Final Project

Introduction to Artificial Intelligence

Spring 2025

“Identification of Primary Instruments in Music”

Present By Group 13

112550106 林瀚璿

112550121 江宸安

112550134 賴儁樞

112550174 林紹安

Outline

Introduction

Related Work

Dataset/Platform

Baseline

Main Approach

Evaluation Metric

Results & Analysis

Github Link & Reference

Contribution of each member

Introduction

Problem Definition

- Multi-Instrument Recognition (MIR): Identifying all instruments active in a music segment.
- Dominant Instrument Detection: Finding the instrument with the longest total playing time in a piece.

Introduction

Importance and Motivation

- Instrument recognition supports music search, transcription, and education.
- Identifying the dominant instrument helps highlight the key melodic or structural role in music.

Introduction

Project Objectives

- Build a deep learning model for multi-label MIR on polyphonic audio.
- Detect the dominant instrument using MIDI-derived cumulative playtime.
- Evaluate performance on the Slakh2100 dataset with proper metrics.

Introduction

Our Approach and Contribution

- We use a CRNN with attention for MIR.
- Our main method is using MIDI annotations to define dominance via playing duration.
- To boost model performance, we apply class-weighted loss and SpecAugment.

Related Work - Previous

The Slakh2100 Dataset:

We use Slakh2100 (Bitton et al., 2020), a large-scale (145 hours) synthesized dataset with precise audio-MIDI alignment, ideal for our tasks.

MIR techniques evolved from traditional feature-based methods to deep learning (CNNs, CRNNs, Transformers) on Mel-spectrograms, mostly for frame-level multi-label classification. Our model is a CRNN.

Related Work - Ours

Duration-Based Dominant Instrument Detection:
A key goal is identifying the track's "dominant" instrument, defined as the one with the longest cumulative playing time.

We **leverage** Slakh2100's MIDI for objective dominance assessment, alongside **refined training** (weighted loss, SpecAugment) and **evaluation** (optimal per-class thresholds).

Dataset/Platform

Overview

- Slakh2100 contains 2,100 automatically mixed audio tracks and corresponding aligned MIDI files.
- Synthesized from 187 instrument patches, categorized into 34 instrument classes. (However, we only use 17 classes.)
- Total duration is approximately 145 hours.

Dataset/Platform

Audio Format

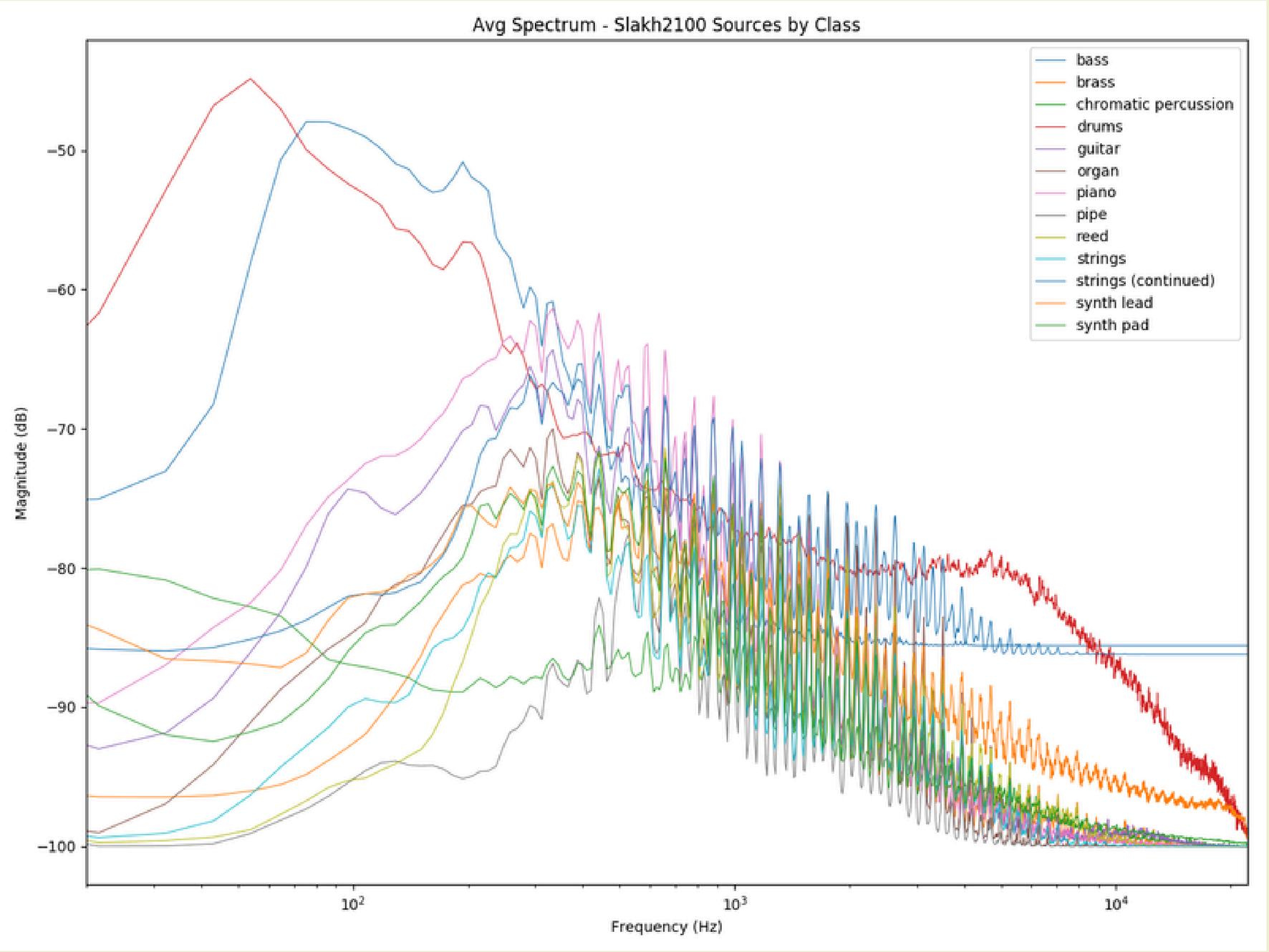
- Format: FLAC
- Mono channel
- 44.1kHz, 16-bit

