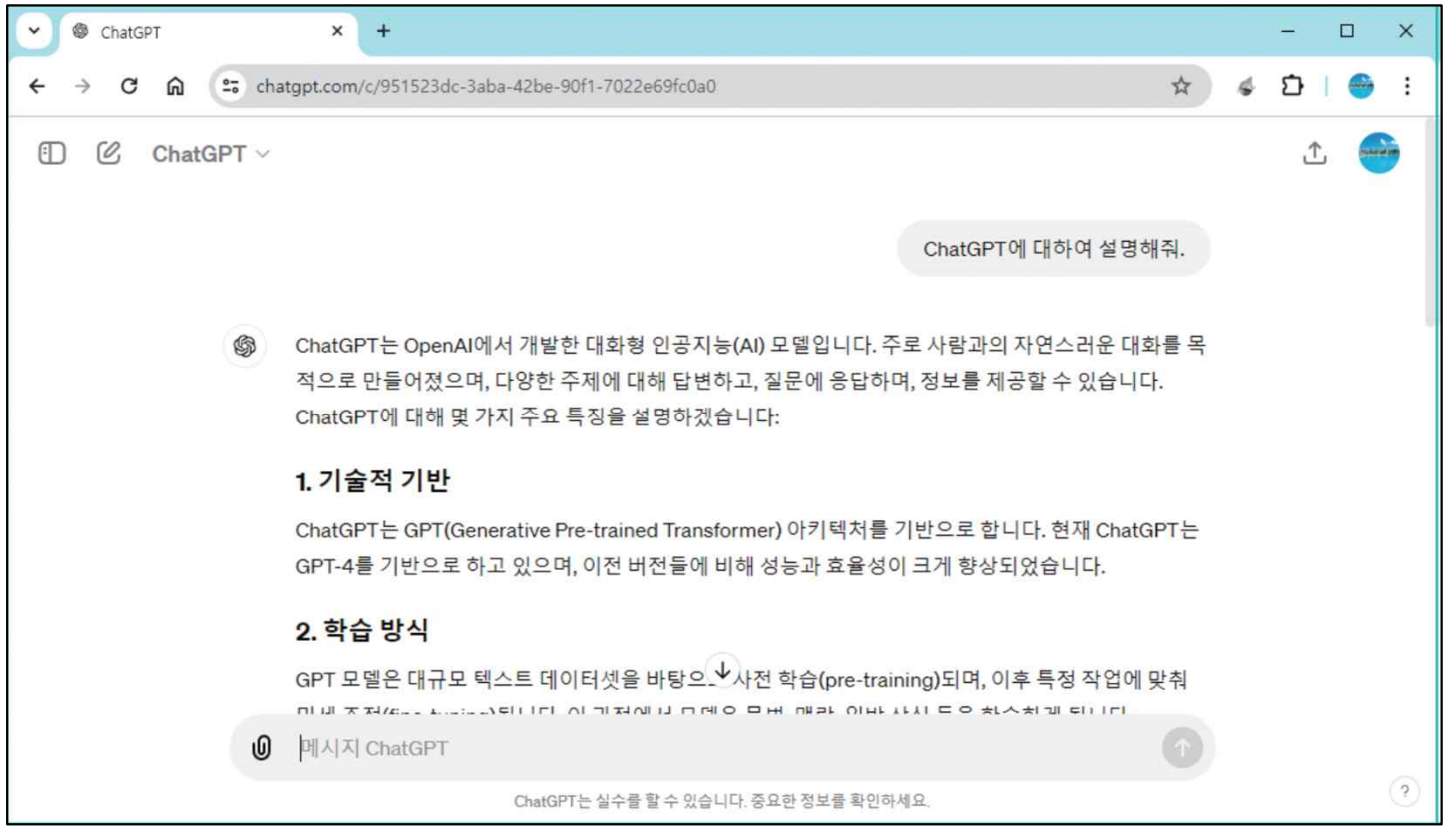
13장 생성형 언어 모델 GPT

학습 목표

- 기존의 자연어 처리 모델의 한계를 이해한다.
- 트랜스포머 모델을 이해한다.
- 셀프 어텐션 메커니즘을 이해한다.
- GPT 모델을 이해한다.
- picoGPT 소스를 파악한다..

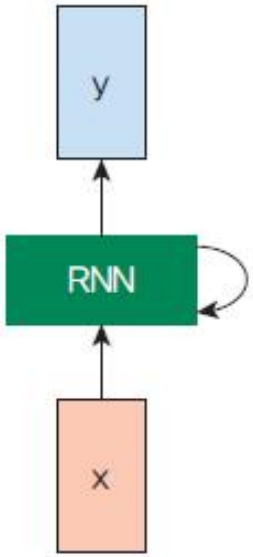
ChatGPT의 등장

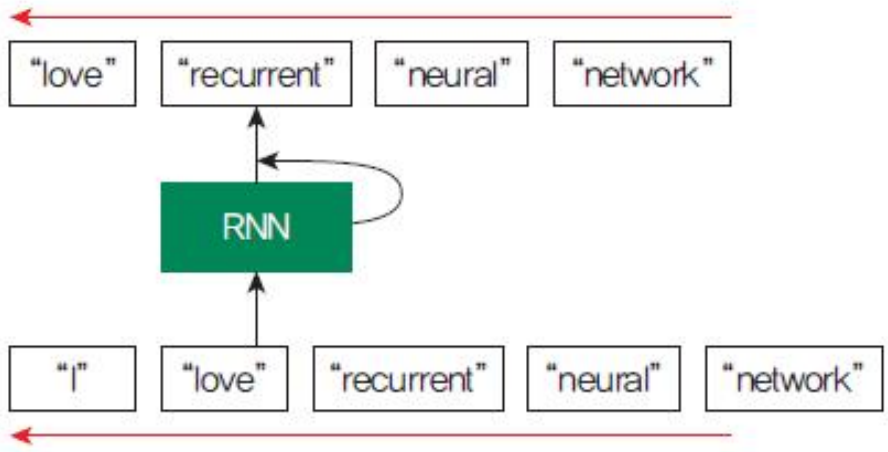
- ChatGPT는 OpenAI에서 개발하고 2022년 11월 30일에 출시된 챗봇이다. ChatGPT는 우리에게 많은 충격을 주었다.
- ChatGPT는 다른 챗봇하고는 다르게, 사람과의 대화가 자연스럽고 일관되게 이루어져서, 전례 없는 강력한 기능을 갖춘 것으로 평가되었다.



이전의 자연어 처리 모델

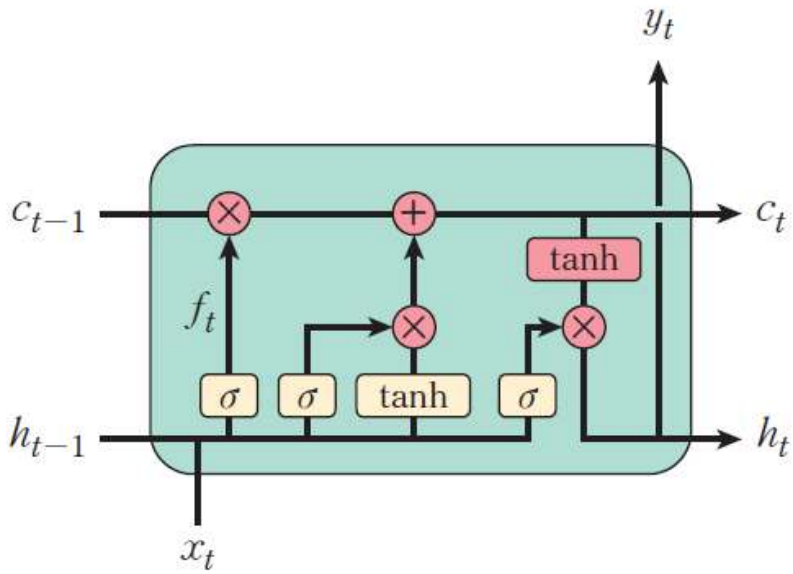
- Recurrent Neural Networks(RNN): RNN은 자연어 처리에서 시퀀스 데이터를 처리하는 데 처음으로 사용되었다. 하지만 RNN은 긴 시퀀스를 처리할 때 그래디언트 소실 또는 그래디언트 폭주와 같은 문제를 겪었다. 이로 인해 장기 의존성(Long-term dependencies)을 잘 학습하지 못하는 문제가 발생했다.





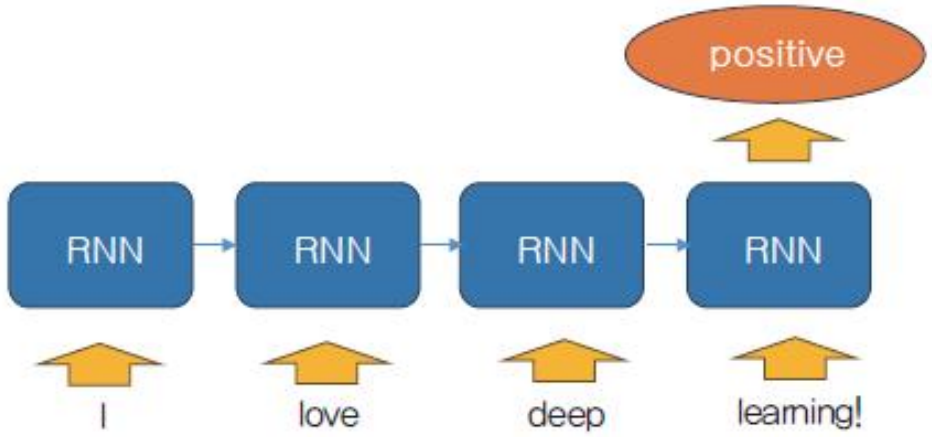
이전의 자연어 처리 모델

- Long Short-Term Memory(LSTM): LSTM은 RNN의 단기 기억 한계와 그래디언트 소실 문제를 해결하기 위해 등장했다. LSTM은 장기 의존성을 더 잘 학습할 수 있도록 설계되었고, 게이트 메커니즘을 사용하여 정보의 흐름을 제어할 수 있게 되었다.
- 하지만 RNN과 마찬가지로 LSTM도 역전파 과정에서 그래디언트가 소멸되는 문제가 발생할 수 있다. 특히 장기 의존성이 큰 시퀀스 데이터에서발생할 수 있으며, 이는 모델이 오래된 정보를 잘 기억하지 못하게 만들 수 있다.



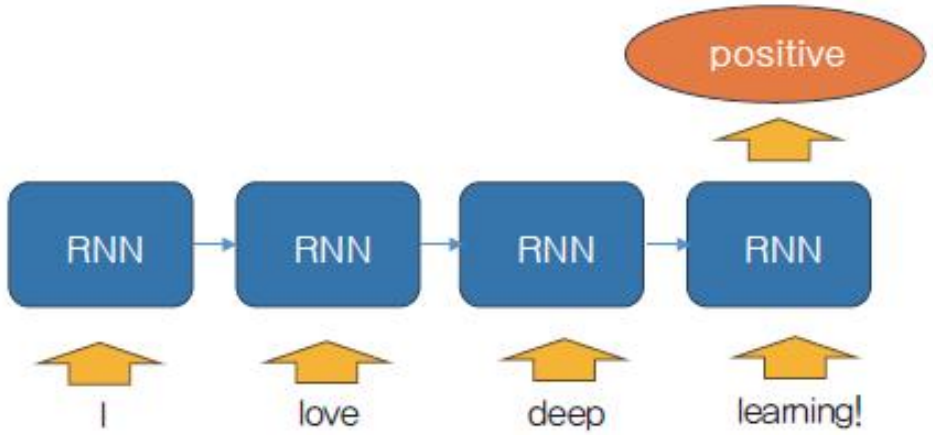
기존의 감정 판단 모델

- RNN을 이용하여 자연어를 처리하는 모델의 전형적인 예를 들어보자. “I love deep learning!” 문장의 단어들을 순차적으로 받아서 문장의 센티먼트를 출력하는 신경망은 다음과 같이 동작할 것이다.
- 여기서 항상 주의할 점은 한 개의 RNN 셀이 순차적으로 각 단어들을 처리한다는 점이다. 문장은 단어들로 분리되어서 RNN에 순차적으로 입력되고 마지막 단어가 입력을 마치면 “positive”라는 센티먼트가 출력된다.



