“여기 common\_utils.py는 세 가지 주요 기능을 제공합니다.

**1. load\_csv(csv\_path, tag) 함수**

**‘load\_csv’ 함수는 CSV 파일에서 특정 센서 태그(tag)의 시계열 데이터를 읽어와, PyTorch가 바로 처리할 수 있는 텐서 형태로 반환합니다.  
구체적으로는 다음 단계를 거칩니다.**

1. **구분자(Separator) 자동 감지**
   * 파일 맨 위 한 줄(hdr)을 읽어서, 세미콜론(;)과 쉼표(,) 중 어떤 구분자가 더 많이 쓰였는지 검사합니다.
   * 세미콜론이 더 많으면 sep=";"로, 그렇지 않으면 sep=","로 설정해서 pandas.read\_csv()를 호출합니다.
   * 이렇게 하면 CSV 파일마다 구분자가 달라도 따로 수정할 필요 없이 자동 처리됩니다.
2. **CSV를 문자열(dtype=str)로 읽고, 타임스탬프와 원하는 센서 태그만 추출**
   * 전체 열을 문자열로 읽은 뒤 열 목록(cols)을 얻습니다.
   * ts\_col 변수에, 열 이름을 모두 정규화(normalize)한 뒤 소문자로 바꾸어 “timestamp” 문구가 포함된 열을 자동으로 찾아 저장합니다.
   * 만약 tag 인자로 받은 센서명이 열 목록에 없다면 ValueError를 날려서 “해당 태그가 CSV에 없다”라고 바로 알립니다.
   * 이후 df = df[[ts\_col, tag]]로 ‘타임스탬프 열’과 ‘센서 열(tag)’만 남긴 부분집합으로 데이터프레임을 만듭니다.
3. **타임스탬프 전처리**
   * df[ts\_col].str.strip()으로 문자열 앞뒤 공백을 제거한 뒤,
   * pd.to\_datetime(..., format=DATE\_FMT, dayfirst=True)를 사용해 “일/월/년 시:분:초 AM/PM” 형식(DATE\_FMT = "%d/%m/%Y %I:%M:%S %p")으로 정확하게 날짜형으로 변환합니다.
   * 변환한 뒤 set\_index(ts\_col, inplace=True)로 그 열을 인덱스로 설정합니다.
4. **중복 타임스탬프 제거**
   * df.index.duplicated(keep='first')를 사용해 만약 동일한 타임스탬프가 여러 행에 등장하면, 첫 번째 행만 남기고 나머지를 삭제합니다.
5. **센서 값 실수(float32)로 변환**
   * df[tag] 열은 콤마(,)가 소수점 구분자로 쓰인 문자열이기 때문에, str.replace(",", ".", regex=False)로 콤마를 점으로 바꿉니다.
   * 그 뒤 .astype("float32")를 통해 32비트 실수 텐서로 변환합니다.
6. **1초 해상도 재샘플(resample) + 선형 보간(interpolate)**
   * df[tag].asfreq("1s")를 호출하면, 인덱스가 1초 단위로 촘촘히 늘어난 새로운 시리즈를 생성합니다.
   * 원본에 값이 없는 구간은 NaN이 되는데, interpolate("linear")를 적용하면 선형 보간 방식으로 비어 있는 값을 채워줍니다.
   * 이렇게 하면, 타임스탬프 간격이 불규칙한 원본을 1초 단위 시계열로 깔끔하게 정리할 수 있습니다.
7. **PyTorch Tensor로 변환 후 반환**
   * 최종적으로 series.values(NumPy 배열)를 torch.tensor(..., dtype=torch.float32)로 바꿔서 1차원 Tensor를 얻습니다.
   * 함수는 (tensor, series.index) 두 개를 반환하므로, 모델 학습 시 “시계열 데이터(Tensor)”와 “타임스탬프 인덱스(DatetimeIndex)”를 둘 다 활용할 수 있습니다.

**2. ARIMAModel(nn.Module) 클래스**

**ARIMAModel은 PyTorch 기반으로 ARIMA(p, d, q) 모델을 직접 구현한 서브클래스입니다.  
ARIMA 모델이란, ‘AR(p) + I(d) + MA(q)’ 구조를 갖는 시계열 예측 모델입니다.**

