Week12_예습과제_팽소원

트랜스포머

트랜스포머 : 2017년 “Attention is All You Need” 논문을 통해 소개된 신경망 아키텍쳐

- 기존의 순환 신경망과 같은 순차적 방식이 아닌 병렬로 입력 시퀀스를 처리
- 순한신경망보다 훨씬 빠르고 효율적으로 처리
- 순차 처리나 반복 연결 의존 X, 셀프 어텐션 기반
- 재귀나 합성곱 연산 없이 입력 토큰 간의 관계를 직접 모델링 가능
- 대용량 데이터세트에서 매우 효율적
- 언어 모델링 및 텍스트 분류와 같은 작업에서 매우 효과적
- 광범위한 자연어 처리 작업에서 높은 효율
- 오토 인코딩, 자기 회귀 방식 또는 두개의 조합으로 학습

오토 인코딩 방식 : 랜덤하게 문장의 일부를 빈칸 토큰으로 만들고 해당 빈칸에 어떤 단어가 적절한지 예측하는 작업 수행

자기 회귀 방식 : 이전 단어들이 주어졌을 때 다음 단어가 무엇인지 맞히는 작업 수행

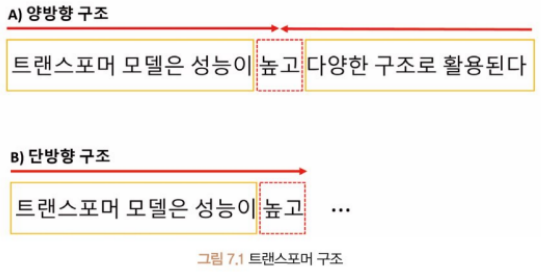

그림 7.1 트랜스포머 구조

양방향 구조 : 인코더

단방향 구조 : 디코더

양방향성 구조의 인코더 (오토 인코딩)

단방향성 디코더(자기 회귀)

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

Transformer : 딥러닝 모델 중 하나로 어텐션 메커니즘만을 사용하여 시퀀스 임베딩 표현

트랜스포머의 어텐션 메커니즘

- 인코더와 디코더 간의 상호작용으로 입력 시퀀스의 중요한 부분에 초점을 맞추어 문맥을 이해하고 적절한 출력 생성
- 인코더 : 입력 시퀀스를 임베딩하여 고차원 벡터로 변환
- 디코더 : 인코더의 출력을 입력으로 받아 출력 시퀀스 생성
- 인코더와 디코더 사이의 상관관계를 계산하여 중요한 정보에 집중

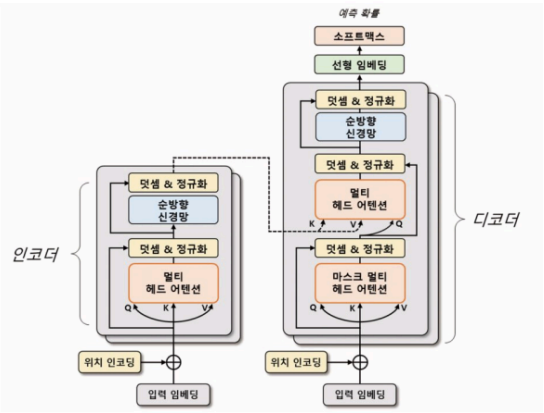

그림 7.2 트랜스포머 모델 구조(Q: 쿼리, K: 키, V: 값)

