Transformer

20200661 고현민

20240371 지은수

Intro

대-명사(代名詞)

편집하기 편집 금지 요청

발음	[대:명사]
품사	「명사」

「003」 어떤 속성을 대표적으로 나타내는 것을 비유적으로 이르는 말.

- 황금은 부를 상징하는 **대명사이다**.
- 얼마 전까지만 해도 우리가 사는 집으로 통하던 한옥이 이젠 아파트에게 확고부동하던 주거의 **대명사** 자리를 내주고 있다.
- 우리 속담에 '가난한 사람 제사 돌아오듯' 어려운 일이 자주 닥친다고 하였는데, 이때 제사라는 것은 어려운 일이란 말의 **대명사다**.≪김흥규 외, 왜 사냐고 물으면, 한샘 출판 주식회사, 1993년≫
- 경지 정리 이전의 서 마지기란 힘깨나 쓰는 독농가(篤農家)나 지주만이 소유할 수 있었던 부의 상징인 동시에, 최소한 6명 이상의 품앗이나 두레꾼이 하루 종일 걸려도 모내기에 벅찬 큰 논배미의 **대명사로** 통했다.≪천규석, 이 땅덩이와 밥상, 창작과 비평사, 1993년≫

Intro

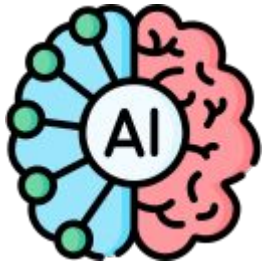
Attention is all you need

A Vaswani, N Shazeer, N Parmar... - Advances in neural ..., 2017 - proceedings.neurips.cc

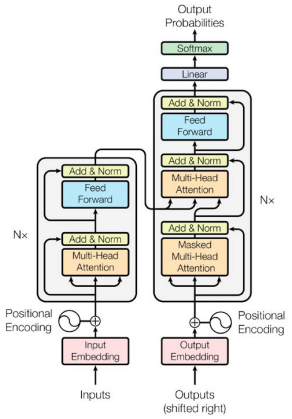
... to attend to **all** positions in the decoder up to and including that position. **We need** to prevent ... **We** implement this inside of scaled dot-product **attention** by masking out (setting to $-\infty$) ...

☆ 저장 99 인용 194312회 인용 관련 학술자료 전체 70개의 버전 »







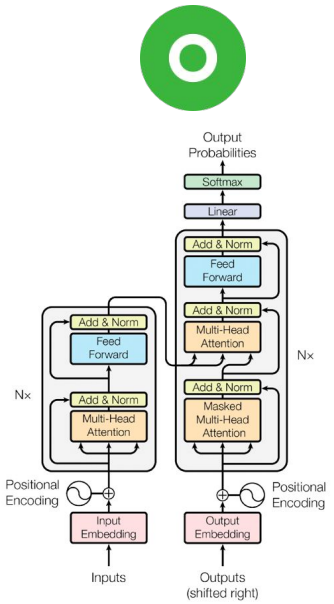


Intro

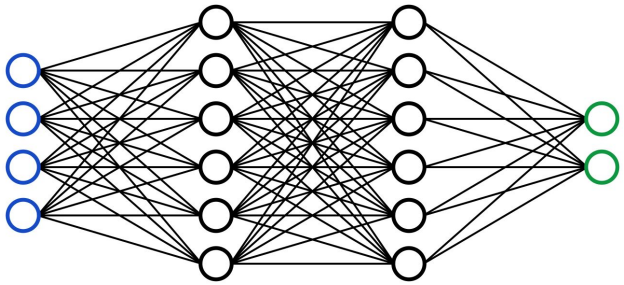
Q. 트랜스포머
아세요?



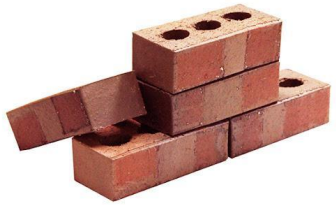




Neural Network



Neurons



Weights & Bias

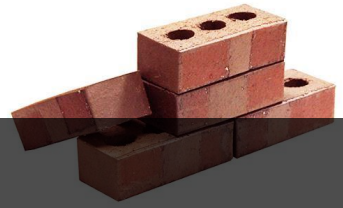


Activation
Function



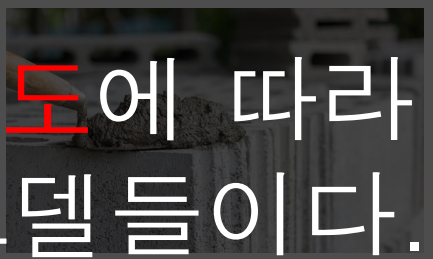
Neural Network

Neurons



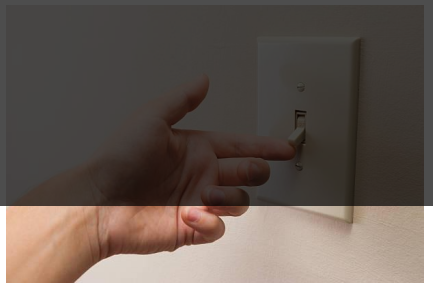


Weights & Bias



이러한 기본 재료들을 가지고 실제 정교한 설계도에 따라 쌓아 올린 하나의 거대한 건축물이 바로 실제 모델들이다.

Activation Function



Deep Learning



Sequential Data Processing

Computer Vision

Generative AI

Reinforcement Learning

정교한 설계도 (아키텍처)

Recurrent Neural Network

Convolutional Neural Network

Generative Adversarial Network

거대한 건축물 (실제 모델)









Deep Learning





Sequential Data Processing

Computer Vision

Generative AI

Reinforcemen t Learning

정교한 설계도 (아키텍처)

Recurrent Neural Network

Convolutional Neural Network

Generative Adversarial Network

거대한 건축물 (실제 모델)



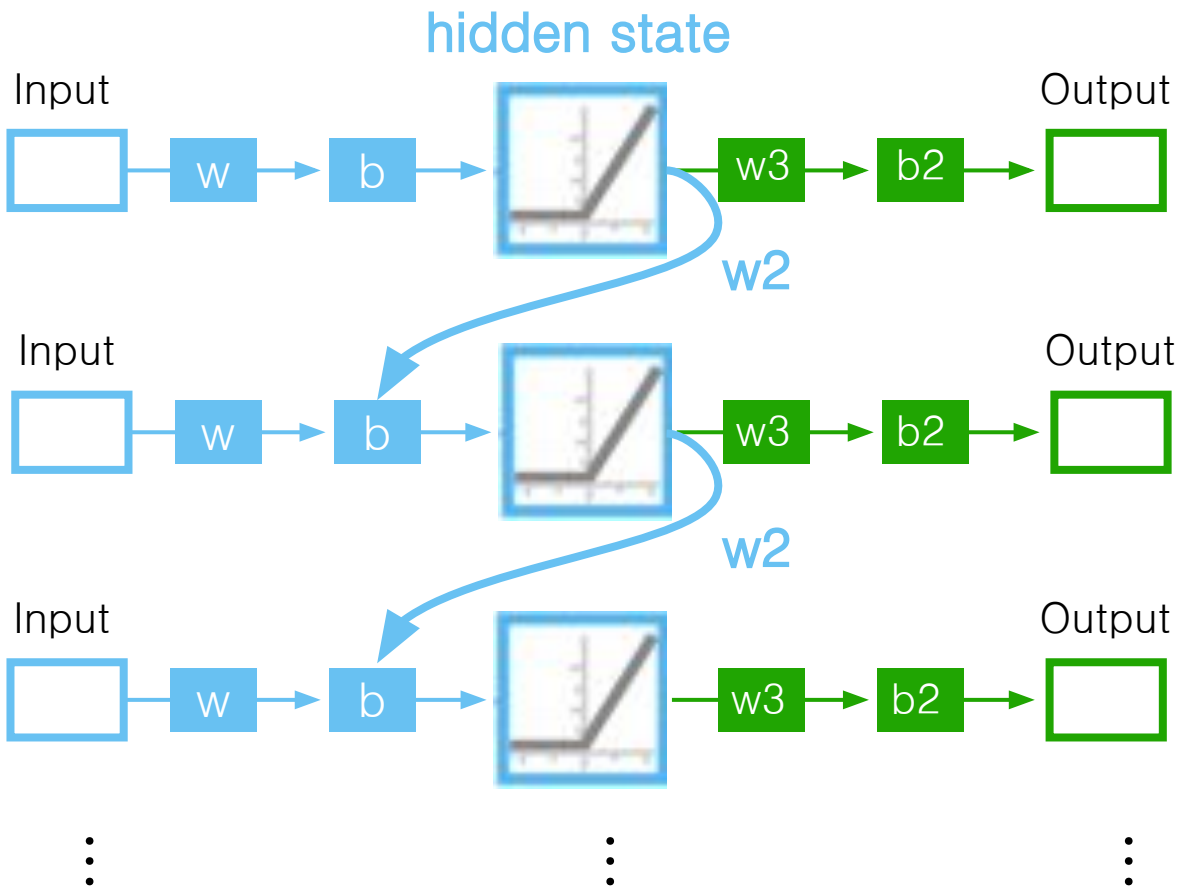






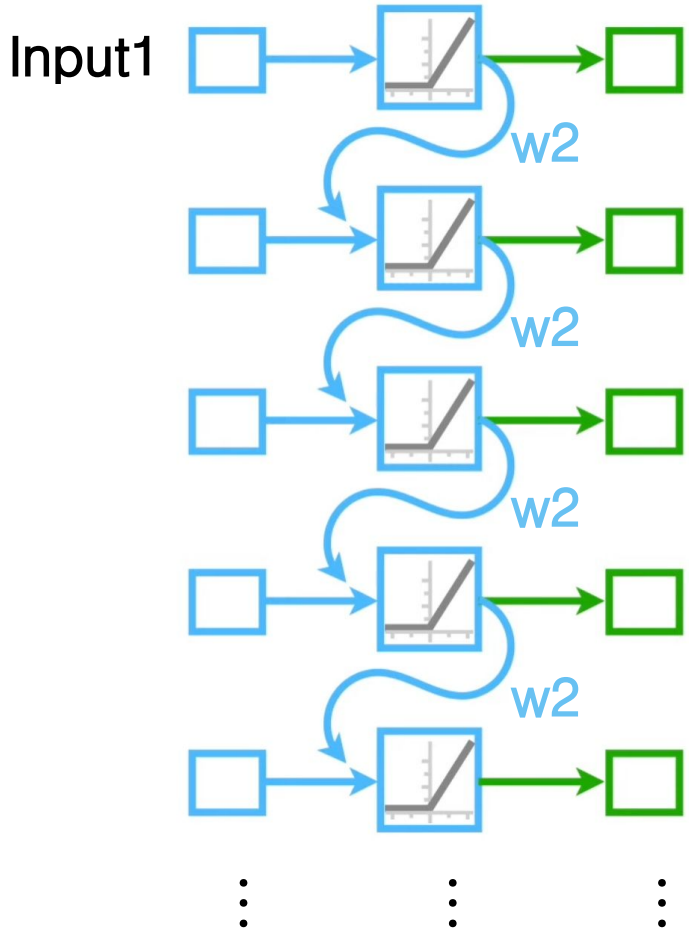
Before the Transformer

① RNNs (Recurrent Neural Networks)



Before the Transformer

① RNNs (Recurrent Neural Networks)



① w2 < 1일 때,

Input1 * w2 * w2 * w2 * w2 …

= Input1 * $w2^{\#unroll}$

= Input1 * NUMBER CLOSE TO 0

⇒ The Vanishing Gradient Prob.

