7장 트랜스포머 (~GPT)

- 트랜스포머(Transformer)는 2017년 「Attention is All You Need」로 소개된 신경망 아키텍처로, 순환(Sequential) 처리나 반복 연결(Recurrent Connections)에 의존하지 않고 **셀프 어텐션(Self-Attention)**으로 입력 토큰 간 관계를 직접 모델링한다.
- 긴 시퀀스를 효율적으로 처리하며, 대용량 데이터셋에서 높은 성능과 학습 효율을 보인다. 기계 번역·언어 모델링·텍스트 요약 등 자연어 처리 전반에 널리 쓰인다.

모델 학습 방식 개요

- 본 장의 트랜스포머 기반 모델들은 **오토 인코딩(Auto-Encoding)**, **자기 회귀(Auto-Regressive)** 또는 두 방법의 조합으로 학습한다.
- 오토 인코딩: 문장의 일부를 빈칸(마스크)으로 만들고 해당 빈칸의 단어를 예측하며, 예측 시 토큰의 **양옆**을 참조하므로 **양방향 구조**를 가진다(인코더).
- 자기 회귀: 이전 단어들만 주어졌을 때 **다음 단어**를 예측하는 **단방향 구조**를 가진다(디코더).

트랜스포머 구조

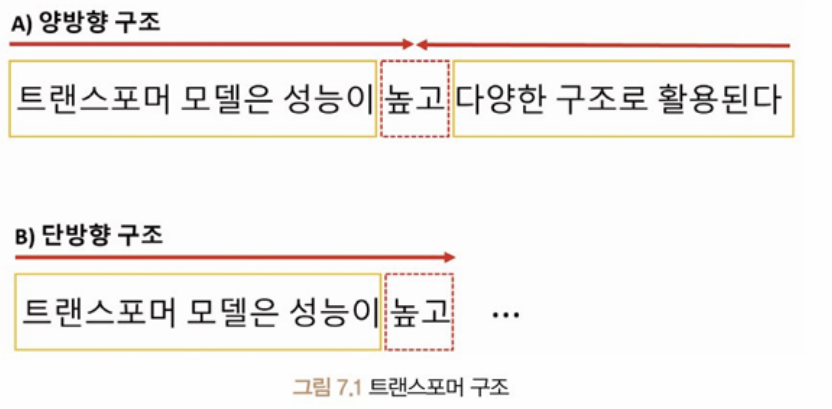


그림 7.1 트랜스포머 구조

- A) **양방향 구조**: "트랜스포머 모델은 성능이 높고, 다양한 구조로 활용된다"를 입력으로, 양옆 문맥을 참조하며 학습.
- B) **단방향 구조**: 같은 문장을 입력으로, 왼쪽(이전) 토큰만 참조하며 다음 토큰을 예측.

트랜스포머 모델 구조

표 7.1 트랜스포머 모델 구조

모델	학습구조	학습방법	학습 방향성
BERT	인코더	오토 인코딩	양방향
GPT	디코더	자기 회귀	단방향
BART	인코더+디코더	오토 인코딩+자기 회귀	양방향+단반향
ELECTRA	인코더+판별기	오토 인코딩+대체 토큰 탐지	양방향
T5	인코더+디코더	오토 인코딩+자기 회귀+다양한 자연어 처리 작업을 학습	양방향

Transformer

- 트랜스포머는 순환 신경망/합성곱과 달리 **어텐션 메커니즘**만을 사용하여 시퀀스 의존성을 모델링한다. 인코더와 디코더 간 상호작용으로 입력 시퀀스의 중요한 부분에 집중해 적절한 출력을 생성한다.
- **장점**: 빠른 학습 속도, 병렬 처리 가능, 대규모 데이터에서 높은 성능, 문장 전체 정보를 고려해 긴 문장에서도 성능 유지.

그림 7.2 트랜스포머 모델 구조(Q·K·V, 인코더/디코더 블록)

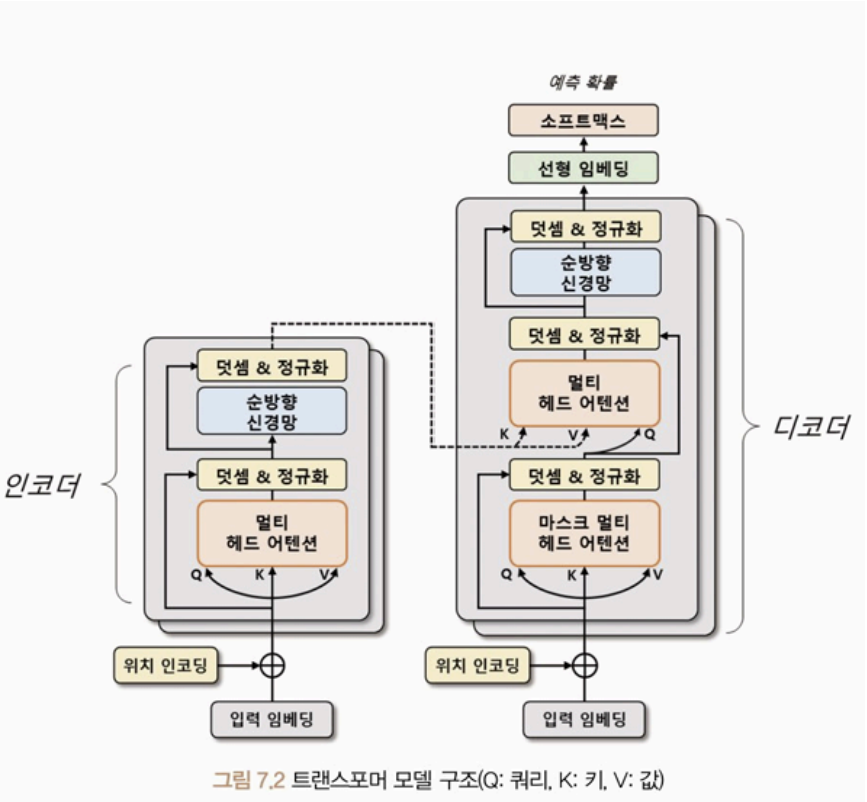


그림 7.2 트랜스포머 모델 구조(Q: 쿼리, K: 키, V: 값)

- 인코더·디코더는 각각 N개의 **트랜스포머 블록**으로 구성. 각 블록은
  - **멀티 헤드 어텐션(Multi-Head Attention)**
  - **덧셈 & 정규화**
  - **순방향 신경망(Feed-Forward)**

    로 이루어진다.
- **Self-Attention**: 입력 시퀀스에서 **Query(Q)**, **Key(K)**, **Value(V)**를 정의하고, Q-K 유사도를 계산해 그 가중치로 V를 가중합한다. 계산된 어텐션 결과는 각 토큰의 임베딩을 갱신한다.
- **순방향 신경망**: 어텐션으로 얻은 임베딩을 추가로 변환하여 의미 표현을 강화한다.