인코더와 디코더는 N개의 **트랜스포머 블록**으로 구성

트랜스포머 블록은 **멀티헤드 어텐션**과 **순방향 신경망**으로 구성

멀티헤드 어텐션

: 입력 시퀀스에서 쿼리, 키, 값 벡터 정의해 입력 시퀀스들의 관계를 셀프 어텐션하는 벡터 표현 방법

: 쿼리와 각 키의 유사도 계산 → 해당 유사도를 가중치로 사용 → 값 벡터 합산

어텐션 행렬은 입력 시퀀스 각 단어의 임베딩 벡터를 대체

순방향 신경망

: 임베딩 벡터를 더 고도화하기 위해 사용

: 여러 개의 선형 계층으로 구성

: 입력 벡터에 가중치를 곱하고, 편향을 더하고, 활성화 함수 적용

:학습된 가중치들은 입력 시퀀스의 각 단어의미를 잘 파악할 수 있는 방식으로 갱신

입력 시퀀스 데이터를 **소스**와 **타깃**으로 나눌 수 있음

ex) 영어를 한글로 번역하는 경우, 한글(생성하는 언어) = 타깃, 영어(참조하는 언어) =소스

인코더 : 소스 데이터를 **위치 인코딩**된 입력 임베딩으로 표현

디코더 : **마스크 멀티 헤드 어텐션**을 사용해 타깃 데이터를 순차적으로 생성

입력 임베딩과 위치 인코딩

트랜스포머 모델은 입력 시퀀스를 병렬 구조로 처리하기 때문에 단어의 순서 정보 제공 X

→ 위치 정보를 임베딩 벡터에 추가하기 위해 위치 인코딩 방식 사용

위치 인코딩

: 입력 시퀀스의 순서 정보를 모델에 전달하는 방법으로 각 단어의 위치 정보를 나타내는 벡터를 더해 임베딩 벡터에 위치 정보를 반영

: sin 함수와 cos 함수 사용하여 임베딩 벡터와 위치 정보 결합

: 각 토큰의 위치를 각도로 표현해 sin 함수와 cos 함수로 위치 인코딩 벡터 계산

```
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)
```

PositionEncoding 클래스는 입력 임베딩 차원(d_model)과 최대 시퀀스 (max_len)을 입력받음

입력 시퀀스의 위치마다 sin함수와 cos 함수로 위치 인코딩 계산

수식 7.1 위치 인코딩

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$$
$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}})$$

