



DSBA Transformer survey paper study

A Survey of Transformers

#1: Transformer Basic

arXiv preprint



고려대학교 산업경영공학과

Data Science & Business Analytics Lab

이유경, 김명섭, 윤훈상, 김지나, 허재혁, 김수빈

발표자 : 김수빈

01 Introduction

02 Background

1) Vanilla Transformer

2) Model Usage

3) Model Analysis

4) Model Comparison

▪ Transformer 모델이란?

- 자연어처리 Task 중, Machine translation을 위해 제안된 Sequence-to-sequence 모델
- 현재는 자연어처리뿐만 아니라, 컴퓨터 비전, 음성처리 등 다양한 분야에서 응용되고 있음

▪ Transformer 기반의 Pre-Trained-Models(PTMs)

- Transformer 모델은 PTM으로서 사용되었을 때 여러 Downstream Task에서 SOTA를 보이고 있음
- 특히, 자연어처리 분야에서 Transformer 모델의 PTM으로서 역할이 강력함

cf. 자연어처리 분야에서의 PTM의 의미

- 이미지나 음성 대비, 자연어 텍스트는 데이터의 축적 속도가 매우 빨라 방대한 양으로 존재
- 대용량의 텍스트를 처음부터 학습을 시키는 것은 시간 및 비용 측면에서 비효율적인데, PTM으로 주요 언어적 특성을 충분히 사전에 학습하여 활용할 수 있음
- 최근에는 잘 학습된 PTM 모델을 Fine-Tuning 하여 높은 성능을 내는 연구가 활발함

- Transformer 모델을 변형한 X-formers

- Transformer의 출현을 기점으로, 크게 다음 세 가지 관점에서 Transformer 모델을 변형한 모델이 활발히 연구 됨

- 1) Model Efficiency

- Limitation : Self-Attention 모듈로 인해 긴 입력 시퀀스를 처리할 때, 연산과 메모리의 비효율성 존재
 - Improvement : Lightweight Attention (Sparse Attention의 변형)

Divide and conquer methods (Recurrent and Hierarchical Mechanism)

- 2) Model Generalization

- Limitation : Transformer는 기본적으로 유연한 Architecture의 성격을 지니며,
input data의 structural bias에 대해서 최소한의 가정만을 하고 있어
적은 데이터셋으로의 강건한 모델 학습에 어려움이 있음
 - Improvement : Structural bias나 regularization의 도입, Large-scale unlabeled data에 대한 사전학습

- 3) Model Adaptation

- Specific한 downstream task에 적용하기 위한 변형

▪ Vanilla Transformer란?

- Transformer 모델이 처음 제안된 논문에서 설명하는 기본 Transformer 모델 (Attention Is All You Need (Vaswani et al., 2017))
- 특징
 - 1) Encoder-Decoder 구조를 지닌 Sequence to sequence model
 - 2) 각 Encoder와 Decoder는 L 개의 동일한 Block이 Stack된 형태를 지님
 - 3) Encoder Block의 구성
 - 1) Multi-head self-attention
 - 2) Position-wise feed-forward network
 - 3) Residual connection
 - 4) Layer Normalization
 - 4) Decoder Block
 - 1) 기본적인 Encoder Block 구성과 동일
 - 2) Cross-attention 모듈을 Multi-head self-attention과 Position-wise FFN 사이에 추가
 - 3) 디코딩 시에 미래 시점의 단어 정보를 사용하는 것을 방지하기 위해 Masked self-attention을 사용

(참고)

cross-attention은 본 Survey paper 상에서의 표현이고,
정확한 명칭은 encoder-decoder attention

Attention Modules

- Query-Key-Value (QKV) 모델로 Attention mechanism 도입
 - Query : 현재의 hidden state 값
 - Key : hidden state와 영향을 주고 받는 값(Query와의 비교 대상)
 - Value : Key 값들을 Query와 Key 간의 유사도에 따라 가중합하여 Query에 반영하게 되는데, 이때 가중합의 대상으로 사용할 값

Scaled dot-product attention

- $Q(\text{Query}) \in \mathbb{R}^{N \times D_K}$, $K(\text{Key}) \in \mathbb{R}^{M \times D_K}$, $V(\text{Value}) \in \mathbb{R}^{M \times D_V}$

