Transformer (LLM)

배경

2010년대 초반 이후 딥러닝은 자연어 처리(NLP), 특히 **기계 번역(Neural Machine Translation, NMT)** 분야에서 빠르게 발전해왔다. 그 중심에는 **순환 신경망(Recurrent Neural Network, RNN)** 과 그 변형인 **LSTM(Long Short-Term Memory), GRU(Gated Recurrent Unit)** 같은 구조가 있었다. 이들은 입력 시퀀스를 순차적으로 처리하면서, 단어 간의 시간적 의존성을 학습할 수 있다는 장점 덕분에 번역, 언어 모델링, 음성 인식 등 다양한 응용에서 사실상 표준 접근법이 되었다.

그러나 RNN 구조에는 근본적인 한계가 있었다. 모델이 한 단어씩 순차적으로 입력을 처리해야 했기 때문에 **병렬화가 어렵고**, 시퀀스가 길어질수록 학습이 **매우 느려지는 문제**가 발생했다. 또한 장기 의존성(long-range dependency)을 학습하는 데 있어 기울기 소실(vanishing gradient) 문제가 남아 있어, 문장 속 멀리 떨어진 단어 사이의 관계를 효과적으로 반영하기 어려웠다.

이 한계를 극복하려는 시도로, 연구자들은 **합성곱 신경망(CNN)** 을 기반으로 한 시퀀스 변환 모델을 탐구했다. 대표적으로 **ByteNet**과 **ConvS2S(Convolutional Sequence-to-Sequence)** 모델은 CNN 구조를 활용하여 시퀀스 전체를 병렬적으로 처리할 수 있도록 설계되었다. 이러한 접근은 병렬화를 가능하게 했지만, 여전히 중요한 문제가 남아 있었다. CNN은 국소적 연산(local receptive field)에 기반하기 때문에 **멀리 떨어진 단어 간 관계를 학습하려면 여러 층을 쌓아야만** 했다. 즉, 장기 의존성을 포착하기 위해서 네트워크가 깊어져야 하고, 이는 학습과 계산 비용을 크게 증가시켰다.

이와 동시에, 연구자들은 **Attention 메커니즘**의 가능성에 주목하기 시작했다. Attention은 입력 시퀀스의 특정 위치가 출력 시퀀스의 특정 위치와 얼마나 관련 있는지를 학습하여, 멀리 떨어진 단어 간의 관계를 직접적으로 모델링할 수 있게 했다. Bahdanau et al. (2014)의 **“Neural Machine Translation by Jointly Learning to Align and Translate”** 논문은 RNN 인코더-디코더 구조에 Attention을 접목시켜 번역 성능을 크게 끌어올렸고, 이후 모든 주요 시퀀스 모델의 핵심 요소로 자리 잡았다.

그러나 여기에도 중요한 제약이 있었다. 기존의 Attention은 보조 모듈로만 쓰였고, 기본 구조는 여전히 **순차적 RNN**이나 **계층적 CNN**에 의존했다. 즉, Attention의 장점을 충분히 발휘하지 못하고 있었던 것이다.

이러한 맥락에서 "굳이 RNN이나 CNN이 필요할까? Attention만으로 시퀀스를 변환할 수 있지 않을까?"라는 질문을 던졌으며,
그 결과 제안된 것이 바로 Transformer이다. Transformer는 RNN이나 CNN을 완전히 배제하고, **Self-Attention만을 핵심 연산으로 채택**하였다. 이로써 모델은 모든 입력 토큰을 한 번에 서로 연결할 수 있고, 병렬화가 가능해졌으며, 장기 의존성 학습도 훨씬 용이해졌다.

- **RNN 기반 모델**: 순차적, 장기 의존성 문제, 학습 느림
- **CNN 기반 모델(ConvS2S)**: 병렬화 가능, 하지만 긴 문맥은 깊은 계층 필요

- **Transformer**:
  - 완전 병렬화 가능
  - 긴 문맥을 한 단계에서 연결 가능
  - 계산 효율성 개선

RNN vs Attention

RNN 기반 모델은 입력을 **순차적으로 한 단어씩 처리**한다. 이 때문에 병렬 연산이 어렵고, 문장이 길어질수록 **장기 의존성(long-term dependency)** 학습이 힘들어진다.

Attention 기반 모델은 모든 단어가 **동시에 서로를 바라보며 상호 연관성을 계산**한다. 따라서 **병렬 처리**가 가능하고, 멀리 떨어진 단어 간의 관계도 쉽게 포착할 수 있다.

Attention

Attention은 **입력 중 중요한 부분에 더 집중하도록 가중치를 주는 메커니즘**이다.

예를 들어, 컴퓨터가 아래 이미지를 보고 "A cat sitting on a chair"라는 문장을 생성해야 한다고 가정해 보면,

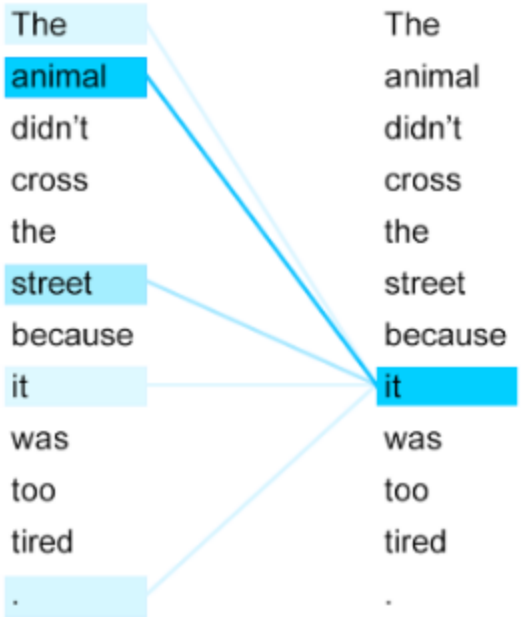
모델이 문장의 첫 단어 "cat"을 생성할 때는 이미지 전체보다 **고양이가 있는 영역**에 집중해야 한다.

다음 단어 "chair"를 생성할 때는 **의자 부분**을 더 주목해야 한다.

즉, Attention은 이미지의 모든 픽셀(혹은 영역) 정보를 다 참고하되, **출력 단어마다 중요한 부분에 가중치를 크게 주어 선택적으로 집중**하는 메커니즘이다.

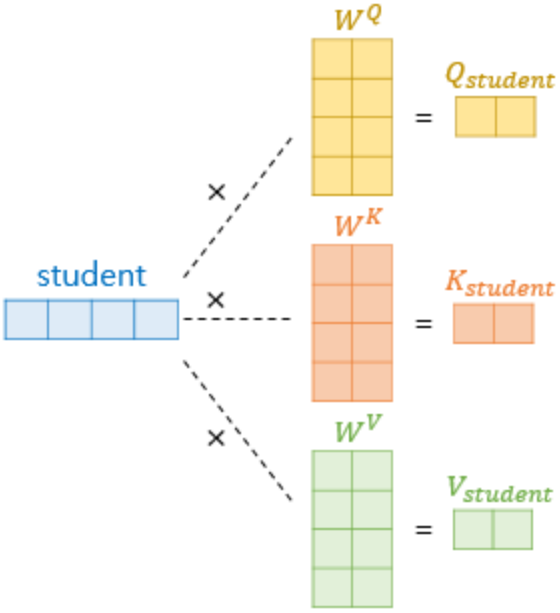
Self Attention

Transformer의 핵심 메커니즘 첫 번째는 **Self-Attention**이다. 말 그대로, Attention을 자기 자신한테 취한다는 의미이다. 즉, LLM에서는 하나의 문장 안에서 각 단어가 자기 자신을 포함한 모든 단어들과 상호작용하도록 하는 메커니즘이다. 이는 **문맥 속에서 단어 간 연관성을 모델이 스스로 학습할 수 있도록 설계된 방식**이다.



