project_report_and_notebook-Team4

August 12, 2024

1 Combined Project Report and Notebook for Team 4

Team members: Brian Morris and Akram Mahmoud

Abstract Music, with its rich and diverse history, is a universal form of artistic expression. Each composer has contributed uniquely to the musical landscape, creating distinct styles and compositions. However, identifying the composer of a specific piece of music can be a complex and challenging task, particularly for novice musicians and listeners. This research seeks to address this challenge by leveraging the capabilities of deep learning. The primary objective of this study is to develop a robust model that can accurately predict the composer of a given musical score. To achieve this, we propose the integration of two powerful deep learning techniques: Long Short-Term Memory (LSTM) networks and Convolutional Neural Networks (CNN). LSTMs, known for their effectiveness in processing sequential data, will be utilized to capture the features in the musical compositions. Concurrently, CNNs will be employed to extract features from the musical data. By combining these techniques, the proposed model aims to enhance the accuracy of composer identification. The outcomes of this research have the potential to contribute significantly to the field of musicology and offer valuable tools for music analysis and education.

In this report we will alternate between explanations of our training process and the code used.

[2]:
```
# Install dependencies
import ast
import matplotlib.pyplot as plt
import music21
import numpy as np
import os

import pandas as pd
import kerastuner as kt
import tensorflow as tf

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,␣
↪Dropout
```

```
from tensorflow.keras.regularizers import l2

from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from imblearn.over_sampling import SMOTE
```

```
/var/folders/x5/cgpxmq_11ys090v206spbw7w0000gn/T/ipykernel_50536/479987768.py:8:
DeprecationWarning: `import kerastuner` is deprecated, please use `import
keras_tuner`.
  import kerastuner as kt
```

1.1 LSTM Model

LSTM - Data Preparation and Augmentation: I started by loading MIDI files from three different directories—training, validation, and test sets. These directories represent different parts of the dataset that I used for training the model, validating its performance during training, and evaluating it afterward. I used the `pretty_midi` library to parse the MIDI files, which allowed me to work with the musical data in a format suitable for machine learning. To make the model more robust and prevent it from overfitting, I applied data augmentation. This involved slightly altering the pitch of notes in the MIDI files, which helped simulate variations in the music and improved the model's ability to generalize to new data.

[ ]:
```
# Directory paths
DATA_DIR = 'Composer_Dataset/NN_midi_files_extended'
DEV_DIR = os.path.join(DATA_DIR, 'dev')
TEST_DIR = os.path.join(DATA_DIR, 'test')
TRAIN_DIR = os.path.join(DATA_DIR, 'train')
```

[ ]:
```
# Load MIDI files and labels
def load_midi_files(directory):
    midi_files = []
    labels = []
    for composer in os.listdir(directory):
        composer_dir = os.path.join(directory, composer)
        if os.path.isdir(composer_dir):
            for file in os.listdir(composer_dir):
                if file.endswith('.mid') or file.endswith('.midi'):
                    file_path = os.path.join(composer_dir, file)
                    midi_data = pretty_midi.PrettyMIDI(file_path)
                    midi_files.append(midi_data)
                    labels.append(composer)
    return midi_files, labels
```

[ ]:
```
train_midi_files, train_labels = load_midi_files(TRAIN_DIR)
dev_midi_files, dev_labels = load_midi_files(DEV_DIR)
test_midi_files, test_labels = load_midi_files(TEST_DIR)
```

[ ]:
```
# Data augmentation
def augment_midi(midi_data):
    for instrument in midi_data.instruments:
        for note in instrument.notes:
            note.pitch = note.pitch + np.random.choice([-1, 1])
    return midi_data

augmented_train_midi_files = [augment_midi(midi) for midi in train_midi_files]
```

LSTM - Feature Extraction: Once the data was ready, I moved on to extracting meaningful features from the MIDI files that could serve as inputs for the LSTM model. I focused on extracting sequences of note pitches, chord start times, and tempo changes. To ensure that these sequences could be used effectively by the model, I padded them to maintain uniform lengths. This step was important because neural networks require inputs of consistent dimensions. The features I extracted captured the musical content in a structured way, preserving the temporal and harmonic relationships in the music.

[ ]:
```
# Extract features from MIDI files
def extract_features(midi_files):
    note_sequences = []
    chord_sequences = []
    tempo_sequences = []

    for midi in midi_files:
        notes = [note.pitch for instrument in midi.instruments for note in␣
 ↪instrument.notes]
        chords = [note.start for instrument in midi.instruments for note in␣
 ↪instrument.notes]
        tempos = midi.get_tempo_changes()[1]

        note_sequences.append(notes)
        chord_sequences.append(chords)
        tempo_sequences.append(tempos)

    # Pad sequences
    note_sequences_padded = pad_sequences(note_sequences, padding='post',␣
 ↪maxlen=1000)
    chord_sequences_padded = pad_sequences(chord_sequences, padding='post',␣
 ↪maxlen=1000)
    tempo_sequences_padded = pad_sequences(tempo_sequences, padding='post',␣
 ↪maxlen=100)

    # Combine all features into a single feature array
    features = np.concatenate([note_sequences_padded, chord_sequences_padded,␣
 ↪tempo_sequences_padded], axis=1)

    return features
```

[ ]:
```
train_features = extract_features(train_midi_files + augmented_train_midi_files)
dev_features = extract_features(dev_midi_files)
test_features = extract_features(test_midi_files)
```

[ ]:
```
# Encode labels
label_encoder = LabelEncoder()
train_labels_encoded = to_categorical(label_encoder.fit_transform(train_labels␣
 ↪+ train_labels))
dev_labels_encoded = to_categorical(label_encoder.transform(dev_labels))
test_labels_encoded = to_categorical(label_encoder.transform(test_labels))
```

LSTM - Model Construction: With the features in hand, I built an LSTM model designed to classify the music based on these inputs. The LSTM model was a natural choice because it's well-suited for handling sequential data like music, where the order of notes and chords matters. I structured the model with two LSTM layers, which allowed it to learn patterns over time, followed by a dropout layer to help prevent overfitting. Finally, I added a dense layer for the actual classification. This setup leveraged the strengths of LSTM networks in capturing the temporal dependencies that are crucial for understanding musical compositions.

