(5주차) Transformer

명지대학교
MYONGJI UNIVERSITY

트랜스포머(Transformer) 개요

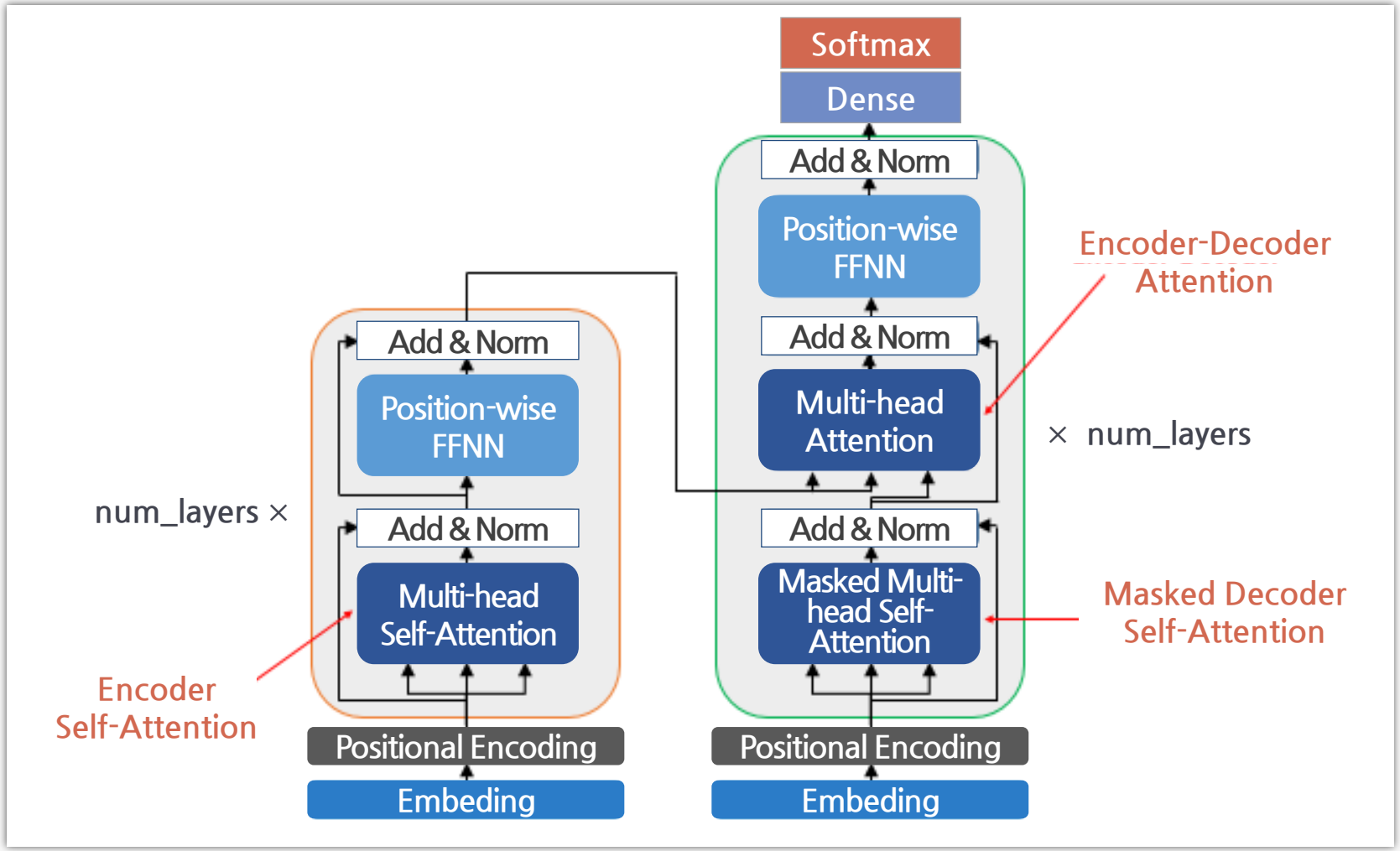
트랜스포머(Transformer)란?

2017년 구글이 "Attention is all you need" 논문에서 발표한 모델

Seq2Seq의 구조인 인코더 - 디코더 구조로 되어 있으면서
RNN을 제외하고 어텐션(Attention)만으로 구현한 모델

학습 속도가 개선되었으며
기존의 RNN을 사용한 모델보다 성능 우수

트랜스포머(Transformer) 개요



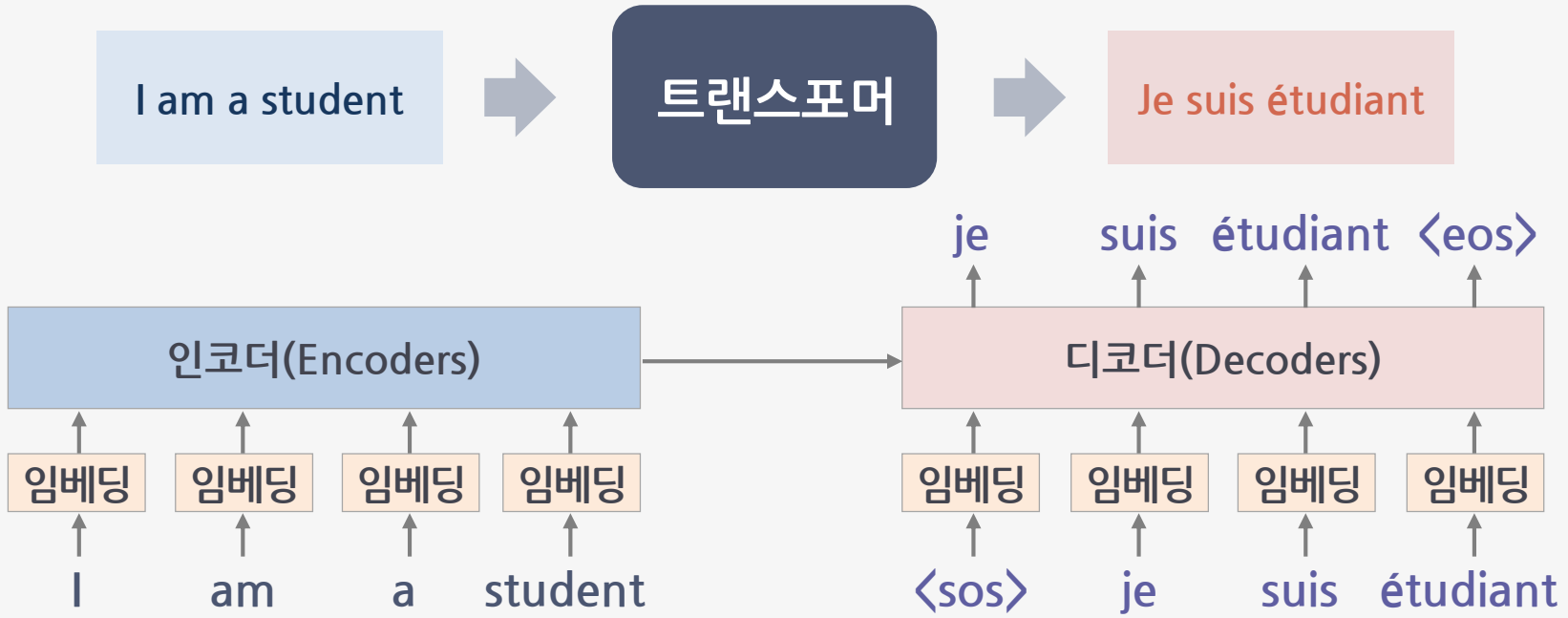
트랜스포머(Transformer)

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 구조

Input 문장에 대한 처리 후 Output 문장을 출력

트랜스포머는 인코더와 디코더 및 인코더와 디코더를 연결하기 위한 Connection을 구성



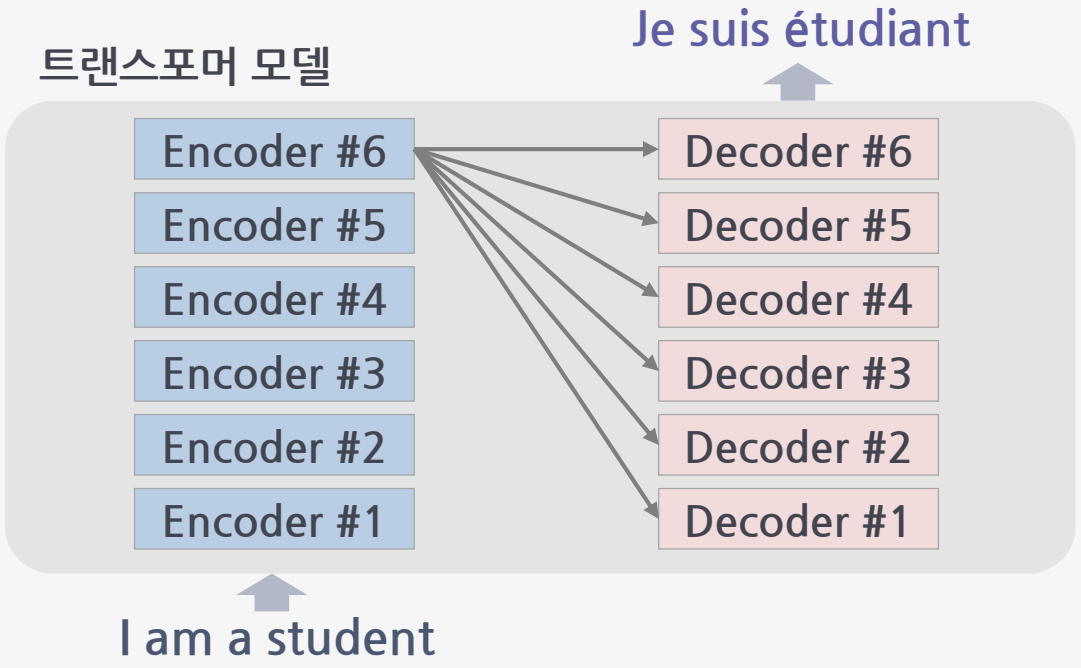
트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 인코더와 디코더 구조