예를 들어 "The animal didn't cross the street because it was too tired" 라는 문장을 사람이 "it"이 "animal"을 가리킨다는 사실을 쉽게 이해하는 것과 달리, 기계는 이를 단순 순차 처리만으로 파악하기 어렵다. self-attention은 각 단어가 다른 단어와 맺는 관계를 수치적으로 표현하여, 이러한 문맥적 의존성을 모델이 직접 학습할 수 있도록 돕는다.

Self-Attention의 계산 과정에는 세 가지 핵심 요소가 존재한다. 바로 Query(Q), Key(K), Value(V)이다.



이들은 모두 같은 입력 임베딩에서 출발하지만, 서로 다른 학습 가중치 행렬 W^Q, W^K, W^V를 통해 서로

다른 공간으로 투사된다. 따라서 **입력은 같지만** 최종적으로 얻게되는 Q,K,V는 서로 다른 역할을 수행한다.

Query는 "무엇을 찾고 있는가"를 나타내고, **Key**는 "내가 가진 정보가 무엇인가"를 설명하며, **Value**는 "실제 전달할 정보"를 담는다. 이 세 요소는 단어 간 관계를 측정하고 가중합을 통해 새로운 표현을 만드는 데 사용된다.

예를 들면, 도서관에서, 어떤 학생이 "역사 관련 책을 찾고 싶어요"라고 질문하는 것은 **Query(Q)** 가 된다.

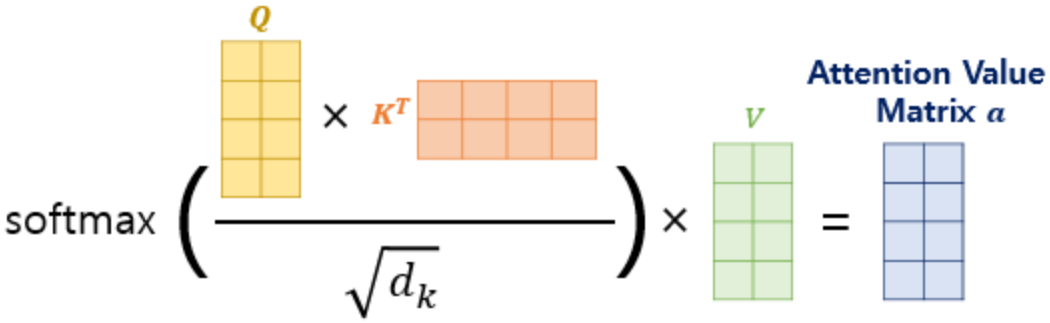
도서관 직원들이 각 서가를 대표해서 "이 서가에는 과학, 저 서가에는 역사, 또 다른 서가에는 문학 책이 있습니다"라고 말하는 것은 **Key(K)** 가 된다.

실제로 그 서가에 꽂혀 있는 책들의 구체적인 내용은 **Value(V)** 가 된다.

이때 학생의 Query(역사 관련 책 요청)와 각 서가의 Key(서가 주제)가 얼마나 잘 맞는지를 비교하여 **Attention Score**를 계산한다. 역사 서가의 **Key**가 가장 높은 점수를 얻게 되고, 따라서 그 서가의 Value(역사 책들)의 정보가 가장 많이 반영된다.

이를 구하는 공식은 다음과 같다.

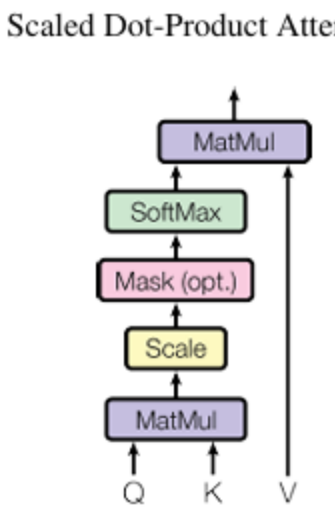
$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



우선 처음에 Q와 K를 내적해준다. 내적해주는 이유는 둘 사이의 연관성을 계산하기 위함이다. 이 내적된 값을 **Attention Socre** 라고 한다.

하지만, 만약 Q와 K의 차원이 커지면 내적 값, Attention Score도 커지게 되어 모델의 학습에 어려움이 생긴다. 이 문제를 해결하기 위해 차원 d_k의 루트만큼을 나누어 주는 스케일링 작업을 진행한다.(여기까

지의 과정을 Scaled dot-product Attention이라고 한다.)



그 다음으로 값들을 정규화시켜주기 위해 Softmax 활성 함수를 거치고, 마지막으로 보정을 위해 지금까지 계산된 Score행렬과 Value 행렬을 내적 해주면 최종적인 Attention 행렬을 얻을 수 있게된다.

예를 들어 "I am a student" 라는 문장이 있다고 가정해보자. 여기서 각 단어를 Attention 메커니즘을 사용하기 위해 임베딩을 해주어야 한다.

$$\mathrm{Q_{I,origial}} = [1, 1, 1, 1] \bullet \mathrm{W^Q} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \mathrm{Q_I} = [1, 1, 1, 1]$$

$$\mathrm{K_{I,origial}} = [1, 1, 1, 1] \bullet \mathrm{W^K} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \mathrm{K_I} = [1, 0, 1, 1]$$

$$\mathrm{V_{I,origial}} = [1, 1, 1, 1] \bullet \mathrm{W^V} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \mathrm{V_I} = [1, 1, 1, 0]$$

단어 "I"의 임베딩이 $[1,1,1,1]$이라 했을 때 처음 I의 처음 Query, Key, Value를 각각 $O_{I,original}, K_{I,original}, V_{I,original}$라고 한다. 이 값들은 Self Attention 메커니즘에서 같아야 되기 때문에 모두 $[1,1,1,1]$로 동일하다. 이때 각각 학습된 weight 값이 W^Q, W^K, W^V라고 할 때, 이를 original 값들과 dot production을 해주면 최종적인 값 Q, K, V가 도출된다.