[ ]:
```
# Define the LSTM model with dropout
def create_lstm_model(input_shape, num_classes):
    model = Sequential()
    model.add(LSTM(128, input_shape=input_shape, return_sequences=True))
    model.add(Dropout(0.5))
    model.add(LSTM(128))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

[ ]:
```
# Input shape and number of classes
input_shape = (train_features.shape[1], 1)
num_classes = len(label_encoder.classes_)
```

LSTM - Model Training: Training the model was the next step, and I did this using the prepared and augmented datasets. I trained the LSTM model over 30 epochs with a batch size of 64. During this process, I carefully monitored its performance on the validation set to detect any signs of overfitting. As the training progressed, the model showed some improvement in accuracy, though I noticed that it struggled to generalize well. The validation accuracy fluctuated, which suggested that while the model was learning from the training data, it wasn't consistently capturing the underlying patterns that would allow it to perform well on new, unseen data.

[ ]:
```
# Create models
lstm_model = create_lstm_model(input_shape, num_classes)
```

[ ]:
```
# Train models
lstm_model.fit(train_features, train_labels_encoded,␣
 ↪validation_data=(dev_features, dev_labels_encoded), epochs=30, batch_size=64)
```

LSTM - Model Evaluation: After training, I evaluated the LSTM model on the test set to see how well it had learned to generalize. The model achieved a test accuracy of around 49%, which was a moderate success. It showed that the model could capture some of the patterns necessary for classification, but there was definitely room for improvement. These results indicated to me that while the LSTM model was somewhat effective, it still faced challenges in generalizing across different composers and compositions. I realized that further refinement of the model or exploring additional features could potentially enhance its performance.

[ ]:
```
# Evaluate models
lstm_eval = lstm_model.evaluate(test_features, test_labels_encoded)
```

[ ]:
```
print(f"LSTM Model - Loss: {lstm_eval[0]}, Accuracy: {lstm_eval[1]}")
```

1.2 CNN Model

Getting Started

1. download the dataset from kaggle
2. Move the dataset to where the notebook will be run, unarchive, and rename the parent folder to data
3. in your terminal run `pip install keras-tuner`
4. in your terminal `pip install music21`

1.2.1 CNN - 1. Data Preprocessing and 2. Feature Extraction

The dataset was provided as a collection of midi files. After investigating various libraries, I decided to use music21 to parse the midi files and extract the notes and chords. music 21 is "a python toolkit designed for computational analysis, and capable of handling a range of filetypes including MIDI" (Simonetta et al., 2023) Load the data from the dataset, use music21 to parse the midi files and extract the notes and chords.

In my first iteration of extraction, I created separate arrays for notes, chords, and tempo. However, extracting data in this pattern seemed suboptimal, as it missed the relationship between the notes and chords in time, and tempo wasn't related to the notes.

I searched for a better approach and found a paper by Verma and Thickstun (2019) that used a similar dataset to do composer prediction. Specifically, they paid more attention to the temporal aspect of the music. After reading their paper, I took a deeper dive of music21 documentation, to understand available features. I decided to re-extract the data, this time creating Tuples consisting of the note pitch, what beat the note occured on, and the note duration (e.g quarter note, half note, whole note). According to the Music21 documentation the note's pitch is "a number between 0 and 127 where middle C (C4) is 60 and C#4/Db4 is 61, B3 is 59, etc." (Music21, n.d.). I captured rests as -1 to better profile the music. For chords, I added up the individual pitches of the composing notes.

Each Tuple is either a: - Note: (note pitch, beat, duration) - Chord: (sum of individual note pitches, beat, duration) - Rest: (pitch of -1, beat, duration)

In this way I can capture the temporal relationship between the notes and chords, and include rests in the data, which seems like an important feature to include in the model. The output then looks like: > (-1, 1.0, 0.5), (48, 1.0, 1.0), (72, 1.5, .25)...

Helper Functions

[ ]:
```
def extract_note_features(midi_file):
    """
    Extracts note features from a MIDI file, including pitch, beat,
    and duration, returning a list of tuples for each note, rest,
    or chord in the file.
    """
    midi = music21.converter.parse(midi_file)

    note_tuples = []
    for thisNote in midi.flatten().getElementsByClass(['Note', 'Rest',␣
↪'Chord']):
        if thisNote.isNote:
            pitch = thisNote.pitch.midi
        elif thisNote.isRest:
            # Use -1 for rest
            pitch = -1
        elif thisNote.isChord:
            combined_pitch = 0
            for pitch in thisNote.pitches:
                combined_pitch += pitch.midi
            pitch = combined_pitch
        note_tuples.append((pitch, round(float(thisNote.beat), 2),␣
↪round(float(thisNote.duration.quarterLength), 2)))

    return note_tuples
```

[4]:
```
def extract_first_last_tuples(sequences, total_len):
    """
    Extracts the first and last segments of a sequence, combines them, and pads
    or truncates the result to a specified total length.
    """
    seq_len = len(sequences)
    half_len = total_len // 2
    if seq_len >= total_len:
        # If the sequence is longer than or equal to total_len, take the first␣
↪and last
        first_half = sequences[:half_len]
        last_half = sequences[-half_len:]
        combined = first_half + last_half
    else:
        combined = sequences
```