Encoding 컴포넌트는 여러 개의 인코더들로 구성

Decoding 컴포넌트는 Encoding 컴포넌트와 동일한 개수의 디코더로 구성

논문에서는 6개의 인코더와 디코더로 구성되어 있으나 임의의 갯수로 변경 가능



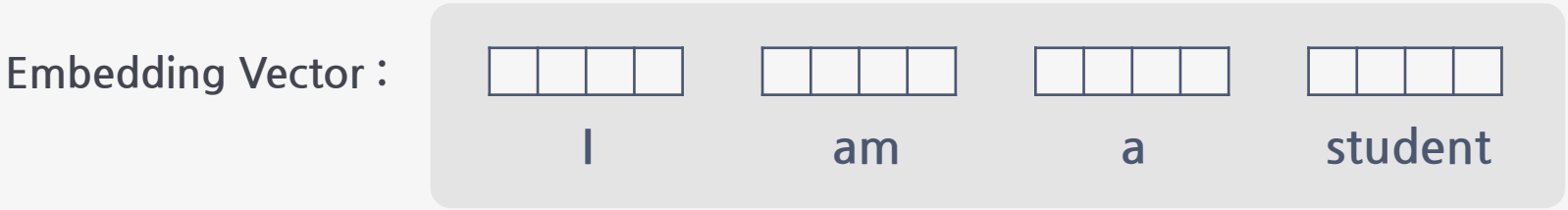
트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 임베딩과 Positional Encoding

- Embedding Vector

입력 단어들을 임베딩하여 벡터로 변환

각 단어들은 512 크기의 벡터로 임베딩



트랜스포머(Transformer) 개요

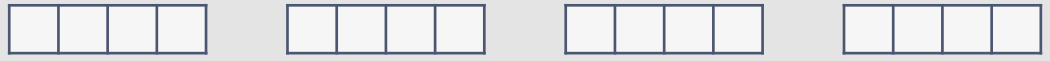
트랜스포머(Transformer)의 임베딩과 Positional Encoding

- Embedding Vector

입력 단어들을 임베딩하여 벡터로 변환

각 단어들은 512 크기의 벡터로 임베딩

Embedding Vector :



순차적 특성을 가지는 RNN을 제거함으로써
Embedding Vector의 각 단어에 대한 Position 파악 불가

트랜스포머에서 각 단어에 대한 Position 정보 부여 필요

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 임베딩과 Positional Encoding

Positional Encoding

Positional Encoding이란?