Synthesis and Mixing

- MIDI Source: Lakh MIDI Dataset v0.1
- Synthesis Method: Professional-grade sample-based virtual instruments
- Mixing: Tracks are normalized for integrated loudness according to ITU-R BS.1770-4 and then mixed into a single audio track

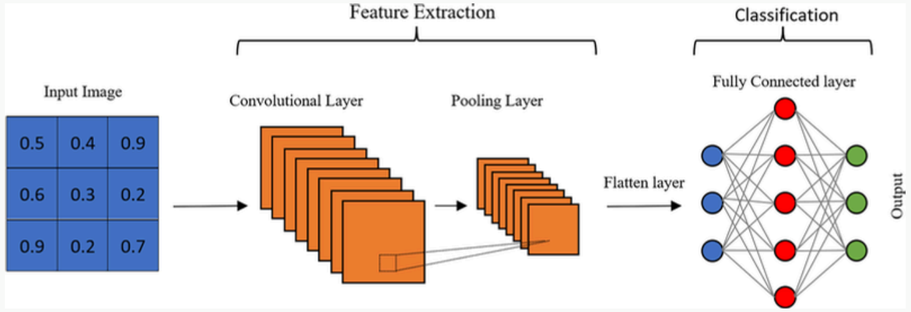
Visual Graph

frequency characteristics for different instrument groups in Slakh2100, highlighting their unique timbral signatures



Baseline

- **Convolution Neural Network:**
  - BCE with Logits Loss & AdamW optimizer
  - epoch = 100 with early stop
  - lr = 0.001, batch size = 16
  - 3 layers
  - ELU



picture 1

```
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)

criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight_tensor

scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=0.5, patience=3)
```

Baseline

- **AdamW:**
  - separates weight decay from the gradient update
  - AdamW = Adam + Weight Decay
  - less overfitting
  - learning rate: 1e-3
  - weight decay: 1e-2

$$
\begin{aligned}
g_t &= \nabla_\theta L(\theta_t) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)
\end{aligned}
$$