내적을 위해 차원이 같아야 함

각 Key에 해당하는 Value 값을 가중합하여 사용

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

attention score by dot-product

row-wise softmax weight

attention weight matrix

scaled : softmax 함수의 gradient vanishing problem 완화

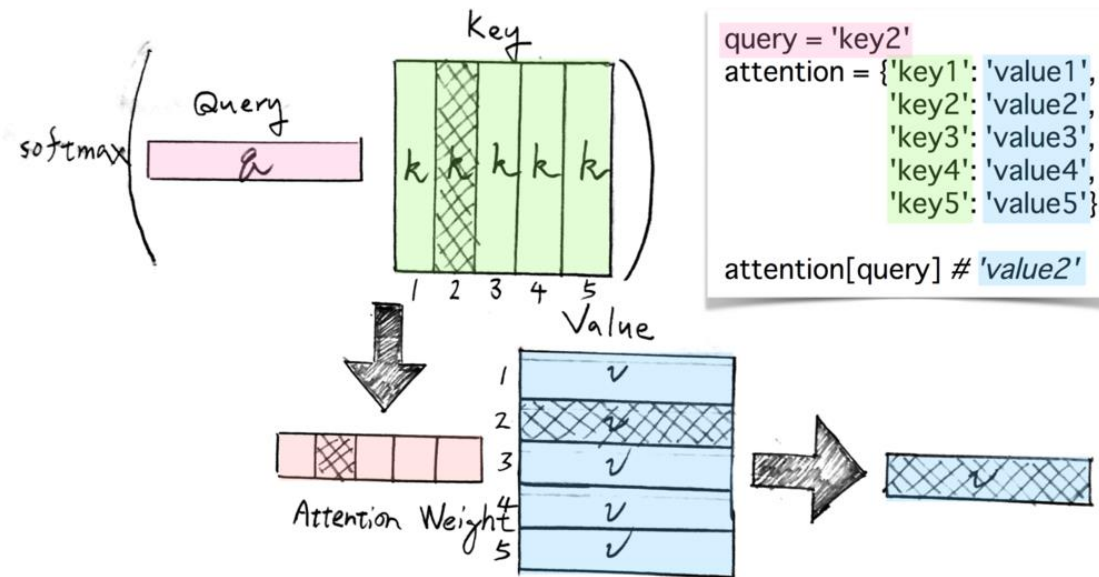
Attention Modules

- Scaled dot-product attention in detail

- $Q(\text{Query}) \in \mathbb{R}^{N \times D_K}$, $K(\text{Key}) \in \mathbb{R}^{M \times D_K}$, $V(\text{Value}) \in \mathbb{R}^{M \times D_V}$

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

[Query 벡터 하나의 attention 계산 과정]



Query 벡터가 Key와의
유사도 정보를 반영한
벡터로 변형

Attention Modules

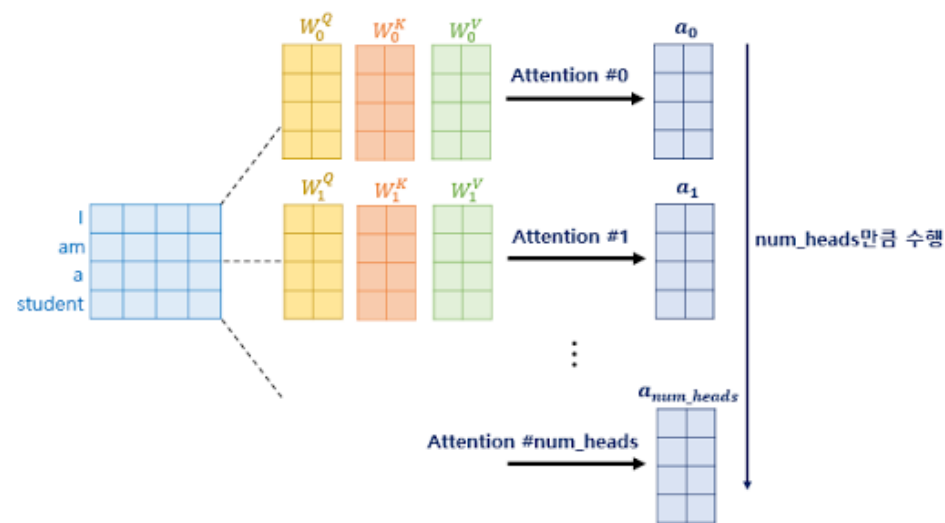
Multi-head Attention

- $N \times D_m$ 크기의 embedding에 대한 $Q(\text{Query}) \in \mathbb{R}^{N \times D_K}$, $K(\text{Key}) \in \mathbb{R}^{M \times D_K}$, $V(\text{Value}) \in \mathbb{R}^{M \times D_V}$ 존재
- 위 Query, Key, Value로 head 의 수(hyper-parameter)만큼, attention을 계산
- 각 head의 output은 concatenate 됨
- Concatenate 된 각 head들의 output은 원래 D_{model} 의 차원으로 다시 사영되어 다음 layer에 전달

cf. the i -th row in Q is the query q_i

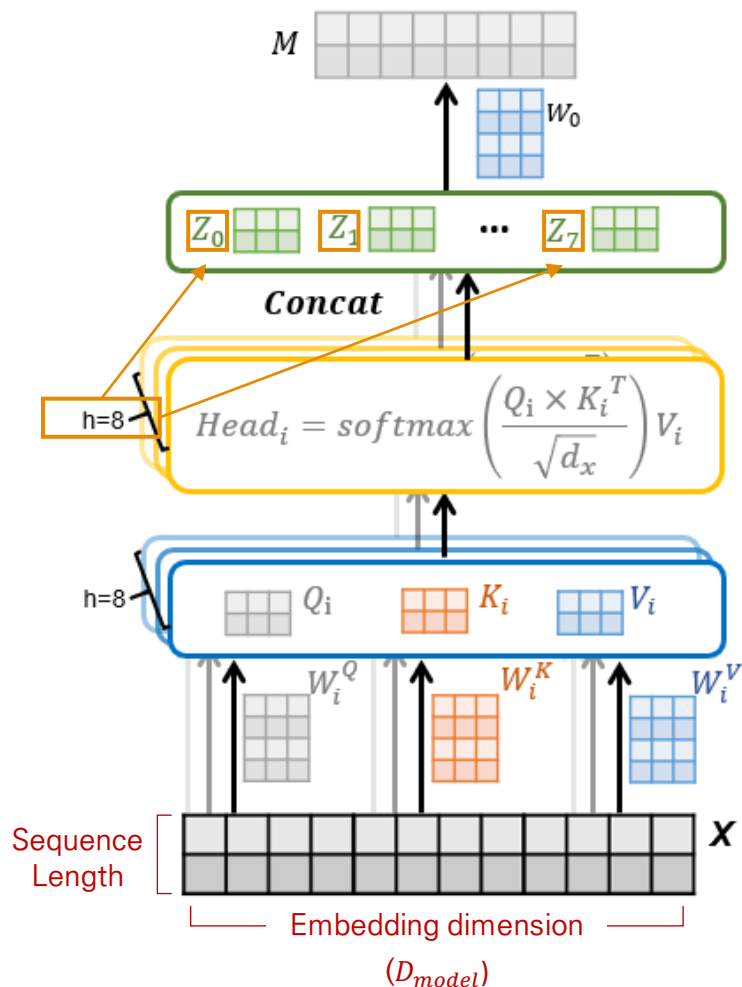
$$\text{MultiHeadAttn}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_H) W^O,$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.



