Attention Is All You Need

논문 리뷰: 트랜스포머, RNN을 대체할 병렬 처리 기반 NLP 모델의 탄생

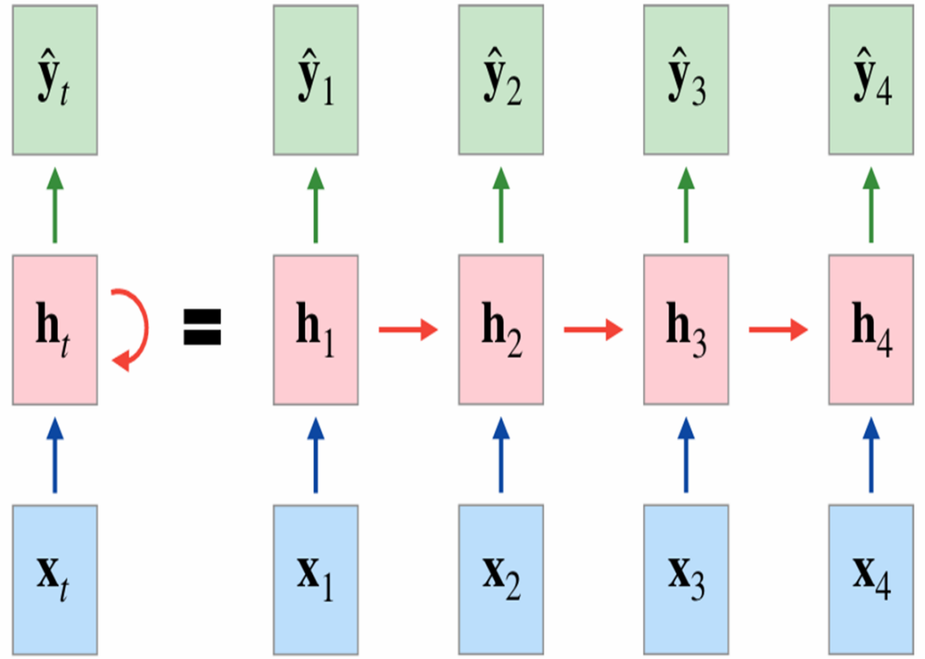
5팀 | STC | 이상엽, 배정윤, 이상진, 김준희

목차

RNN의 한계점

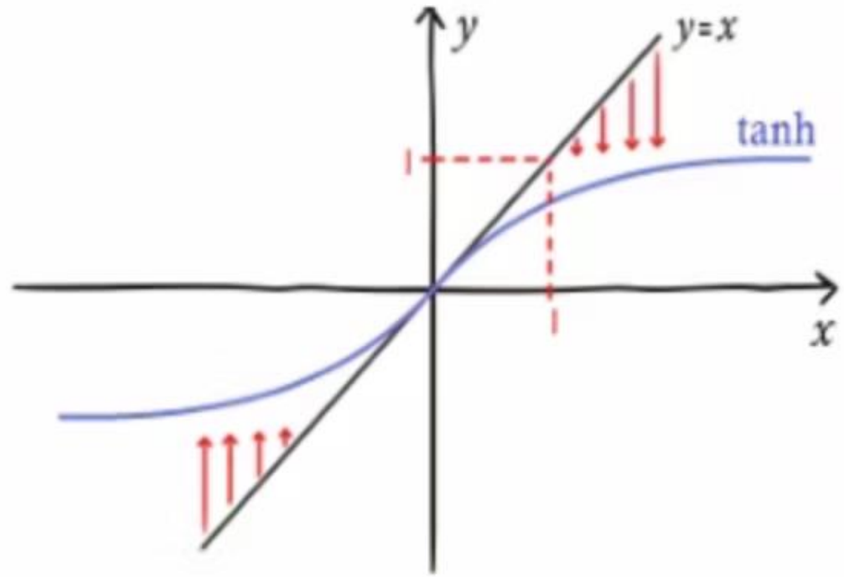
- **순차적 처리 (Sequential Processing)**

  순환 신경망 (RNN)은 입력을 순서대로 처리하며, 정보를 단계별로 전달



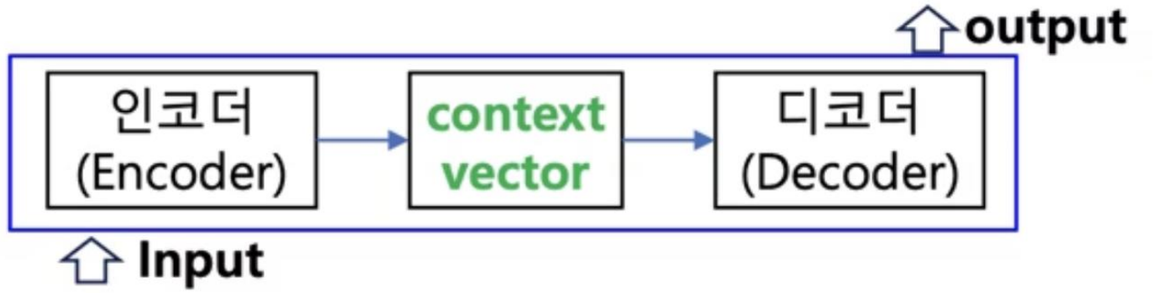
- **기울기 소실(Vanishing Gradient)**

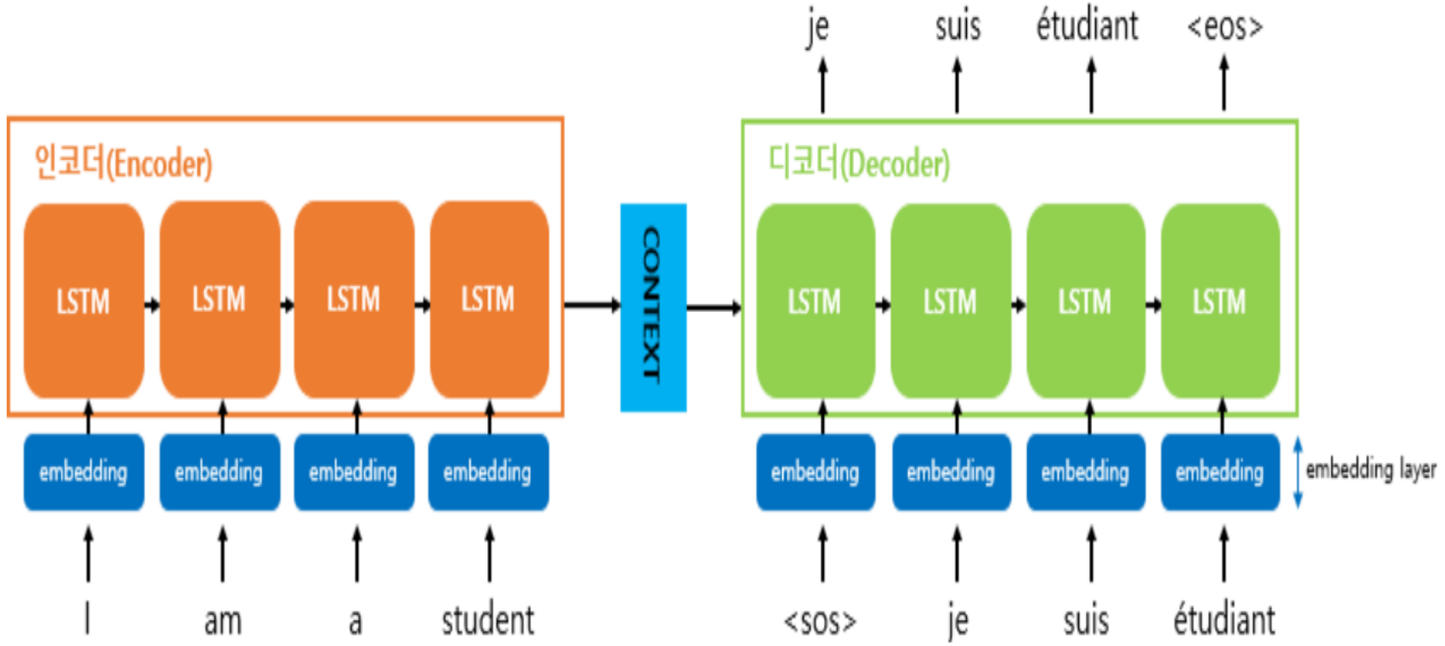
  특히 긴 시퀀스(문장)의 경우, 기울기 소실 문제로 인해 학습이 어렵다는 문제가 존재