② w2 > 1일 때,

Input1 * w2 * w2 * w2 * w2 …

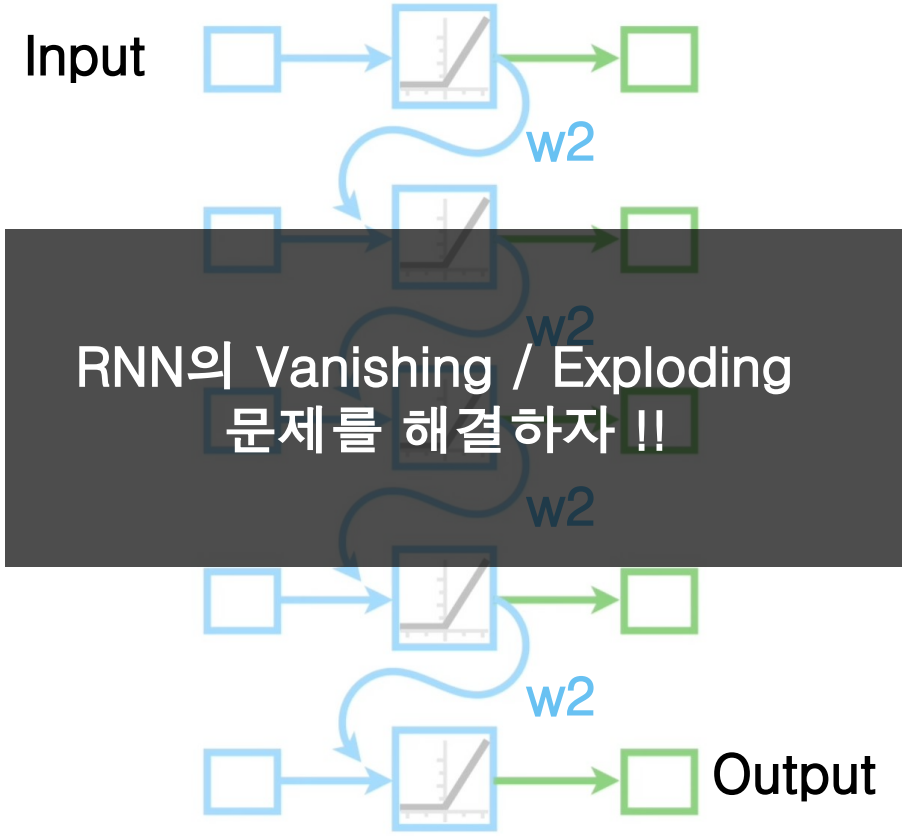
= Input1 * $w2^{\#unroll}$

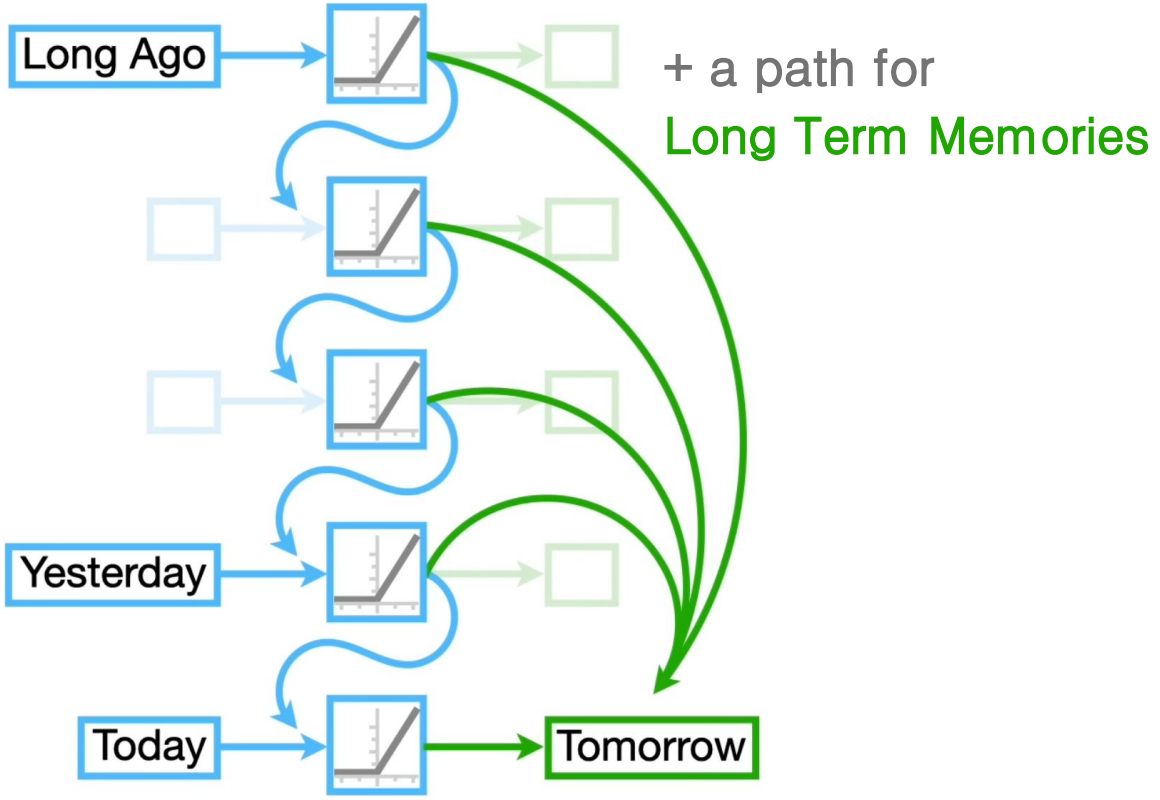
= Input1 * HUGE NUMBER

⇒ The Exploding Gradient Prob.

Before the Transformer

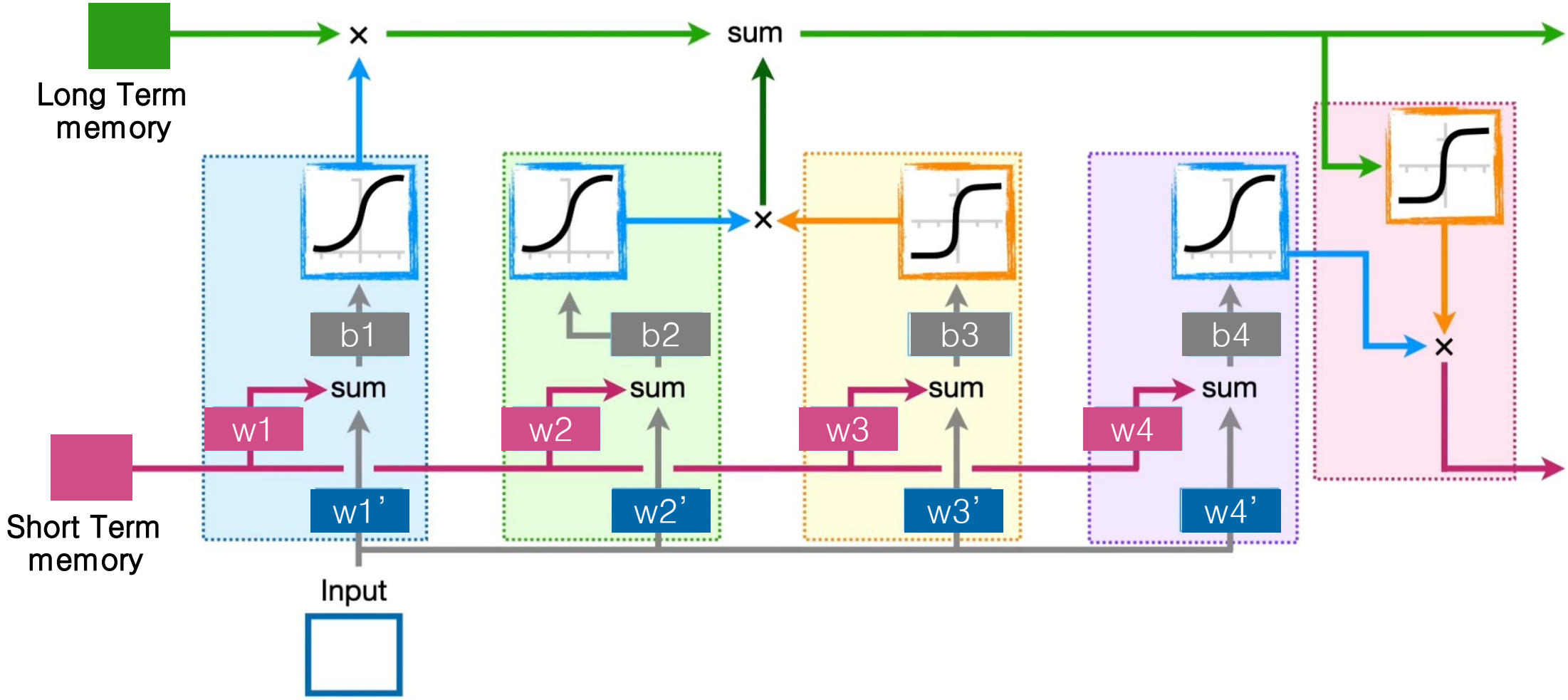
② LSTM (Long Short-Term Memory)





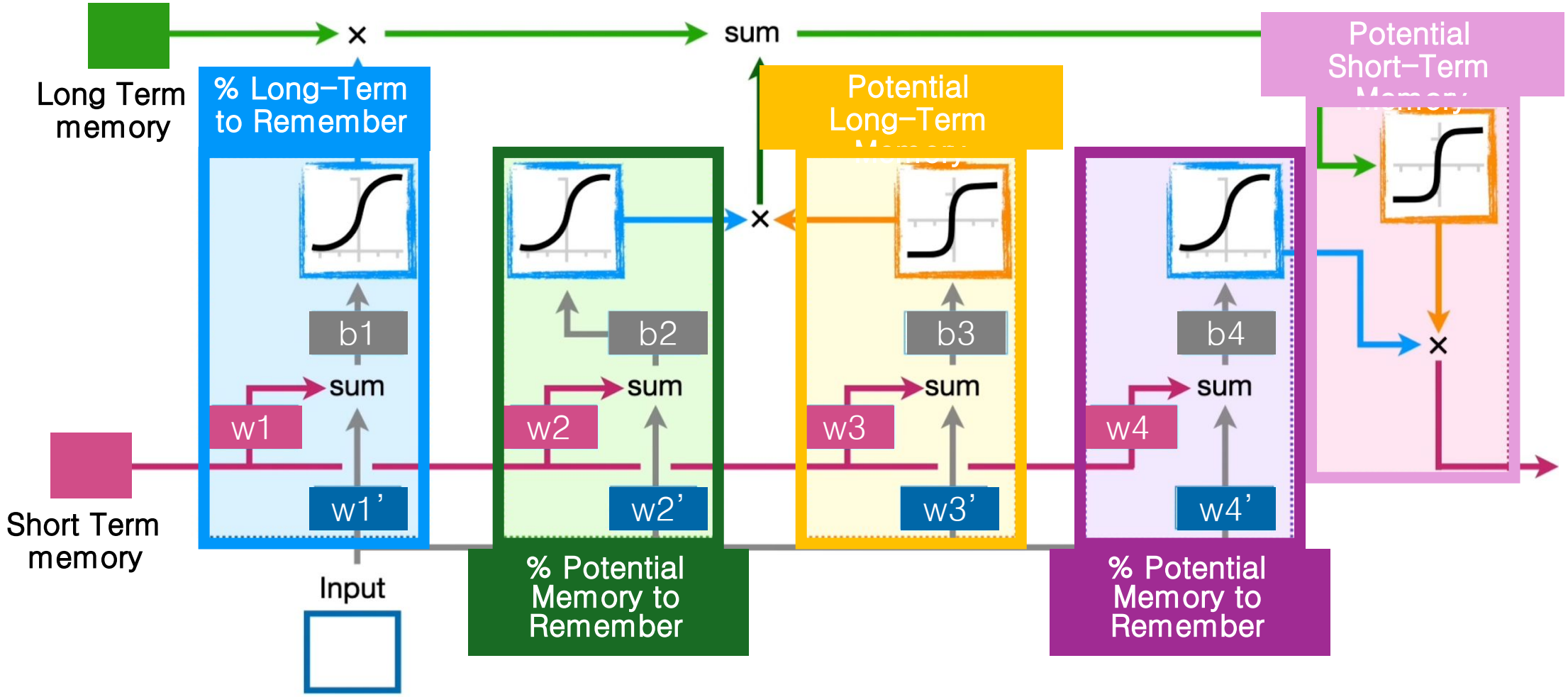
Before the Transformer

② LSTM (Long Short-Term Memory)



Before the Transformer

② LSTM (Long Short-Term Memory)



Trials before the Transformer

about **Attention !!**

① Bahdanau Attention (2014)

② Luong Attention (2015)

③ Document Classification (HAN)

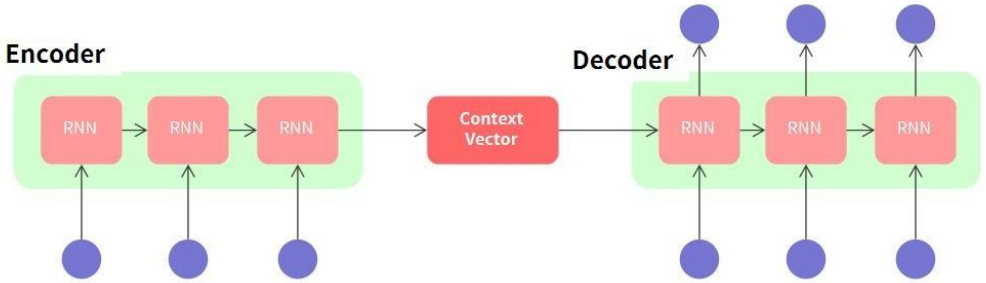
④ Sentence Embedding (Self-Attention on LSTM)

순차적 데이터 처리의 한계

개선을 위한 새로운 개념 등장!

Attention

Attention

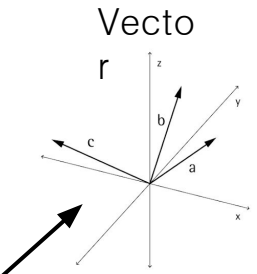


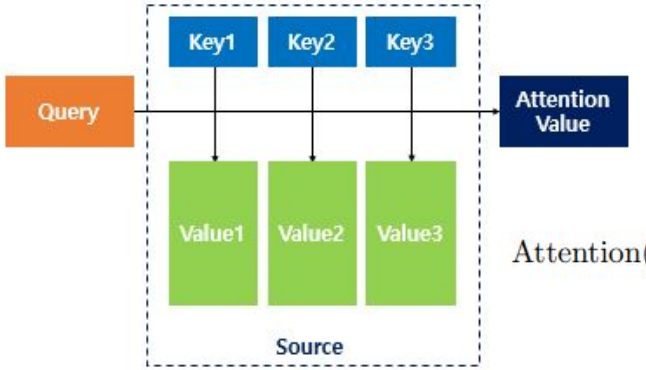
Transformer 이전에도 이미 있던 개념!
RNN의 정보 손실 문제를 보정해주기 위해 나온 기법!