Attention Modules

Multi-head Attention



Concatenate each attention head's output

각 attention head의 output concatenate

head 수만큼 attention 계산

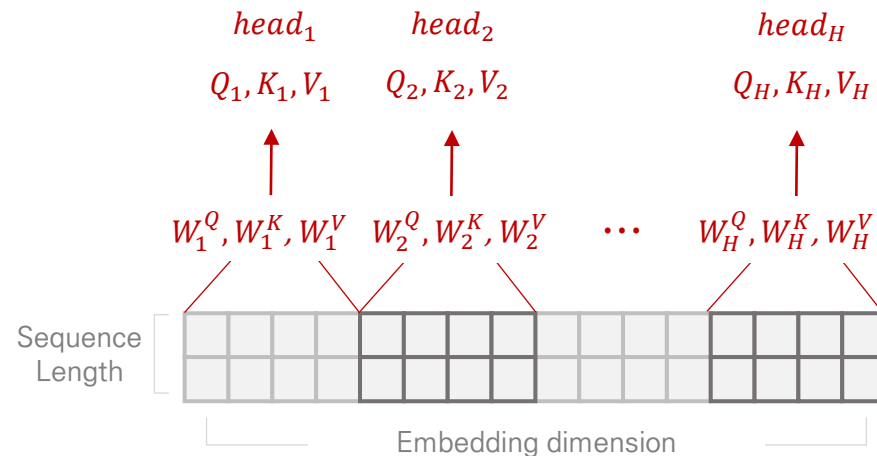
Weight matrices for Q, K, V

Sequence Embedding

[Discussion]

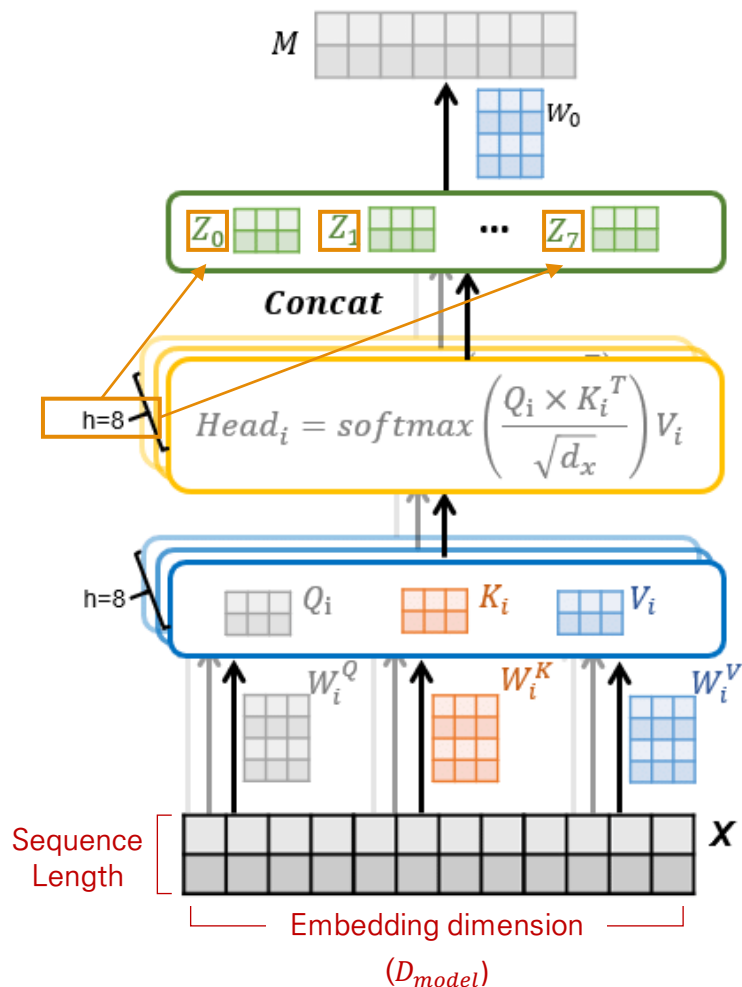
[1] Sequence Embedding에서 embedding dimension을 head의 수만큼 쪼개어 attention을 구하는 것이 아니라 **Input Sequence embedding 전체를 Query, Key, Value를 구하는 weight matrices를 통해 특정 차원을 가지는 Query, Key, Value를 matrices를 만듦**

[※ 잘못된 이해]



Attention Modules

Multi-head Attention



Concatenate each attention head's output

각 attention head의 output concatenate

head 수만큼 attention 계산

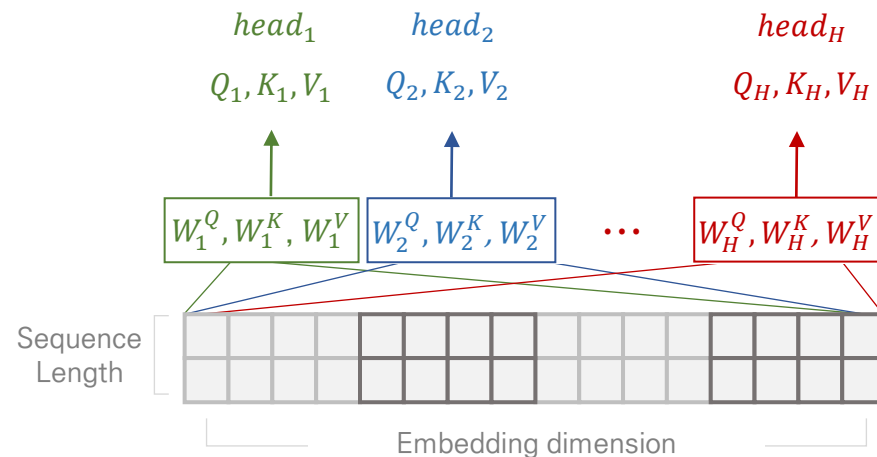
Weight matrices for Q, K, V

Sequence Embedding

[Discussion]

