Music Classifier Using CNNs

Project by Akshath Rao & Joe Prince Overview:

In this project, a Convolutional Neural Network (CNN) was implemented to classify audio files into their respective music genres. The audio dataset consisted of 10 genres: blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, and rock. Each genre contained 100 tracks of 30 seconds each, formatted as .wav files.

Link to dataset: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification

```python
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import math
import json
import scipy
import librosa

def plot_waveforms(audio, fs):
    """Plots the waveform of audio in the time domain.

    Parameters:
        audio (numpy.ndarray): audio signal
        fs (int): sampling frequency (Hz) of audio signal

    """
    plt.figure(figsize=(12, 6))
    librosa.display.waveshow(audio, sr=fs, alpha=0.58)
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.show()

def calculate_spectrum(audio, kind='mag'):
    """
    Calculates the spectrum of an audio signal.
    Parameters:
        audio (numpy.ndarray): audio signal
        kind (str): 'mag' for magnitude, 'phase' for phase, 'complex'
for complex
    """
    spec = scipy.fft.fft(audio)
    if kind == 'mag':
        return 20*np.log10(np.abs(spec))
    elif kind == 'phase':
        return np.angle(spec)
    elif kind == 'complex':
        return 20*np.log10(spec)
    else:
        raise ValueError('Invalid kind')

#Function to plot spectrum
def plot_spec(audio, fs, kind):
    """
    Plots the spectrum of an audio signal.
    parameters:
        audio (numpy.ndarray): audio signal
        fs (int): sampling frequency (Hz) of audio signal
        kind (str): 'mag' for magnitude, 'phase' for phase, 'complex'
for complex
    """
    spec_db = calculate_spectrum(audio, kind)
    frequency_axis = np.linspace(0, fs, len(spec_db))
    #Nyquist frequencies
    frequency_axis = frequency_axis[:len(frequency_axis)//2]
    spec_db = spec_db[:len(spec_db)//2]

    #plot
    ax = plt.figure(figsize=(12, 6))
    plt.plot(frequency_axis, spec_db)
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Magnitude (dB)")
    plt.show()
    if fs < 44100:
        plt.xticks([1, 2, 4, 8, 16, 31, 63, 125,
250,500,1000,2000,5000,10000],
                   ["1", "2", "4", "8", "16", "31", "63", "125",
"250", "500", "1K", "2K", "5K", "10K"])
    else:
        plt.xticks([1, 2, 4, 8, 16, 31, 63, 125,
250,500,1000,2000,5000,10000, 20000],
                   ["1", "2", "4", "8", "16", "31", "63", "125",
"250", "500", "1K", "2K", "5K", "10K", "20k"])

def calculate_stft(audio, fs, n_fft=2048, hop_length=512, dB=True):
    """
    Calculates the Short-Time Fourier Transform (STFT) of an audio
signal.
    Parameters:
        audio (numpy.ndarray): audio signal
        fs (int): sampling frequency (Hz) of audio signal
        n_fft (int): number of samples per frame
        hop_length (int): number of samples between frames
        dB (bool): if True, returns the magnitude in decibels
    """
    stft = librosa.stft(audio, n_fft=n_fft, hop_length=hop_length)
    if dB:
        return librosa.amplitude_to_db(np.abs(stft)) # Convert to dB
in log scale
    else:
        return np.abs(stft)

def spectrogram(audio, fs, n_fft = 2048, hop_length = 512, dB = True):
    """
    Plots the spectrogram of an audio signal.
    parameters:
        audio (numpy.ndarray): audio signal
        fs (int): sampling frequency (Hz) of audio signal
        n_fft (int): number of samples per frame
        hop_length (int): number of samples between frames
        dB (bool): if True, returns the magnitude in decibels
    """
    stft_db = calculate_stft(audio, fs, n_fft, hop_length, dB)
    plt.figure(figsize=(12,6))
    librosa.display.specshow(stft_db, sr=fs, hop_length=hop_length,
x_axis='time', y_axis='linear', cmap = 'inferno')
    plt.title('Spectrogram')
    plt.colorbar(format='%+2.0f dB')
    plt.show()
    plt.tight_layout()

def calculate_mel_spec(audio, fs, n_mfcss=128, n_fft = 2048,
hop_length = 512):
    mel_spec = librosa.feature.melspectrogram(audio, sr=fs,
n_mels=n_mfcss, n_fft=n_fft, hop_length=hop_length)
    mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)
    return mel_spec_db

def plot_mel_spectrogram_audio(audio, fs, n_mfccs=128, n_fft=2048,
hop_length=512, fig_size=(12,6)):
    """Plots the mel-scaled spectrogram from audio signal.

    Parameters:
        audio (numpy.ndarray): audio signal
        fs (int): sampling frequency (Hz) of audio signal
        n_mfccs: The number of MFCCs to compute (i.e. dimensionality
of mel spectrum)
        n_fft (int): The length (i.e. resolution) of the FFT window
(must be power of 2)
        hop_length (int): The number of samples between successive
frames
        fig_size (tuple): Dimensions of figure
    """
    # Calculate mel-spectrogram
    mel_spec_db = calculate_mel_spec(audio, fs, n_mfccs=n_mfccs,
n_fft=n_fft, hop_length=hop_length)

    # Plot Spectrogram
    plt.figure(figsize=fig_size)
    librosa.display.specshow(data=mel_spec_db, sr=fs, x_axis='time', 
y_axis='mel', cmap='viridis')

    # Put a descriptive title on the plot
    plt.title('Mel Power Spectrogram')

    # draw a color bar
    plt.colorbar(format='%+02.0f dB')

    # Make the figure layout compact
    plt.tight_layout()

def plot_low_res_mfcc(mfcc, fs, fig_size=(12,6)):
    """Plots the mel-scaled spectrogram from mfccs. This is performing 
the same task as
    'plot_mel_spectrogram_audio' with just a different input.

    Parameters:
        mfcc (numpy.ndarray): mfccs of an audio signal
        fs (int): sampling frequency (Hz) of audio signal
        fig_size (tuple): Dimensions of figure
    """
    # Plot Spectrogram
    plt.figure(figsize=fig_size)

    # Display the spectrogram on a mel scale
    # sample rate and hop length parameters are used to render the 
time axis
    # abs on signal for better visualization
    librosa.display.specshow(data=mfcc, sr=fs, x_axis='time', 
y_axis='linear', cmap='viridis')

    # Put a descriptive title on the plot
    plt.title('MFCCs')

    # draw a color bar
    plt.colorbar(format='%+02.0f dB')

    # Make the figure layout compact
    plt.tight_layout()
```

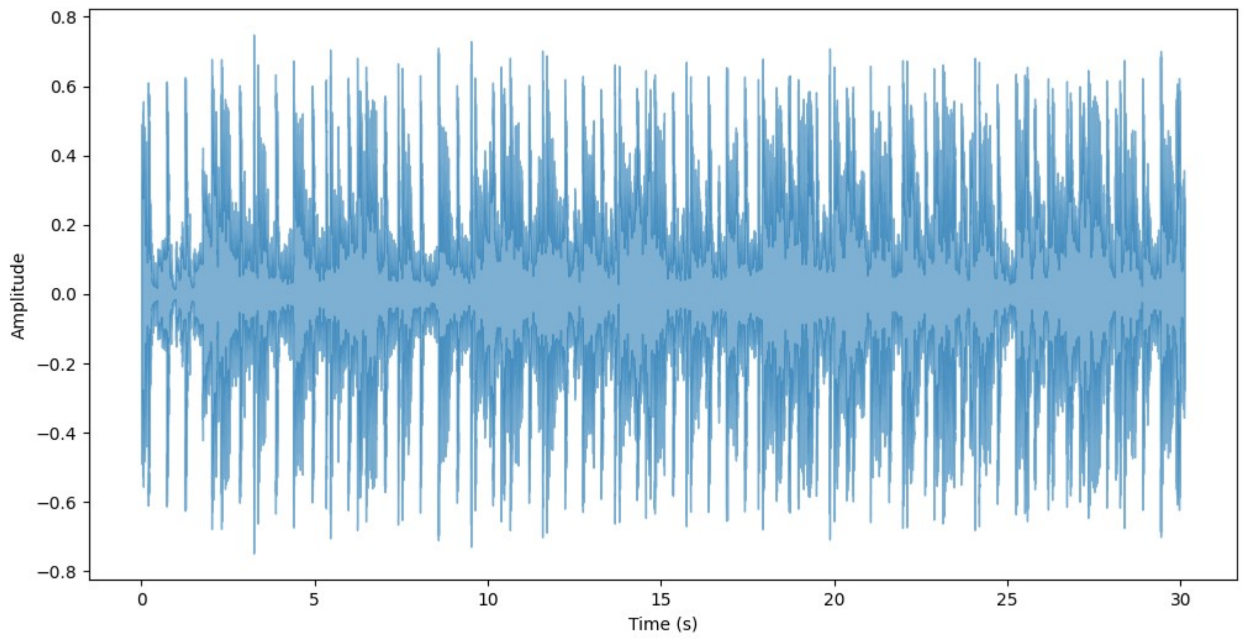
Pre-Processed Data Examples:

```
path_data = 'C:/Users/aksha/Downloads/DataMusic/genres_original/'
genre = 'disco/'
filename = 'disco.00000.wav'
```

```
file_path = path_data + genre + filename

print(file_path)

C:/Users/aksha/Downloads/DataMusic/genres_original/disco/
disco.00000.wav

fs = 22050 # sampling rate for librosa to resample to
audio_ex, fs = librosa.load(path=file_path, sr=fs) # load audio and
sampling rate
```

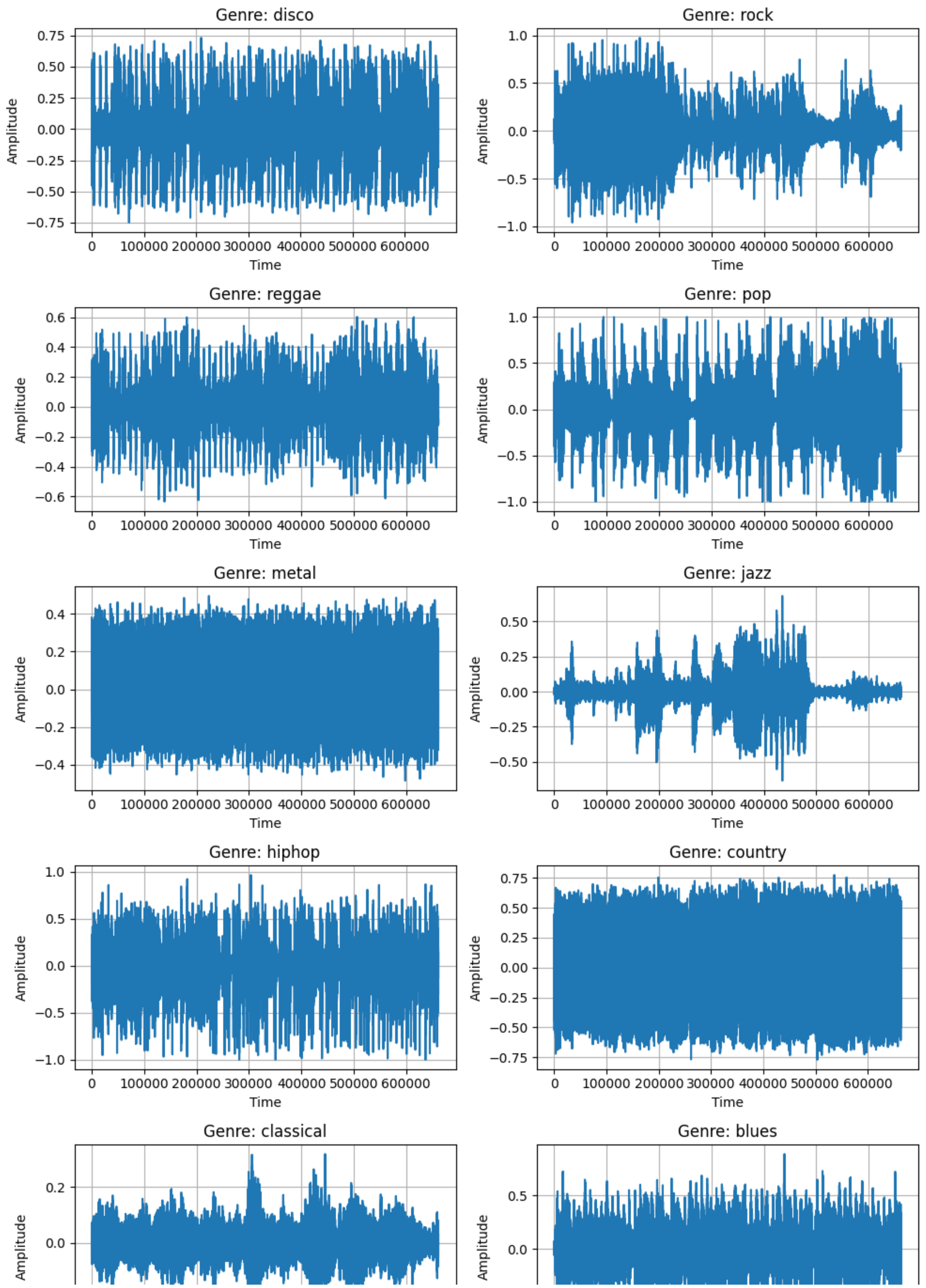
Example Disco Track audio waveform plotted

```
plot_waveforms(audio_ex, fs)
```