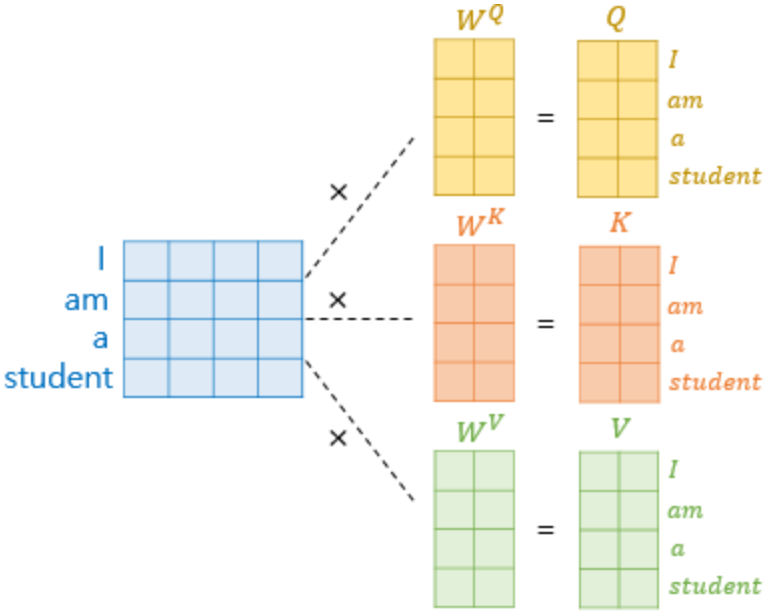
$$\frac{QK^T}{\sqrt{d_k}} = \frac{3}{\sqrt{4}} = 1.5$$

이 Q,K,V 값을 이용해 위의 Attention Score를 구해주면 1.5라는 값이 나온다. 행렬 Q,K는 dot production을 해주고, 여기서 행렬 Q,K,V의 차원은 4이므로 루트 4를 나누어 준 것이다.

이 과정을 "I" 뿐만 아니라 모든 단어간에 Self Attention을 해주면 아래와 같은 결과가 나온다.

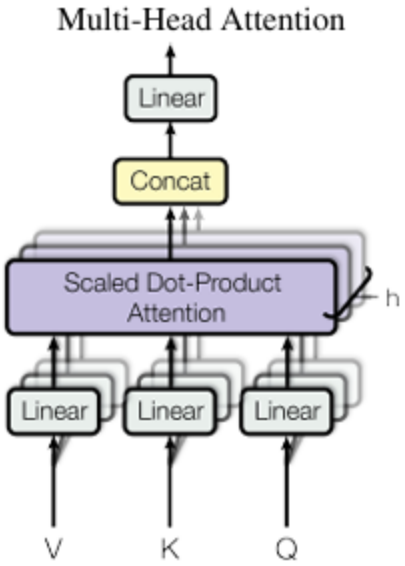
	I	am	a	student
I	1.5	0.5	0.3	1.0
am	0.5	1.5	0.4	0.2
a	0.3	0.4	1.5	0.1
student	1.0	0.2	0.1	1.5

가운데 노란색 부분은 자기 자신에 대한 Attention이므로 당연히 값이 제일 크고, 양쪽 초록색 부분을 보면 이 역시 점수가 높다. 따라서 단어 "I"와 "student" 사이의 상관 관계가 있는 것을 확인할 수 있다.

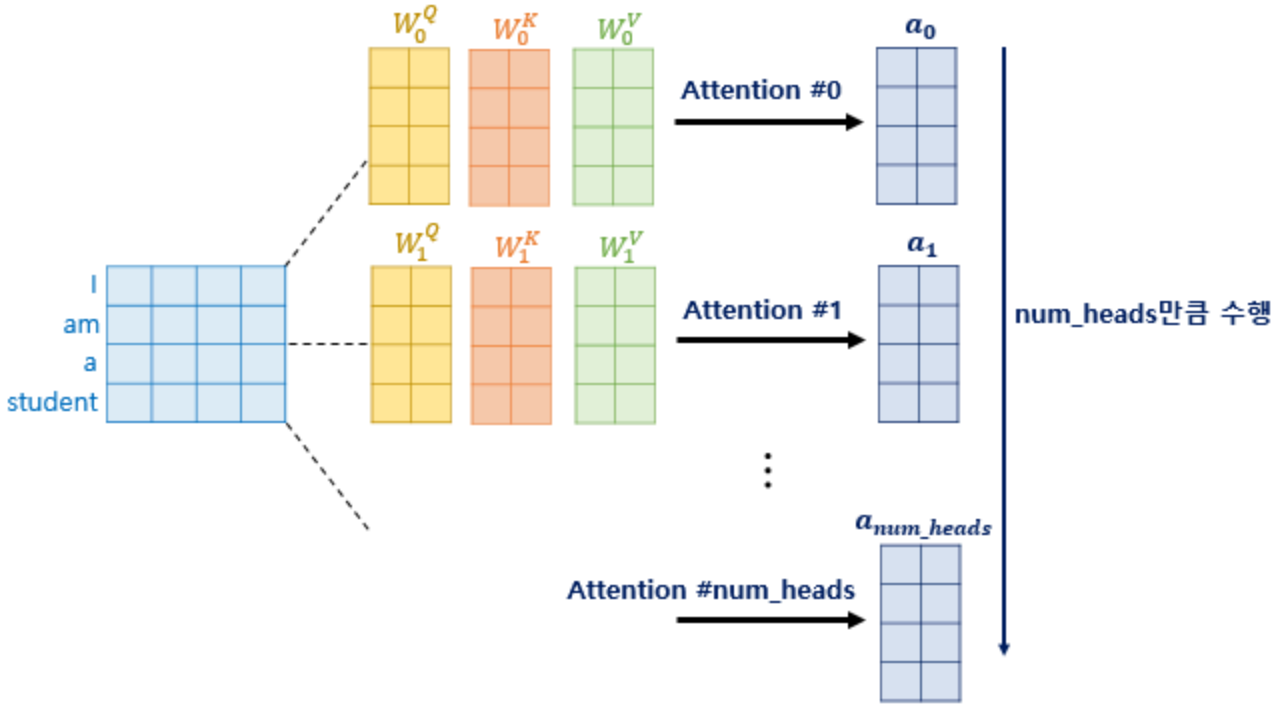


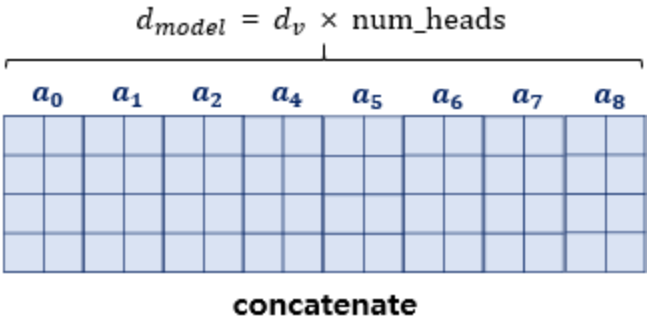
위 연산들은 실제로 위 그림과 같이 병렬 처리해 계산을 하여, 연산 속도를 높인다.

Multi-Head Attention



한 개의 Self-Attention만으로는 문장의 여러 관계를 동시에 보기 어렵다. 그래서 Transformer는 **여러 개의 Attention Head**를 병렬적으로 두어 서로 다른 "관점"에서 토큰 간 관계를 학습한다.