입력 임베딩과 위치 인코딩

- 입력 토큰은 **임베딩 벡터**로 변환되며, 트랜스포머는 시퀀스를 **병렬**로 처리하므로 토큰의 **순서 정보**를 별도로 제공해야 한다 → **위치 인코딩(Positional Encoding)** 사용.
- 인코더 출력은 디코더가 참조하고, 디코더는 **마스크드 멀티 헤드 어텐션**으로 **미래 토큰을 가리지(masking)** 않도록 하며, 최종 출력은 **선형+소프트맥스**를 거쳐 언어 모델 확률로 변환된다.

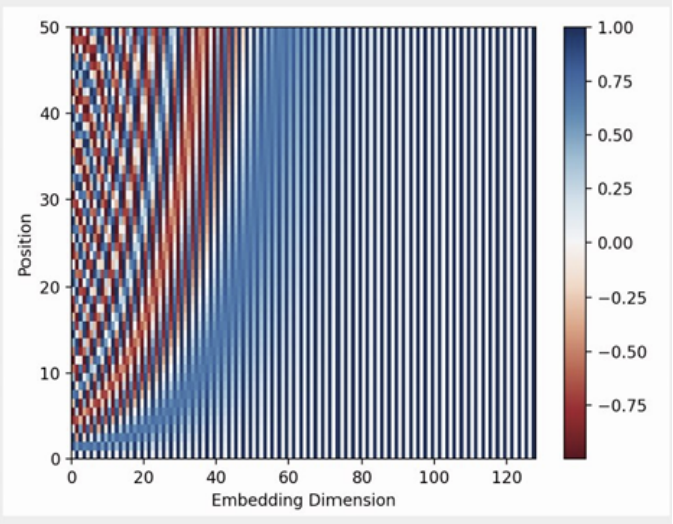
예제 7.1 위치 인코딩

```
import math
import torch
from torch import nn
from matplotlib import pyplot as plt

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.
0) / d_model))

        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
```

```
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[: x.size(0)]
        return self.dropout(x)

encoding = PositionalEncoding(d_model=128, max_len=50)

plt.pcolormesh(encoding.pe.numpy().squeeze(), cmap="RdBu")
plt.xlabel("Embedding Dimension")
plt.xlim((0, 128))
plt.ylabel("Position")
plt.colorbar()
plt.show()
```

출력 결과

- Position×Embedding Dimension 격자에서 **sin/cos 주기 패턴**이 나타난다.



수식 7.1 위치 인코딩

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{\,2i/d_{\text{model}}}}\right), \quad \mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\,2i/d_{\text{model}}}}\right)$$

- pos: 시퀀스 내 위치, iii: 임베딩 차원 인덱스(짝수/홀수에 각각 sin/cos 적용).
- $1/10000^{2i/d_{\text{model}}}$: 위치 신호의 **스케일 조절**.

- 코드의 `register_buffer`는 위치 행렬을 **학습 파라미터로 갱신하지 않도록** 고정한다.

용어

- **소스(Source)/타깃(Target)**: 예) 영→한 번역에서 영문=소스, 한글=타깃.
- **마스킹(Masking)**: 디코더가 **현재 이후**의 토큰을 보지 못하도록 가림.

특수 토큰

- 트랜스포머는 단어 토큰 외에 **특수 토큰**으로 문장의 시작/끝을 표시하거나, **마스킹 영역**을 표현한다.
- **BOS**(Beginning of Sentence): 문장 시작, **EOS**(End of Sentence): 문장 종료, **UNK**(Unknown): 사전에 없는 단어, **PAD**(Padding): 문장 길이 맞춤.

그림 7.3 특수 토큰을 활용한 문장 토큰 배열

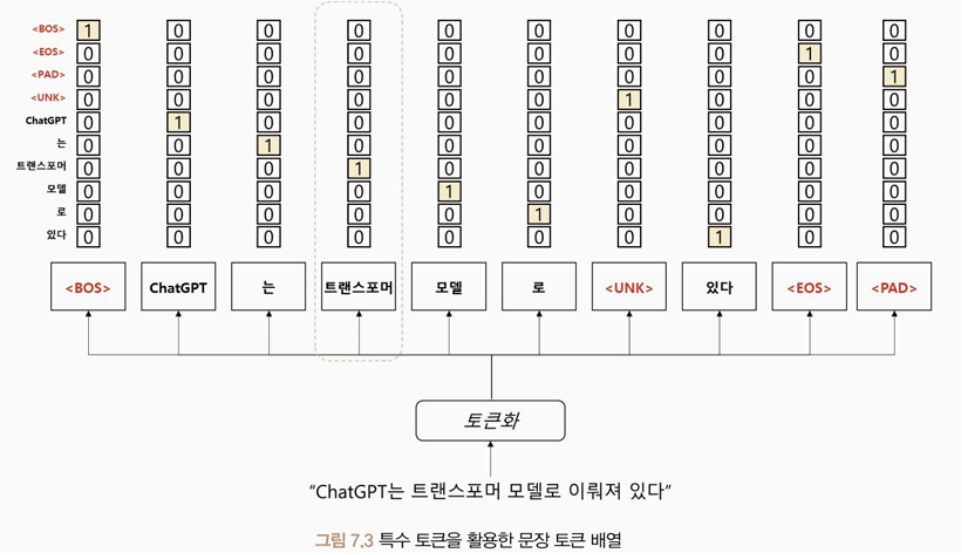


그림 7.3 특수 토큰을 활용한 문장 토큰 배열

- 예시 문장 "ChatGPT는 트랜스포머 모델로 이루어져 있다"에 <BOS>...<EOS>/<PAD>/<UNK>가 포함된 토큰 시퀀스가 제시된다.