Baseline

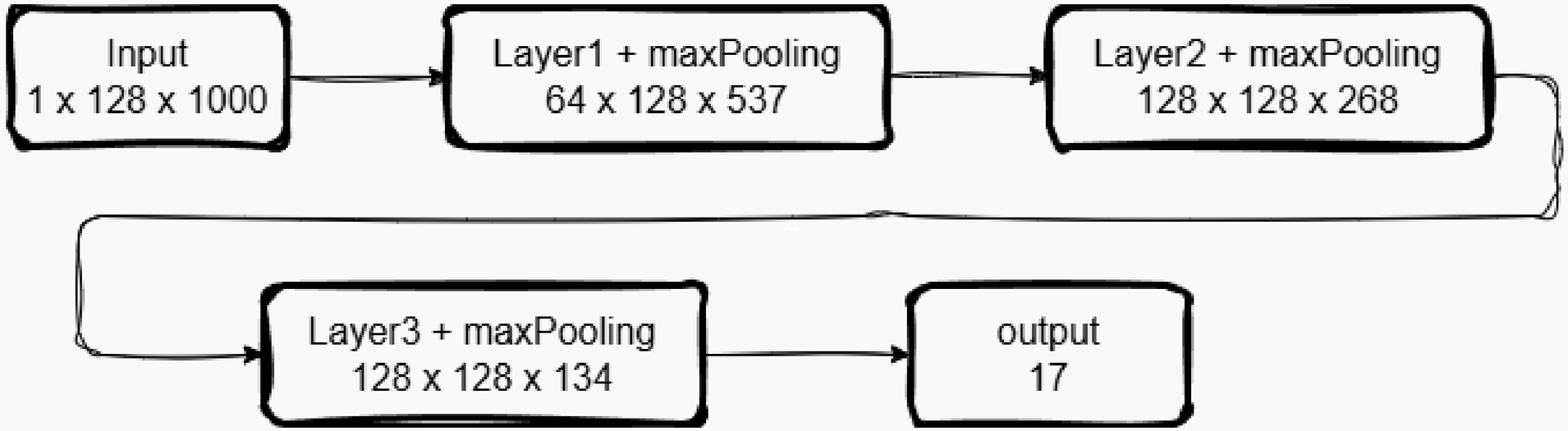
- **Convolution Neural Network:**

```python
class CNN(nn.Module):
    def __init__(self, num_classes=17):
        super(CNN, self).__init__()

        self.pad = nn.ZeroPad2d((37, 37, 0, 0))
        self.bn0 = nn.BatchNorm2d(1)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop3 = nn.Dropout(0.5)

        self.drop_final = nn.Dropout(0.7)
        self.fc = nn.Linear(128, num_classes)
```

```python
def forward(self, x):
        # x: (batch, 1, n_mels, time)
        x = self.pad(x)
        x = self.bn0(x)

        x = F.elu(self.bn1(self.conv1(x)))
        x = self.pool1(x)
        x = self.drop1(x)

        x = F.elu(self.bn2(self.conv2(x)))
        x = self.pool2(x)
        x = self.drop2(x)

        x = F.elu(self.bn3(self.conv3(x)))
        x = self.pool3(x)
        x = self.drop3(x)

        x = x.mean(dim=2)  # (batch, channels, pooled_time)
        x = x.permute(0, 2, 1)  # (batch, pooled_time, channels)

        x = self.drop_final(x)
        x = self.fc(x)  # (batch, pooled_time, num_classes)

        if x.shape[1] != 1000:
            x = x.permute(0, 2, 1)
            x = F.interpolate(x, size=1000, mode="linear", align_corners=False)
            x = x.permute(0, 2, 1)
        return x
```

Baseline

- **Convolution Neural Network:**



Main Approach

- **Convolution recurrent neural network:**
  - CRNN combines CNN and RNN models to enhance the precision of voice data analysis.
  - The CNN part extracts local time-frequency features from the audio spectrogram.
  - The RNN part (GRU) then processes this sequence of features over time, helping us capture and utilize the temporal dependencies and context within the audio signal