```
    combined = pad_sequences([combined], maxlen=total_len, padding='post',␣
 ↪truncating='post', dtype='float32')[0]

    return combined
```

[5]:
```
composer_list = ['Bach', 'Beethoven', 'Chopin', 'Mozart']
data = []
```

[ ]:
```
# Not executed the notebook submitted for the report, as it takes a long time␣
 ↪to run
for composer in composer_list:
    for dirname, _, filenames in os.walk(f'./data/{composer}/'):
        for filename in filenames:
            if filename.endswith('.mid'):
                midi_path = os.path.join(dirname, filename)
                note_tuples = extract_note_features(midi_path)
                data.append({
                    'midi_file': filename,
                    'composer': composer,
                    'path': dirname,
                    'note_tuple': note_tuples,
                })
```

[ ]:
```
df = pd.DataFrame(data)
```

[7]:
```
df.head()
```

[7]:

	midi_file	composer	path	note_tuple
0	Bwv0997 Partita for Lute 1mov.mid	Bach	./data/Bach/	[(-1, 4.0, 2.0), (48, 4.0, 4.0), (72, 6.0, 1.0…
1	Bwv0535 Prelude and Fugue.mid	Bach	./data/Bach/	[(-1, 4.0, 16.0), (55, 4.0, 1.0), (-1, 4.0, 16…
2	Bwv0806 English Suite n1 05mov.mid	Bach	./data/Bach/	[(69, 4.0, 2.0), (-1, 4.0, 2.0), (69, 5.0, 9.3…
3	Bwv0998 Prelude Fugue Allegro for Lute 3mov.mid	Bach	./data/Bach/	[(-1, 4.0, 1.0), (39, 4.0, 4.0), (-1, 4.0, 6.0…
4	Jesu Joy of Man Desiring.mid	Bach	./data/Bach/	[(-1, 4.0, 1.33), (-1, 4.0, 1.33), (-1, 4.0, 4…

[8]:
```
print(df.shape)
print(df.isna().sum())
```

```
(1530, 4)
midi_file     0
composer      0
path          0
```

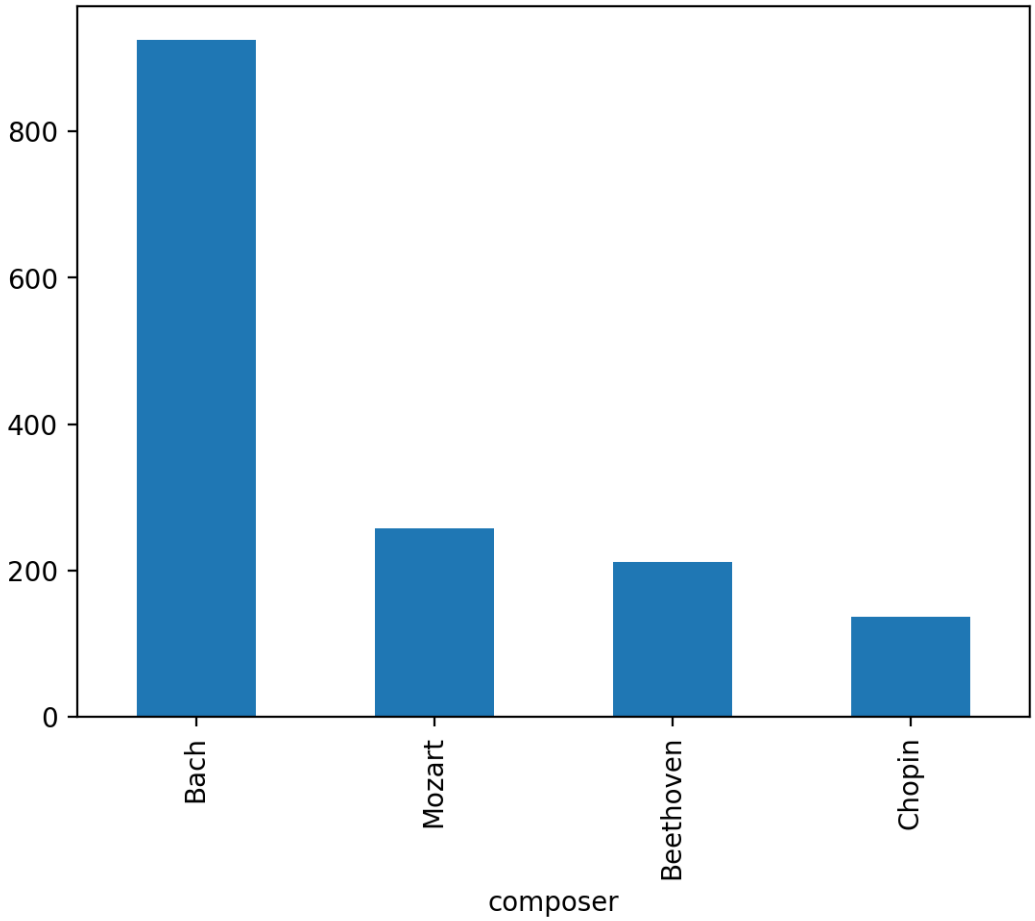
```
note_tuple    0
dtype: int64
```

[9]:
```
df['composer'].value_counts()
```

[9]:
```
composer
Bach          925
Mozart        257
Beethoven     212
Chopin        136
Name: count, dtype: int64
```

[10]:
```
# Visualize the distribution of composers
df['composer'].value_counts().plot(kind='bar')
```

[10]:
```
<Axes: xlabel='composer'>
```



[ ]:
```
# Save the extracted features for Model training
df.to_csv('./data/preprocessed_tuple_with_midi.csv', index=False)
```

[6]:
```
# Combining my prepocessing and model training notebooks into a single notebook␣
 ↪for submission
df = pd.read_csv('./data/preprocessed_tuple_with_midi.csv')
```

[11]:
```
# Need to convert the note_tuple column from string to list of tuples
df['note_tuple'] = df['note_tuple'].apply(lambda x: ast.literal_eval(x))
```

[12]:
```
TOTAL_NOTE_COUNT = 500
# Extract the first and last segments of the note tuples
df['first_last_tuples'] = df['note_tuple'].apply(lambda x:␣
 ↪extract_first_last_tuples(x, TOTAL_NOTE_COUNT))
```

CNN - 3. Model Building I ran several dozen experiments trying to identify the best model composition, hyperparameters, and features. I initially extracted the note pitch, the root chord, and the tempo. I ran a few experiments determining the impact of adding tempo, chords, and SMOTE to the training. Additionally, I started with the first 1000 notes, and played around with increasing the number of notes to 5000 and 10,000 under the assumption that more notes would provide more context to the model.

However, this was not the case, and the model consistently achieved a high training accuracy above 98%, but performed poorly on the validation set. I did see some small improvements with SMOTE and inclusion of tempo and chords in the training data, but the model was still overfitting, even with aggressive dropout rates. Other adjustments to reduce overfitting, such as reducing the batch size and layers were also unsuccessful. Model accuracy in the test set before I started hyperparameter tuning was in the 50%, and only improved to the 60-70% range with hours of hyperparameter tuning.

This lead me to conduct more research on the problem. I found a paper by Verma and Thickstun (2019) that discussed the challenges of overfitting when training models on entire musical scores, particularly due to the high dimensionality of the data. To mitigate this, they propose sub-sampling techniques, such as using the first and last few hundred notes from each score. This approach balances the need to capture essential features while avoiding overfitting to specific patterns that may not generalize well across different compositions. The authors found that this method improves model performance by focusing on the most informative sections of the score without overwhelming the model with redundant data (Verma & Thickstun, 2019).

The data is highly imbalanced, with Bach comprising of a majority of the datase (925). I used SMOTE to balance the data, but only saw a small improvement in the model's performance.

This approach is what is currently shown here in the code, with hyperparameter tuning applied to this approach.

Helper Functions

[13]:
```
def apply_smote(X, y):
    """
    Applies SMOTE to balance the classes in the dataset.
```