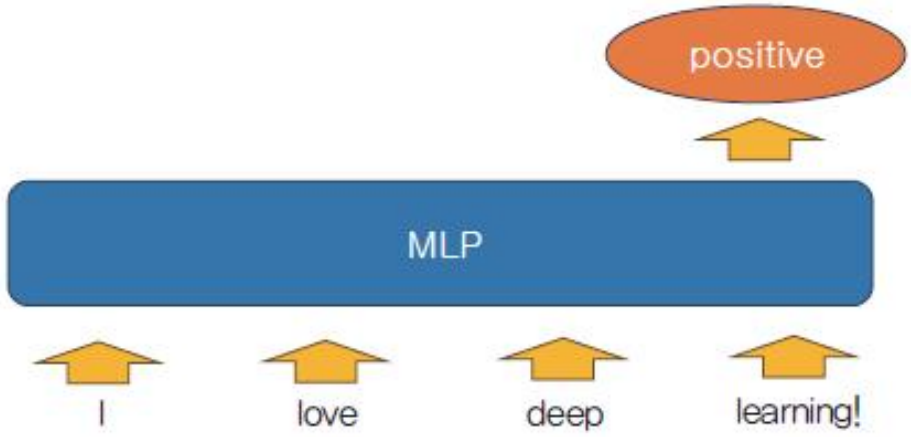
단점

- 병렬 처리 어려움: RNN은 각 시간 단계가 이전 시간 단계의 출력에 의존하기 때문에 병렬 처리가 어렵다.
- 장기 의존성 모델링 어려움: RNN은 긴 시퀀스에서 장기 의존성을 학습하는 데 어려움을 겪을 수 있으며, 이로 인해 일부 정보가 손실될 수 있다.
- 그래디언트 소실 문제(Vanishing Gradient Problem): RNN은 긴 시퀀스에 대한 정보를 전파하기 어려워 "그래디언트 소실" 문제가 발생할 수 있다.



RNN 모델의 한계를 극복하려면?

- 순차성을 극복하는 하나의 방법은 입력 단어들을 연결하여 MLP와 같은 순방향 신경망에 전부 한 번에 제시하는 방법이다. 예를 들어서 “I love deep learning!” 문장의 단어들의 엠베딩 벡터 v1, v2, v3, v4를 전부 연결하여서 MLP과 같은 신경망에 [v1, v2, v3, v4]와 같이 제시하고 모델을 학습시켜서 문장의 센티먼트를 출력한다고 가정하자.



이 방법의 문제점

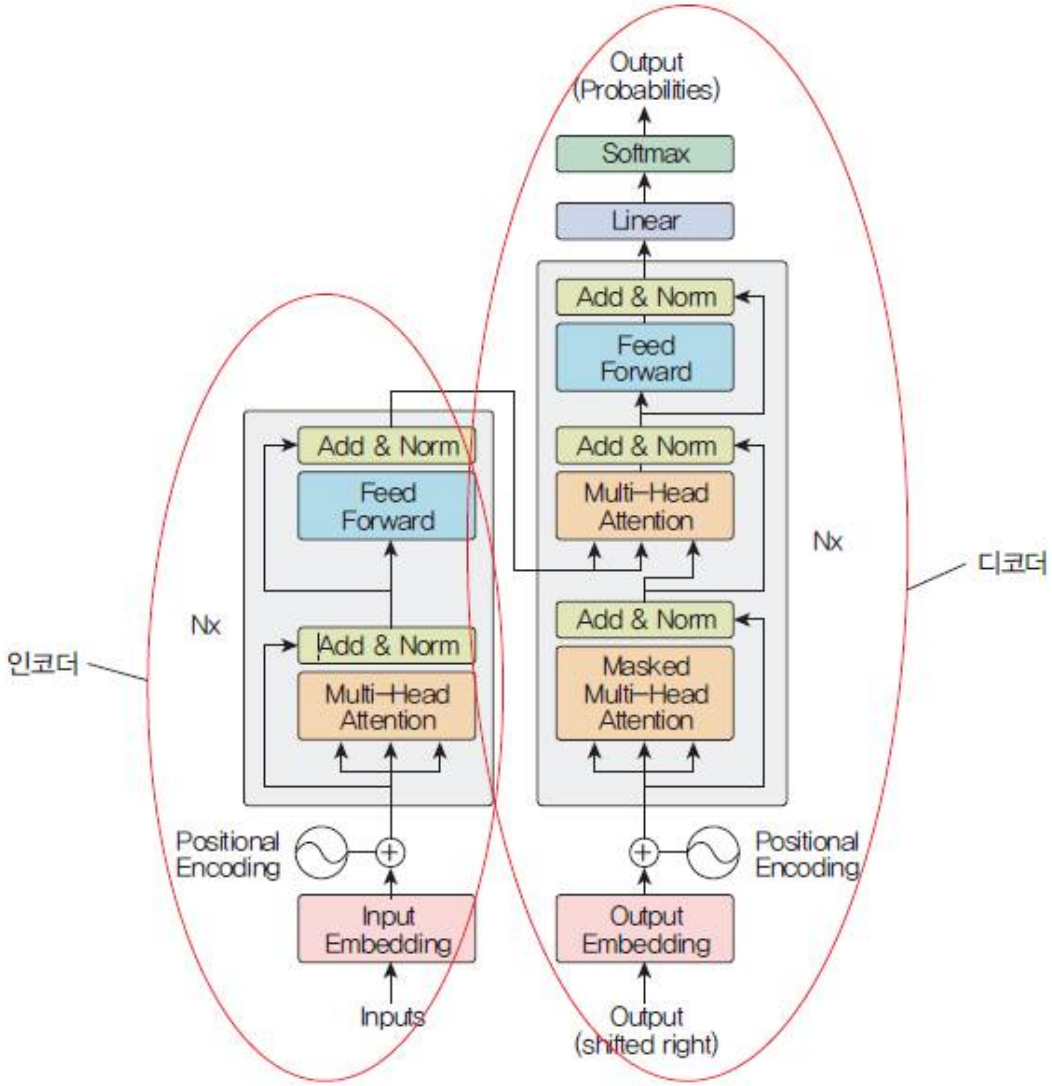
- 순서 정보 손실: MLP는 입력 벡터의 순서를 인식하지 않는다. 실제로는 문장이나 단어로 분리하지도 않을 것이다. 고정된 MLP 모델은 가변적인 입력을 처리하는데 문제가 발생한다.
- 문맥 이해 부족: 자연어 처리에서는 단어의 의미는 주변 단어와의 관계에 의해 결정된다. MLP는 단어 간 상호작용(의미 관계)을 고려하지 않으며, 따라서 문맥을 이해하는 데 어려움을 겪을 수 있다. MLP는 문장을 단어로 분리하지도 않는다.
- RNN과의 비교: 반면, RNN은 시퀀스 데이터를 처리하는 데 특화되어 있다. 각 단계에서 이전 단계의 출력을 입력으로 사용하여 순서 정보를 유지하고 문맥을 이해할 수 있다. 따라서 자연어 처리에서는 RNN이 MLP보다 더 적합한 선택이다.

트랜스포머 모델

- 기존의 자연어 처리 모델에서는 순환 신경망(RNN, Recurrent Neural Network)이나 LSTM(Long Short-Term Memory) 같은 신경망 모델이 사용되었다. 그러나 트랜스포머는 이러한 순환 모델을 사용하지 않고, 위치 엔코딩 메커니즘을 사용한다.
- 위치 인코딩은 단어 임베딩에 위치 정보를 추가하여, 신경망 모델이 단어의 위치를 인식할 수 있도록 돕는다.
- 순환 구조가 없기 때문에, 병렬 처리가 가능해 학습 및 처리 속도가 빠르다.
- 트랜스포머 모델은 문맥 이해를 셀프 어텐션(Self Attention) 메커니즘으로 해결한다. 셀프 어텐션 메커니즘은 입력 시퀀스의 각 단어가 다른 모든 단어와의 관계를 학습할 수 있도록 도와준다.

트랜스포머 모델의 구조

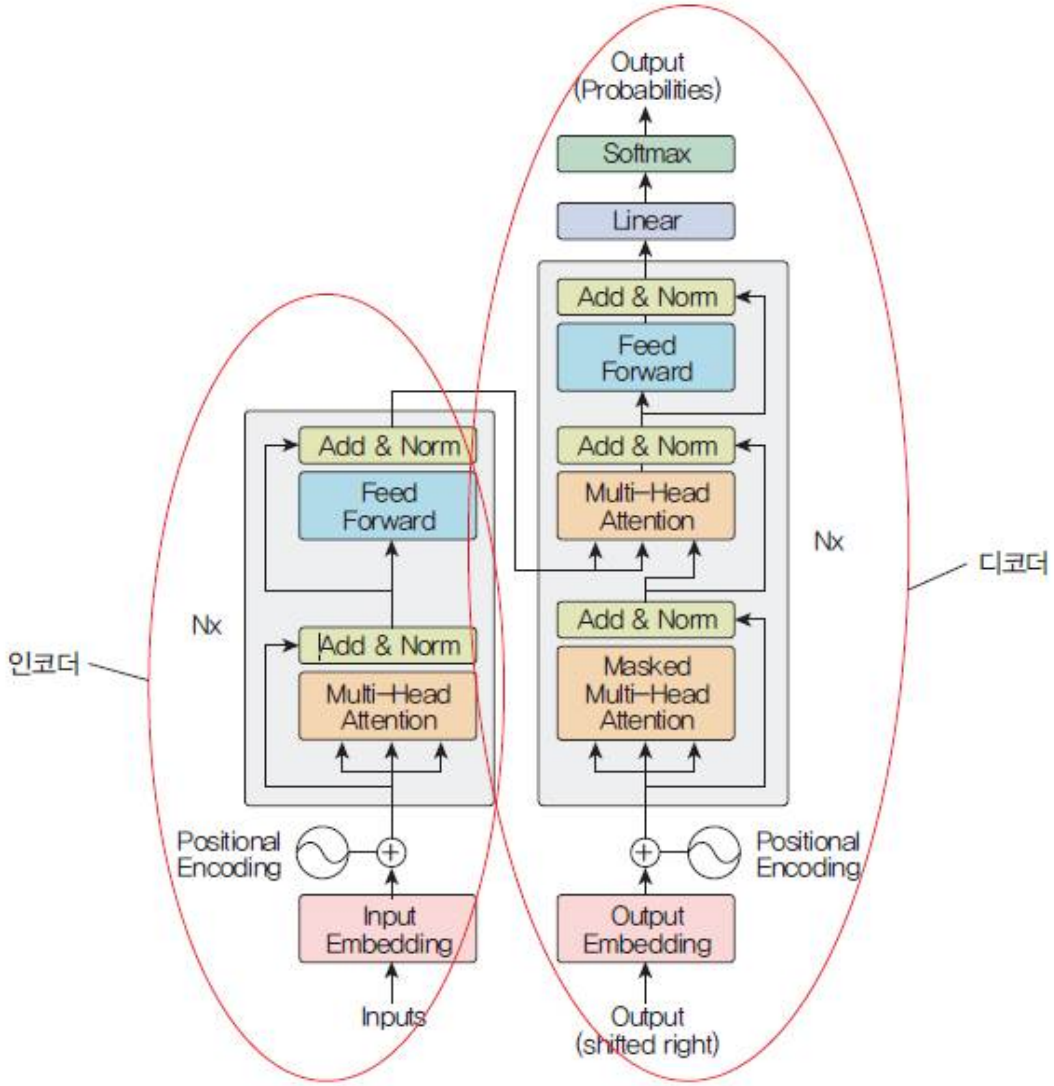
- 트랜스포머 모델은 기계 번역을 위한 구조이다. 트랜스포머 모델은 인코더와 디코더 블록으로 구성된다.
- 인코더는 입력 언어를 받아들여서 분석한다. 디코더는 인코더로부터 컨텍스트 정보를 받아서 출력 언어로 문장을 번역하여 출력한다.



인코더와 디코더

- 인코더: 인코더는 동일한 6개의 층으로 구성된다. 첫 번째 하위 층은 멀티-헤드 셀프-어텐션 층이고, 두 번째 하위 층은 완전 연결 순방향 신경망이다.

- 디코더: 디코더 역시 동일한 6개의 층으로 구성된다. 디코더는 인코더 층의 두 개의 하위 층에, 멀티-헤드 어텐션을 수행하는 세 번째 하위 층을 추가한다.