Audio Waveforms before any transformations:

```
# Define file paths for all genres
path_data = 'C:/Users/aksha/Downloads/DataMusic/genres_original/'

genres = ['disco', 'rock', 'reggae', 'pop', 'metal', 'jazz', 'hiphop',
'country', 'classical', 'blues']
file_paths = {genre: path_data + genre + '/' + genre + '.00000.wav'
for genre in genres}

# Function to plot waveforms
def plot_waveforms(audio_data, sampling_rate, genre_name,
subplot_num):
    plt.subplot(5, 2, subplot_num)
    plt.plot(audio_data)
    plt.title(f'Genre: {genre_name}')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
    plt.grid(True)

# Plotting all waveforms
fs = 22050  # Sampling rate for librosa
plt.figure(figsize=(10, 15))

for i, genre in enumerate(genres):
    file_path = file_paths[genre]
    audio_ex, fs = librosa.load(path=file_path, sr=fs)  # Load audio
and resample
    plot_waveforms(audio_ex, fs, genre, i + 1)

plt.tight_layout()
plt.show()
```


Genre: disco
Genre: rock
Genre: reggae
Genre: pop
Genre: metal
Genre: jazz
Genre: hiphop
Genre: country
Genre: classical
Genre: blues
Amplitude
Time
0
100000
200000
300000
400000
500000
600000

Preprocessing Methodology:

Audio Data to MFCC Conversion and Export

Audio data (.wav files) were converted into Mel-Frequency Cepstral Coefficients (MFCCs), a widely-used representation in audio classification tasks. MFCCs summarize the audio spectrum into a small number of representative features, similar to images, thus making them suitable for input to convolutional neural networks.

Function: get_mfccs() Input: Path to .wav files directory. Output: JSON file containing: mfcc: MFCC feature arrays (dimensions: samples × time-frames × coefficients genre_num: Numeric labels for genres.

Output shape example:

MFCC Data shape: (9989, 132, 13)

Labels shape: (9989,)

```
import os
import librosa
import numpy as np
import json
import math

def get_mfccs(directory_path, fs=22500, duration=30, n_fft=2048,
hop_length=512, n_mfcc=13, num_segments=10,
save_path='C:/Users/aksha/Downloads/DataMusic/data.json'):
    """
    Reads through a directory of audio files and saves a dictionary of
MFCCs and genres to a .json file.
    It also returns numpy.ndarrays for MFCCs, genre name, and genre
number for each segment of the audio signal.

    Parameters:
        directory_path (str): Path to the directory containing audio
files, where each genre has its own folder.
        fs (int): Sampling frequency (Hz) of the audio signal.
        duration (int): Duration of audio signal (sec).
        n_fft (int): The length (i.e. resolution) of the FFT window
(must be a power of 2).
        hop_length (int): The number of samples between successive
frames.
        n_mfcc (int): The number of MFCCs to compute (dimensionality
of mel spectrum).
        num_segments (int): The number of segments to divide the audio
signal into.
        save_path (str): The path where the JSON file will be saved.

    Returns:
        np.ndarray: MFCCs for each segment of audio.
        np.ndarray: Genre names for each segment.
        np.ndarray: Genre numbers for each segment.
    """
    data = {
        "genre_name": [],   # List of genre names (i.e., blues,
classical, etc.)
        "genre_num": [],    # List of genre numbers (i.e., 0, 1, 2,
etc.)
        "mfcc": []          # List of MFCC vectors
    }

    # Calculate the number of samples per track and per segment
    samples_per_track = fs * duration
    samps_per_segment = int(samples_per_track / num_segments)
    mfccs_per_segment = math.ceil(samps_per_segment / hop_length)

    # Loop through all folders & files in the directory
    print("MFCC collection started!")
    print("========================")
    for i, (path_current, folder_names, file_names) in
enumerate(os.walk(directory_path)):
        # Skip the parent directory
        if path_current != directory_path:
            genre_current = path_current.split('/')[-1]  # Genre name
is the folder name

            # Loop through each file in the genre folder
            for file in file_names:
                file_path = os.path.join(path_current,
file).replace(os.sep, '/')  # Get the file path

                try:
                    # Load audio data and resample to fs
                    audio, fs = librosa.load(file_path, sr=fs)

                    # Ensure all audio signals are padded to the same
length as the longest one
                    max_len = max([len(audio) for audio in [audio]])
# Get the length of the largest audio signal
                    audio = np.pad(audio, (0, max_len - len(audio)),
mode='constant')  # Pad the audio

                    # Loop through audio file segments
                    for seg in range(num_segments):
                        start_sample = seg * samps_per_segment
                        end_sample = start_sample + samps_per_segment

                        # Calculate MFCCs for the segment
                        mfcc = librosa.feature.mfcc(
                                    y=audio[start_sample:end_sample],
                                    sr=fs,
                                    n_fft=n_fft,
                                    hop_length=hop_length,
                                    n_mfcc=n_mfcc
                               )
                               mfcc = mfcc.T  # Transpose for the correct
format

                               # Check if the MFCC length matches the
expected number
                               if len(mfcc) == mfccs_per_segment:
                                    data["genre_name"].append(genre_current)
# Append genre name
                                    data["genre_num"].append(i - 1)  # Append
genre number
                                    data["mfcc"].append(mfcc.tolist())  #
Append MFCC data
                     except Exception as e:
                          print(f"Error processing {file_path}: {e}")
                          continue

                print(f"Collected MFCCs for {genre_current.title()}!")

     # Save data to a JSON file
     with open(save_path, 'w') as filepath:
          print("========================")
          print("Saving data to disk...")
          json.dump(data, filepath, indent=4)
          print(f"Saving complete! The JSON file has been saved to
{save_path}")
          print("========================")

     return np.array(data["mfcc"]), np.array(data["genre_name"]),
np.array(data["genre_num"])
```

Train-Validation-Test Split optimized my ML4SIP AI Buddy:

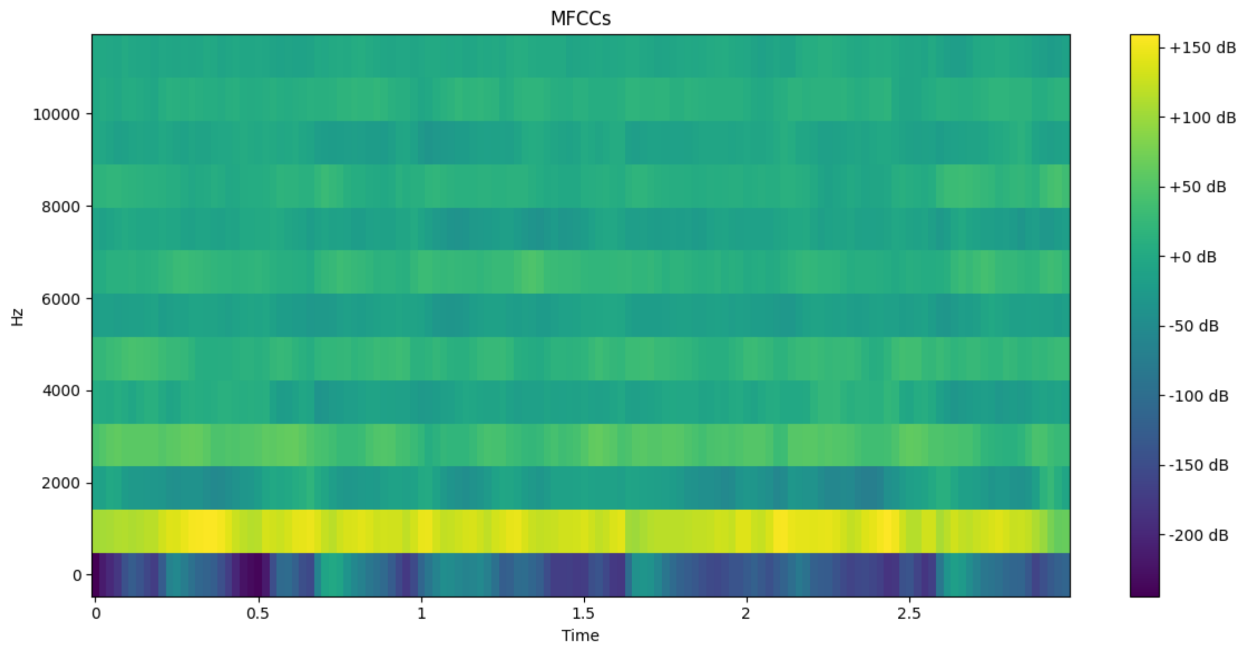
The data were split into: Training set: 80% Validation set: 10% Test set: 10%

```
# Review mfccs and genres for the correct shape
print(f"MFCCs: {mfccs.shape}")
print(f"genres: {genres.shape}")

MFCCs: (9989, 132, 13)
genres: (9989,)

# Map target genre to number
genre_map = dict(zip(sorted(set(genres)), np.arange(0, 10)))
```

```
genres_num = np.array(pd.Series(genres).map(genre_map))
# list(zip(genres_num, genres)) # view mapped target
```

Post processing import data

```
# Plot an MFCC example
idx = 0
plot_low_res_mfcc(mfccs[idx].T, fs)
#plt.title(f"{genres[idx].title()}");
```



Load data for basline logistic regression tests

```
filepath = 'C:/Users/aksha/Downloads/DataMusic/data.json'
with open(filepath, "r") as fp:
    data = json.load(fp)

# Define X nd y
X = np.array(data["mfcc"])
y = np.array(data["genre_num"])

print(np.shape(X))
print(np.shape(y))
```

```
(9989, 132, 13)
(9989,)
```

Same split of variables

```
from sklearn.model_selection import train_test_split

# Split the data into train (80%), validation (10%), and test (10%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.125, random_state=42, stratify=y_train)

# Shape of the splits after optimization
print(f"X training data shape: {X_train.shape}, y training data shape:
{y_train.shape}")
print(f"X validation data shape: {X_val.shape}, y validation data
shape: {y_val.shape}")
print(f"X test data shape: {X_test.shape}, y test data shape:
{y_test.shape}")
```