Main Approach

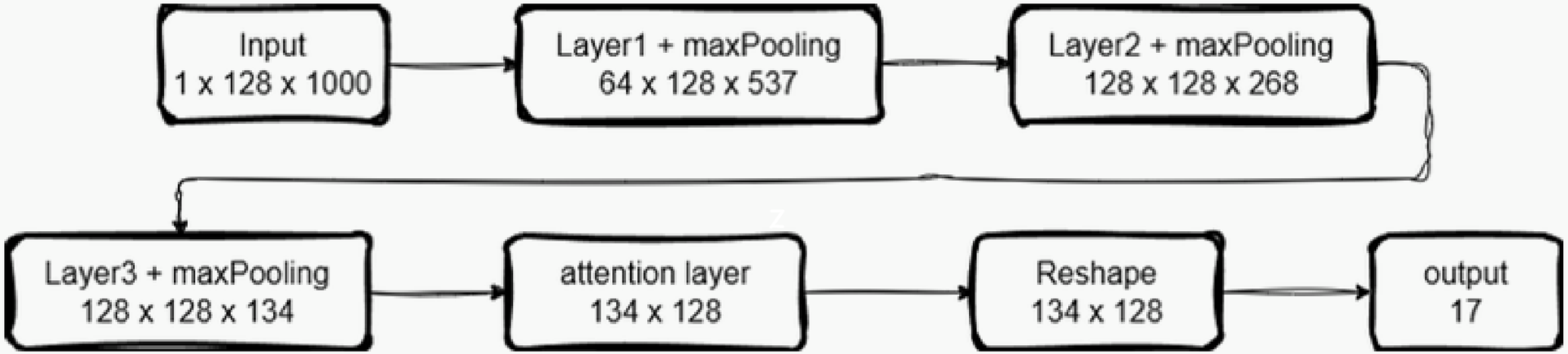
- **Convolution recurrent neural network:**

```python
class CRNN(nn.Module):
    def __init__(self, num_classes=17):
        super(CRNN, self).__init__()

        self.pad = nn.ZeroPad2d((37, 37, 0, 0))
        self.bn0 = nn.BatchNorm2d(1)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop3 = nn.Dropout(0.5)

        self.gru1 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )
        self.gru2 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )

        self.attention = nn.MultiheadAttention(
            embed_dim=128, num_heads=2, batch_first=True
        )
        self.attn_norm = nn.LayerNorm(128)
        self.attn_dropout = nn.Dropout(0.5)

        self.drop_final = nn.Dropout(0.7)
        self.fc = nn.Linear(128, num_classes)
```

```python
def forward(self, x):
        # x: (batch, 1, n_mels, time)
        x = self.pad(x)
        x = self.bn0(x)

        x = F.elu(self.bn1(self.conv1(x)))
        x = self.pool1(x)
        x = self.drop1(x)

        x = F.elu(self.bn2(self.conv2(x)))
        x = self.pool2(x)
        x = self.drop2(x)

        x = F.elu(self.bn3(self.conv3(x)))
        x = self.pool3(x)
        x = self.drop3(x)

        x = x.mean(dim=2)  # (batch, channels, pooled_time)
        x = x.permute(0, 2, 1)  # (batch, pooled_time, channels)

        x, _ = self.gru1(x)
        x, _ = self.gru2(x)

        attn_output, _ = self.attention(x, x, x, need_weights=False)
        x = self.attn_norm(x + attn_output)
        x = self.attn_dropout(x)

        x = self.drop_final(x)
        x = self.fc(x)  # (batch, pooled_time, num_classes)

        if x.shape[1] != 1000:
            x = x.permute(0, 2, 1)
            x = F.interpolate(x, size=1000, mode="linear", align_corners=False)
            x = x.permute(0, 2, 1)
        return x
```

Main Approach

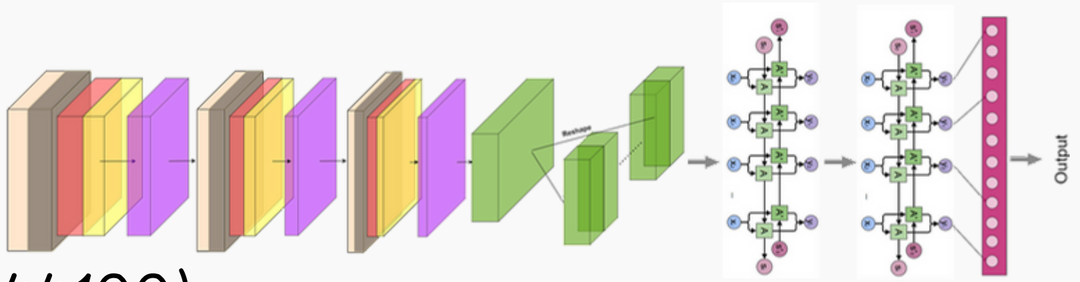
- **Convolution recurrent neural network:**



Main Approach

Experiment method :

- Experiment 01:
  - CRNN & CNN (both with SR = 44100)



picture 2

- Experiment 02:
  - CRNN with attention & CRNN without attention (both with SR = 44100)
- Experiment 03:
  - CRNN with SR = 22050 & CRNN with SR = 44100 (both with attention)

We want to check the affects of attention and SR on the CRNN model

Main Approach

- **Experiment 01: Baseline vs. Main Approach**

```python
class CNN(nn.Module):
    def __init__(self, num_classes=17):
        super(CNN, self).__init__()

        self.pad = nn.ZeroPad2d((37, 37, 0, 0))
        self.bn0 = nn.BatchNorm2d(1)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop3 = nn.Dropout(0.5)

        self.drop_final = nn.Dropout(0.7)
        self.fc = nn.Linear(128, num_classes)
```

```python
class CRNN(nn.Module):
    def __init__(self, num_classes=17):
        super(CRNN, self).__init__()

        self.pad = nn.ZeroPad2d((37, 37, 0, 0))
        self.bn0 = nn.BatchNorm2d(1)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop3 = nn.Dropout(0.5)

        self.gru1 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )
        self.gru2 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )

        self.attention = nn.MultiheadAttention(
            embed_dim=128, num_heads=2, batch_first=True
        )
        self.attn_norm = nn.LayerNorm(128)
        self.attn_dropout = nn.Dropout(0.5)

        self.drop_final = nn.Dropout(0.7)
        self.fc = nn.Linear(128, num_classes)
```