Seq2Seq의 구조

<인코더와 디코더>





- Encoder, Decoder가 RNN으로 구성되어 있어, RNN의 문제점이 남음
- Context vector에 마지막 단어의 정보를 가장 뚜렷하게 담게 되는 문제를 내포

  → 즉 마지막 단어의 영향력이 높다
- 하나의 고정된 크기의 Context vector 순서대로 단계별로 전달 되어 긴 시퀀스의 경우 학습률이 희석

트랜스포머: Attention is All You Need

- **2017년 구글의 연구원들과 과학자들**의 "**Attention is All You Need**"라는 **논문**에서 처음 소개됨

- 순환 신경망(RNN)이나 컨볼루션 신경망(CNN)과는 달리 **순서를 따라가며 기억해야 하는 제약이 없고, 병렬처리가 가능**

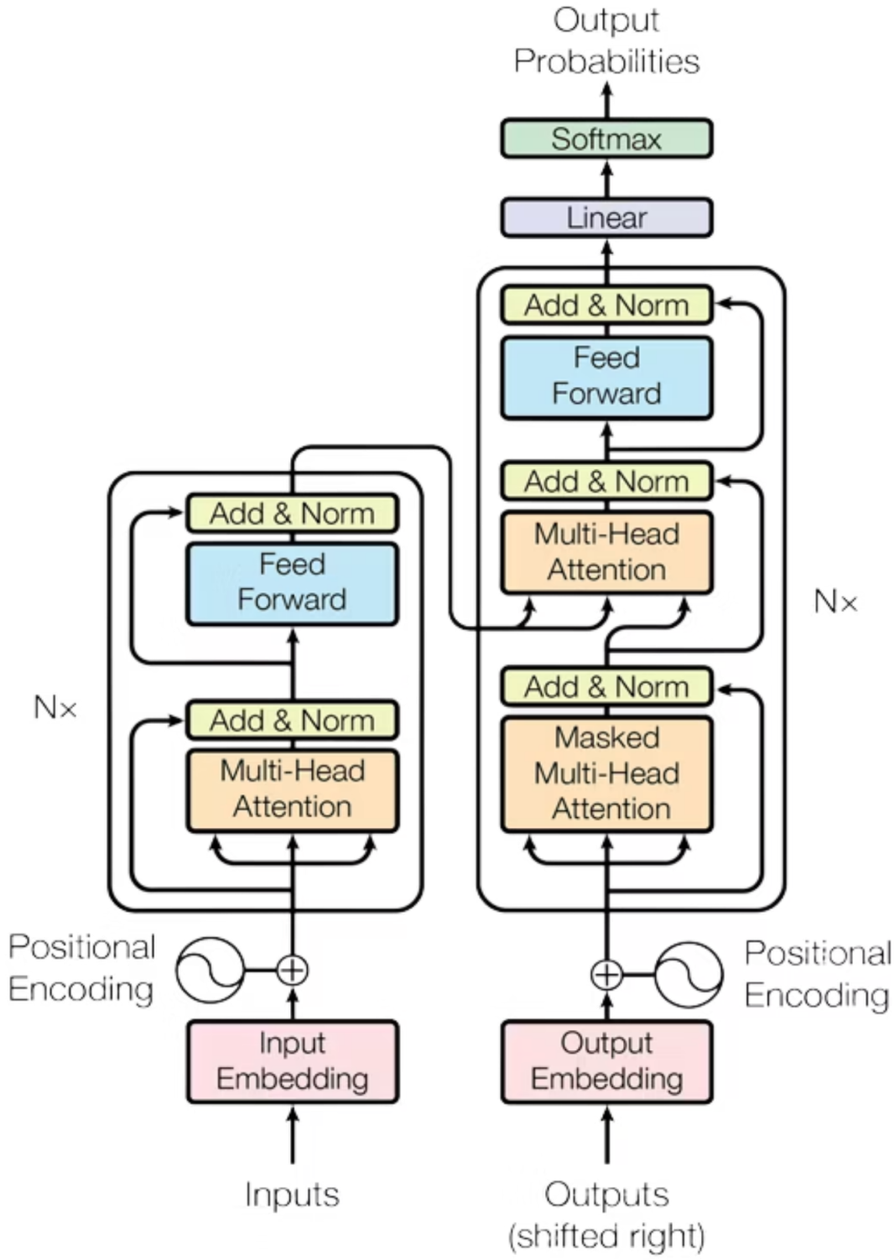


Figure 1: The Transformer - model architecture.

트랜스포머: Attention is All You Need

- 트랜스포머는 Seq2Seq처럼,
  인코더(Encoder)와 **디코더(Decoder)** 구조를 가짐
- **Attention 만을 활용**하여 구성함 (*RNN을 사용하지 않음)
  → 학습 속도가 대폭 향상
- **Multi-Head Attention 메커니즘을 사용**하여 **병렬 처리**하는 자연어 처리(NLP: Natural Language Processing) 분야에 대표적인 모델
- 문맥상 멀리 떨어진 **단어들 간의 관계를 효과적으로 파악** 가능

