실시간 추론 시스템 API 레퍼런스

개요

Violence Detection 실시간 추론 시스템의 모든 클래스, 메서드, 함수의 상세한 API 레퍼런스를 제공한다.

목차

파이프라인 API

InferencePipeline

클래스 정의

```
class InferencePipeline(BasePipeline):
    """실시가ㄴ 추론 파이프라인 메인 클래스"""
```

생성자

```
def __init__(self, config: Dict[str, Any])
```

매개변수:

- `config` (Dict[str, Any]): 통합 설정 딕셔너리

반환값: None

예외:

- `ConfigurationError`: 설정 오류 시
- `ModuleInitializationError`: 모듈 초기화 실패 시

사용 예시:

```
config = load_config('config.yaml')
pipeline = InferencePipeline(config)
```

메서드

initialize_pipeline()

```
def initialize_pipeline(self) -> bool
```

설명: 파이프라인의 모든 모듈을 초기화합니다.

반환값:

- `bool`: 초기화 성공 여부

예외:

- `PoseEstimatorError`: 포즈 추정기 초기화 실패
- `TrackerError`: 추적기 초기화 실패
- `ClassifierError`: 분류기 초기화 실패

사용 예시:

```
if pipeline.initialize_pipeline():
    print("Pipeline initialized successfully")
else:
    print("Pipeline initialization failed")
```

run_realtime_mode()

```
def run_realtime_mode(self, input_source: str) -> bool
```

설명: 실시간 모드로 파이프라인을 실행합니다.

매개변수:

- `input_source` (str): 입력 소스 (파일 경로, RTSP URL, 웹캠 인덱스)

반환값:

- `bool`: 실행 성공 여부

예외:

- `InputSourceError`: 입력 소스 오류
- `RuntimeError`: 실행 중 오류

사용 예시:

```
# 비디오 파일
pipeline.run_realtime_mode("/path/to/video.mp4")

# RTSP 스트림
pipeline.run_realtime_mode("rtsp://192.168.1.100:554/stream")

# 웹캠
pipeline.run_realtime_mode("0")
```

process_frame()

```
def process_frame(self, frame: np.ndarray, frame_idx: int) ->
Tuple[FramePoses, Dict[str, Any]]
```

설명: 단일 프레임을 처리합니다.

매개변수:

- `frame` (np.ndarray): 입력 프레임 (H×W×3)
- `frame_idx` (int): 프레임 인덱스

반환값:

- `Tuple[FramePoses, Dict[str, Any]]`: (처리된 포즈 데이터, 오버레이 정보)

예외:

- `FrameProcessingError`: 프레임 처리 실패

get_performance_stats()

```
def get_performance_stats(self) -> Dict[str, Any]
```

설명: 성능 통계를 반환합니다.

반환값:

```
{
    'overall_fps': float,
```

```
    'pose_estimation_fps': float,
    'tracking_fps': float,
    'scoring_fps': float,
    'classification_fps': float,
    'avg_processing_time': float,
    'windows_classified': int,
    'total_alerts': int,
    'frames_skipped': int
}
```

입력 관리 API

RealtimeInputManager

클래스 정의

```
class RealtimeInputManager:
    """실시가ㄴ 입력 관리자"""
```

생성자

```
def __init__(self,
             input_source: Union[str, int],
             buffer_size: int = 10,
             target_fps: Optional[int] = None,
             frame_skip: int = 0)
```

매개변수:

- `input_source` (Union[str, int]): 입력 소스
- `buffer_size` (int): 프레임 버퍼 크기 (기본값: 10)
- `target_fps` (Optional[int]): 목표 FPS (기본값: None)
- `frame_skip` (int): 건너뛸 프레임 수 (기본값: 0)

메서드

start()

```
def start(self) -> bool
```

설명: 입력 스트림을 시작합니다.

반환값:

- `bool`: 시작 성공 여부

예외:

- `InputSourceError`: 입력 소스 열기 실패
- `CameraError`: 카메라 접근 실패

get_frame()

```
def get_frame(self) -> Optional[Tuple[np.ndarray, int]]
```

설명: 다음 프레임을 가져옵니다.