Main Approach

- **Experiment 02: with / without self attention mechanism:**
  - let RNN model has the feature representation from the entire sequence.

```
class CRNN(nn.Module):
    def __init__(self, num_classes=17):
        super(CRNN, self).__init__()

        self.pad = nn.ZeroPad2d((37, 37, 0, 0))
        self.bn0 = nn.BatchNorm2d(1)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(1, 2), stride=(1, 2))
        self.drop3 = nn.Dropout(0.5)

        self.gru1 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )
        self.gru2 = nn.GRU(
            input_size=128, hidden_size=64, batch_first=True, bidirectional=True
        )

        self.attention = nn.MultiheadAttention(
            embed_dim=128, num_heads=2, batch_first=True
        )
        self.attn_norm = nn.LayerNorm(128)
        self.attn_dropout = nn.Dropout(0.5)

        self.drop_final = nn.Dropout(0.7)
        self.fc = nn.Linear(128, num_classes)
```

Main Approach

- **Experiment 03: Different Sampling Rate(22050 / 44100 Hz):**
  - According to Nyquist Frequency Limit, higher sampling rate can adapt higher frequency of the instrument.

```python
def gen_melgram(path, num_frame=1000, segment_sec=100, is_train=False):
    SR = 22050
    N_MELS = 128
    N_FFT = 2048

    try:
        src, sr = librosa.load(path, sr=SR)
    except Exception as e:
        return None

    total_sec = len(src) / sr

    if total_sec < 200:
        return None

    center_sec = total_sec / 2
    start_sec = max(0, center_sec - segment_sec / 2)
    end_sec = min(
        total_sec, start_sec + segment_sec
    )
    end_sec = min(end_sec, total_sec)  # Cannot exceed total duration

    if (
        end_sec - start_sec
    ) < 10.0:
        return None

    start_sample = int(start_sec * sr)
    end_sample = int(end_sec * sr)
    src_segment = src[start_sample:end_sample]

    if len(src_segment) < N_FFT:
        return None

    ...
```

```python
def gen_melgram(path, num_frame=1000, segment_sec=100, is_train=False):
    SR = 44100
    N_MELS = 128
    N_FFT = 2048

    try:
        src, sr = librosa.load(path, sr=SR)
    except Exception as e:
        return None

    total_sec = len(src) / sr

    if total_sec < 200:
        return None

    center_sec = total_sec / 2
    start_sec = max(0, center_sec - segment_sec / 2)
    end_sec = min(
        total_sec, start_sec + segment_sec
    )
    end_sec = min(end_sec, total_sec)  # Cannot exceed total duration

    if (
        end_sec - start_sec
    ) < 10.0:
        return None

    start_sample = int(start_sec * sr)
    end_sample = int(end_sec * sr)
    src_segment = src[start_sample:end_sample]

    if len(src_segment) < N_FFT:
        return None

    ...
```

Evaluation Metrics

Precision:

- **Measures:** The accuracy of the model's positive predictions.
- **Denominator represents:** All instances predicted by the model as positive for Instrument X .
- **Importance:** High precision means the model makes few false alarms.

Recall:

- **Measures:** The model's ability to find all actual positive instances.
- **Denominator represents:** All instances where Instrument X actually is present in the ground truth.
- **Importance:** High recall means the model misses few actual instrument occurrences.

Evaluation Metrics

F1-score:

- **Measures:** The harmonic mean of Precision and Recall, providing a balanced score.
- **Importance:** Useful when both precision and recall matter, especially with class imbalance.
- **Micro-Averaging (e.g., Micro F1-score):**
  - Calculates global F1 using total true/false positives and false negatives across all classes and frames.
  - Emphasizes overall performance, favoring frequent classes.
- **Macro-Averaging (e.g., Macro F1-score):**
  - Computes F1 per class, then averages equally across all classes.
  - Balances evaluation across both common and rare instruments.