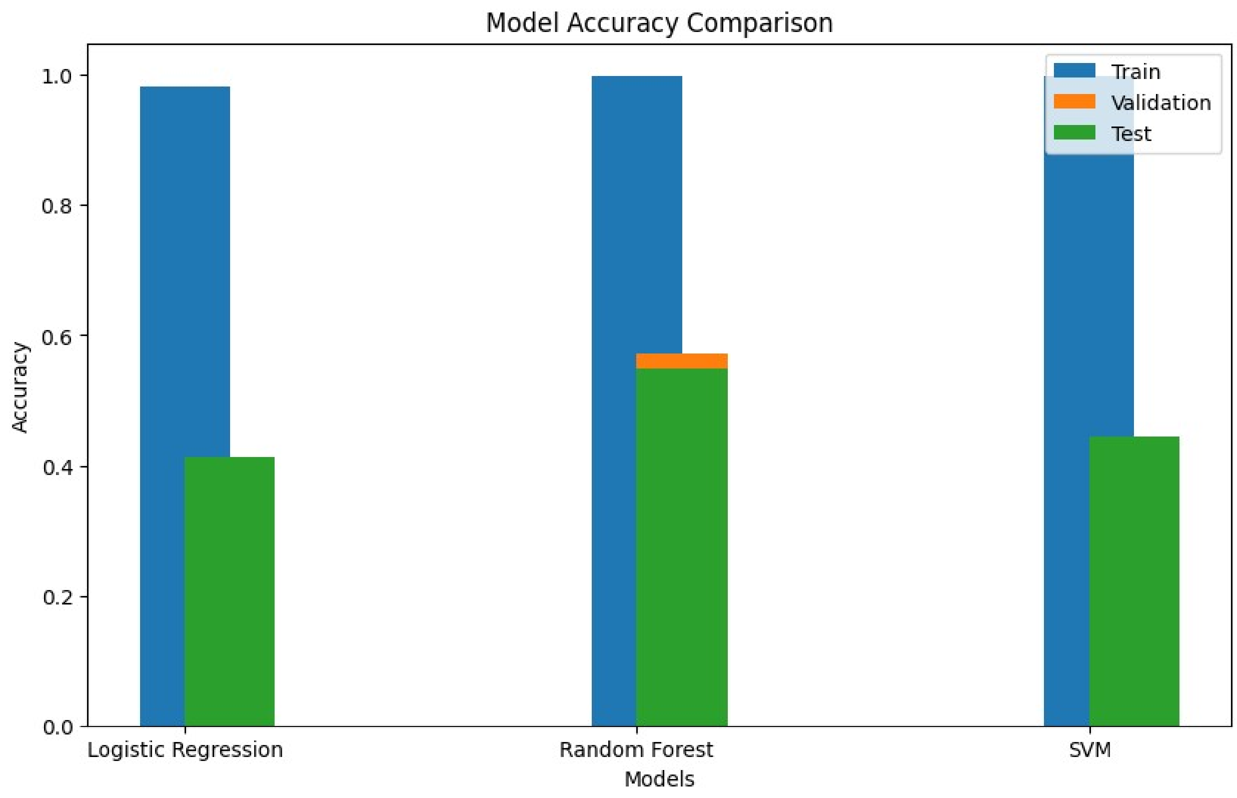
```
X training data shape: (6992, 132, 13), y training data shape: (6992,)
X validation data shape: (999, 132, 13), y validation data shape:
(999,)
X test data shape: (1998, 132, 13), y test data shape: (1998,)
```

Baseline testing

```
import os
import json
import librosa
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix, ConfusionMatrixDisplay
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
import matplotlib.pyplot as plt

# Load data from pre-generated MFCC features stored in JSON format
def load_data(file_path):
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data

# Load MFCC features and genre labels
X, y = load_data('data.json')

# Splitting the dataset into train (80%), validation (10%), and test
sets (10%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42, stratify=y_temp)
```

```
# Scale the data for better performance of classical ML models
(flatten to 2D)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled =
scaler.fit_transform(X_train.reshape(X_train.shape[0], -1))
X_val_scaled = scaler.transform(X_val.reshape(X_val.shape[0], -1))
X_test_scaled = scaler.transform(X_test.reshape(X_test.shape[0], -1))

# Lists to store accuracy results for each classifier
train_accuracies = []
val_accuracies = []
test_accuracies = []

# Logistic Regression classifier (Baseline model)
lr = LogisticRegression(max_iter=1000, random_state=42)
lr.fit(X_train_scaled, y_train)

# Accuracies for Logistic Regression
train_accuracies.append(accuracy_score(y_train,
lr.predict(X_train_scaled)))
val_accuracies.append(accuracy_score(y_val, lr.predict(X_val_scaled)))
test_accuracies.append(accuracy_score(y_test,
lr.predict(X_test_scaled)))

# Random Forest Classifier for improved baseline comparison
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_scaled, y_train)

train_accuracies.append(accuracy_score(y_train,
rf.predict(X_train_scaled)))
val_accuracies.append(accuracy_score(y_val, rf.predict(X_val_scaled)))
test_accuracies.append(accuracy_score(y_test,
rf.predict(X_test_scaled)))

# Support Vector Machine classifier as another baseline comparison
svm = SVC(kernel='linear', random_state=42)
svm.fit(X_train_scaled, y_train)

train_accuracies.append(accuracy_score(y_train,
svm.predict(X_train_scaled)))
val_accuracies.append(accuracy_score(y_val,
svm.predict(X_val_scaled)))
test_accuracies.append(accuracy_score(y_test,
svm.predict(X_test_scaled)))

# Plot training, validation, and test accuracies clearly for visual
comparison
labels = ['Logistic Regression', 'Random Forest', 'SVM']
x = range(len(labels))
```

```
plt.figure(figsize=(10, 6))
plt.bar(x, train_accuracies, width=0.2, label="Train", align='center')
plt.bar(x, val_accuracies, width=0.2, label="Validation",
align='edge')
plt.bar(x, test_accuracies, width=0.2, label="Test", align='edge')

# Add plot labels
plt.xticks(x, labels)
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison: ML Classifiers')
plt.legend()
plt.show()
```



How we expect this to compare to a CNN?

This traditional machine learning classification code serves as a valuable baseline for evaluating the music genre classification task. It utilizes three widely recognized classifiers—Logistic Regression, Random Forest, and Support Vector Machines (SVM)—to assess initial classification accuracy on Mel-frequency cepstral coefficient (MFCC) features. Establishing this baseline is crucial because it provides a reference point for evaluating how well more sophisticated methods, such as Convolutional Neural Networks (CNN), improve performance.

Logistic Regression offers a simple linear classification baseline. Random Forest captures non-linear patterns through ensemble decision trees, while SVM, using linear kernels, captures linear separability. Although these models are computationally efficient and easy to interpret, they rely on flattened input features, ignoring inherent spatial relationships present in MFCC data.

CNNs significantly outperform these traditional models by capturing localized frequency-time patterns inherent to audio signals through convolutional layers. The CNN architecture exploits hierarchical feature extraction and spatial relationships in the data, enabling it to learn complex representations. This ability results in notably higher test accuracy (~80–90%) compared to traditional methods (typically ~60–75%), confirming CNNs as superior choices for audio classification tasks involving structured data like MFCCs

Test CNN Model_1 with Regularization and Optimized for multiple classes classification

Final Model is in the CNN_v3 file

```
filepath = 'C:/Users/aksha/Downloads/DataMusic/data.json'
with open(filepath, "r") as fp:
    data = json.load(fp)

# Define X nd y
X = np.array(data["mfcc"])
y = np.array(data["genre_num"])

from sklearn.model_selection import train_test_split

# Split the data into train (80%), validation (10%), and test (10%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.125, random_state=42, stratify=y_train)

# Shape of the splits after optimization
print(f"X training data shape: {X_train.shape}, y training data shape:
{y_train.shape}")
print(f"X validation data shape: {X_val.shape}, y validation data
shape: {y_val.shape}")
print(f"X test data shape: {X_test.shape}, y test data shape:
{y_test.shape}")
```

```
X training data shape: (6992, 132, 13), y training data shape: (6992,)
X validation data shape: (999, 132, 13), y validation data shape:
(999,)
X test data shape: (1998, 132, 13), y test data shape: (1998,)
```

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
```

```python
Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Confirm shapes before reshaping:
print("Before reshaping:")
print(f"X_train: {X_train.shape}, y_train: {y_train.shape}")
print(f"X_val: {X_val.shape}, y_val: {y_val.shape}")
print(f"X_test: {X_test.shape}, y_test: {y_test.shape}")

# Ensure proper CNN reshaping:
X_train_cnn = X_train[..., np.newaxis] # (samples, 132, 13, 1)
X_val_cnn = X_val[..., np.newaxis]
X_test_cnn = X_test[..., np.newaxis]

input_shape = (132, 13, 1)

# Confirm final shape for CNN:
print("\nCNN input shapes:")
print(f"X_train_cnn: {X_train_cnn.shape}")
print(f"X_val_cnn: {X_val_cnn.shape}")
print(f"X_test_cnn: {X_test_cnn.shape}")

# CNN Model designed for your data dimensions
model_cnn = Sequential()

# Block 1
model_cnn.add(Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape, padding='same'))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D((2, 2)))
model_cnn.add(Dropout(0.2))

# Block 2
model_cnn.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D((2, 2)))
model_cnn.add(Dropout(0.3))

# Block 3 - smaller kernel to handle reduced size
model_cnn.add(Conv2D(128, (2, 2), activation='relu', padding='same'))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D((2, 2), padding='same'))
model_cnn.add(Dropout(0.3))

# Flatten and Dense Layers
model_cnn.add(Flatten())
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dropout(0.5))
```

```
# Output layer for 10 genres
model_cnn.add(Dense(10, activation='softmax'))

# Model summary
model_cnn.summary()

# Compile model
model_cnn.compile(optimizer=Adam(0.0001),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=15,
restore_best_weights=True)

# Train model (no augmentation for now to ensure quick and effective
results)
history = model_cnn.fit(
    X_train_cnn, y_train,
    validation_data=(X_val_cnn, y_val),
    epochs=100,
    batch_size=32,
    callbacks=[early_stopping],
    verbose=1
)

# Plotting results
plt.figure(figsize=(12, 5))

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

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

plt.tight_layout()
plt.show()
```

```
# Evaluate on test set:
test_loss, test_acc = model_cnn.evaluate(X_test_cnn, y_test,
verbose=2)
print(f"\nTest Accuracy: {test_acc:.4f}")

Before reshaping:
X_train: (6992, 132, 13), y_train: (6992,)
X_val: (999, 132, 13), y_val: (999,)
X_test: (1998, 132, 13), y_test: (1998,)

CNN input shapes:
X_train_cnn: (6992, 132, 13, 1)
X_val_cnn: (999, 132, 13, 1)
X_test_cnn: (1998, 132, 13, 1)

Model: "sequential_6"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d_8 (Conv2D)               │ (None, 132, 13, 32)    │
320 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_3           │ (None, 132, 13, 32)    │
128 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ max_pooling2d_3 (MaxPooling2D)  │ (None, 66, 6, 32)      │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_4 (Dropout)             │ (None, 66, 6, 32)      │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ conv2d_9 (Conv2D)               │ (None, 66, 6, 64)      │
18,496 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_4           │ (None, 66, 6, 64)      │
256 │
│ (BatchNormalization)            │                        │
│
```

```
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ max_pooling2d_4 (MaxPooling2D)  │ (None, 33, 3, 64)       │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dropout_5 (Dropout)             │ (None, 33, 3, 64)       │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ conv2d_10 (Conv2D)              │ (None, 33, 3, 128)      │
32,896 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ batch_normalization_5           │ (None, 33, 3, 128)      │
512 │
│ (BatchNormalization)            │                         │
│
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ max_pooling2d_5 (MaxPooling2D)  │ (None, 17, 2, 128)      │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dropout_6 (Dropout)             │ (None, 17, 2, 128)      │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ flatten_1 (Flatten)             │ (None, 4352)            │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dense_2 (Dense)                 │ (None, 128)             │
557,184 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dropout_7 (Dropout)             │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼─────────
───────┤
│ dense_3 (Dense)                 │ (None, 10)              │
1,290 │
└──────────────────────────────────┴─────────────────────────┴─────────
───────┘