pos : 입력 시퀀스에서 해당 단어의 위치를 나타냄

i : 임베딩 벡터의 차원 인덱스, 짝수라면 첫 번째 sin 수식 사용, 홀수라면 두번째 cos 수식 사용

특수 토큰

트랜스포머는 단어 토큰 이외의 특수 토큰을 활용하여 문장을 표현

특수 토큰: 입력 시퀀스의 시작과 끝을 나타내거나 마스킹 영역으로 사용

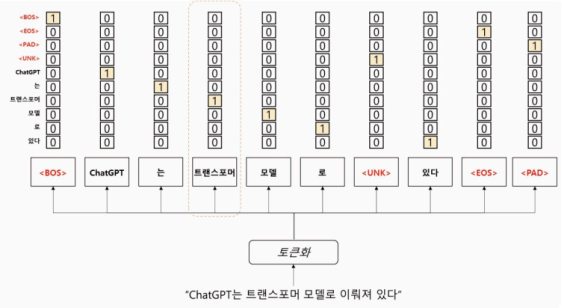

그림 7.3 특수 토큰을 활용한 문장 토큰 배열

BOS, EOS, UNK, PAD 토큰은 모두 특수 토큰임

BOS : 문장의 시작

EOS : 문장의 종료

UNK : 어휘 사전에 없는 단어

PAD : 모든 문장을 일정한 길이로 맞추기 위해 사용

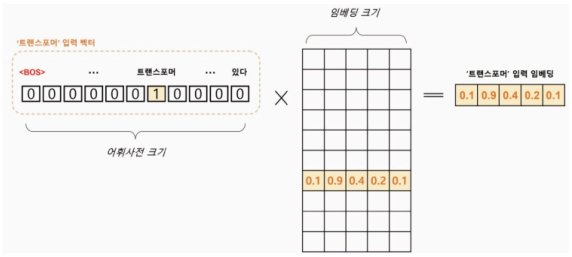

그림 7.4 입력 임베딩 산출 과정

문장 토큰 배열을 어휘 사전에 등장하는 위치에 원-핫 인코딩으로 표현

입력 임베딩으로 변환되는 과정은 Word2Vec 방법과 동일

트랜스포머 인코더

트랜스포머 인코더: 입력 시퀀스를 받아 여러 개의 계층으로 구성된 인코더 계층을 거쳐 연산 수행

각 인코더 계층은 멀티 헤드 언텐션과 순방향 신경망으로 구성되며, 입력 데이터에 대한 정보 추출후 다음 계층으로 전달

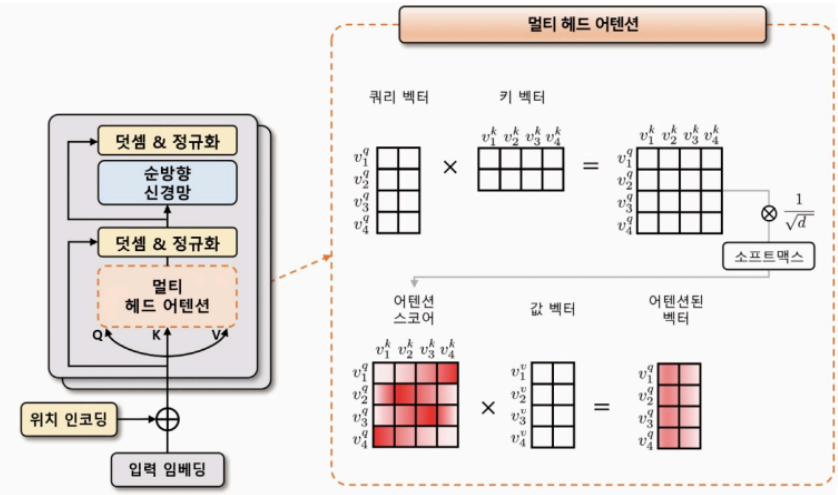

그림 7.5 트랜스포머 인코더 블록 연산 과정

입력 임베딩은 선형 변환을 통해 3개의 임베딩 벡터를 생성하고 각각의 베터를 쿼리, 키, 값 벡터라고 정의

쿼리 벡터 : 현재 시점에서 참고하고자 하는 정보의 위치를 나타내는 벡터로, 인코더의 각 시점마다 생성

키 벡터 : 쿼리 벡터와 비교되는 대상으로 쿼리 벡터를 제외한 입력 시퀀스에서 탐색되는 벡터

값 벡터 : 쿼리 벡터와 키 벡터로 생성된 어텐션 스코어를 얼마나 반영할지 설정하는 가중치 역할

수식 7.2 어텐션 스코어 계산식

$$score(v^q, v^k) = \text{softmax}\left(\frac{(v^q)^T \cdot v^k}{\sqrt{d}}\right)$$

쿼리, 키 값 벡터를 내적해 어텐션 스코어를 구하고, 이 스코어 값에 √d 값 만큼 나누어 보정한다

벡터 차원이 커질 때 스코어값이 같이 커지는 문제를 완화하기 위해 보정한다.

보정된 어텐션 스코어를 소프트맥스 함수를 이용하여 확률적으로 재표현하고, 값 벡터와 내적하여 셀프 어텐션 된 벡터 생성

이 과정을 반복해 셀프 어텐션된 스코어 맵 생성

멀티 헤드 : 셀프 어텐션을 여러 번 수행해 여러 개의 헤드 생성

각각의 헤드가 독립적으로 어텐션 수행 후 결과를 합침

입력 [N, S, d] 텐서에서 k개의 셀프 어텐션 벡터를 생성하면 [N. k. S. d/k] 텐서 형태로 구성

k개의 셀프 어텐션 벡터는 임베딩 차원 ㅈ=축으로 다시 병합 : [N, S, d]

트랜스 포머 인코더는 여러 개의 트랜스포머 인코더 블록으로 구성

이전 블록에서 출력된 벡터는 다음 블록의 입력으로 전달

마지막 인코더 블록에서 출력된 벡터는 디코더에서 사용

트랜스포머 디코더

트랜스포머 디코더 : 위치 인코딩이 적용된 타깃 데이터의 입력 임베딩을 입력받음

인코더의 멀티 헤드 어텐션 모듈 —인과성 반영→ 디코더의 마스크 멀티 헤드 어텐션 모듈

마스크 멀티 헤드 어텐션

- 스코어 맵을 계산할 때 첫 번째 쿼리 벡터가 첫번째 키 벡터만을 바라볼 수 있게 마스크를 씌우고, 두번째 쿼리는 첫번째와 두번째 키 벡터를 바라보게 마스크를 씌움
- 인과성을 보장하면서 셀프 어텐션 수행
- 타깃 데이터가 쿼리 벡터로 사용, 인코더의 소스 데이터가 키와 값 벡터로 사용
- 셀프 어텐션을 방지하기 위해 마스킹 적용
- 마지막 디코더 블록의 출력 텐서에 선형 변환 및 소프트맥스 함수를 적용해 각 타깃 시퀀스 위치마다 예측 확률 계산