그림 7.4 입력 임베딩 산출 과정

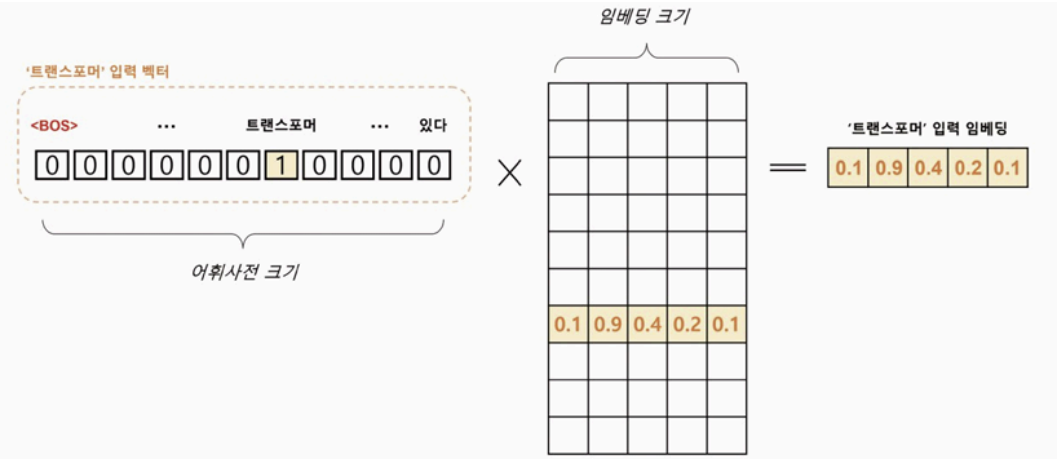


그림 7.4 입력 임베딩 산출 과정

- 원-핫 벡터 [1, V] × 임베딩 행렬 [V, d] → 임베딩 벡터 [1, d].
- 배치화 시 N개 문장·최대 길이 S라면 [N,S,V] → **임베딩 텐서 [N,S,d]** 로 변환되며, 이 임베딩은 트랜스포머 전 계층에서 **공유**된다.

트랜스포머 인코더

- 인코더는 **위치 인코딩이 적용된 소스 임베딩**을 입력받아, 각 계층의 **멀티 헤드 어텐션**과 **순방향 신경망**을 거치며 정보를 추출하고 다음 계층으로 전달한다.
- 인코더 각 시점에서 생성되는 3개 벡터를 각각 **쿼리(Q)**, **키(K)**, **값(V)**로 정의한다(현재 시점의 질문·참조 기준·정보).

그림 7.5 트랜스포머 인코더 블록의 연산 과정

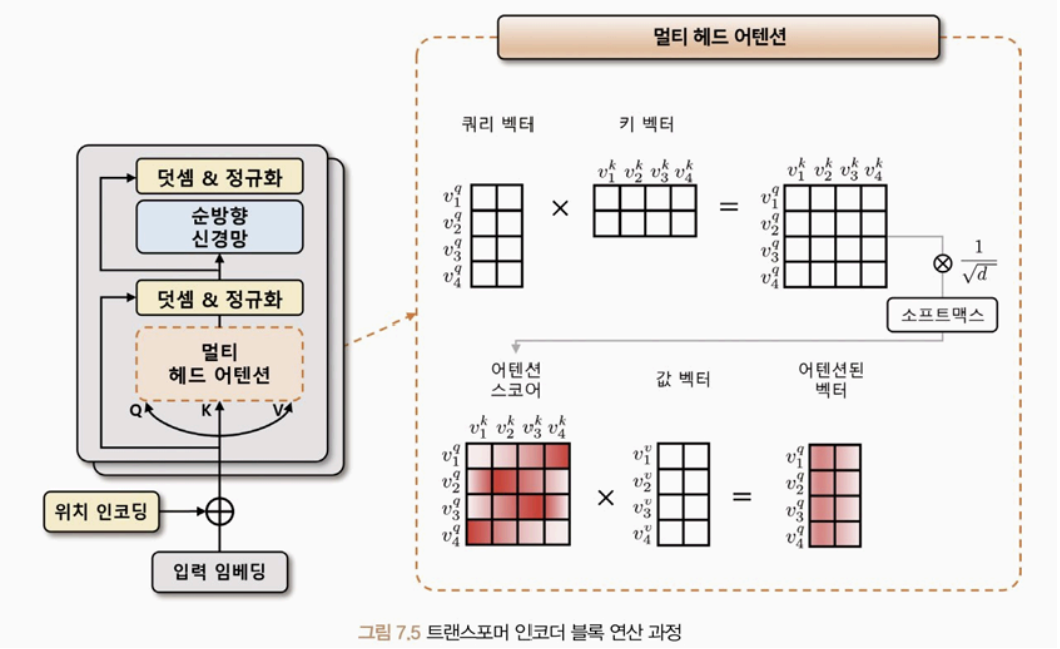


그림 7.5 트랜스포머 인코더 블록 연산 과정

- **Q, K, V** 생성 → 어텐션 스코어 계산(소프트맥스 포함) → 가중합으로 **어텐션 벡터** 도출 → **덧셈 & 정규화**, **순방향 신경망**을 통해 출력.

어텐션 스코어 · 멀티 헤드

- **키 벡터(K)**: 쿼리 벡터와 비교되는 대상 벡터로, 인코더의 각 시점에서 생성.
- **값 벡터(V)**: 어텐션 스코어를 반영해 가중합될 정보 벡터의 역할.
- Q·K·V는 초기에는 무작위에 가깝지만 학습을 거치며 의미 있는 표현을 획득.
- **어텐션 스코어**는 내적 연산으로 계산하고, 차원 d에 대한 보정으로 $\sqrt{d}$로 나눔. 소프트맥스로 정규화하여 확률적으로 표현.
- **수식 7.2 어텐션 스코어 계산식**

$$\text{score}(q^t, k^{t'}) = \text{softmax}\left(\frac{(q^t)^\top k^{t'}}{\sqrt{d}}\right)$$

- **멀티 헤드(Multi-Head)**: 셀프 어텐션을 k개 헤드로 병렬 수행.

