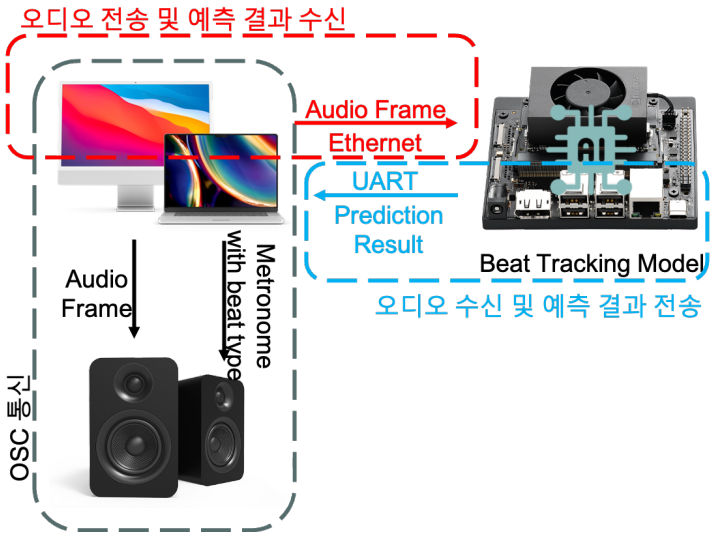
실시간 박자 추적 프로그램 매뉴얼

작성자	카이스트 문화기술대학원 박사과정 김종수 (실무자)
작성완료일	2025. 01. 13. (월)
문의	jongsoo.kim@kaist.ac.kr

1. 전체 시스템 개요



[그림 1] 전체 시스템 개요

실시간 박자 추적 시스템은 아래와 같이 크게 3가지 영역으로 나누어져 있음:

1. client.py & com_setup.py	Device	오디오 전송, OSC 통신 및 예측 결과 수신
2. server.py & util.py & com_setup.py	Jetson	오디오 수신, 모델 추론 및 예측 결과 전송
3. osc_server.py	Device	OSC 통신 및 오디오 재생

Jetson 보드와 Device 양쪽 모두에 있는 “com_setup.py” 파일은 통신과 추가 기능에 필요한 값들을 설정하는 파일이다. “client.py”는 device 단의 메인 프로그램이며, “server.py”는 Jetson 보드 단의 메인 프로그램이다. “osc_server.py”는 client 프로그램이 예측 결과를 받았을 때, device에서 osc 통신으로 오디오를 재생하기 위한 프로그램이다. “util.py”는 Jetson 보드에서 모델 추론 및 로드에 필요한 함수 및 클래스가 정의된 파이썬 모듈 파일이다. “make_trt.py” 파일은 모델의 체크포인트를 로드하여 사전 학습된 박자 추적 모델을 TensorRT 엔진으로 변환하는 프로그램이다. 본 연구의 경우, device를 윈도우 PC로 세팅하여 실험을 진행하였다.

2. 설치 가이드

1) 윈도우 PC 라이브러리 및 파이썬 패키지 설치

[라이브러리] *ffmpeg* (conda 명령어를 통해 설치하는 것을 추천) [직접설치][conda설치]
[파이썬 패키지] pip 명령어를 통해 설치
pyserial, numpy, pydub, pythonosc, sounddevice

---

2) Jetson 보드 라이브러리 및 파이썬 패키지 설치

[라이브러리]
TensorRT [설치가이드]
cmake >= 3.20
ffmpeg
[파이썬 패키지]
python == 3.8.x
pytorch == 2.1.0a+41361538.nv23.6 [다운로드 링크]
torchaudio == 2.1.0
numpy <= 1.23.0
madmom **(구 개발자 버전)**
omegaconf
[필요시] *NATTEN == 0.17.3*

Jetson 보드의 경우, GPU 특성 상 파이썬 패키지들의 버전 의존성이 매우 크므로 반드시 해당 버전을 지켜줘야 한다. madmom의 경우, 제공된 구 개발자 버전 설치 파일을 통해 설치를 진행해야 한다. TensorRT의 경우, 설치가 올바르게 되었다면 python 프로그램에서 tensorrt 패키지를 import할 때 에러가 발생하지 않는다.

Torchaudio의 설치 방법은 아래와 같다.

 # 설치 명령어
$ pip install ninja
$ sudo apt install libavformat-dev libavcodec-dev libavutil-dev libavdevice-dev libavfilter-dev
$ git clone –branch release/2.1.0 https://github.com/pytorch/audio
$ cd audio
$ USE_CUDA=1 pip install -v -e . --no-use-pep517

 # 설치 후 확인 작업

$ python

> import torchaudio

> print(torchaudio.__version__)

 # 정상적으로 버전이 출력되면 올바르게 설치된 것

상황에 따라 제공된 TensorRT 모델 엔진 사용 시 경고 메시지 혹은 에러가 발생할 수 있다. 이는 해당 엔진이 현재 사용 중인 Jetson 보드의 호환성 경고 문제일 수도 있어 이럴 경우에는 직접 해당 Jetson 보드에서 모델 엔진 변환을 수행해야 한다. 이를 위해 필요한 파이썬 패키지는 NATTEN 패키지이며 추가로 제공되는 make_trt.py 파일을 통해 모델 엔진 변환이 가능하다.