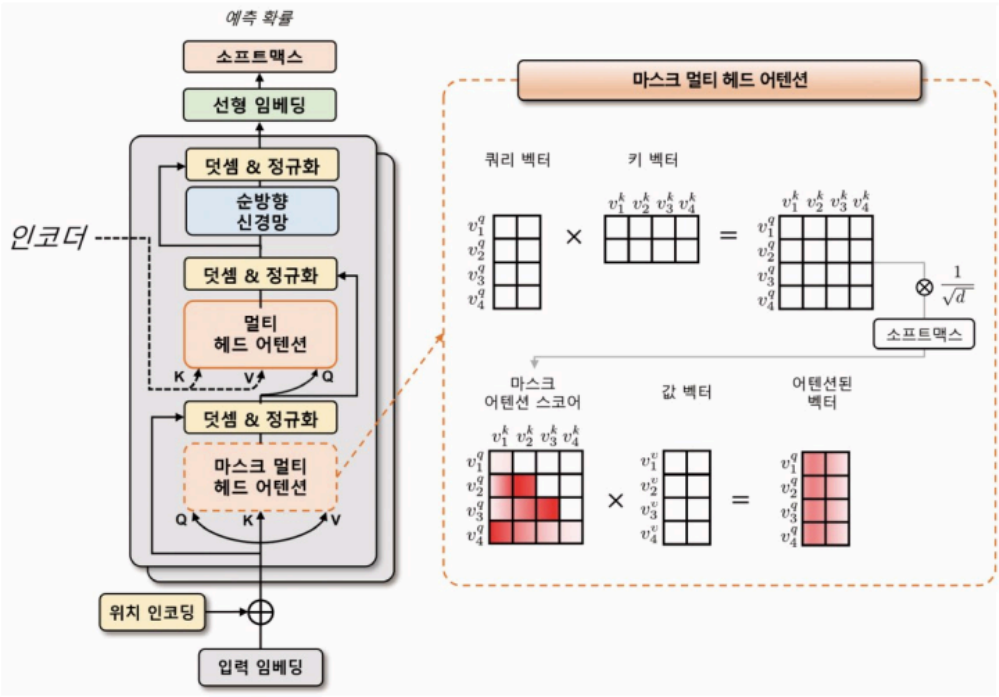

그림 7.6 트랜스포머 디코더 블록 연산 과정

모델 실습

Multi30k : 영어-독일어 병렬 말뭉치로 약 30000개의 데이터 제공

portalocker : 파이썬에서 파일 락을 관리하기 위한 라이브러리

```
# 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.functionsal.relu,
    layer_norm_eps=1e-05,
)
```

임베딩 차원(d_model) : 트랜스포머 모델의 입력과 출력 차원의 크기 정의

헤드(nhead) : 멀티 헤드 어텐션의 헤드의 개수를 정의

헤드의 개수가 많을수록 모델의 병렬 처리 능력 증가 (but 매개변수의 수도 증가)

인코더 계층 개수(num_encoder_layers)와 **디코더 계층 개수(num_decoder_layers)** : 인코더와 디코더의 계층 수

모델의 복잡도와 성능에 영향을 준다 계층 개수가 많을수록 더 복잡한 문제를 해결할 수 있지만, 너무 많을 경우 과적합

순방향 신경망 크기(dim_feedforward) : 순방향 신경망의 은닉층 크기 정의

모델의 복잡도와 성능에 영향을 미침

드롭아웃(dropout) : 인코더와 디코더 계층에 적용되는 드롭아웃 비율 적용

활성화 함수(activation) : 순방향 신경망에 적용되는 활성화 함수 의미

계층 정규화 입실론(layer_norm_eps) : 계층 정규화를 수행할 때 분모에 더해지는 입실론 값 정의

```
# 트랜스포머 순방향 메서드
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,
)
```

소스(src)와 **타깃(tgt)** : 인코더와 디코더에 대한 시퀀스로 [소스(타깃) 시퀀스 길이, 배치 크기, 임베딩 차원] 형태의 데이터를 입력받는다

소스 마스크(src_mask)와 **타깃 마스크(tgt_mask)** : 소스와 타깃 시퀀스의 마스크로 [소스(타깃) 시퀀스 길이, 시퀀스 길이] 형태의 데이터를 입력 받는다

if mask 값 = 0: 모든 입력 단어가 동일한 가중치를 갖고 어텐션 수행

if 1 : 모든 입력 단어의 가중치가 0으로 설정돼 어텐션 연산 수행X

if -inf : 어텐션 연산 결과에 0으로 가중치가 보여돼 마스킹된 위치의 정보를 모델이 무시하게 만듦

if +inf : 모든 입력 단어에 무한대의 가중치가 부여돼 어텐션 연산 경과가 해당 위치에 대한 정보만으로 구성(일반적으로 사용X)