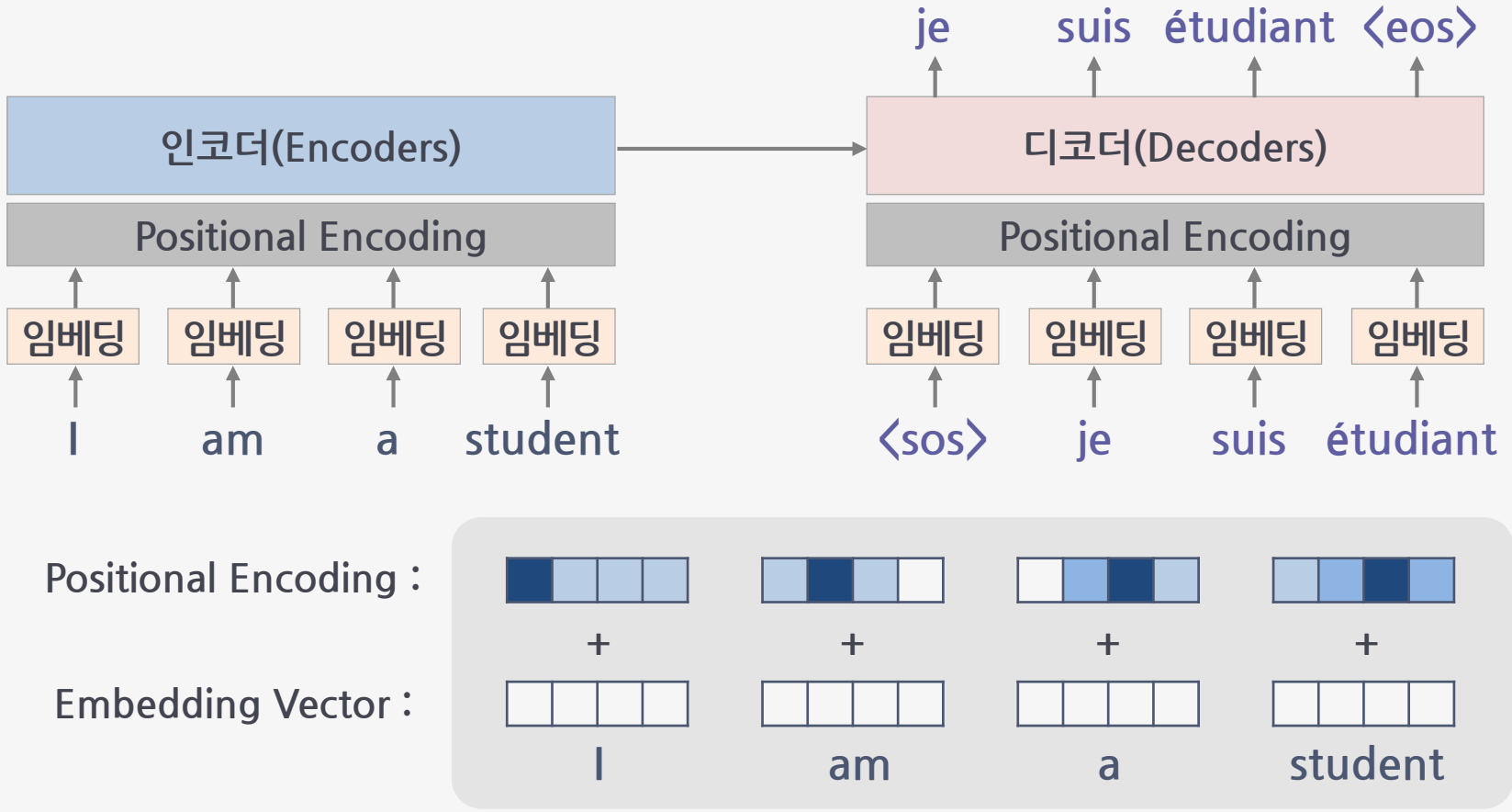
트랜스포머에서 각 Input 단어의 Position 정보 부여를 위해

각 단어의 임베딩 벡터에
Position 정보들을 추가하여 모델의 입력으로 사용하는 방법

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 임베딩과 Positional Encoding

Positional Encoding



트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

> ‘The animal didn’t cross the street because it was too tired’라는 문장의 경우

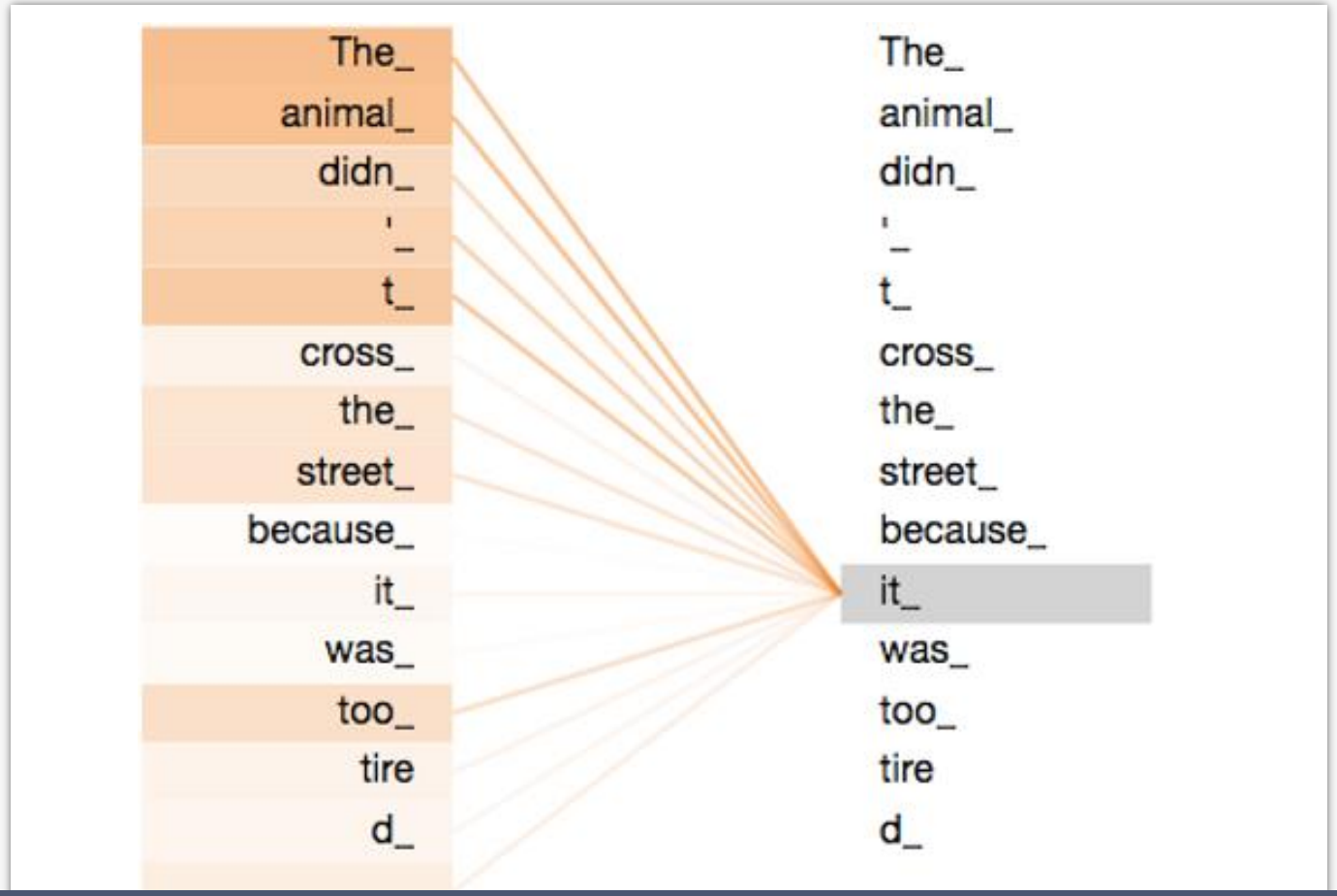
“it”이 가리키는 것은 무엇일까? animal or street

Self-Attention을 이용하여 animal과 it 연결 가능

트랜스포머(Transformer)

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention



트랜스포머의 Self-Attention은 현재 처리 중인 단어에 대해
다른 연관 있는 단어들과의 맥락을 파악하기 위한 방법 제공

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

트랜스포머(Transformer)의 주요 하이퍼파라미터

- d_model = 512 : 트랜스포머의 인코더와 디코더에서의 정해진 입력과 출력의 크기
- num_layers = 6 : 인코더와 디코더 각각의 층(layer) 수
- num_heads = 8 : 분할해서 병렬로 수행할 어텐션 수

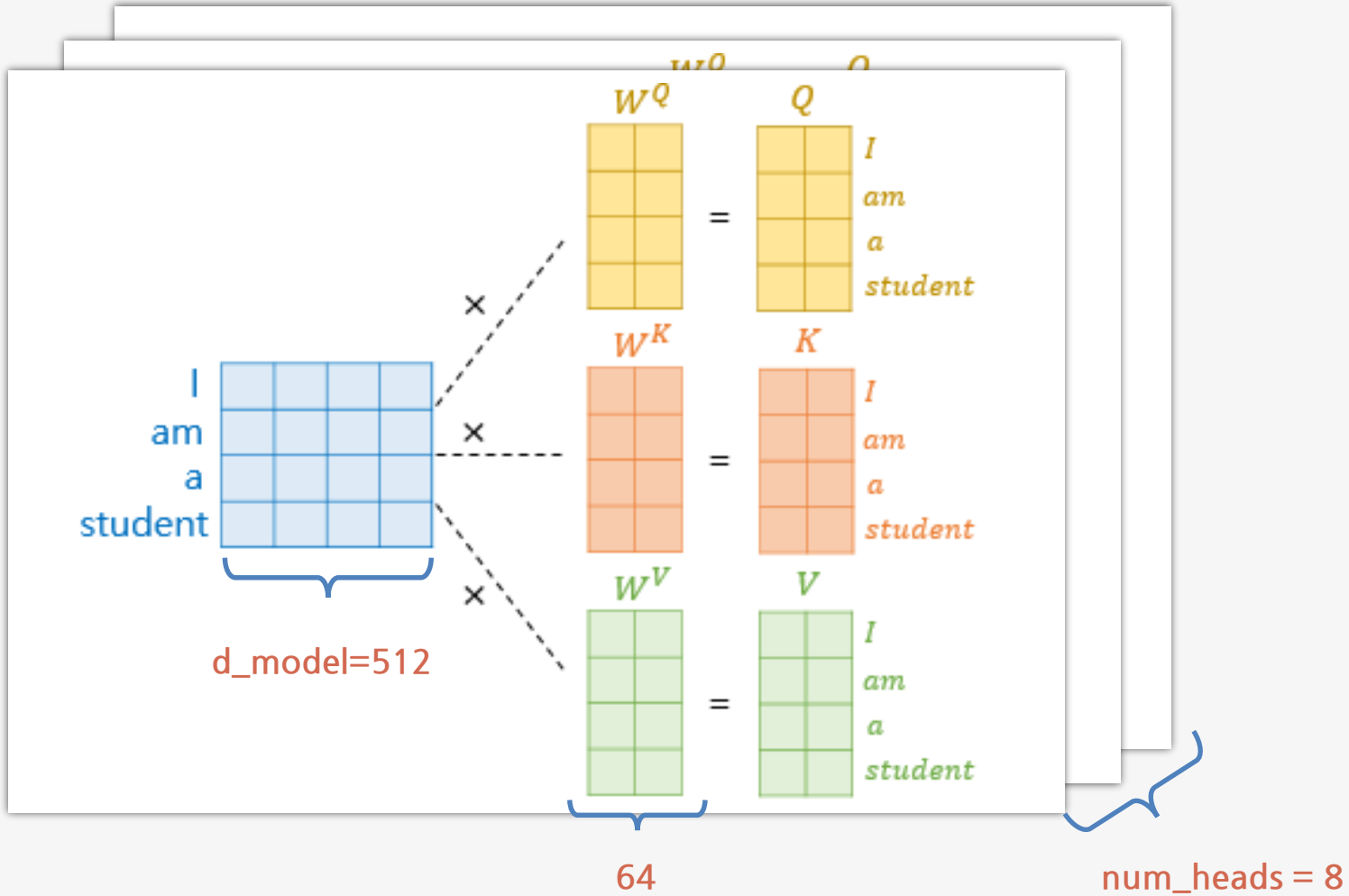
Self-Attention의 Query, Key, Value 계산

- Self-Attention의 첫 단계는 입력 문장에 대해 Query, Key, Value 계산
- Input 문장의 512 크기의 벡터와 학습할 Weight(WQ, WK, WV)를 곱하여 64 크기의 Query, Key, Value 벡터 생성