NATTEN 패키지의 설치 방법은 아래와 같다.

 # 설치 명령어

$ git clone --recursive https://github.com/SHI-Labs/NATTEN

$ cd NATTEN

$ pip install -r requirements.txt

$ make WORKERS=3 CUDA_ARCH="8.7"

 # 이 때 빌드 95% 쯤에서 종료가 되면 바로 아래 명령어를 실행

$ make WORKERS=3

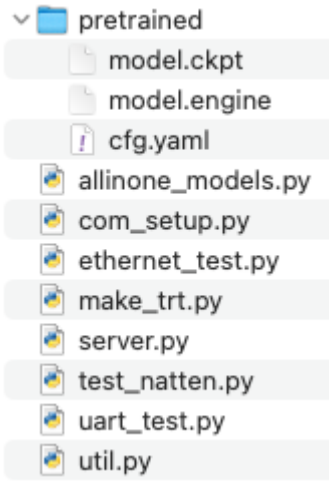
 # 설치 후 확인 작업

$ python test_natten.py

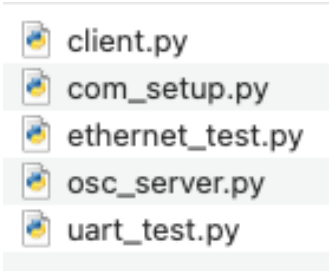
NATTEN 패키지의 경우, 빌드 과정에서 시간이 약 3~4시간 정도 소요될 수 있다.

3) 기초 세팅

제공되는 파일의 구성은 아래의 표와 같이 구성해야 한다.



[그림 2] Jetson 내 파일 구성



[그림 3] Device 내 파일 구성

윈도우 PC와 Jetson 보드 간의 이더넷 통신을 위해 USB-C to RJ45 기가비트 케이블을 사용하였으며, UART 통신을 위해서는 CP2102 컨버터를 사용하였다. 또한, 약간의 CPU 성능 향상을 위해 jetson clocks를 활성화하여 실험을 진행하였다.

3. 실행 가이드

1) 실행 전 세팅

(1) 이더넷 통신 세팅

[윈도우 PC에서의 com_setup.py]

host_i: str = '10.0.0.1'

port: int = 12345

client_ip: str = '10.0.0.2'

[Jetson 보드에서의 com_setup.py]

host_ip: str = '10.0.0.1'

port: int = 12345

Jetson 보드에서 오디오 데이터 수신을 위해 이더넷을 사용하고자 하면, 아래와 같은 명령어를 터미널에서 실행시켜 주어야 한다. (jetson 재부팅 후에는 아래의 명령어를 다시 실행해야 함)

$ sudo ifconfig eth0 10.0.0.1 netmask 255.255.255.0 up
$ sudo ip route add 10.0.0.0/24 dev eth0

윈도우 PC에서도 아래와 같이 네트워크 설정에서 이더넷에 대한 IP 주소 값을 지정해줘야 한다.

[그림 4] 윈도우 PC 이더넷 네트워크 설정

Jetson 보드와 윈도우 PC의 이더넷 관련 설정이 모두 정상적으로 완료되었는지 확인하기 위해서는 제공된 각 기기 폴더에 있는 "ethernet_test.py" 파일을 통해 확인할 수 있다. Jetson 보드에서의 ethernet_test.py 파일부터 실행시킨 다음, 윈도우 PC의 ethernet_test.py 파일을 실행시켜 정상적으로 결과가 출력되는지 확인한다.

(2) UART 통신 세팅

[윈도우 PC에서의 com_setup.py]
serial_port: str = 'COM3'
baud_rate: int = 38400

[Jetson 보드에서의 com_setup.py]
serial_port: str = '/dev/ttyTHS0'
baud_rate: int = 38400

이 때 두 기기의 baud_rate는 반드시 같은 값이어야 하며, 두 기기의 serial_port는 자신의 환경에 맞게 알맞게 변경하는 것이 중요하다.

윈도우 PC에서의 적절한 serial_port는 아래 그림과 같이 장치 관리자의 포트에서 알 수 있다. 연결된 장치명 우측 끝 괄호 안에 serial_port가 명시되어 있다.