트지ㅇ

- 셀프 어텐션 메커니즘: 트랜스포머의 가장 핵심적인 요소 중 하나는 셀프 어텐션 메커니즘이다. 셀프 어텐션은 RNN과 같은 순환적인 구조 없이도 입력 시퀀스 내 단어 간의 상호 의존성을 학습한다. 순환적인 구조를 사용하지 않기 때문에 학습 과정을 병렬화시킬 수 있다.

- 위치 인코딩: 트랜스포머 모델은 입력 시퀀스 내 단어의 상대적인 위치 정보를 나타내기 위해 위치 인코딩을 사용한다. 이를 통해 모델은 입력 시퀀스 내 단어의 순서 정보를 학습할 수 있다.

- 멀티 어텐션 헤드: 단일 어텐션은 각 단어의 관계를 단일 관점에서만 파악할 수 있다. 이를 보완하기 위해 트랜스포머는 여러 개의 셀프 어텐션을 병렬로 수행하는 멀티 어텐션을 사용한다. 이는 다양한 관점에서 단어 간의 관계를 파악할 수 있게 한다.

기계 번역의 예시

- 여기서는 영어 문장을 프랑스어 문장으로 번역하는 예를 들어 설명해보자.

(1) 토큰화: 영어 문장을 단어 또는 서브워드 단위로 토큰화한다.

입력: ["I", "love", "deep", "learning", "."]

(2) 임베딩: 각 토큰을 고유한 정수 ID로 변환한 후, 임베딩 벡터로 변환한다.

"I" -> [0.1, 0.2, ...]

"love" -> [0.3, 0.4, ...]

"deep" -> [0.5, 0.6, ...]

"learning" -> [0.7, 0.8, ...]

기계 번역의 예시

(3) 위치 인코딩: 각 임베딩 벡터에 위치 정보를 더하여 입력 시퀀스의 순서를 반영한다.

(4) 인코더: 인코더가 입력 시퀀스를 처리하여 각 위치에 대한 컨텍스트 정보를 포함하는 일련의 벡터를 출력한다.

예: [Enc1, Enc2, Enc3, Enc4]

(5) 디코더: 디코더는 처음에 시작 토큰 <s>을 입력으로 받아 인코더의 컨텍스트 벡터를 참조하여 프랑스어의 첫 단어 “J”를 예측한다.

시작 토큰: "<s>"

예: "J'"

기계 번역의 예시

(6) 반복적 예측: 디코더는 이전에 예측한 단어들을 다시 입력으로 받아 다음 단어를 예측한다.

입력: ["<s>", "J'"]

출력: "aime"

입력: ["<s>", "J'", "aime"]

출력: "l"

입력: ["<s>", "J'", "aime", "l'"]

출력: "apprentissage"

...

최종 출력: "J'aime l'apprentissage profond."

트랜스포머와 GPT 모델

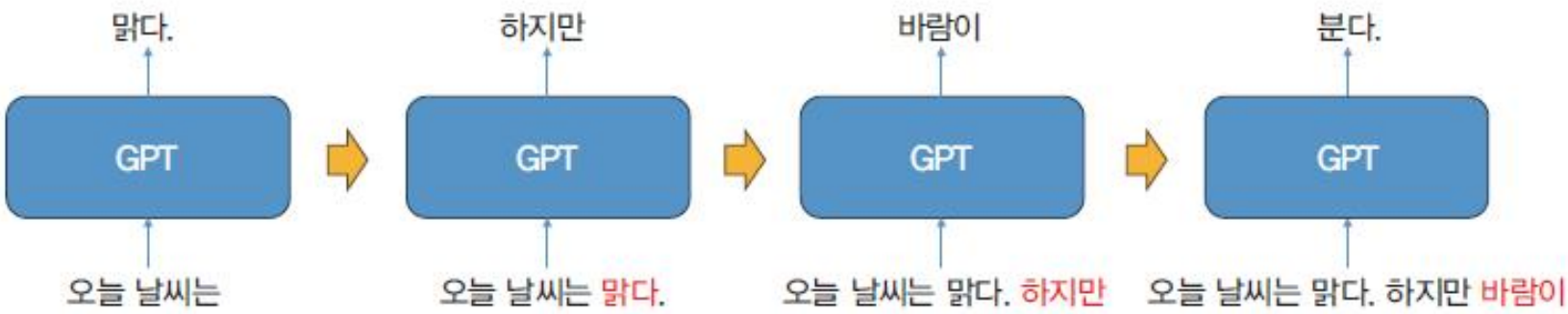
- GPT(Generative Pre-trained Transformer)는 트랜스포머 아키텍처를 기반으로 한 자연어 처리 모델이다. GPT와 같은 생성형 AI에서는 입력 언어와 출력 언어가 동일하다. 따라서 인코더가 필요 없다. 실제로 GPT는 트랜스포머의 디코더만을 사용한다.

- GPT는 이러한 트랜스포머 아키텍처를 활용하여 대규모 텍스트 데이터셋을 사전 학습한 후, 다양한 자연어 처리 작업에 적용될 수 있다.

- GPT는 기본적으로 주어진 시작 문장에서 다음 단어를 예측하고, 그 단어를 다시 입력에 추가하여 반복하는 방식으로 텍스트를 생성한다. 이것을 자기 회귀(Self Regression)이라고 한다.

GPT 모델

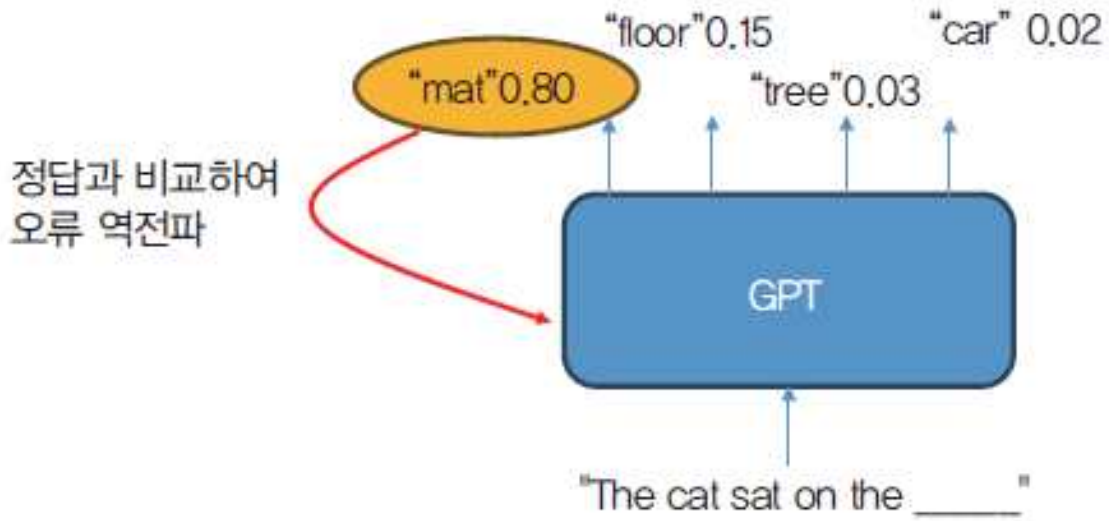
- GPT는 입력된 토큰 시퀀스를 기반으로 다음에 올 단어의 확률 분포를 계산한다. 예를 들어, “오늘 날씨는” 다음에 올 단어로 “좋다”, “나쁘다”, “맑다” 등의 단어에 각각의 확률을 할당한다.
- 확률 분포에서 가장 높은 확률을 가진 단어를 선택하거나, 확률에 따라 샘플링하여 다음 단어를 결정한다.



GPT 모델의 학습 과정

- 학습 과정에서 GPT에게 다음과 같이 빈칸이 있는 텍스트를 제공하고 빈칸의 단어를 추측하게 한다.

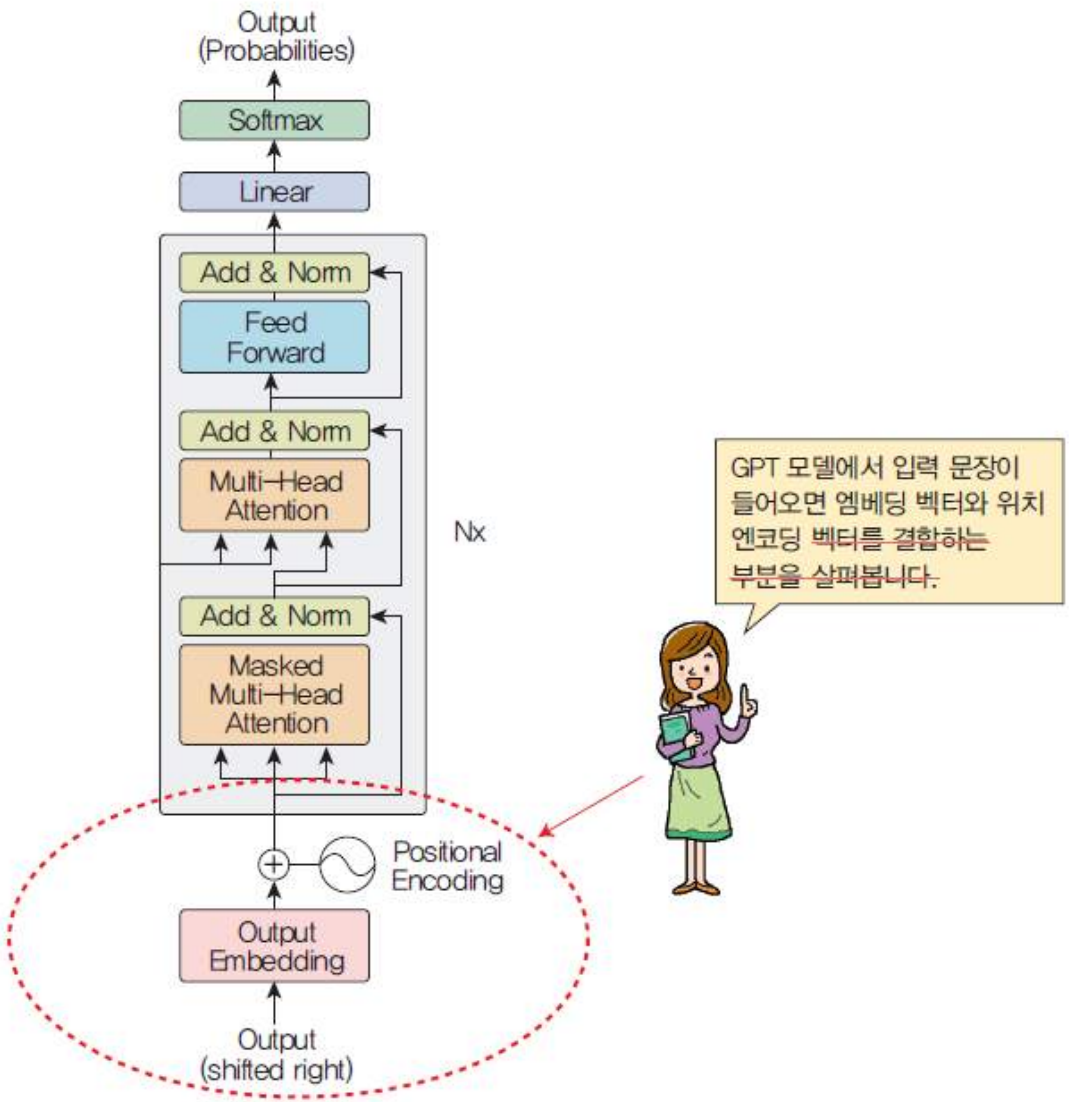
"The cat sat on the _____"



GPT 모델의 학습 자료

- GPT는 인터넷에 공개된 다양한 텍스트 데이터를 기반으로 학습되었다.
  - 서적: 다양한 주제의 책들
  - 웹사이트: 위키백과, 뉴스 사이트, 포럼, 블로그 등
  - 학술 논문: arXiv와 같은 공개된 학술 논문 저장소
  - 기타: 기타 공개된 텍스트 자료

GPT에서의 엠베딩 벡터와 위치 엔코딩



엠베딩 벡터 계산

(1) 토큰화(Tokenization)