```
    """
    smote = SMOTE(random_state=42)

    num_samples, num_timesteps, num_features = X.shape
    X_flatten = X.reshape(num_samples, -1)

    smote = SMOTE(random_state=42)
    X_smote, y_smote = smote.fit_resample(X_flatten, y)

    X_smote_reshaped = X_smote.reshape(X_smote.shape[0], num_timesteps,␣
↪num_features)

    return X_smote_reshaped, y_smote
```

[14]:
```
class CNNHyperModel(kt.HyperModel):
    """
    CNN model for hyperparameter tuning using Keras Tuner.
    The keras Tuner doesn't support epochs or batch_size as a parameter, so I␣
↪overloaded the fit method to include them.
    """
    def build(self, hp):
        model = Sequential()
        model.add(Conv1D(filters=hp.Int('filters_1', min_value=64,␣
↪max_value=192, step=64),
                         kernel_size=3,
                         activation='relu',
                         kernel_regularizer=l2(hp.Choice('l2', [0.001, 0.0001])),
                         input_shape=input_shape))
        model.add(MaxPooling1D(2))
        model.add(Dropout(hp.Choice('dropout_1', [0.2, 0.3, 0.4])))

        model.add(Conv1D(filters=hp.Int('filters_2', min_value=64,␣
↪max_value=192, step=64),
                            kernel_size=3,
                            activation='relu',
                            kernel_regularizer=l2(hp.Choice('l2', [0.001, 0.
↪0001]))))
        model.add(MaxPooling1D(2))
        model.add(Dropout(hp.Choice('dropout_2', [0.2, 0.3, 0.4])))

        if hp.Boolean('add_third_conv_layer'):
            model.add(Conv1D(filters=hp.Int('filters_3', min_value=64,␣
↪max_value=192, step=64),
                            kernel_size=3,
                            activation='relu'))
            model.add(MaxPooling1D(2))
            model.add(Dropout(hp.Choice('dropout_3', [0.2, 0.3, 0.4])))
```

```
        model.add(Flatten())
        model.add(Dense(units=hp.Int('dense_units', min_value=32,␣
 ↪max_value=128, step=32),
                        activation='relu',
                        kernel_regularizer=l2(hp.Choice('l2', [0.001, 0.
 ↪0001]))))
        model.add(Dropout(hp.Choice('dropout_4', [0.2, 0.3, 0.4])))

        model.add(Dense(num_classes, activation='softmax'))

        learning_rate = hp.Choice('learning_rate', [0.001, 0.0005, 0.0001])
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

        model.compile(optimizer=optimizer,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        return model

    def fit(self, hp, model, *args, **kwargs):
        return model.fit(
            *args,
            batch_size=hp.Choice("batch_size", [24, 32, 40, 48]),
            epochs=hp.Choice("epochs", [15, 20, 30, 50, 100]),
            **kwargs,
        )
```

[15]:
```
X = np.array(df['first_last_tuples'].tolist())
```

[16]:
```
label_encoder = LabelEncoder()

y = label_encoder.fit_transform(df['composer'])
num_classes = len(label_encoder.classes_)
```

[17]:
```
X_train, X_temp, y_train, y_temp  = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

print("Shape before Smote", X_train.shape, y_train.shape)
X_train, y_train = apply_smote(X_train, y_train)
print("Shape after Smote", X_train.shape, y_train.shape)

y_train = to_categorical(y_train, num_classes=num_classes)
# Need to apply smote to training before we convert to categorical
y_temp = to_categorical(y_temp, num_classes=num_classes)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)
```

```
Shape before Smote (1224, 500, 3) (1224,)
```

```
Shape after Smote (3012, 500, 3) (3012,)
```

1.2.2 CNN - 4. Model Training

I chose to use the Keras Tuner library for hyperparameter optimiztion. Specifically the BayesianOptimization tuner, as my initial tuning results were giving me a lot of noise, and I wanted to add a high degree of flexibility in the hyperparameter search space. I'm less familiar with tuning hyperparameters for a CNN based model, so decided to search over a lot of my hypothesis, such as learning rate, batch size, a third convolutional layer, and more. BayesianOptimization seems to perform well over large search spaces.

I tried tuning with early stopping but got worse results, so left it out.

Despite many hours of tuning, I had intermittent results with the validation set. Occassionally I had validation set accuracies break into the 70's, but this did not consistenly deliver in the test set. Convergence was also poor, with training and validation accuracies diverging. I suspect that the model is overfitting to the training data despite many interventions to correct that, and that the model is not generalizing well to the test set.

[19]:
```
input_shape = X_train.shape[1:]

tuner = kt.BayesianOptimization(
    CNNHyperModel(),
    objective='val_accuracy',
    max_trials=15,
    executions_per_trial=2,
    directory='training_dir',
    project_name='cnn_tuning'
)

tuner.search(
    X_train, y_train,
    validation_data=(X_val, y_val)
)
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
```