기본 아이디어

- 디코더에서 출력 단어를 예측하는 매 시점(time step)마다, 인코더에서의 전체 입력 문장을 다시 한 번 참고
- 단, 전체 입력 문장을 전부 다 동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야 할 단어와 연관이 있는 입력 단어 부분을 좀 더 집중(attention)해서 봄

Attention



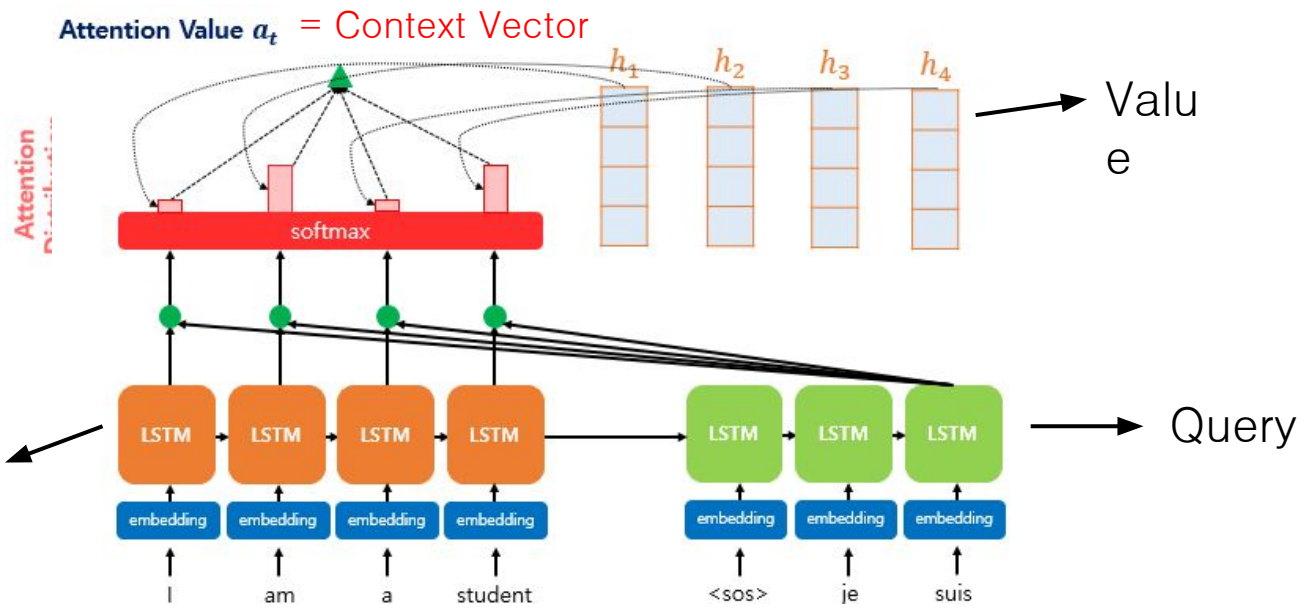


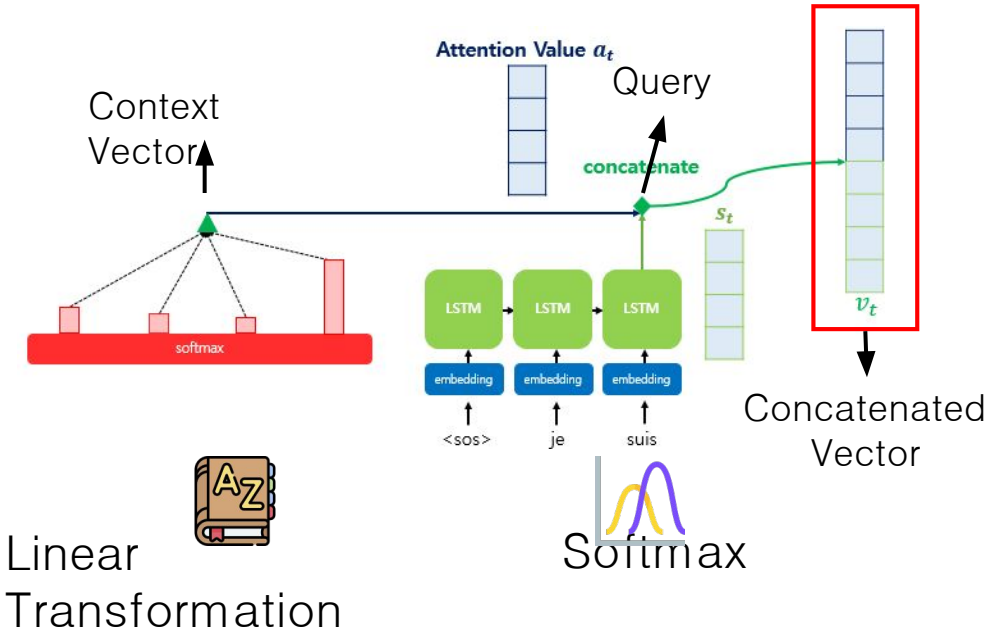
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention의 핵심 요소

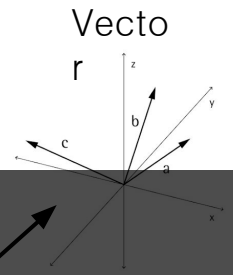
- Query : 구체적인 질문 = Input Embedding Vector * W_Q
- Key : 각 단어들의 정보를 알려주는 이름표 = Input Embedding Vector * W_K
- Value : 각 단어들의 실제 value의 내용을 담음 = Input Embedding Vector * W_V

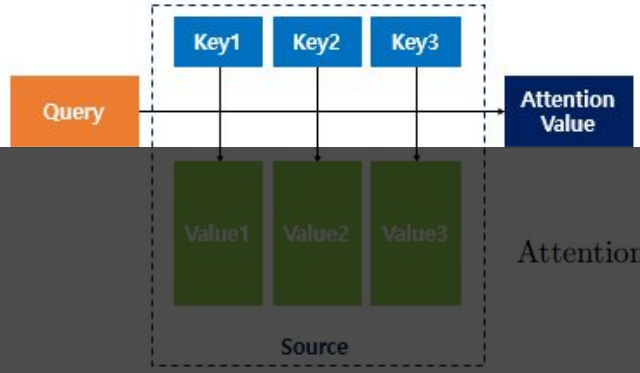
W_Q, W_K, W_V: 가중치 행렬(학습되면서 최적화)





Attention



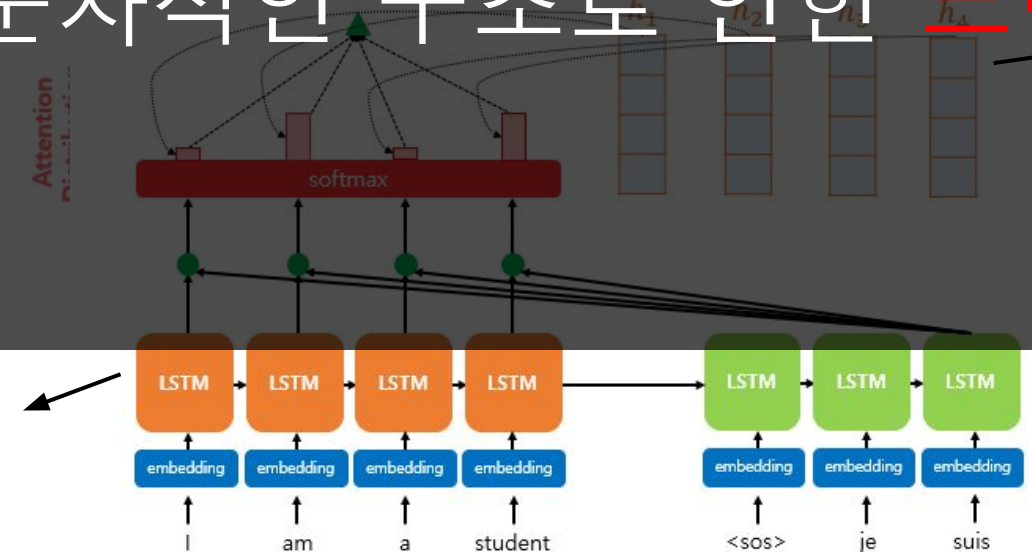


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention의 핵심 요소

- Query : 구체적인 질문 = Input Embedding Vector * W_Q
- Key : 각 단어들의 정보를 알려주는 이름표 = Input Embedding Vector * W_K
- Value : 각 단어들의 실제 의미 내용을 담은 값 = Input Embedding Vector * W_V (학습을 통해 W_Q, W_K, W_V를 가장 적절한 값으로 최적화)

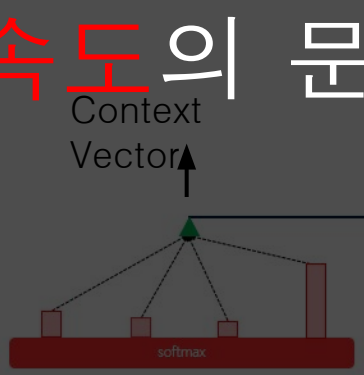
Attention은 정보 손실 문제를 해결했지만,
순차적인 구조로 인한 느린 학습 속도의 문제는 여전함

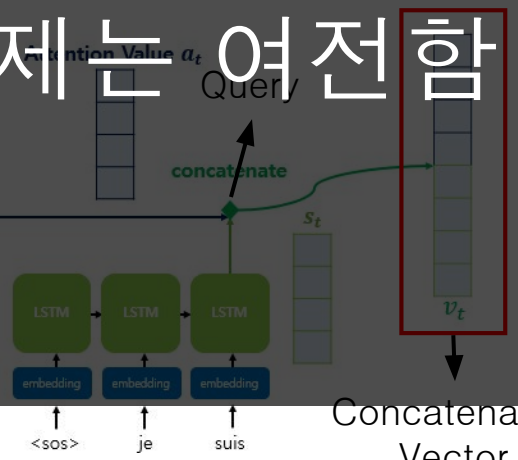


Key

Value

Query





Concatenated Vector

Linear Transformation

Softmax

```python
class ScaleDotProductAttention(nn.Module):
    """
    compute scale dot product attention

    Query : given sentence that we focused on (decoder)
    Key : every sentence to check relationship with Qeury(encoder)
    Value : every sentence same with Key (encoder)
    """

    def __init__(self):
        super(ScaleDotProductAttention, self).__init__()
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, q, k, v, mask=None, e=1e-12):
        # input is 4 dimension tensor
        # [batch_size, head, length, d_tensor]
        batch_size, head, length, d_tensor = k.size()

        # 1. dot product Query with Key^T to compute similarity
        k_t = k.transpose(2, 3)  # transpose
        score = (q @ k_t) / math.sqrt(d_tensor)  # scaled dot product

        # 2. apply masking (opt)
        if mask is not None:
            score = score.masked_fill(mask == 0, -10000)

        # 3. pass them softmax to make [0, 1] range
        score = self.softmax(score)

        # 4. multiply with Value
        v = score @ v

        return v, score
```

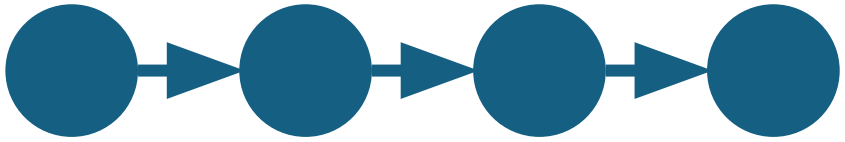
problem solver: the Transformer

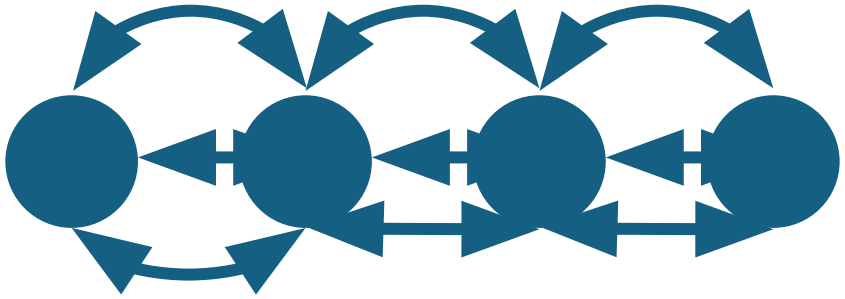
$$h_t = f(x_t, h_{t-1})$$

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

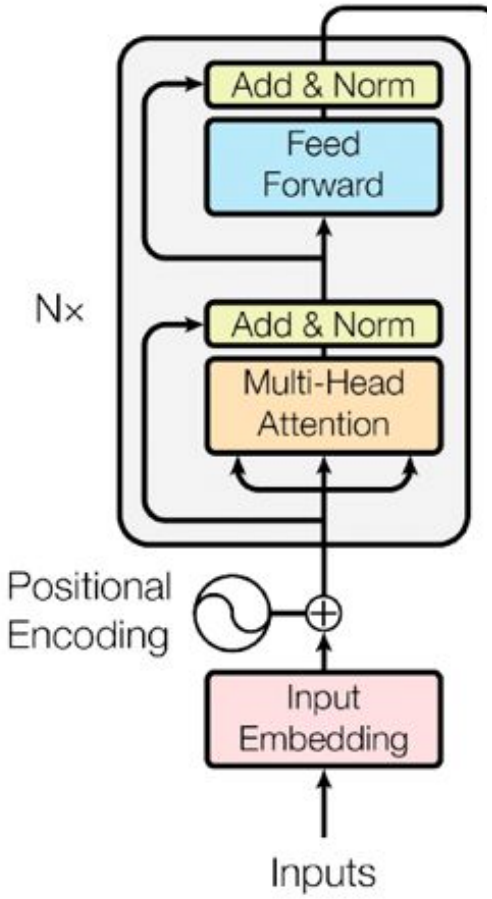
순차적 기억의 전달

관계 기반 집중





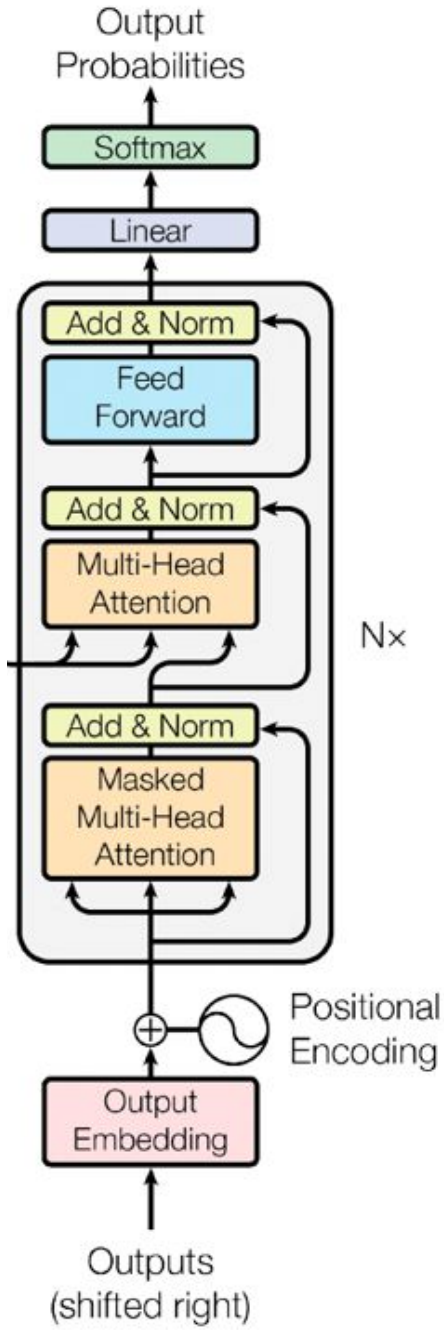
the Transformer



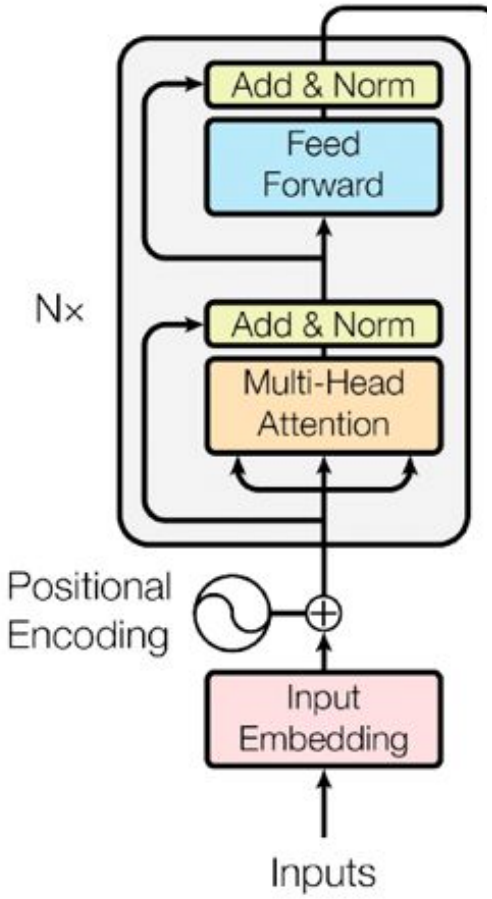
< The Encoder >

Similar but Different !