- 입력된 텍스트는 먼저 토큰으로 분리된다. GPT-3의 어휘 사전(vocabulary) 크기는 50,257 단어이다.
- 이 사전 크기는 Byte Pair Encoding(BPE) 기법을 사용하여 텍스트를 토큰으로 분할하여 구성된다.
- BPE는 자주 등장하는 문자 또는 문자열 쌍을 병합하여 서브워드 단위의 토큰을 만들어내는 방법으로, 단어와 서브워드, 심지어 개별 문자까지도 토큰화할 수 있다.

```
["I", "love", "deep", "learning"]
```

엠베딩 벡터 계산

(2) 토큰 인덱싱(Token Indexing)

- 토큰으로 분리된 단어들은 사전에 정의된 어휘(vocabulary)에서 해당 토큰에 할당된 인덱스로
- 변환된다. 예를 들어서 [“I”, “love” “deep” “learning”]은 다음과 같은 정수 리스트가 될 수 있다.

```
[105, 6877, 5422, 3578]
```

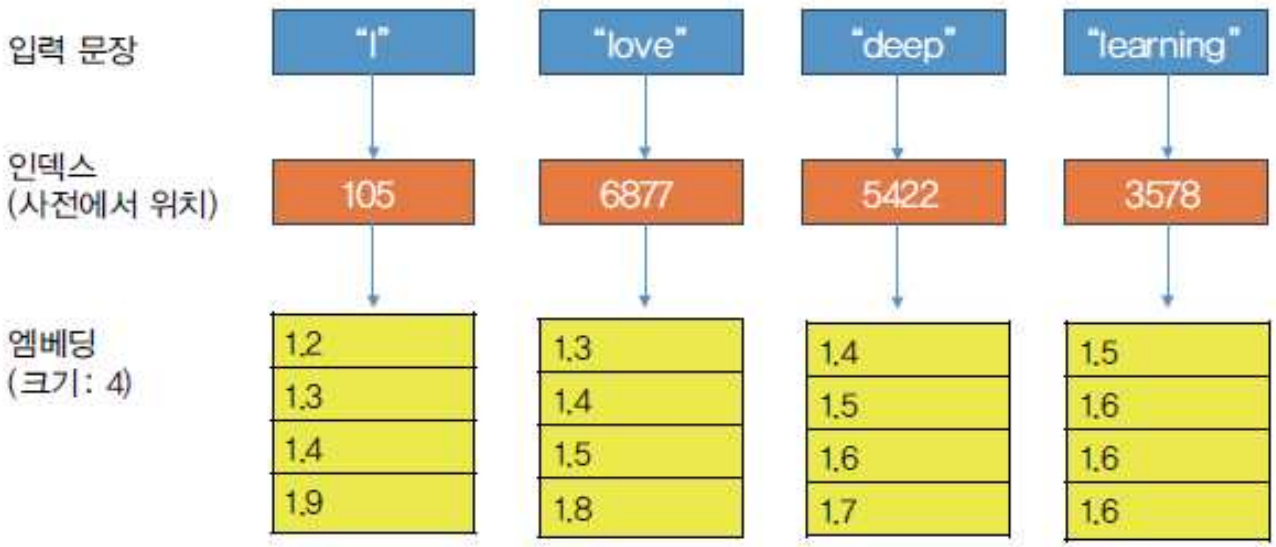
엠베딩 벡터 계산

(3) 엠베딩 벡터(Embedding Vector)

- 각 토큰 인덱스는 고정된 크기의 엠베딩 벡터로 변환된다. 트랜스포머 모델은 학습 과정에서 이 임베딩 벡터를 최적화한다.
- 즉 토큰을 엠베딩 벡터로 변환하는 엠베딩 행렬이 있고 학습 과정에서 이 행렬도 학습된다. 이 벡터는 일반적으로 512, 1024, 2048 차원 등 고차원 공간에서의 벡터이다.

```
[1.2, 1.3, 1.4, 1.9]
```

엠베딩 벡터 계산


입력 문장
"I"
"love"
"deep"
"learning"
인덱스
(사전에서 위치)
105
6877
5422
3578
엠베딩
(크기: 4)
1.2
1.3
1.4
1.9
1.3
1.4
1.5
1.8
1.4
1.5
1.6
1.7
1.5
1.6
1.6
1.6

위치 인코딩

- 자연어 처리 모델에서 문장 내 단어의 순서와 위치를 이해하는 것이 텍스트의 의미를 파악하는 데 매우 중요하다.
- 순차적으로 데이터를 처리하는 RNN(순환 신경망)과 달리, 트랜스포머(Transformer) 모델은 모든 단어를 병렬로 처리한다. 따라서 모델이 문장 내에서 단어의 위치 정보를 알 수 있도록 추가적인 메커니즘이 필요하며, 이를 위해 위치 인코딩(Position Encoding)을 사용한다.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

여기서 pos는 토큰의 위치, i는 위치 인코딩 벡터에서의 인덱스, d_{model}은 임베딩 차원

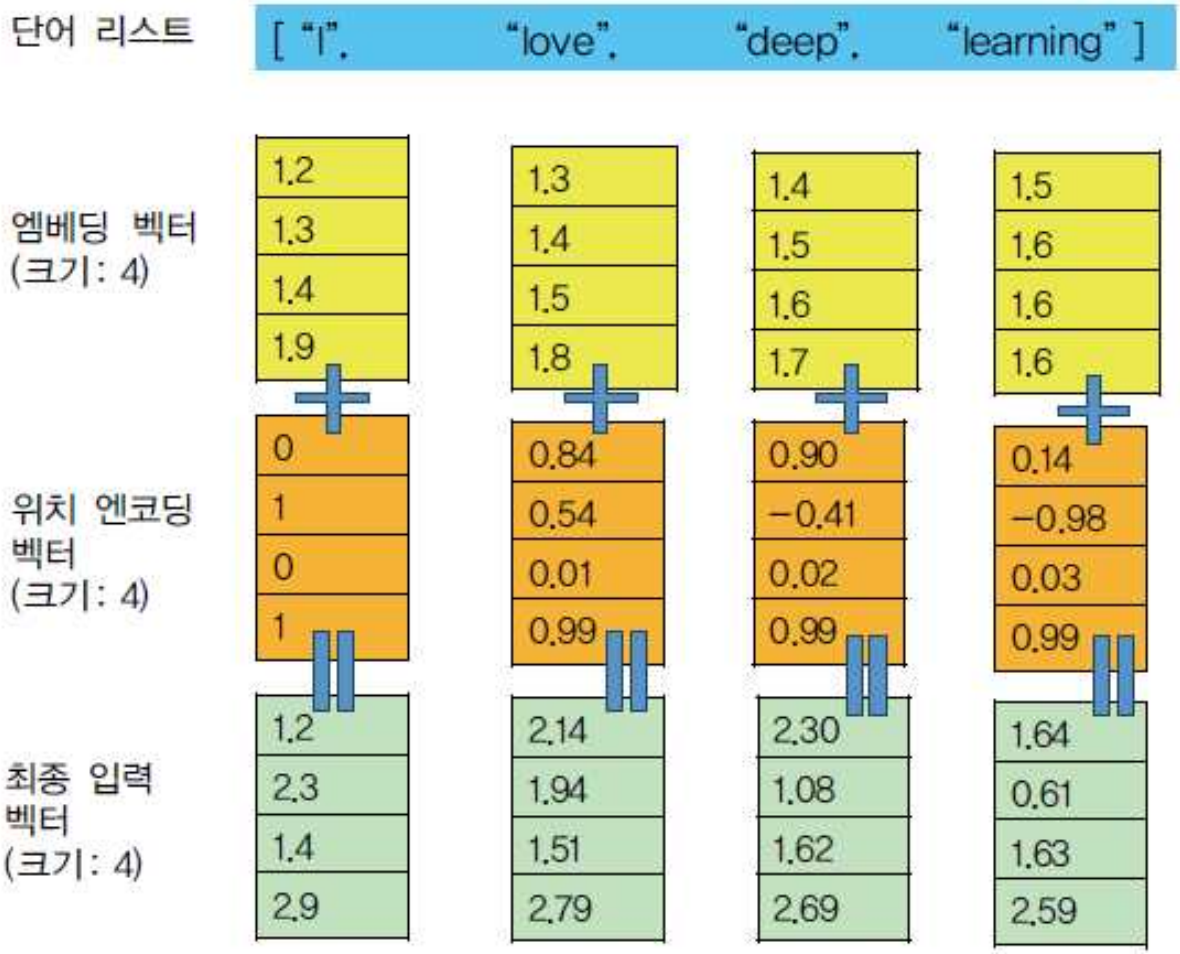
위치 인코딩의 예

- 예를 들어, 임베딩 벡터의 차원이 4라면, 첫 번째 단어의 위치 인코딩 벡터는 다음과 같이 계산된다.

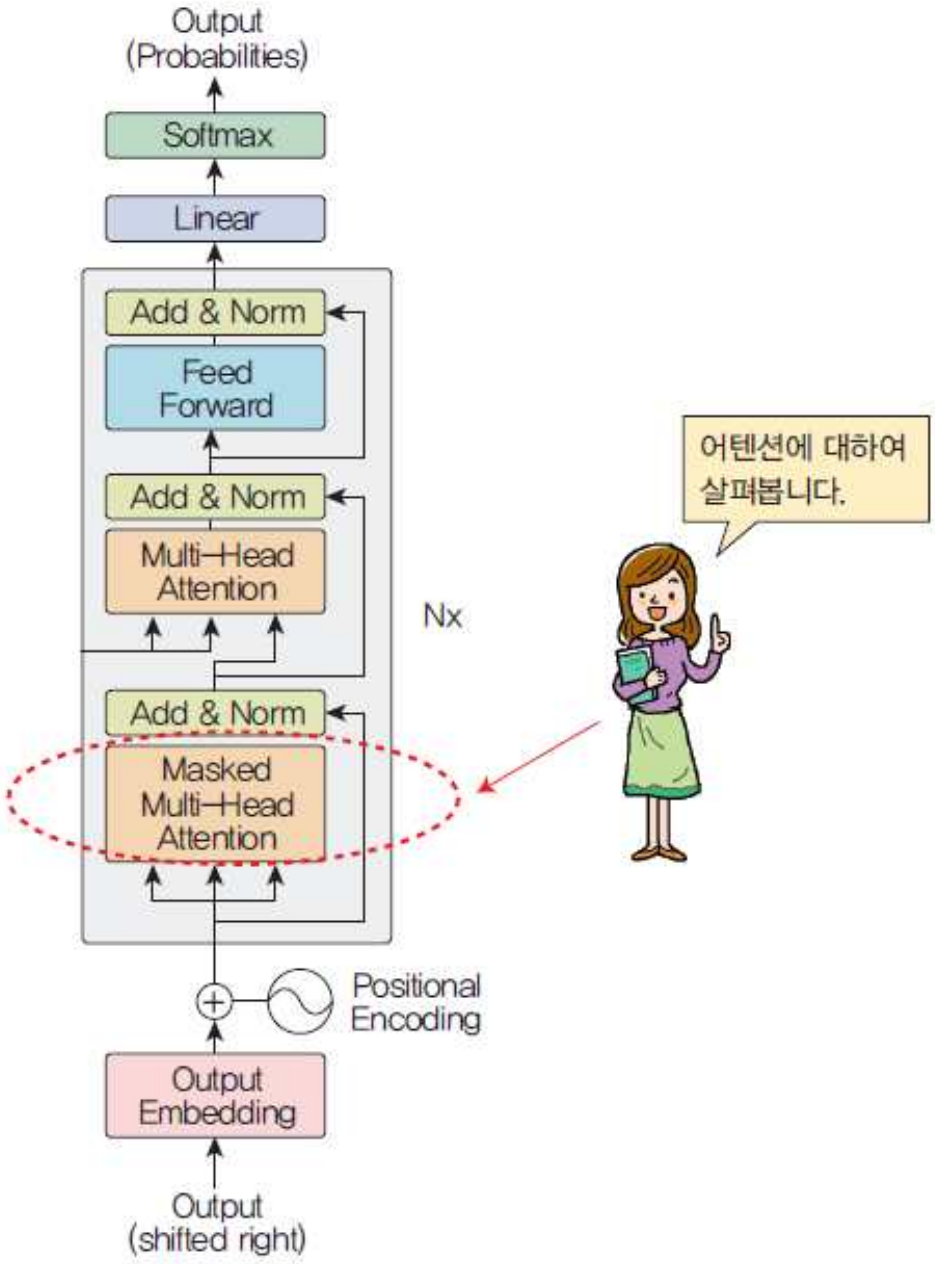
- $PE(0,0) = \sin\left(\frac{0}{10000^{0/4}}\right) = \sin(0) = 0$
- $PE(0,1) = \cos\left(\frac{0}{10000^{1/4}}\right) = \cos(0) = 1$
- $PE(0,2) = \cos\left(\frac{0}{10000^{2/4}}\right) = \sin(0) = 0$
- $PE(0,3) = \cos\left(\frac{0}{10000^{3/4}}\right) = \cos(0) = 1$

임베딩 벡터와 위치 엔코딩 벡터의 결합

- 이 위치 인코딩 벡터는 각 단어의 임베딩 벡터에 더해져서 단어의 의미적 정보와 위치 정보를 결합한다.