  입력 텐서 [N,S,d] → 각 헤드 [N,S,d/k]을 k개 생성 → **병합(concatenation)** 후 [N,S,d]로 복원.

- **덧셈 & 정규화(Add & Norm)**: 멀티 헤드 출력과 이전 입력(잔차 연결)을 더하고 정규화하여 학습 안정화.
- **순방향 신경망**: 선형 임베딩+ReLU(또는 1D 합성곱) → 다시 덧셈 & 정규화.

정리

- 어텐션 블록: Q/K의 내적 → 스코어 행렬 → 소프트맥스 → V 가중합 → 잔차 연결+정규화.
- 멀티 헤드: 여러 헤드의 결과를 **병합**하여 동일 차원으로 반환.

트랜스포머 디코더

- 디코더 입력: **위치 인코딩**이 적용된 타깃 임베딩.
- **인과성(Causality)** 반영: **마스크드 멀티 헤드 어텐션**으로 현재 시점이 **이전 단어만** 참조하도록 마스킹.
  - 첫 번째 쿼리→첫 번째 키/값만, 두 번째 쿼리→둘째까지 ...
  - 마스크 영역은 매우 작은 값($-inf$)로 대체되어 소프트맥스 후 가중치가 0에 수렴.
- 두 번째 어텐션 모듈(인코더–디코더 어텐션): 쿼리=타깃, 키·값=인코더 출력(소스 정보).
- 디코더도 트랜스포머 **디코더 블록**을 여러 개 적층하여 사용.

그림 7.6 디코더 블록 연산 과정

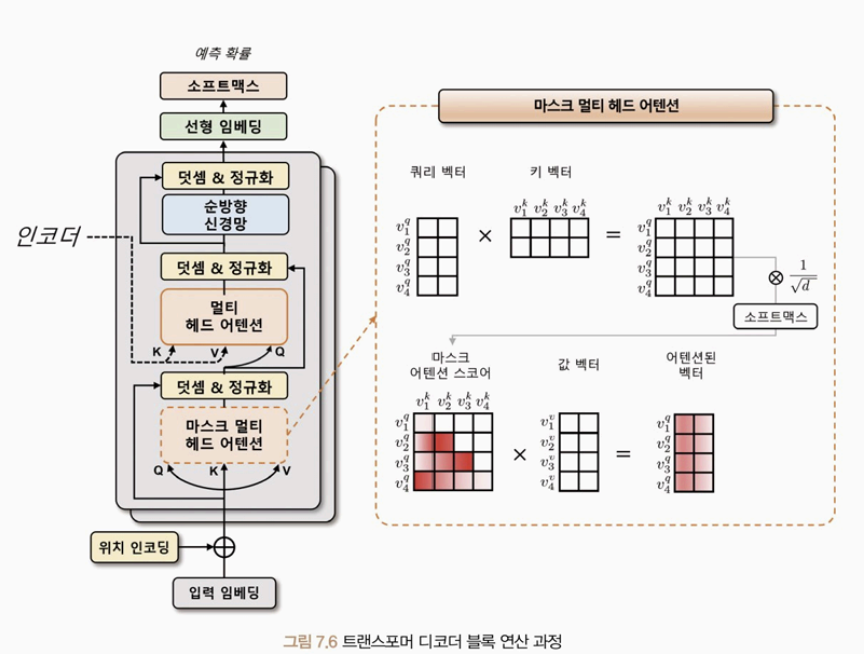


그림 7.6 트랜스포머 디코더 블록 연산 과정

- (1) **마스크드 멀티 헤드 셀프 어텐션**
- (2) **인코더–디코더 멀티 헤드 어텐션**(쿼리=타깃, 키/값=인코더 메모리)

- (3) **순방향 신경망** → 각 단계마다 **덧셈 & 정규화**

표 7.2 인과성을 고려한 디코더 입·출력 예시

표 7.2 인과성을 고려한 디코더 입력과 출력 예시

추론 순서	디코더 입력	디코더 출력
1	[BOS], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]
2	[BOS], 'ChatGPT', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]
3	[BOS], 'ChatGPT', '는', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', [PAD], [PAD], [PAD], [PAD], [PAD]
4	[BOS], 'ChatGPT', '는', '트랜스포머', [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', [PAD], [PAD], [PAD], [PAD]
5	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', [PAD], [PAD], [PAD]
6	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [PAD], [PAD]
7	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [EOS], [PAD]

- 문장 "ChatGPT는 트랜스포머 모델로 이루어져 있다"를 단계별로 생성.
- 각 추론 단계에서 입력은 <BOS>와 **PAD**로 길이를 맞추고, 출력은 한 단어씩 늘어남(마지막에 <EOS> 포함).

모델 실습 — 데이터셋/라이브러리

- 사용 데이터: **Multi30k**(영–독 병렬 말뭉치, 약 30k 문장).
- 설치: 토치 데이터·텍스트와 파일 잠금 유틸리티.

```
pip install torchdata torchtext portalocker
```

- `torchdata`: 대규모 데이터셋 로딩·변환·배치 유틸리티.
- `torchtext`: 파이토치용 텍스트 처리 라이브러리(토크나이저/어휘 사전 등).
- `portalocker`: 데이터 다운로드 중 다중 프로세스 충돌 방지.