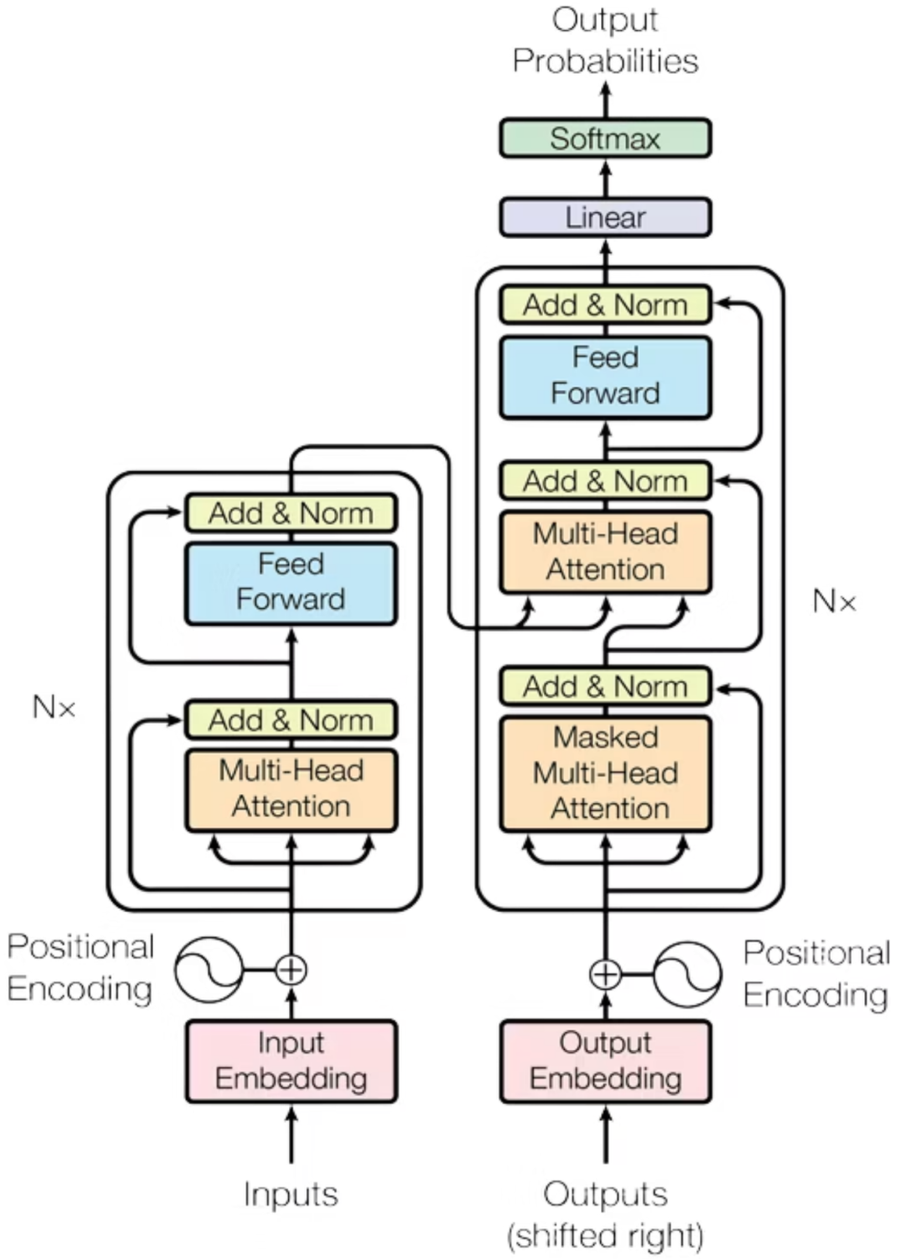
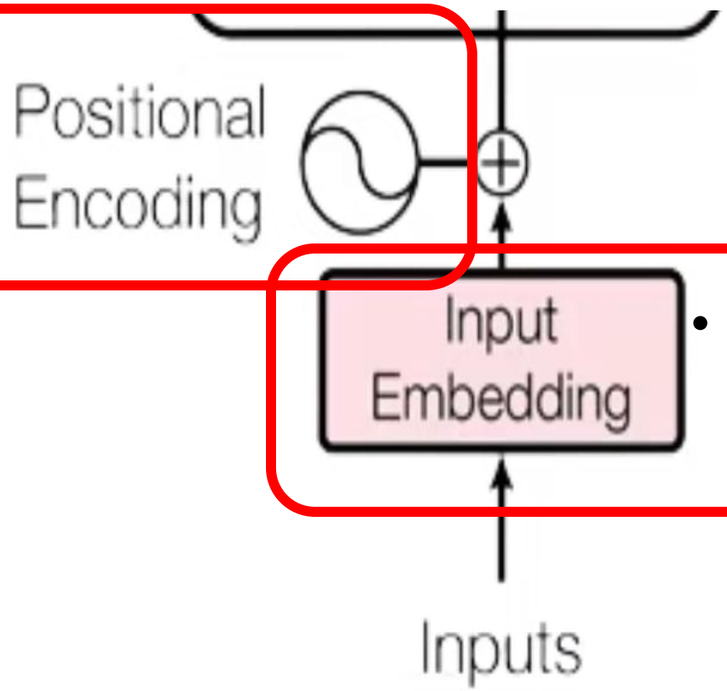


Figure 1: The Transformer - model architecture.

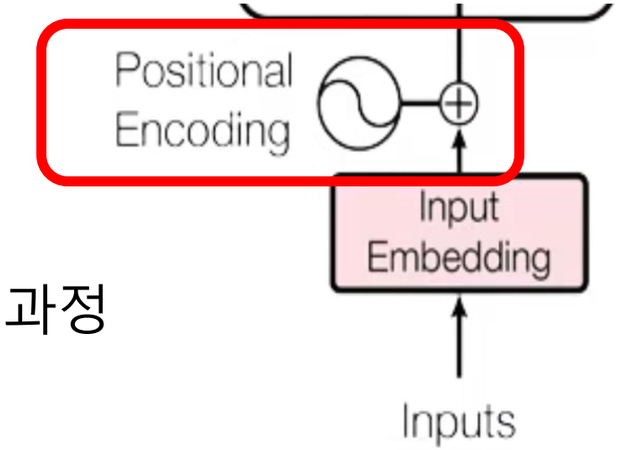
Input Embedding & Positional Encoding

- RNN처럼 순차적으로 데이터를 처리하지 않아 단어의 순서를 알 수 없기 때문에 별도의 위치 정보 추가 필요



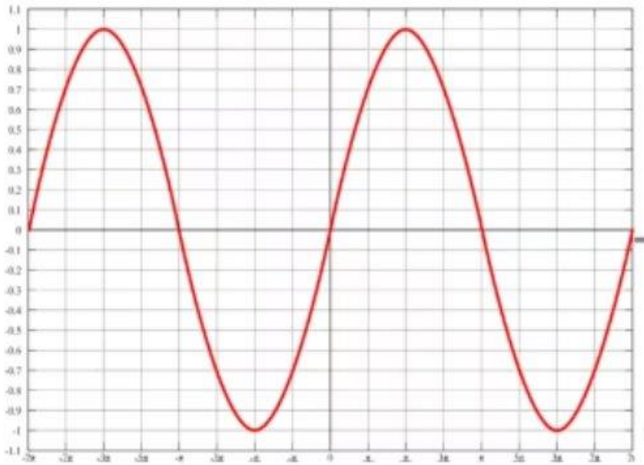
- 입력된 문장의 각 단어 또는 기호(token)를 고차원 벡터로 변환하는 과정.

Positional Encoding
위치 정보 부여



- token 값이 몇 번째 문장에 몇 번째 단어로 들어가 있는지를 알게 해주는 과정
- 위치 정보를 수학적 주기 함수(sin/cos)로 표현.

  sin함수와 cos 함수를 겹쳐 써서 조합하여 각 token에 대한 순서(위치) 차이를 구분 가능



: 사인(sine) 주기 함수 예시

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

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

- pos: 토큰의 위치
- i: 문자열 층
- d_model: 임베딩 벡터 차원

- sin/cos 장점
  1) 상대적 위치 학습 가능
  2) 무한 확장 가능 : 훈련때 보지 못한 긴 문장도 처리 가능
  3) 학습 불필요 : 고정된 함수로, 파라미터 학습이 필요 없음

Attention

Query

- Query : 물어볼 기준 단어 벡터
- 질문을 잘하자!

Key

- Key : Query 에 답할 후보 단어 벡터
- 답변을 잘하자!