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

Self-Attention의 Query, Key, Value 계산

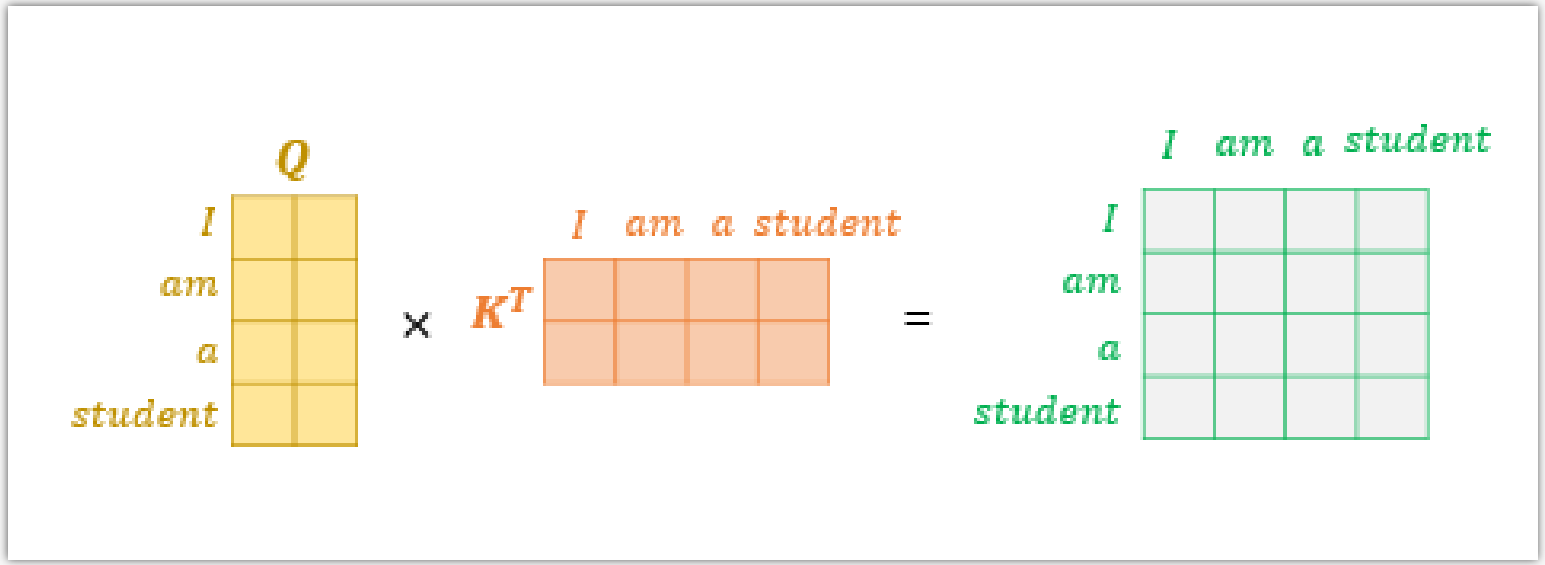


트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

Self-Attention의 계산 정리

- Query에 Key를 Transpose한 행렬을 내적(Dot Product) 을 계산



트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

Self-Attention의 계산 정리

- Query에 Key의 Transpose한 행렬을 내적(Dot Product)

Query와 Key가 특정 문장에서 중요한 역할을 하고 있다면
트랜스포머는 이들 사이의 내적(Dot Product) 값을 크게 하는 방향으로 학습

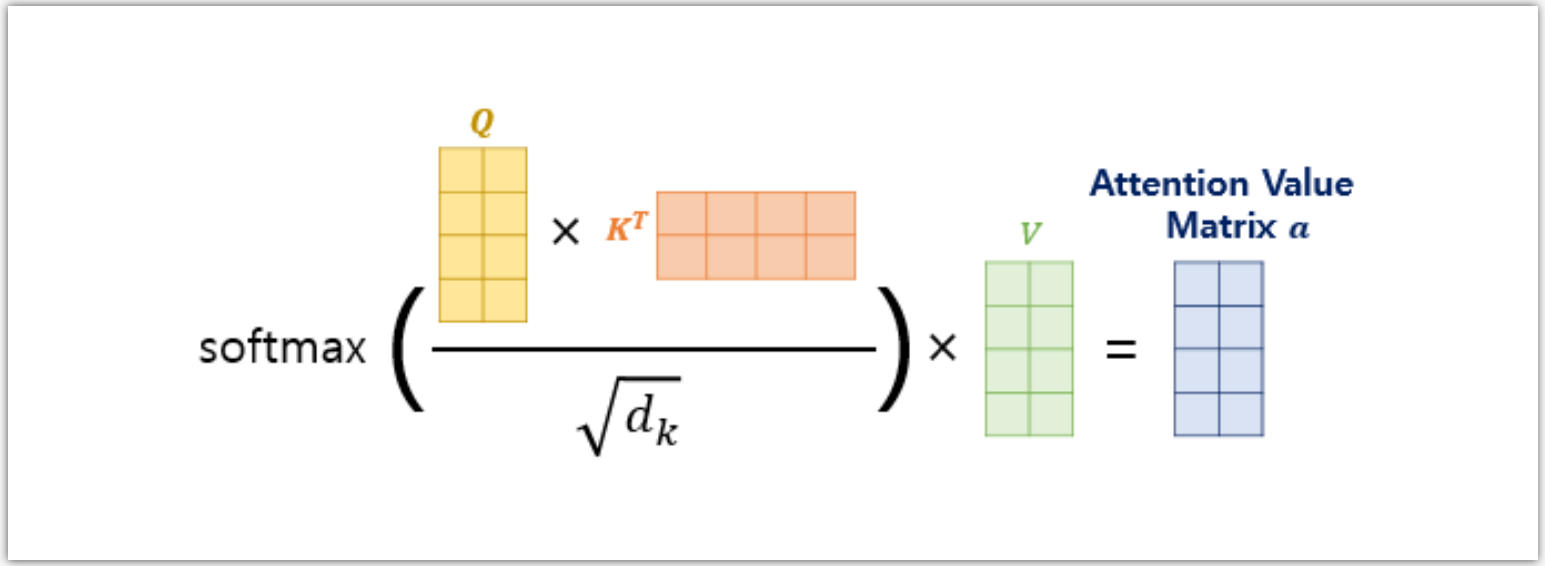
내적 값이 커지면 해당 Query와 Key가 벡터 공간 상
가까이에 있을 확률이 큼

트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

Self-Attention의 계산 정리

- Key의 벡터 크기인 64의 Root 값인 8로 나눈 후 소프트맥스 함수 적용
- Value와 Dot Product를 곱하여 Attention Value인 Z 계산



트랜스포머(Transformer) 개요

트랜스포머(Transformer)의 Self-Attention

Self-Attention의 계산 정리

- Key의 벡터 크기인 64의 Root 값인 8로 나눈 후 소프트맥스 함수 적용
- Value와 Dot Product를 곱하여 Attention Value인 Z 계산

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

$\sqrt{d_k}$로 나눈 이유는 Query와 Key의 내적 행렬의
분산(Variance)을 축소하고 Gradient Vanishing 발생 방지

Multi-head Self Attention

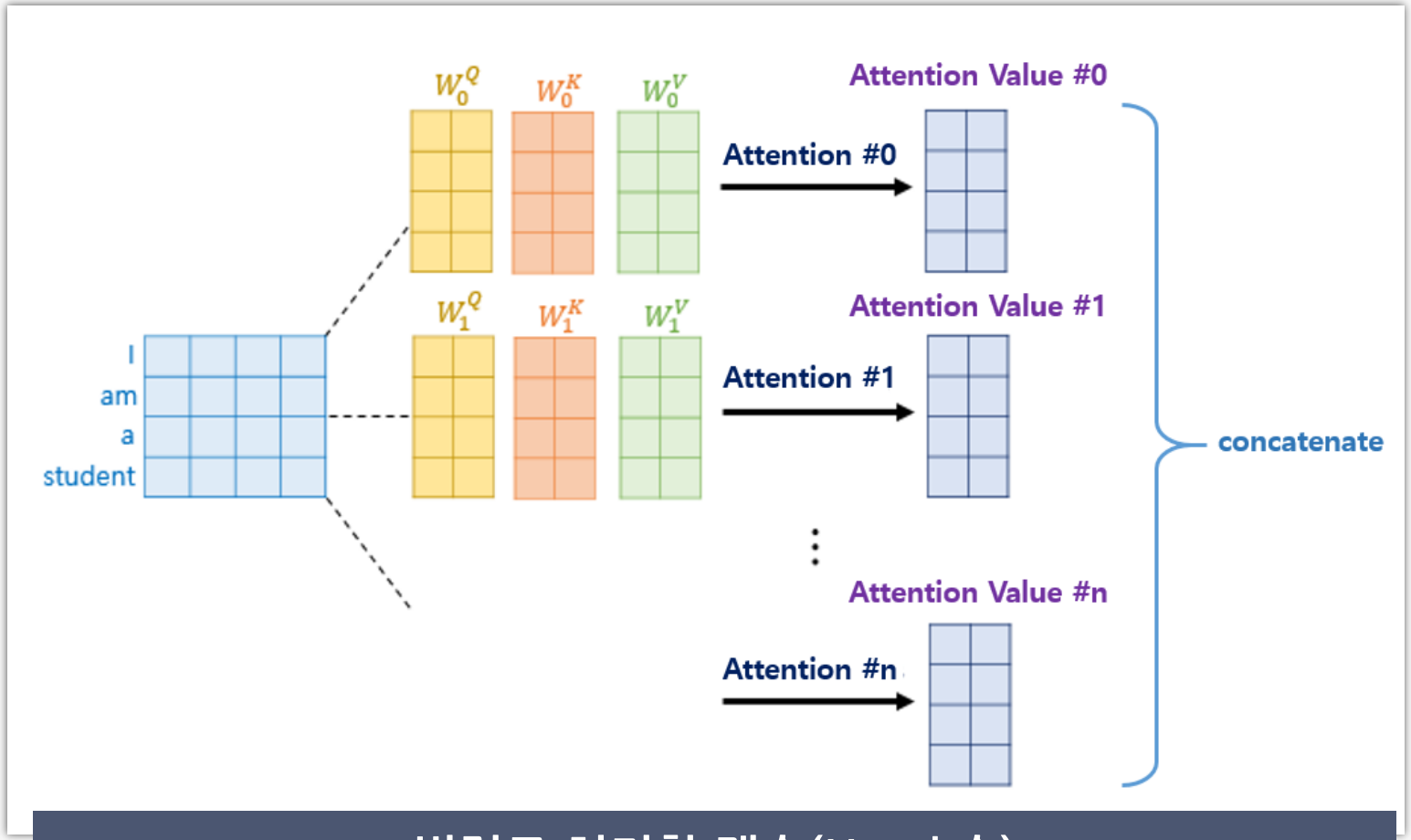
Multi-head Self Attention이란?