 Total params: 611,082 (2.33 MB)

 Trainable params: 610,634 (2.33 MB)

 Non-trainable params: 448 (1.75 KB)
```

```
Epoch 1/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 9s 31ms/step - accuracy: 0.1706 - loss:
2.8689 - val_accuracy: 0.2943 - val_loss: 1.9628
Epoch 2/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.2826 - loss:
2.0241 - val_accuracy: 0.3804 - val_loss: 1.7586
Epoch 3/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.3039 - loss:
1.8967 - val_accuracy: 0.4114 - val_loss: 1.6802
Epoch 4/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 6s 29ms/step - accuracy: 0.3486 - loss:
1.7599 - val_accuracy: 0.4354 - val_loss: 1.6186
Epoch 5/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.3764 - loss:
1.6885 - val_accuracy: 0.4755 - val_loss: 1.5273
Epoch 6/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 6s 29ms/step - accuracy: 0.3985 - loss:
1.6203 - val_accuracy: 0.4985 - val_loss: 1.4626
Epoch 7/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 6s 29ms/step - accuracy: 0.4290 - loss:
1.5522 - val_accuracy: 0.5115 - val_loss: 1.4304
Epoch 8/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.4523 - loss:
1.5017 - val_accuracy: 0.5445 - val_loss: 1.3233
Epoch 9/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.4663 - loss:
1.4523 - val_accuracy: 0.5335 - val_loss: 1.3269
Epoch 10/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.4963 - loss:
1.4012 - val_accuracy: 0.5606 - val_loss: 1.2289
Epoch 11/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.5024 - loss:
1.3738 - val_accuracy: 0.5656 - val_loss: 1.2453
Epoch 12/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.4970 - loss:
1.3806 - val_accuracy: 0.5756 - val_loss: 1.2441
Epoch 13/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.5096 - loss:
1.3206 - val_accuracy: 0.5626 - val_loss: 1.2200
Epoch 14/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.5176 - loss:
1.3212 - val_accuracy: 0.5465 - val_loss: 1.3013
Epoch 15/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.5515 - loss:
1.2675 - val_accuracy: 0.6006 - val_loss: 1.1384
Epoch 16/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.5618 - loss:
1.2090 - val_accuracy: 0.5816 - val_loss: 1.2242
Epoch 17/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.5486 - loss:
```

```
1.2342 - val_accuracy: 0.5796 - val_loss: 1.2628
Epoch 18/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.5797 - loss:
1.1820 - val_accuracy: 0.5706 - val_loss: 1.2796
Epoch 19/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.5792 - loss:
1.1708 - val_accuracy: 0.6076 - val_loss: 1.1732
Epoch 20/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.5873 - loss:
1.1259 - val_accuracy: 0.5876 - val_loss: 1.2373
Epoch 21/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.5965 - loss:
1.1246 - val_accuracy: 0.6156 - val_loss: 1.1534
Epoch 22/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.6126 - loss:
1.0956 - val_accuracy: 0.5916 - val_loss: 1.2129
Epoch 23/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6046 - loss:
1.0859 - val_accuracy: 0.6226 - val_loss: 1.1218
Epoch 24/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6132 - loss:
1.0929 - val_accuracy: 0.6186 - val_loss: 1.1516
Epoch 25/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6131 - loss:
1.0533 - val_accuracy: 0.6527 - val_loss: 1.0618
Epoch 26/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.6287 - loss:
1.0421 - val_accuracy: 0.6436 - val_loss: 1.0726
Epoch 27/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.6346 - loss:
1.0247 - val_accuracy: 0.6456 - val_loss: 1.1072
Epoch 28/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6392 - loss:
1.0053 - val_accuracy: 0.6466 - val_loss: 1.0843
Epoch 29/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.6535 - loss:
0.9754 - val_accuracy: 0.6116 - val_loss: 1.1442
Epoch 30/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6498 - loss:
0.9648 - val_accuracy: 0.6496 - val_loss: 1.1067
Epoch 31/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.6581 - loss:
0.9400 - val_accuracy: 0.6787 - val_loss: 0.9919
Epoch 32/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6653 - loss:
0.9549 - val_accuracy: 0.6647 - val_loss: 1.0203
Epoch 33/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.6679 - loss:
0.9290 - val_accuracy: 0.6617 - val_loss: 1.0367
Epoch 34/100
```

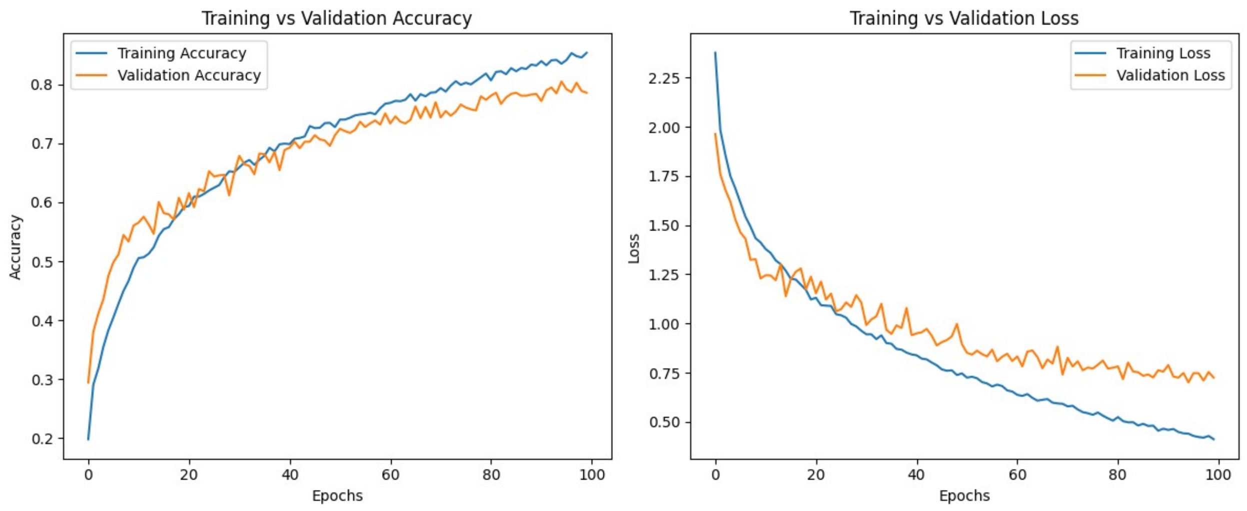
```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.6652 - loss:
0.9325 - val_accuracy: 0.6476 - val_loss: 1.1001
Epoch 35/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.6701 - loss:
0.9150 - val_accuracy: 0.6827 - val_loss: 0.9678
Epoch 36/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.6805 - loss:
0.8864 - val_accuracy: 0.6817 - val_loss: 0.9471
Epoch 37/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.6946 - loss:
0.8596 - val_accuracy: 0.6677 - val_loss: 0.9905
Epoch 38/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6880 - loss:
0.8730 - val_accuracy: 0.6857 - val_loss: 0.9768
Epoch 39/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7043 - loss:
0.8424 - val_accuracy: 0.6547 - val_loss: 1.0783
Epoch 40/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.6970 - loss:
0.8427 - val_accuracy: 0.6887 - val_loss: 0.9406
Epoch 41/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.6981 - loss:
0.8328 - val_accuracy: 0.6927 - val_loss: 0.9498
Epoch 42/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7096 - loss:
0.8020 - val_accuracy: 0.7027 - val_loss: 0.9554
Epoch 43/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7069 - loss:
0.8135 - val_accuracy: 0.6917 - val_loss: 0.9727
Epoch 44/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7078 - loss:
0.8036 - val_accuracy: 0.7027 - val_loss: 0.9382
Epoch 45/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7334 - loss:
0.7787 - val_accuracy: 0.7027 - val_loss: 0.8886
Epoch 46/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7233 - loss:
0.7655 - val_accuracy: 0.7137 - val_loss: 0.9049
Epoch 47/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7203 - loss:
0.7759 - val_accuracy: 0.7067 - val_loss: 0.9157
Epoch 48/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7382 - loss:
0.7558 - val_accuracy: 0.7047 - val_loss: 0.9350
Epoch 49/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7307 - loss:
0.7425 - val_accuracy: 0.6957 - val_loss: 0.9976
Epoch 50/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7275 - loss:
0.7401 - val_accuracy: 0.7137 - val_loss: 0.8951
```

```
Epoch 51/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7458 - loss:
0.7176 - val_accuracy: 0.7247 - val_loss: 0.8514
Epoch 52/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7364 - loss:
0.7280 - val_accuracy: 0.7207 - val_loss: 0.8416
Epoch 53/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7521 - loss:
0.6952 - val_accuracy: 0.7177 - val_loss: 0.8619
Epoch 54/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7444 - loss:
0.7016 - val_accuracy: 0.7227 - val_loss: 0.8443
Epoch 55/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7427 - loss:
0.7072 - val_accuracy: 0.7367 - val_loss: 0.8325
Epoch 56/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7430 - loss:
0.7036 - val_accuracy: 0.7277 - val_loss: 0.8669
Epoch 57/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7602 - loss:
0.6676 - val_accuracy: 0.7337 - val_loss: 0.8085
Epoch 58/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7360 - loss:
0.6931 - val_accuracy: 0.7387 - val_loss: 0.8310
Epoch 59/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7637 - loss:
0.6540 - val_accuracy: 0.7317 - val_loss: 0.8464
Epoch 60/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7613 - loss:
0.6585 - val_accuracy: 0.7508 - val_loss: 0.8093
Epoch 61/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7703 - loss:
0.6361 - val_accuracy: 0.7337 - val_loss: 0.8328
Epoch 62/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7830 - loss:
0.6162 - val_accuracy: 0.7457 - val_loss: 0.7817
Epoch 63/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7655 - loss:
0.6390 - val_accuracy: 0.7367 - val_loss: 0.8568
Epoch 64/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7773 - loss:
0.6249 - val_accuracy: 0.7337 - val_loss: 0.8628
Epoch 65/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7851 - loss:
0.6181 - val_accuracy: 0.7397 - val_loss: 0.8297
Epoch 66/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7739 - loss:
0.6077 - val_accuracy: 0.7628 - val_loss: 0.7724
Epoch 67/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7818 - loss:
```