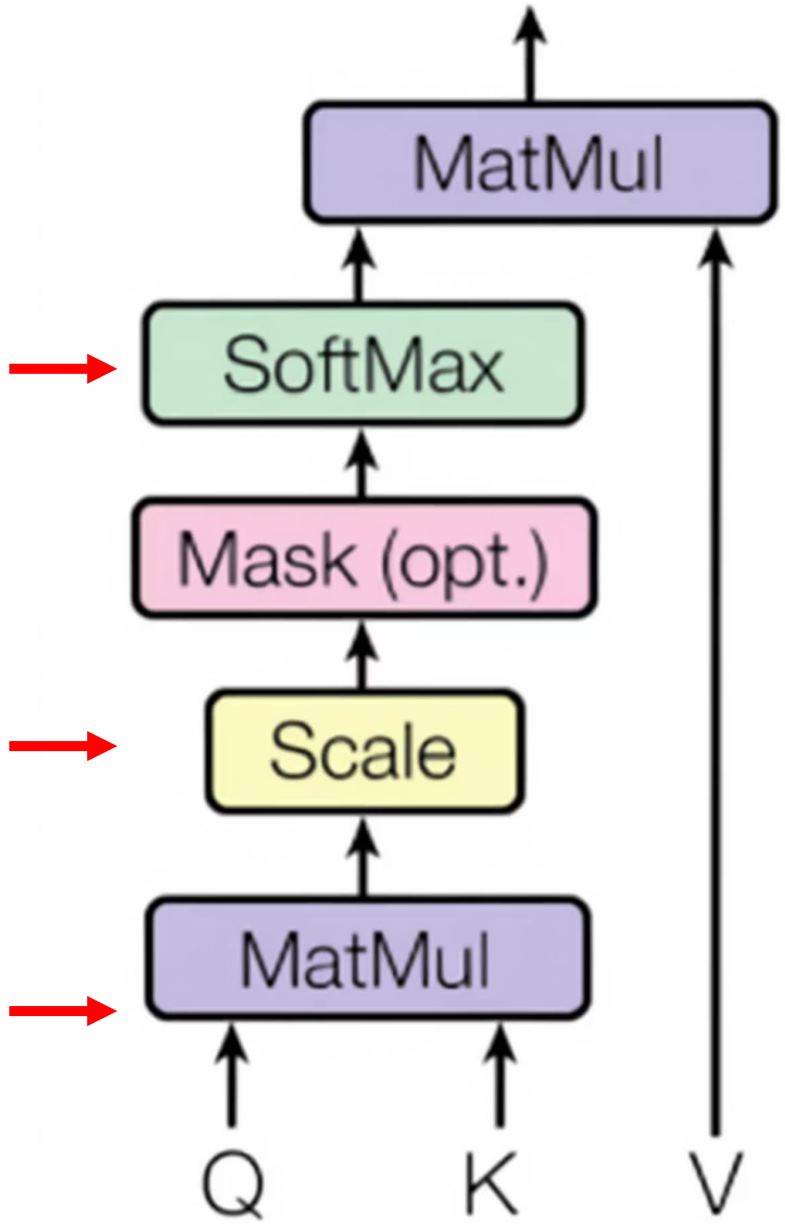
Value

- Value : 키 단어의 의미를 담은 벡터
- 표현을 잘하자!

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

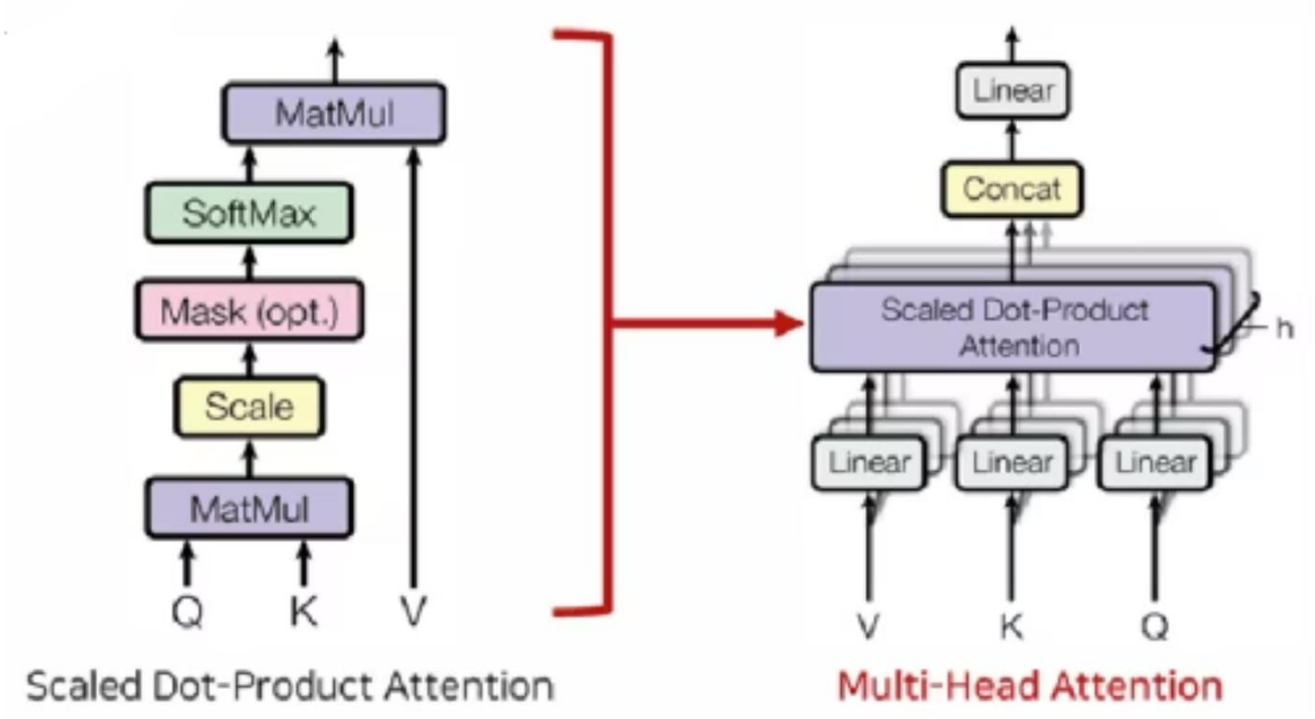
Attention

- 스케일링된 점수를 소프트맥스 함수를 이용하여 입력 단어에 대한 확률 값으로 변환, 모든 점수의 합이 1이 되도록 분포
- 기울기를 안정화하고, 특히 고차원 키 벡터에서 내적 값이 과도하게 커지는 현상을 방지
- 각 단어를 Query, Key, Value 벡터로 변환한 후 가중치 (Attention score)를 계산 어텐션 가중치는 Value 벡터들의 가중합 계산에 활용 전체 시퀀스의 관련 정보를 고려하여 단어의 맥락 인식 표현 생성



Multi-Head Attention

Self-attention | 입력 문장의 모든 단어 간 관계를 병렬로 계산



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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