단일한 Self-Attention을 수행하는 것보다
다수의 Self-Attention을 병렬로 수행하는 것이 효과적

Input 문장에 대해 N개의 Query, Key, Value를 계산하기 위한
N개의 가중치(WQ, WK, WV)를 생성하여
병렬로 Self Attention 수행 및 N개의 Attention Value 계산

Attention을 병렬로 수행함으로써
다양한 관점에서 단어 간 관계 정보 파악 가능

Multi-head Self Attention

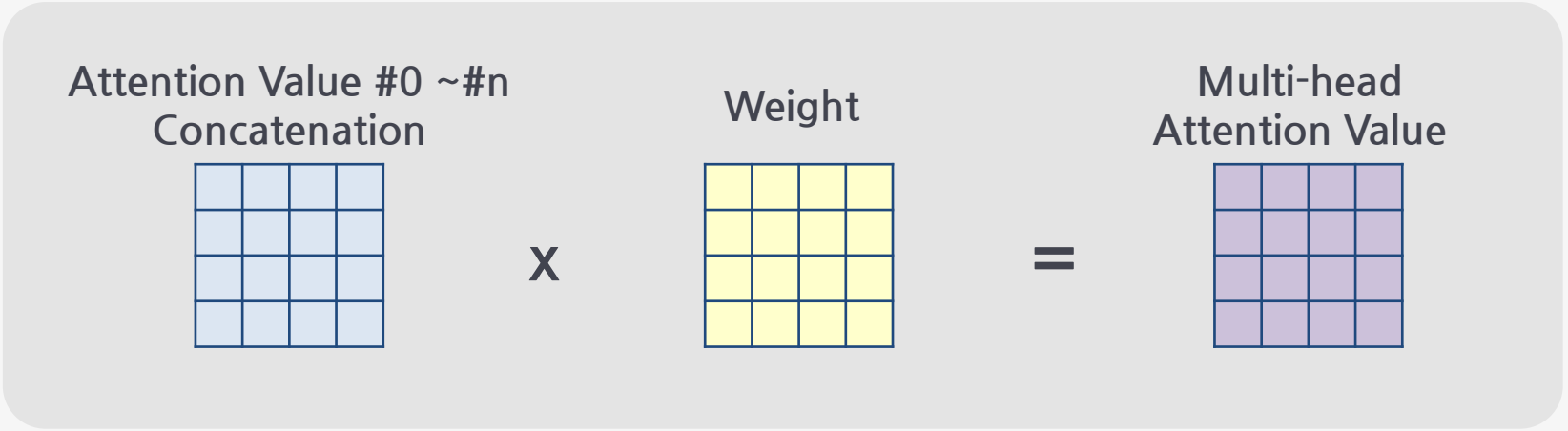


병렬로 처리할 갯수(Head 수)
논문에서는 num_heads = 8

Multi-head Self Attention

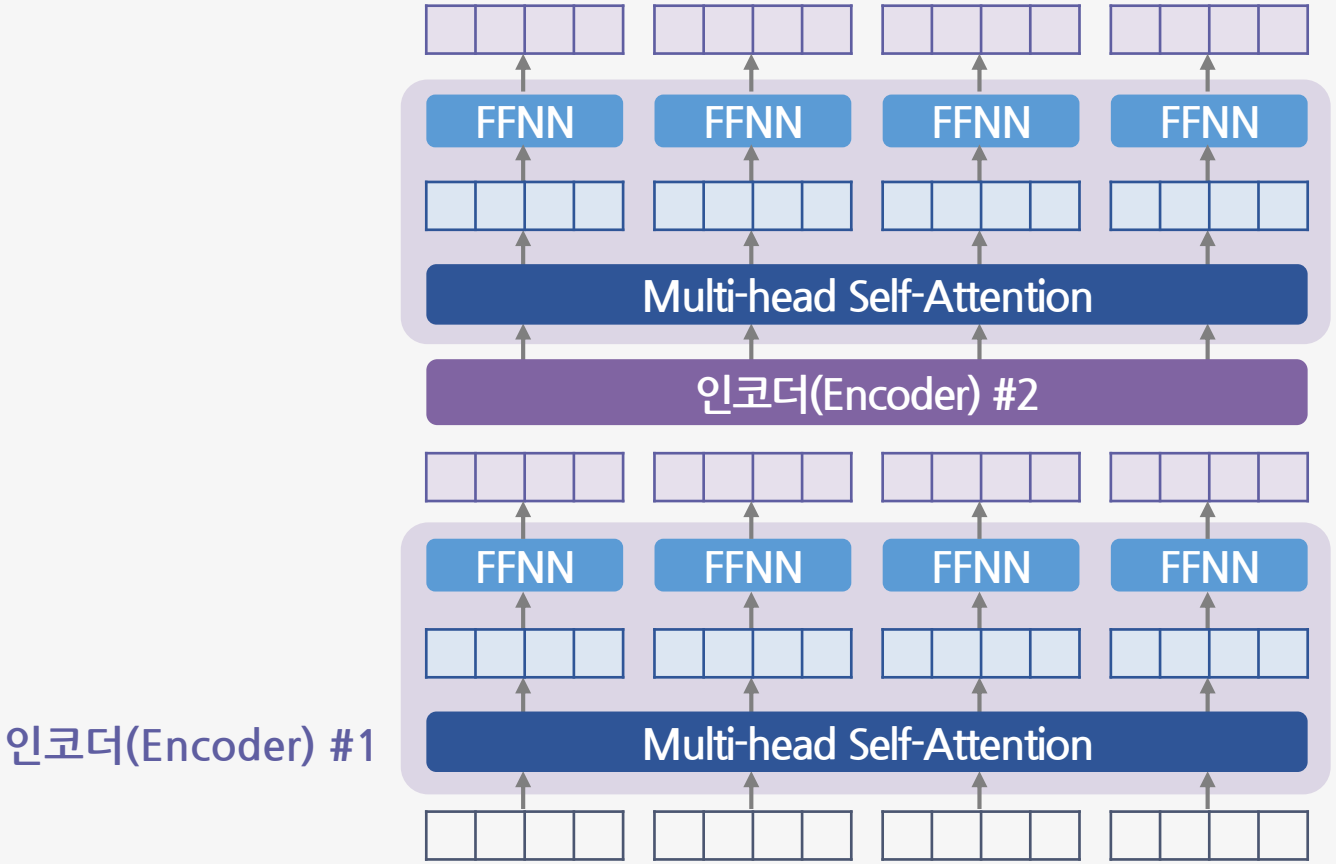
Multi-head Attention Value Matrix 계산

- Multi-head Attention 수행 결과 Concatenation
- 학습할 Weight를 곱하여 Multi-head Attention Value Matrix를 최종 결과로 도출