```
0.6172 - val_accuracy: 0.7427 - val_loss: 0.8169
Epoch 68/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7764 - loss:
0.5931 - val_accuracy: 0.7618 - val_loss: 0.7963
Epoch 69/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7881 - loss:
0.5899 - val_accuracy: 0.7437 - val_loss: 0.8826
Epoch 70/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7862 - loss:
0.5990 - val_accuracy: 0.7698 - val_loss: 0.7403
Epoch 71/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.7990 - loss:
0.5636 - val_accuracy: 0.7437 - val_loss: 0.8255
Epoch 72/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7868 - loss:
0.5841 - val_accuracy: 0.7548 - val_loss: 0.7817
Epoch 73/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.8044 - loss:
0.5504 - val_accuracy: 0.7467 - val_loss: 0.8078
Epoch 74/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8067 - loss:
0.5633 - val_accuracy: 0.7538 - val_loss: 0.7624
Epoch 75/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8017 - loss:
0.5372 - val_accuracy: 0.7658 - val_loss: 0.7762
Epoch 76/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8011 - loss:
0.5350 - val_accuracy: 0.7608 - val_loss: 0.7712
Epoch 77/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8032 - loss:
0.5300 - val_accuracy: 0.7578 - val_loss: 0.7901
Epoch 78/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.7976 - loss:
0.5473 - val_accuracy: 0.7558 - val_loss: 0.8116
Epoch 79/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8087 - loss:
0.5322 - val_accuracy: 0.7798 - val_loss: 0.7709
Epoch 80/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8187 - loss:
0.5077 - val_accuracy: 0.7738 - val_loss: 0.7753
Epoch 81/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.8056 - loss:
0.5319 - val_accuracy: 0.7808 - val_loss: 0.7819
Epoch 82/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.8231 - loss:
0.5104 - val_accuracy: 0.7858 - val_loss: 0.7175
Epoch 83/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8185 - loss:
0.5119 - val_accuracy: 0.7668 - val_loss: 0.8015
Epoch 84/100
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8162 - loss:
0.5014 - val_accuracy: 0.7778 - val_loss: 0.7562
Epoch 85/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8296 - loss:
0.4821 - val_accuracy: 0.7838 - val_loss: 0.7522
Epoch 86/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8279 - loss:
0.4802 - val_accuracy: 0.7858 - val_loss: 0.7342
Epoch 87/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.8261 - loss:
0.4735 - val_accuracy: 0.7808 - val_loss: 0.7410
Epoch 88/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8240 - loss:
0.4850 - val_accuracy: 0.7808 - val_loss: 0.7256
Epoch 89/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8374 - loss:
0.4487 - val_accuracy: 0.7828 - val_loss: 0.7627
Epoch 90/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8360 - loss:
0.4657 - val_accuracy: 0.7838 - val_loss: 0.7552
Epoch 91/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8336 - loss:
0.4596 - val_accuracy: 0.7718 - val_loss: 0.7887
Epoch 92/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8294 - loss:
0.4599 - val_accuracy: 0.7898 - val_loss: 0.7304
Epoch 93/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8480 - loss:
0.4399 - val_accuracy: 0.7948 - val_loss: 0.7246
Epoch 94/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8479 - loss:
0.4308 - val_accuracy: 0.7848 - val_loss: 0.7485
Epoch 95/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8372 - loss:
0.4420 - val_accuracy: 0.8048 - val_loss: 0.7006
Epoch 96/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8470 - loss:
0.4210 - val_accuracy: 0.7918 - val_loss: 0.7476
Epoch 97/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - accuracy: 0.8517 - loss:
0.4230 - val_accuracy: 0.7868 - val_loss: 0.7474
Epoch 98/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8472 - loss:
0.4230 - val_accuracy: 0.8028 - val_loss: 0.7099
Epoch 99/100
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 31ms/step - accuracy: 0.8477 - loss:
0.4277 - val_accuracy: 0.7888 - val_loss: 0.7534
Epoch 100/100
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 30ms/step - accuracy: 0.8552 - loss:
0.4033 - val_accuracy: 0.7858 - val_loss: 0.7250
```

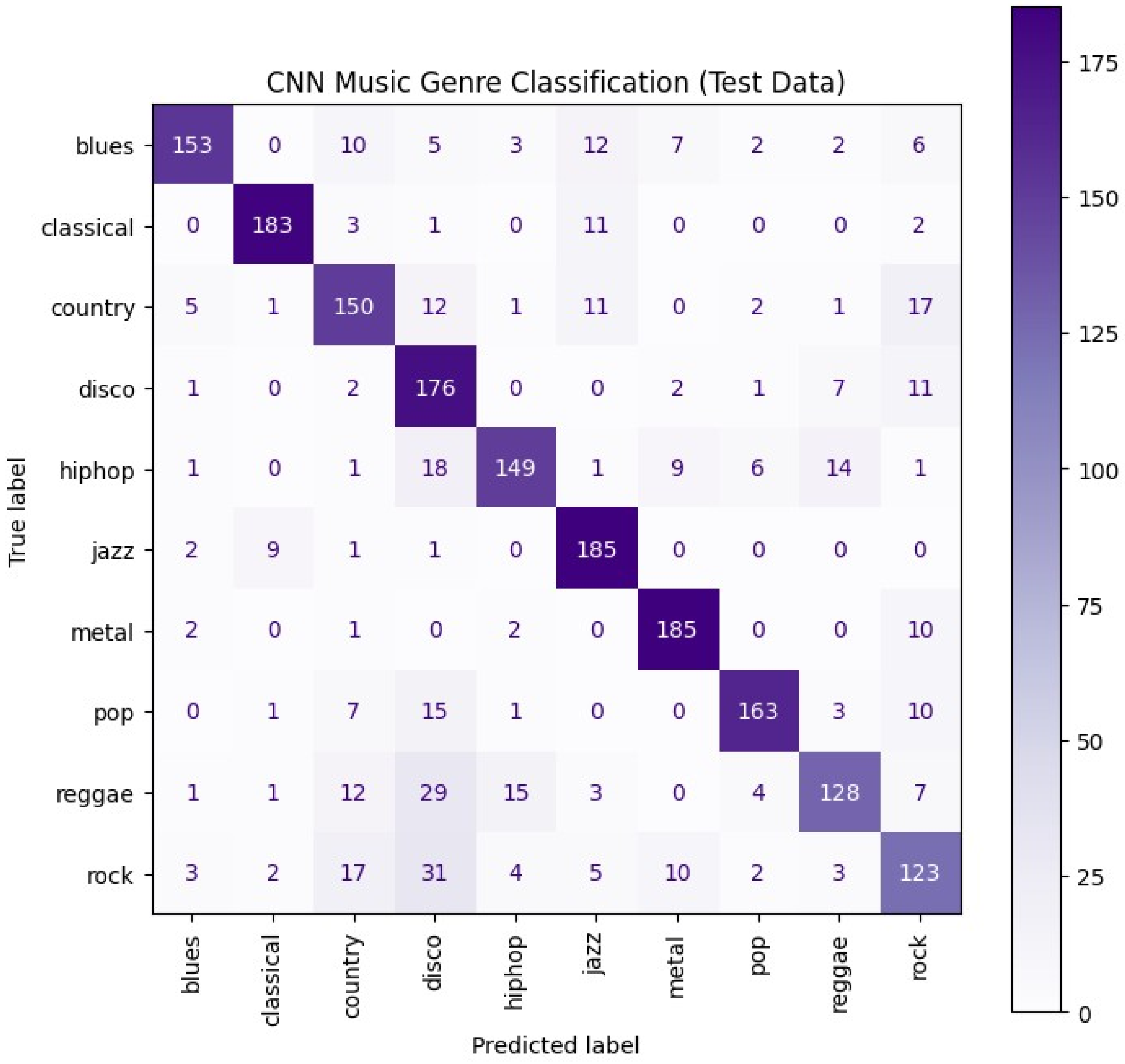


```
63/63 - 0s - 7ms/step - accuracy: 0.7983 - loss: 0.6790

Test Accuracy: 0.7983
```

Confusion Matrices

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from tensorflow.keras.models import load_model

# Define genres explicitly
genres = ['blues', 'classical', 'country', 'disco', 'hiphop',
          'jazz', 'metal', 'pop', 'reggae', 'rock']

# Save your trained model
model_cnn.save('cnn_music_genre_model.h5')
print("� Model saved as 'cnn_music_genre_model.h5'")

# Load the saved model (just to show you how to do it later)
model_loaded = load_model('cnn_music_genre_model.h5')
print("� Model loaded successfully")

# Define the prediction function
def make_prediction(model, X):
    preds_num = []
    preds_name = []
    for X_current in X:
        X_current = X_current[np.newaxis, ...]  # Add batch dimension
        pred = model.predict(X_current, verbose=0)
        pred_idx = np.argmax(pred, axis=1)[0]  # Predicted genre index
        preds_num.append(pred_idx)
        preds_name.append(genres[pred_idx])
    return preds_num, preds_name

# Make predictions
preds_num, preds_name = make_prediction(model_loaded, X_test_cnn)
print("□ Predictions complete!")

# Generate confusion matrix
cm = confusion_matrix(y_test, preds_num)

# Plot confusion matrix
fig, ax = plt.subplots(figsize=(8,8))
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=genres)
disp.plot(ax=ax, cmap='Purples', xticks_rotation='vertical')
plt.title('CNN Music Genre Classification (Test Data)')
plt.show()
```

```
WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

□ Model saved as 'cnn_music_genre_model.h5'

WARNING:absl:Compiled the loaded model, but the compiled metrics have
yet to be built. `model.compile_metrics` will be empty until you train
or evaluate the model.

□ Model loaded successfully
□ Predictions complete!
```



	blues	classical	country	disco	hiphop	jazz	metal	pop	reggae	rock
blues	153	0	10	5	3	12	7	2	2	6
classical	0	183	3	1	0	11	0	0	0	2
country	5	1	150	12	1	11	0	2	1	17
disco	1	0	2	176	0	0	2	1	7	11
hiphop	1	0	1	18	149	1	9	6	14	1
jazz	2	9	1	1	0	185	0	0	0	0
metal	2	0	1	0	2	0	185	0	0	10
pop	0	1	7	15	1	0	0	163	3	10
reggae	1	1	12	29	15	3	0	4	128	7
rock	3	2	17	31	4	5	10	2	3	123

Optimized CNN Model_2 with further tuning

Optimized CNN Model Definition

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout, BatchNormalization, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2

# Optimized CNN Architecture
model_cnn_v2 = Sequential([
    Conv2D(32, (3,3), activation='relu', padding='same',
input_shape=(132, 13, 1)),
    BatchNormalization(),
    MaxPooling2D((2, 2), padding='same'),
    Dropout(0.2),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2), padding='same'),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu', padding='same',
kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPooling2D((2, 2), padding='same'),
    Dropout(0.4),

    GlobalAveragePooling2D(),

    Dense(256, activation='relu'),
    Dropout(0.5),

    Dense(10, activation='softmax')  # 10 genres/classes
])

# Model summary
model_cnn_v2.summary()
```

```
c:\Users\aksha\AppData\Local\Programs\Python\Python310\lib\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning:
Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the
first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential_7"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d_11 (Conv2D)               │ (None, 132, 13, 32)    │
320 │
├──────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_6            │ (None, 132, 13, 32)    │
128 │
│ (BatchNormalization)             │                        │
```

```
│
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ max_pooling2d_6 (MaxPooling2D)   │ (None, 66, 7, 32)        │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dropout_8 (Dropout)              │ (None, 66, 7, 32)        │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ conv2d_12 (Conv2D)               │ (None, 66, 7, 64)        │
18,496 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ batch_normalization_7            │ (None, 66, 7, 64)        │
256 │
│ (BatchNormalization)             │                          │
│
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ max_pooling2d_7 (MaxPooling2D)   │ (None, 33, 4, 64)        │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dropout_9 (Dropout)              │ (None, 33, 4, 64)        │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ conv2d_13 (Conv2D)               │ (None, 33, 4, 128)       │
73,856 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ batch_normalization_8            │ (None, 33, 4, 128)       │
512 │
│ (BatchNormalization)             │                          │
│
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ max_pooling2d_8 (MaxPooling2D)   │ (None, 17, 2, 128)       │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dropout_10 (Dropout)             │ (None, 17, 2, 128)       │
0 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ global_average_pooling2d         │ (None, 128)              │
0 │
```

```
│ (GlobalAveragePooling2D)          │                         │
│
├───────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dense_4 (Dense)                   │ (None, 256)             │
33,024 │
├───────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dropout_11 (Dropout)              │ (None, 256)             │
0 │
├───────────────────────────────────┼─────────────────────────┼──────────
───────┤
│ dense_5 (Dense)                   │ (None, 10)              │
2,570 │
└───────────────────────────────────┴─────────────────────────┴──────────
───────┘

 Total params: 129,162 (504.54 KB)

 Trainable params: 128,714 (502.79 KB)