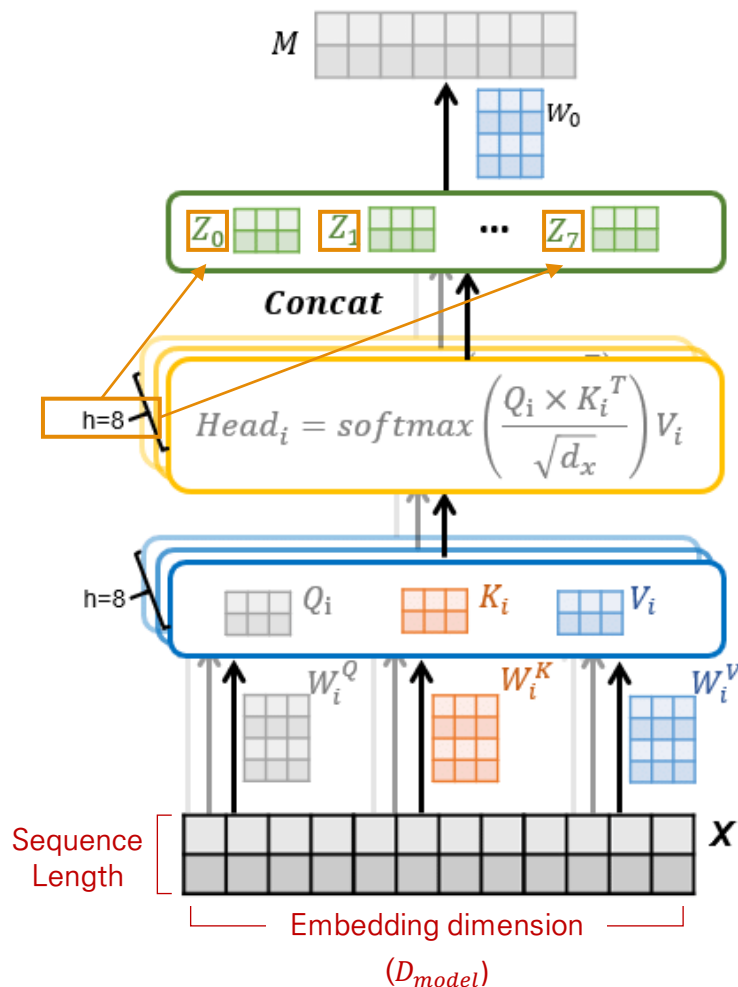
[1] Sequence Embedding에서 embedding dimension을 head의 수만큼 쪼개어 attention을 구하는 것이 아니라 **Input Sequence embedding 전체를 Query, Key, Value를 구하는 weight matrices를 통해** 특정 차원을 가지는 Query, Key, Value를 만듦

[※ 올바른 이해]



Attention Modules

Multi-head Attention



Concatenate each attention head's output

각 attention head의 output concatenate

head 수만큼 attention 계산

Weight matrices for Q, K, V

Sequence Embedding

[Discussion]

[2] 다음 수식에서, Concatenate한 각 head의 attention output을 W^O 을 통해 D_{model} 크기로 사영시키는 이유는? 특히, transformer 논문 구현 상 concatenate된 후의 dimension과 W^O 으로 사영시키고자 하는 dimension이 D_{model} 로 서로 같은데, 왜 동일한 차원으로 다시 사영시키는가?

[참고]

$$\text{MultiHeadAttn}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_H) W^O,$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

[답변]

- 1) Concatenate를 구성하는 값들은 순서대로 각 head의 특성을 반영하고 있게 되는데, 이러한 위치에 따른 각 head의 특성 정보를 W^O projection을 통해 mix하여 사용하고자 함
- 1) 기본적으로 transformer 구조는 각 layer의 input output의 차원을 동일하게 가져가고자 함 (D_{model} 차원 유지)

1) Vanilla Transformer

▪ Attention Modules

- Multi-head Attention

[Discussion]

[3] Multi-head Attention 코드 구현 (PyTorch)

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

```
class Transformer(nn.Module):  
    ''' A sequence to sequence model with attention mechanism. '''  
  
    def __init__(  
        self, n_src_vocab, n_trg_vocab, src_pad_idx, trg_pad_idx,  
        d_word_vec=512, d_model=512, d_inner=2048,  
        n_layers=6, n_head=8, d_k=64, d_v=64, dropout=0.1, n_position=200,  
        trg_emb_prj_weight_sharing=True, emb_src_trg_weight_sharing=True,  
        scale_emb_or_prj='prj'):
```

✓ Transformer 논문에서 언급한 대로 head 수는 8, key 와 value의 dimension은 $d_{\text{model}}/n_{\text{head}}$ 인 64로 정의

▪ Attention Modules

- Multi-head Attention

```
class Encoder(nn.Module):  
    ''' A encoder model with self attention mechanism. '''  
  
    def __init__(  
        self, n_src_vocab, d_word_vec, n_layers, n_head, d_k, d_v,  
        d_model, d_inner, pad_idx, dropout=0.1, n_position=200, scale_emb=False):  
  
        super().__init__()  
  
        self.src_word_emb = nn.Embedding(n_src_vocab, d_word_vec, padding_idx=pad_idx)  
        self.position_enc = PositionalEncoding(d_word_vec, n_position=n_position)  
        self.dropout = nn.Dropout(p=dropout)  
  
        self.layer_stack = nn.ModuleList([  
            EncoderLayer(d_model, d_inner, n_head, d_k, d_v, dropout=dropout)  
            for _ in range(n_layers)])  
  
        self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)  
        self.scale_emb = scale_emb  
        self.d_model = d_model
```