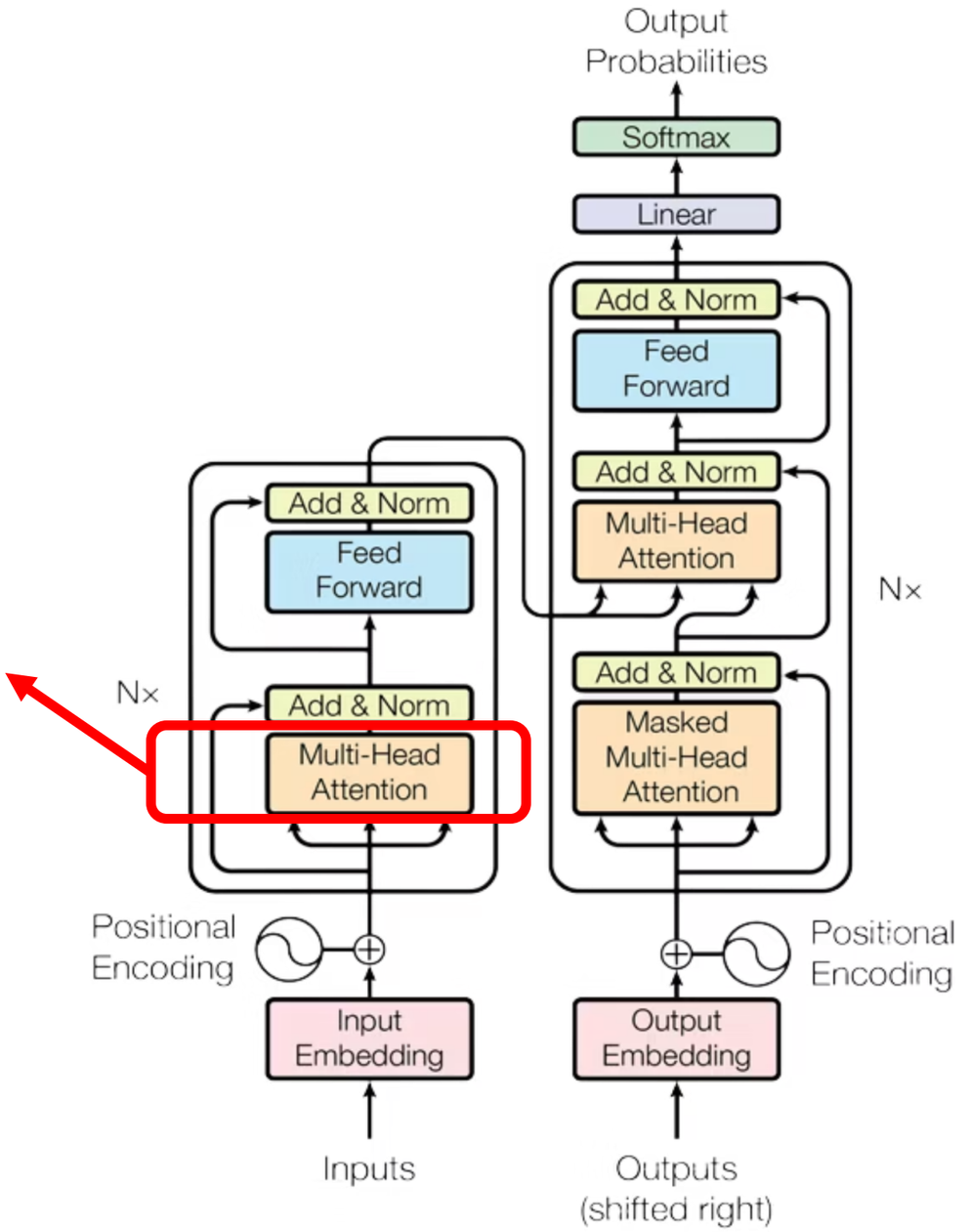
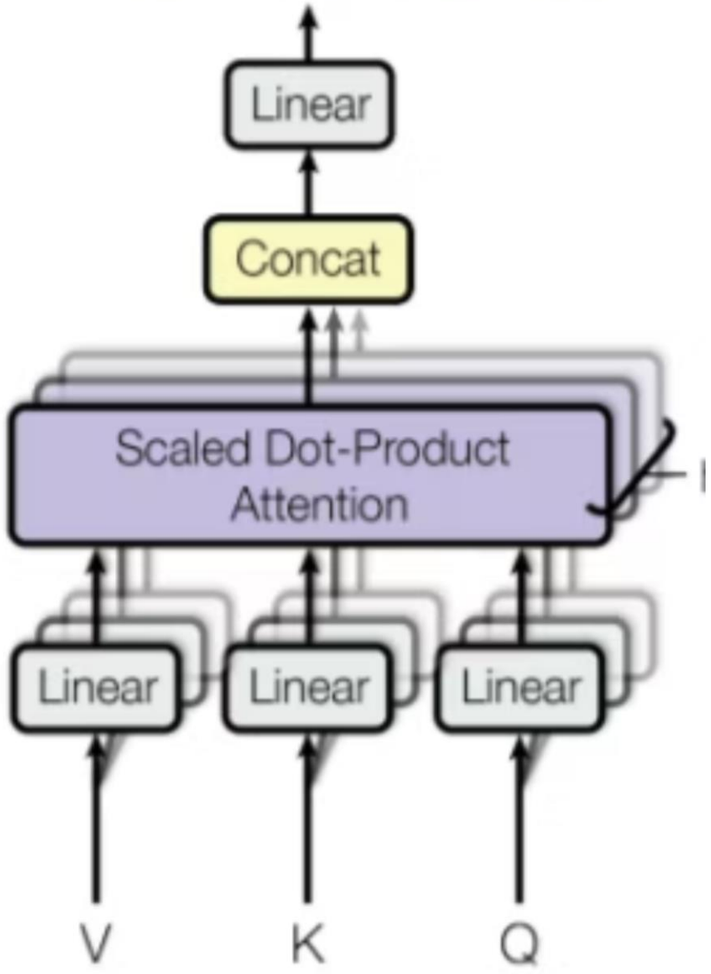


Figure 1: The Transformer - model architecture.

Multi-Head Attention



- 단일 Attention만 사용하면 정보의 제한이 있으므로**Attention을 여러 관점으로 동시에 학습**

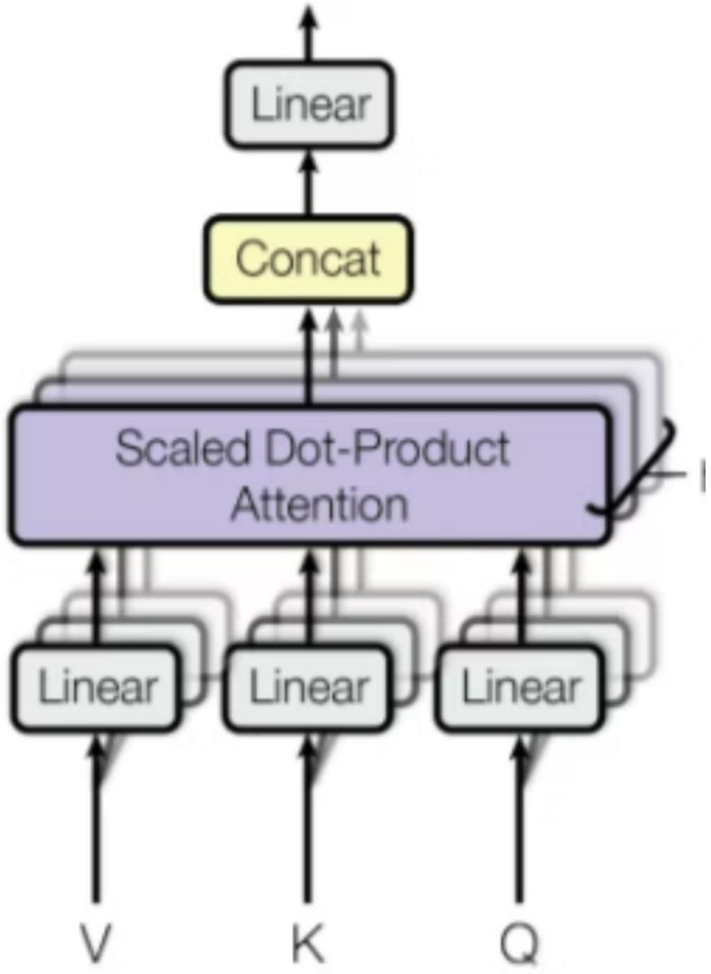
- 구체적 **예시**로 이해하기

  "The animal didn't cross the street because it was too tired."

  → Single Attention
  : "it"이 "animal"과 관련있다는 것만 파악

  → **Multi-Head Attention (h=8개 헤드):**
  : Head 1: "it" ↔ "animal" (대명사 해결)
  Head 2: "tired" ↔ "animal" (상태 연결)
  Head 3: "cross" ↔ "street" (동사-목적어)
  Head 4: "didn't" ↔ "cross" (부정어 관계)
  Head 5: 문장 전체의 인과관계 ("because" 중심)
  ... 나머지 헤드들도 각자 다른 패턴 학습

Multi-Head Attention



$$MultiHead(Q, K, V) = \quad Concat(head_1, \ldots, head_h)W^O$$

Concat(Consitenation)

- 서로 다른 Attention Head들의 출력을 하나의 큰 벡터로 통합

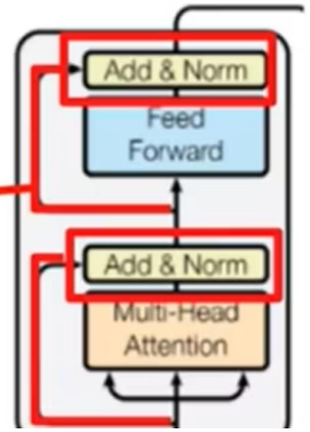
Linear

- 선형 변환을 하면, 수학적으로는 헤드별 출력을 가중합하여 새로운 표현을 만듬
- 각 헤드가 독립적으로 학습한 특성을 충분히 보존해 둔 뒤, 학습 가능한 방식으로 섞도록 설계