[그림 5] 윈도우 PC 장치 관리자 포트 화면

Jetson 보드에서 터미널 창에서 'dmesg | grep tty' 명령어를 통해 serial_port를 알 수 있다. UART 통신이 올바르게 작동되는 것을 확인하기 위해서는 먼저 윈도우 PC에서 uart_test.py를 실행시키고, jetson 보드에서 uart_test.py를 실행시킨다.

(3) 기타 세팅

[윈도우 PC에서의 com_setup.py]
debug_dl = False
file_path = '음악_파일_경로'

[Jetson 보드에서의 com_setup.py]
pretrained_dir = '모델_파일_경로'

윈도우 PC에서 debug_dl을 True 값으로 설정하면, 음악 파일을 1분 정도만 재생하며, 프로그램 종료 시에 오디오 전송 및 예측 결과 수신 시점에 대한 시간값 등이 저장된다. file_path의 경우, 실험에 사용할 음악 파일의 경로를 값으로 받는 변수이다. 오디오 파일의 경우, mp3나 wav 확장자의 오디오 파일 사용을 권장한다.

Jetson 보드에서 pretrained_dir은 제공되는 모델의 체크포인트 혹은 엔진 파일이 들어있는 폴더의 경로를 의미한다. 또한, 엔진 파일을 "make_trt.py"을 통해 생성할 경우, 엔진 파일이 해당 경로의 폴더에 생성된다.

2) 프로그램 실행 순서 및 방법

(1) 윈도우 PC에서 osc_server.py 실행
(2) Jetson 보드에서 server.py 실행
(3) Listening 문구가 뜰 때까지 대기
(4) 윈도우 PC에서 client.py 실행

3) TensorRT 관련 에러 및 경고 메시지가 발생할 경우

해당 경우에는 우선 Jetson 보드에 NATTEN 패키지를 설치해야 한다. 이후, jetson 보드에서 make_trt.py 파일을 실행시켜 pretrained_dir 경로에 해당되는 폴더에 model.engine 파일을 생성시켜준다. 이 때, 모델 엔진 변환 과정에서 시간이 10분 가량 소요 될 수 있다.

4. Jetson 주요 코드 설명

1) allinone_models.py

박자 추적 모델들의 코드들로 구성된 파일이며, make_trt.py 파일 사용 싱에만 필요한 파일이다. 새로운 모델을 구현하여 학습할 경우에는 새로운 모델의 코드를 해당 파일에 추가 작성하여야 한다.

2) com_setup.py

(1) [변수] pretrained_dir

해당 변수는 model.ckpt, cfg.yaml, model.onnx, model.engine 등이 포함된 디렉토리의 경로 값이 문자열 형태로 저장된 변수이다.

(2) [변수] threshold_beat, threshold_downbeat

본 모델은 DBN이 아니 max pooling 기반의 새로운 방식으로 후처리를 하는데, 이때 beat와 downbeat 확률의 임계값이 필요하다. 두 변수 모두 0~1 사이의 실수값을 가진다.

(3) [변수] dont_need_beat_buffer

해당 변수는 학습된 모델의 종류에 따라 다르다. 대부분의 모델의 경우에는 해당 변수를 False로 하면 되지만, 모델이 AllInOneTempo_with_Head인 경우에는 해당 변수의 값을 True로 해야 한다. 해당 변수는 True 혹은 False 값만 가진다.

(4) [변수] bpf_band_dir

해당 변수는 모델의 cfg.yaml 파일에서 data의 bpfed가 True인 경우에만 사용되며, 제공된 sub_band_data의 경로를 저장하여야 한다.

(5) [변수] tempo_measure_method

해당 변수는 “model” 혹은 “interval” 중 하나의 값만 가질 수 있다. 기본적으로 allinone 기반의 모델은 tempo(BPM) 값도 모델을 통해서 예측하는데 해당 방식으로 tempo 값을 예측하는 것이 “model”이다. 그러나 해당 방식의 정확도가 모델의 종류와 학습 방식에 따라 매번 다를 수 있기 때문에 “interval”이라는 추가 방식도 함께 제공한다. “interval” 방식은 모델 자체의 tempo 값 예측 방식을 사용하지 않고 이전에 예측한 박자들의 시간 간격들을 계산하여 tempo 값을 예측한다.