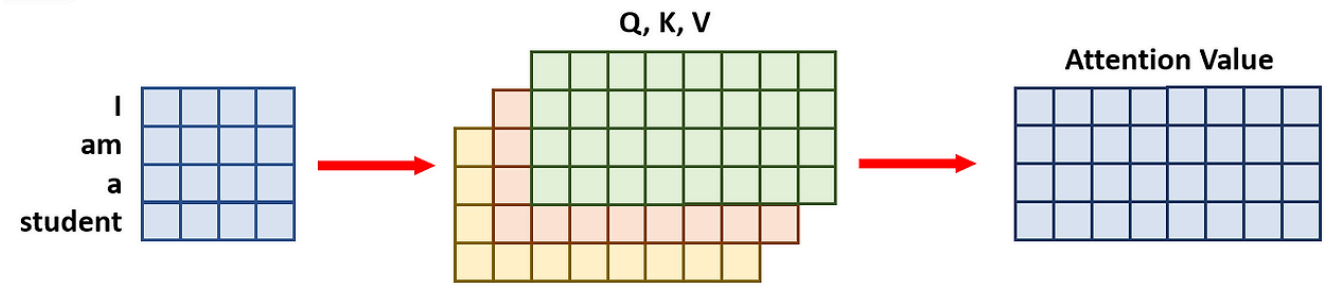




위 그림에서 볼 수 있듯이 head의 수만큼 Attention을 각각 병렬로 나누어 계산을 한다. 도출된

Attention Value들은 마지막에 concatenate를 통해 하나로 합쳐진다. 이렇게 하면 Attention을 한 번 사용할 때와 같은 크기의 결과가 도출된다.

예를 들어 $[4 \times 4]$ 크기의 문장 임베딩 벡터와 $[4 \times 8]$의 Query, Key, Value가 있을 때, **Attention 메커니즘**은 $[4\times 4] * [4\times 8] = [4 \times 8]$의 Attention Value가 한 번에 도출된다.



반면, Multi-head Attention 메커니즘은 아래와 같은 구조로 계산이 된다.

head의 수를 4개라고 가정하면, head의 수가 4개이므로 각 연산과정의 1/4만큼만 필요하다는 이야기이다. 때문에 위에서 크기가 $[4 \times 8]$이었던 Query, Key, Value를 4등분하여 $[4 \times 2]$로 만든다. 이렇게 되면 자연스럽게 각 Attention Value는 $[4 \times 2]$가 된다.

이 Attention Value들을 마지막에 concatenate를 시켜주면 일반적인 Attention 결과와 동일하게 된다.

Multi-Head Attention은 head를 늘려도 각 head의 차원을 줄여 계산하므로 총 연산량은 단일 Attention과 유사하다.

하지만, 성능 차원에서는 Multi-head Attention이 더 좋다. 최종 출력의 차원은 단일 Attention과 동일하지만, 내부적으로는 서로 다른 부분 공간에서 병렬적으로 다양한 관계를 학습할 수 있기 때문에 성능이 크게 향상될 것으로 생각되고, GPU 병렬화로 실제 연산 속도 또한 빠른 편이다.(RNN 대비 훨씬 빠른 속도)

Position-wise Feed-Forward Networks

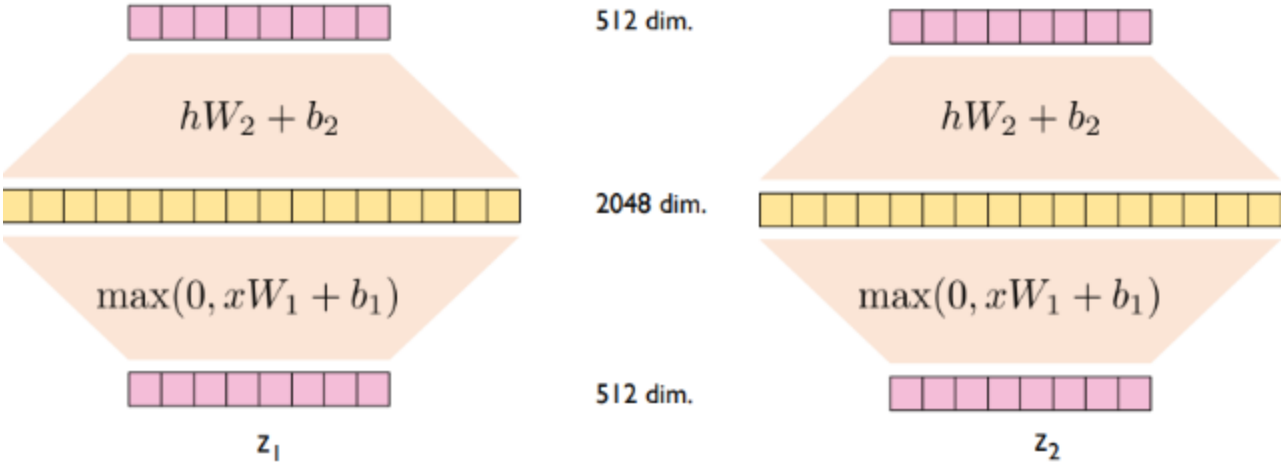
Transformer의 각 인코더와 디코더 레이어에는 공통적으로 Self-Attention 이후에 Feed-Forward Network가 따라온다.

이 Feed-Forward Network는 **포지션별 완전연결 신경망(Position-wise Fully Connected Network)** 이라고도 불리며, 다음과 같은 구조를 갖는다.

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

- Linear Transformation : 입력 벡터 x를 더 높은 차원(보통 2048차원)으로 확장한다.
- ReLU 활성화 함수 : 비선형성을 부여하여 단순 선형 변환 이상의 복잡한 패턴을 학습할 수 있게 한다.

- 두 번째 선형 변환 : 다시 모델 차원(예: 512차원)으로 축소하여 원래 차원으로 되돌린다.



여기서 중요한 점은 모든 위치(position)에 대해 동일한 FFN이 독립적으로 적용된다는 것이다. 즉, 시퀀스의 각 단어 벡터에 대해 같은 FFN이 개별적으로 수행된다 그래서 'positioin-wise'라는 표현을 사용한다.

사용 이유

- 표현력 확장 : Self-Attention은 단어 간의 관계를 잘 포착하지만, 각 위치의 벡터 자체를 깊게 변환하는 능력은 제한적이다. FFN은 이러한 한계를 보완하여 더 풍부하고 정교한 표현을 학습한다.
- 비선형 변환 제공 : Self-Attention은 기본적으로 선형 변환과 가중치 조합이므로 복잡한 비선형 패턴을 학습하기 어렵다. ReLU를 포함한 FFN은 네트워크에 비선형성을 추가하여 더 복잡한 함수 근사를 가능하게 한다.
- 각 위치별 독립 처리(Position-wise Transformation) : Attention은 여러 위치 간 상호작용을 학습하는 반면, FFN은 각 위치의 벡터를 독립적으로 강화한다. 이렇게 전역적(Context) 관계 학습과 지역적(Feature) 강화 학습이 결합되어 Transformer의 성능을 끌어올린다.
- 계산 효율성(Efficiency) : FFN은 단순히 두 번의 선형 변호나과 하나의 비선형 활성화만 포함하기 때문에 계산이 매우 빠르고 병렬화하기도 쉽다. 따라서 대규모 모델에서도 확장성이 좋다.