Evaluation Metric

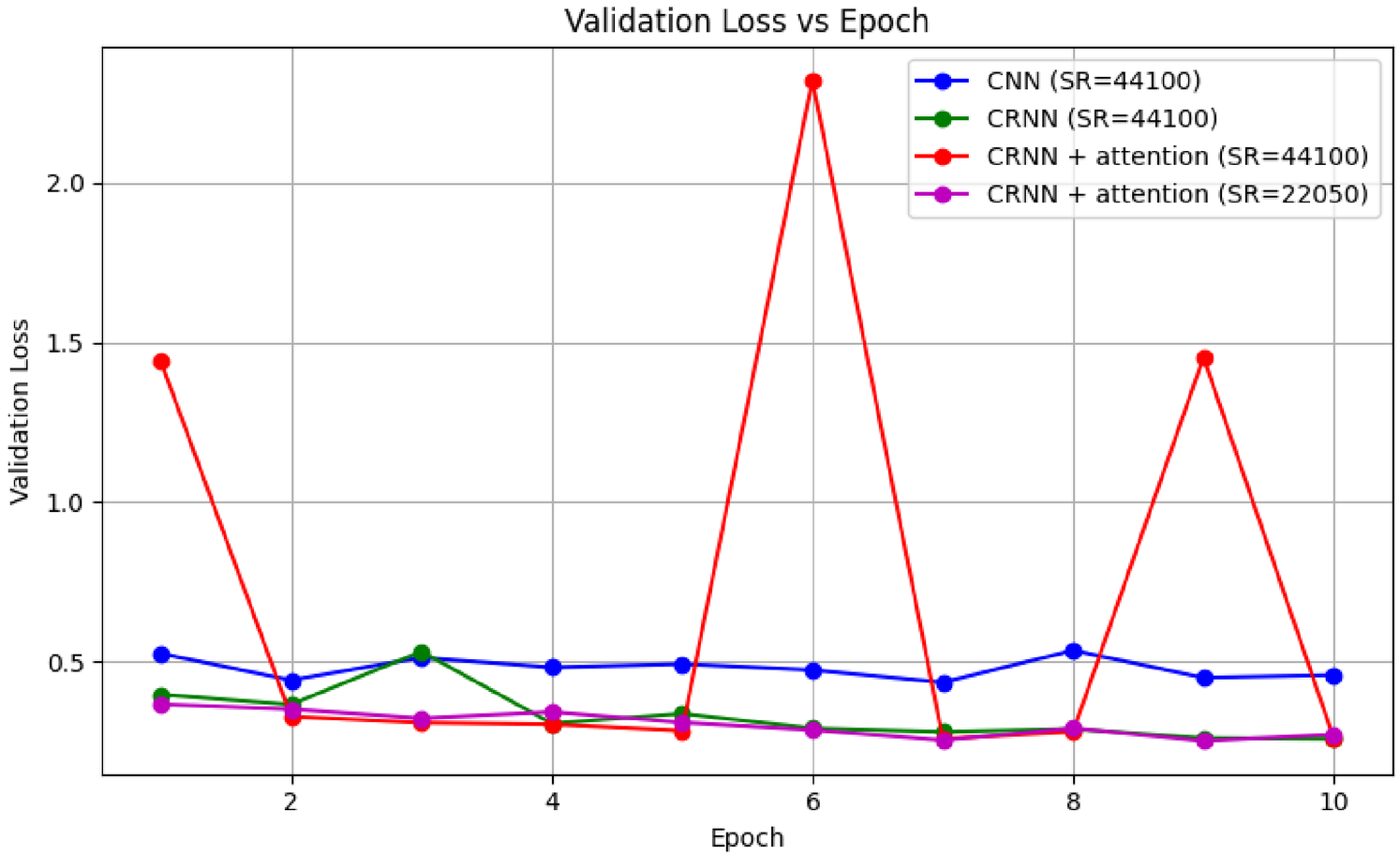
$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