GPT에서의 셀프 어텐션



셀프 어텐션이란?

- 어텐션(attention)은 우리말로 문장의 “주의”, 또는 “문맥”을 의미한다. 먼저 왜 자연어 처리에서 문맥이 중요한지 설명해보자.

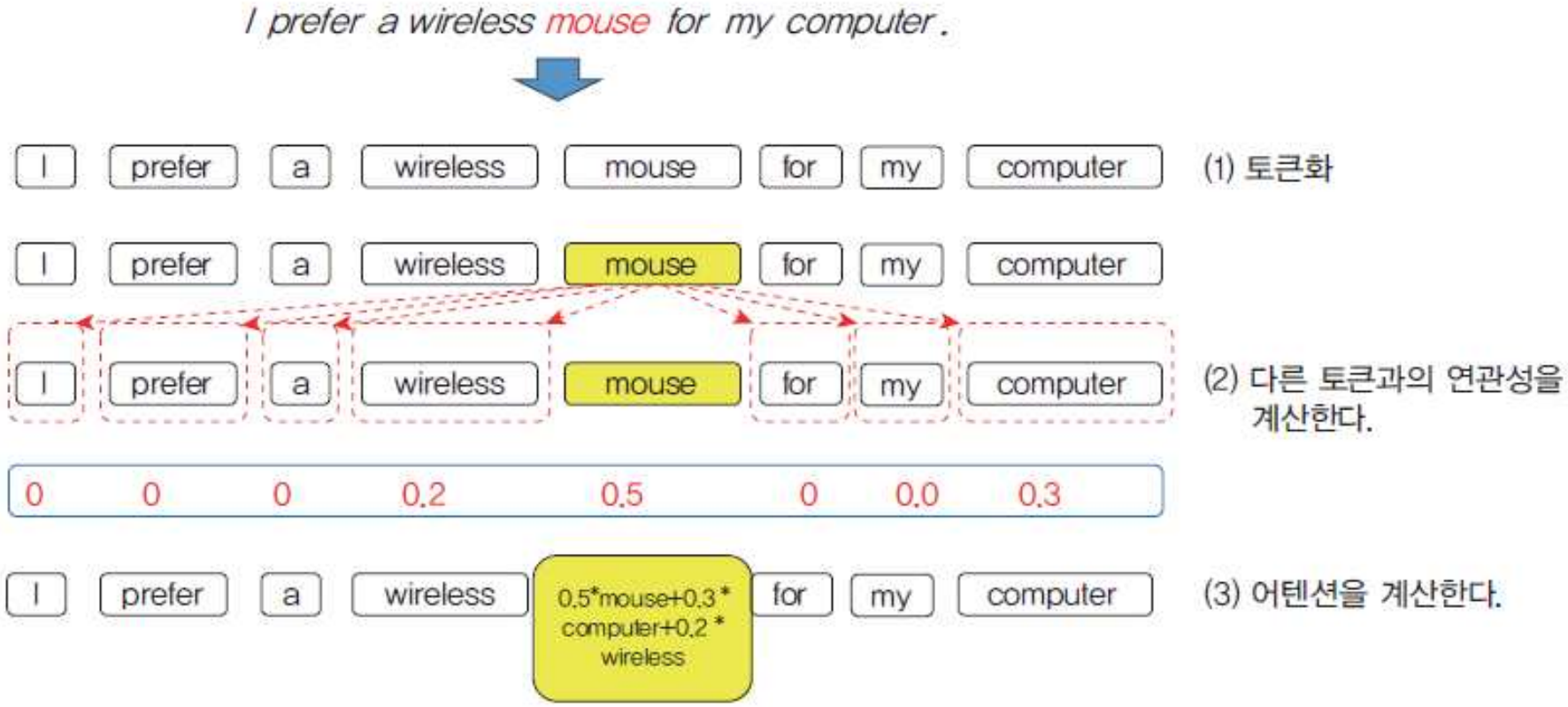
- 여기서 mouse라는 단어의 의미는 무엇일까? “쥐”일까? 아니면 컴퓨터에서 사용되는 입력 장치인 “마우스”일까? 이것을 알려면 mouse 주위의 단어들을 살펴보아야 한다.

셀프 어텐션

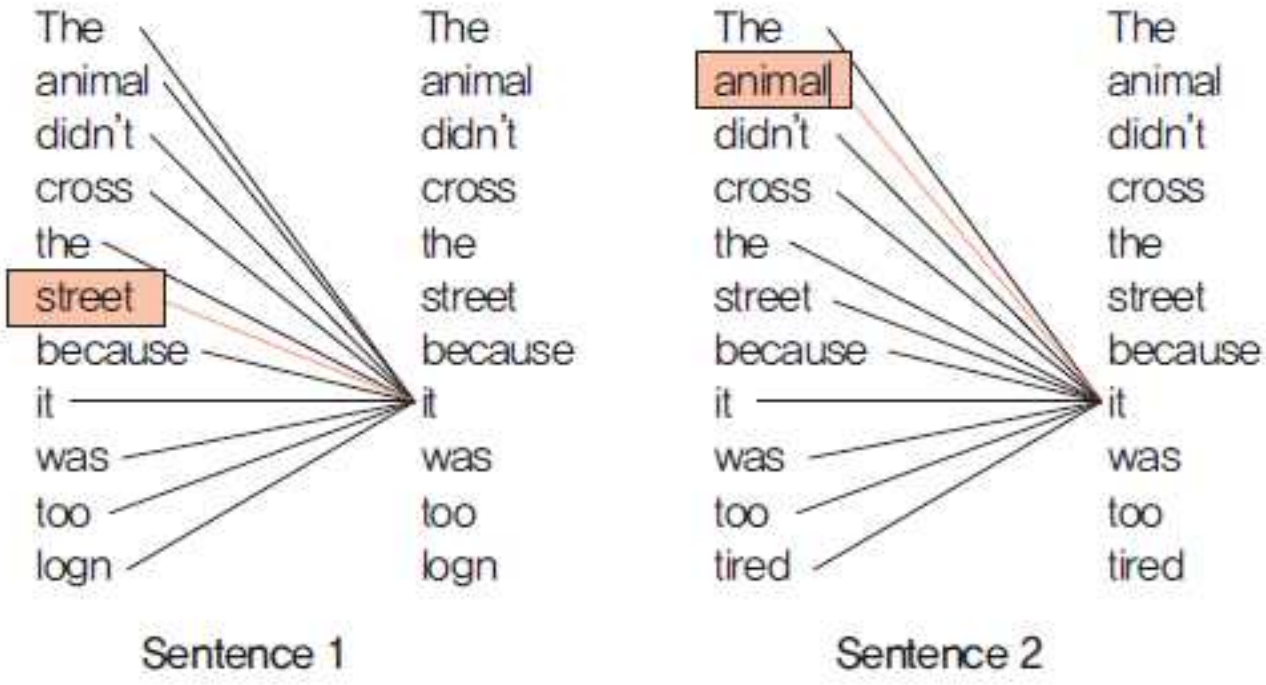
- 현재 단어의 의미 해석을 향상시킬 수 있는, 주변 단어들을 평가하는 메카니즘을 만들 수 있을까?

- 문장에서 주변 단어의 영향을 평가할 수 있는 방법을 찾으면 된다. 이들 값을 이용하여 우리는 현재 관심 단어의 의미를 변경하거나 향상시킬 수 있다.

셀프 어텐션


I prefer a wireless mouse for my computer .
I prefer a wireless mouse for my computer
(1) 토큰화
(2) 다른 토큰과의 연관성을 계산한다.
0 0 0 0.2 0.5 0 0.0 0.3
0.5*mouse+0.3*computer+0.2*wireless
(3) 어텐션을 계산한다.

셀프 어텐션의 다른 예


The
animal
didn't
cross
the
street
because
it
was
too
logn
The
animal
didn't
cross
the
street
because
it
was
too
logn
Sentence 1
The
animal
didn't
cross
the
street
because
it
was
too
tired
The
animal
didn't
cross
the
street
because
it
was
too
tired
Sentence 2

셀프 어텐션 메커니즘

(1) 쿼리(Query), 키(Key), 값(Value) 벡터 생성: 입력 문장의 각 단어에 대해 쿼리, 키, 값 벡터를 생성한다. 이들 벡터는 단어 간의 관련성을 판단하는 데 사용된다. 보통은 입력 문장의 각 단어에 대해 행렬을 곱하여 쿼리, 키, 값 벡터를 생성한다.

(2) 점수 계산: 각 단어의 쿼리 벡터와 다른 단어들의 키 벡터 사이의 점수를 계산한다. 이는 내적(dot product)을 사용하여 계산될 수 있다. 이 단계를 통해 각 단어마다 다른 단어들과의 상관 관계를 나타내는 점수가 계산된다.

(3) 소프트맥스 함수 적용: 각 단어에 대한 점수에 소프트맥스 함수를 적용하여 정규화한다. 이렇게 하면 각 단어에 대한 가중치가 생성된다. 이 가중치는 해당 단어가 다른 단어들과 얼마나 관련이 있는지를 나타낸다.

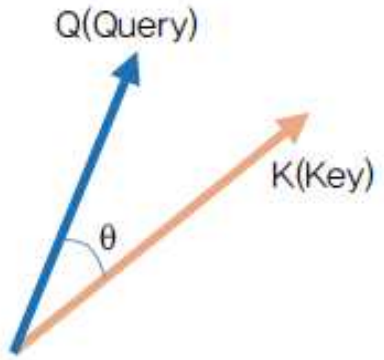
(4) 가중합 계산: 각 단어의 값 벡터에 계산된 가중치를 곱하여 가중합을 계산한다. 이 과정을 통해 해당 단어가 다른 단어들과 어떤 정보를 주로 주고받았는지를 나타내는 새로운 벡터가 생성된다.

어떻게 각 단어들 사이의 관련성을 계산할 수 있을까?