(6) [변수] future_prediction, tolerance

future_prediction 변수는 None 값 혹은 실수형 형태의 값을 가지게 된다. None 값인 경우에는 미래 박자 예측을 수행하지 않고, 실수형 형태의 값으로 입력할 경우에는 해당 값이 미래 박자 예측의 시점으로 입력된다. 즉, 해당 변수에 0.04를 입력할 경우에 시스템은 0.04초 이후 시점의 박자 종류를 예측한다.

tolerance 변수는 future_prediction 변수의 값이 None이 아닌 경우에만 작성한다. tolerance 변수는 어느 범위의 값을 박자로 허용할 것인지 결정해주는 함수이다. 미래

박자 예측 방식은 이전의 박자들을 기준으로 다음 박자의 시점을 예측하고 해당 시점이 미래 시점과 가까우면 미래 시점에 박자가 있다고 예측한다. 그 가까움의 정도를 결정하는 것이 tolerance 변수이다. 예를 들어, future_prediction 변수에 저장된 값이 0.1초이고 tolerance 변수에 저장된 값이 0.07초라고 가정하면, 현재 프레임으로부터 0.1초 이후의 시점과 다음 박자의 시점의 시간 차이가 0.07초 이내이면 0.1초 이후에 박자가 있다고 시스템이 예측하는 방식이다.

(7) [변수] debug_dl

해당 변수는 디버깅을 수행할지 말지를 결정하는 변수이다. 일반적으로는 False 값을 사용하지만 시스템의 분석을 위해서는 True 값을 사용하면 된다. 해당 변수가 True일 경우에, Jetson 보드가 오디오 데이터를 수신하는 시점들과 모델의 예측 결과를 송신하는 시점들을 csv 파일로 저장시켜준다.

3) server.py

(1) 코드 요약 설명

해당 코드는 Jetson 보드 내 메인 프로그램이며, 해당 프로그램에서는 Device로부터 오디오 데이터를 프레임 단위로 수신하고, 해당 오디오 데이터를 모델의 입력으로 넣어 박자 유형을 예측한다. 예측한 결과는 바로 Device로 전송해준다. 해당 파일을 시작하면 본격적인 프로그램 시작에 앞서 Model, UART, Ethernet setup을 우선적으로 진행하고 이후에는 모델의 초기화를 진행한다. setup과 초기화 작업이 끝나면 Program Start 영역의 코드들이 수행되어 본격적인 프로그램이 실행된다.

(2) [함수] save_data_to_csv

- **역할:** 오디오를 수신한 시점과 비트 예측 결과를 송신한 시점들을 csv 파일에 저장하는 함수
- **주요 변수:** “send_time_history”는 비트 예측 결과 송신 시점 값이 저장된 변수이며, “receive_time_history”는 오디오 수신 시점 값이 저장된 변수이다.

(3) Model setup 영역

해당 영역은 “Model setup”이라 주석 표기되어 있다. 모델은 util.py에서 TensorRTAllInOne 클래스를 import하여 모델을 로드한다. 그리고 해당 영역에서 다음의 변수들을 선언하고 일부 변수들은 초기값을 설정해준다.

- **depth** 오디오 데이터의 bit depth를 의미하며 2를 반드시 사용
- **sample_rate** 오디오 데이터의 샘플률이며 모델의 세팅을 따라감
- **slen** 오디오 한 프레임의 길이이자 오디오 데이터 송신 간격으로 초(s) 단위 (모델의 세팅을 따라감)
- **mslen** slen을 ms 단위로 변환한 값이 저장되어 있음
- **BUUFER_SIZE** 모델에 입력되는 오디오 버퍼의 사이즈를 의미하며 모델의 세팅을 따라감

- **global_buffer** 오디오 버퍼가 저장될 변수 (초기값은 모두 0으로 채움)
- **beat_buffer** 오디오 버퍼가 아닌 비트 버퍼로 과거 프레임들의 beat, downbeat 확률값이 저장되는 변수로 5초 길이로 고정 (초기값은 모두 0으로 채움)
- **beat_time_list** 이전 beat 및 downbeat 시점들의 시간값이 저장된 변수로 50개의 값들이 저장 (초기값은 모두 0으로 채움)
- **tempo_list** 이전에 예측한 tempo 값들이 저장된 변수로 50개의 값들이 저장 (초기값은 모두 0으로 채움)

(4) UART setup, Ethernet setup 영역

UART setup에서는 모델의 예측 결과를 송신하기 위해 사용되는 UART 통신의 초기 세팅을 수행한다. Ethernet setup에서는 오디오 데이터를 수신할 때 사용되는 이더넷 통신의 초기 세팅을 수행한다. 두 setup에 필요한 값들은 com_setup.py 파일에서 변경 및 수정이 가능하다.

(5) Initialization 영역

모델이 처음 GPU에 로드되고 첫 추론 과정을 진행할 때에 시간이 오래 소요되기 때문에 안정적인 시스템 동작을 위해서는 본격적인 프로그램 시작 전에 dummy 데이터를 이용해 초기화를 진행해주어야 한다. 모델의 “process” 및 “postprocessing” 메서드의 입력 및 출력이 수정될 경우 해당 코드 또한 그에 맞게 수정해야 한다.