반환값:

- `Optional[Tuple[np.ndarray, int]]`: (프레임, 프레임 인덱스) 또는 None

예외:

- `FrameCaptureError`: 프레임 캡처 실패

stop()

```
def stop(self) -> None
```

설명: 입력 스트림을 중지합니다.

get_video_info()

```
def get_video_info(self) -> Dict[str, Any]
```

설명: 비디오 정보를 반환합니다.

반환값:

```
{
    'width': int,
    'height': int,
    'fps': float,
    'frame_count': int,
    'duration': float,
    'source_type': str
}
```

포즈 추정 API

RTMOONNXEstimator

클래스 정의

```
class RTMOONNXEstimator(BasePoseEstimator):
    """RTMO ONNX 포즈 추정기"""
```

생성자

```
def __init__(self, config: Dict[str, Any])
```

매개변수:

- `config` (Dict[str, Any]): 포즈 추정기 설정

설정 예시:

```
config = {
    'model_path': '/path/to/rtmo.onnx',
    'device': 'cuda:0',
    'score_threshold': 0.3,
    'input_size': [640, 640]
}
```

메서드

estimate_poses()

```
def estimate_poses(self, frame: np.ndarray) -> FramePoses
```

설명: 프레임에서 포즈를 추정합니다.

매개변수:

- `frame` (np.ndarray): 입력 프레임 (H×W×3)

반환값:

- `FramePoses`: 추정된 포즈 데이터

예외:

- `InferenceError`: 추론 실패
- `PreprocessingError`: 전처리 실패

set_score_threshold()

```
def set_score_threshold(self, threshold: float) -> None
```

설명: 점수 임계값을 설정합니다.

매개변수:

- `threshold` (float): 점수 임계값 (0.0-1.0)

get_model_info()

```
def get_model_info(self) -> Dict[str, Any]
```

설명: 모델 정보를 반환합니다.

반환값:

```
{
    'model_path': str,
    'input_shape': List[int],
    'output_shape': List[int],
    'num_keypoints': int,
    'device': str
}
```

추적 API

ByteTrackerWrapper

클래스 정의

```
class ByteTrackerWrapper(BaseTracker):
    """ByteTracker 래퍼 클래스"""
```

생성자

```
def __init__(self, config: Dict[str, Any])
```

매개변수:

- `config` (Dict[str, Any]): 추적기 설정

설정 예시:

```
config = {
    'track_thresh': 0.5,
    'track_buffer': 30,
    'match_thresh': 0.8,
    'frame_rate': 30
}
```

메서드

track()

```
def track(self, poses: FramePoses) -> FramePoses
```

설명: 포즈에 추적 ID를 할당합니다.

매개변수:

- `poses` (FramePoses): 입력 포즈 데이터

반환값:

- `FramePoses`: 추적 ID가 할당된 포즈 데이터

예외:

- `TrackingError`: 추적 실패

reset()

```
def reset(self) -> None
```

설명: 추적기를 초기화합니다.

get_active_tracks()

```
def get_active_tracks(self) -> List[int]
```

설명: 활성 추적 ID 목록을 반환합니다.

반환값:

- `List[int]`: 활성 추적 ID 목록

분류 API

STGCNActionClassifier

클래스 정의

```
class STGCNActionClassifier(BaseActionClassifier):
    """ST-GCN++ 행동 분류기"""
```

생성자

```
def __init__(self, config: Dict[str, Any])
```

매개변수:

- `config` (Dict[str, Any]): 분류기 설정

설정 예시:

```
config = {
    'checkpoint_path': '/path/to/stgcn.pth',
    'config_file': '/path/to/config.py',
    'device': 'cuda:0',
    'num_classes': 2,
    'confidence_threshold': 0.4,
    'window_size': 100
}
```

메서드

classify_window()

```
def classify_window(self, window_data: np.ndarray) -> ClassificationResult
```

설명: 윈도우 데이터를 분류합니다.