예제 7.2 데이터셋 다운로드 및 전처리

```
from torchtext.datasets import Multi30k
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator

def generate_tokens(text_iter, language):
    language_index = {SRC_LANGUAGE: 0, TGT_LANGUAGE: 1}
    for text in text_iter:
        yield token_transform[language](text[language_index[language]])
```

```
SRC_LANGUAGE = "de"
TGT_LANGUAGE = "en"

UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX = 0, 1, 2, 3
special_symbols = ["<unk>", "<pad>", "<bos>", "<eos>"]

token_transform = {
    SRC_LANGUAGE: get_tokenizer("spacy", language="de_core_news_s
m"),
    TGT_LANGUAGE: get_tokenizer("spacy", language="en_core_web_sm"),
}
print("Token Transform:")
print(token_transform)

vocab_transform = {}
for language in [SRC_LANGUAGE, TGT_LANGUAGE]:
    train_iter = Multi30k(split="train", language_pair=(SRC_LANGUAGE, TGT_
LANGUAGE))
    vocab_transform[language] = build_vocab_from_iterator(
        generate_tokens(train_iter, language),
        min_freq=1,
        specials=special_symbols,
        special_first=True,
    )

for language in [SRC_LANGUAGE, TGT_LANGUAGE]:
    vocab_transform[language].set_default_index(UNK_IDX)
```

```
print("Vocab Transform:")
print(vocab_transform)
```

spaCy 형태소 모델은 (예: `python -m spacy download de_core_news_sm`)와 같이 사전 설치 필요.

출력 결과

- `Token Transform:` 언어별 spaCy 토크나이저 객체
- `Vocab Transform:` `{'de': Vocab(), 'en': Vocab()}`

체크리스트

- ☐ `special_symbols` 순서: `<unk>`, `<pad>`, `<bos>`, `<eos>`
- ☐ `set_default_index(UNK_IDX)`로 OOV 처리 기본값 설정

예제 7.3 트랜스포머 모델 구성

코드 — PositionalEncoding / TokenEmbedding / Seq2SeqTransformer

```
import math
import torch
from torch import nn

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.
0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer("pe", pe)
    def forward(self, x):
        x = x + self.pe[: x.size(0)]
        return self.dropout(x)
```

```
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size, emb_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size
    def forward(self, tokens):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)

class Seq2SeqTransformer(nn.Module):
    def __init__(
        self,
        num_encoder_layers,
        num_decoder_layers,
        emb_size,
        max_len,
        src_vocab_size,
        tgt_vocab_size,
        dim_feedforward,
        dropout=0.1,
    ):
        super().__init__()
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(
            d_model=emb_size, max_len=max_len, dropout=dropout
        )
        self.transformer = nn.Transformer(
            d_model=emb_size,
            nhead=nhead,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
        )
        self.generator = nn.Linear(emb_size, tgt_vocab_size)

    def forward(
        self,
```

```
        src,
        trg,
        src_mask,
        tgt_mask,
        src_padding_mask,
        tgt_padding_mask,
        memory_key_padding_mask,
    ):
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        outs = self.transformer(
            src=src_emb,
            tgt=tgt_emb,
            src_mask=src_mask,
            tgt_mask=tgt_mask,
            memory_mask=None,
            src_key_padding_mask=src_padding_mask,
            tgt_key_padding_mask=tgt_padding_mask,
            memory_key_padding_mask=memory_key_padding_mask,
        )
        return self.generator(outs)

    def encode(self, src, src_mask):
        return self.transformer.encoder(
            self.positional_encoding(self.src_tok_emb(src)), src_mask
        )

    def decode(self, tgt, memory, tgt_mask):
        return self.transformer.decoder(
            self.positional_encoding(self.tgt_tok_emb(tgt)), memory, tgt_mask
        )
```

트랜스포머 클래스

```
transformer = torch.nn.Transformer(
    d_model=512,
    nhead=8,
```

```
    num_encoder_layers=6,
    num_decoder_layers=6,
    dim_feedforward=2048,
    dropout=0.1,
    activation=torch.nn.functional.relu,
    layer_norm_eps=1e-05,
)
```

- 인코더/디코더 블록 적층 구조, 각 블록의 **마스크 적용 위치**, Q/K/V 생성–스코어 계산–가중합–선형·정규화의 흐름이 도식으로 제시됨.

체크리스트

- [ ] `TokenEmbedding` 출력에 emb_size\sqrt{\text{emb_size}}emb_size 스케일 적용
- [ ] `src_key_padding_mask` / `tgt_key_padding_mask` / `memory_key_padding_mask` 인자 매칭
- [ ] `generator`가 마지막 로짓(어휘 크기) 산출

용어 정리

- **마스킹**: 미래 토큰 차단(인과성 보장), PAD 등 무의미 위치 제외.
- **인코더–디코더 어텐션**: 타깃 쿼리와 소스 메모리(K/V) 간의 교차 참조.
- **병합(concatenation)**: 멀티 헤드 결과를 임베딩 차원 축으로 이어 붙이는 연산.

모델 실습 — 하이퍼파라미터·순방향·마스크·배치/학습 준비

- **모델 하이퍼파라미터**

  `d_model`(임베딩 차원), `nhead`(멀티헤드 수), `num_encoder_layers` / `num_decoder_layers`(인코더/디코더 층 수), `dim_feedforward`(FFN 은닉 크기), `dropout`, `activation`(FFN 활성화), `layer_norm_eps`(LayerNorm ε)로 트랜스포머를 정의한다.

- **순방향 메서드 인자**

  `forward(src, tgt, src_mask, tgt_mask, memory_mask, src_key_padding_mask, tgt_key_padding_mask, memory_key_padding_mask)`

  → 마스크/패딩마스크를 정확한 **형태**로 넣는 것이 핵심.

- **마스크의 의미와 형태**
  - `src_mask`, `tgt_mask` : `[시퀀스 길이, 시퀀스 길이]`

        - 값 **0** → 정상 어텐션 / **1** → 해당 위치 어텐션 차단.
        - 값 **inf** → 소프트맥스 전 가중치가 0이 되도록 완전 차단.
        - 값 **+inf** → 특정 위치만 보게 할 때 사용(일반적으로 사용하지 않음).
    - `memory_mask` : `[타깃 길이, 소스 길이]`
    - `_key_padding_mask` : `[배치, 시퀀스 길이]`의 **불리언** 마스크(패딩 위치를 True로 가림).
    - **디코더 인과성**은 `tgt_mask`를 **상삼각 마스크**로 만들어 미래 토큰을 가린다.
- **배치·토크나이즈·패딩**

  `sequential_transforms = token_transform → vocab_transform → input_transform`

  `input_transform`은 각 문장에 **<BOS> 앞·<EOS> 뒤**를 부착.