```
Trial 15 Complete [00h 02m 45s]
val_accuracy: 0.6895425021648407

Best val_accuracy So Far: 0.6895425021648407
Total elapsed time: 00h 26m 11s
```

[20]:
```
best_batch_size = best_hps.get('batch_size')
best_epoch = best_hps.get('epochs')
print(f"""
The hyperparameter search is complete.
Best parameters:
Filters: {best_hps.get('filters_1')} and {best_hps.get('filters_2')}
Dense Units: {best_hps.get('dense_units')}
```

```
Dropout: {best_hps.get('dropout_1')}, {best_hps.get('dropout_2')}, {best_hps.
  ↪get('dropout_3')}, {best_hps.get('dropout_4')}
Regularization: {best_hps.get('l2')}
2nd convo layer: {best_hps.get('add_third_conv_layer')}
Learning Rate: {best_hps.get('learning_rate')}
Batch Size: {best_batch_size}
Epoch: {best_epoch}
""")
```

```
The hyperparameter search is complete.
Best parameters:
Filters: 64 and 64
Dense Units: 64
Dropout: 0.2, 0.4, 0.2, 0.2
Regularization: 0.0001
2nd convo layer: False
Learning Rate: 0.001
Batch Size: 48
Epoch: 100
```

[21]:
```
# Train the model with the best hyperparameters
model = tuner.hypermodel.build(best_hps)

history = model.fit(X_train, y_train, validation_data=(X_val, y_val),␣
  ↪epochs=best_epoch, batch_size=best_batch_size)
```

```
Epoch 1/100
63/63             1s 14ms/step -
accuracy: 0.2656 - loss: 9.9131 - val_accuracy: 0.2288 - val_loss: 1.4025
Epoch 2/100
63/63             1s 12ms/step -
accuracy: 0.2437 - loss: 1.4040 - val_accuracy: 0.3922 - val_loss: 1.3430
Epoch 3/100
63/63             1s 13ms/step -
accuracy: 0.3128 - loss: 1.3657 - val_accuracy: 0.4575 - val_loss: 1.1502
Epoch 4/100
63/63             1s 13ms/step -
accuracy: 0.3377 - loss: 1.3014 - val_accuracy: 0.5294 - val_loss: 1.0534
Epoch 5/100
63/63             1s 12ms/step -
accuracy: 0.3495 - loss: 1.2998 - val_accuracy: 0.5425 - val_loss: 1.0383
Epoch 6/100
63/63             1s 13ms/step -
accuracy: 0.3890 - loss: 1.2438 - val_accuracy: 0.5098 - val_loss: 1.0436
Epoch 7/100
63/63             1s 13ms/step -
```

```
accuracy: 0.4108 - loss: 1.2104 - val_accuracy: 0.4706 - val_loss: 1.0397
Epoch 8/100
63/63             1s 13ms/step -
accuracy: 0.4820 - loss: 1.1482 - val_accuracy: 0.4967 - val_loss: 1.0560
Epoch 9/100
63/63             1s 14ms/step -
accuracy: 0.5180 - loss: 1.0294 - val_accuracy: 0.5686 - val_loss: 1.0278
Epoch 10/100
63/63             1s 13ms/step -
accuracy: 0.5442 - loss: 0.9639 - val_accuracy: 0.5556 - val_loss: 1.0119
Epoch 11/100
63/63             1s 13ms/step -
accuracy: 0.5885 - loss: 0.9028 - val_accuracy: 0.5098 - val_loss: 1.0196
Epoch 12/100
63/63             1s 13ms/step -
accuracy: 0.6823 - loss: 0.7407 - val_accuracy: 0.5229 - val_loss: 1.1114
Epoch 13/100
63/63             1s 12ms/step -
accuracy: 0.7017 - loss: 0.6717 - val_accuracy: 0.5621 - val_loss: 1.0966
Epoch 14/100
63/63             1s 14ms/step -
accuracy: 0.7112 - loss: 0.6680 - val_accuracy: 0.5621 - val_loss: 1.0317
Epoch 15/100
63/63             1s 13ms/step -
accuracy: 0.7766 - loss: 0.5861 - val_accuracy: 0.5686 - val_loss: 1.1472
Epoch 16/100
63/63             1s 13ms/step -
accuracy: 0.7795 - loss: 0.5524 - val_accuracy: 0.5817 - val_loss: 0.9965
Epoch 17/100
63/63             1s 13ms/step -
accuracy: 0.7985 - loss: 0.5187 - val_accuracy: 0.5556 - val_loss: 1.0783
Epoch 18/100
63/63             1s 13ms/step -
accuracy: 0.8087 - loss: 0.4861 - val_accuracy: 0.6078 - val_loss: 1.0319
Epoch 19/100
63/63             1s 12ms/step -
accuracy: 0.8241 - loss: 0.4873 - val_accuracy: 0.5621 - val_loss: 1.0780
Epoch 20/100
63/63             1s 13ms/step -
accuracy: 0.8178 - loss: 0.4576 - val_accuracy: 0.5882 - val_loss: 1.1611
Epoch 21/100
63/63             1s 13ms/step -
accuracy: 0.8404 - loss: 0.4000 - val_accuracy: 0.5556 - val_loss: 1.0329
Epoch 22/100
63/63             1s 12ms/step -
accuracy: 0.8494 - loss: 0.3906 - val_accuracy: 0.5686 - val_loss: 1.1660
Epoch 23/100
63/63             1s 13ms/step -
```