(6) Program Start 영역

해당 영역은 while 문을 통해 수행된다. “data = conn.recv(chunk_size)” 코드에서 오디오 데이터가 수신될 경우에만 모델 처리가 수행된다. 수신된 오디오로부터 먼저 에너지 값을 계산하여 sound_magnitude에 저장하고 곧바로 오디오 정규화가 수행된다. 정규환 된 오디오는 global_buffer에 업데이트된다. global_buffer는 모델의 입력으로 들어가 모델의 출력이 result 변수에 저장된다.

이 result는 model의 get_prob 메서드에 입력으로 들어가 tempo(BPM), beat_buffer(global_buffer의 프레임 별 박자 예측 결과), beat_prob(해당 프레임의 beat 확률), downbeat_prob(해당 프레임의 downbeat 확률) 등을 반환한다.

이 후 , model의 postprocessing 메서드를 통해 최종으로 박자를 예측하는데 이는 beat_type이라는 변수에 저장된다. 추가로 해당 메서드를 통해 last_beat_time(가장 최근 beat 혹은 downbeat의 시점값), tempo(BPM), beat_time_list, index_for_prob를 반환한다. 이 때 index_for_prob는 미래 박자 예측 시 필요한 값이며, 가장 마지막 beat 혹은 downbeat의 beat_buffer index 값이다.

4) util.py

(1) [함수] predict_future_beat_type

- **역할:** 현재 프레임부터 delay 만큼 이후 시점의 박자를 예측하는 함수
- **입력 변수:**

[last_beat_time]	지금까지의 가장 마지막 beat 혹은 downbeat의 시점값
[frame_no]	현재 오디오 프레임
[beat_times]	*Postprocessor 참조
[beat_position]	*Postprocessor 참조
[tempo]	예측한 tempo(BPM) 값 혹은 None (“interval” 방식의 경우, 본 함수에서 해당 변수는 None 값이 들어감)
[delay]	미래 프레임의 시점을 결정하는 변수 (현재 프레임으로부터 delay만큼 이후 시점이 미래 프레임이 됨)
[tolerance]	*com_setup.py 참조
[buffer_length]	오디오 버퍼의 길이에서 한 프레임을 뺀 시간 길이 (기본값 4.98초는 버퍼 길이 5초에서 프레임 길이 0.02초가 제외된 값을 의미)
[fslen]	오디오 프레임의 길이 (기본값은 0.02초)
[beat_time_list]	*Model setup 영역 참조

- **주요 변수:**

[time_diffs]	time_diffs는 두가지가 있는데, elif 문에 있는 time_diffs는 “interval” 경우에서 과거 beat 및 downbeat 간의 간격을 계산하여 저장하는 변수를 의미함. 이후 등장하는 time_diffs는 미래의 가능한 beat 시점들과 알고 싶은 미래 시점의 시간 차이값이 저장된 변수.
[within_tolerance]	time_diffs에서 실제로 tolerance보다 작은 값의 inedx
[closest_beat_index]	within_tolerance에서 가장 앞에 있는 index
[temp_beat_time]	미래 박자로 예측된 시점의 절대 시간값
[is_not_close]	temp_beat_time과 last_beat_time이 0.33초보다 크면 True, 작으면 False 값을 가지는 변수
[last_downbeat_index]	beat_times와 beat_position에서 downbeat가 없다고 예측된 경우 해당 변수는 None 값을 가지면, 있을 경우 가장 마지막 downbeat의 index 값을 가지는 변수 (index는 beat_position 기준)
[expected_downbeat_indices]	예상되는 downbeat index 값들이 저장되는 변수
[beat_type]	예측한 비트 종류 (0: NoBeat, 1: Beat, 2: Downbeat)

- **코드 설명:**

```
if tempo is not None:
    beat_interval = 60 / tempo
elif beat_times.size(0) > 1:
    time_diffs = beat_times[1:] - beat_times[:-1]
    time_diffs = time_diffs[time_diffs != 0]
    beat_interval = torch.mean(time_diffs).item()
```

com_setup.py 파일에서 tempo_measure_method가 “model”일 경우, tempo 값이 None이 아니기 때문에 해당 tempo 값을 바탕으로 평균 비트 간격을 계산한다. “interval”일 경우에는 tempo 값이 None이기 때문에 해당 코드를 통해 평균 비트 간격을 계산한다. 평균 비트 간격은 beat_interval 변수에 저장된다.

```
time_diffs = np.abs(buffer_length + delay - (float(beat_times[-1]) + np.arange(1,5) * beat_interval))
within_tolerance = np.where(time_diffs<=tolerance)[0]
```