< The Decoder >

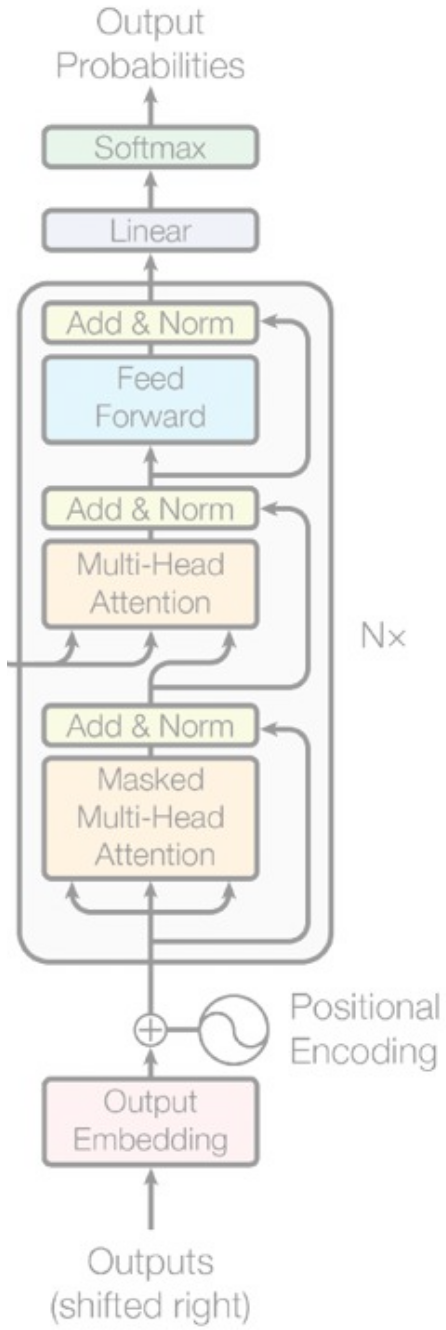


the Transformer

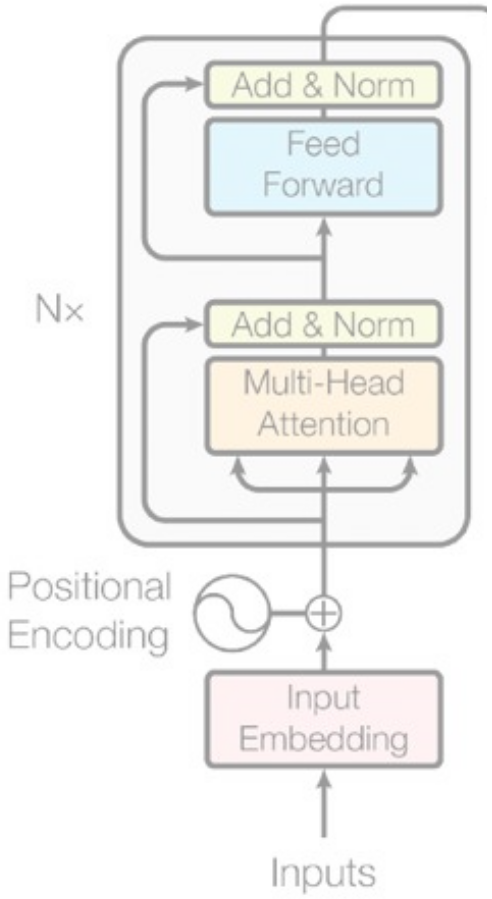


< The Encoder >

- ✔ Positional Encoding
- ✔ Multi-Head Self-Attention
- ✔ Add & Normalization
- ✔ Feed Forward

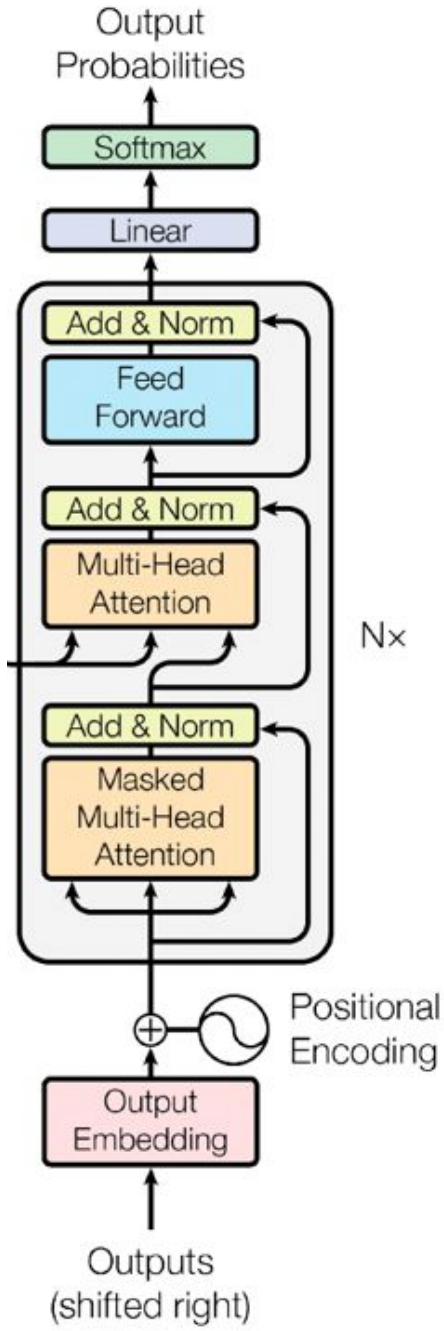


the Transformer

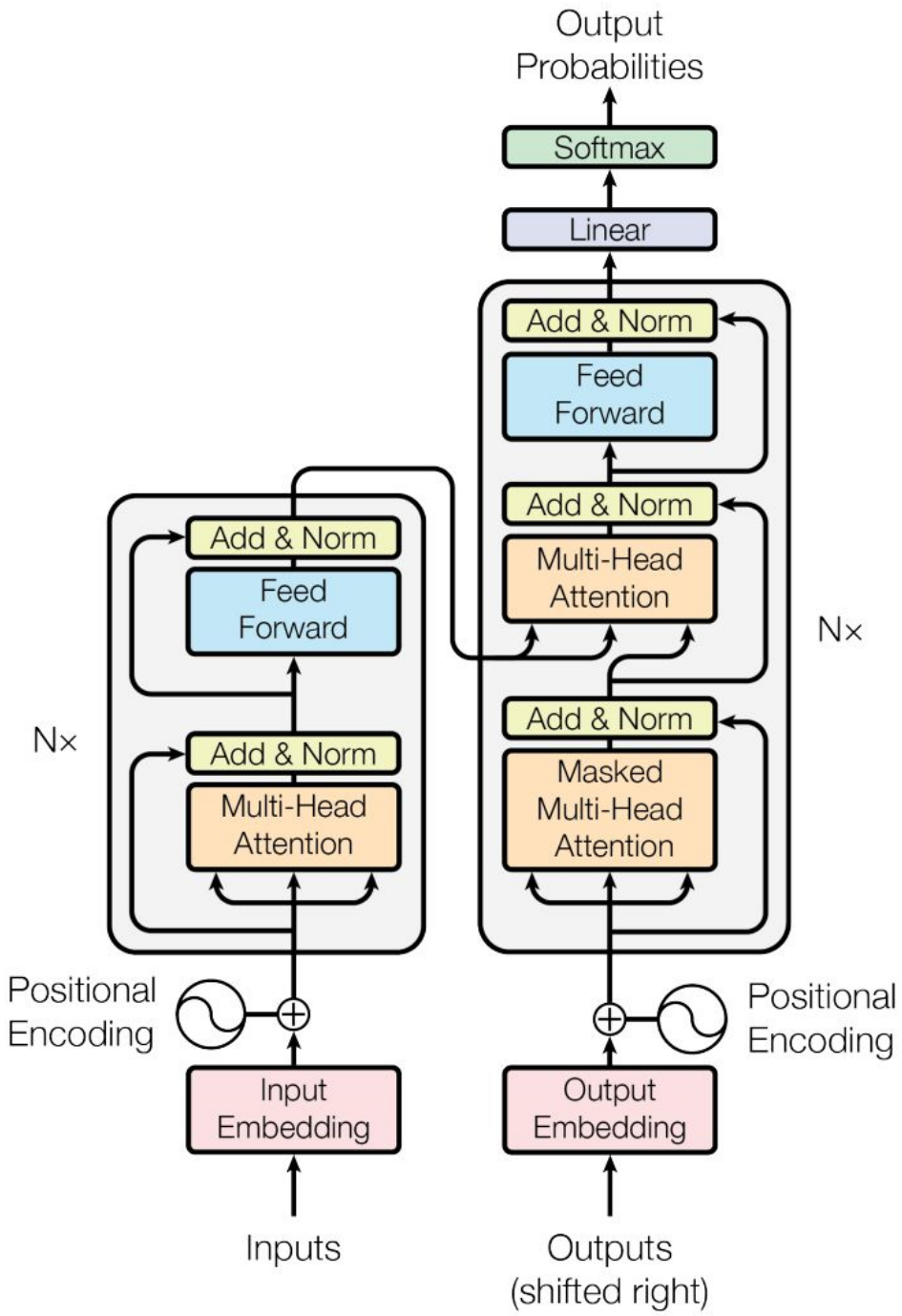


< The Decoder >

- ✔ Positional Encoding
- ✔ Masked Multi-Head Attention
- ✔ Add & Normalization
- ✔ Feed Forward
- ✔ Linear
- ✔ Softmax