매개변수:

- `window_data` (np.ndarray): 윈도우 데이터 (T×N×V×C)

반환값:

- `ClassificationResult`: 분류 결과

예외:

- `ClassificationError`: 분류 실패
- `InvalidWindowDataError`: 잘못된 윈도우 데이터

set_confidence_threshold()

```
def set_confidence_threshold(self, threshold: float) -> None
```

설명: 신뢰도 임계값을 설정합니다.

매개변수:

- `threshold` (float): 신뢰도 임계값 (0.0-1.0)

get_class_names()

```
def get_class_names(self) -> List[str]
```

설명: 클래스 이름 목록을 반환합니다.

반환값:

- `List[str]`: 클래스 이름 목록 (예: ['NonFight', 'Fight'])

warmup()

```
def warmup(self, num_runs: int = 1) -> None
```

설명: 모델을 워밍업합니다.

매개변수:

- `num_runs` (int): 워밍업 실행 횟수 (기본값: 1)

이벤트 관리 API

EventManager

클래스 정의

```
class EventManager:
    """이벤트 관리 시스템"""
```

생성자

```
def __init__(self, config: EventConfig)
```

매개변수:

- `config` (EventConfig): 이벤트 설정

메서드

process_classification_result()

```
def process_classification_result(self, result: Dict[str, Any]) ->
Optional[EventData]
```

설명: 분류 결과를 처리하여 이벤트를 생성합니다.

매개변수:

- `result` (Dict[str, Any]): 분류 결과

입력 형식:

```
result = {
    'window_id': int,
    'prediction': str,  # 'violence' or 'normal'
    'confidence': float,
    'timestamp': float,
    'probabilities': List[float]
}
```

반환값:

- `Optional[EventData]`: 생성된 이벤트 또는 None

add_event_callback()

```

```

```
def add_event_callback(self, event_type: EventType, callback:
Callable[[EventData], None]) -> None
```

설명: 이벤트 콜백을 등록합니다.

매개변수:

- `event_type` (EventType): 이벤트 타입
- `callback` (Callable): 콜백 함수

사용 예시:

```
def on_violence_start(event_data: EventData):
    print(f"Violence detected: {event_data.confidence:.3f}")

event_manager.add_event_callback(EventType.VIOLENCE_START,
on_violence_start)
```

get_current_status()

```
def get_current_status(self) -> Dict[str, Any]
```

설명: 현재 이벤트 상태를 반환합니다.

반환값:

```
{
    'event_active': bool,
    'consecutive_violence': int,
    'consecutive_normal': int,
    'current_event_duration': Optional[float],
    'last_event_time': Optional[float]
}
```

get_event_history()

```
def get_event_history(self, limit: Optional[int] = None) -> List[EventData]
```

설명: 이벤트 히스토리를 반환합니다.

매개변수:

- `limit` (Optional[int]): 반환할 이벤트 수 제한

반환값:

- `List[EventData]`: 이벤트 히스토리

EventConfig

클래스 정의

```
@dataclass
class EventConfig:
    """이벤트 관리 설정"""
```

필드

```
alert_threshold: float = 0.7               # 폭력 탐지 신뢰도 임계가ㅅ
min_consecutive_detections: int = 3        # 연속 탐지 최소 횟수
normal_threshold: float = 0.5              # 정상 상태 신뢰도 임계가ㅅ
min_consecutive_normal: int = 5            # 연속 정상 최소 횟수
min_event_duration: float = 2.0            # 최소 이벤트 지속 시가ㄴ (초)
max_event_duration: float = 300.0          # 최대 이벤트 지속 시가ㄴ (초)
cooldown_duration: float = 10.0            # 이벤트 쿨다운 시가ㄴ (초)
enable_ongoing_alerts: bool = True         # 진행 중 알림 활성화
ongoing_alert_interval: float = 30.0       # 진행 중 알림 가ㄴ격 (초)
save_event_log: bool = True               # 이벤트 로그 저장 여부
event_log_format: str = "json"            # 로그 형식 (json/csv)
event_log_path: str = "output/event_logs" # 로그 저장 경로
```

EventLogger

클래스 정의

```
class EventLogger:
    """이벤트 로깅 시스템"""
```

생성자

```
def __init__(self,
             log_path: str = "output/event_logs",
             log_format: str = "json",
             enable_logging: bool = True)
```

메서드

log_event()

```
def log_event(self, event_data: EventData) -> bool
```

설명: 이벤트를 로그에 기록합니다.