Add & Norm : Residual Connection

$$\boldsymbol{Add}\ \&\ \boldsymbol{Norm}$$

$$LN = LayerNorm(x + Sublayer(x))$$



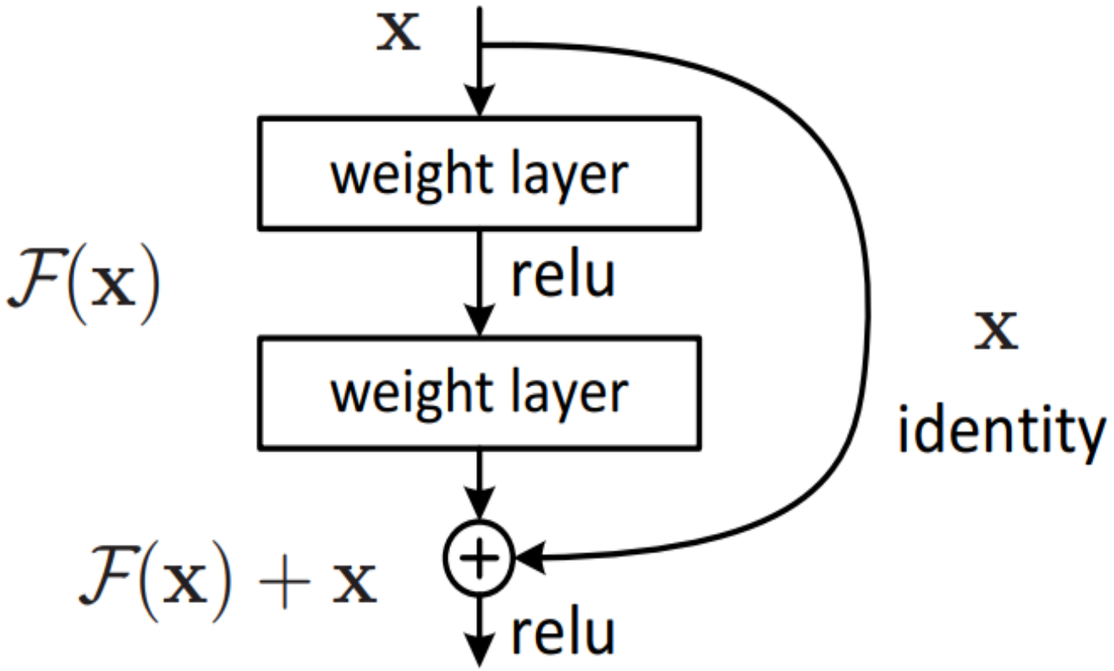
- 중간에 Add&Norm을 삽입하여 기울기 흐름을 개선하고 학습 과정 안정화
  (*Transformer의 Encoder와 Decoder를 각각 6번 반복 후 기울기 손실 방지)

Residual Connection

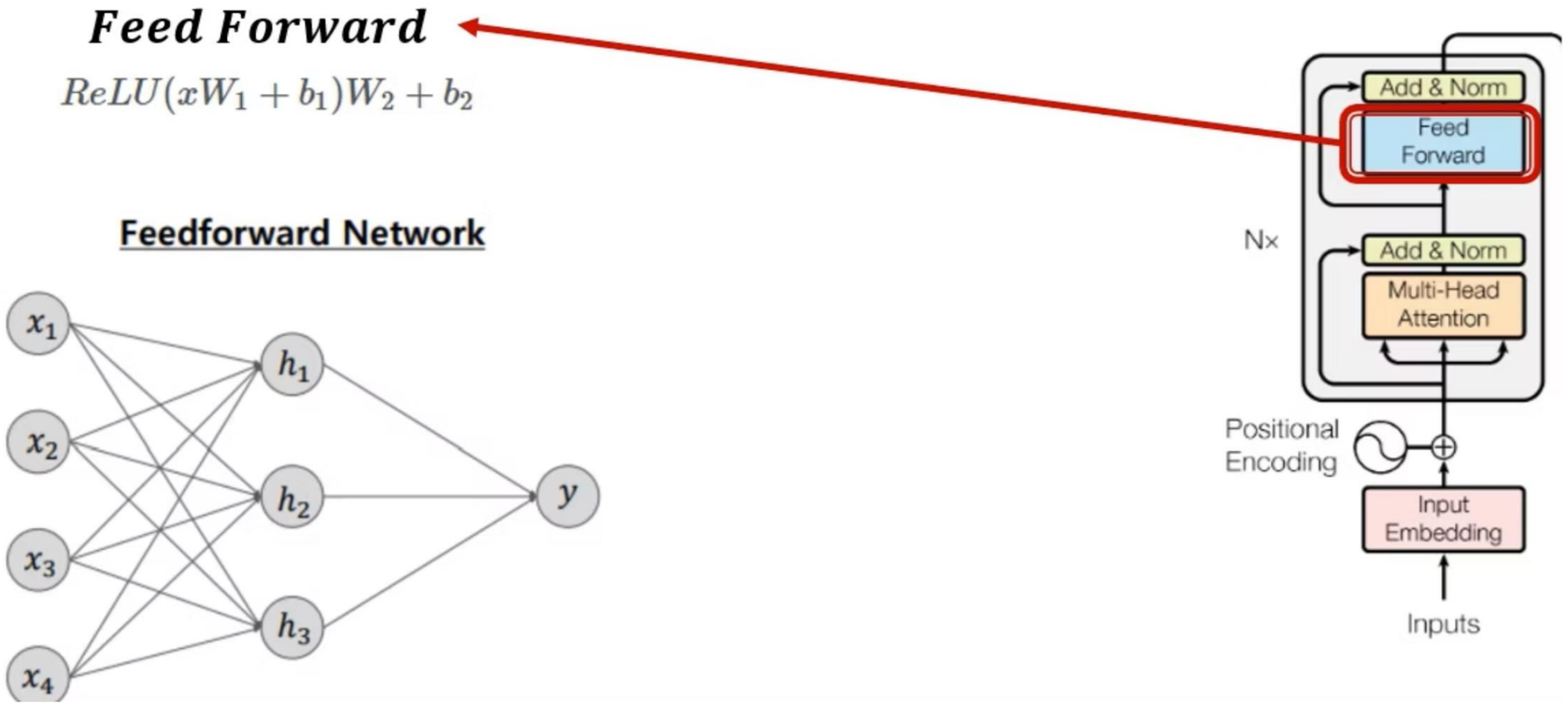
- **개념**
  출력 = 변환 결과 F(x)+입력 x
  입력을 그대로 더하여 **정보 손실 최소화**

- **필요성**
  깊은 네트워크에서 성능 저하(Degradation) 문제 발생
  Residual 구조 → Gradient 흐름 원활 → 학습 안정화

- **장점**
  잔차(Residual)만 학습하므로 학습 용이하고,
  다양한 경로가 생겨 앙상블 효과 발생



Feed Forward Neural Network 구성



- 더 복잡한 문맥의 고차원적인 의미 포착
- 각 층의 MLP 구조, Attention을 통해 얻은 **토큰별 표현**을 입력
- 두 개의 선형 변환과 그 사이에 위치하는 활성화 함수(예: ReLU) → Linear 통과
- Feed Forward 순서: Linear (W1) → 비선형 함수 (ReLU) → Linear (W2)