```
accuracy: 0.8641 - loss: 0.3674 - val_accuracy: 0.5752 - val_loss: 1.3107
Epoch 24/100
63/63             1s 13ms/step -
accuracy: 0.8734 - loss: 0.3455 - val_accuracy: 0.6013 - val_loss: 1.1545
Epoch 25/100
63/63             1s 12ms/step -
accuracy: 0.8725 - loss: 0.3499 - val_accuracy: 0.6013 - val_loss: 1.0651
Epoch 26/100
63/63             1s 13ms/step -
accuracy: 0.8768 - loss: 0.3536 - val_accuracy: 0.6013 - val_loss: 1.2673
Epoch 27/100
63/63             1s 13ms/step -
accuracy: 0.8834 - loss: 0.3399 - val_accuracy: 0.6078 - val_loss: 1.2555
Epoch 28/100
63/63             1s 12ms/step -
accuracy: 0.8882 - loss: 0.3164 - val_accuracy: 0.5882 - val_loss: 1.1662
Epoch 29/100
63/63             1s 12ms/step -
accuracy: 0.8905 - loss: 0.2921 - val_accuracy: 0.5686 - val_loss: 1.2828
Epoch 30/100
63/63             1s 13ms/step -
accuracy: 0.8795 - loss: 0.3395 - val_accuracy: 0.5882 - val_loss: 1.2474
Epoch 31/100
63/63             1s 13ms/step -
accuracy: 0.8995 - loss: 0.3176 - val_accuracy: 0.6340 - val_loss: 1.2842
Epoch 32/100
63/63             1s 13ms/step -
accuracy: 0.9098 - loss: 0.2589 - val_accuracy: 0.6340 - val_loss: 1.1797
Epoch 33/100
63/63             1s 12ms/step -
accuracy: 0.8963 - loss: 0.3018 - val_accuracy: 0.5948 - val_loss: 1.5190
Epoch 34/100
63/63             1s 13ms/step -
accuracy: 0.9067 - loss: 0.2671 - val_accuracy: 0.6013 - val_loss: 1.3026
Epoch 35/100
63/63             1s 13ms/step -
accuracy: 0.9178 - loss: 0.2608 - val_accuracy: 0.6209 - val_loss: 1.2946
Epoch 36/100
63/63             1s 13ms/step -
accuracy: 0.9145 - loss: 0.2540 - val_accuracy: 0.6209 - val_loss: 1.2639
Epoch 37/100
63/63             1s 12ms/step -
accuracy: 0.9293 - loss: 0.2334 - val_accuracy: 0.5686 - val_loss: 1.3702
Epoch 38/100
63/63             1s 12ms/step -
accuracy: 0.9218 - loss: 0.2326 - val_accuracy: 0.6275 - val_loss: 1.4531
Epoch 39/100
63/63             1s 13ms/step -
```

```
accuracy: 0.9062 - loss: 0.2671 - val_accuracy: 0.5882 - val_loss: 1.5231
Epoch 40/100
63/63             1s 12ms/step -
accuracy: 0.9282 - loss: 0.2401 - val_accuracy: 0.5686 - val_loss: 1.4579
Epoch 41/100
63/63             1s 14ms/step -
accuracy: 0.9395 - loss: 0.2086 - val_accuracy: 0.6275 - val_loss: 1.4863
Epoch 42/100
63/63             1s 13ms/step -
accuracy: 0.9280 - loss: 0.2224 - val_accuracy: 0.6667 - val_loss: 1.4069
Epoch 43/100
63/63             1s 13ms/step -
accuracy: 0.9298 - loss: 0.2276 - val_accuracy: 0.5948 - val_loss: 1.6159
Epoch 44/100
63/63             1s 13ms/step -
accuracy: 0.9433 - loss: 0.1958 - val_accuracy: 0.5621 - val_loss: 1.6686
Epoch 45/100
63/63             1s 13ms/step -
accuracy: 0.9391 - loss: 0.2006 - val_accuracy: 0.6209 - val_loss: 1.4780
Epoch 46/100
63/63             1s 12ms/step -
accuracy: 0.9573 - loss: 0.1822 - val_accuracy: 0.6013 - val_loss: 1.4140
Epoch 47/100
63/63             1s 13ms/step -
accuracy: 0.9432 - loss: 0.2024 - val_accuracy: 0.6340 - val_loss: 1.5854
Epoch 48/100
63/63             1s 12ms/step -
accuracy: 0.9459 - loss: 0.1866 - val_accuracy: 0.5752 - val_loss: 1.7192
Epoch 49/100
63/63             1s 13ms/step -
accuracy: 0.9545 - loss: 0.1690 - val_accuracy: 0.6340 - val_loss: 1.4109
Epoch 50/100
63/63             1s 13ms/step -
accuracy: 0.9526 - loss: 0.1788 - val_accuracy: 0.6275 - val_loss: 1.6822
Epoch 51/100
63/63             1s 13ms/step -
accuracy: 0.9464 - loss: 0.2019 - val_accuracy: 0.5948 - val_loss: 1.5981
Epoch 52/100
63/63             1s 13ms/step -
accuracy: 0.9521 - loss: 0.1760 - val_accuracy: 0.6144 - val_loss: 1.7578
Epoch 53/100
63/63             1s 12ms/step -
accuracy: 0.9512 - loss: 0.1875 - val_accuracy: 0.6340 - val_loss: 1.4701
Epoch 54/100
63/63             1s 12ms/step -
accuracy: 0.9455 - loss: 0.1874 - val_accuracy: 0.6601 - val_loss: 1.4168
Epoch 55/100
63/63             1s 13ms/step -
```