메모리 마스크(memory_mask) : 인코더 출력의 마스크로 [타깃 시퀀스 길이, 소스 시퀀스 길이] 형태로, 메모리 마스크의 값이 0인 위치에서는 어텐션 연산 수행X

소스, 타깃, 메모리 키 패딩 마스크(key_padding_mask) : 소스, 타깃, 메모리 시퀀스에 대한 패딩 마스크를 의미, [배치 크기, 소스(타깃) 시퀀스 길이] 형태의 데이터를 입력받음

GPT

GPT(Generative Pre-trained Transformer) : 2018년 OpenAI에서 발표한 트랜스포머 기반 언어 모델

- 트랜스포머의 디코더를 여러 층 쌓아 만든 언어 모델
- 트랜스포머의 인코더는 입력 문장의 특징을 추출하는데 초점, 디코더는 입력 문장과 출력 문장 간의 관계를 이해하고 출력 문장을 생성하는데 초점

⇒ 디코더를 쌓아 모델을 구성하면 자연어 생성과 같은 언어 모델링 작업에서 더 높은 성능 발휘

- GPT는 일반적으로 자연어 생성, 기계 번역, 질의응답, 요약 등 다양한 자연어 처리 작업에 활용
- 예측 성능이 뛰어나고 적은 데이터로도 높은 성능
- GPT-1은 약 1억 2천만 개의 매개변수, GPT-2는 약 15억 개의 매개변수 존개, GPT-3은 약 1750억 개의 매개변수
- 많은 양의 매개변수를 줄이기 위해 RLHF 방법 도입

RLHF(Reinforcement Learning from Human Feedback) : 인간의 피드백을 이용한 강화 학습 방법

- RLHF 방법을 사용해 GPT-3.5는 약 60억 개의 가중치로도 뛰어난 성능

GPT-1

GPT-1 : 트랜스포머 구조를 바탕으로 한 **단방향** 언어 모델

단방향 언어 모델 : 텍스트를 생성할 때 현재 위치에서이전 단어들에 대한 정보만을 참고해 다음 단어를 예측(학습 속도가 빠르고, 모델의 크기가 작아질 수 있다는 장점)

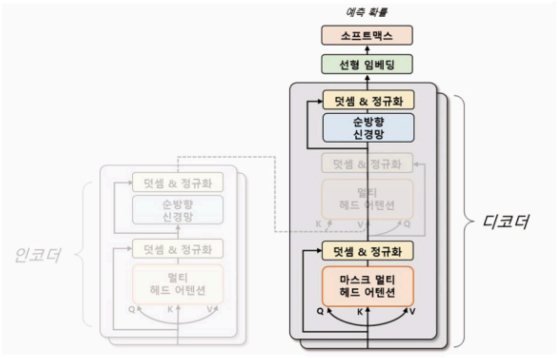

그림 7.7 트랜스포머와 GPT-1 모델 비교

- 이전 단어들에 대한 정보로 다음 단어를 예측할 때 트랜스포머 구조 활용
- 트랜스포머 구조 중 디코더만 사용, 인코더-디코더 간 어텐션 연산 제외
- 디코더 부분만 사용하는 방식으로 모델의 매개변수 수 ↓, 성능 ↑
- 12개의 헤드를 가진 트랜스포머의 디코더 12층을 쌓아 모델 구성
- 디코더의 두번재 다중헤드 어텐션 사용 X
- 1억 2천만 개의 매개변수로 학습

- 약 4.5GB의 BookCorpus 데이터세트를 사용해 언어 모델 사전 학습
- 입력 문장의 임의의 위치까지 보여주고, 뒤에 이어지는 문장을 모델이 자동 예측
- 비지도 학습 (컴퓨팅 자원만 충분하다면 제약 없이 학습 가능)