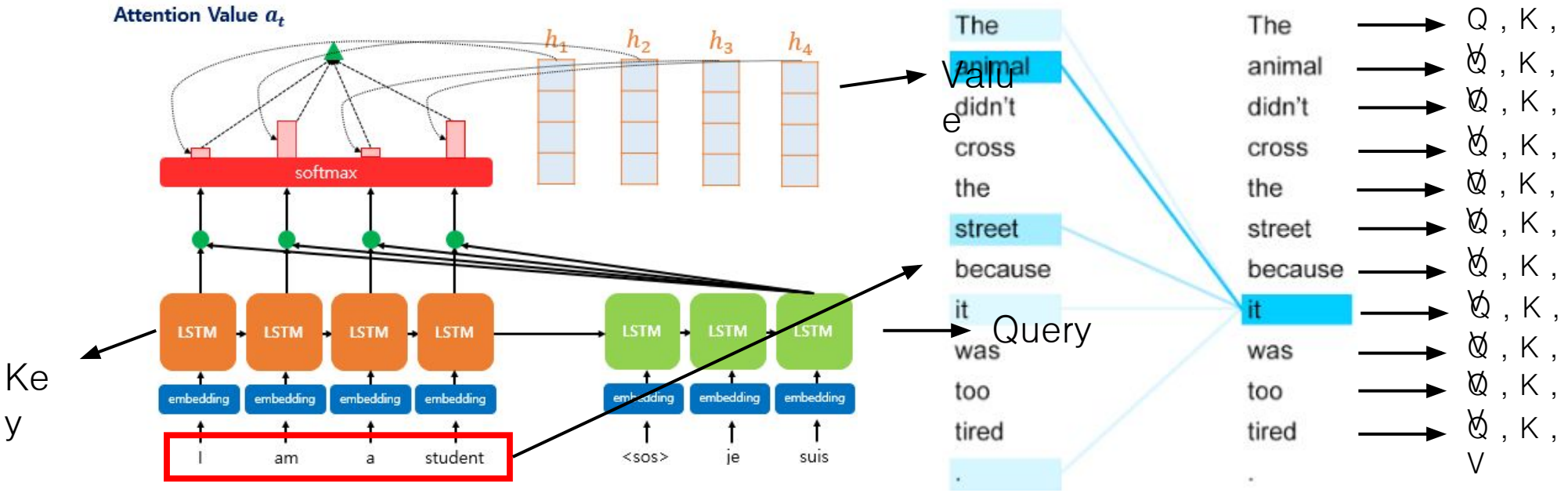
Self-Attention



Self-Attention

기존 Attention = Decoder가 Encoder를 참고하는 방식

Self-Attention = 문장 속 단어들끼리의 관계를 파악하는 방식



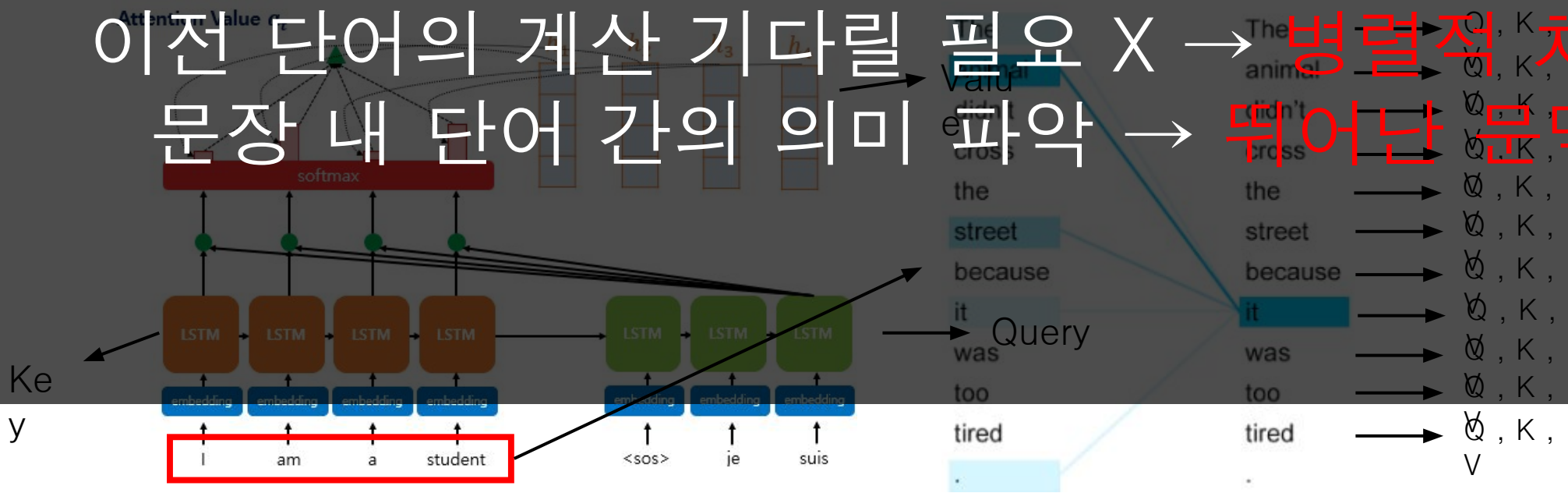
Self-Attention

기존 Attention = Decoder가 Encoder를 참고하는 방식

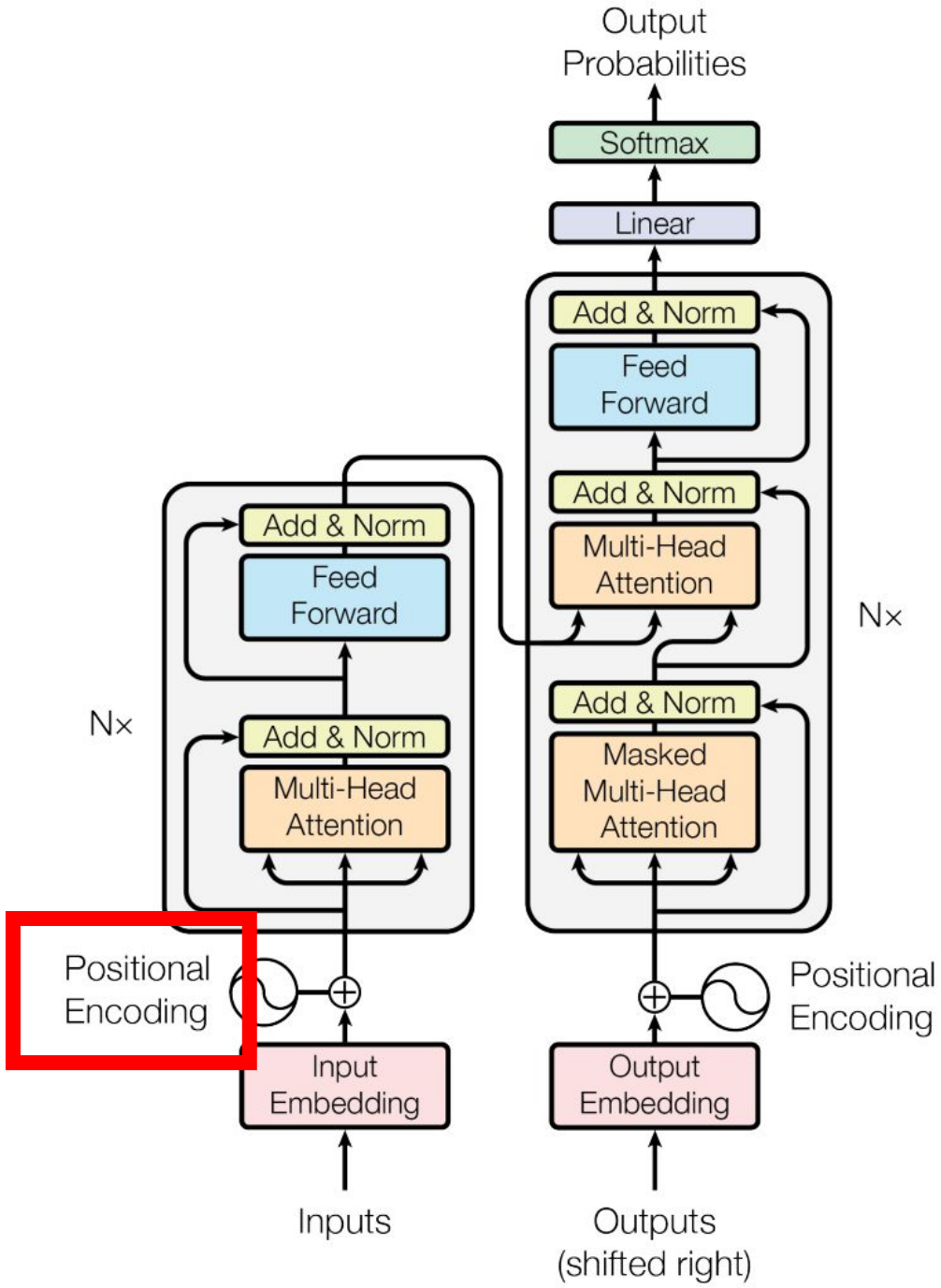
Self-Attention = 문장 속 단어들끼리의 관계를 파악하는 방식

이전 단어의 계산 기다릴 필요 X → 병렬적 처리 가능

문장 내 단어 간의 의미 파악 → 뛰어난 문맥 이해



Positional Encoding

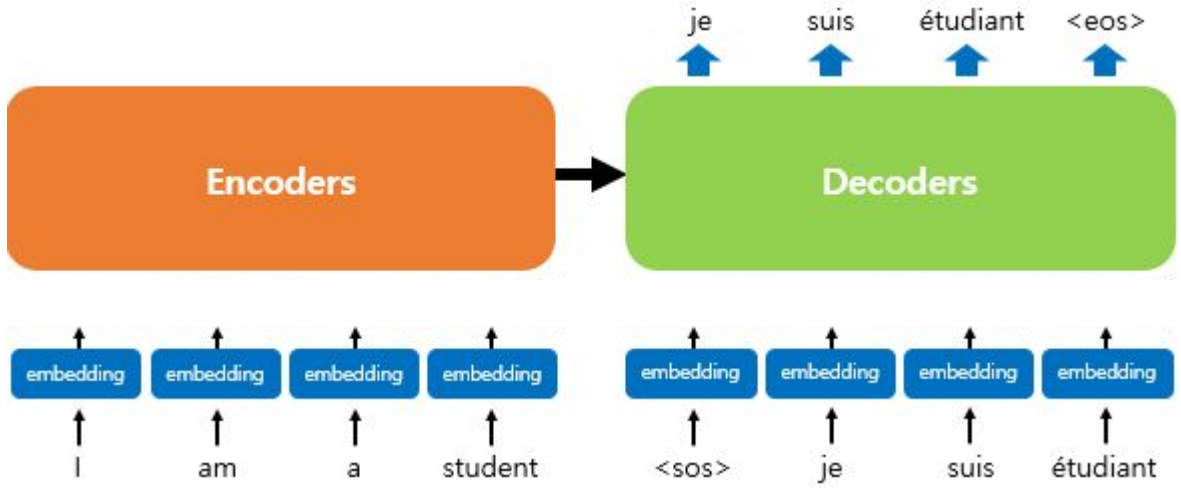


Positional Encoding

현민이가 강아지에게 물렸다.
강아지가 현민이에게 물렸다.

RNN = 순차적 입력 방식으로 인해 각 단어의 위치 정보 파악 용이!

Transformer = 병렬 방식의 도입으로 인해 해당 기능 부재…



pos: 위치를 나타내는 정수
i: 차원 인덱스
d_{model}: 임베딩 벡터 차원 크기

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

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



Why sin, cos?

왜 사인(sin)과 코사인(cos)을 사용할까? ✨

단순히 위치마다 다른 값을 주기 위함이라면 더 간단한 방법을 쓸 수도 있었을 텐데, 굳이 복잡해 보이는 삼각함수를 사용한 데에는 아주 중요한 이유가 있습니다.

핵심 아이디어: 상대적 위치 표현

가장 중요한 이유는 **상대적인 위치 관계를 모델이 쉽게 학습할 수 있도록** 하기 위함입니다.

`pos+k` 위치의 포지셔널 인코딩 값은 `pos` 위치의 포지셔널 인코딩 값을 **선형 변환(Linear Transformation)**하여 표현할 수 있습니다. 이는 삼각함수의 덧셈 정리(`sin(A+B)`, `cos(A+B)`)를 통해 증명할 수 있습니다.

이게 왜 중요하냐면, Self-Attention은 내부적으로 수많은 선형 변환과 내적(Dot Product)으로 이루어져 있습니다. 위치 정보를 선형 변환으로 표현할 수 있다는 것은, 어텐션 메커니즘이 ***현재 위치에서 k만큼 떨어진 위치의 정보를 가져와***와 같은 상대적인 위치 계산을 아주 쉽게 할 수 있다는 의미입니다. 모델은 모든 절대 위치를 외울 필요 없이, '상대적인 이동'이라는 개념 하나만 학습하면 됩니다.

구체적인 장점들

1. **고유한 위치 값 생성:** 각 위치(`pos`)마다 고유한 패턴의 벡터가 생성되어 절대적인 위치를 구분할 수 있습니다.
2. **일정한 범위의 값:** sin, cos 함수의 결과는 항상 -1과 1 사이의 값을 가집니다. 이는 포지셔널 인코딩 값이 너무 커져서 기존 단어 임베딩의 의미를 해치지 않도록 합니다 (수치적 안정성).
3. **긴 문장에 대한 일반화:** 모델이 훈련 과정에서 보지 못했던 더 긴 길이의 문장이 들어오더라도, 수식을 통해 얼마든지 위치 값을 계산하여 일반화할 수 있습니다.

```python
class PositionalEncoding(nn.Module):
    """
    compute sinusoid encoding.
    """
    def __init__(self, d_model, max_len, device):
        """
        constructor of sinusoid encoding class

        :param d_model: dimension of model
        :param max_len: max sequence length
        :param device: hardware device setting
        """
        super(PositionalEncoding, self).__init__()

        # same size with input matrix (for adding with input matrix)
        self.encoding = torch.zeros(max_len, d_model, device=device)
        self.encoding.requires_grad = False  # we don't need to compute gradient

        pos = torch.arange(0, max_len, device=device)
        pos = pos.float().unsqueeze(dim=1)
        # 1D => 2D unsqueeze to represent word's position

        _2i = torch.arange(0, d_model, step=2, device=device).float()
        # 'i' means index of d_model (e.g. embedding size = 50, 'i' = [0,50])
        # "step=2" means 'i' multiplied with two (same with 2 * i)

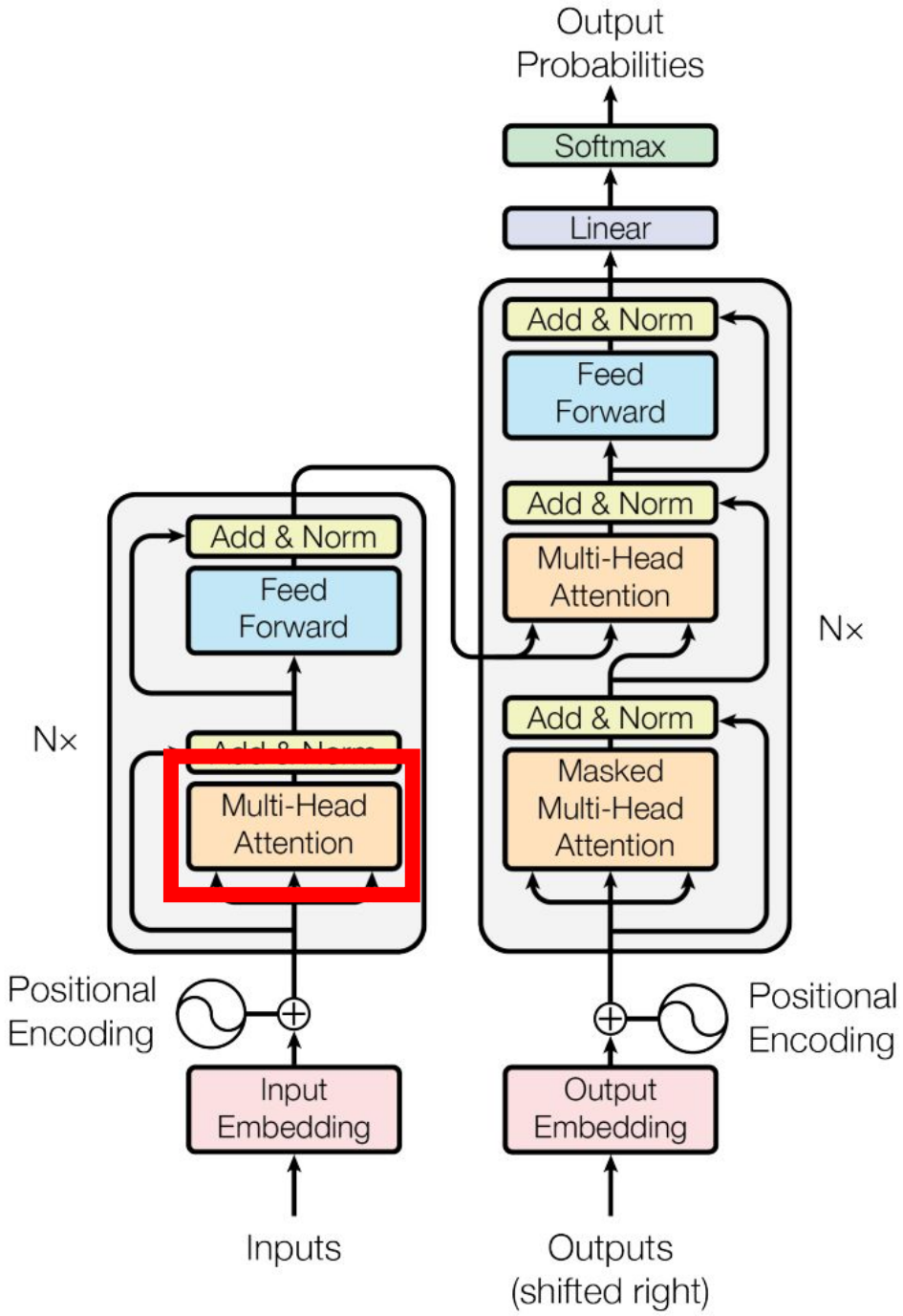
        self.encoding[:, 0::2] = torch.sin(pos / (10000 ** (_2i / d_model)))
        self.encoding[:, 1::2] = torch.cos(pos / (10000 ** (_2i / d_model)))
        # compute positional encoding to consider positional information of words

    def forward(self, x):
        # self.encoding
        # [max_len = 512, d_model = 512]

        batch_size, seq_len = x.size()
        # [batch_size = 128, seq_len = 30]

        return self.encoding[:seq_len, :]
        # [seq_len = 30, d_model = 512]
        # it will add with tok_emb : [128, 30, 512]
```