매개변수:

- `event_data` (EventData): 이벤트 데이터

반환값:

- `bool`: 로그 기록 성공 여부

set_session()

```
def set_session(self, session_id: Optional[str] = None) -> str
```

설명: 새로운 세션을 시작합니다.

매개변수:

- `session_id` (Optional[str]): 세션 ID (None이면 자동 생성)

반환값:

- `str`: 세션 ID

시각화 API

RealtimeVisualizer

클래스 정의

```
class RealtimeVisualizer:
    """실시가ㄴ 시가ㄱ화 클래스"""
```

생성자

```
def __init__(self,
             window_name: str = "Violence Detection",
             display_width: int = 1280,
```

```
                display_height: int = 720,
                fps_limit: int = 30,
                save_output: bool = False,
                output_path: Optional[str] = None,
                max_persons: int = 4,
                processing_mode: str = "realtime",
                confidence_threshold: float = 0.4)
```

메서드

start_display()

```
def start_display(self) -> None
```

설명: 디스플레이 창을 시작합니다.

show_frame()

```
def show_frame(self,
               frame: np.ndarray,
               poses: Optional[FramePoses] = None,
               classification: Optional[Dict[str, Any]] = None,
               additional_info: Optional[Dict[str, Any]] = None,
               overlay_data: Optional[Dict[str, Any]] = None) -> bool
```

설명: 프레임을 화면에 표시합니다.

매개변수:

- `frame` (np.ndarray): 원본 프레임
- `poses` (Optional[FramePoses]): 포즈 데이터
- `classification` (Optional[Dict[str, Any]]): 분류 결과
- `additional_info` (Optional[Dict[str, Any]]): 추가 정보
- `overlay_data` (Optional[Dict[str, Any]]): 오버레이 데이터

반환값:

- `bool`: 계속 표시할지 여부

update_event_history()

```
def update_event_history(self, event_data: Dict[str, Any]) -> None
```

설명: 이벤트 히스토리를 업데이트합니다.

매개변수:

- `event_data` (Dict[str, Any]): 이벤트 데이터

update_classification_history()

```
def update_classification_history(self, classification: Dict[str, Any]) ->
None
```

설명: 분류 결과 히스토리를 업데이트합니다.

매개변수:

- `classification` (Dict[str, Any]): 분류 결과

stop_display()

```
def stop_display(self) -> None
```

설명: 디스플레이 창을 종료합니다.

유틸리티 API

SlidingWindowProcessor

클래스 정의

```
class SlidingWindowProcessor:
    """슬라이딩 윈도우 프로세서"""
```

생성자

```
def __init__(self,
             window_size: int = 100,
             window_stride: int = 50,
             max_persons: int = 4,
             coordinate_dimensions: int = 2)
```

메서드

add_frame_data()

```
def add_frame_data(self, frame_poses: FramePoses) -> None
```

설명: 프레임 데이터를 윈도우에 추가합니다.

매개변수:

- `frame_poses` (FramePoses): 프레임 포즈 데이터

is_ready()

```
def is_ready(self) -> bool
```

설명: 윈도우가 분류 준비 상태인지 확인합니다.

반환값:

- `bool`: 준비 상태 여부

get_window_data()

```
def get_window_data(self) -> Tuple[np.ndarray, int]
```

설명: 윈도우 데이터를 반환합니다.

반환값:

- `Tuple[np.ndarray, int]`: (윈도우 데이터, 윈도우 ID)

reset()

```
def reset(self) -> None
```

설명: 윈도우 프로세서를 초기화합니다.