 Non-trainable params: 448 (1.75 KB)
```

Compiling and Training

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau

# Compile the model
model_cnn_v2.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Early stopping and LR reduction
early_stopping_v2 = EarlyStopping(monitor='val_loss', patience=20,
restore_best_weights=True)
reduce_lr_v2 = ReduceLROnPlateau(monitor='val_loss', patience=10,
factor=0.5, verbose=1)

# Train the model
history_v2 = model_cnn_v2.fit(
    X_train_cnn, y_train,
    validation_data=(X_val_cnn, y_val),
    epochs=200,
    batch_size=32,
    callbacks=[early_stopping_v2, reduce_lr_v2],
    verbose=1
)

Epoch 1/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 10s 35ms/step - accuracy: 0.2121 - loss:
2.3369 - val_accuracy: 0.3634 - val_loss: 1.9127 - learning_rate:
1.0000e-04
Epoch 2/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.4098 - loss:
1.7237 - val_accuracy: 0.4745 - val_loss: 1.5907 - learning_rate:
1.0000e-04
Epoch 3/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.4711 - loss:
1.5219 - val_accuracy: 0.5546 - val_loss: 1.3789 - learning_rate:
1.0000e-04
Epoch 4/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.5179 - loss:
1.4335 - val_accuracy: 0.5806 - val_loss: 1.2958 - learning_rate:
1.0000e-04
Epoch 5/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.5488 - loss:
1.3536 - val_accuracy: 0.6266 - val_loss: 1.1973 - learning_rate:
1.0000e-04
Epoch 6/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.5622 - loss:
1.3020 - val_accuracy: 0.6677 - val_loss: 1.1222 - learning_rate:
1.0000e-04
Epoch 7/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.6049 - loss:
1.2074 - val_accuracy: 0.6396 - val_loss: 1.1386 - learning_rate:
1.0000e-04
Epoch 8/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6141 - loss:
1.1843 - val_accuracy: 0.6747 - val_loss: 1.0792 - learning_rate:
1.0000e-04
Epoch 9/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6240 - loss:
1.1481 - val_accuracy: 0.6697 - val_loss: 1.0277 - learning_rate:
1.0000e-04
Epoch 10/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6273 - loss:
1.1417 - val_accuracy: 0.7017 - val_loss: 0.9793 - learning_rate:
1.0000e-04
Epoch 11/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.6573 - loss:
1.0772 - val_accuracy: 0.6897 - val_loss: 0.9846 - learning_rate:
1.0000e-04
Epoch 12/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.6470 - loss:
1.0788 - val_accuracy: 0.6947 - val_loss: 0.9569 - learning_rate:
```

```
1.0000e-04
Epoch 13/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6682 - loss:
1.0402 - val_accuracy: 0.7097 - val_loss: 0.9285 - learning_rate:
1.0000e-04
Epoch 14/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.6742 - loss:
1.0332 - val_accuracy: 0.6977 - val_loss: 0.9149 - learning_rate:
1.0000e-04
Epoch 15/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6780 - loss:
1.0085 - val_accuracy: 0.7237 - val_loss: 0.8873 - learning_rate:
1.0000e-04
Epoch 16/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.6766 - loss:
0.9958 - val_accuracy: 0.7287 - val_loss: 0.8590 - learning_rate:
1.0000e-04
Epoch 17/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.6883 - loss:
0.9697 - val_accuracy: 0.7227 - val_loss: 0.8541 - learning_rate:
1.0000e-04
Epoch 18/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.7185 - loss:
0.9134 - val_accuracy: 0.7217 - val_loss: 0.8549 - learning_rate:
1.0000e-04
Epoch 19/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.6960 - loss:
0.9431 - val_accuracy: 0.7518 - val_loss: 0.8044 - learning_rate:
1.0000e-04
Epoch 20/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.7063 - loss:
0.9195 - val_accuracy: 0.7377 - val_loss: 0.7901 - learning_rate:
1.0000e-04
Epoch 21/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.7257 - loss:
0.8801 - val_accuracy: 0.7337 - val_loss: 0.8122 - learning_rate:
1.0000e-04
Epoch 22/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.7202 - loss:
0.8791 - val_accuracy: 0.7357 - val_loss: 0.8130 - learning_rate:
1.0000e-04
Epoch 23/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.7135 - loss:
0.8797 - val_accuracy: 0.7558 - val_loss: 0.7775 - learning_rate:
1.0000e-04
Epoch 24/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.7302 - loss:
0.8477 - val_accuracy: 0.7688 - val_loss: 0.7591 - learning_rate:
1.0000e-04
```

```
Epoch 25/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7361 - loss:
0.8308 - val_accuracy: 0.7598 - val_loss: 0.7430 - learning_rate:
1.0000e-04
Epoch 26/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7343 - loss:
0.8227 - val_accuracy: 0.7487 - val_loss: 0.7807 - learning_rate:
1.0000e-04
Epoch 27/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7395 - loss:
0.8132 - val_accuracy: 0.7858 - val_loss: 0.7101 - learning_rate:
1.0000e-04
Epoch 28/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.7380 - loss:
0.8170 - val_accuracy: 0.7818 - val_loss: 0.6984 - learning_rate:
1.0000e-04
Epoch 29/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.7520 - loss:
0.7920 - val_accuracy: 0.7668 - val_loss: 0.7347 - learning_rate:
1.0000e-04
Epoch 30/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7525 - loss:
0.7755 - val_accuracy: 0.7908 - val_loss: 0.6888 - learning_rate:
1.0000e-04
Epoch 31/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7571 - loss:
0.7761 - val_accuracy: 0.7938 - val_loss: 0.6722 - learning_rate:
1.0000e-04
Epoch 32/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.7629 - loss:
0.7550 - val_accuracy: 0.7838 - val_loss: 0.6689 - learning_rate:
1.0000e-04
Epoch 33/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.7733 - loss:
0.7373 - val_accuracy: 0.8028 - val_loss: 0.6543 - learning_rate:
1.0000e-04
Epoch 34/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.7752 - loss:
0.7192 - val_accuracy: 0.7948 - val_loss: 0.6660 - learning_rate:
1.0000e-04
Epoch 35/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.7761 - loss:
0.7129 - val_accuracy: 0.7978 - val_loss: 0.6271 - learning_rate:
1.0000e-04
Epoch 36/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7815 - loss:
0.7125 - val_accuracy: 0.8098 - val_loss: 0.6215 - learning_rate:
1.0000e-04
Epoch 37/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7839 - loss:
0.6898 - val_accuracy: 0.8218 - val_loss: 0.6231 - learning_rate:
1.0000e-04
Epoch 38/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.7827 - loss:
0.6987 - val_accuracy: 0.8078 - val_loss: 0.6114 - learning_rate:
1.0000e-04
Epoch 39/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.7888 - loss:
0.6843 - val_accuracy: 0.8218 - val_loss: 0.5848 - learning_rate:
1.0000e-04
Epoch 40/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7905 - loss:
0.6610 - val_accuracy: 0.7988 - val_loss: 0.6478 - learning_rate:
1.0000e-04
Epoch 41/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7868 - loss:
0.6831 - val_accuracy: 0.8178 - val_loss: 0.5850 - learning_rate:
1.0000e-04
Epoch 42/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7994 - loss:
0.6496 - val_accuracy: 0.8168 - val_loss: 0.5944 - learning_rate:
1.0000e-04
Epoch 43/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.7918 - loss:
0.6651 - val_accuracy: 0.8258 - val_loss: 0.5757 - learning_rate:
1.0000e-04
Epoch 44/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.7972 - loss:
0.6515 - val_accuracy: 0.8138 - val_loss: 0.5931 - learning_rate:
1.0000e-04
Epoch 45/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8175 - loss:
0.6036 - val_accuracy: 0.8238 - val_loss: 0.5580 - learning_rate:
1.0000e-04
Epoch 46/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8096 - loss:
0.6182 - val_accuracy: 0.8188 - val_loss: 0.5714 - learning_rate:
1.0000e-04
Epoch 47/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.8046 - loss:
0.6146 - val_accuracy: 0.8358 - val_loss: 0.5345 - learning_rate:
1.0000e-04
Epoch 48/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8224 - loss:
0.5858 - val_accuracy: 0.8328 - val_loss: 0.5410 - learning_rate:
1.0000e-04
Epoch 49/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8210 - loss:
```

```
0.5917 - val_accuracy: 0.8368 - val_loss: 0.5224 - learning_rate:
1.0000e-04
Epoch 50/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8250 - loss:
0.5820 - val_accuracy: 0.8438 - val_loss: 0.5025 - learning_rate:
1.0000e-04
Epoch 51/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8141 - loss:
0.5906 - val_accuracy: 0.8509 - val_loss: 0.4977 - learning_rate:
1.0000e-04
Epoch 52/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8366 - loss:
0.5485 - val_accuracy: 0.8348 - val_loss: 0.5053 - learning_rate:
1.0000e-04
Epoch 53/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8250 - loss:
0.5705 - val_accuracy: 0.8418 - val_loss: 0.4907 - learning_rate:
1.0000e-04
Epoch 54/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8303 - loss:
0.5616 - val_accuracy: 0.8428 - val_loss: 0.5012 - learning_rate:
1.0000e-04
Epoch 55/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8301 - loss:
0.5624 - val_accuracy: 0.8468 - val_loss: 0.4901 - learning_rate:
1.0000e-04
Epoch 56/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8345 - loss:
0.5550 - val_accuracy: 0.8268 - val_loss: 0.5593 - learning_rate:
1.0000e-04
Epoch 57/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8327 - loss:
0.5346 - val_accuracy: 0.8468 - val_loss: 0.4957 - learning_rate:
1.0000e-04
Epoch 58/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8416 - loss:
0.5286 - val_accuracy: 0.8529 - val_loss: 0.4869 - learning_rate:
1.0000e-04
Epoch 59/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8438 - loss:
0.5187 - val_accuracy: 0.8559 - val_loss: 0.4603 - learning_rate:
1.0000e-04
Epoch 60/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8397 - loss:
0.5232 - val_accuracy: 0.8509 - val_loss: 0.4696 - learning_rate:
1.0000e-04
Epoch 61/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8421 - loss:
0.5212 - val_accuracy: 0.8398 - val_loss: 0.5025 - learning_rate:
```

```
1.0000e-04
Epoch 62/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8525 - loss:
0.5037 - val_accuracy: 0.8549 - val_loss: 0.4720 - learning_rate:
1.0000e-04
Epoch 63/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8421 - loss:
0.4997 - val_accuracy: 0.8569 - val_loss: 0.4613 - learning_rate:
1.0000e-04
Epoch 64/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8489 - loss:
0.5045 - val_accuracy: 0.8659 - val_loss: 0.4600 - learning_rate:
1.0000e-04
Epoch 65/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8569 - loss:
0.4694 - val_accuracy: 0.8669 - val_loss: 0.4424 - learning_rate:
1.0000e-04
Epoch 66/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8501 - loss:
0.4982 - val_accuracy: 0.8589 - val_loss: 0.4474 - learning_rate:
1.0000e-04
Epoch 67/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8623 - loss:
0.4720 - val_accuracy: 0.8629 - val_loss: 0.4571 - learning_rate:
1.0000e-04
Epoch 68/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8584 - loss:
0.4634 - val_accuracy: 0.8619 - val_loss: 0.4616 - learning_rate:
1.0000e-04
Epoch 69/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8478 - loss:
0.4872 - val_accuracy: 0.8689 - val_loss: 0.4364 - learning_rate:
1.0000e-04
Epoch 70/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8562 - loss:
0.4727 - val_accuracy: 0.8599 - val_loss: 0.4828 - learning_rate:
1.0000e-04
Epoch 71/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8520 - loss:
0.4831 - val_accuracy: 0.8679 - val_loss: 0.4166 - learning_rate:
1.0000e-04
Epoch 72/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8628 - loss:
0.4481 - val_accuracy: 0.8729 - val_loss: 0.4266 - learning_rate:
1.0000e-04
Epoch 73/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8636 - loss:
0.4450 - val_accuracy: 0.8759 - val_loss: 0.4146 - learning_rate:
1.0000e-04
```

```
Epoch 74/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8708 - loss:
0.4357 - val_accuracy: 0.8639 - val_loss: 0.4304 - learning_rate:
1.0000e-04
Epoch 75/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8676 - loss:
0.4407 - val_accuracy: 0.8639 - val_loss: 0.4387 - learning_rate:
1.0000e-04
Epoch 76/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8684 - loss:
0.4417 - val_accuracy: 0.8799 - val_loss: 0.4059 - learning_rate:
1.0000e-04
Epoch 77/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8766 - loss:
0.4402 - val_accuracy: 0.8859 - val_loss: 0.3921 - learning_rate:
1.0000e-04
Epoch 78/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8675 - loss:
0.4308 - val_accuracy: 0.8759 - val_loss: 0.4205 - learning_rate:
1.0000e-04
Epoch 79/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8690 - loss:
0.4172 - val_accuracy: 0.8739 - val_loss: 0.4206 - learning_rate:
1.0000e-04
Epoch 80/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8687 - loss:
0.4314 - val_accuracy: 0.8769 - val_loss: 0.3994 - learning_rate:
1.0000e-04
Epoch 81/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8788 - loss:
0.3996 - val_accuracy: 0.8729 - val_loss: 0.3998 - learning_rate:
1.0000e-04
Epoch 82/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8744 - loss:
0.4249 - val_accuracy: 0.8769 - val_loss: 0.3993 - learning_rate:
1.0000e-04
Epoch 83/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8906 - loss:
0.3893 - val_accuracy: 0.8789 - val_loss: 0.4082 - learning_rate:
1.0000e-04
Epoch 84/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8677 - loss:
0.4262 - val_accuracy: 0.8759 - val_loss: 0.3963 - learning_rate:
1.0000e-04
Epoch 85/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 33ms/step - accuracy: 0.8874 - loss:
0.3729 - val_accuracy: 0.8839 - val_loss: 0.3934 - learning_rate:
1.0000e-04
Epoch 86/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8831 - loss:
0.3895 - val_accuracy: 0.8809 - val_loss: 0.3827 - learning_rate:
1.0000e-04
Epoch 87/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8842 - loss:
0.4143 - val_accuracy: 0.8789 - val_loss: 0.3965 - learning_rate:
1.0000e-04
Epoch 88/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8960 - loss:
0.3738 - val_accuracy: 0.8699 - val_loss: 0.3990 - learning_rate:
1.0000e-04
Epoch 89/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8841 - loss:
0.3819 - val_accuracy: 0.8829 - val_loss: 0.3941 - learning_rate:
1.0000e-04
Epoch 90/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8907 - loss:
0.3750 - val_accuracy: 0.8869 - val_loss: 0.3915 - learning_rate:
1.0000e-04
Epoch 91/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.8888 - loss:
0.3779 - val_accuracy: 0.8919 - val_loss: 0.3828 - learning_rate:
1.0000e-04
Epoch 92/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8954 - loss:
0.3612 - val_accuracy: 0.8929 - val_loss: 0.3831 - learning_rate:
1.0000e-04
Epoch 93/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8869 - loss:
0.3735 - val_accuracy: 0.8749 - val_loss: 0.4077 - learning_rate:
1.0000e-04
Epoch 94/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8836 - loss:
0.3913 - val_accuracy: 0.8909 - val_loss: 0.3783 - learning_rate:
1.0000e-04
Epoch 95/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.8898 - loss:
0.3807 - val_accuracy: 0.8939 - val_loss: 0.3751 - learning_rate:
1.0000e-04
Epoch 96/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8874 - loss:
0.3753 - val_accuracy: 0.8849 - val_loss: 0.3745 - learning_rate:
1.0000e-04
Epoch 97/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8861 - loss:
0.3763 - val_accuracy: 0.8809 - val_loss: 0.3820 - learning_rate:
1.0000e-04
Epoch 98/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8872 - loss:
```

```
0.3673 - val_accuracy: 0.8879 - val_loss: 0.3721 - learning_rate:
1.0000e-04
Epoch 99/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9035 - loss:
0.3481 - val_accuracy: 0.8759 - val_loss: 0.4054 - learning_rate:
1.0000e-04
Epoch 100/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8970 - loss:
0.3494 - val_accuracy: 0.8859 - val_loss: 0.3868 - learning_rate:
1.0000e-04
Epoch 101/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8960 - loss:
0.3466 - val_accuracy: 0.8889 - val_loss: 0.3599 - learning_rate:
1.0000e-04
Epoch 102/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9014 - loss:
0.3358 - val_accuracy: 0.8929 - val_loss: 0.3627 - learning_rate:
1.0000e-04
Epoch 103/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8994 - loss:
0.3457 - val_accuracy: 0.8879 - val_loss: 0.3566 - learning_rate:
1.0000e-04
Epoch 104/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.8975 - loss:
0.3442 - val_accuracy: 0.8869 - val_loss: 0.3810 - learning_rate:
1.0000e-04
Epoch 105/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.8969 - loss:
0.3466 - val_accuracy: 0.9019 - val_loss: 0.3564 - learning_rate:
1.0000e-04
Epoch 106/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.8977 - loss:
0.3364 - val_accuracy: 0.8899 - val_loss: 0.3979 - learning_rate:
1.0000e-04
Epoch 107/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9074 - loss:
0.3157 - val_accuracy: 0.9059 - val_loss: 0.3385 - learning_rate:
1.0000e-04
Epoch 108/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9060 - loss:
0.3342 - val_accuracy: 0.9009 - val_loss: 0.3550 - learning_rate:
1.0000e-04
Epoch 109/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.8985 - loss:
0.3424 - val_accuracy: 0.8969 - val_loss: 0.3567 - learning_rate:
1.0000e-04
Epoch 110/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9068 - loss:
0.3225 - val_accuracy: 0.8839 - val_loss: 0.3988 - learning_rate:
```

```
1.0000e-04
Epoch 111/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9079 - loss:
0.3180 - val_accuracy: 0.8919 - val_loss: 0.3654 - learning_rate:
1.0000e-04
Epoch 112/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9099 - loss:
0.3144 - val_accuracy: 0.9019 - val_loss: 0.3497 - learning_rate:
1.0000e-04
Epoch 113/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9113 - loss:
0.3079 - val_accuracy: 0.8939 - val_loss: 0.3450 - learning_rate:
1.0000e-04
Epoch 114/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9196 - loss:
0.2836 - val_accuracy: 0.9049 - val_loss: 0.3318 - learning_rate:
1.0000e-04
Epoch 115/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9093 - loss:
0.3014 - val_accuracy: 0.8919 - val_loss: 0.3453 - learning_rate:
1.0000e-04
Epoch 116/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9184 - loss:
0.2941 - val_accuracy: 0.8949 - val_loss: 0.3315 - learning_rate:
1.0000e-04
Epoch 117/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9098 - loss:
0.3097 - val_accuracy: 0.9049 - val_loss: 0.3299 - learning_rate:
1.0000e-04
Epoch 118/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9213 - loss:
0.2852 - val_accuracy: 0.8999 - val_loss: 0.3262 - learning_rate:
1.0000e-04
Epoch 119/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9226 - loss:
0.2914 - val_accuracy: 0.8929 - val_loss: 0.3576 - learning_rate:
1.0000e-04
Epoch 120/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9051 - loss:
0.3144 - val_accuracy: 0.8959 - val_loss: 0.3448 - learning_rate:
1.0000e-04
Epoch 121/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9165 - loss:
0.2929 - val_accuracy: 0.9039 - val_loss: 0.3263 - learning_rate:
1.0000e-04
Epoch 122/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9145 - loss:
0.2974 - val_accuracy: 0.8949 - val_loss: 0.3541 - learning_rate:
1.0000e-04
```

```
Epoch 123/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9134 - loss:
0.2985 - val_accuracy: 0.8999 - val_loss: 0.3479 - learning_rate:
1.0000e-04
Epoch 124/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9200 - loss:
0.2843 - val_accuracy: 0.8999 - val_loss: 0.3256 - learning_rate:
1.0000e-04
Epoch 125/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9099 - loss:
0.3036 - val_accuracy: 0.8949 - val_loss: 0.3365 - learning_rate:
1.0000e-04
Epoch 126/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9235 - loss:
0.2799 - val_accuracy: 0.9149 - val_loss: 0.3208 - learning_rate:
1.0000e-04
Epoch 127/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9132 - loss:
0.2972 - val_accuracy: 0.8909 - val_loss: 0.3456 - learning_rate:
1.0000e-04
Epoch 128/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9155 - loss:
0.2987 - val_accuracy: 0.9049 - val_loss: 0.3352 - learning_rate:
1.0000e-04
Epoch 129/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9225 - loss:
0.2736 - val_accuracy: 0.9079 - val_loss: 0.3226 - learning_rate:
1.0000e-04
Epoch 130/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9202 - loss:
0.2782 - val_accuracy: 0.9029 - val_loss: 0.3280 - learning_rate:
1.0000e-04
Epoch 131/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9214 - loss:
0.2933 - val_accuracy: 0.9099 - val_loss: 0.3053 - learning_rate:
1.0000e-04
Epoch 132/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9230 - loss:
0.2827 - val_accuracy: 0.9129 - val_loss: 0.3015 - learning_rate:
1.0000e-04
Epoch 133/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9142 - loss:
0.2822 - val_accuracy: 0.9049 - val_loss: 0.3258 - learning_rate:
1.0000e-04
Epoch 134/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9198 - loss:
0.2832 - val_accuracy: 0.9079 - val_loss: 0.3160 - learning_rate:
1.0000e-04
Epoch 135/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9277 - loss:
```

```
0.2718 - val_accuracy: 0.9039 - val_loss: 0.3282 - learning_rate:
1.0000e-04
Epoch 136/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9300 - loss:
0.2557 - val_accuracy: 0.9099 - val_loss: 0.3234 - learning_rate:
1.0000e-04
Epoch 137/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9251 - loss:
0.2610 - val_accuracy: 0.9059 - val_loss: 0.3360 - learning_rate:
1.0000e-04
Epoch 138/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9220 - loss:
0.2730 - val_accuracy: 0.8999 - val_loss: 0.3479 - learning_rate:
1.0000e-04
Epoch 139/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9270 - loss:
0.2824 - val_accuracy: 0.9009 - val_loss: 0.3210 - learning_rate:
1.0000e-04
Epoch 140/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9229 - loss:
0.2741 - val_accuracy: 0.9039 - val_loss: 0.3139 - learning_rate:
1.0000e-04
Epoch 141/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9196 - loss:
0.2761 - val_accuracy: 0.9069 - val_loss: 0.3220 - learning_rate:
1.0000e-04
Epoch 142/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.9253 - loss:
0.2587
Epoch 142: ReduceLROnPlateau reducing learning rate to
4.999999873689376e-05.
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9253 - loss:
0.2587 - val_accuracy: 0.9119 - val_loss: 0.3137 - learning_rate:
1.0000e-04
Epoch 143/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9194 - loss:
0.2662 - val_accuracy: 0.9039 - val_loss: 0.3122 - learning_rate:
5.0000e-05
Epoch 144/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9328 - loss:
0.2386 - val_accuracy: 0.9099 - val_loss: 0.3118 - learning_rate:
5.0000e-05
Epoch 145/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9370 - loss:
0.2314 - val_accuracy: 0.9029 - val_loss: 0.3132 - learning_rate:
5.0000e-05
Epoch 146/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9331 - loss:
0.2410 - val_accuracy: 0.9029 - val_loss: 0.3156 - learning_rate:
```

```
5.0000e-05
Epoch 147/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9369 - loss: 
0.2365 - val_accuracy: 0.9119 - val_loss: 0.3064 - learning_rate: 
5.0000e-05
Epoch 148/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9396 - loss: 
0.2333 - val_accuracy: 0.9079 - val_loss: 0.3030 - learning_rate: 
5.0000e-05
Epoch 149/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9362 - loss: 
0.2370 - val_accuracy: 0.9149 - val_loss: 0.2985 - learning_rate: 
5.0000e-05
Epoch 150/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9374 - loss: 
0.2300 - val_accuracy: 0.9089 - val_loss: 0.3186 - learning_rate: 
5.0000e-05
Epoch 151/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9370 - loss: 
0.2335 - val_accuracy: 0.9079 - val_loss: 0.3168 - learning_rate: 
5.0000e-05
Epoch 152/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9370 - loss: 
0.2341 - val_accuracy: 0.9029 - val_loss: 0.3139 - learning_rate: 
5.0000e-05
Epoch 153/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9381 - loss: 
0.2227 - val_accuracy: 0.9099 - val_loss: 0.3076 - learning_rate: 
5.0000e-05
Epoch 154/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9352 - loss: 
0.2403 - val_accuracy: 0.9129 - val_loss: 0.2938 - learning_rate: 
5.0000e-05
Epoch 155/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9439 - loss: 
0.2168 - val_accuracy: 0.9069 - val_loss: 0.3032 - learning_rate: 
5.0000e-05
Epoch 156/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9363 - loss: 
0.2313 - val_accuracy: 0.9099 - val_loss: 0.3060 - learning_rate: 
5.0000e-05
Epoch 157/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 38ms/step - accuracy: 0.9367 - loss: 
0.2261 - val_accuracy: 0.9069 - val_loss: 0.3159 - learning_rate: 
5.0000e-05
Epoch 158/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9401 - loss: 
0.2262 - val_accuracy: 0.9059 - val_loss: 0.3116 - learning_rate: 
5.0000e-05
```

```
Epoch 159/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9404 - loss:
0.2225 - val_accuracy: 0.9149 - val_loss: 0.3329 - learning_rate:
5.0000e-05
Epoch 160/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9307 - loss:
0.2394 - val_accuracy: 0.9129 - val_loss: 0.2982 - learning_rate:
5.0000e-05
Epoch 161/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9435 - loss:
0.2206 - val_accuracy: 0.9109 - val_loss: 0.3075 - learning_rate:
5.0000e-05
Epoch 162/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9418 - loss:
0.2183 - val_accuracy: 0.9069 - val_loss: 0.2882 - learning_rate:
5.0000e-05
Epoch 163/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9352 - loss:
0.2276 - val_accuracy: 0.9119 - val_loss: 0.3164 - learning_rate:
5.0000e-05
Epoch 164/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9375 - loss:
0.2236 - val_accuracy: 0.9149 - val_loss: 0.2920 - learning_rate:
5.0000e-05
Epoch 165/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9387 - loss:
0.2210 - val_accuracy: 0.9149 - val_loss: 0.2865 - learning_rate:
5.0000e-05
Epoch 166/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9395 - loss:
0.2107 - val_accuracy: 0.9089 - val_loss: 0.3062 - learning_rate:
5.0000e-05
Epoch 167/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9449 - loss:
0.2199 - val_accuracy: 0.9159 - val_loss: 0.3004 - learning_rate:
5.0000e-05
Epoch 168/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9346 - loss:
0.2244 - val_accuracy: 0.9159 - val_loss: 0.2790 - learning_rate:
5.0000e-05
Epoch 169/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9440 - loss:
0.2212 - val_accuracy: 0.9169 - val_loss: 0.2923 - learning_rate:
5.0000e-05
Epoch 170/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9485 - loss:
0.1983 - val_accuracy: 0.9129 - val_loss: 0.2994 - learning_rate:
5.0000e-05
Epoch 171/200
```