✓ Encoder(혹은 Decoder) Layer에
다음과 같이 dimension의 크기를 전달

▪ Attention Modules

- Multi-head Attention

```
class EncoderLayer(nn.Module):  
    ''' Compose with two layers '''  
  
    def __init__(self, d_model, d_inner, n_head, d_k, d_v, dropout=0.1):  
        super(EncoderLayer, self).__init__()  
        self.slf_attn = MultiHeadAttention(n_head, d_model, d_k, d_v, dropout=dropout)  
        self.pos_ffn = PositionwiseFeedForward(d_model, d_inner, dropout=dropout)  
  
    def forward(self, enc_input, slf_attn_mask=None):  
        enc_output, enc_slf_attn = self.slf_attn(  
            enc_input, enc_input, enc_input, mask=slf_attn_mask)  
        enc_output = self.pos_ffn(enc_output)  
        return enc_output, enc_slf_attn
```

✓ 앞서 전달 받은 dimension 크기를
Multi-head Attention을 계산하는 클래스에 전달

Attention Modules

Multi-head Attention

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

```
    ''' Multi-Head Attention module '''
```

```
    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
```

```
        super().__init__()
```

```
        self.n_head = n_head
```

```
        self.d_k = d_k
```

```
        self.d_v = d_v
```

```
        self.w_qs = nn.Linear(d_model, n_head * d_k, bias=False)
```

```
        self.w_ks = nn.Linear(d_model, n_head * d_k, bias=False)
```

```
        self.w_vs = nn.Linear(d_model, n_head * d_v, bias=False)
```

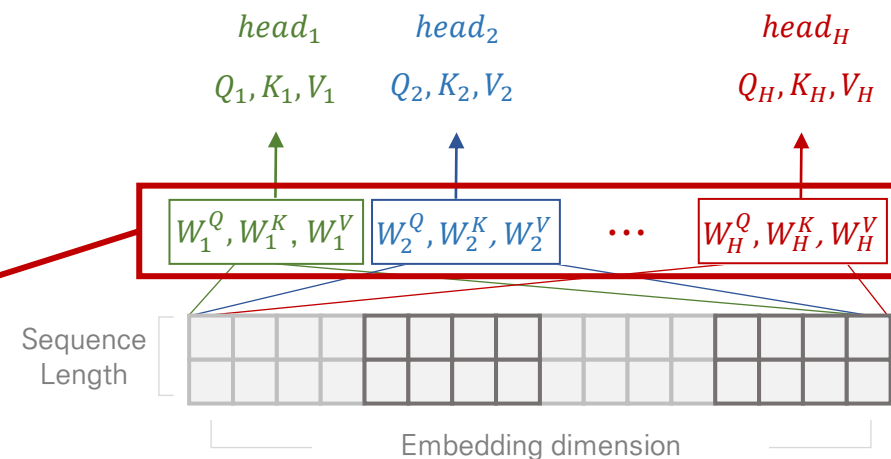
```
        self.fc = nn.Linear(n_head * d_v, d_model, bias=False)
```

```
        self.attention = ScaledDotProductAttention(temperature=d_k ** 0.5)
```

```
        self.dropout = nn.Dropout(dropout)
```

```
        self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)
```

[그림 p10]



- ✓ Query, Key, Value를 projection하는 Weight matrices 생성
- ✓ Weight matrices를 처음부터 독립적으로 head 수만큼 정의하지 않고, $n_{head} \times d_{k(v)}$ 의 dimension을 갖는 weight matrix를 만든 후 나눠서 사용하는 방식

■ Attention Modules

• Multi-head Attention

```
def forward(self, q, k, v, mask=None):
```

```
    d_k, d_v, n_head = self.d_k, self.d_v, self.n_head
    sz_b, len_q, len_k, len_v = q.size(0), q.size(1), k.size(1), v.size(1)
```

```
    residual = q
```

```
    # Pass through the pre-attention projection: b x lq x (n*dv)
    # Separate different heads: b x lq x n x dv
    q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
    k = self.w_ks(k).view(sz_b, len_k, n_head, d_k)
    v = self.w_vs(v).view(sz_b, len_v, n_head, d_v)

    # Transpose for attention dot product: b x n x lq x dv
    q, k, v = q.transpose(1, 2), k.transpose(1, 2), v.transpose(1, 2)
```

```
    if mask is not None:
        mask = mask.unsqueeze(1) # For head axis broadcasting.
```

```
    q, attn = self.attention(q, k, v, mask=mask)
```

```
    # Transpose to move the head dimension back: b x lq x n x dv
    # Combine the last two dimensions to concatenate all the heads together: b x lq x (n*dv)
    q = q.transpose(1, 2).contiguous().view(sz_b, len_q, -1)
    q = self.dropout(self.fc(q))
    q += residual

    q = self.layer_norm(q)

    return q, attn
```

- ✓ 앞서 생성한 Weight Matrix의 차원을 조정함으로써 head 수만큼 Query, Key, Value Weight의 영역 구분
- ✓ 즉, $sequence_length \times (n_head \times d_{k(v)})$ 의 크기를 갖는 Weight Matrix를 기반으로 n_heads 의 수만큼 attention이 계산될 수 있도록 차원을 변경하여 사용 = 병렬적으로 계산 가능!
- ✓ Attention 계산 후, 차원을 다시 조정하여 concatenate된 matrix 생성

1) Vanilla Transformer

- Attention Modules

- Three types of attention

- 1) Self-attention

- $Q=K=V=X$ (the outputs of the previous layer)

- 2) Masked Self-attention (Transformer Decoder Only)

- Decoder 부분에서, 해당 시점에 token 대한 output을 생성하는 데 있어서, 해당 시점 이후의 token 정보를 사용하지 않고자 함
- generation의 autoregressive 또는 causal한 특성 반영
- attention score matrix에서 masking 할 부분의 값을 $-\infty$ 으로 대체

- 3) Cross-attention (Transformer Decoder Only)

- Encoder-Decoder attention
- Query : previous decoder layer output로부터 생성
- Key & Value : outputs of encoder로부터 생성

Scores
(before softmax)

0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Apply Attention
Mask

Masked Scores
(before softmax)

0.11	$-\infty$	$-\infty$	$-\infty$
0.19	0.50	$-\infty$	$-\infty$
0.53	0.98	0.95	$-\infty$
0.81	0.86	0.38	0.90