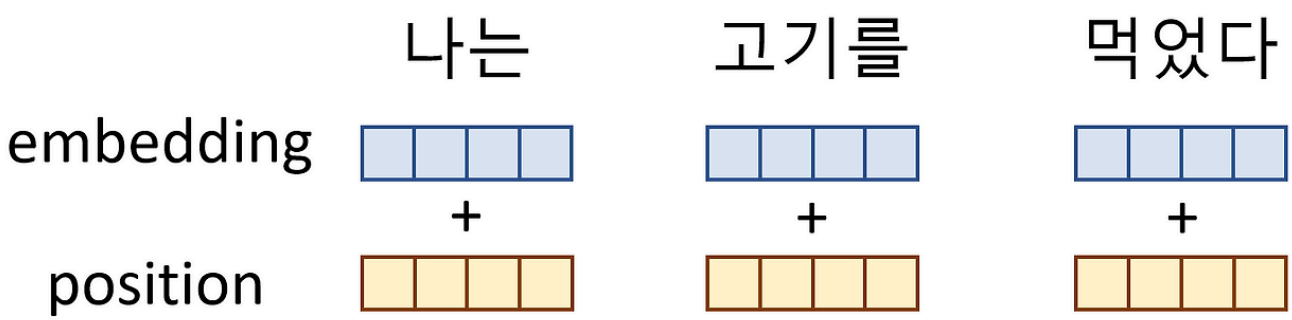
Positional Encoding

Transformer의 장점 중 하나인 **입력을 한 번에 병렬 처리할 수 있다는 점**으로 인해 새로운 문제가 발생한다. RNN 계열 모델은 순차적으로 단어를 처리하므로 문장 내 단어의 순서가 자연스럽게 반영된다. 반면 Transformer는 문장을 병렬로 처리하기 때문에, 단어 간의 순서 정보가 학습 과정에서 손실된다. 이는 언어 처리에서 치명적인 문제를 일으킬 수 있다.

예를 들어 "나는 고기를 먹었다"와 "고기는 나를 먹었다." 두 문장의 단어 구성은 같지만, 단어 순서가 바뀌면 의미가 전혀 달라진다. 따라서 Transformer에는 단어의 순서를 알려주는 별도의 장치가 필요하다. 이 장치가 바로 **Positional Encoding**이다.

Positional Encoding의 기본 아이디어는 간단하다. 각 단어 임베딩 벡터에 단어가 문장에서 차지하는 위치 정보를 더해 주는 것이다. 이렇게 하면 모델은 단어의 의미 표현뿐만 아니라 그 단어가 문장에서 어

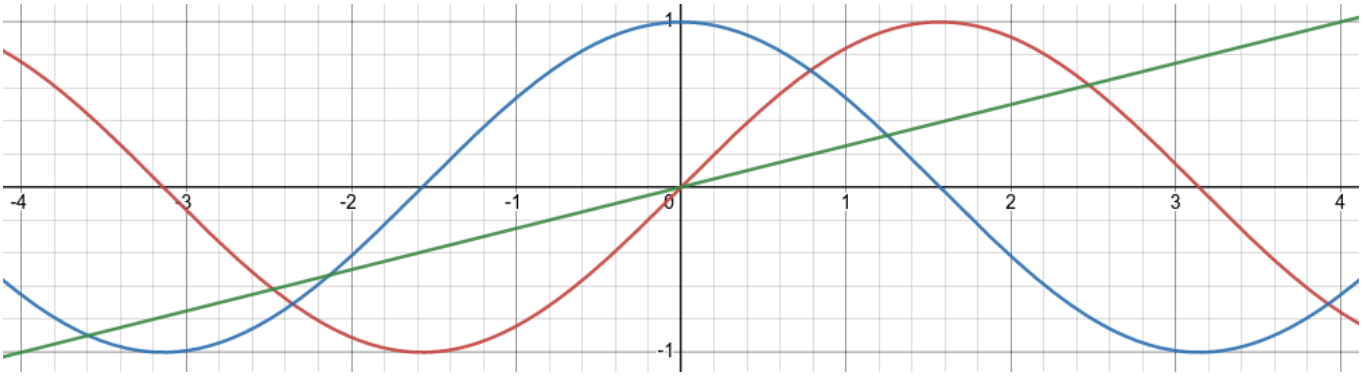
느 위치에 있는지도 동시에 학습할 수 있다.



이때, Positional Encoding에는 두 가지 중요한 조건이 있다.

- 값의 크기가 지나치게 커지지 않아야 한다.
  - 값이 너무 크면 원래의 단어 임베딩 정보가 묻혀 학습이 제대로 이루어지지 않는다.
- 입력 문장의 길이에 상관 없이 언제나 위치 정보를 계산할 수 있어야 한다.

이때, 이 두 조건을 만족시키는 대표적인 함수가 바로 **삼각 함수(sin, cos)** 이다. 삼각함수는 출력 범위가 $[-1, 1]$로 안정적이고, 주기성을 가지기 때문에 어떤 위치 값이 입력되더라도 유효한 출력을 만들어낼 수 있다.



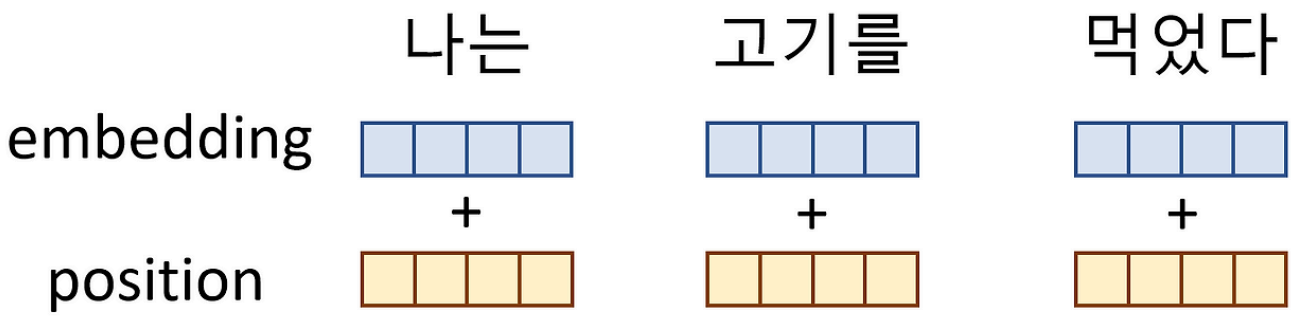
그러나 Sin 함수의 치명적인 단점은 y값이 주기적으로 반복돼, 정보가 겹친다는 것이다. 이를 해결하기 위해 cos, sin 함수를 모두 사용하고, 각 임베딩의 차원별로 sin, cos 함수를 번갈아가면서 사용한다.

임베딩 차원이 짝수이면 sin 함수를, 홀수면 cos 함수를 사용한다.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

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

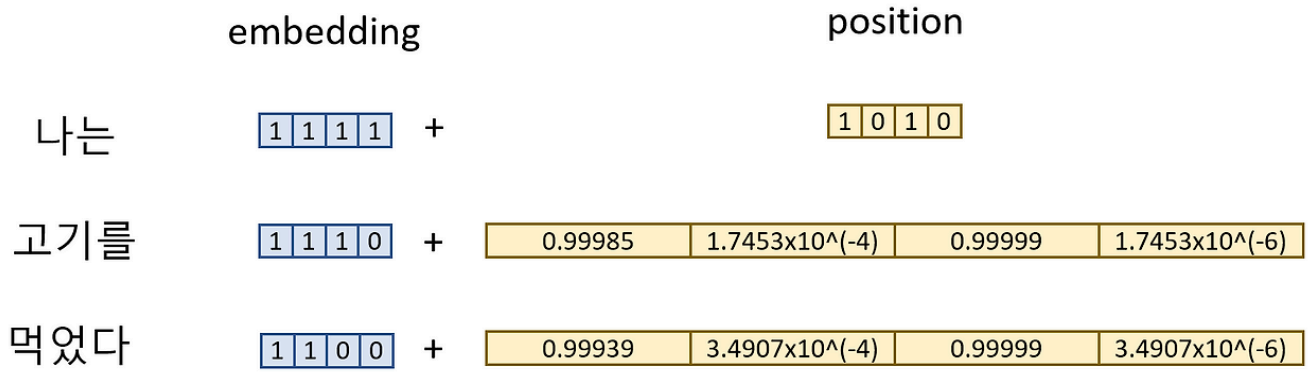
이렇게 구한 positional empbedding 벡터를 아래와 같이 더해주면, Positional encoding 작업이 완성된다.