- 벡터 간의 유사도를 계산하는 표준적인 방법은 벡터의 내적(inner product)을 사용하는 것이다

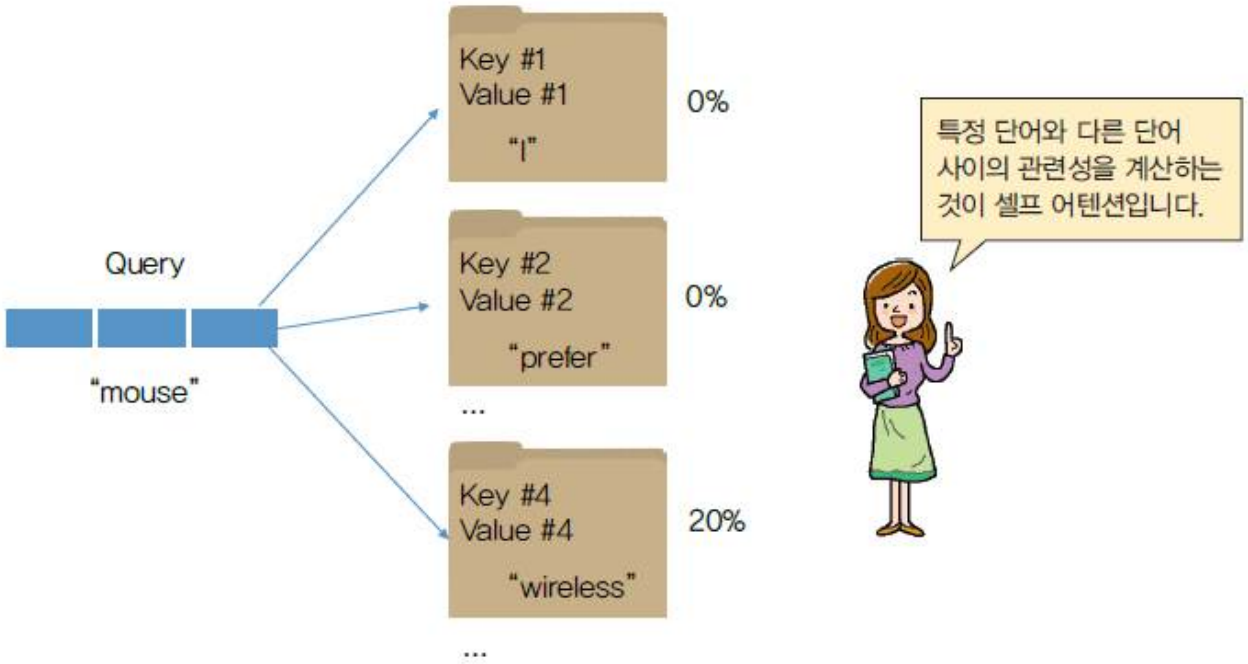
$$Q \cdot K = |Q||K|\cos(\theta)$$

- $Q \cdot K$는 Q와 K의 내적을 나타낸다.
- $|Q|$는 벡터 Q의 크기(또는 길이)를 나타낸다.
- $|K|$는 벡터 K의 크기(또는 길이)를 나타낸다.
- θ는 Q와 K 사이의 각도를 나타낸다.



쿼리(Query), 키(Key), 값(Value)

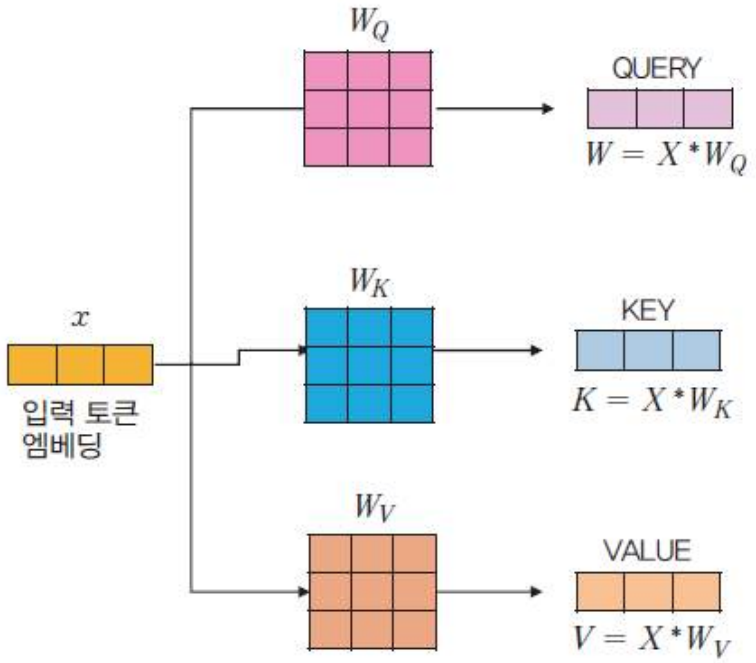
- 어텐션이 원래 기계 번역에 사용되었던 이유로 이러한 이름이 사용된다.
- 쿼리(Query) 벡터: 특정 단어의 정보가 담긴 벡터로, 다른 단어들과의 연관성을 계산할 때 사용된다.
- 키(Key) 벡터: 각 단어에 대한 식별 정보를 담고 있는 벡터로, 쿼리 벡터와 비교하여 연관성을 계산한다.
- 값(Value) 벡터: 각 단어의 실제 정보를 담고 있는 벡터로, 최종적인 결과를 형성한다.



쿼리(Query), 키(Key), 값(Value) 계산 과정

(1) 입력 토큰의 임베딩 벡터 X가 주어지면, 여기에 3개의 행렬을 곱하여 쿼리, 키, 값 벡터를 생성한다.

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$



쿼리(Query), 키(Key), 값(Value) 계산 과정

(2) 어텐션 점수 계산: 쿼리(Query)와 키(Key) 간의 유사도(스코어)를 계산한다. 이를 위해 쿼리 벡터와 키 벡터의 내적을 계산한다.

$$QK^T$$

어텐션 스코어	a ↓ E_1 ↓ K_1	wireless ↓ E_2 ↓ K_2	mouse ↓ E_3 ↓ K_3	for ↓ E_4 ↓ K_4	my ↓ E_5 ↓ K_5	computer ↓ E_6 ↓ K_6
a → E_1 → Q_1						
wireless → E_2 → Q_2						
mouse → E_3 → Q_3						
for → E_4 → Q_4						
my → E_5 → Q_5						
computer → E_6 → Q_6						

쿼리(Query), 키(Key), 값(Value) 계산 과정

(3) 내적의 크기 조정벡터들이 가까우면 벡터 내적은 커지게 된다. 내적의 값은 아주 클 수도 있고 아주 작을 수도 있어서 크기 조정을 위한 정규화 과정이 필요하다.

$$\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

쿼리(Query), 키(Key), 값(Value) 계산 과정

(4) 마스킹 연산: 디코더는 출력 시퀀스를 한 단계씩 생성하며, 각 단계에서는 이전 단계의 출력만을 참조해야 한다. 이를 위해 현재 단계 이후의 토큰들에 대한 어텐션을 방지해야 한다. 인과 마스크(Causal Mask) 또는 룩어헤드 마스크(Look-A-Head Mask)는 현재 위치 이후의 모든 위치를 마스킹하여, 신경망 모델이 미래의 정보를 참조하지 못하도록 하는 것이다

어텐션 스코어	a ↓ E_1 ↓ K_1	wireless ↓ E_2 ↓ K_2	mouse ↓ E_3 ↓ K_3	for ↓ E_4 ↓ K_4	my ↓ E_5 ↓ K_5	computer ↓ E_6 ↓ K_6
a → E_1 → Q_1	1.0	−∞	−∞	−∞	−∞	−∞
wireless → E_2 → Q_2		1.0	−∞	−∞	−∞	−∞
mouse → E_3 → Q_3		0.4	0.6	−∞	−∞	−∞
for → E_4 → Q_4				1.0	−∞	−∞
my → E_5 → Q_5					1.0	−∞
computer → E_6 → Q_6		0.2	0.3			0.5

쿼리(Query), 키(Key), 값(Value) 계산 과정

(5) 정규화: 어텐션 점수는 소프트맥스(softmax) 함수를 통해 정규화된다. 이를 통해 각 점수는 0과 1 사이의 값으로 변환되며, 합이 1이 되도록 한다.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

어텐션 스코어	a ↓ E_1 ↓ K_1	wireless ↓ E_2 ↓ K_2	mouse ↓ E_3 ↓ K_3	for ↓ E_4 ↓ K_4	my ↓ E_5 ↓ K_5	computer ↓ E_6 ↓ K_6
a → E_1 → Q_1	1.0	0	0	0	0	0
wireless → E_2 → Q_2		1.0	0	0	0	0
mouse → E_3 → Q_3		0.4	0.6	0	0	0
for → E_4 → Q_4				1.0	0	0
my → E_5 → Q_5					1.0	0
computer → E_6 → Q_6		0.2	0.3			0.5

쿼리(Query), 키(Key), 값(Value) 계산 과정

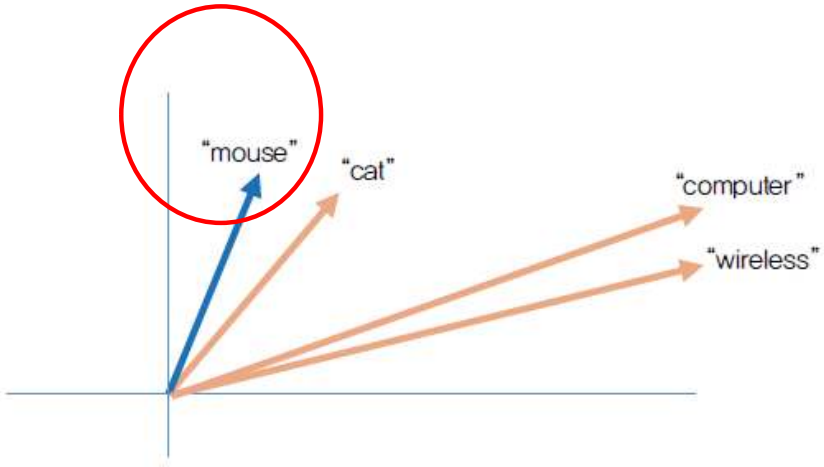
(6) 어텐션 스코어를 사용하여 값 벡터의 가중합을 구한다. 이렇게 얻어진 값이 어텐션 층의 출력이다.

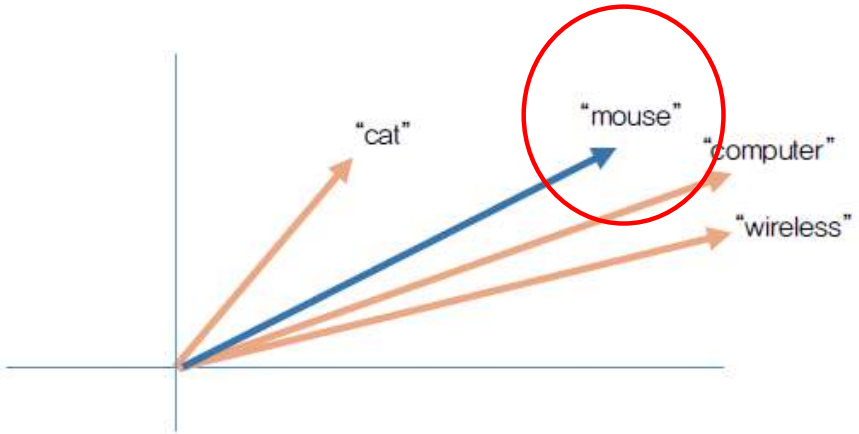
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

어텐션 스코어	a ↓ E_1 ↓ V_1	wireless ↓ E_2 ↓ V_2	mouse ↓ E_3 ↓ V_3	for ↓ E_4 ↓ V_4	my ↓ E_5 ↓ V_5	computer ↓ E_6 ↓ V_6
a → E_1 → Q_1	$1.0V_1$					
wireless → E_2 → Q_2		$1.0V_2$				
mouse → E_3 → Q_3		$0.4V_2$	$0.6V_3$			
for → E_4 → Q_4				$1.0V_4$		
my → E_5 → Q_5					$1.0V_5$	
computer → E_6 → Q_6		$0.2V_2$	$0.3V_3$			$0.5V_6$