```
accuracy: 0.9540 - loss: 0.1729 - val_accuracy: 0.6013 - val_loss: 1.6703
Epoch 56/100
63/63             1s 12ms/step -
accuracy: 0.9519 - loss: 0.1823 - val_accuracy: 0.6209 - val_loss: 1.5954
Epoch 57/100
63/63             1s 12ms/step -
accuracy: 0.9460 - loss: 0.1901 - val_accuracy: 0.6013 - val_loss: 1.7665
Epoch 58/100
63/63             1s 13ms/step -
accuracy: 0.9649 - loss: 0.1469 - val_accuracy: 0.6144 - val_loss: 1.6468
Epoch 59/100
63/63             1s 12ms/step -
accuracy: 0.9558 - loss: 0.1794 - val_accuracy: 0.6732 - val_loss: 1.5394
Epoch 60/100
63/63             1s 13ms/step -
accuracy: 0.9569 - loss: 0.1599 - val_accuracy: 0.6209 - val_loss: 1.8760
Epoch 61/100
63/63             1s 13ms/step -
accuracy: 0.9539 - loss: 0.1840 - val_accuracy: 0.6078 - val_loss: 1.7273
Epoch 62/100
63/63             1s 13ms/step -
accuracy: 0.9609 - loss: 0.1473 - val_accuracy: 0.6275 - val_loss: 1.4594
Epoch 63/100
63/63             1s 12ms/step -
accuracy: 0.9570 - loss: 0.1733 - val_accuracy: 0.6340 - val_loss: 1.8093
Epoch 64/100
63/63             1s 13ms/step -
accuracy: 0.9682 - loss: 0.1530 - val_accuracy: 0.6601 - val_loss: 1.6158
Epoch 65/100
63/63             1s 14ms/step -
accuracy: 0.9583 - loss: 0.1644 - val_accuracy: 0.6013 - val_loss: 1.5960
Epoch 66/100
63/63             1s 13ms/step -
accuracy: 0.9651 - loss: 0.1528 - val_accuracy: 0.6536 - val_loss: 1.6752
Epoch 67/100
63/63             1s 13ms/step -
accuracy: 0.9655 - loss: 0.1486 - val_accuracy: 0.5948 - val_loss: 1.6657
Epoch 68/100
63/63             1s 13ms/step -
accuracy: 0.9565 - loss: 0.1652 - val_accuracy: 0.6144 - val_loss: 1.9493
Epoch 69/100
63/63             1s 12ms/step -
accuracy: 0.9625 - loss: 0.1534 - val_accuracy: 0.6340 - val_loss: 1.7489
Epoch 70/100
63/63             1s 13ms/step -
accuracy: 0.9603 - loss: 0.1600 - val_accuracy: 0.6536 - val_loss: 1.6756
Epoch 71/100
63/63             1s 13ms/step -
```

```
accuracy: 0.9678 - loss: 0.1387 - val_accuracy: 0.5359 - val_loss: 2.0665
Epoch 72/100
63/63            1s 13ms/step -
accuracy: 0.9697 - loss: 0.1457 - val_accuracy: 0.6013 - val_loss: 1.9573
Epoch 73/100
63/63            1s 12ms/step -
accuracy: 0.9661 - loss: 0.1461 - val_accuracy: 0.5882 - val_loss: 1.7027
Epoch 74/100
63/63            1s 13ms/step -
accuracy: 0.9679 - loss: 0.1387 - val_accuracy: 0.6078 - val_loss: 1.8817
Epoch 75/100
63/63            1s 12ms/step -
accuracy: 0.9754 - loss: 0.1309 - val_accuracy: 0.5490 - val_loss: 1.9726
Epoch 76/100
63/63            1s 12ms/step -
accuracy: 0.9718 - loss: 0.1368 - val_accuracy: 0.6275 - val_loss: 1.9326
Epoch 77/100
63/63            1s 13ms/step -
accuracy: 0.9648 - loss: 0.1535 - val_accuracy: 0.6013 - val_loss: 1.4438
Epoch 78/100
63/63            1s 13ms/step -
accuracy: 0.9727 - loss: 0.1329 - val_accuracy: 0.5752 - val_loss: 2.1024
Epoch 79/100
63/63            1s 13ms/step -
accuracy: 0.9760 - loss: 0.1179 - val_accuracy: 0.6209 - val_loss: 1.6683
Epoch 80/100
63/63            1s 13ms/step -
accuracy: 0.9764 - loss: 0.1196 - val_accuracy: 0.5948 - val_loss: 1.6689
Epoch 81/100
63/63            1s 14ms/step -
accuracy: 0.9650 - loss: 0.1514 - val_accuracy: 0.5817 - val_loss: 1.5089
Epoch 82/100
63/63            1s 13ms/step -
accuracy: 0.9702 - loss: 0.1390 - val_accuracy: 0.6144 - val_loss: 2.0939
Epoch 83/100
63/63            1s 14ms/step -
accuracy: 0.9661 - loss: 0.1508 - val_accuracy: 0.5686 - val_loss: 1.8864
Epoch 84/100
63/63            1s 14ms/step -
accuracy: 0.9694 - loss: 0.1440 - val_accuracy: 0.5948 - val_loss: 2.1162
Epoch 85/100
63/63            1s 14ms/step -
accuracy: 0.9660 - loss: 0.1838 - val_accuracy: 0.5686 - val_loss: 1.9981
Epoch 86/100
63/63            1s 13ms/step -
accuracy: 0.9668 - loss: 0.1503 - val_accuracy: 0.6536 - val_loss: 1.8455
Epoch 87/100
63/63            1s 15ms/step -
```