"나는 고기를 먹었다."로 예를 들어보면,
이 문장은 "나는", "고기를", "먹었다" 세 개의 단어로 구성된다. 각각 위치는 0,1,2에 대응된다.

임베딩 차원을 4로 설정하면 각 단어는 네 개의 좌표를 가지게 된다.

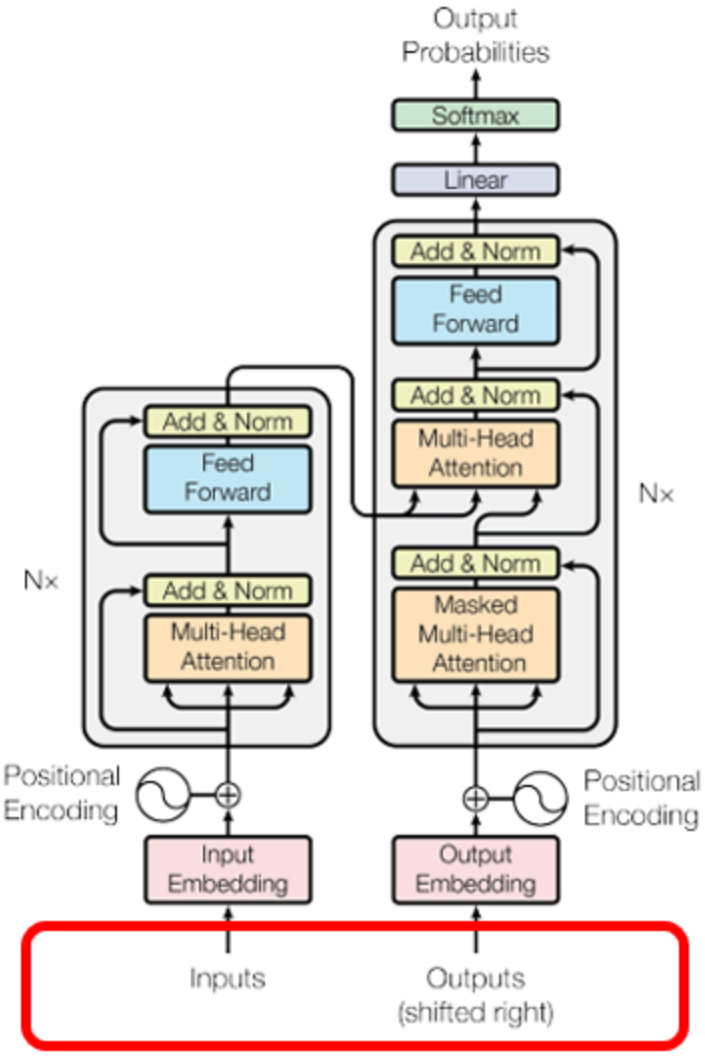
위치 0("나는")의 첫 번째 차원은 $\cos(0) = 1$, 두 번째 차원은 $\sin(0)$이 되고, 나머지 차원도 같은 방식으로 계산된다. 이렇게 얻은 Positional Encoding 벡터는 단어 임베딩에 더해져 최종 입력 표현을 형성한다.



결과적으로 Positional Encoding은 **Transformer가 단어의 순서를 학습할 수 있도록 돕는 장치**이다.
이는 병렬 처리라는 장점을 유지하면서도, 언어의 시계열적 특성을 보존할 수 있게 해준다.

Model Architecture

단어 토큰화 Input/Output Tokenization



자연어 문장은 그대로는 모델이 처리할 수 없기 때문에, 먼저 이를 모델이 이해할 수 있는 **토큰 시퀀스 (Token Sequence)** 형태로 변환해야한다. 이 과정을 토큰화(Tokenization)이라고 한다.

토큰화 방식에는 여러 Level이 있다.

예를 들어
`"I study AI."`
라는 문장을 토큰화하면 다음과 같이 변환된다.

```
["I", "study", "AI", "."]
```

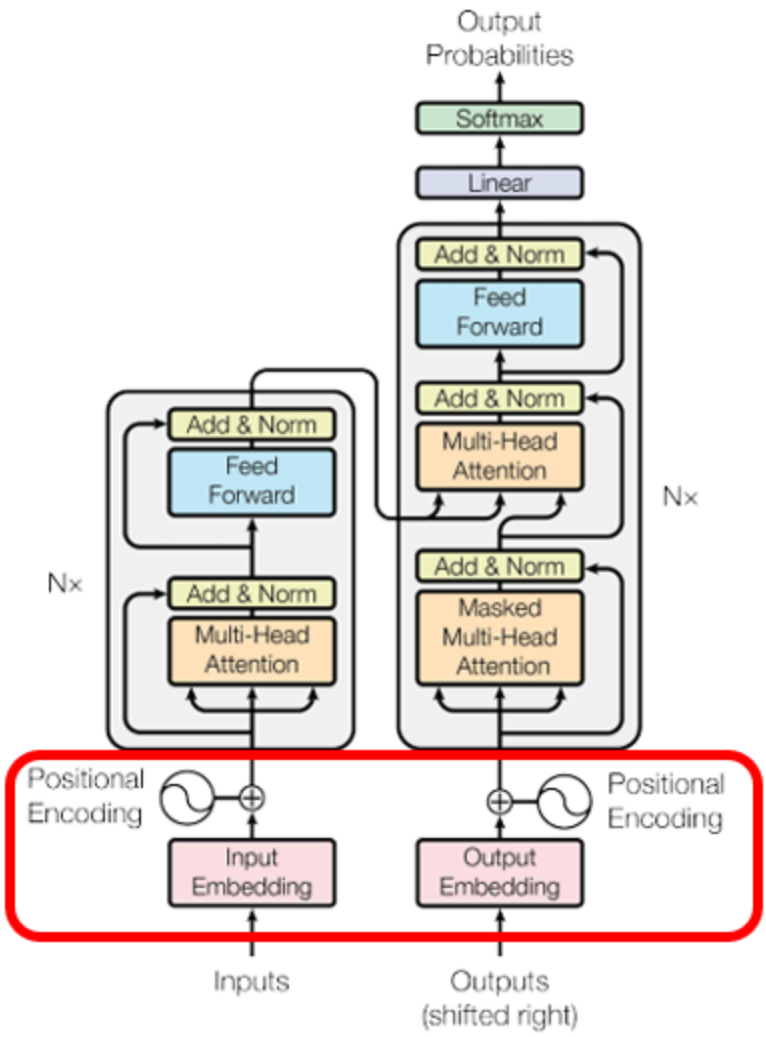
토큰화된 각각의 단어는 모델 내부에서 고유한 정수 ID로 매핑된다. 이를 **숫자 매핑(Numerical Mapping)**이라고 하며, 사전에 정의된 **단어 집합(vocabulary)** 을 기반으로 이루어진다.

```
["I", "study", "AI", "."]→ [23128, 17, 182, 0]
```

이후 입력의 길이가 일정해야 학습이 원활히 이루어지므로, 문장의 길이를 맞추기 위해 **패딩(Padding)** 이 사용된다. 문장이 짧으면 패딩 토큰을 추가하고, 길면 잘라낸다.