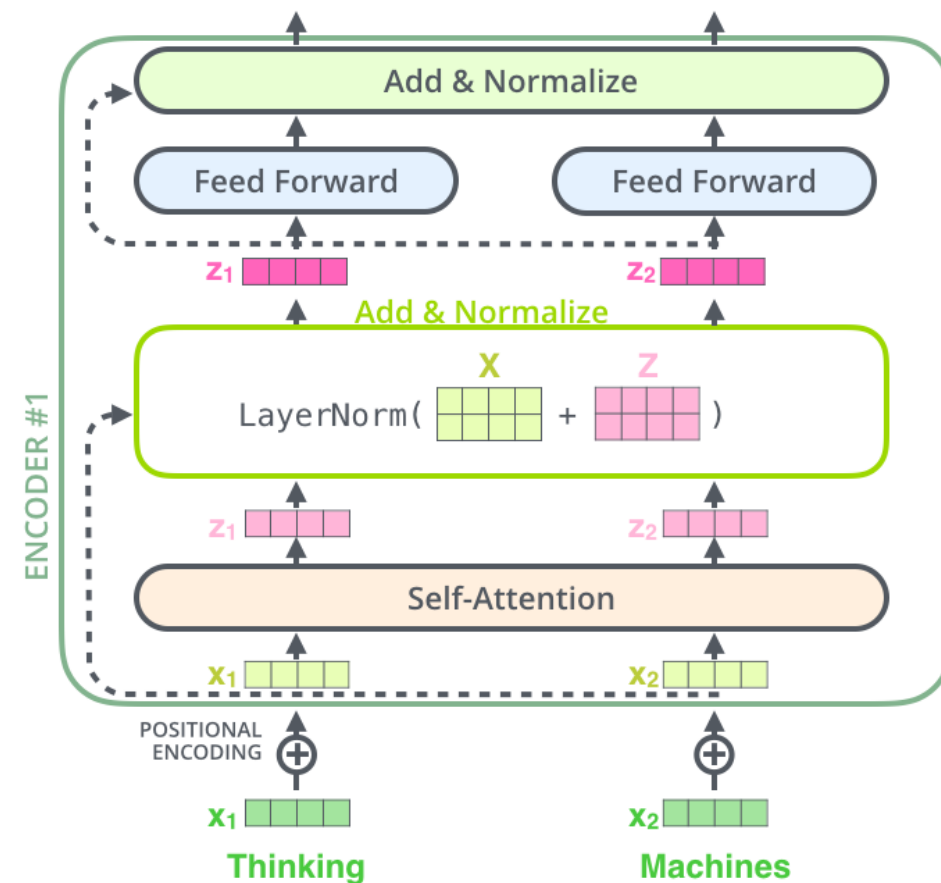
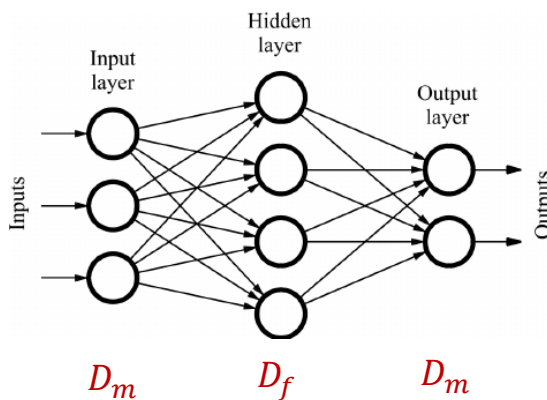
input sequence 전반에 걸쳐서
decoder가 attend할 수 있도록 함

Position-wise FFN

- fully connected feed-forward module를 적용하는 부분
- position 마다, 즉 개별 단어마다 적용되기 때문에 position-wise
- 한 블록 내에서, 단어 간의 FFN의 parameter는 공유됨

$$\text{FFN}(H') = \text{ReLU}(H'W^1 + b^1)W^2 + b^2$$

- H' : 이전 layer의 output
- $W^1 \in \mathbb{R}^{D_m \times D_f}$, $W^2 \in \mathbb{R}^{D_f \times D_m}$, $b^1 \in \mathbb{R}^{D_f}$, $b^2 \in \mathbb{R}^{D_m}$
- 보통, $D_f > D_m$ 으로 설정