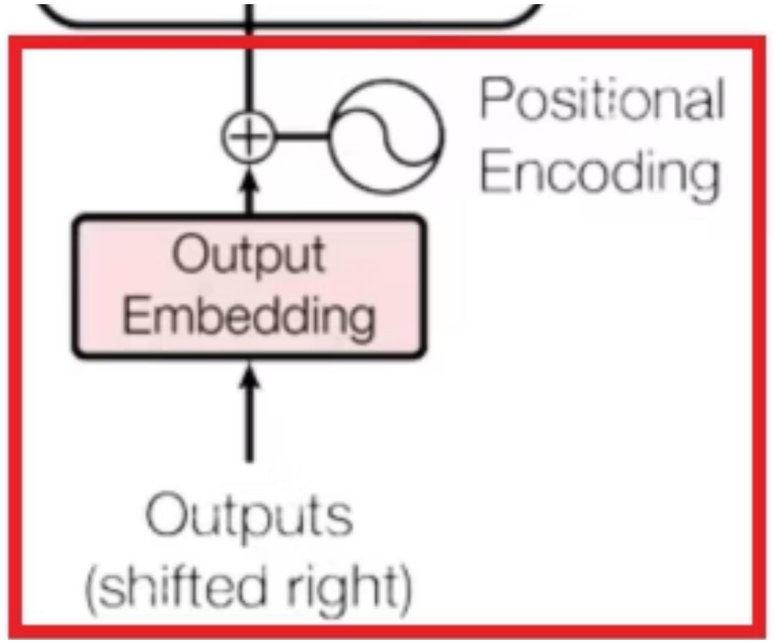
Output Embedding + Positional Encoding

1. Output Embedding

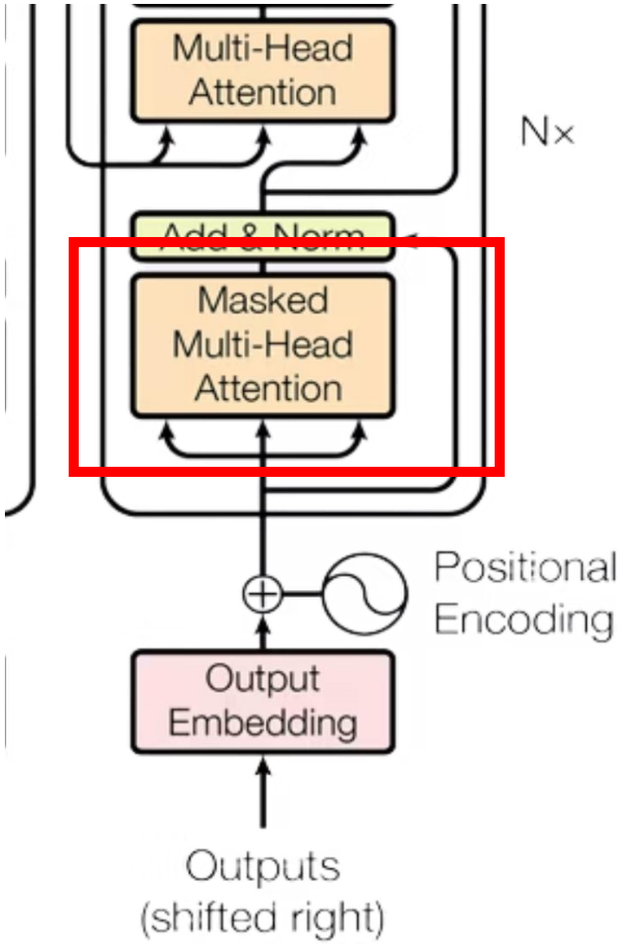
- Decoder 입력으로 들어가는 출력 토큰을 임베딩
- Encoder와 동일하게 차원이 맞춰진 벡터로 변환
- Output Embedding 과 Positional Encoding 더해 순서에 반영

2. Shifted Right (출력 이동)

- Decoder는 이전 시점까지의 출력만 보고 다음 단어를 예측
- 학습 시 정답 문장을 한 칸 오른쪽으로 shift하여 입력으로 사용
- 이렇게 해야 Decoder가 미래 토큰을 미리 보지 않고 학습 가능

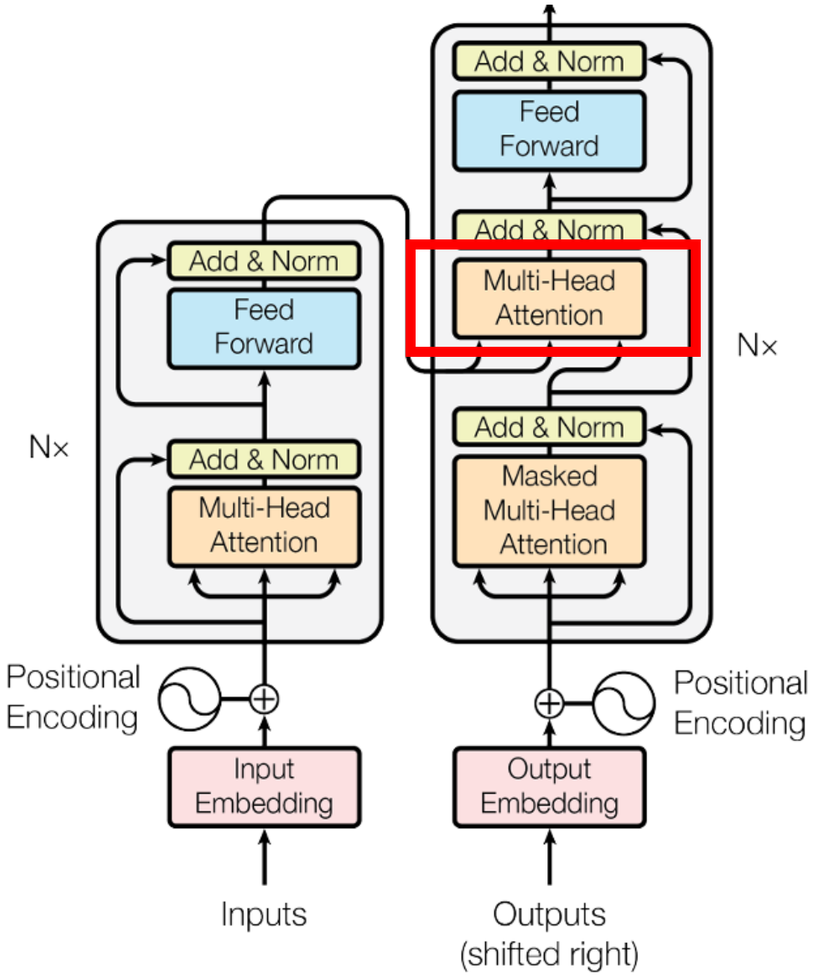


Masked Multi-Head Attention



- 디코더 셀프 어텐션에서 각 위치가 이후 위치를 보지 못하게 제한
- 자기회귀 생성에서 필수
- 마스크(Mask)를 사용해 현재 위치 이후 단어 차단
- 셀프 어텐션에서 마스킹을 통해 미래 단어를 보지 못하게 하여 올바른 다음 단어 학습 보장

Multi-Head Encoder-Decoder Attention



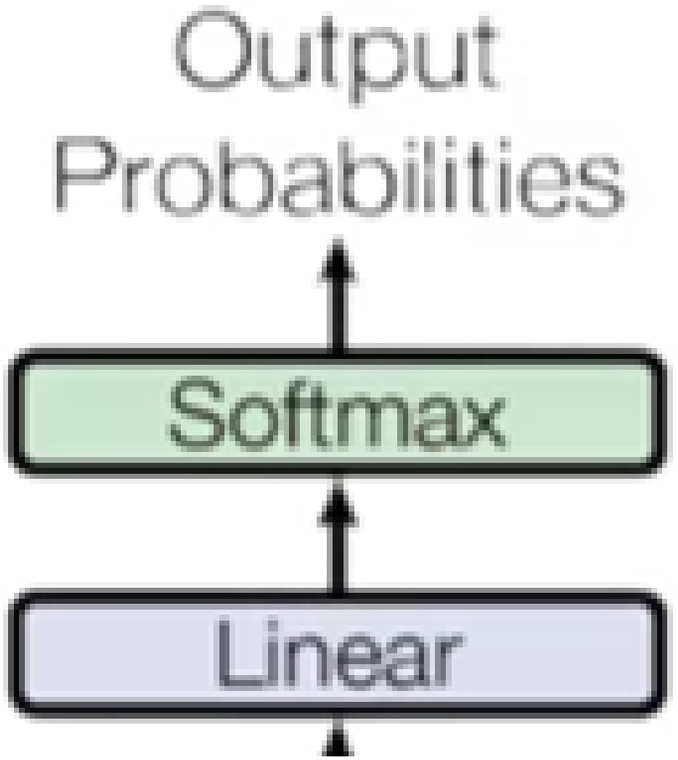
- 인코더 출력과 현재 디코더 상태를 연결하는 역할, 크로스 어텐션(Cross-Attention)이라고도 부름
- 디코더가 "무엇을 생성할지"를 소스 문맥과 정렬
- 만나는 지점: 인코더 → Key & Value, 디코더 → Query
- 여러 번의 Attention을 병렬 계산하여 성능 높은 문맥 이해

Linear & Softmax

최종적으로 단어 사전에 있는 모든 단어에 대해 확률 값을 계산 후 가장 확률이 높은 단어를 다음 단어로 선택.