time_diff를 계산할 때 buffer_length+delay는 버퍼를 기준으로 미래 시점을 의미한다. float(beat_times[-1])+np.arange(1,5)*beat_interval은 버퍼에서의 가장 마지막 비트 시점에서 평균 비트 간격을 4번 더하여 가능한 미래 박자 시점들을 계산한 것이다. 이 두 값을 빼주어 가능한 미래 박자 시점들과 알고 싶은 미래 시점들의 시간 차이를 계산해준다. within_tolerance는 이 time_diffs에서 tolerance 이내로 들어오는 값들만 여과해준 것이다.

```
if len(within_tolerance) > 0: # A beat/downbeat is predicted for the future frame
    closest_beat_index = within_tolerance[0]
    temp_beat_time = float(float(beat_times[-1]) + closest_beat_index * beat_interval - buffer_leng
    is_not_close = abs(last_beat_time - temp_beat_time) > 0.33
    if is_not_close and temp_beat_time > last_beat_time:
        last_beat_time = temp_beat_time # Update last predicted beat
        last_downbeat_index = len(beat_position) - 1 - beat_position[::-1].index(1) if 1 in beat_pc
        if last_downbeat_index is not None: # If a downbeat was predicted within the past 5 seconds
            expected_downbeat_indices = last_downbeat_index + 4 * np.arange(1,3)
            closest_beat_index += len(beat_position)
            beat_type = 2 if closest_beat_index in expected_downbeat_indices else 1
        else:
            beat_type = 2 if len(beat_position) == 3 else 1
        if beat_time_list is not None:
            beat_time_list = torch.cat((beat_time_list[1:], torch.tensor([last_beat_time], dtype=be
    else:
        beat_type = 0
else:
    beat_type = 0
```

within_tolerance에 값이 있는 경우 실행되며, 이는 미래 시점에 beat나 downbeat가 있다고 예측한 경우이다. is_not_close는 해당 시점이 이전에 예측한 가장 마지막 박자 시점과 가까운지 판단한 boolean 값이 저장된 변수이다. 가깝다면 이미 예측한 박자 시점이기 때문에 해당 경우에는 nobeat(0)를 반환하고, 먼 경우에는 새로운 미래 박자 시점이라는 것을 의미하기 때문에 박자 유형을 예측해야 한다. 만약 beat_position 중에 downbeat가 있을 경우, last_downbeat_index가 None 값이 아니기 때문에 이를 기준으로 미래 박자 시점이 downbeat인지 아닌지를 판단한다. last_downbeat_index가 None 값인 경우에는 버퍼 내에서 downbeat로 예측한 된 것이 없기 때문에 beat가 3개인 경우에만 미래 박자는 downbeat(2)라 예측을 하고 그렇지 않은 경우에는 downbeat 판단의 근거가 부족하기 때문에 beat(1)라고 예측한다.

(2) [함수] get_probs_realtime

이는 buffer가 None인 경우와 None이 아닌 경우 두가지로 나뉘는데 None인 경우는 com_setup.py 파일에서 설명한 dont_need_buffer가 True인 경우를 의미한다. AllInOneTempo_with_Head 모델의 경우에는 beat buffer가 필요없어 해당 함수에서 buffer 변수는 None 값이 저장된다. 이 경우와 None 값이 아닌 경우에는 logits 값들에서 beat와 downbeat 확률들을 계산하는 방식이 조금 다르다.

tempo_measure_method가 "model"인 경우에는 모델의 logits 값으로부터 tempo(BPM) 값을 계산할 수 있기 때문에 softmax 함수를 이용해 계산한다. 만약, 이 값이 180보다 크면 반으로 나누고 60보다 작으면 2배로 키운다. tempo(BPM) 계산 방식이 "model"이 아니라 "interval"인 경우, tempo를 None 값으로 저장하고 이후 과정에서 다른 함수를 통해 tempo를 계산한다.

(3) [클래스] Postprocessor

해당 클래스는 후처리를 수행하기 위한 클래스이면 주요 변수로는 kernel_size와 threshold_beat, threshold_downbeat 등이 있다. 작동 방식은 max pooling을 사용한다. beat_buffer를 입력받는데 __call__(pred)에서 pred 자리에 beat_buffer가 들어가게 된다. beat_buffer는 버퍼 내 프레임 별 beat 및 downbeat의 확률값이 저장되어 있다. 이 때, beat와 downbeat 각각에 대해서 kernel_size만큼의 max pooling이 수행되고 이 때 확률 값이 각각 threshold_beat와 threshold_downbeat를 넘겨야 한다. 이를 통해서 확률이 가장 높은 beat 및 downbeat index만 남게 된다. 이러한 후처리 과정을 통해 버퍼 내의 beat 및 downbeat의 위치를 예측한다. 이 때, sorted_times는 예측한 beat와 downbeat의 시간값이 저장되며 이 시간값은 0~buffer_length 사이의 값을 가진다. 예를 들어, buffer_length가 5초인 경우, 마지막 프레임 길이만큼을 제외한 0~4.98초 중에서 프레임 단위의 값을 가질 수 있다. sorted_positions는 sorted_times에 대응되는 관계인데 해당 시점의 예측값이 beat인지 downbeat인지에 대한 박자 유형이 저장되어 있다. downbeat는 1이 저장되어 있고 beat는 2가 저장되어 있다. sorted_times와 sorted_positions는 결합되어 반환된다.