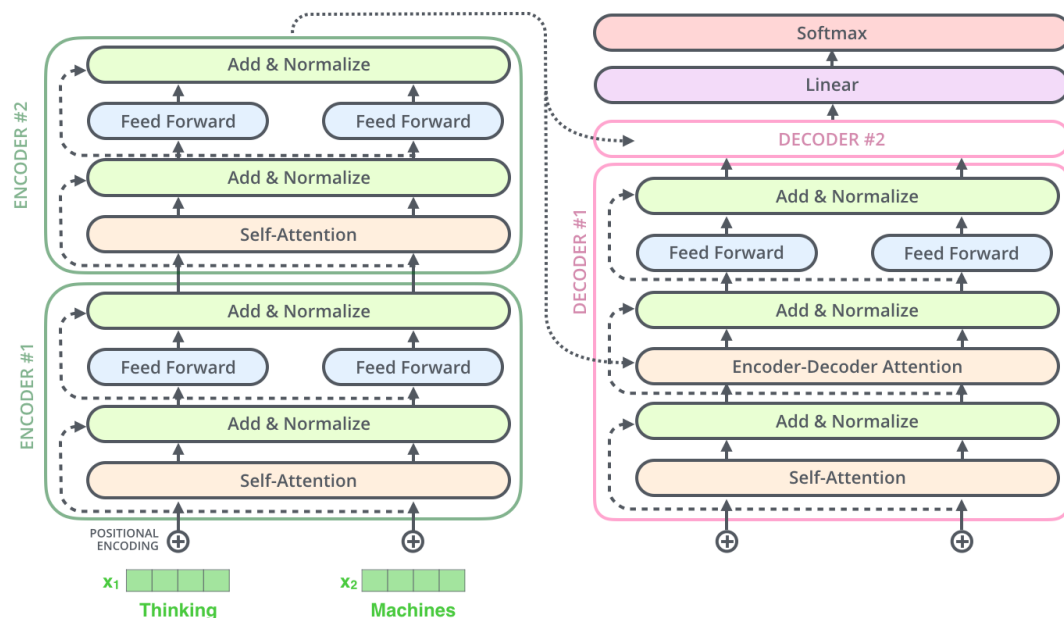


Residual Connection and Normalization

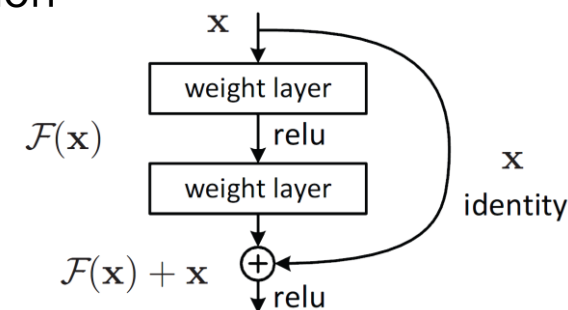
- Gradient exploding/vanishing 문제를 완화하고, deep한 네트워크를 안정적으로 학습하기 위해 도입