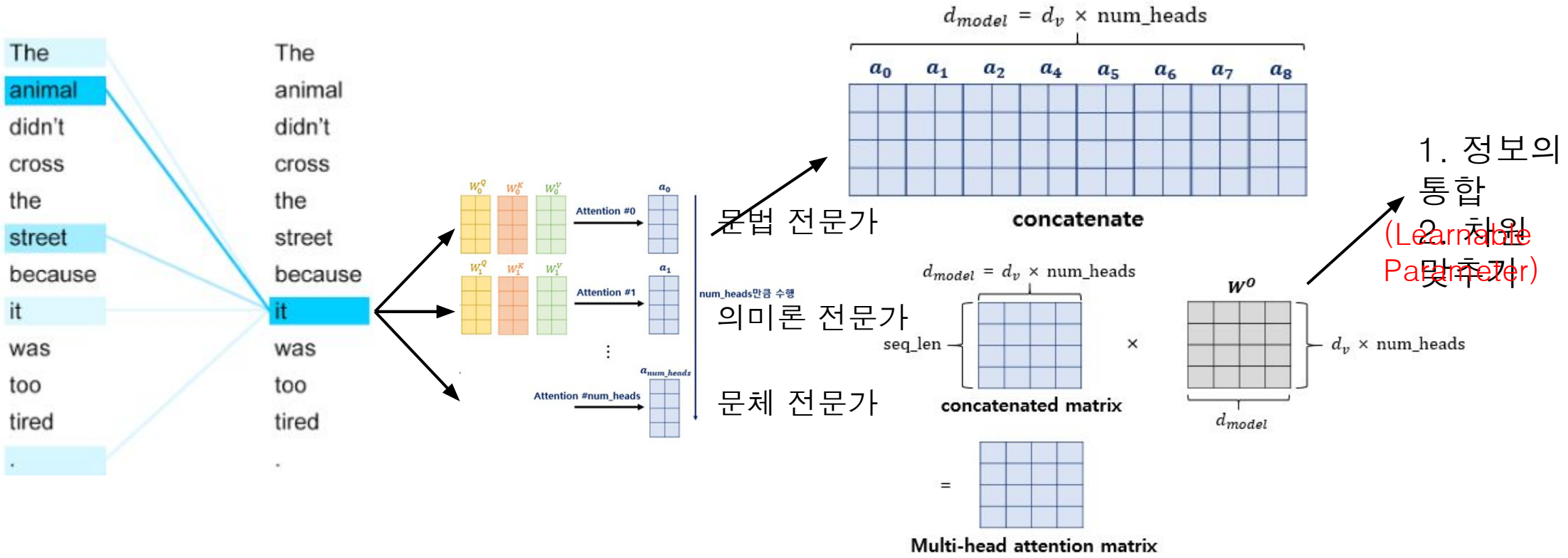
Multi-Head Attention



Multi-Head Attention

Self-Attention = 문장 분석 전문가 한 명에게 분석 맡기기

Multi-Head Attention = 여러 분야 전문가들에게 분석 맡기기



```python
class MultiHeadAttention(nn.Module):

    def __init__(self, d_model, n_head):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.attention = ScaleDotProductAttention()
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_concat = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        # 1. dot product with weight matrices
        q, k, v = self.w_q(q), self.w_k(k), self.w_v(v)

        # 2. split tensor by number of heads
        q, k, v = self.split(q), self.split(k), self.split(v)

        # 3. do scale dot product to compute similarity
        out, attention = self.attention(q, k, v, mask=mask)

        # 4. concat and pass to linear layer
        out = self.concat(out)
        out = self.w_concat(out)

        # 5. visualize attention map
        # TODO : we should implement visualization

        return out

    def split(self, tensor):
        """
        split tensor by number of head

        :param tensor: [batch_size, length, d_model]
        :return: [batch_size, head, length, d_tensor]
        """
        batch_size, length, d_model = tensor.size()

        d_tensor = d_model // self.n_head
        tensor = tensor.view(batch_size, length, self.n_head, d_tensor).transpose(1, 2)
        # it is similar with group convolution (split by number of heads)

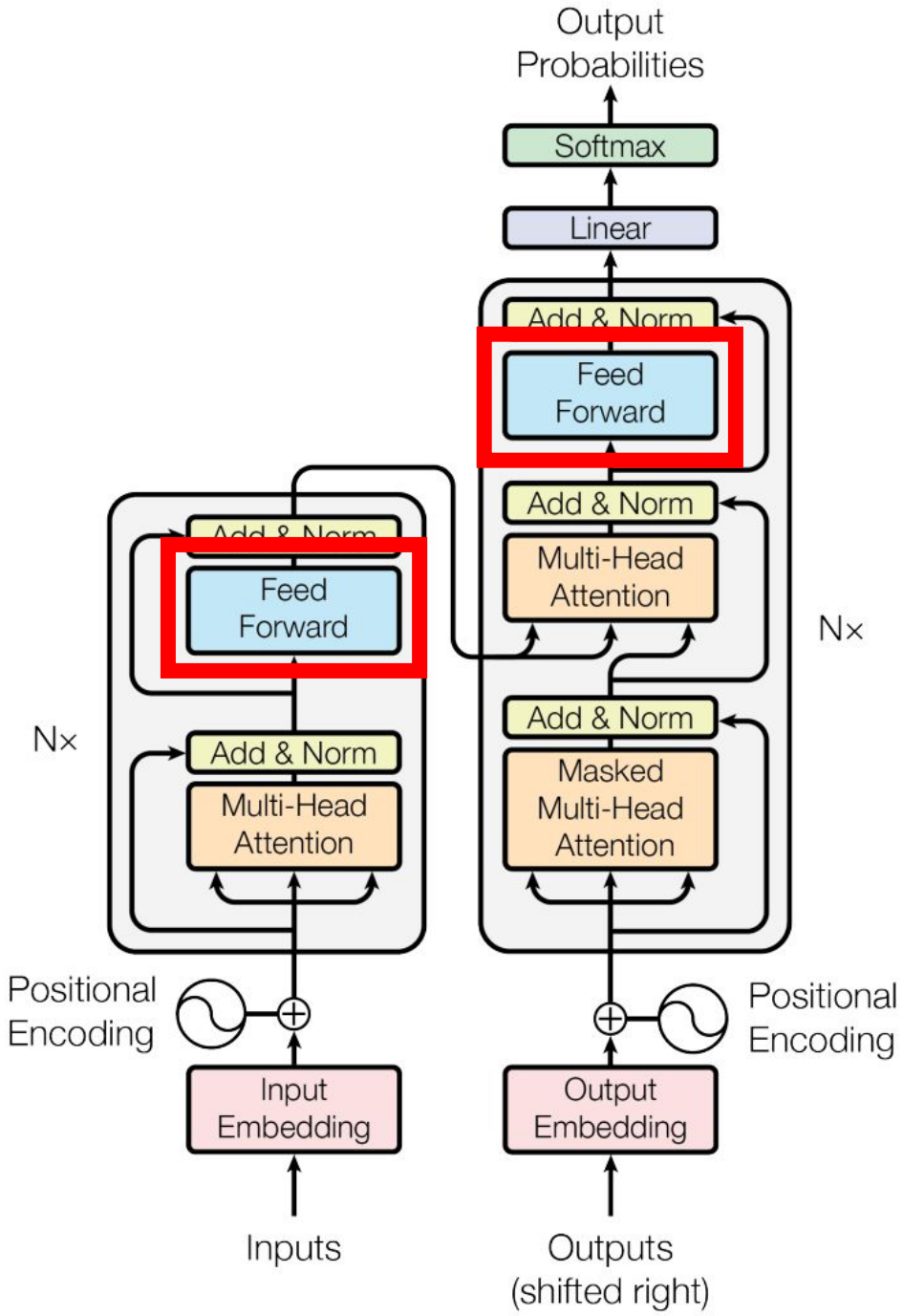
        return tensor

    def concat(self, tensor):
        """
        inverse function of self.split(tensor : torch.Tensor)

        :param tensor: [batch_size, head, length, d_tensor]
        :return: [batch_size, length, d_model]
        """
        batch_size, head, length, d_tensor = tensor.size()
        d_model = head * d_tensor

        tensor = tensor.transpose(1, 2).contiguous().view(batch_size, length, d_model)
        return tensor
```

Feed Forward



Feed Forward

Position-wise FFN $\mathbf{FFN}(x) = W_2\,\phi(W_1 x + b_1) + b_2$

$x \in \mathbb{R}^{d_{\text{model}}}$

① 각 토큰 벡터 입력

$W_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$

② 차원을 넓힘

$d_{\text{model}} \rightarrow d_{\text{ff}}$

ϕ

③ 비선형 변환

(ReLU, GELU 등)

$W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$

④ 원래 차원으로 축소

$d_{\text{ff}} \rightarrow d_{\text{model}}$



Feed Forward

Position-wise FFN $\mathbf{FFN}(x) = W_2\,\phi(W_1 x + b_1) + b_2$

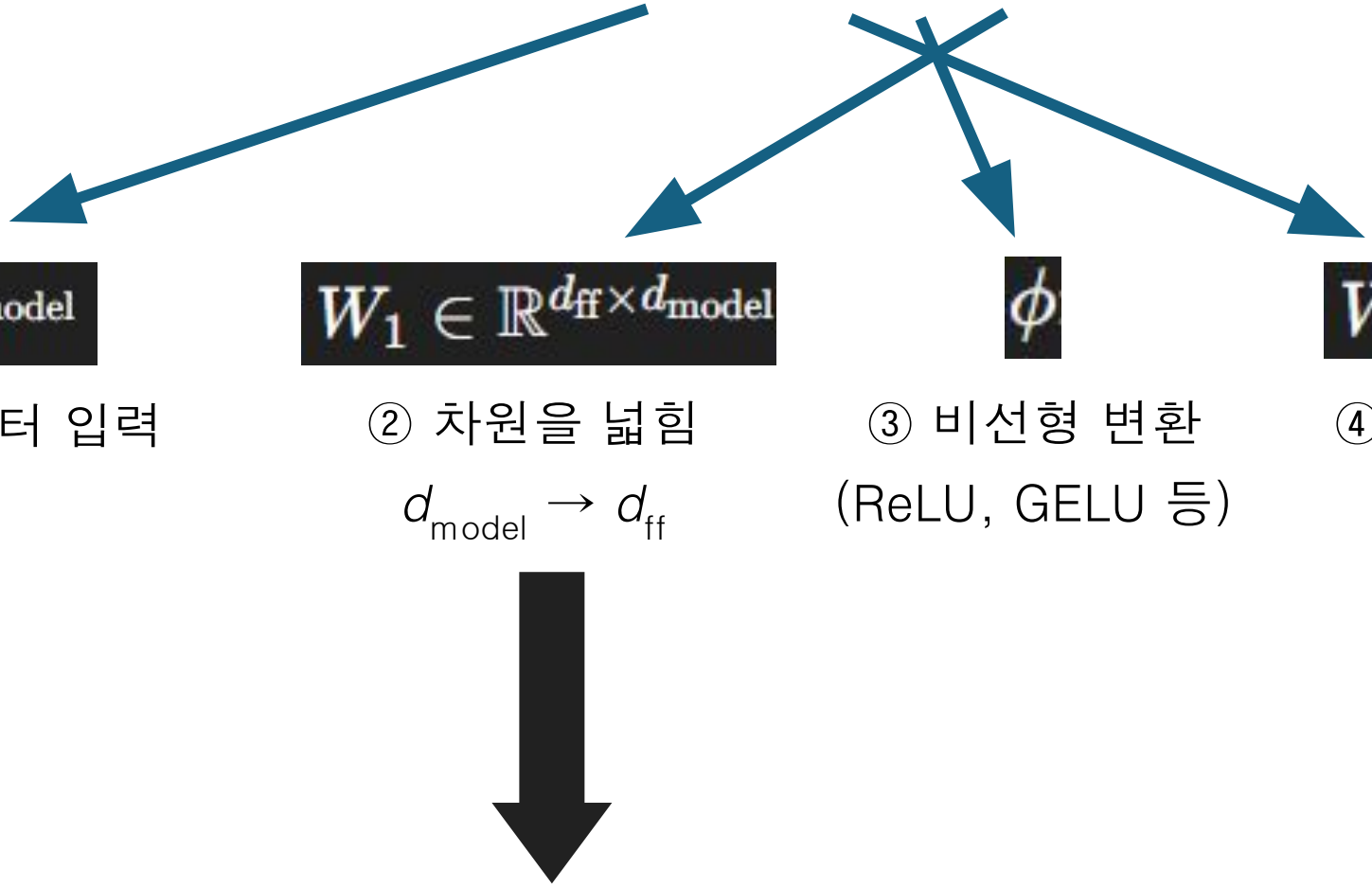
ϕ

비선형 변환

- ✔ 선형 결합 구조를 바꾸자!
- ✔ ReLU와 같은 비선형 활성화 함수를 가진 FNN을 통해 복잡한 패턴 학습
- ✔ 딥러닝의 핵심인 “선형변환 + 비선형변환”의 반복

Feed Forward

Position-wise FFN $\mathbf{FFN}(x) = W_2\,\phi(W_1 x + b_1) + b_2$



$x \in \mathbb{R}^{d_{\text{model}}}$

① 각 토큰 벡터 입력

$W_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$

② 차원을 넓힘

$d_{\text{model}} \rightarrow d_{\text{ff}}$

ϕ

③ 비선형 변환

(ReLU, GELU 등)

$W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$

④ 원래 차원으로 축소

$d_{\text{ff}} \rightarrow d_{\text{model}}$

Feed Forward

Position-wise FFN $\mathbf{FFN}(x) = W_2\,\phi(W_1 x + b_1) + b_2$

$W_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$

$W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$

✔ 입력 특징을 다양한 관점에서 변환하고 압축하여서 풍부한 표현력을 얻게 됨 !