(4) [클래스] TORCH_SPECT

해당 클래스는 모델의 전처리를 위한 클래스이다. process_audio 메서드를 통해서 STFT(Short-Time Fourier Transform)이 진행되고, 필요한 경우에 따라 sub band filtering도 함께 수행된다. 해당 모델에 사용되는 변수들은 모두 모델의 세팅을 따른다.

(5) [클래스] TensorRTAllInOne

- **역할:** 박자 추적 모델을 TensorRT 엔진 형태로 불러와 실질적으로 박자 예측을 수행하는 모델 클래스
- **주요 변수:**

  [self.buffer] 모델의 입력으로 들어갈 오디오 스펙트로그램이 저장될 변수

  [self.context] 모델의 TensorRT 엔진이 로드되는 변수
- **함수 설명:**

  [process] self.proc이 TORCH_SPECT 클래스로 만든 인스턴스인데, 이를 통해 입력 오디오를 스펙트로그램으로 변환하고 이를 모델의 입력으로 넣어 추론하여 출력 logits 값을 반환하는 함수

  [get_prob] get_probs_realtime 함수를 사용하여 beat_buffer, tempo, prob_beat, prob_downbeat를 반환하는 함수

  [postprocessing] 먼저 beat_time_list가 None인 경우는 tempo(BPM) 계산 방식이 “interval”인 경우를 의미한다. 해당 함수에서는 우선 이 경우에 tempo부터 계산한다. 그리고 self.postprocessor_downbeat를 통해 후처리를 진행한다. “if future_prediction is None” 문을 통해 미래 박자 예측을 하지 않을 경우에는 현재 예측한 박자가 유효한지와 유효한 경우는 박자 유형을 예측한다. 미래 박자를 예측하는 경우에는 predic_future_beat_type 함수를 통해 미래 박자를 예측한다.

5) make_trt.py

- **역할:** 학습된 모델을 불러와 TensorRT 엔진 파일로 변환하는 프로그램
- **주요 변수:** *[pretrained_dir]*
  모델의 checkpoint, config 파일이 저장된 경로이자 동시에 해당 모델의 onnx 파일 및 TensorRT 엔진 파일이 저장될 디렉토리 경로.
  (단, 저장된 경로값을 수정하기 위해서는 com_setup.py 파일에서 수정해야 한다.)
- **코드 설명:**

```python
def load_model(pretrained_dir='pretrained/', device='cpu'):
    ckpt_path = os.path.join(pretrained_dir, "model.ckpt")
    cfg_path = os.path.join(pretrained_dir, "cfg.yaml")

    cfg = OmegaConf.load(cfg_path)
    model_type = cfg['model']

    checkpoint = torch.load(ckpt_path, map_location=device)
    checkpoint["state_dict"] = {k.replace("model.", ""): v for k, v in checkpoint["state_dict"].items()}

    if model_type == "allinonetempo":
        model = AllInOneTempo(cfg).to(device)
    elif model_type == "nobufferallin1":
        model = AllInOneTempo_with_Head(cfg).to(device)
    else:
        raise NotImplementedError(f'Unknown model: {model_type}')

    model.load_state_dict(checkpoint['state_dict'])
    model.eval()
    print("Model Loading Complete!")

    return model, cfg
```

pretrained_dir 변수에 저장된 디렉토리 경로에서 model.ckpt와 cfg.yaml 파일을 찾아 파이토치 모델을 로드하는 함수이다. 함수가 반환하는 값은 model, cfg이며 이때 model은 pytorch 모델 형태이며, cfg는 omegaconf의 configuration 형태이다.

```python
if os.path.isfile(os.path.join(pdir,"model.onnx")) == False:
    model,cfg = load_model(pretrained_dir=pdir, device='cpu')
    dummy_input = torch.randn(1, 1, cfg.buffer_length*cfg.fps, 83)
    torch.onnx.export(
        model,                          # PyTorch model
        dummy_input,                    # dummpy input
        os.path.join(pdir,"model.onnx"),                    # ONNX file path for save
        export_params=True,             # weights of model
        opset_version=11,               # ONNX opset version
        do_constant_folding=True,       # optimization of constant folding
        input_names=['input'],          # input name
        output_names=['output'],        # output name
        verbose=True,
    )
```

pytorch 모델 형태의 박자 추적 모델을 ONNX 모델 형태로 중간 변환을 해주는 코드이다. 해당 코드를 실행하면 model.onnx 파일이 pretrained_dir 변수의 디렉토리 경로에 저장된다. 만약, 이미 해당 경로에 model.onnx 파일이 있을 경우에는 수행되지 않는다.

```python
command = ["trtexec", f"--onnx={os.path.join(pdir,'model.onnx')}", f"--saveEngine={os.path.join(pdir,'model.engine')}"]

try:
    result = subprocess.run(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

    if result.returncode == 0:
        print("TensorRT engine conversion succeeded!")
        print(result.stdout)
    else:
        print("TensorRT engine conversion failed!")
        print(result.stderr)

except FileNotFoundError:
    print("The 'trtexec' command was not found. Please ensure that TensorRT is installed correctly.")
except Exception as e:
    print(f"An unknown error occurred: {e}")
```