  `pad_sequence`로 배치 내 길이를 맞추며, PAD 토큰은 이후 **padding mask**로 가린다.
- **학습 루프 골자**

  교사강요(teacher forcing): `target_input = target[:-1, :]`, `target_output = target[1:, :]`로 시프트.

  마스크 생성 → 모델 순전파 → `CrossEntropyLoss(ignore_index=PAD_IDX)`로 손실 계산.

코드

1) 순방향 호출 서명

```
output = transformer.forward(
    src, tgt,
    src_mask=None, tgt_mask=None, memory_mask=None,
    src_key_padding_mask=None, tgt_key_padding_mask=None,
    memory_key_padding_mask=None,
)
```

2) 배치·전처리 뼈대

```
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

def sequential_transforms(*transforms):
    def func(x):
```

```
        for t in transforms: x = t(x)
        return x
    return func

def input_transform(token_ids):
    return torch.cat([torch.tensor([BOS_IDX]), torch.tensor(token_ids), torch.t
ensor([EOS_IDX])])

def collator(batch):
    src_batch, tgt_batch = [], []
    for src_sample, tgt_sample in batch:
        src_batch.append(text_transform[SRC_LANG](src_sample.rstrip("\n")))
        tgt_batch.append(text_transform[TGT_LANG](tgt_sample.rstrip("\n")))
    src_batch = pad_sequence(src_batch, padding_value=PAD_IDX)
    tgt_batch = pad_sequence(tgt_batch, padding_value=PAD_IDX)
    return src_batch, tgt_batch
```

3) 어텐션 마스크 생성

```
def generate_square_subsequent_mask(s):
    mask = torch.triu(torch.ones((s, s), device=DEVICE) == 1).T  # 상삼각
    mask = (mask.float()
            .masked_fill(mask == 0, float('-inf'))  # 미래 가리기
            .masked_fill(mask == 1, 0.0))
    return mask

def create_mask(src, tgt):
    src_len, tgt_len = src.shape[0], tgt.shape[0]
    tgt_mask = generate_square_subsequent_mask(tgt_len)
    src_mask = torch.zeros((src_len, src_len), device=DEVICE).type(torch.bo
ol)
    src_padding_mask = (src == PAD_IDX).T     # [배치, 길이]
    tgt_padding_mask = (tgt == PAD_IDX).T
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask

# 교사강요용 시프트
target_input = target_tensor[:-1, :]
```

```
target_output = target_tensor[1:, :]
```

4) 모델 구성·요약 출력

```
from torch import optim

model = Seq2SeqTransformer(
    num_encoder_layers=3, num_decoder_layers=3,
    emb_size=512, max_len=512, nhead=8,
    src_vocab_size=len(vocab_transform[SRC_LANGUAGE]),
    tgt_vocab_size=len(vocab_transform[TGT_LANGUAGE]),
    dim_feedforward=512,  # 예제 설정
).to(DEVICE)

criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX).to(DEVICE)
optimizer = optim.Adam(model.parameters())
```

5) 학습/평가 루프 골자

```
def run(model, optimizer, criterion, split):
    model.train() if split == "train" else model.eval()
    data_iter = Multi30k(split=split, language_pair=(SRC_LANGUAGE, TGT_L
ANGUAGE))
    dataloader = DataLoader(data_iter, batch_size=BATCH_SIZE, collate_fn=
collator)

    for source_batch, target_batch in dataloader:
        source_batch, target_batch = source_batch.to(DEVICE), target_batch.t
o(DEVICE)
        target_input  = target_batch[:-1, :]
        target_output = target_batch[1:, :]

        src_mask, tgt_mask, src_pad_mask, tgt_pad_mask = create_mask(sour
ce_batch, target_input)
```

```
        logits = model(
            src=source_batch, trg=target_input,
            src_mask=src_mask, tgt_mask=tgt_mask, memory_mask=None,
            src_padding_mask=src_pad_mask, tgt_padding_mask=tgt_pad_mas
k,
            memory_key_padding_mask=src_pad_mask,
        )
        # 이후: logits → 손실 계산/역전파(or 평가)
```

용어 · 하이퍼파라미터 설명

- `d_model`: 입력/출력 임베딩 차원(모든 블록의 통일 차원).
- `nhead`: 멀티헤드 수(병렬 주의집중 경로).
- `num_encoder_layers` / `num_decoder_layers`: 인코더/디코더 적층 개수.
- `dim_feedforward`: 각 블록 FFN의 은닉 크기.
- `dropout` / `activation`: FFN·임베딩 등에 적용되는 드롭아웃 비율과 활성화 함수.
- `layer_norm_eps`: LayerNorm 수치 안정화용 ε.
- `src_mask`: 소스 셀프어텐션용(보통 전부 허용: 0/False).
- `tgt_mask`: 디코더 인과성 마스크(상삼각, 미래 차단).
- `memory_mask`: 인코더-디코더 어텐션에서 타깃↔소스 위치 제한.
- `_key_padding_mask`: PAD 위치(True)를 가려 어텐션/연산에서 제외.
- **마스크 값의 의미**
  - `0`: 정상 계산, `1`: 차단, `inf`: 확실한 차단(softmax 이전 가중치 0 유도), `+inf`: 특정 위치만 보게 할 때(특수 상황).

트랜스포머 디코더·마스킹·PyTorch 구현·GPT 시리즈

- **어텐션 스코어**: 쿼리/키 내적을 차원 d의 제곱근으로 스케일해 소프트맥스 적용.

$$\mathrm{Attention}(Q,K,V)=\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

- **멀티헤드**: k개의 헤드가 병렬로 셀프-어텐션 수행 → [N,S,d/k]들을 concat하여 [N,S,d] 출력. 드롭아웃·정규화 포함.