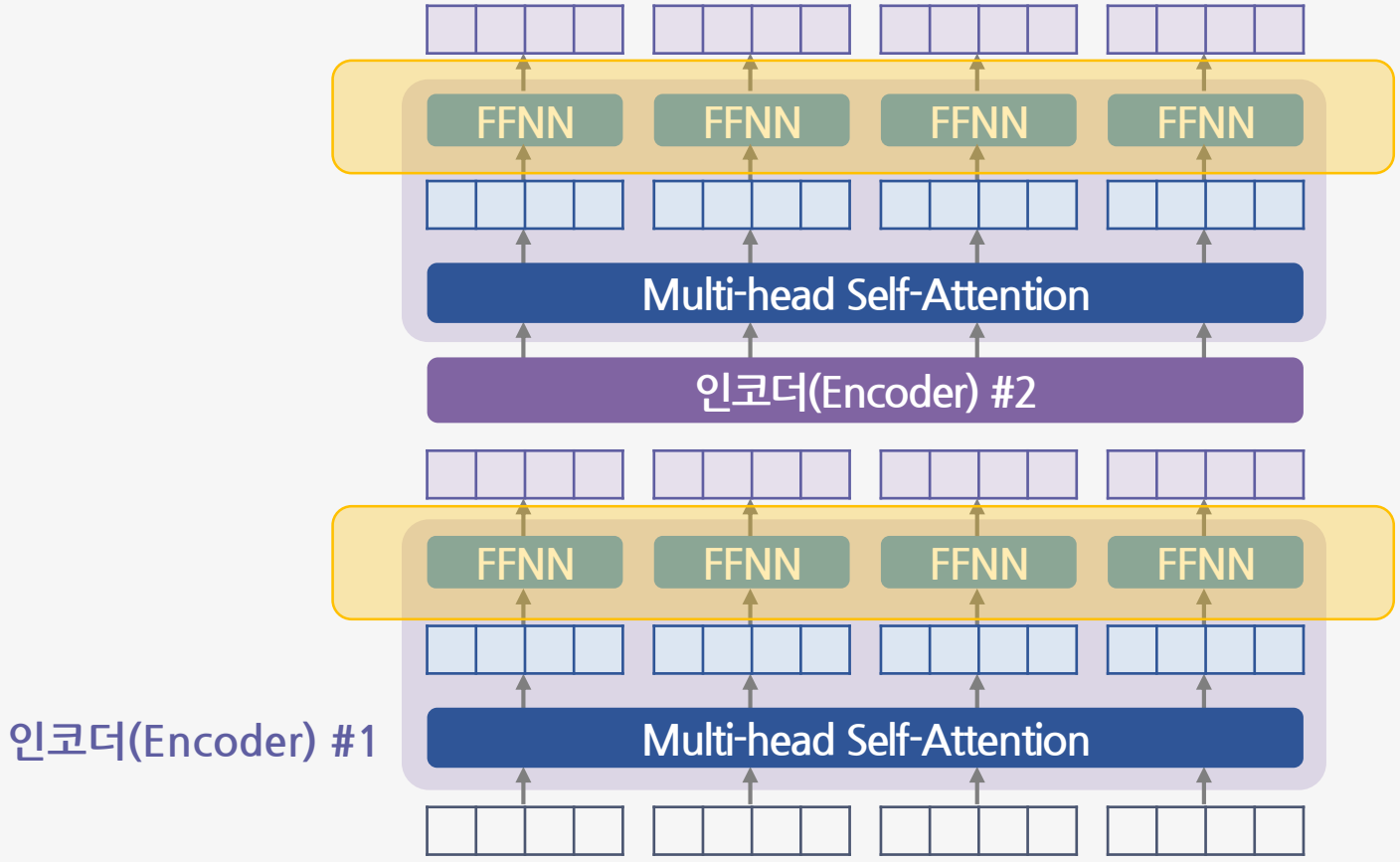
Position-wise FFNN(Feed Forward Neural Network)

- Position-wise FFNN은 인코더와 디코더에서 공통으로 포함된 Sub-layer
- Position-wise FFNN은 일종의 FCNN(Fully-connected Neural Network)



Position-wise FFNN(Feed Forward Neural Network)

- Position-wise FFNN은 인코더와 디코더에서 공통으로 포함된 Sub-layer
- Position-wise FFNN은 일종의 FCNN(Fully-connected Neural Network)



Add & Normalize

잔차 연결(Residual Connection)

잔차 연결(Residual Connection)

뉴럴 네트워크가 깊어지면 Gradient Vanishing/Exploding 발생으로 인해 학습이 잘 이루어지지 않는 경우가 존재

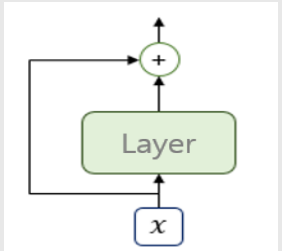
- Degradation 문제를 해결하기 위해 Google은 ResNet을 통해 Residual Connection 적용
- 네트워크의 입력과 출력이 더해진 것을 다음 레이어의 입력으로 사용 Skip Connection

Add & Normalize

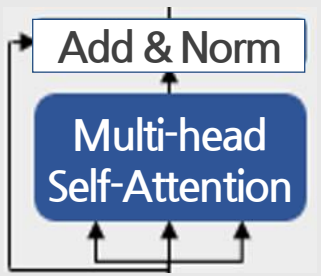
잔차 연결(Residual Connection)

잔차 연결(Residual Connection)

- 스킵 연결을 구현 하는 것은 덧셈 연산의 추가 만으로 가능
- 추가적인 연산량이나 파라미터 불필요
- Residual Connection은 Back Propagation 수행 과정에서 Gradient가 이전 레이어로 잘 전달되도록 함



Residual Connection



Transformer Residual

트랜스포머(Transformer)

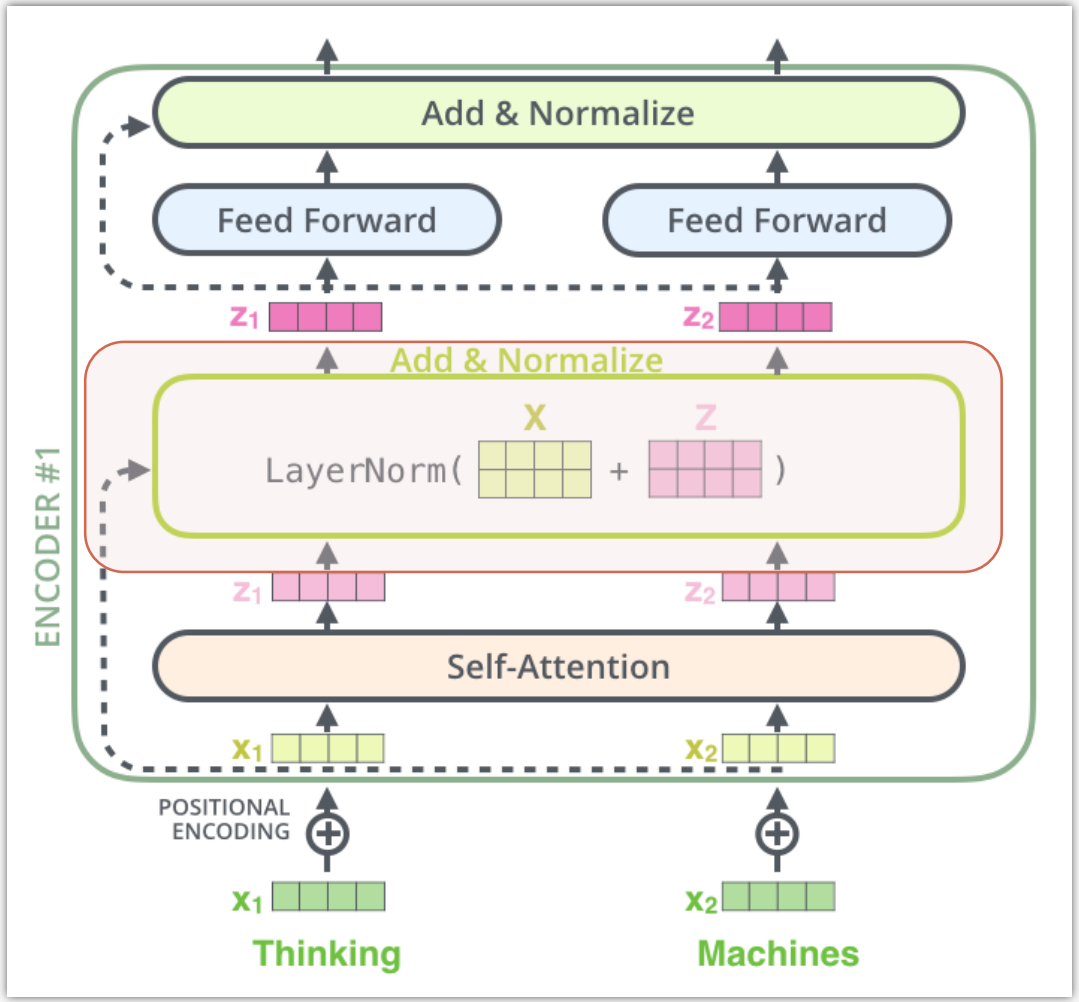
- Add & Normalize
  - Layer Normalization

Residual Connection으로 전달된 입력값과
Multi-head Attention Value를 더한 후 Normalization 수행

Position-wise FFNN의 입력 값으로 전달하기 전
Normalization을 수행함으로써 Over-fitting 방지

트랜스포머(Transformer)