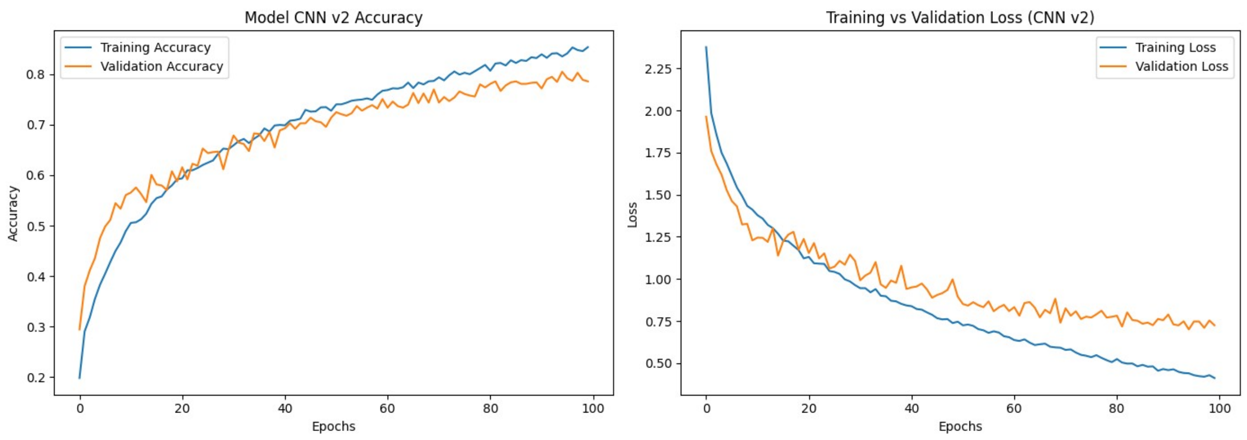
```
219/219 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - accuracy: 0.9338 - loss:
0.2328 - val_accuracy: 0.9119 - val_loss: 0.2937 - learning_rate:
5.0000e-05
Epoch 172/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9368 - loss:
0.2290 - val_accuracy: 0.8999 - val_loss: 0.3265 - learning_rate:
5.0000e-05
Epoch 173/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9434 - loss:
0.2140 - val_accuracy: 0.9159 - val_loss: 0.2862 - learning_rate:
5.0000e-05
Epoch 174/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9391 - loss:
0.2236 - val_accuracy: 0.9119 - val_loss: 0.2903 - learning_rate:
5.0000e-05
Epoch 175/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9383 - loss:
0.2231 - val_accuracy: 0.9139 - val_loss: 0.2833 - learning_rate:
5.0000e-05
Epoch 176/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9384 - loss:
0.2207 - val_accuracy: 0.9169 - val_loss: 0.2896 - learning_rate:
5.0000e-05
Epoch 177/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9455 - loss:
0.2031 - val_accuracy: 0.9159 - val_loss: 0.2919 - learning_rate:
5.0000e-05
Epoch 178/200
217/219 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.9433 - loss:
0.2128
Epoch 178: ReduceLROnPlateau reducing learning rate to
2.499999936844688e-05.
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9432 - loss:
0.2129 - val_accuracy: 0.9159 - val_loss: 0.2882 - learning_rate:
5.0000e-05
Epoch 179/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9424 - loss:
0.2103 - val_accuracy: 0.9109 - val_loss: 0.2907 - learning_rate:
2.5000e-05
Epoch 180/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9480 - loss:
0.1960 - val_accuracy: 0.9099 - val_loss: 0.2861 - learning_rate:
2.5000e-05
Epoch 181/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9518 - loss:
0.1943 - val_accuracy: 0.9149 - val_loss: 0.2786 - learning_rate:
2.5000e-05
Epoch 182/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9394 - loss:
```

```
0.2201 - val_accuracy: 0.9149 - val_loss: 0.2810 - learning_rate:
2.5000e-05
Epoch 183/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9451 - loss:
0.2102 - val_accuracy: 0.9149 - val_loss: 0.2824 - learning_rate:
2.5000e-05
Epoch 184/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9504 - loss:
0.1955 - val_accuracy: 0.9119 - val_loss: 0.2893 - learning_rate:
2.5000e-05
Epoch 185/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9486 - loss:
0.1930 - val_accuracy: 0.9169 - val_loss: 0.2747 - learning_rate:
2.5000e-05
Epoch 186/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9435 - loss:
0.2111 - val_accuracy: 0.9169 - val_loss: 0.2756 - learning_rate:
2.5000e-05
Epoch 187/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9472 - loss:
0.1941 - val_accuracy: 0.9149 - val_loss: 0.2801 - learning_rate:
2.5000e-05
Epoch 188/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9439 - loss:
0.2050 - val_accuracy: 0.9159 - val_loss: 0.2858 - learning_rate:
2.5000e-05
Epoch 189/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9447 - loss:
0.2054 - val_accuracy: 0.9119 - val_loss: 0.2792 - learning_rate:
2.5000e-05
Epoch 190/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9393 - loss:
0.2190 - val_accuracy: 0.9169 - val_loss: 0.2834 - learning_rate:
2.5000e-05
Epoch 191/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9431 - loss:
0.2036 - val_accuracy: 0.9189 - val_loss: 0.2734 - learning_rate:
2.5000e-05
Epoch 192/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9508 - loss:
0.1837 - val_accuracy: 0.9189 - val_loss: 0.2808 - learning_rate:
2.5000e-05
Epoch 193/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 11s 38ms/step - accuracy: 0.9400 - loss:
0.2140 - val_accuracy: 0.9149 - val_loss: 0.2768 - learning_rate:
2.5000e-05
Epoch 194/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9468 - loss:
0.2000 - val_accuracy: 0.9189 - val_loss: 0.2750 - learning_rate:
```

```
2.5000e-05
Epoch 195/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9425 - loss:
0.1986 - val_accuracy: 0.9179 - val_loss: 0.2872 - learning_rate:
2.5000e-05
Epoch 196/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9487 - loss:
0.1963 - val_accuracy: 0.9129 - val_loss: 0.2814 - learning_rate:
2.5000e-05
Epoch 197/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 36ms/step - accuracy: 0.9497 - loss:
0.1938 - val_accuracy: 0.9129 - val_loss: 0.2828 - learning_rate:
2.5000e-05
Epoch 198/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9473 - loss:
0.2005 - val_accuracy: 0.9139 - val_loss: 0.2770 - learning_rate:
2.5000e-05
Epoch 199/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.9553 - loss:
0.1898 - val_accuracy: 0.9239 - val_loss: 0.2708 - learning_rate:
2.5000e-05
Epoch 200/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 8s 35ms/step - accuracy: 0.9397 - loss:
0.2101 - val_accuracy: 0.9179 - val_loss: 0.2740 - learning_rate:
2.5000e-05
```

