“여기 train\_arima.py는 ARIMA 모델을 학습하는 전체 파이프라인을 담당하는 스크립트입니다.

**4) 모델 생성**

* base\_model = ARIMAModel().to(device)로 ARIMA 모델 객체를 생성한 뒤,  
  • ARIMAModel은 common\_utils.py에 정의된 PyTorch 모듈로, 기본 차수 (p=2, d=1, q=2)를 가집니다.  
  • .to(device)를 통해 GPU 메모리 혹은 CPU 메모리로 이동합니다.
* model = DataParallel(base\_model) if n\_gpu > 1 else base\_model  
  • 만약 GPU 개수(n\_gpu)가 2개 이상이라면, DataParallel(base\_model)로 모델을 감싸서 자동으로 여러 GPU에 학습 배치를 분산합니다.  
  • 그렇지 않으면 단일 base\_model만 사용합니다.
* optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)  
  • Adam 옵티마이저를 생성하며, 학습률은 args.lr에서 지정한 값을 사용합니다.  
  • model.parameters()로 ARIMA 모델의 학습 가능한 파라미터(phi, theta, mu)를 모두 전달합니다.

**5) 학습 수행 (train\_loop)**

다음 train\_loop() 함수로 실제 학습을 진행합니다:

def train\_loop(model, data, epochs, optimizer):

"""

• model : ARIMA 모델

• data : 입력 텐서 (배치 차원 포함)

• epochs : 학습 반복 횟수 (정수)

• optimizer: 옵티마이저 객체

"""

# 5-1) 학습 시작 시각 및 퍼포먼스 타이머 기록

start\_wall = datetime.now() # 실제 날짜·시간

start\_perf = time.perf\_counter() # 고해상도 성능 타이머

print(f"학습 시작: {start\_wall:%Y-%m-%d %H:%M:%S}")

prev\_perf = start\_perf

# 5-2) Epoch 루프: trange로 진행바 표시

for epoch in trange(1, epochs + 1, desc="Epochs"):

# (1) 기울기 초기화

optimizer.zero\_grad()

# (2) 순전파 + MSE 손실 계산

loss = torch.mean((model(data)) \*\* 2)

# (3) 역전파

loss.backward()

# (4) 파라미터 업데이트

optimizer.step()

# (5) 경과 시간 계산 (밀리초)

now\_perf = time.perf\_counter()

delta\_ms = (now\_perf - prev\_perf) \* 1000

prev\_perf = now\_perf

# (6) 에폭별 로그 출력

print(f"[{epoch:03d}/{epochs}] loss={loss.item():.6f} | Δt={delta\_ms:.1f} ms")

# 5-3) 전체 학습 소요 시간 출력

total\_time = time.perf\_counter() - start\_perf

print(f"학습 종료: {datetime.now():%Y-%m-%d %H:%M:%S} (총 {total\_time:.2f}s)")

return total\_time

* start\_wall과 start\_perf는 “실제 시각”과 “고해상도 타이머”를 기록합니다.
* trange(1, epochs+1, desc="Epochs")는 터미널에서 진행바(progress bar)를 표시해 줍니다.
* 매 에폭마다
  1. optimizer.zero\_grad()로 이전 기울기를 초기화
  2. loss = torch.mean((model(data)) \*\* 2)로 모델에 데이터를 순전파하여 “잔차(residual)”를 구하고 MSE(평균제곱오차) 손실로 계산
  3. loss.backward()로 기울기를 계산(역전파)
  4. optimizer.step()으로 파라미터 업데이트
  5. delta\_ms = (now\_perf - prev\_perf)\*1000로 “이 에폭이 소요된 시간(밀리초)”을 계산
  6. print(f"[{epoch:03d}/{epochs}] loss=… | Δt=… ms")로 손실 값과 에폭 소요 시간을 로그로 남깁니다.
* 마지막으로 전체 학습이 끝나면 total\_time = ...으로 총 소요 시간을 계산해 “학습 종료: 시각 (총 XX.XXs)”를 출력하고 반환합니다.

**6) 모델 저장**

* 학습이 끝난 후, Path("models").mkdir(exist\_ok=True)로 models 폴더가 없으면 생성합니다.
* torch.save(base\_model.state\_dict(), f"models/arima\_{args.tag}.pt")로  
  • base\_model.state\_dict()는 ARIMA 모델의 학습된 파라미터(phi, theta, mu)만 저장합니다.  
  • 파일명은 arima\_<센서태그>.pt 형식으로 만들어서, 나중에 predict\_arima.py에서 바로 로드할 수 있도록 합니다.
* 마지막 print(f"모델 저장 완료: models/arima\_{args.tag}.pt")로 저장된 경로를 확인할 수 있게 합니다.