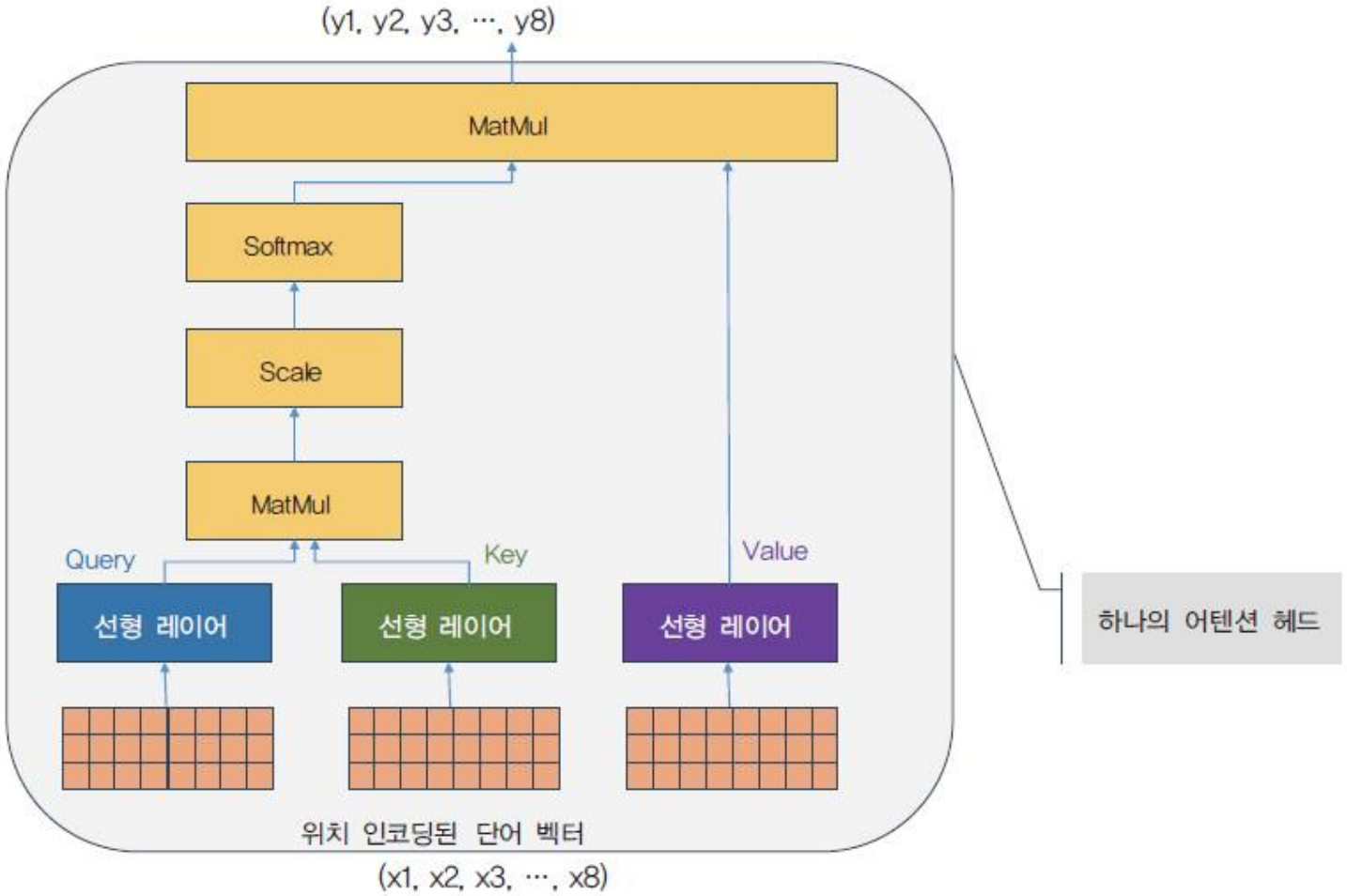
어텐션 값의 해석

- 어텐션 값은 현재 단어의 엠베딩 벡터가 주변 단어의 영향을 받아서 변경된 엠베딩 벡터로 생각할 수 있다.



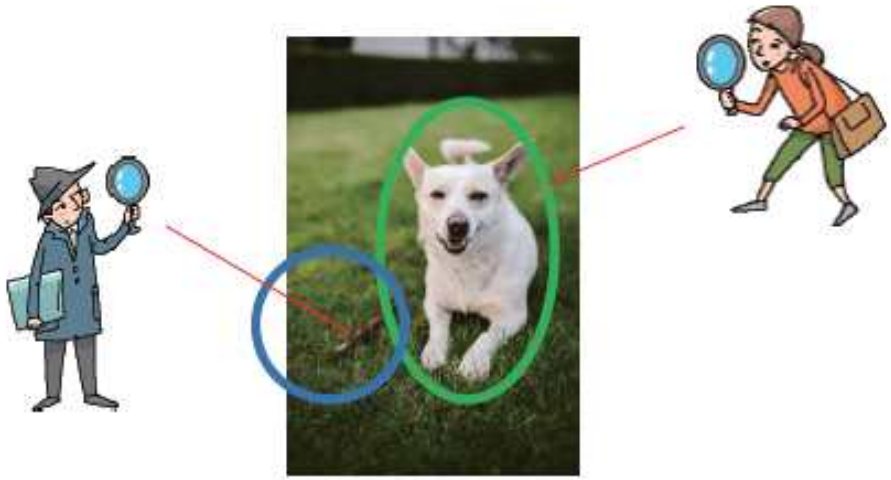


단일 어텐션 헤드


(y1, y2, y3, ···, y8)
MatMul
Softmax
Scale
MatMul
Query
Key
Value
선형 레이어
선형 레이어
선형 레이어
위치 인코딩된 단어 벡터
(x1, x2, x3, ···, x8)
하나의 어텐션 헤드

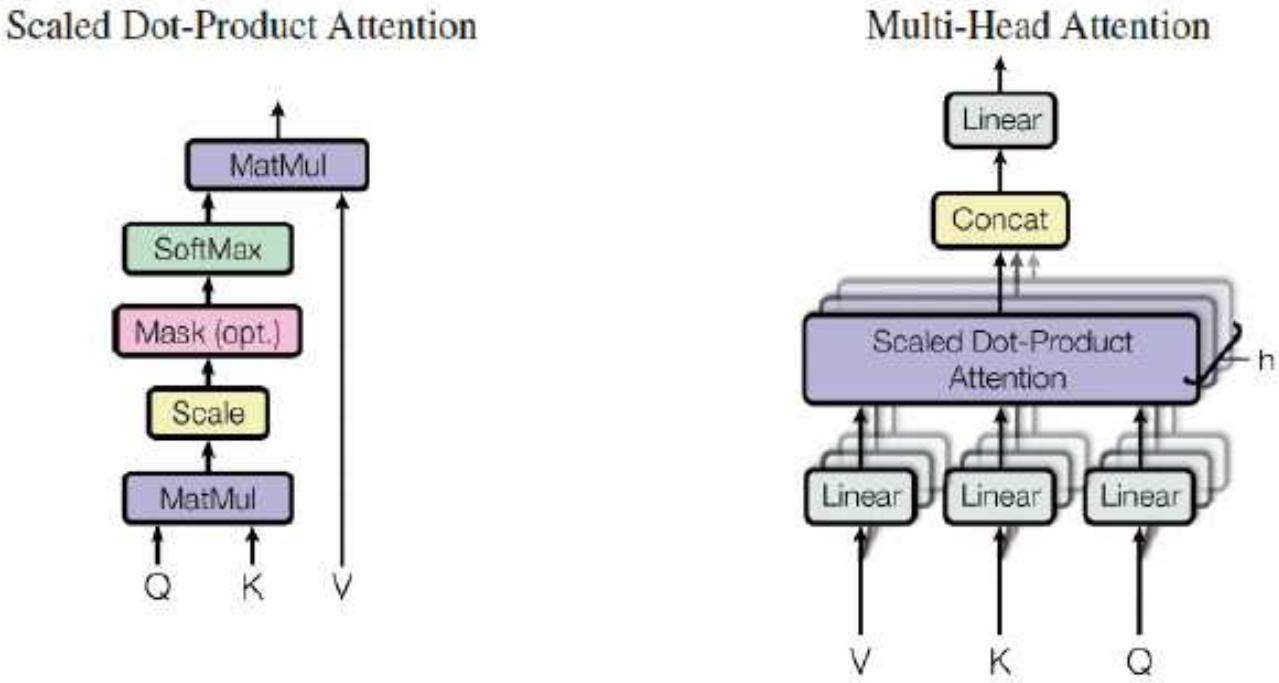
멀티 헤드 어텐션 헤드가 되면?

- 인간이 문장의 단어를 볼 때, 하나 이상의 단어에 어텐션을 준다. 만약 mouse라는 단어를 볼때, 처음에는 wireless 단어에 주의하게 되지만, 다음 단계로는 prefer 단어도 볼 수 있다.



멀티 헤드 어텐션 헤드

- 멀티 헤드 어텐션은 어텐션 메커니즘을 여러 번 병렬로 수행하는 것이다. 각각의 “헤드”는 어텐션을 독립적으로 수행하고, 그 결과를 결합하여 최종 출력을 생성한다.

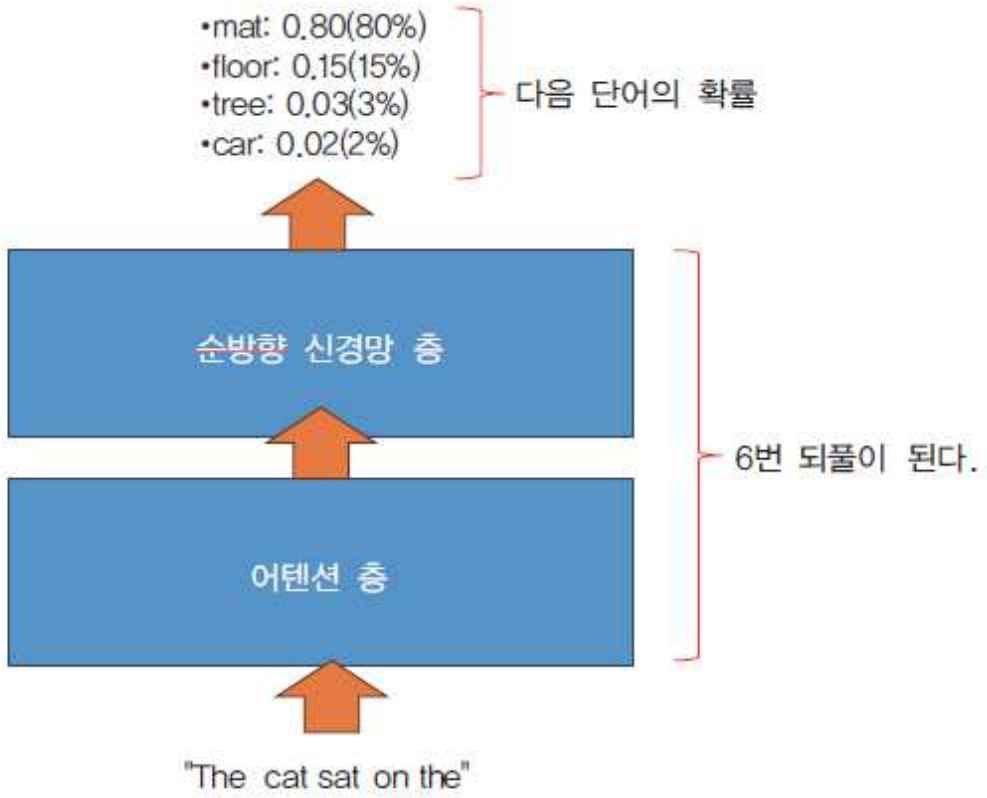


피드_포워드 신경망 층

- 어텐션 벡터 자체는 본질적으로 모두 일련의 행렬 곱셈이므로 선형 변환이다.

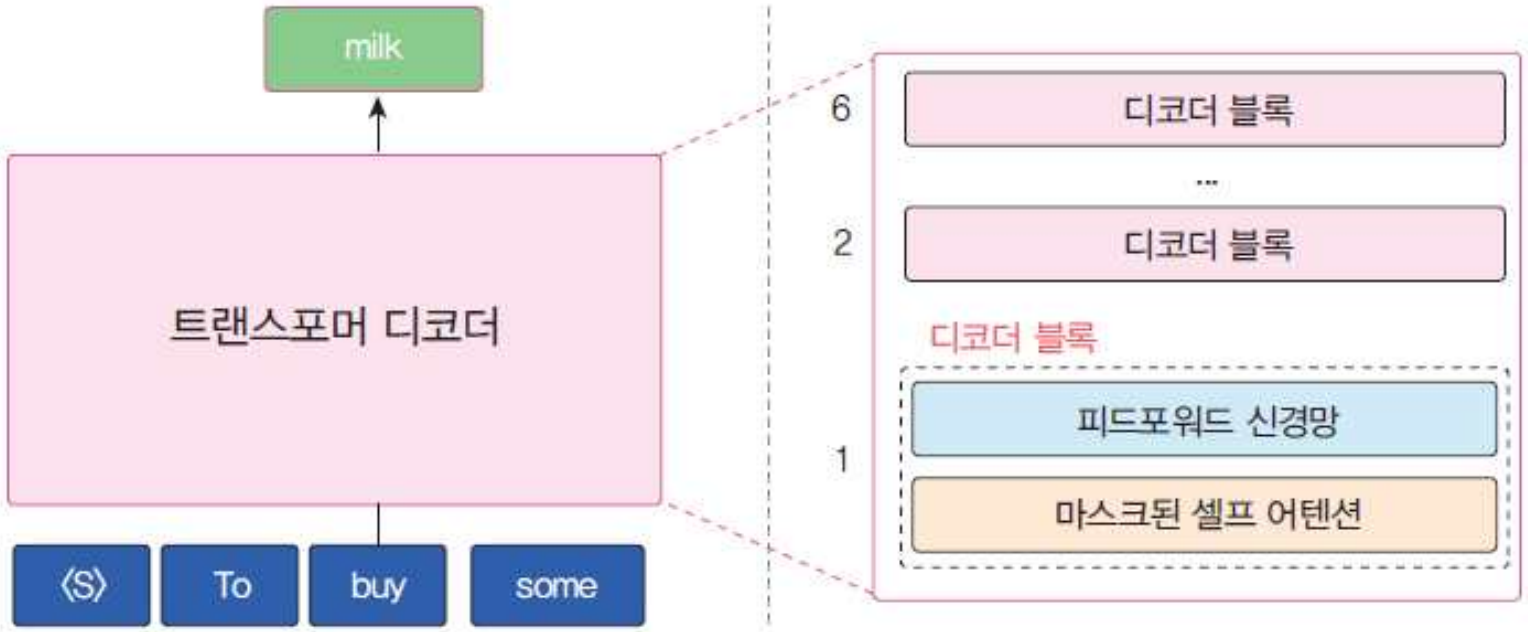
- 비선형 활성화 함수 없이는 복잡한 함수를 근사화할 수 없다. 따라서 어텐션 메커니즘 바로 뒤에 완전히 연결된 순방향 신경망(FFN)이 존재한다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)\,W_2 + b_2$$