해당 코드는 onnx 형태로 변환된 박자 추적 모델을 TensorRT 엔진 형태로 최종 변환을 해주는 코드이다. 해당 코드를 실행하면 model.engine 파일이 pretrained_dir 변수의 디렉토리 경로에 저장된다.

5. Device 주요 코드

1) client.py

(1) 코드 요약 설명

해당 코드는 Device 내 메인 프로그램이며, 해당 프로그램에서 오디오 파일을 불러 프레임 단위로 오디오 데이터를 Jetson 보드로 전송해주고, 동시에 Jetson 보드로부터 모델의 예측 결과를 수신한다. 또한, osc 통신을 통해서 음악과 예측 결과의 메트로놈 소리를 동시에 함께 재생해준다. 소리를 재생시켜주는 코드는 해당 프로그램이 아닌 osc_server.py에 구현되어 있다.

(2) [함수] main

메인 함수는 setup 코드들과 프로그램 실행 코드들도 이루어져 있다. setup 코드의 경우에는 UART setup, 이더넷 setup, 초기화로 이루어져 있으며 해당 영역은 주석으로 표시되어 있다. Jetson 보드에서 모델의 초기 추론에서 많은 시간이 소요되기 때문에 초기화 단계를 추가해주었다. 초기화(Initialization) 단계에서는 오디오 데이터와 동일한 형태의 dummy 데이터를 전송하고 예측 결과를 수신하여, Jetson 보드에서 모델의 초기 추론을 프로그램 실행 전에 선행하여 이후 모델의 일정한 추론 시간을 보장해주었다. setup과 초기화가 모두 진행되면, threading을 통해 receive_data 함수와 send_audio_over_ethernet 함수를 동시에 수행시킨다.

(3) [함수] save_data_to_csv

- **역할:** 오디오를 송신한 시점과 비트 예측 결과를 수신한 시점들을 csv 파일에 저장하는 함수
- **주요 변수:** “send_time_history”는 오디오 송신 시점 값이 저장된 변수이며, “receive_time_history”는 비트 예측 결과 수신 시점 값이 저장된 변수이다.

(4) [함수] send_audio_over_ethernet

- **역할:** 이더넷 통신을 사용하여 프레임 단위로 오디오 데이터를 Jetson 보드로 전송해주는 함수
- **주요 변수:**

  [chunk_size] 오디오 한 프레임의 크기 (byte 단위)

  [elapsed_time] 프로그램 시작부터 현재까지 소요된 시간
- **코드 설명:**

```
chunk = audio_data[i:i+chunk_size]
if debug: ...
sock.send(chunk)
```

chunk_size를 통해 한 프레임 길이의 오디오 데이터를 chunk라는 변수에 저장하고, sock.send(chunk)라는 코드를 통해 오디오 데이터를 이더넷 통신으로 전송한다.

(5) [함수] recevice_data

- **역할:** UART 통신을 사용하여 프레임 단위로 비트 예측 결과를 Jetson 보드로부터 수신하는 함수
- **주요 변수:**

[frame_no]	(uint16, 2bytes) 수신한 비트 예측 결과의 프레임 번호
[energy]	(uint16, 2bytes) 오디오 에너지 값
[beat_type]	(uint8, 1byte) 예측한 비트 종류 (0: NoBeat, 1: Beat, 2: Downbeat)
[beat_prob]	(float, 4bytes) 비트 확률
[downbeat_prob]	(float, 4bytes) 다운비트 확률
[tempo]	(uint16, 2bytes) 예측한 BPM 값

- **코드 설명:**

```
data = ser.read(15)
frame_no, energy, beat_type, beat_prob, downbeat_prob, tempo = struct.unpack(">HHBffH", data)
```

위 작성된 주요 변수들의 전체 크기는 총 15 bytes이기 때문에, 첫 번째 줄 코드에서는 UART 통신을 통해 정확히 15 bytes 크기로 데이터를 읽는다. 두 번째 줄 코드에서는 읽어들인 데이터를 [2,2,1,4,4,2] bytes 형태로 unpack하여 각 주요 변수에 예측 결과값을 저장한다.