`[23128, 17, 182, 0] → [23128, 17, 182, 0, 50256, 50256]`
(50256은 토큰 ID로 가정)

단어 임베딩(Input/Output Embedding)



숫자로 변환된 토큰ID는 여전히 모델이 처리하기엔 단순한 정수일 뿐이다. 따라서 이를 고차원 **실수 벡터 공간**으로 변환하는 과정이 필요하다. 이르르 **임베딩(Embedding)**이라고 한다.

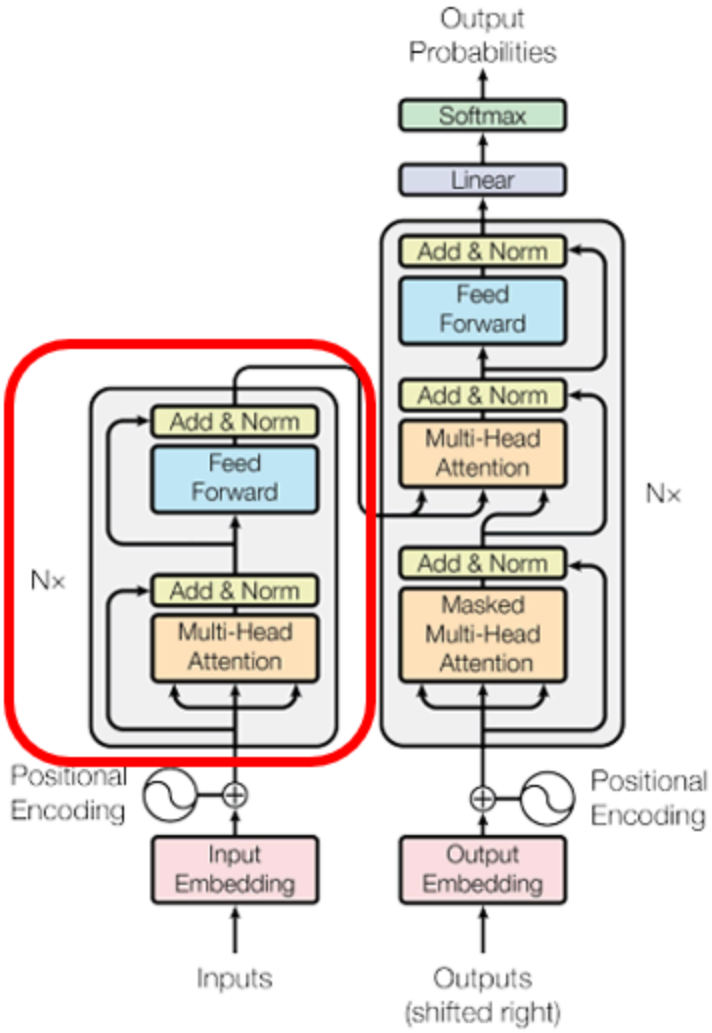
임베딩 레이어는 각 토큰 ID를 지정된 차원의 벡터로 변환한다. 보통 이 차원은 $d = 128, 512, 768, 2048$ 처럼 모델 크기에 따라 다르다. 예를 들어 "I"라는 토큰이 23128번 ID로 매핑되었다면, 임베딩 레이어를 통해 512차원의 벡터로 변환된다.

또한 Transformer는 RNN이나 CNN과 달리 **순서 정보**를 자체적으로 기억하지 못한다. 따라서 단어의 순서를 보존하기 위해 **위치 임베딩(Positional Embedding)**을 추가한다. 즉, 최종 입력 벡터는

$$입력벡터 = 단어임베딩 + 위치임베딩$$

형태가 된다. 이렇게 하면 모델은 단어의 의미뿐만 아니라, 해당 단어가 문장에서 어디에 위치하는지까지 함께 고려할 수 있다.

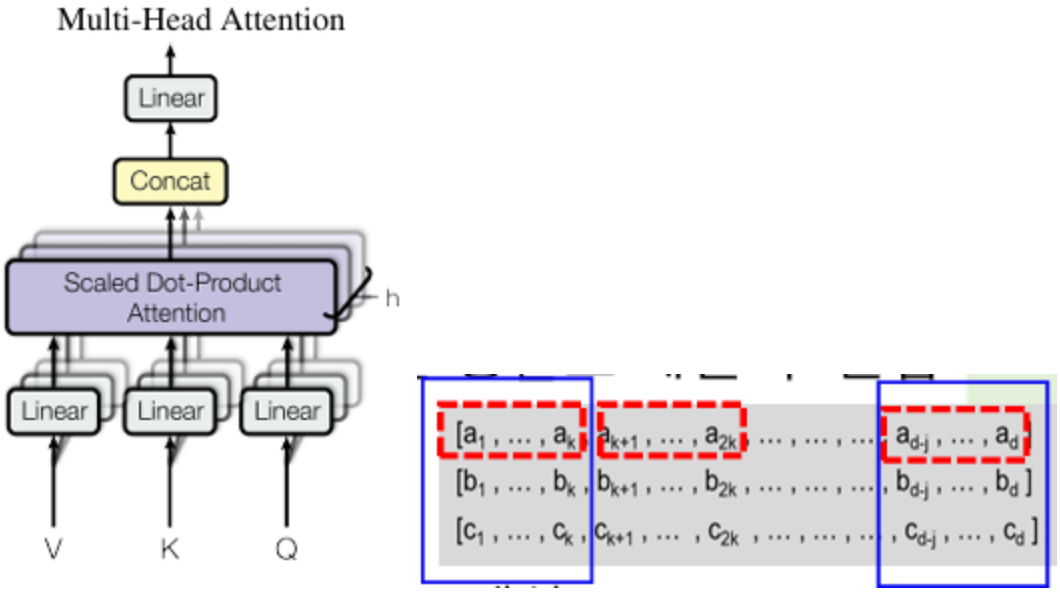
인코더 Encoder



Transformer에서 인코더는 입력 텍스트의 단어들 사이의 문맥적 관계를 파악하고 이를 의미있는 벡터 표현으로 변환하는 역할을 한다. 입력이 $n \times d$ 차원의 행렬이라면, 인코더의 출력 역시 동일한 $n \times d$ 차원을 유지하면서 더 풍부한 의미적 정보를 담게 된다.

인코더는 총 **6개의 동일한 레이어(stack)** 가 차례로 쌓여 있는 구조를 가진다. 각 레이어는 크게 두 개의 서브 레이어로 구성되어 있으며, 이 두 서브 레이어가 반복적으로 쌓이면서 단어들 간의 복잡한 관계를 학습하게 된다.

첫 번째 서브 레이어는 **Multi Head Attention**이다. 이는 하나의 어텐션만 사용하는 것이 아니라 여러 개의 독립적인 어텐션 헤드를 동시에 활용하여 입력 토큰들 간의 다양한 관점에서의 연관성을 학습한다. 어떤 헤드는 단어 간의 짧은 거리 의존성을, 다른 헤드는 긴 문맥 의존성을 포착할 수 있다. 이렇게 병렬적으로 얻어진 결과들은 다시 하나로 이어 붙인 뒤(concat) 선형 변환을 거쳐 최종 출력으로 정리된다.