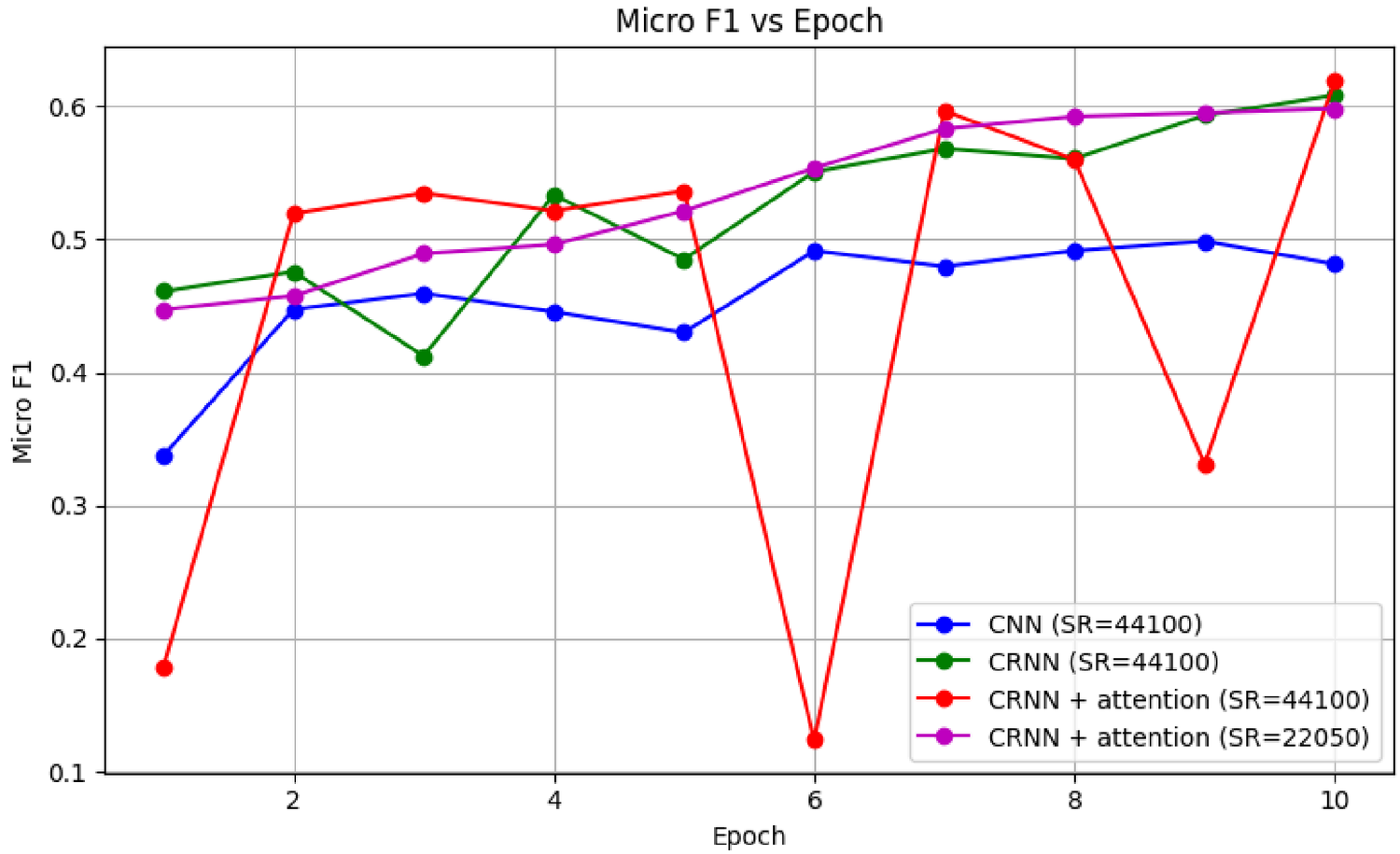
$$F1\ \text{score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **True Positive:** The model correctly predicts a class as present when it actually is.
- **True Negative:** The model correctly predicts a class as absent when it actually is not present.
- **False Positive:** The model predicts a class as present when it is actually absent.
- **False Negative:** The model predicts a class as absent when it is actually present.