- Linear Layer (Fully Connected Layer): 디코더에서 나온 마지막 hidden state를 단어 집합(Vocabulary) 크기에 맞는 벡터로 변환

- Softmax Layer: Linear의 출력을 확률 분포로 전환



Multi-Head Attention

```python
class MHA(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()

        self.n_heads = n_heads

        self.fc_q = nn.Linear(d_model, d_model) # 차 or 개x차 or 개x개x차 로 입력해줘야
        self.fc_k = nn.Linear(d_model, d_model)
        self.fc_v = nn.Linear(d_model, d_model)
        self.fc_o = nn.Linear(d_model, d_model)

        self.scale = torch.sqrt(torch.tensor(d_model / n_heads))

    def forward(self, Q, K, V, mask = None):

        Q = self.fc_q(Q) # 개단차
        K = self.fc_k(K)
        V = self.fc_v(V)

        Q = rearrange(Q, '개 단 (헤 차) -> 개 헤 단 차', 헤 = self.n_heads) # 개단차 -> 개헤단
        K = rearrange(K, '개 단 (헤 차) -> 개 헤 단 차', 헤 = self.n_heads)
        V = rearrange(V, '개 단 (헤 차) -> 개 헤 단 차', 헤 = self.n_heads)

        attention_score = Q @ K.transpose(-2,-1)/self.scale # 개헤단단

        if mask is not None:
            attention_score[mask] = -1e10

        attention_weights = torch.softmax(attention_score, dim=-1) # 개헤단단

        attention = attention_weights @ V # 개헤단차

        x = rearrange(attention, '개 헤 단 차 -> 개 단 (헤 차)') # 개헤단차 -> 개단차
        x = self.fc_o(x) # 개단차

        return x, attention_weights
```

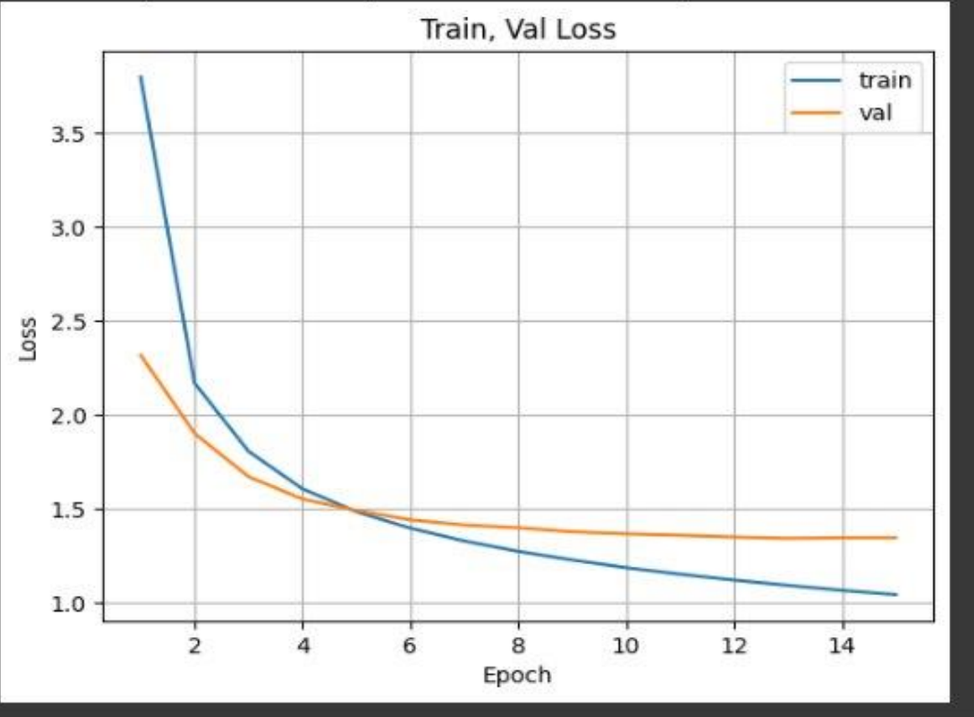
```python
class Transformer(nn.Module):
    def __init__(self, vocab_size, max_len, n_layers, d_model, d_ff, n_heads, drop_p):
        super().__init__()

        self.input_embedding = nn.Embedding(vocab_size, d_model)
        self.encoder = Encoder(self.input_embedding, max_len, n_layers, d_model, d_ff, n_heads, drop_p)
        self.decoder = Decoder(self.input_embedding, max_len, n_layers, d_model, d_ff, n_heads, drop_p)

        self.n_heads = n_heads

        # for m in self.modules():
        #     if hasattr(m,'weight') and m.weight.dim() > 1: # layer norm에 대해선 initial 안하겠다는 뜻
        #         nn.init.kaiming_uniform_(m.weight) # Kaiming의 분산은 2/Nin

        for m in self.modules():
            if hasattr(m,'weight') and m.weight.dim() > 1:
                nn.init.xavier_uniform_(m.weight) # xavier의 분산은 2/(Nin+Nout) 즉, 분산이 더 작다. => 그래서
```




```
1 Test(load_model, test_DL, criterion)
2 count_params(load_model)
```

```
Test loss: 0.868 | Test PPL: 2.381
37350377
```

[ ]

```
# Perplxity 구하기
y_hat = torch.tensor([[[0.3659, 0.7025, 0.3104]], [[0.0097, 0.6577, 0.1947]],[[0.5659, 0.0025, 0.0104]], [[0.9097, 0.0577, 0.7947]]])
target = torch.tensor([[2], [1], [2], [1]])

soft = nn.Softmax(dim=-1)
y_hat_soft = soft(y_hat)
print(y_hat_soft.shape)
v=1
for i, val in enumerate(y_hat_soft):
    v*=val[0,target[i]]
print(v**(-1/target.shape[0]))
# 3.5257

criterion_test = nn.CrossEntropyLoss()
print(y_hat.permute(0,2,1).shape)
print(target.shape)
print(torch.exp(criterion_test(y_hat.permute(0,2,1), target))) # 결론: loss에 torch.exp 취하셈
# 3.5257
```

```
torch.Size([4, 1, 3])
tensor([3.5257])
torch.Size([4, 3, 1])
torch.Size([4, 1])
tensor(3.5257)
```

[47] ✓ 6초

```
# 내 번역기 써보기!
src_text = "나는 어제 친구와 함께 아주 재미있는 영화를 보러 갔는데, 그 영화관은 정말 크고 좌석도 편안했으며, 음향도 훌륭했고, 스크린도 매우 선명했다. 그런데 그 영화의 제목이 기억나지 않는다."
print(f"입력: {src_text}")

translated_text = translation(load_model, src_text)[0]
print(f"AI의 번역: {translated_text}")
```

```
입력: 나는 어제 친구와 함께 아주 재미있는 영화를 보러 갔는데, 그 영화관은 정말 크고 좌석도 편안했으며, 음향도 훌륭했고, 스크린도 매우 선명했다. 그런데 그 영화의 제목이 기억나지 않는다.
AI의 번역: </s> I went to the movie yesterday with my friend, but the movie was very big, and the seat was very big, and the seat was very big.</s>
```

참고자료

- ‘Attention is All You Need’ 논문

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin

- The Illustrated Transformer (Jay Alammar)

링크: https://jalammar.github.io/illustrated-transformer/

- IBM 트랜스포머 모델이란 무엇인가요?

링크: https://www.ibm.com/kr-ko/think/topics/transformer-model

감사합니다

Q&A