1. **초기화(\_\_init\_\_)**
   * \_\_init\_\_(self, p=2, d=1, q=2)로 세 가지 핵심 파라미터를 기본값 (2,1,2)로 받습니다.
     + p=2 → 과거 2시점 관측치(AR 차수)
     + d=1 → 한 번 차분해서 안정화(I 차수)
     + q=2 → 과거 2시점 잔차 관측(MA 차수)
   * self.phi = nn.Parameter(torch.zeros(p))
     + 이 벡터는 AR(p) 계수(φ₁, φ₂, …, φₚ)를 의미하며, 모두 0으로 초기화된 후 역전파로 학습됩니다.
   * self.theta = nn.Parameter(torch.zeros(q))
     + 이 벡터는 MA(q) 계수(θ₁, θ₂, …, θ\_q)를 의미합니다. 역시 0으로 초기화됩니다.
   * self.mu = nn.Parameter(torch.zeros(1))
     + 이 스칼라는 모델의 편향(bias 또는 상수항 μ) 역할을 하며, 마찬가지로 학습 가능한 파라미터입니다.
2. **순전파(forward)**
   * 입력 y: torch.Tensor는 시계열 데이터이며, 1차원(T,) 또는 2차원(B, T) 형태로 들어올 수 있습니다.
   * 먼저 if y.dim() == 1: y = y.unsqueeze(0)로 1차원일 때 배치 차원(B=1)을 추가해 (1, T)로 만듭니다.
   * B, Tseq = y.shape로 배치 크기 B와 전체 시퀀스 길이 Tseq를 각각 얻습니다.
   * d, p, q = self.d, self.p, self.q로 클래스에 저장된 차수 값을 로컬 변수에 할당합니다.
   * T = Tseq – d는 “차분 뒤 실제 예측 가능한 타임스탬프 개수”를 의미합니다.
   * eps = y.new\_zeros(B, T+q)
     + 이 텐서는 MA 계수를 계산할 때 필요한 과거 잔차를 저장하기 위한 공간입니다. 길이가 T+q인 이유는 초기 q개 스텝에 대한 잔차도 저장 공간을 확보하기 위함입니다.
   * yd = y[:, d:] – y[:, :-d]
     + 원본 시계열 y를 d번 차분한 결과입니다. 예를 들어 d=1이면 yd[t] = y[t+1] – y[t] 형태로 1차 차분된 시계열이 나옵니다.
   * 메인 루프: for t in range(p, T):
     + t가 p부터 시작하는 이유는, AR(p) 부분 계산을 위해 최소 p개 과거 관측치가 필요하기 때문입니다.
     + **AR 계산**:

python

복사편집

ar = (self.phi \* torch.flip(y[:, d+t-p:d+t], dims=[1])).sum(dim=1)

* + - * y[:, d+t-p : d+t]는, “현재 스텝 t의 차분 이후 시점 기준으로” 과거 p개 관측치 구간입니다.
      * torch.flip(..., dims=[1])로 시계열을 뒤집어 [y\_{t-1}, y\_{t-2}, …, y\_{t-p}] 순서로 만든 뒤, self.phi와 차원별 곱을 하고 sum해서 AR 예측 기여도를 계산합니다.
    - **MA 계산**:

python

복사편집

ma = (self.theta \* torch.flip(eps[:, t-q:t], dims=[1])).sum(dim=1)

* + - * eps[:, t-q:t]는 과거 q개 시점의 잔차(ε)입니다. 이 구간을 뒤집어서 [ε\_{t-1}, ε\_{t-2}, …, ε\_{t-q}]로 만든 뒤, self.theta 가중치 합을 구합니다.
    - **예측값(pred)**: pred = self.mu + ar + ma
      * 상수항 μ(self.mu), AR 계산값, MA 계산값을 더해 현재 시점 차분값에 대한 예측을 만듭니다.
    - **잔차 저장**: eps[:, t] = yd[:, t] – pred
      * 실제 d차분 시계열 yd[:, t]에서 예측값을 빼면 현재 시점 잔차가 나옵니다. 이 값을 eps 버퍼에 저장해서 이후 MA 계산 때 사용합니다.
  + 반복이 끝나면, 앞 p개 스텝은 초기 버퍼 용도로 계산한 값이므로 제외하고 return eps[:, p:] 로 실제 사용 가능한 잔차만 반환합니다. 반환 차원은 (B, T-p)가 됩니다.

**3. get\_device() 함수**

**GPU(쿠다) 사용 여부를 확인하고, 사용 가능한 GPU 이름과 개수를 출력한 후,  
만약 사용할 수 있다면 ('cuda', gpu\_count)를,  
없으면 ('cpu', 0)을 반환합니다.**

1. print("CUDA available:", torch.cuda.is\_available())
   * 가장 먼저 현재 환경에서 GPU 사용 여부(torch.cuda.is\_available())를 프린트합니다.
2. if torch.cuda.is\_available():
   * GPU가 사용 가능하면
     + count = torch.cuda.device\_count()로 사용 가능한 GPU 개수를 가져와서
     + for idx in range(count): print(f"[{idx}] {torch.cuda.get\_device\_name(idx)}") 형태로 각 GPU의 이름을 출력합니다.
     + 이후 return torch.device("cuda"), count로 디바이스 객체와 GPU 개수를 반환합니다.
3. else:
   * GPU를 사용할 수 없는 환경이면
     + print("GPU 미감지: CPU로 실행") 메시지를 띄우고
     + return torch.device("cpu"), 0을 반환합니다.