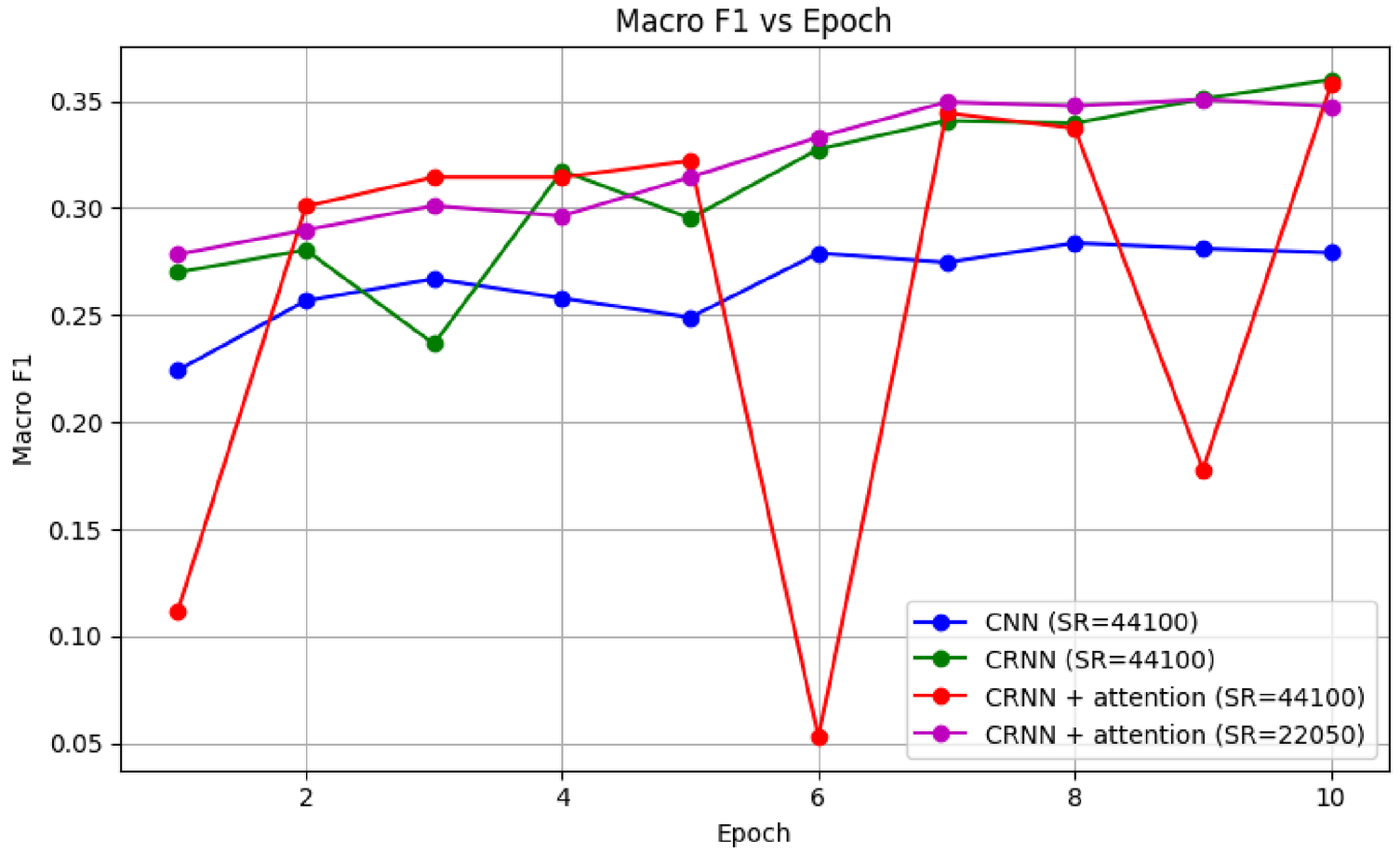
Result and Analysis



Result and Analysis



Result and Analysis



Result and Analysis

Observation 1: RNN model sequential dependencies

	CNN	CRNN	CRNN + Attention (SR = 44100)	CRNN + Attention (SR = 22050)
F1	0.5585	0.6861	0.6202	0.6688
precision	0.4688	0.6340	0.5248	0.6100
recall	0.7581	0.7931	0.8103	0.7756

By adding a recurrent neural network, the model can better understand the sequential order of the notes.

Result and Analysis

Observation 2: Attention is not all you need

	CNN	CRNN	CRNN + Attention (SR = 44100)	CRNN + Attention (SR = 22050)
F1	0.5585	0.6861	0.6202	0.6688
precision	0.4688	0.6340	0.5248	0.6100
recall	0.7581	0.7931	0.8103	0.7756

The conventional CRNN outperforms the CRNN with an attention mechanism. Since the CRNN model may have already extracted sufficient features, adding an attention mechanism increases complexity, which can lead to overfitting.

Result and Analysis

Observation 3: Too more classes destroyed the model

	CRNN for 17 classes	CRNN for 129 classes
F1	0.6861	0.3570
Precision	0.6340	0.3798
Recall	0.7931	0.3693

Having too many subclasses may lead to lower model accuracy, as the model may struggle to distinguish between them.

Github Link & Reference

Github Link

- https://github.com/NYCU-room429/AI_fianl_project

Reference

- librosa: Audio and Music Signal Analysis in Python
- DataSet: zSlakh2100
- Attention Is All You Need
- OCR：CRNN+CTC開源加詳細解析
- CRNN Pytorch
- Precision, Recall, F1-score簡單介紹
- Picture 1: https://peerj.com/articles/cs-1395/
- Picture 2: https://reurl.cc/NYdQgx

Contribution of each member

- 112550106 林瀚璿 - 25%
  - CRNN architecture
- 112550121 江宸安 - 25%
  - GUI implementation, Plot, main.py
- 112550134 賴儁樞 - 25%
  - experiments, Canva slide, test.py
- 112550174 林紹安 - 25%
  - ReadMe, Github management, Dataset, utils.py

Thank You!