time. Notice how the scores are close enough to the trained data from the same model earlier with 500 epochs but without early stopping. See image 14

Image 14: Train vs. Validation Accuracy (v.3)

A graph of a graph with blue and orange lines

Description automatically generated

## **CNN MODEL (Convolutional Neural Network)**

Next version is about testing the CNN model with a regular numbering dataset.

Step one is to load the related Python libraries:

**from** tensorflow.keras.layers **import** Conv1D, MaxPooling1D, Flatten, Dense, Dropout

The model is structured of two hidden layers, utilizing the activation function of ReLU and softmax on the classification. See Item 16.

Item 16. CNN Model Version 1

model **=** Sequential([

Conv1D(64, 3, activation**=**'relu', input\_shape**=**(num\_steps, X\_train**.**shape[2])),

MaxPooling1D(2),

Conv1D(128, 3, activation**=**'relu'),

MaxPooling1D(2),

Flatten(),

Dense(128, activation**=**'relu'),

Dropout(0.3),

Dense(num\_classes, activation**=**'softmax')

])

Results:

Structure:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv1d (Conv1D) (None, 25, 64) 1024

max\_pooling1d (MaxPooling1D (None, 12, 64) 0

)

conv1d\_1 (Conv1D) (None, 10, 128) 24704

max\_pooling1d\_1 (MaxPooling (None, 5, 128) 0

1D)

flatten (Flatten) (None, 640) 0

dense\_6 (Dense) (None, 128) 82048

dropout\_7 (Dropout) (None, 128) 0

dense\_7 (Dense) (None, 9) 1161

=================================================================

Total params: 108,937

Trainable params: 108,937

Classification has improved, see Item 12.2

Classification Report - CNN Version 1 Model:

precision recall f1-score support

bach 0.50 0.56 0.53 9

bartok 0.62 0.62 0.62 8

byrd 0.56 0.62 0.59 8

chopin 0.55 0.75 0.63 8

handel 0.43 0.38 0.40 8

hummel 0.80 0.44 0.57 9

mendelssohn 0.25 0.38 0.30 8

mozart 0.50 0.38 0.43 8

schumann 0.50 0.38 0.43 8

accuracy 0.50 74

macro avg 0.52 0.50 0.50 74

weighted avg 0.53 0.50 0.50 74

In this model composer Bartok has top performance with a f1 score of 0.62 (62%) and the lowest score is with composer Mendelssohn at 0.30 (30%)