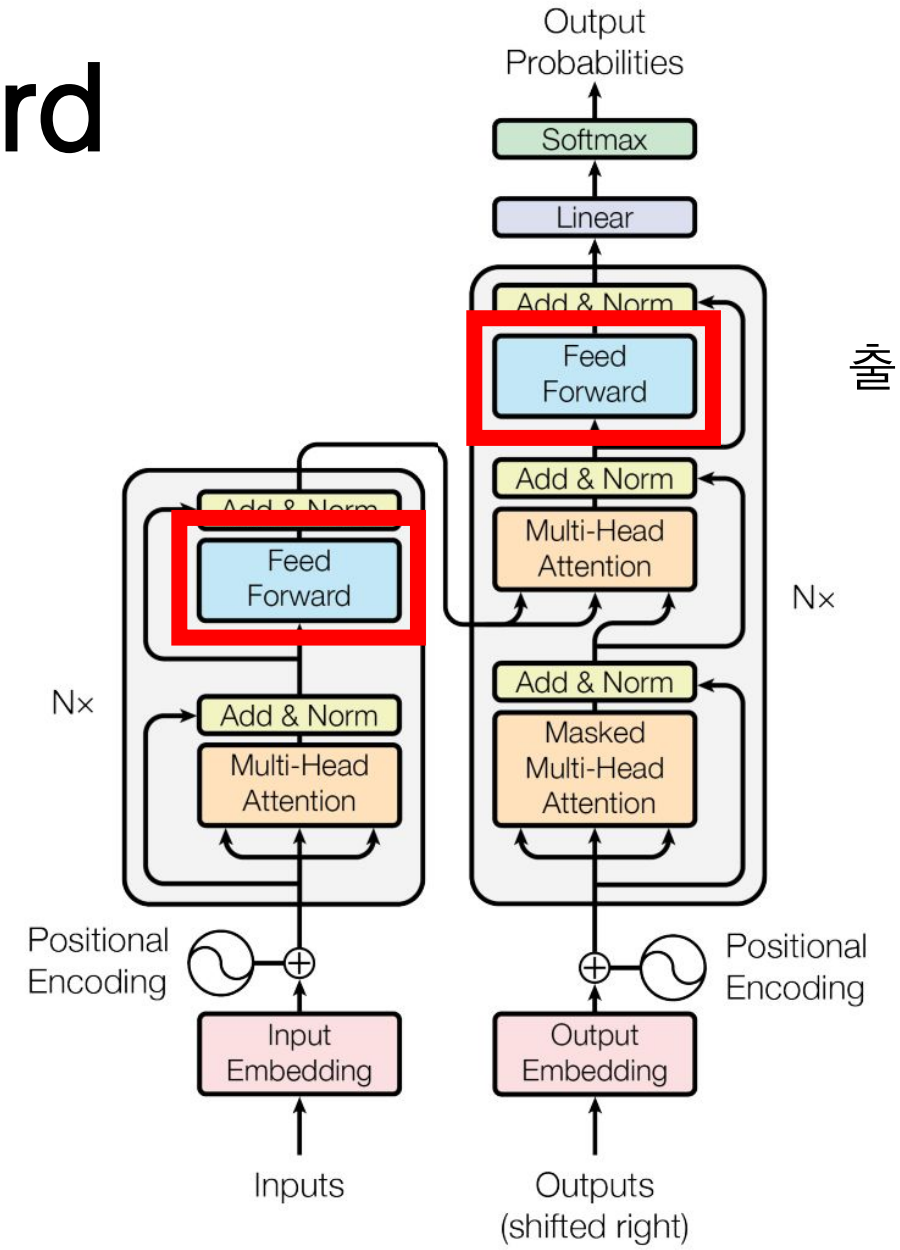
Feed Forward

Position-wise FFN

< Encoder >

입력 문장의

의미 표현 추상화



< Decoder >

출력 단어 예측에 적합한

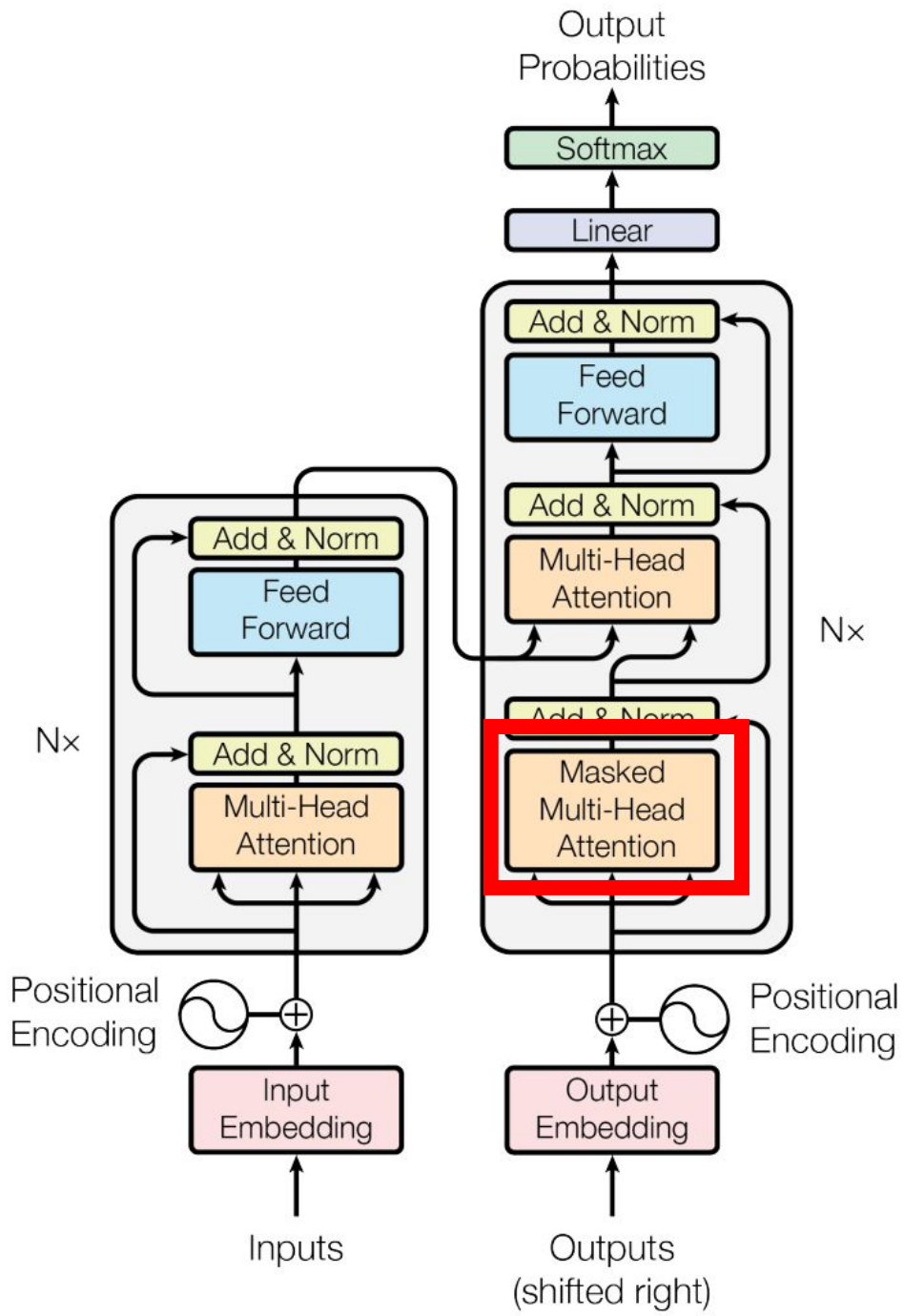
표현으로 정제

```
class PositionwiseFeedForward(nn.Module):

    def __init__(self, d_model, hidden, drop_prob=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.linear1 = nn.Linear(d_model, hidden)
        self.linear2 = nn.Linear(hidden, d_model)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=drop_prob)

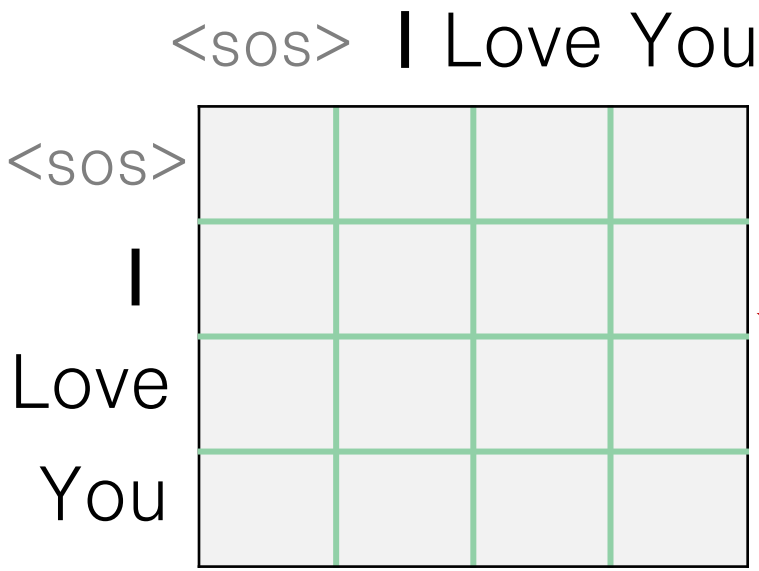
    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        return x
```

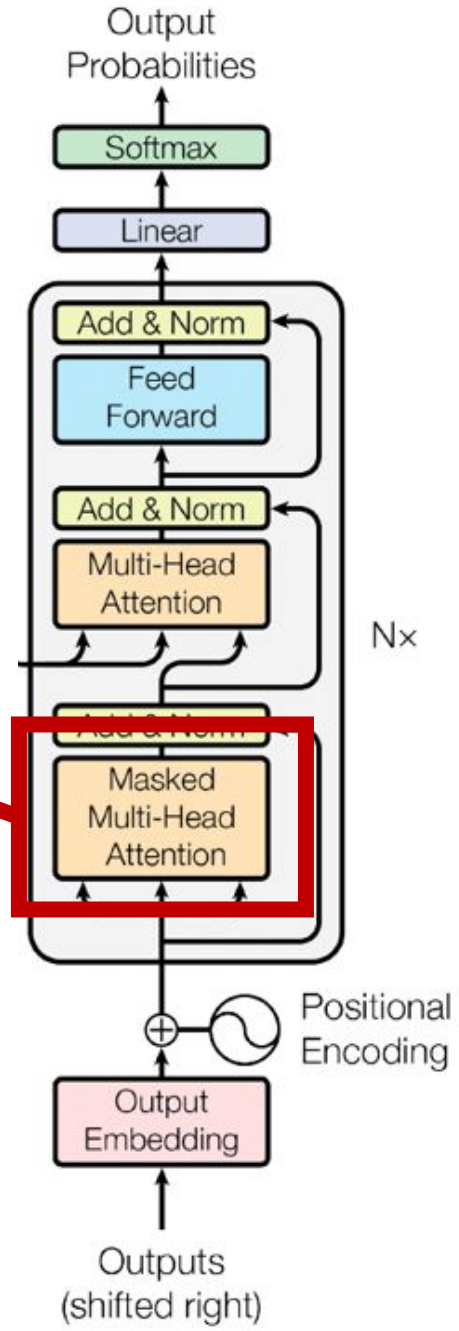
Masked Multi-Head Attention



Masking

“I Love You”

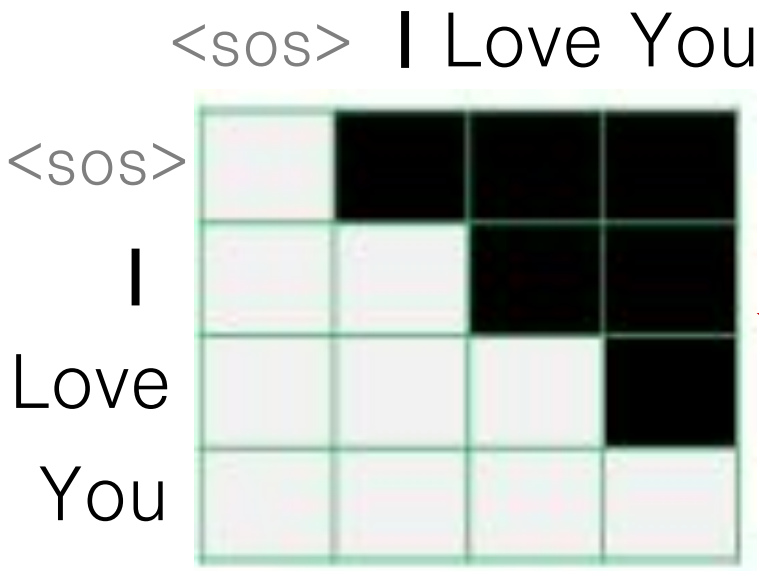




“I Love”를 생성하는데 “You”를 참고한다 ??
(❌)