```
accuracy: 0.9643 - loss: 0.1521 - val_accuracy: 0.5752 - val_loss: 2.0656
Epoch 88/100
63/63             1s 13ms/step -
accuracy: 0.9731 - loss: 0.1458 - val_accuracy: 0.6078 - val_loss: 1.8053
Epoch 89/100
63/63             1s 14ms/step -
accuracy: 0.9702 - loss: 0.1307 - val_accuracy: 0.5948 - val_loss: 2.0545
Epoch 90/100
63/63             1s 13ms/step -
accuracy: 0.9761 - loss: 0.1238 - val_accuracy: 0.6209 - val_loss: 1.9872
Epoch 91/100
63/63             1s 14ms/step -
accuracy: 0.9696 - loss: 0.1379 - val_accuracy: 0.6405 - val_loss: 1.5201
Epoch 92/100
63/63             1s 13ms/step -
accuracy: 0.9744 - loss: 0.1225 - val_accuracy: 0.6144 - val_loss: 1.6033
Epoch 93/100
63/63             1s 13ms/step -
accuracy: 0.9687 - loss: 0.1408 - val_accuracy: 0.5817 - val_loss: 1.8140
Epoch 94/100
63/63             1s 12ms/step -
accuracy: 0.9718 - loss: 0.1302 - val_accuracy: 0.6013 - val_loss: 1.5154
Epoch 95/100
63/63             1s 14ms/step -
accuracy: 0.9784 - loss: 0.1169 - val_accuracy: 0.5882 - val_loss: 1.9393
Epoch 96/100
63/63             1s 12ms/step -
accuracy: 0.9746 - loss: 0.1269 - val_accuracy: 0.5882 - val_loss: 2.4029
Epoch 97/100
63/63             1s 12ms/step -
accuracy: 0.9770 - loss: 0.1221 - val_accuracy: 0.6078 - val_loss: 1.6036
Epoch 98/100
63/63             1s 12ms/step -
accuracy: 0.9770 - loss: 0.1221 - val_accuracy: 0.5686 - val_loss: 2.2495
Epoch 99/100
63/63             1s 12ms/step -
accuracy: 0.9721 - loss: 0.1362 - val_accuracy: 0.5817 - val_loss: 1.7383
Epoch 100/100
63/63             1s 12ms/step -
accuracy: 0.9731 - loss: 0.1259 - val_accuracy: 0.5882 - val_loss: 2.6194
```

1.2.3 CNN - 5. Model Evaluation

[22]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')
```

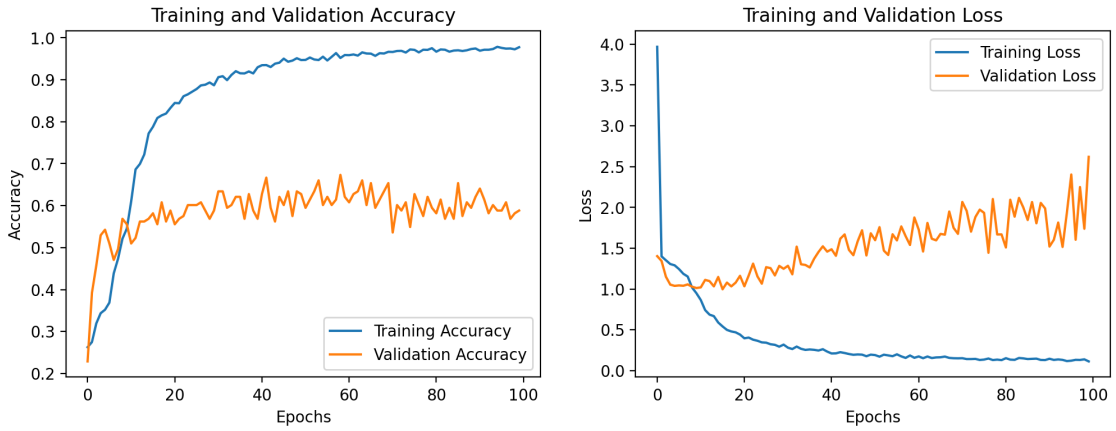
```
5/5             0s 3ms/step -
accuracy: 0.6662 - loss: 2.9096
```

```
Test accuracy: 0.6666666865348816
```

[25]:
```
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

print(classification_report(y_true, y_pred_classes, target_names=label_encoder.
 ↪classes_))
```

```
5/5             0s 3ms/step
              precision    recall  f1-score   support

        Bach       0.77      0.91      0.84        87
   Beethoven       0.54      0.28      0.37        25
      Chopin       0.75      0.30      0.43        10
      Mozart       0.38      0.42      0.40        31

    accuracy                           0.67       153
   macro avg       0.61      0.48      0.51       153
weighted avg       0.65      0.67      0.64       153
```

[24]:
```
# Plotting the training and validation accuracy
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



1.2.4 CNN - 6. Additional Model analysis and Next Steps

As you can see, even with a high degree of hyperparameter tuning and feature extraction, the model is still overfitting, but the validation accuracy is much higher than before. We perform quite high on Bach, but even applying SMOTE to increase the data for the less prevalent composers doesn't offer much improvements. A clear next step would be to gather additional training works for those composers. I believe this is due to the sub-sampling technique I used to extract the data.

In their study, Verma and Thickstun (2019) discuss the development of a hybrid model for composer identification that combines two CNN models (temporal and harmonic) to enhance predictive accuracy while avoiding overfitting. These models are applied separately to the input data, and their features are then combined in a final layer to make predictions. Given additional time, I would like to explore similar approaches to improve the accuracy of a CNN based approach.

That said, even in their study, Verma and Thickstun were only able to achieve a peak accuracy of 81.7% and that was on a different dataset with 2500 scores, albeit higher number of composers (2019). I think the complexity of music data and the high dimensionality of the data makes it difficult to achieve high accuracy with a CNN model.

1.2.5 References:

- Music21. (n.d.). *Users' guide: Chapter 3 pitches*. Retrieved August 11, 2024, from https://www.music21.org/music21docs/usersGuide/usersGuide_03_pitches.html
- Verma, H., & Thickstun, J. (2019). Convolutional composer classification. *arXiv preprint arXiv:1911.11737*.
- Simonetta, F., Llorens, A., Serrano, M., García-Portugués, E., & Torrente, Á. (2023). Optimizing feature extraction for symbolic music. *arXiv*. https://doi.org/10.48550/arXiv.2307.05107

Libraries Used:

- imbalanced-learn - SMOTE: Lemaitre, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A Python toolbox to tackle the curse of imbalanced datasets in machine learning.

Journal of Machine Learning Research, 18(17), 1-5. http://jmlr.org/papers/v18/16-365.html

- keras-tuner: O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., & others. (2019). *Keras Tuner.* GitHub. https://github.com/keras-team/keras-tuner
- music21: Cuthbert, M. S. A. (2006-2023). *music21: A toolkit for computer-aided musicology.* GitHub. https://github.com/cuthbertLab/music21

Source code:

- Morris, B. Mahmoud, A. AAI_511_Neural_Networks [Source code]. GitHub. https://github.com/Briantmorr/AAI_511_Neural_Networks