- **디코더 인과성(Causality)**: 현재 토큰이 **미래 토큰을 보지 못하도록** `마스크 멀티헤드 어텐션` 사용.
  - **tgt_mask**: 상삼각(후속 위치) 가리기(−inf), 현재/이전만 참조.
  - **padding masks**: PAD 토큰 위치를 무시.
- **입출력/마스크 인자**(PyTorch `nn.Transformer`): `src, tgt, src_mask, tgt_mask, src_key_padding_mask, tgt_key_padding_mask, memory_key_padding_mask`.
- **데이터셋/전처리**: **Multi30k(EN–DE)** + **torchtext/torchdata**. spaCy 토크나이저로 토큰화 → `build_vocab_from_iterator`로 **vocab** 생성. 특수토큰 **UNK/PAD/BOS/EOS** 지정, PAD는 손실에서 `ignore_index`.
- **포지셔널 인코딩**: 사인/코사인 기반 위치 인코딩을 임베딩에 더함.
- **모델 구성**: `Seq2SeqTransformer`—토큰 임베딩(src/tgt) + 포지셔널 인코딩 + `nn.Transformer`(인코더·디코더 층 구성) + 최종 Linear(generator).
- **학습 루프**: 미니배치 → BOS/EOS 추가·패딩 → 마스크 생성 → CrossEntropyLoss(IGNORE=PAD) → 5 epoch 예시로 Train/Val loss 하강.
- **추론(Greedy decoding)**: 인코더 메모리 고정, 디코더를 step-by-step으로 호출하며 **가장 확률 높은 토큰**을 이어 붙임(EOS까지).
- **GPT 개요**: 트랜스포머 **디코더 스택만** 사용한 단방향 LM.
  - **GPT-1**: 12층, BookCorpus(≈4.5GB), 최대 512 토큰.
  - **GPT-2**: 48층, 더 큰 말뭉치(≈40GB Web), 최대 1024 토큰, 제로샷 가능.
  - **GPT-3**: 96층, 파라미터 대폭 증가, 최대 2048 토큰, 웹/위키/서적 대규모(≈45TB 수집)로 사전학습, **프롬프트** 방식 활용.
  - **ChatGPT/3.5**: GPT-3 계열 + **RLHF**로 대화 품질 향상. GPT-4는 멀티모달 확장.

수식 · 코드

- **스케일드 점수**:$\alpha = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)$

  가중 합: $\mathrm{Attn}(Q,K,V) = \alpha V$
- **포지셔널 인코딩**

  $\mathrm{PE}_{(pos,2i)} = \sin\left(pos/10000^{2i/d}\right),\ \mathrm{PE}_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d}\right)$
- **디코더 마스크 생성:** 상삼각(미래) 영역을 **−inf**로 채워 softmax에서 확률 0이 되게 함.
  - Subsequent mask: $M_{ij} = -\infty\ if j > i, else 0$→ 미래 차단.

    - Key padding mask: PAD 위치(True) → 해당 위치 가중치 0 처리.
        - `src_mask` : `(S,S)` 전부 0(제한 없음)
        - `tgt_mask` : `(T,T)` 상삼각 `inf` (미래 가림, causal)
        - `_padding_mask` : `(N,S or T)` 에서 패딩 위치 True
- **손실**: $\mathcal{L} = \mathrm{CrossEntropy}\Big(\mathrm{reshape}(\mathrm{logits}, -1, V),\ \mathrm{reshape}(\mathrm{target}, -1)\Big)$

  (V: 타깃 어휘 크기)
    - `CrossEntropyLoss(ignore_index=PAD_IDX)`
    - `loss = criterion(logits.reshape(-1, V), target_output.reshape(-1))`
- **타깃 시프트**:
    - `target_input = target[:, :-1]` (BOS부터 마지막-1)
    - `target_output = target[:, 1:]` (첫-1부터 EOS)
- **그리디 디코딩 절차**:
    1. `memory = encode(src, src_mask)`
    2. `ys = [BOS]` 부터 시작
    3. 매 스텝 `tgt_mask` 갱신 → `decode(memory, ys)` → `generator` → `argmax` 토큰 추가
    4. `EOS` 면 중단, 아니면 반복
- **Greedy decode 루프**

```
memory = model.encode(src, src_mask)
ys = [BOS]
for t in range(max_len-1):
    tgt_mask = causal_mask(len(ys))
    out = model.decode(ys, memory, tgt_mask)
    prob = model.generator(out[-1])
    next_id = prob.argmax()
    ys.append(next_id)
    if next_id == EOS: break
```

- **학습 루프 스텝**

1. `optimizer.zero_grad()`
2. `loss.backward()`
3. `optimizer.step()`

용어 · 하이퍼파라미터

- **모델/학습**: `d_model`(임베딩 차원), `nhead`(헤드 수), `num_encoder_layers`, `num_decoder_layers`, `dim_feedforward`, `dropout`, `activation`, `layer_norm_eps`.
- **특수 토큰**: `UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX`
- **마스크 종류**: `src_mask`, `tgt_mask`(인과성), `memory_mask`, `_key_padding_mask`
- **텐서 형상**:
  - 임베딩 입력/출력 [S, N, d] (PyTorch `nn.Transformer` 기본)
  - 멀티헤드 내부 [N, S, d/k]× k → concat [N, S, d]
- **Greedy vs Beam**: Greedy는 매 스텝 argmax 하나 선택(빠름, 탐색 얕음). Beam은 상위 k 후보 트래킹(품질↑, 비용↑).
- **BOS / EOS / PAD / UNK**: 문장 시작/종료/패딩/미등록 토큰. `ignore_index=PAD`로 손실 집계에서 제외.
- **logits**: `[T, N, V]` 혹은 `[N, T, V]` 모양의 비정규화 점수. 학습 시 `reshape`로 `[N*T, V]`.
- **src_mask**: 셀프어텐션에서 **참조 허용 범위**(예시: 전부 허용).
- **tgt_mask**: **미래 정보 차단**(causal mask).
- **(src|tgt)_padding_mask**: PAD 위치를 True로 설정해 **어텐션 제외**.
- **memory_key_padding_mask**: 인코더 메모리의 PAD 차단.
- **num_tokens / max_len**: 소스 토큰 수 / 디코딩 상한. 보통 `max_len ≥ num_tokens`.
- **RLHF**: 사람 피드백으로 보상 모델을 학습해 **지시 따르기** 성능을 개선.
- **GPT-1/2/3 차이 핵심**
  - 아키텍처: **디코더 전용** 공통.
  - **규모·컨텍스트·데이터**가 세대가 갈수록 크게 증가(예: 512→1024→2048 토큰).
  - 사용 방식: GPT-3는 **프롬프트**로 범용 작업 수행.
- **GPT-2 분류 세팅(CoLA)**
  - 토크나이저: `AutoTokenizer.from_pretrained("gpt2")`