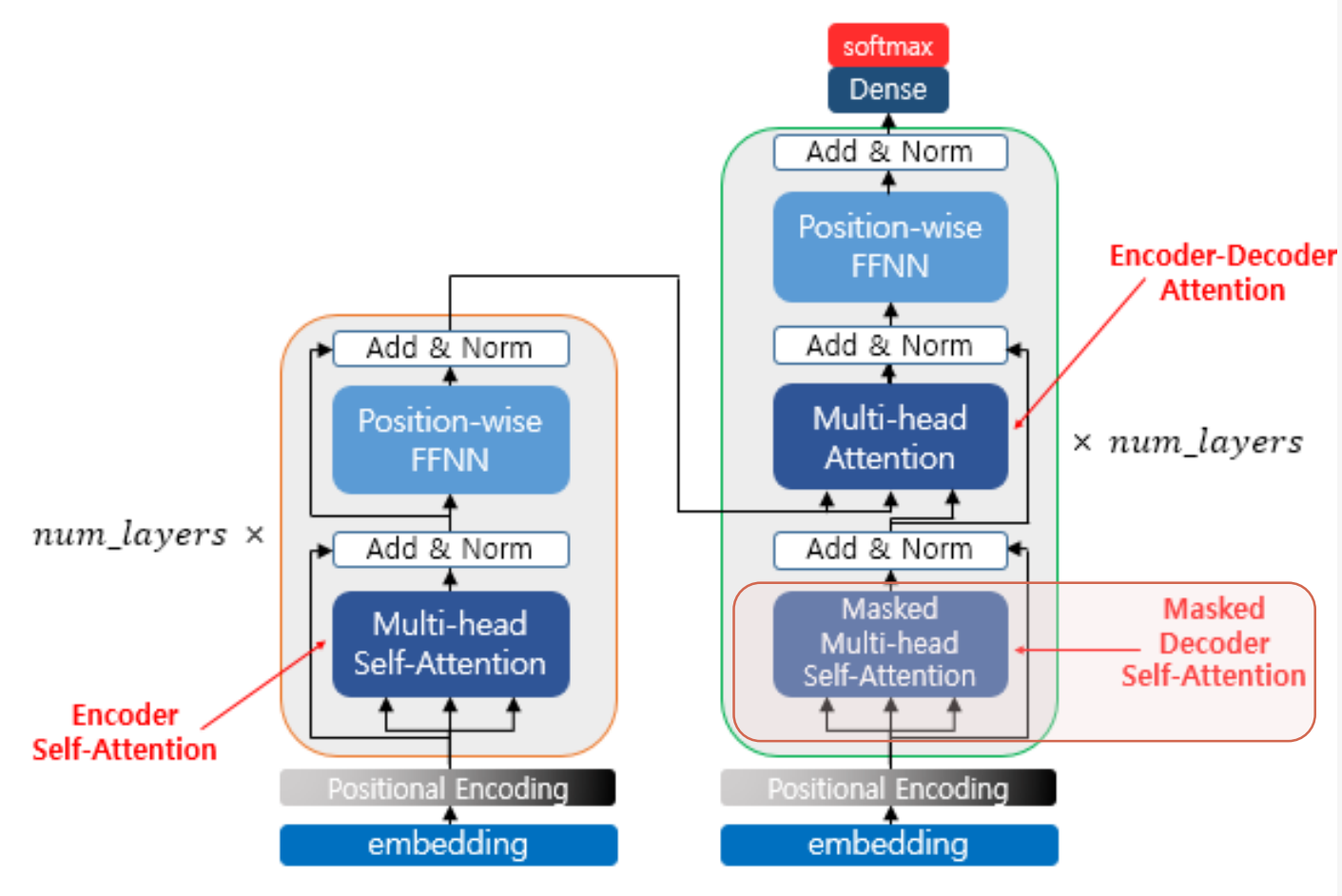
- Add & Normalize
  - Layer Normalization



트랜스포머(Transformer)

Decoder의 Masked Multi-head Self Attention

Decoder의 look-ahead mask



Decoder의 Masked Multi-head Self Attention

Decoder의 look-ahead mask

look-ahead mask란?

미래 참조를 방지하기 위한 masking 기법

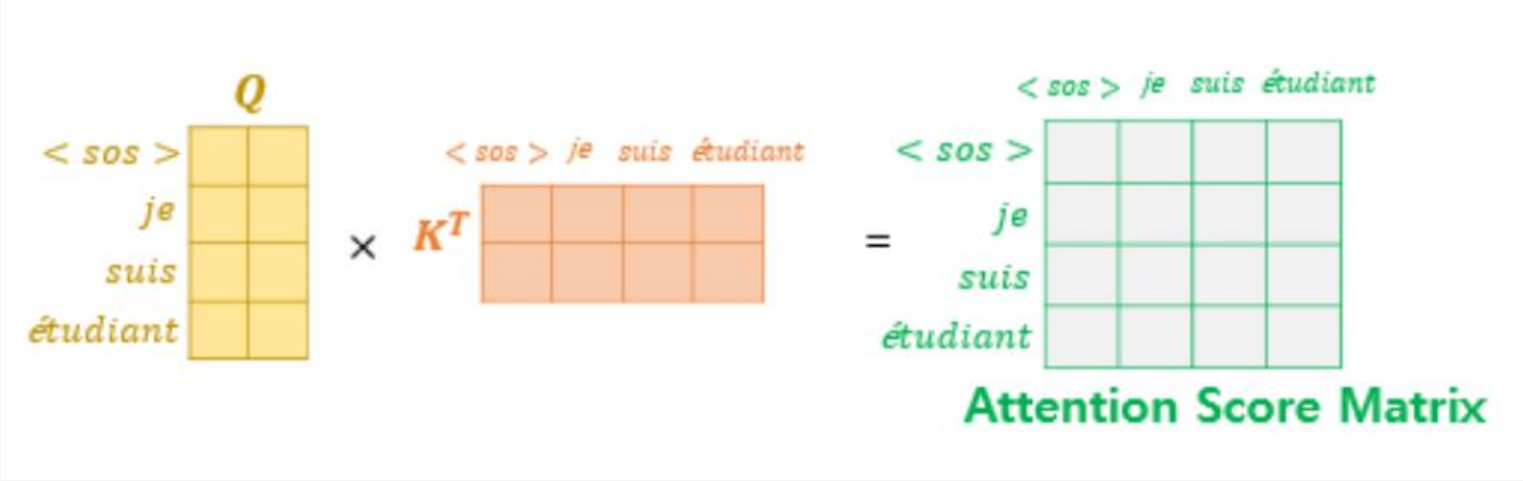
RNN 계열의 신경망은 입력 단어를 매 시점마다 순차적으로 입력
다음 단어 예측에 현재 시점을 포함한 이전 시점에 입력된 단어들만 참고

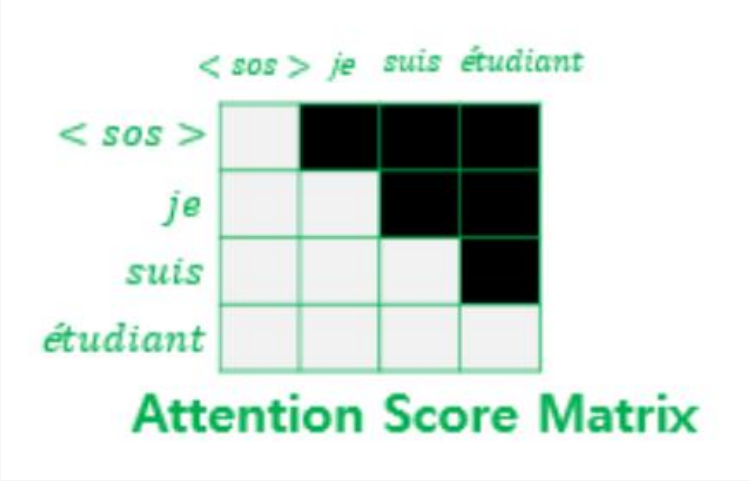
트랜스포머는 문장 행렬로 입력을 한 번에 받으므로 현재 시점의 단어를
예측하고자 할 때, 입력 문장 행렬로부터 미래 시점의 단어까지도 참조

디코더에서는 현재 시점의 예측 시
현재 시점보다 미래에 있는 단어들을 참조 방지

트랜스포머(Transformer)

- Decoder의 Masked Multi-head Self Attention
  - Decoder의 look-ahead mask





Decoder의 Masked Multi-head Self Attention

Decoder의 look-ahead mask 예시

- Sequence 길이가 4인 경우

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- 0: 어텐션을 허용(현재 및 이전 토큰 참조 가능).
- $-\infty$: 어텐션을 차단(미래 토큰 참조 불가).

Decoder의 Masked Multi-head Self Attention

Causal mask와 padding mask는 Transformer에서 자주 함께 사용되지만, 목적이 다릅니다:

- **Causal Mask**:
  - 미래 토큰을 차단하여 인과적 관계를 유지.
  - 주로 디코더에서 사용.
  - Shape: (S, S), 시퀀스 길이 S에 대해 상삼각 행렬.
- **Padding Mask** (`key_padding_mask`):
  - 패딩 토큰(예: `<pad>`)을 어텐션 계산에서 제외.
  - 인코더와 디코더 모두에서 사용.
  - Shape: (N, S), 배치 크기 N, 시퀀스 길이 S.