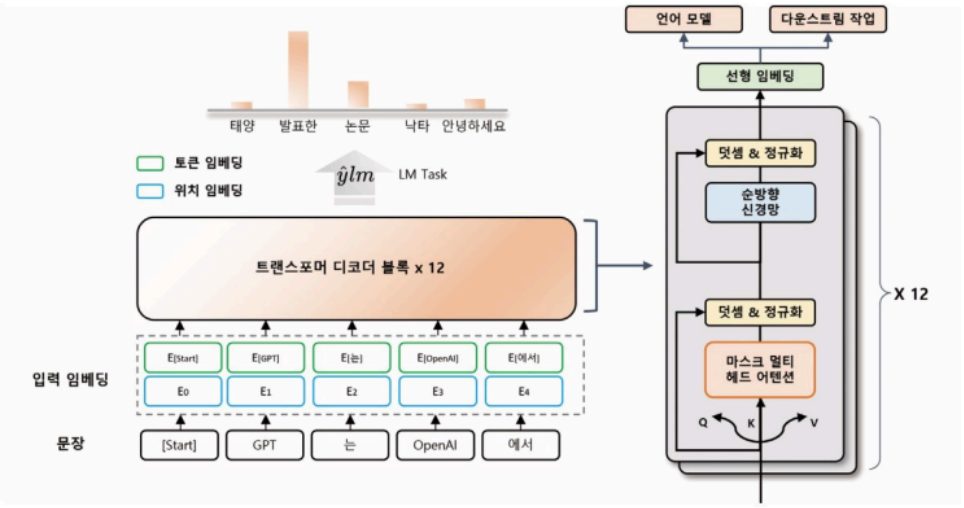

그림 7.8 GPT-1의 사전 학습 작업 구조

- 입력 임베딩은 토큰 임베딩과 위치 임베딩을 이용해 계산
- 토큰 임베딩은 각 토큰을 고정된 차원의 벡터로 매핑, 위치 임베딩은 입력 문장에서 각 토큰의 위치 정보를 나타내는 벡터를 매핑
- 각 작업에 따라 입출력 구조를 바꾸지 않고 언어 모델을 **보조 학습**해 사용

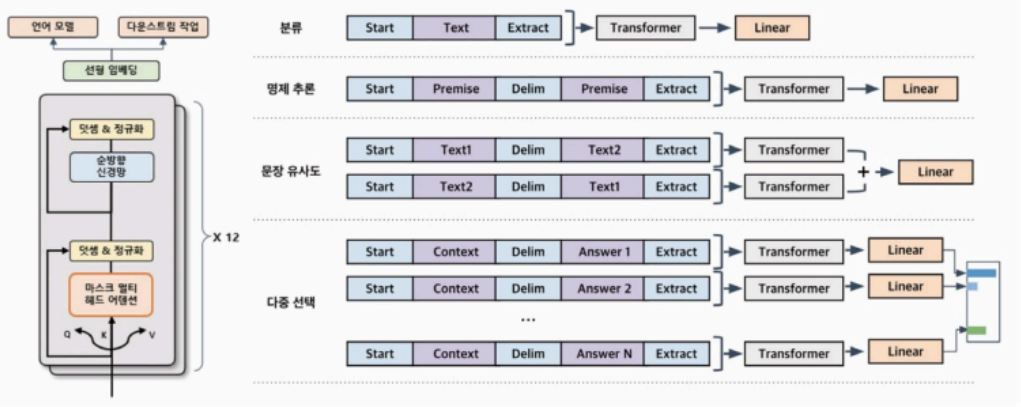

그림 7.9 GPT-1의 다운스트림 작업 구조

다운스트림 작업에서는 추가적으로 손실 함수가 필요하다

두개의 손실 함수를 사용해 최적화하면 보조 학습을 통해 언어 모델 개선+다운스트림 작업 목표 달성

다운스트림 작업을 위한 특수 토큰은 문장의 시작을 의미하는 <start>, 두 문장의 경계를 의미하는 <delim>, 문장의 마지막을 의미하는 <extract> 토큰 용

GPT-2

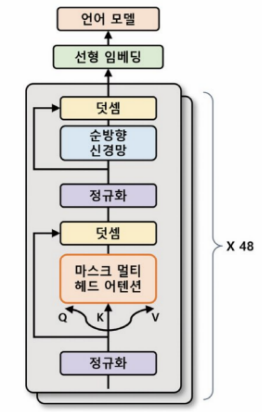

그림 7.10 GPT-2의 모델 구조

- GPT-2에서는 1024의 길이까지 입력을 처리 가능
- 12개의 디코더 층을 사용한 GPT-1 과 달리, 48개의 디코더 계층 사용 → 더 복잡한 패턴 학습
- GPT-1에서는 1200만개의 매개변수 사용, GPT-2에서는 15억개의 매개변수
- GPT-2는 GPT-1보다 훨씬 많은 양의 데이터를 이용해 사전학습함 (미국 웹사이트에서 스크래핑한 약8백만 개의 문서를 사용한 40GB 데이터)
- 제로 샷 학습 가능