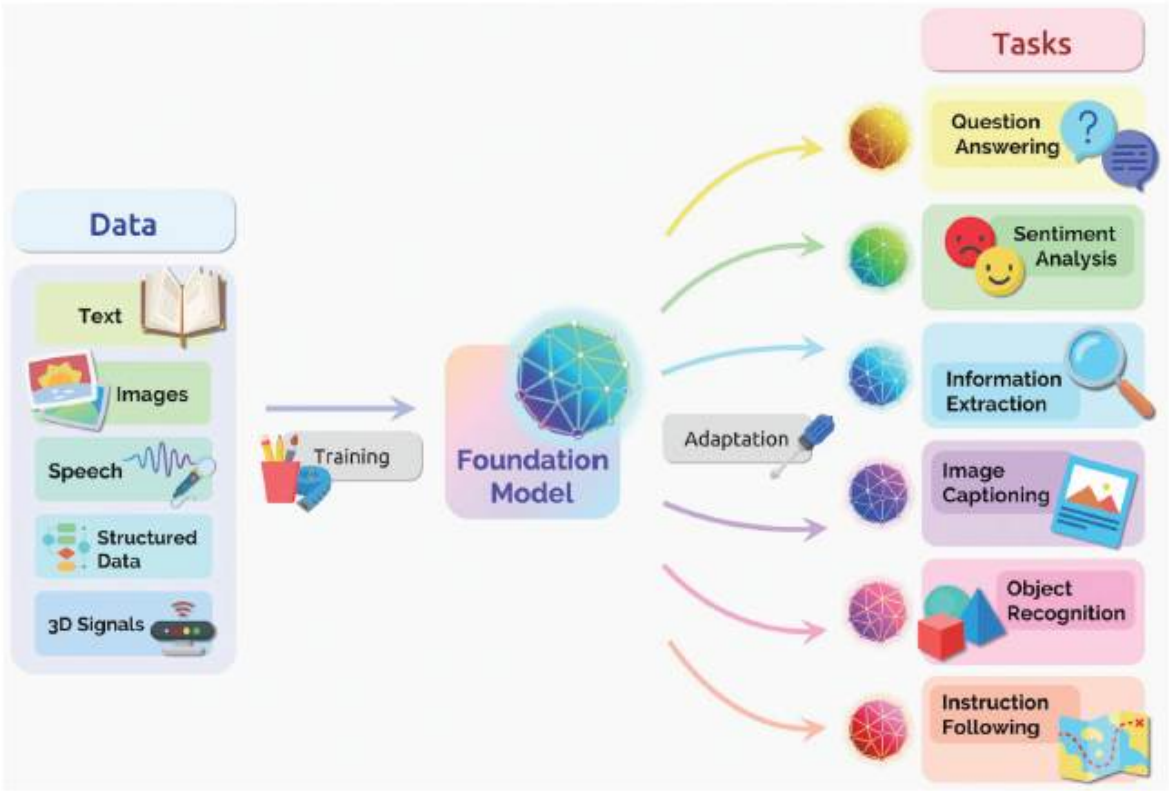
동일한 층을 6번 반복

- 이러한 반복적인 구조는 모델이 입력 시퀀스의 복잡한 관계를 학습하고 더 깊이 있는 표현을 추출하는 데 도움을 준다



트랜스포머 응용 분야

- 구글의 2017년 논문에서 처음 설명된 트랜스포머는 현재까지 발명된 최신 모델이자 가장 강력한 모델 클래스 중 하나이다



(출처: 엔비디아 홈페이지)

GPT 모델

- GPT(Generative Pre-trained Transformer)는 트랜스포머 아키텍처를 기반으로 한 자연어 처리 모델이다.
- 디코더 아키텍처: GPT는 트랜스포머의 디코더 부분만을 사용하여, 주어진 입력으로부터 다음 단어를 예측하는 언어 모델이다. 인코더 부분이 없고, 디코더만으로 구성되어 있다.
- 언어 모델링 목표: GPT는 주로 언어 모델링, 즉 다음 단어 예측에 초점을 맞춘다. 문장의 앞부분을 기반으로 다음 단어를 예측하는 방식으로 학습된다.
- 사전 학습과 미세 조정: GPT는 대규모 텍스트 데이터로 사전 학습(pre-training)을 거친 후, 특정 작업에 대해 미세 조정(fine-tuning)된다. 이 과정을 통해 다양한 NLP 작업에 적용될 수 있다.
- 자기 회귀(Autoregressive): GPT는 자기 회귀 방식으로, 이전 단어들을 기반으로 다음 단어를 한 번에 하나씩 생성한다.

GPT 모델의 미세조정

- 사전 학습 후 미세 조정(Fine-Tuning)은 사전 학습된 언어 모델을 특정 작업에 맞게 조정하는 과정이다.

① 특정 작업 데이터 수집 및 전처리: 미세 조정을 위해서는 특정 작업에 대한 데이터셋이 필요하다. 전처리 과정은 사전 학습 때와 유사하지만, 특정 작업에 맞게 조정된다.

② 사전 학습된 모델 로드: 사전 학습된 GPT 모델을 로드한다. 이 모델은 이미 대규모 텍스트 데이터에서 언어의 일반적인 패턴과 구조를 학습한 상태이다.

③ 미세 조정: 특정 작업에 맞는 손실 함수를 정의할 수 있다. 예를 들어, 분류 작업에서는 교차 엔트로피 손실을 사용할 수 있다.

④ 평가 및 테스트: 미세 조정된 모델을 검증 데이터셋을 사용하여 평가한다.

실습 예제: *picoGPT*

- 이 코드는 Jay Mody의 2023년 블로그 “GPT in 60 Lines of NumPy”에서 가져온 것이다.

- 이미 GPT-2에서 학습된 가중치 파일을 가져와서 디코더만을 구현하였다.
- 주석을 제외하면 약 40줄 정도의 코드 크기로 구현하였다.
- 학습 코드는 포함되지 않는다.
- 디코더만 구현하였다.

라이브러리 설치

```
d:\picoGPT-main>pip install numpy enter
d:\picoGPT-main>pip install regex enter
d:\picoGPT-main>pip install requests enter
d:\picoGPT-main>pip install tqdm enter
d:\picoGPT-main>pip install fire enter
d:\picoGPT-main>pip install tensorflow enter
```

gpt2.py 파일을 실행

```
d:\picoGPT-main>python gpt2.py "Alan Turing theorized that computers would one
day become" enter

 the most powerful machines on the planet.

The computer is a machine that can perform complex calculations, and it can
perform these calculations in a way that is very similar to the human brain.
```

^ㅇ^ #1

```python
import numpy as np

# GELU 활성화 함수
def gelu(x):
    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

# 소프트맥스 함수
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# 레이어 정규화 함수
def layer_norm(x, g, b, eps: float = 1e-5):
    mean = np.mean(x, axis=-1, keepdims=True)
    variance = np.var(x, axis=-1, keepdims=True)
    return g * (x - mean) / np.sqrt(variance + eps) + b

# 선형 변환 함수
def linear(x, w, b):
    return x @ w + b

# 피드포워드 네트워크 (FFN) 함수
def ffn(x, c_fc, c_proj):
    return linear(gelu(linear(x, **c_fc)), **c_proj)
```

^.^ #2

```python
# 어텐션 함수
def attention(q, k, v, mask):
    return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v

# 멀티헤드 어텐션 (MHA) 함수
def mha(x, c_attn, c_proj, n_head):
    x = linear(x, **c_attn)  # 선형 변환
    qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), np.split(x, 3,
                    axis=-1)))  # Q, K, V 분할
    causal_mask = (1 - np.tri(x.shape[0], dtype=x.dtype)) * -1e10  # 인과 마스크 생성
    out_heads = [attention(q, k, v, causal_mask) for q, k, v in zip(*qkv_heads)]
                                                                    # 어텐션 계산
    x = linear(np.hstack(out_heads), **c_proj)  # 출력 결합 후 선형 변환
    return x
```

^_^ #3

```
def transformer_block(x, mlp, attn, ln_1, ln_2, n_head):
    x = x + mha(layer_norm(x, **ln_1), **attn, n_head=n_head)  # MHA 및 잔차 연결
    x = x + ffn(layer_norm(x, **ln_2), **mlp)  # FFN 및 잔차 연결
    return x

# GPT-2 모델 함수
def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):
    x = wte[inputs] + wpe[range(len(inputs))]  # 토큰 임베딩 및 위치 임베딩 합산
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)  # 각 트랜스포머 블록 적용
    return layer_norm(x, **ln_f) @ wte.T  # 출력 정규화 후 토큰 임베딩의 전치와 행렬 곱

# 텍스트 생성 함수
def generate(inputs, params, n_head, n_tokens_to_generate):
    from tqdm import tqdm
    for _ in tqdm(range(n_tokens_to_generate), "generating"):
        logits = gpt2(inputs, **params, n_head=n_head)  # GPT-2 모델의 로그잇 계산
        next_id = np.argmax(logits[-1])  # 다음 토큰 선택
        inputs.append(int(next_id))  # 선택한 토큰을 입력에 추가
    return inputs[len(inputs) - n_tokens_to_generate :]
```

^.^ #4

```
# 메인 함수
def main(prompt: str, n_tokens_to_generate: int = 40, model_size: str = "124M",
models_dir: str = "models"):
    from utils import load_encoder_hparams_and_params
    encoder, hparams, params = load_encoder_hparams_and_params(model_size,
                                models_dir)  # 인코더 및 하이퍼파라미터 로드
    input_ids = encoder.encode(prompt)  # 프롬프트 인코딩
    assert len(input_ids) + n_tokens_to_generate < hparams["n_ctx"]  # 입력  길이
                                                                      확인
    output_ids = generate(input_ids, params, hparams["n_head"], n_tokens_to_
                          generate)  # 텍스트 생성
    output_text = encoder.decode(output_ids)  # 출력 디코딩
    return output_text

if __name__ == "__main__":
    import fire
    fire.Fire(main)  # 명령어 라인 인터페이스 실행
```

Summary

- 트랜스포머(Transformer) 모델은 2017년 Vaswani et al.의 논문 “Attention is All You Need”에서 소개된 자연어 처리 모델이다. 트랜스포머는 RNN이나 LSTM과 달리 순차적 처리가 아닌 병렬 처리를 통해 더 빠르고 효율적으로 데이터를 처리할 수 있다.

- 인코더-디코더 구조: 트랜스포머는 인코더와 디코더로 구성되며, 인코더는 입력 시퀀스를 인코딩하고, 디코더는 인코딩된 정보를 기반으로 출력 시퀀스를 생성한다.

- 셀프 어텐션(Self-Attention): 각 단어의 표현을 다른 단어들과의 관계를 고려하여 계산하는 메커니즘으로, 병렬 처리가 가능하게 한다.

- 위치 인코딩(Positional Encoding): 입력 데이터의 순서 정보를 포함시키기 위해 사용된다.

Q & A