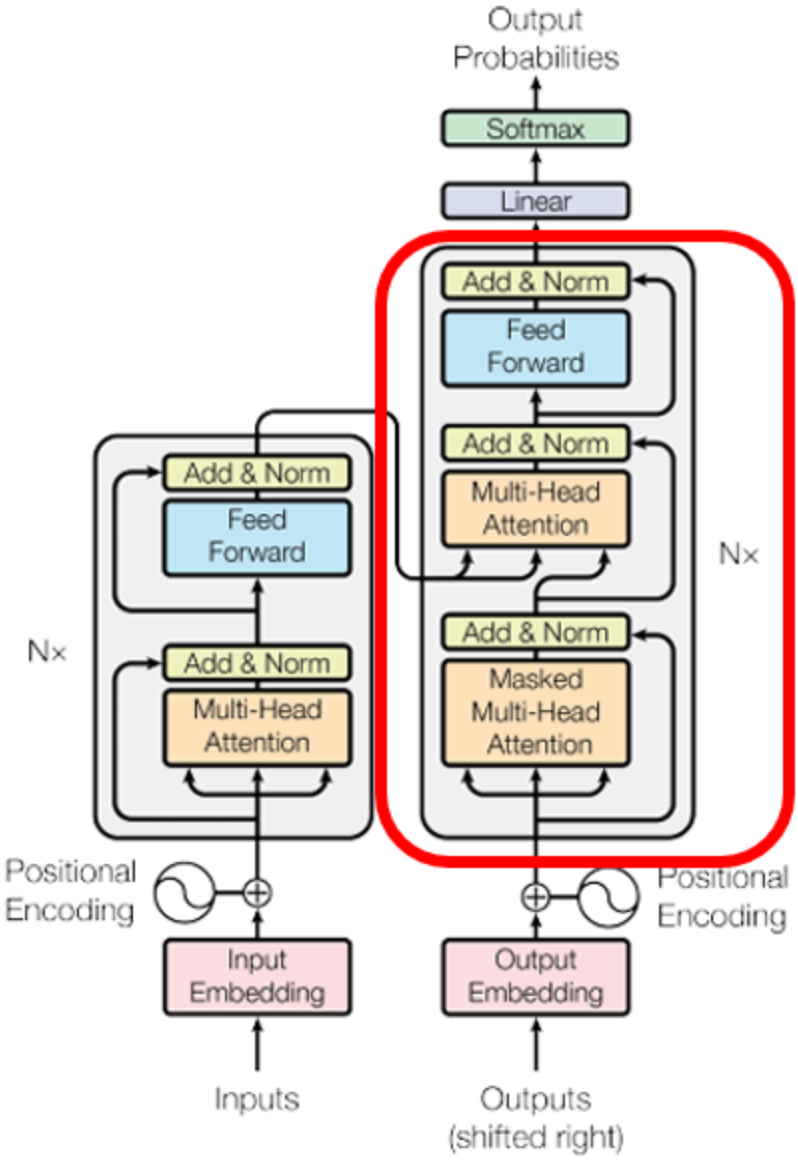
(각 토큰 벡터(d차원)를 여러 개(h)의 헤드(k 차원 k=d/h) 별로 나누어 각 헤드에서의 attention을 병렬로 계산 후 결합)

두 번째 서브 레이어는 **Position-wise Feedforward Network** 이다. 이는 각 위치의 벡터에 대해 동일하게 적용되는 작은 신경망으로, 비선형 변환을 통해 더 풍부하고 정교한 표현을 학습하도록 돕는다. 이렇게 함으로써 단순한 어텐션 정보에 비해 더 복잡한 특징들을 모델이 내재화할 수 있다.

또한 Transformer 인코더의 중요한 특징은 **잔차 연결(Residual Connection)** 과 **레이어 정규화(Layer Normalization)** 이다. 각 서브 레잉어의 출력에 입력을 더해주는 잔차 연결은 정보 손실을 막고, 깊은 신경망에서도 학습이 안정적으로 이루어지도록 한다. 이어서 적용되는 레이어 정규화는 각 출력값을 정규화하여 학습의 안정성을 한층 높여준다.

결과적으로 인코더는 입력 문장을 여러 층의 어텐션과 피드포워드 연산을 거치며 점차 고차원적이고 정교한 의미 표현으로 변환된다. 이는 이후 디코다가 올바른 출력을 생성하는 데 핵심적인 기반이 된다.

디코더(Decorder)



디코더는 인코더에서 만들어진 의미 표현을 입력으로 받아, 목표 언어의 단어를 하나씩 예측하면서 최종 번역 문장을 생성하는 역할을 한다. 디코더의 출력은 타겟 어휘 집합에 대한 확률 분포이며, 이 확률 벡터를 통해 모델은 어떤 단어가 다음에 올지를 순차적으로 결정한다.

디코더 역시 인코더와 마찬가지로 **6개의 동일한 레이어 스택** 으로 구성된다. 그러나 각 레이어는 인코더와 달리 **3개의 서브 레이어** 를 포함하고 있다.

첫 번째 서브 레이어는 **Masked Multi-Head Self-Attention** 이다.
이는 인코더의 셀프 어텐션과 유사하지만, 차이점이 있다. 디코더는 이미 생성된 단어들을 기반으로 다음 단어를 예측해야 하므로, 현재 시점 이후의 단어를 참조해서는 안ㄴ된다. 이를 위해 **Look-ahead Masking** 기법을 사용하여 미래 시점의 토큰은 마스킹 처리하고, 현재 시점 이전의 토큰 정보만 활용하도록 제한한다. 이렇게 함으로써 디코더는 **순차적(auto-regressive)** 특성을 유지할 수 있다.

두 번째 서브 레이어는 **Multi-Head Encoder-Decoder Attention**이다. 이 단계에서는 디코더의 Query 벡터가 인코더의 출력(Key, Vale)과 상호작용 한다. 이를 통해 디코더는 입력 문장(소스 시퀀스)의 어떤 부분이 현재 번역하려는 단어와 가장 관련성이 높은지를 학습한다. 예를 들어 번역 과정에서 영

어의 "apple"을 한국어로 옮길 때, 디코더는 인코더에서 "apple"과 관련된 표현을 집중(attend)하여 올바른 대응 단어를 생성하게 된다.

세 번째 서브 레이어는 **Feed Forward Network**이다 이 부분은 인코더와 동일한 구조로, 각 위치의 벡터에 독립적으로 적용되는 작은 신경망이다. 비선형변환을 통해 디코더의 표현을 한층 더 정교하게 다듬어준다.

모든 서브 레이어에는 인코더와 동일하게 **잔차 연결(Residual Connection)**과 **Layer Normalization**이 적용된다. 이를 통해 정보 손실을 최소화하고 학습을 안정화한다.

최종적으로 디코더는 학습 시와 추론 시에 조금 다르게 동작한다.

- 학습 시(Training): 전체 소스 시퀀스와 정답 타겟 시퀀스를 모두 입력받아, 각 위치에서의 타겟 단어 분포를 동시에 학습한다.
- 추론 시(Inference) : 전체 소스 시퀀스와 지금까지 생성된 타겟 단어들을 입력받아, 새로운 타겟 단어에 대한 확률 벡터를 순차적으로 생성한다.