ModuleFactory

클래스 정의

```
class ModuleFactory:
    """모듈 팩토리 클래스"""
```

클래스 메서드

register_pose_estimator()

```
@classmethod
def register_pose_estimator(cls,
                            name: str,
                            estimator_class: Type[BasePoseEstimator],
                            default_config: Dict[str, Any]) -> None
```

설명: 포즈 추정기를 등록합니다.

create_pose_estimator()

```
@classmethod
def create_pose_estimator(cls,
                          name: str,
                          config: Dict[str, Any]) -> BasePoseEstimator
```

설명: 포즈 추정기를 생성합니다.

register_tracker()

```
@classmethod
def register_tracker(cls,
                     name: str,
                     tracker_class: Type[BaseTracker],
                     default_config: Dict[str, Any]) -> None
```

create_tracker()

```
@classmethod
def create_tracker(cls,
                   name: str,
                   config: Dict[str, Any]) -> BaseTracker
```

register_classifier()

```
@classmethod
def register_classifier(cls,
                        name: str,
                        classifier_class: Type[BaseActionClassifier],
                        default_config: Dict[str, Any]) -> None
```

create_classifier()

```
@classmethod
def create_classifier(cls,
                      name: str,
                      config: Dict[str, Any]) -> BaseActionClassifier
```

데이터 구조

FramePoses

클래스 정의

```
@dataclass
class FramePoses:
    """프레임 내 모든 포즈 데이터"""
```

필드

```
persons: List[PersonPose]
frame_idx: int
timestamp: float
video_info: Dict[str, Any]
```

메서드

```
def get_valid_persons(self) -> List[PersonPose]:
    """유효한 인체 포즈만 반환"""

def to_dict(self) -> Dict[str, Any]:
    """딕셔너리로 변환"""
```

PersonPose

클래스 정의

```
@dataclass
class PersonPose:
    """개별 인체 포즈 데이터"""
```

필드

```
keypoints: np.ndarray              # (17, 3) [x, y, confidence]
bbox: Optional[np.ndarray]         # [x1, y1, x2, y2]
track_id: Optional[int]            # 추적 ID
score: float                       # 포즈 점수
detection_confidence: float        # 탐지 신뢰도
```

메서드

```
def get_keypoint(self, index: int) -> Tuple[float, float, float]:
    """특정 키포인트 반환"""

def get_bbox_center(self) -> Tuple[float, float]:
    """바운딩 박스 중심점 반환"""

def is_valid(self) -> bool:
    """유효한 포즈인지 확인"""
```

ClassificationResult

클래스 정의

```
@dataclass
class ClassificationResult:
    """분류 결과 데이터"""
```

필드

```
prediction: int                    # 예측 클래스 인덱스
confidence: float                  # 신뢰도
probabilities: List[float]         # 클래스별 확률
processing_time: float             # 처리 시가ㄴ
window_id: int                     # 윈도우 ID
timestamp: float                   # 타임스탬프
```

메서드

```
def get_predicted_class_name(self, class_names: List[str]) -> str:
    """예측된 클래스 이름 반환"""
```

```
    def to_dict(self) -> Dict[str, Any]:
        """딕셔너리로 변환"""
```

EventData

클래스 정의

```
@dataclass
class EventData:
    """이벤트 데이터"""
```

필드

```
event_type: EventType
timestamp: float
window_id: int
confidence: float
duration: Optional[float] = None
additional_info: Optional[Dict[str, Any]] = None
```

메서드

```
def to_dict(self) -> Dict[str, Any]:
    """딕셔너리로 변환"""

def to_json(self) -> str:
    """JSON 문자열로 변환"""
```

EventType

열거형 정의

```
class EventType(Enum):
    """이벤트 타입"""
    VIOLENCE_START = "violence_start"
    VIOLENCE_END = "violence_end"
    VIOLENCE_ONGOING = "violence_ongoing"
    NORMAL = "normal"
```

예외 처리

기본 예외

ViolenceDetectionError

```
class ViolenceDetectionError(Exception):
    """기본 시스템 예외"""
```

ConfigurationError

```
class ConfigurationError(ViolenceDetectionError):
    """설정 관련 예외"""
```

ModuleInitializationError

```
class ModuleInitializationError(ViolenceDetectionError):
    """모듈 초기화 예외"""
```