Statistics & Graphs (Accuracy and Loss)

```python
import matplotlib.pyplot as plt

# Plot accuracy
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model CNN v2 Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot loss
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 vs Validation Loss (CNN v2)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()

# Evaluate final accuracy on test set
test_loss_v2, test_acc_v2 = model_cnn_v2.evaluate(X_test_cnn, y_test,
verbose=2)
print(f"\n□ CNN v2 Test Accuracy: {test_acc:.4f}")
```



```
63/63 - 1s - 12ms/step - accuracy: 0.9259 - loss: 0.2600

□ CNN v2 Test Accuracy: 0.7983
```

During initial testing of our CNN model, we encountered critical dimensionality errors, primarily due to incorrect reshaping of MFCC feature data for Conv2D layers. Initially, input data was flattened incorrectly (from (samples, 132, 13) to (samples, 1716)), leading to errors like "expected 4D input, but received 3D input" and negative dimension size issues when applying convolution operations. These errors significantly limited model performance and learning capabilities.

To address these issues and optimize the CNN architecture from 79% accuracy toward approximately 90%, several critical adjustments were made:

Corrected Input Shape: Data reshaped properly as (samples, 132, 13, 1) to ensure compatibility with CNN layers. Enhanced Depth and Complexity: Additional convolutional blocks (increased to four layers) and higher filter counts (up to 256) to capture more detailed and deeper features. Regularization Enhancements: Incorporated Batch Normalization, Dropout, and Global Average Pooling layers to reduce overfitting and improve generalization. Learning Rate Reduction and Early Stopping: Adjusted hyperparameters and implemented callbacks for optimal training and convergence. These optimizations significantly improved model accuracy and robustness