GPT-3

GPT-3 : GPT-2와 거의 동일한 모델 구조지만, 몇몇 모델 매개변수 변화를 통해 GPT-2보다 약 116배 많은 매개변수를 가진 모델

- 96개의 헤드를 가진 멀티 헤드 어텐션 사용
- 트랜스포머 디코더 층도 96개의 층
- 희소 어텐션과 일반 어텐션을 섞어서 사용

- 토큰 임베딩 크기도 1600에서 12888로 대폭 증가
- GPT-2의 2배인 최대 2048개 토큰까지 입력 가능
- 웹 크롤링, 위키백과, 서적 등에서 수집한 약 45TB에 달하는 방대한 양의 데이터세트를 이용하여 모델 학습
- GPT-2와 같이 프롬프트 형식으로 작업 수행

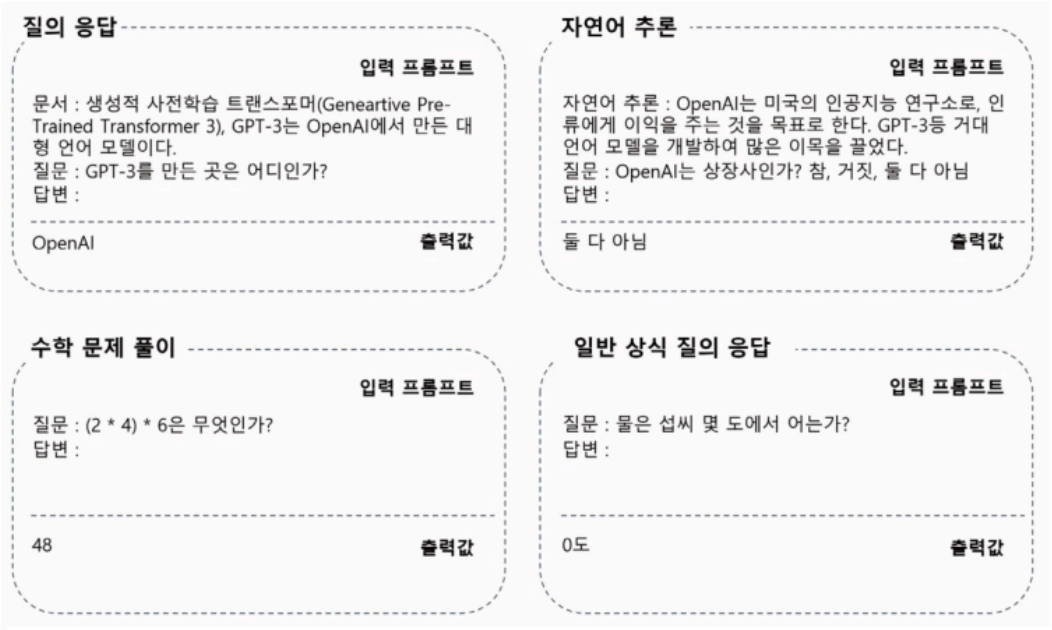

그림 7.11 GPT-3의 다운스트림 작업 프롬프트 예시

- 새로운 도메인에서의 작업에 대해서도 높은 일반화 능력
- BUT 너무 큰 모델이기 때문에 매우 느리고 비용이 많이 듦

GPT-3.5

GPT-3.5(InstructGPT) : GPT 3 의 모델 구조를 그대로 따르면서, 새로운 학습 방법인 RLHF를 도입해 모델의 매개변수 수를 줄이고 모델의 자연스러움을 높임