- 패딩 대체: GPT-2는 패딩 토큰 없음 → `pad_token_id = eos_token_id`로 지정
- 모델: `GPT2ForSequenceClassification(..., num_labels=2)`

순서

1. **환경 설치**
   - `pip install torchdata torchtext portalocker`
   - spaCy 모델: `python -m spacy download de_core_news_sm en_core_web_sm`
2. **토큰화 & Vocab**
   - `get_tokenizer("spacy", language="de_core_news_sm" / "en_core_web_sm")`
   - `build_vocab_from_iterator(generate_tokens(...), specials=[UNK,PAD,BOS,EOS])`
   - `vocab.set_default_index(UNK_IDX)`
3. **텍스트 변환 파이프라인**
   - (토큰화 → vocab 숫자화 → `[BOS] + ids + [EOS]` 추가)
   - 배치 `collate_fn` : `pad_sequence(..., padding_value=PAD_IDX)`로 src/tgt 각각 패딩.
4. **마스크 만들기**
   - `generate_square_subsequent_mask(L)`로 **tgt_mask**(상삼각 −inf/0)
   - `key_padding_mask = (batch == PAD_IDX).transpose(0,1)`
5. **모델 구성**
   - `Seq2SeqTransformer(d_model=512, nhead=8, num_encoder_layers=3, num_decoder_layers=3, dim_feedforward=2048, dropout=0.1)`
   - 임베딩(√d 스케일), 포지셔널 인코딩, 최종 `nn.Linear` to vocab size.
6. **학습 루프**
   - 입력: `src=[S,B]` , `tgt=[T,B]` → `tgt_input=tgt[:-1]` , `tgt_out=tgt[1:]`
   - 마스크: `src_mask=zeros` , `tgt_mask=subsequent` , padding masks 생성
   - 전향 → `logits.shape=[T-1,B,V]` → CE Loss(IGNORE=PAD) → 옵티마이저 스텝
   - Train/Val loss 모니터링(예시 5 epoch)
7. **추론(Greedy)**
   - 인코더로 `memory` 계산 → `ys=[BOS]`에서 시작, 한 토큰씩 디코더 호출
   - 가장 높은 확률 토큰 추가, `EOS`에서 중단 → vocab으로 detokenize

8. **디버깅 포인트**
    - 마스크/배치 축: `src=[S,B]`, `key_padding_mask=[B,S]` 일치 확인
    - `tgt_mask` 크기 `T×T`, dtype(float or bool) 규칙에 맞추기
    - 손실에서 PAD 무시(`ignore_index`) 꼭 설정
    - 마스크 값: **−inf vs 0** 혼용 주의(softmax 입력에는 −inf).
    - **OOV**가 잦으면: 어휘 확장, 서브워드 토크나이저, 데이터 증대/도메인 전이 고려.
    - **max_len**이 짧아 **EOS 전 강제 종료**되지 않게 여유분 확보.

모델 실습 — 모델 훈련 평가

- **훈련 루프**: `train()`으로 배치별 **outputs.loss**(HF 모델이 `labels` 주면 자동 계산)를 누적 →역전파→옵티마이저 스텝.
- **검증 루프**: `evaluation()`에서 `torch.no_grad()`로 **CrossEntropyLoss**와 **정확도**(argmax 비교) 계산.
- **모델 선택**: 에폭마다 **val_loss**가 **best_loss**보다 작으면 `state_dict`를 저장(간이 early-stopping/체크포인트).
- **최종 평가**: 저장한 가중치 로드 후 **test 데이터**로 손실·정확도 산출.
- **결과**: `Test Loss = 0.5712`, `Test Accuracy = 0.7431` → 데이터가 작아도 기본 성능 확보, 추가 개선 여지 있음.

코드 요약

- **Train**
    - `outputs = model(input_ids, attention_mask, labels)`
    - `loss = outputs.loss → loss.backward() → optimizer.step()`
- **Eval**
    - `criterion = nn.CrossEntropyLoss()`
    - `logits = outputs.logits`
    - `val_loss += criterion(logits, labels)`
    - `accuracy = mean(argmax(logits) == labels)`
- **체크포인트**

- `if val_loss < best_loss: torch.save(model.state_dict(), path)`
- `model.load_state_dict(torch.load(path))` 후 test 평가

순서

1. 모델/토크나이저 준비, `pad_token_id = eos_token_id` 설정.
2. `DataLoader` OK? 텍스트→ `input_ids/attention_mask, labels` 변환 확인.
3. `optimizer`와 `epochs` 설정, `train()` 루프에서 **outputs.loss**로 학습.
4. 매 에폭 **검증** 실행, `best_loss` 기준으로 `state_dict` 저장.
5. 최종적으로 **최고 가중치 로드** → **test** 세트 평가(손실·정확도 출력).
6. 로그 기록: `Train/Val loss·Val acc` 프린트로 과적합 징후 점검.
7. 재현성: 시드 고정, `max_len`·배치 크기·러닝레이트 기록.
8. 개선 아이디어: 더 큰 데이터/정규화(드롭아웃 증대), 러닝레이트 스케줄러, 층 동결 후 미세조정, 가중치 감쇠.

용어·하이퍼파라미터

- **outputs.loss**: HF 모델이 `labels` 받으면 내부 CE 손실 반환.
- **logits**: 소프트맥스 전 점수(클래스별). 정확도 계산은 `argmax`.
- **best_loss**: 현재까지 최소 **검증 손실**.
- **pad/eos 토큰**: GPT-2는 패딩 토큰이 없어 **eos**를 `pad_token_id`로 대체해 마스킹.
- **모드 전환**: `model.train()` ↔ `model.eval()` 필수.
- **no_grad**: 평가 시 그래디언트 비활성화로 속도·메모리 절감.