입력 관련 예외

InputSourceError

```
class InputSourceError(ViolenceDetectionError):
    """입력 소스 예외"""
```

FrameCaptureError

```
class FrameCaptureError(ViolenceDetectionError):
    """프레임 캡처 예외"""
```

처리 관련 예외

PoseEstimationError

```
class PoseEstimationError(ViolenceDetectionError):
    """포즈 추정 예외"""
```

TrackingError

```
class TrackingError(ViolenceDetectionError):
    """추적 예외"""
```

ClassificationError

```
class ClassificationError(ViolenceDetectionError):
    """분류 예외"""
```

EventProcessingError

```
class EventProcessingError(ViolenceDetectionError):
    """이벤트 처리 예외"""
```

예외 처리 예시

```
try:
    pipeline = InferencePipeline(config)
    pipeline.initialize_pipeline()
    pipeline.run_realtime_mode(input_source)

except ConfigurationError as e:
    logging.error(f"Configuration error: {e}")

except ModuleInitializationError as e:
    logging.error(f"Module initialization failed: {e}")

except InputSourceError as e:
    logging.error(f"Input source error: {e}")

except PoseEstimationError as e:
    logging.warning(f"Pose estimation failed: {e}")
    # 빈 포즈로 계속 진행

except ClassificationError as e:
    logging.warning(f"Classification failed: {e}")
    # 분류 없이 계속 진행

except ViolenceDetectionError as e:
    logging.error(f"System error: {e}")

except Exception as e:
    logging.critical(f"Unexpected error: {e}")
```

사용 예시

기본 사용법

```
from recognizer.pipelines.inference.pipeline import InferencePipeline
from recognizer.utils.config_loader import load_config
from recognizer.events.event_types import EventType

# 1. 설정 로드
config = load_config('config.yaml')

# 2. 파이프라인 초기화
pipeline = InferencePipeline(config)

# 3. 이벤트 콜백 등록
def on_violence_detected(event_data):
    print(f"Violence detected! Confidence: {event_data.confidence:.3f}")

pipeline.event_manager.add_event_callback(
    EventType.VIOLENCE_START,
    on_violence_detected
)

# 4. 파이프라인 실행
try:
    if pipeline.initialize_pipeline():
        success = pipeline.run_realtime_mode("/path/to/video.mp4")

        if success:
            # 성능 통계 출력
            stats = pipeline.get_performance_stats()
            print(f"Processing completed. Overall FPS:
{stats['overall_fps']:.1f}")
        else:
            print("Processing failed")
    else:
        print("Pipeline initialization failed")

except Exception as e:
    print(f"Error: {e}")
```

고급 사용법

```
import numpy as np
from recognizer.pipelines.inference.pipeline import InferencePipeline
from recognizer.utils.realtime_input import RealtimeInputManager
from recognizer.visualization.realtime_visualizer import RealtimeVisualizer

# 커스텀 입력 및 시가ㄱ화 설정
input_manager = RealtimeInputManager(
```

```
    input_source="rtsp://192.168.1.100:554/stream",
    buffer_size=15,
    target_fps=25,
    frame_skip=1
)

visualizer = RealtimeVisualizer(
    window_name="Custom Violence Detection",
    display_width=1920,
    display_height=1080,
    fps_limit=25,
    confidence_threshold=0.5
)

# 파이프라인과 연동
pipeline = InferencePipeline(config)
pipeline.initialize_pipeline()

# 수동 프레임 처리 루프
input_manager.start()
visualizer.start_display()

try:
    while True:
        frame_data = input_manager.get_frame()
        if frame_data is None:
            break

        frame, frame_idx = frame_data
        poses, overlay_data = pipeline.process_frame(frame, frame_idx)

        # 추가 정보 수집
        additional_info = pipeline.get_performance_stats()

        # 시가ㄱ화
        should_continue = visualizer.show_frame(
            frame=frame,
            poses=poses,
            additional_info=additional_info,
            overlay_data=overlay_data
        )

        if not should_continue:
            break

finally:
    input_manager.stop()
    visualizer.stop_display()
```