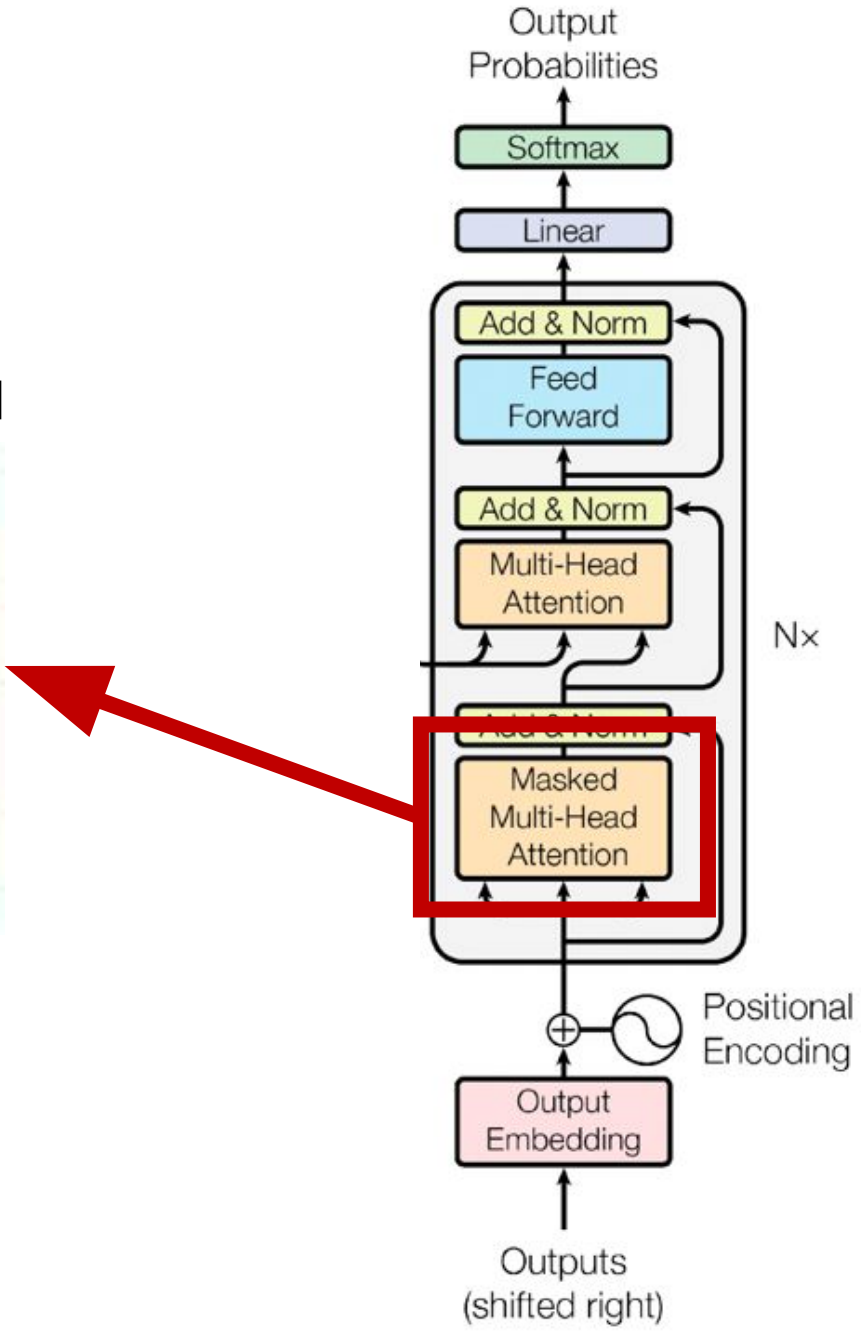
Masking

"I Love You"



∵ Self-Attention은 모든 단어를 동시에 볼 수 있음!

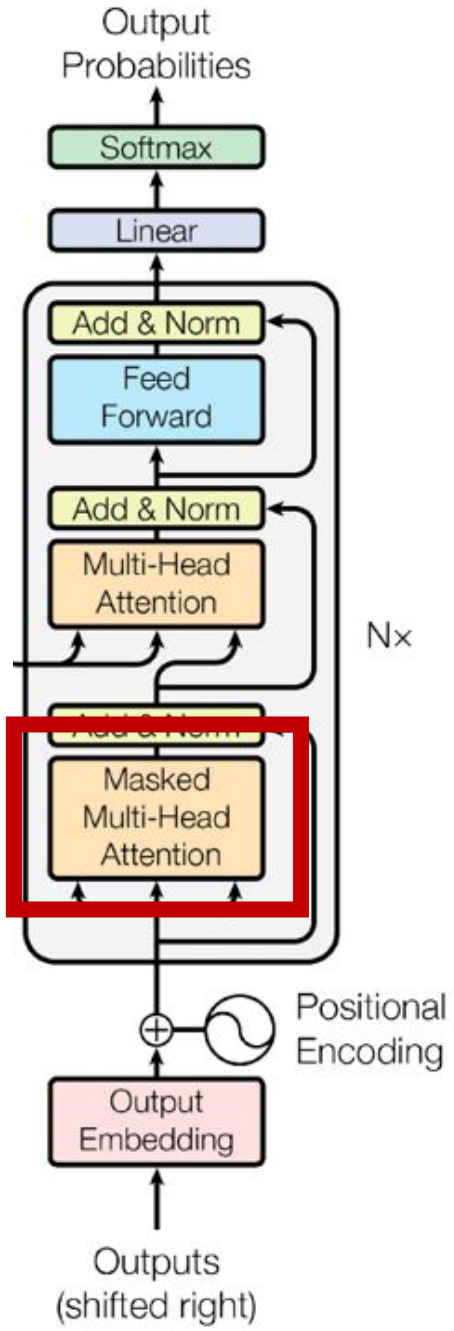
⇒ 미래 단어를 참조하면 학습이 부정확해짐



Masking

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

	<sos>	I	Love	You
<sos>	0	-∞	-∞	-∞
I	0	0	-∞	-∞
Love	0	0	0	-∞
You	0	0	0	0



```python
class MultiHeadAttention(nn.Module):

    def __init__(self, d_model, n_head):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.attention = ScaleDotProductAttention()
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_concat = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        # 1. dot product with weight matrices
        q, k, v = self.w_q(q), self.w_k(k), self.w_v(v)

        # 2. split tensor by number of heads
        q, k, v = self.split(q), self.split(k), self.split(v)

        # 3. do scale dot product to compute similarity
        out, attention = self.attention(q, k, v, mask=mask)

        # 4. concat and pass to linear layer
        out = self.concat(out)
        out = self.w_concat(out)

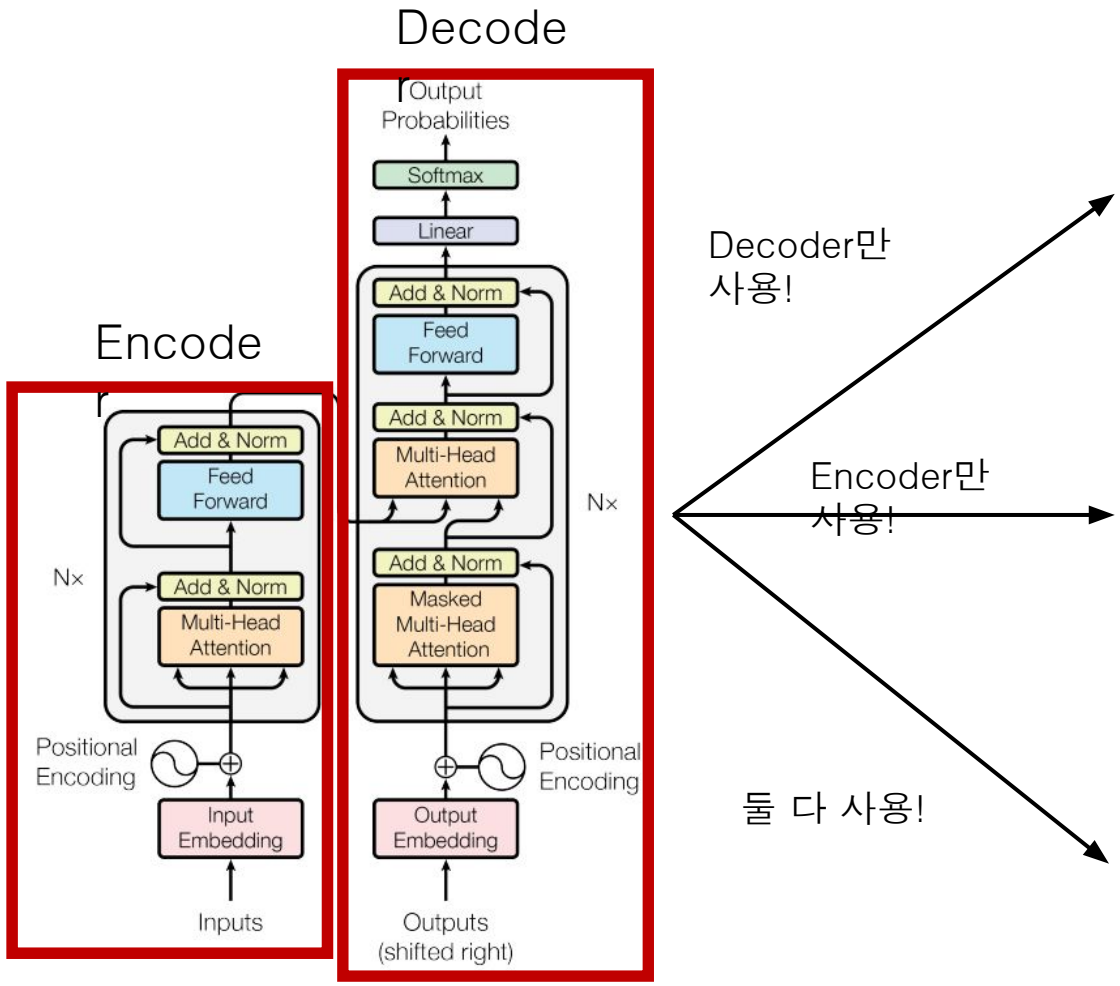
        # 5. visualize attention map
        # TODO : we should implement visualization

        return out
```

Foundation Model

그밖에도 …

ViT(이미지 처리)
DALL-E(이미지 생성)
Longformer(장문 특화)
ELECTRA(효율적 학습)



Decoder만 사용!

Encoder만 사용!

둘 다 사용!

a.k.a 언어 생성의 왕

주요 활용 분야: 챗봇, 텍스트 생성, 요약, 창작 등 결과물을 만들어내는 작업

Google BERT

a.k.a 문맥 이해의 왕

주요 활용 분야: 문장 분류, 개체명 인식, 질의응답 등 분석이 필요한 작업

T5

a.k.a 문장 변환의 왕

주요 활용 분야: 기계 번역, 문서 요약, 텍스트 스타일 변환 등 형태를 바꾸는 작업

Limitations

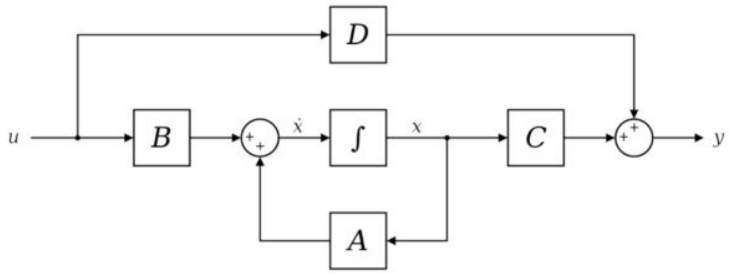
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Transformer의 단점?

- 엄청난 계산량
- 장거리 의존성 문제(Long Range Dependency)

Transformer를 대체하겠다! SSM(State Space Model)

- 장거리 의존성 문제를 효과적으로 다룸
- 하지만 여전히 Transformer에는 성능이 못 미침…



Outro

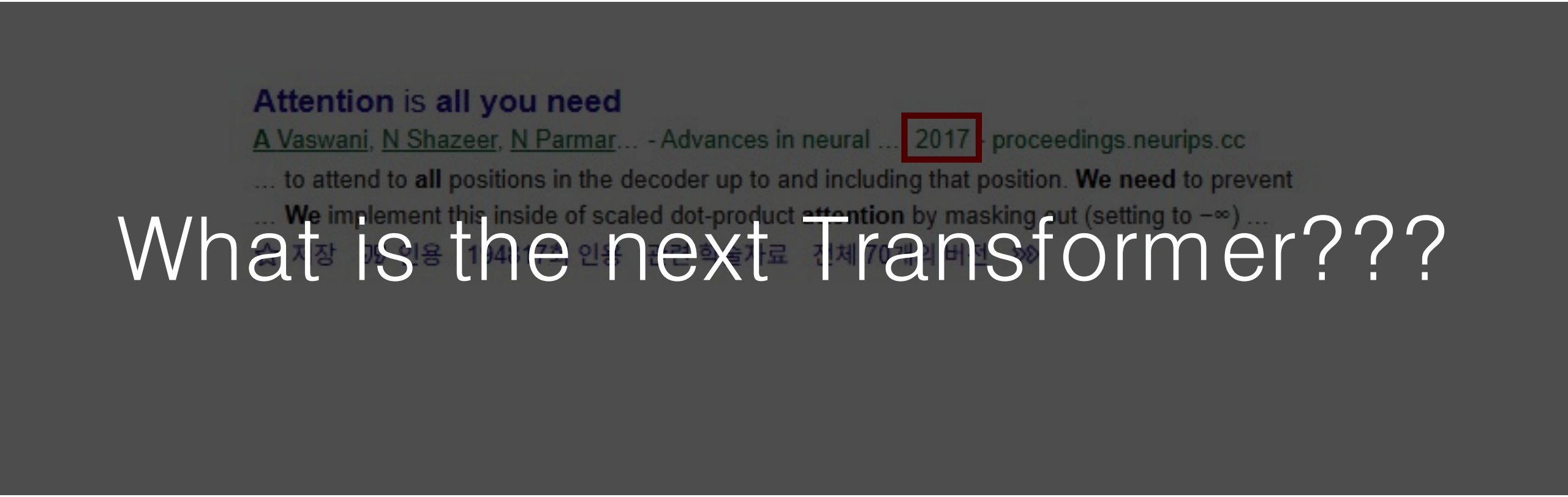
Attention is **all you need**

A Vaswani, N Shazeer, N Parmar... - Advances in neural ... 2017 - proceedings.neurips.cc

... to attend to **all** positions in the decoder up to and including that position. **We need** to prevent ... **We** implement this inside of scaled dot-product **attention** by masking out (setting to $-\infty$) ...

☆ 저장 99 인용 194817회 인용 관련 학술자료 전체 70개의 버전 »

Outro



What is the next Transformer???

Source

15-01 어텐션 메커니즘 (Attention Mechanism) - 딥 러닝을 이용한 자연어 처리 입문

16-01 트랜스포머(Transformer) - 딥 러닝을 이용한 자연어 처리 입문

트랜스포머, ChatGPT가 트랜스포머로 만들어졌죠. - DL5

그 이름도 유명한 어텐션, 이 영상만 보면 이해 완료! - DL6

Attention/Transformer 시각화로 설명

[Deep Learning 101] 트랜스포머, 스텝 바이 스텝

[논문리뷰] Attention is All you need의 이해

감사합니다.