Decoder의 Masked Multi-head Self Attention

Decoder의 look-ahead mask 예시

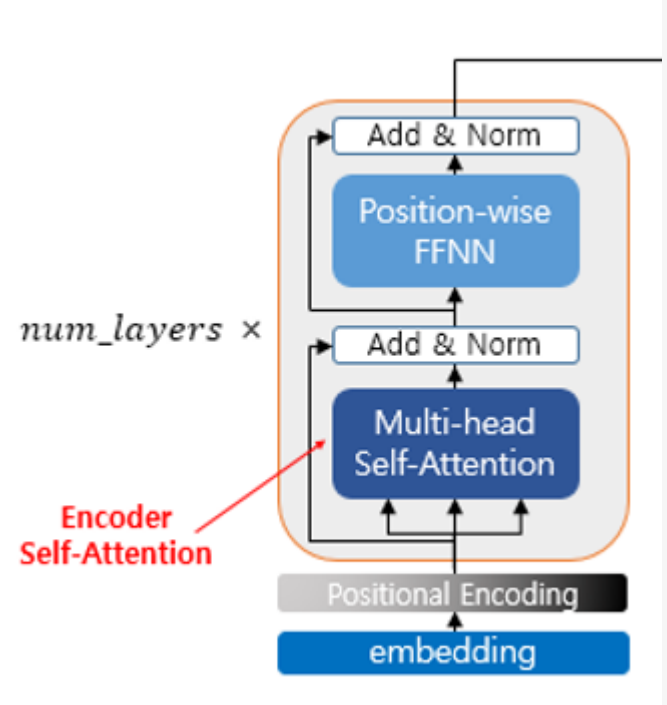
causal mask(look-ahead mask) vs padding mask

- **Causal Mask**:
  - 미래 토큰을 차단하여 인과적 관계를 유지.
  - 주로 디코더에서 사용.
  - Shape: (S, S), 시퀀스 길이 S에 대해 상삼각 행렬.
- **Padding Mask** (`key_padding_mask`):
  - 패딩 토큰(예: `<pad>`)을 어텐션 계산에서 제외.
  - 인코더와 디코더 모두에서 사용.
  - Shape: (N, S), 배치 크기 N, 시퀀스 길이 S.

트랜스포머(Transformer)

Transformer 적용

pytorch transformer

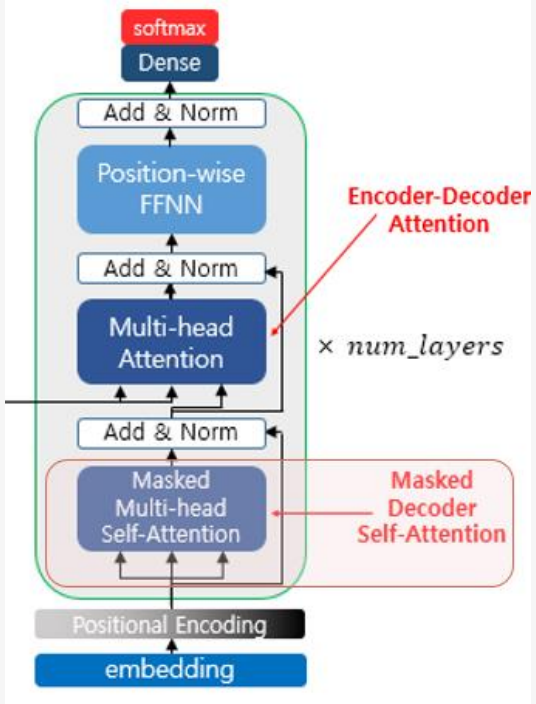



```
def forward(self, src: Tensor, src_mask: Optional[Tensor] = None,
            src_key_padding_mask: Optional[Tensor] = None,
            is_causal: Optional[bool] = None) -> Tensor:
    x = src
    if self.norm_first:
        x = x + self._sa_block(self.norm1(x), src_mask, src_key_padding_mask, is_causal)
        x = x + self._ff_block(self.norm2(x))
    else:
        x = self.norm1(x + self._sa_block(x, src_mask, src_key_padding_mask, is_causal))
        x = self.norm2(x + self._ff_block(x))

    return x

def _sa_block(self, x: Tensor, attn_mask: Optional[Tensor], key_padding_mask: Optional[Tensor],
              is_causal: Optional[bool]) -> Tensor:
    x = self.self_attn(x, x, x,
                       attn_mask=attn_mask,
                       key_padding_mask=key_padding_mask,
                       need_weights=False,
                       is_causal=is_causal)[0]
    return self.dropout1(x)

def _ff_block(self, x: Tensor) -> Tensor:
    x = self.linear2(self.dropout(self.activation(self.linear1(x))))
    return self.dropout2(x)
```

트랜스포머(Transformer)

Transformer 적용

pytorch transformer

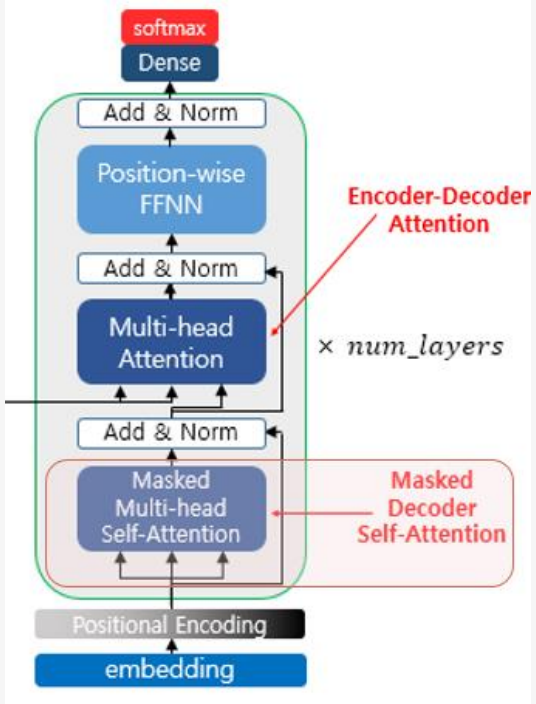



```
def forward(self, tgt: Tensor, memory: Tensor, tgt_mask: Optional[Tensor] = None,
            memory_mask: Optional[Tensor] = None, tgt_key_padding_mask: Optional[Tensor] = None,
            memory_key_padding_mask: Optional[Tensor] = None,
            tgt_is_causal: Optional[bool] = None, memory_is_causal: bool = False) -> Tensor:
    x = tgt
    if self.norm_first:
        # Self-attention block
        x = x + self._sa_block(self.norm1(x), tgt_mask, tgt_key_padding_mask, tgt_is_causal)
        # Multihead-attention block (cross-attention)
        x = x + self._mha_block(self.norm2(x), memory, memory_mask, memory_key_padding_mask, memory_is_causal)
        # Feedforward block
        x = x + self._ff_block(self.norm3(x))
    else:
        # Self-attention block
        x = self.norm1(x + self._sa_block(x, tgt_mask, tgt_key_padding_mask, tgt_is_causal))
        # Multihead-attention block (cross-attention)
        x = self.norm2(x + self._mha_block(x, memory, memory_mask, memory_key_padding_mask, memory_is_causal))
        # Feedforward block
        x = self.norm3(x + self._ff_block(x))

    return x
```

트랜스포머(Transformer)

Transformer 적용

pytorch transformer




```
def _sa_block(self, x: Tensor, attn_mask: Optional[Tensor], key_padding_mask: Optional[Tensor],
              is_causal: Optional[bool]) -> Tensor:
    x = self.self_attn(x, x, x,
                       attn_mask=attn_mask,
                       key_padding_mask=key_padding_mask,
                       need_weights=False,
                       is_causal=is_causal)[0]
    return self.dropout1(x)

def _mha_block(self, x: Tensor, mem: Tensor, attn_mask: Optional[Tensor],
               key_padding_mask: Optional[Tensor], is_causal: bool) -> Tensor:
    x = self.multihead_attn(x, mem, mem,
                            attn_mask=attn_mask,
                            key_padding_mask=key_padding_mask,
                            need_weights=False,
                            is_causal=is_causal)[0]
    return self.dropout2(x)

def _ff_block(self, x: Tensor) -> Tensor:
    x = self.linear2(self.dropout(self.activation(self.linear1(x))))
    return self.dropout3(x)
```

트랜스포머(Transformer)

Transformer 적용

transformer 실습

```
def forward(self, src_input_ids, tgt_input_ids, src_attention_mask, tgt_attention_mask):
    src_embedded = self.src_embedding(src_input_ids) * torch.sqrt(torch.tensor(self.d_model, dtype=torch.float))
    tgt_embedded = self.tgt_embedding(tgt_input_ids) * torch.sqrt(torch.tensor(self.d_model, dtype=torch.float))

    # 변경: 위치 인코딩 추가
    src_embedded = src_embedded + self.positional_encoding[:, :src_input_ids.size(1), :].to(self.device)
    tgt_embedded = tgt_embedded + self.positional_encoding[:, :tgt_input_ids.size(1), :].to(self.device)

    src_embedded = src_embedded.permute(1, 0, 2)  # (batch_size, seq_len, d_model) -> (seq_len, batch_size, d_model)
    tg
    #
    tg
    sr
    tgt_key_padding_mask = ~tgt_attention_mask.bool()

    # 변경: Transformer 모델 호출
    output = self.transformer(
        src=src_embedded,
        tgt=tgt_embedded,
        tgt_mask=tgt_mask,
        src_key_padding_mask=src_key_padding_mask,
        tgt_key_padding_mask=tgt_key_padding_mask
    )

    output = output.permute(1, 0, 2)  # (seq_len, batch_size, d_model) -> (batch_size, seq_len, d_model)
    output = self.fc(output)
    return output
```

5주차_transformer.ipynb