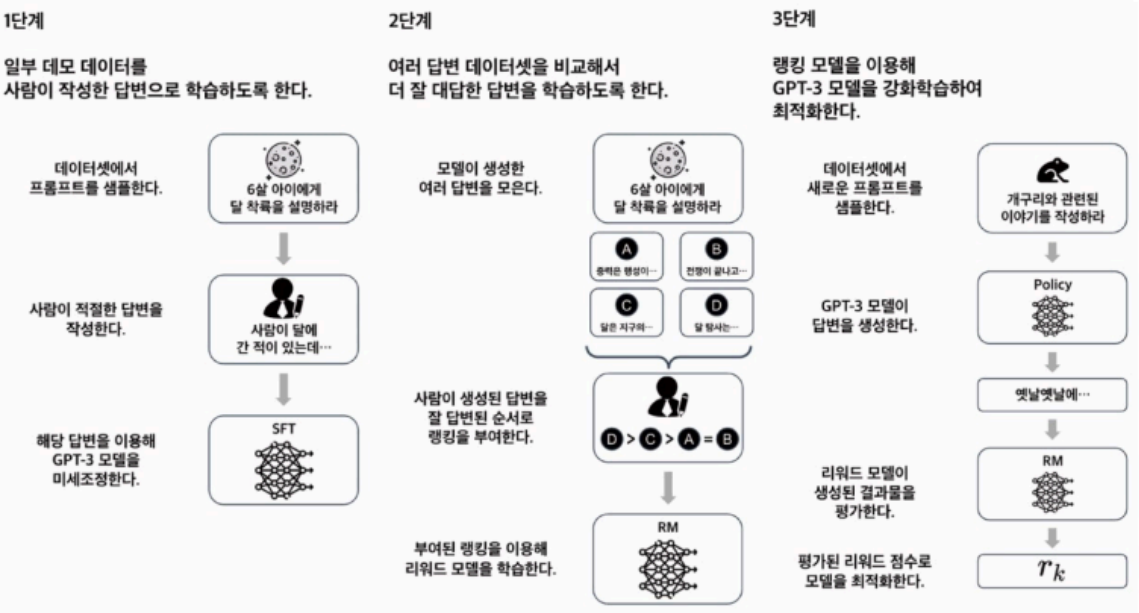

그림 7.12 RLHF GPT-3 학습 방법

RLHF 학습 방법

1. 데이터세트에서 임의의 프롬프트를 가져온다
2. 이 프롬프트에 대해 사람이 직접 적절한 답을 작성
3. 이때 작성된 답변은 모델이 생성한 문장과 함께 평가해 모델이 생성한 문장이 자연스러우면 보상받고, 그렇지 않으면 보상 X (강화학습) → 지도 미세 조정

그러나 현실적으로 모든 프롬프트에 사람이 답변을 다는 것은 어렵기 때문에 GPT가 생성한 여러 답변들에 랭킹을 부여한다. 사람이 부여한 랭킹은 답변의 우수성을 평가한 결과이다.

GPT 3언어 모델이 답변을 생성하고 사람의 피드백을 기반으로 한 학습된 보상 모델을 평가한다.

하지만 여전히 모델이 인간과 같은 추론과 판단 능력을 갖추지 못하고 부적절하거나 오류가 많은 답변을 제시하는 환각 현상이 존재한다.

GPT-4

GPT-4 : 이전의 GPT 모델과 달리 이미지 데이터도 인식 가능한 멀티모달 모델

- GPT3.5의 8배인 최대 32,768 개의 토큰을 입력받을 수 있음
- 25000개의 단어 처리 가능
- MMLU 벤치 마크 데스트 결과, GPT3.5에서는 약 70%의 성능을, GPT4에서는 약 85% 성능
- 한국어로 번역한 MMLU 벤치마크에서 77% 성능으로, GPT 3.5 영어보다 좋은 성능

- GPT 4는 모델의 매개변수 수나 학습데이터 및 방법 공개 X

모델 실습

```
model = GPT2LMHeadModel.from_pretrained(pretrained_model_name_or_pat
```

GPT2LMHeadModel 클래스의 from_pretrained 메서드로 사전학습된 GPT-2 모델을 불러옴

```
from transformers import pipeline

generator = pipeline(task="text-generation", model="gpt2")
outputs = generator(
    text_inputs="Machine learning is",
    max_length=20,
    num_return_sequences=3,
    pad_token_id=generator.tokenizer.eos_token_id
)
print(outputs)
```

입력 텍스트 (text_inputs) : 생성하려는 문장의 입력문맥 전달

최대길이(max_length) : 생성될 문장의 최대 토큰 수 제한

반환 시퀀스 개수(num_return_sequence) : 생성할 텍스트 시퀀스의 수 의미

패딩 토큰 ID (pad_token_id) : 모델의 자유생성 여부 설정