$$H' = \text{LayerNorm}(\text{SelfAttention}(X) + X)$$

$$H = \text{LayerNorm}(\text{FFN}(H') + H')$$

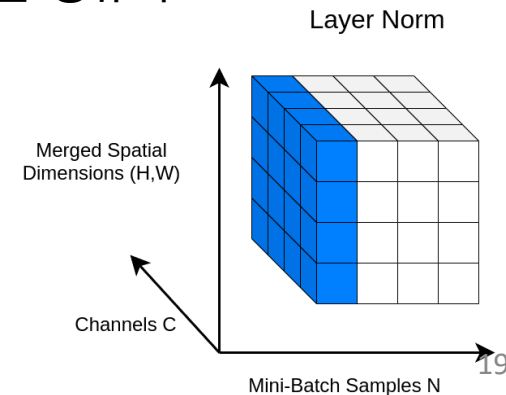


Residual Connection



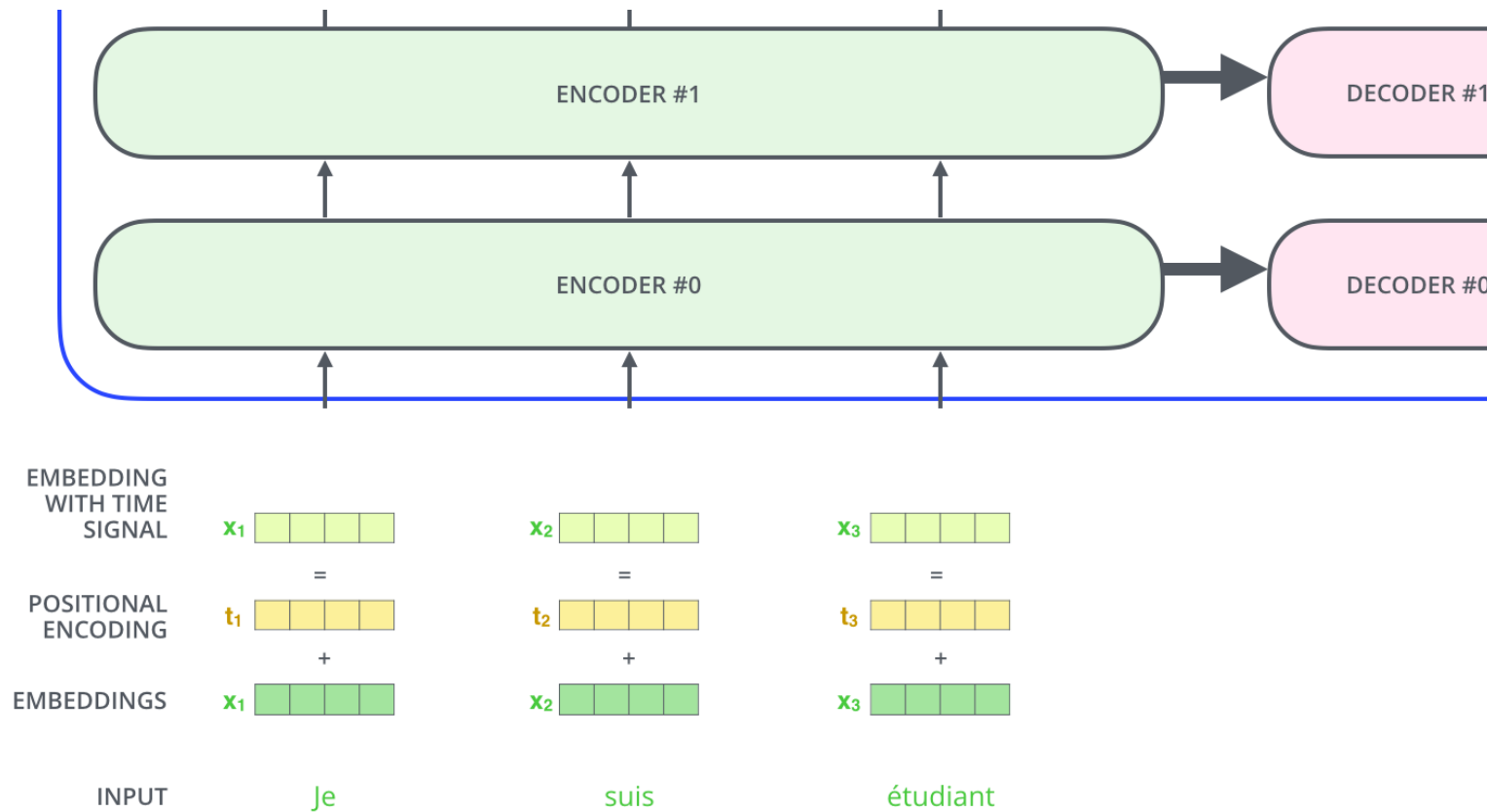
Layer Normalization

- 각 중간층의 출력을 정규화



Position Encodings

- token의 순서정보를 반영해주기 위해 position encoding 사용



▪ Transformer Architecture의 활용 방식

- Encoder-Decoder
 - Full Transformer Architecture
 - 보통 sequence to sequence modeling(e.g. neural machine translation)에서 사용
- Encoder only
 - Transformer의 Encoder만 사용
 - Encoder의 output은 input sequence의 representation으로 활용
 - 보통 Encoder로 생성된 representation을 바탕으로 classification이나 sequence labeling problem에 사용
- Decoder only
 - 기존 Transformer 구조의 Decoder 내의 encoder-decoder cross-attention 모듈은 제거되고 사용
 - 보통 sequence generation (e.g. language modeling)에 사용

연산 복잡도와 parameter 수 분석

- Self-attention과 position-wise FFN 모듈
- D : hidden dimension of the model (D_m)
- T : input sequence length
- $4D$: intermediate dimension of FFN
- D/H : the dimension of keys and values

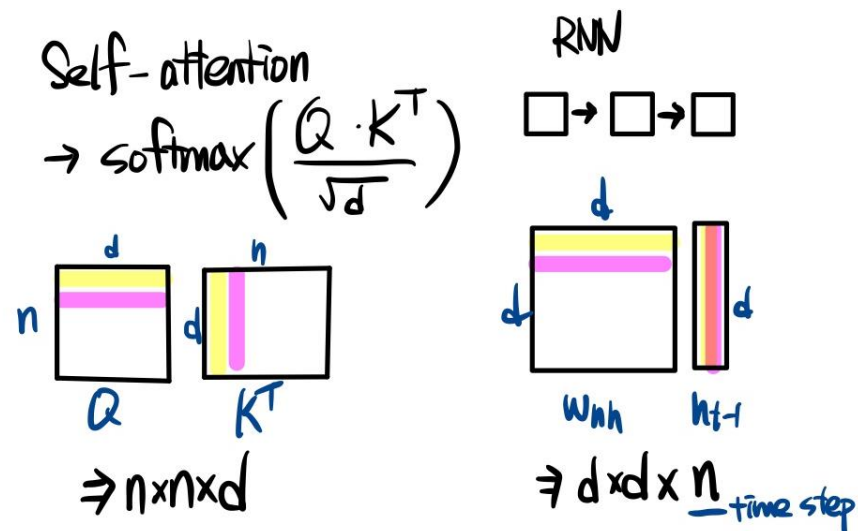


Table 1. Complexity and parameter counts of self-attention and position-wise FFN

Module	Complexity	#Parameters
self-attention	$O(T^2 \cdot D)$	$4D^2$
position-wise FFN	$O(T \cdot D^2)$	$8D^2$

▪ Self-attention 분석

- Self-attention은 variable-length input에 대해 유연한 매커니즘을 지님
- complexity, sequential operations, maximum path length 측면에서 layer 타입 비교
 - 1) fully connected layer처럼 maximum path length 값이 같음 어떤 시점에서 비교하고 싶은 다른 시점의 token이 있을 때, 필요한 계산 횟수
 - long – range dependencies modeling에 적합
 - parameter-efficient, variable-length input에 대해 유연
 - 2) constant maximum path length는 constant number of layer가 long-range dependencies를 잘 모델링할 수 있도록 함 (\Leftrightarrow convolution layer는 receptive field가 제한적)
 - 3) constant sequential operation과 maximum path length는 self-attention을 병렬처리가 가능하도록 하여 recurrent layer보다 long-range modeling에 더 적합

* The maximum length of the paths forward and backward signals have to traverse to get from any input position to arbitrary output position. Shorter length implies a better potential for learning long-range dependencies.

Self-attention 분석

- ✓ total computational complexity per layer
- ✓ parallelized될 수 있는 computation의 양
- ✓ network 상에서 long range dependency의 path length (→ path length가 짧아질수록 long range dependency를 학습하기 훨씬 쉬움)

Table 2. Per-layer complexity, minimum number of sequential operations and maximum path lengths for different layer types. T is the sequence length, D is the representation dimension and K is the kernel size of convolutions [137].

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(T^2 \cdot D)$	$O(1)$	$O(1)$
Fully Connected	$O(T^2 \cdot D^2)$	$O(1)$	$O(1)$
Convolutional	$O(K \cdot T \cdot D^2)$	$O(1)$	$O(\log_K(T))$
Recurrent	$O(T \cdot D^2)$	$O(T)$	$O(T)$

일반적으로 $T < D$ 이므로,
연산량이 가장 적음

병렬화 가능

long-term dependency
문제 해결

- Inductive Bias 측면

- CNN, RNN 과 비교

- Convolution : inductive biases of translation invariance(물체의 위치변화)and locality(window 내만 본다) with shared local kernel functions
- Recurrent : inductive biases of temporal invariance(시간 정보를 가지고 있음, 과거의 시간 정보) and locality(보게 될 time step 범위 정해줌) via their Markovian structure
- Transformer: few assumptions
 - time step이나, filter size 등에 대한 구조적 가정 없음
 - Universal, flexible architecture
 - structural bias의 부족은 small-scale data에 overfitting될 문제가 있음 → 조금 더 큰 데이터 필요

▪ Inductive Bias 측면

- GNN과의 비교
 - Transformer = complete directed graph 구조의 GNN(self-loop)으로 볼 수 있음
 - Transformer는 input data 구조에 대해 prior knowledge 사용 안함
 - message passing(노드 간 정보 공유)이 content 간의 유사도에만 의존
 - graph structure(그래프 형태, 노드, 인접행렬)와 같은 prior knowledge 필요 없음

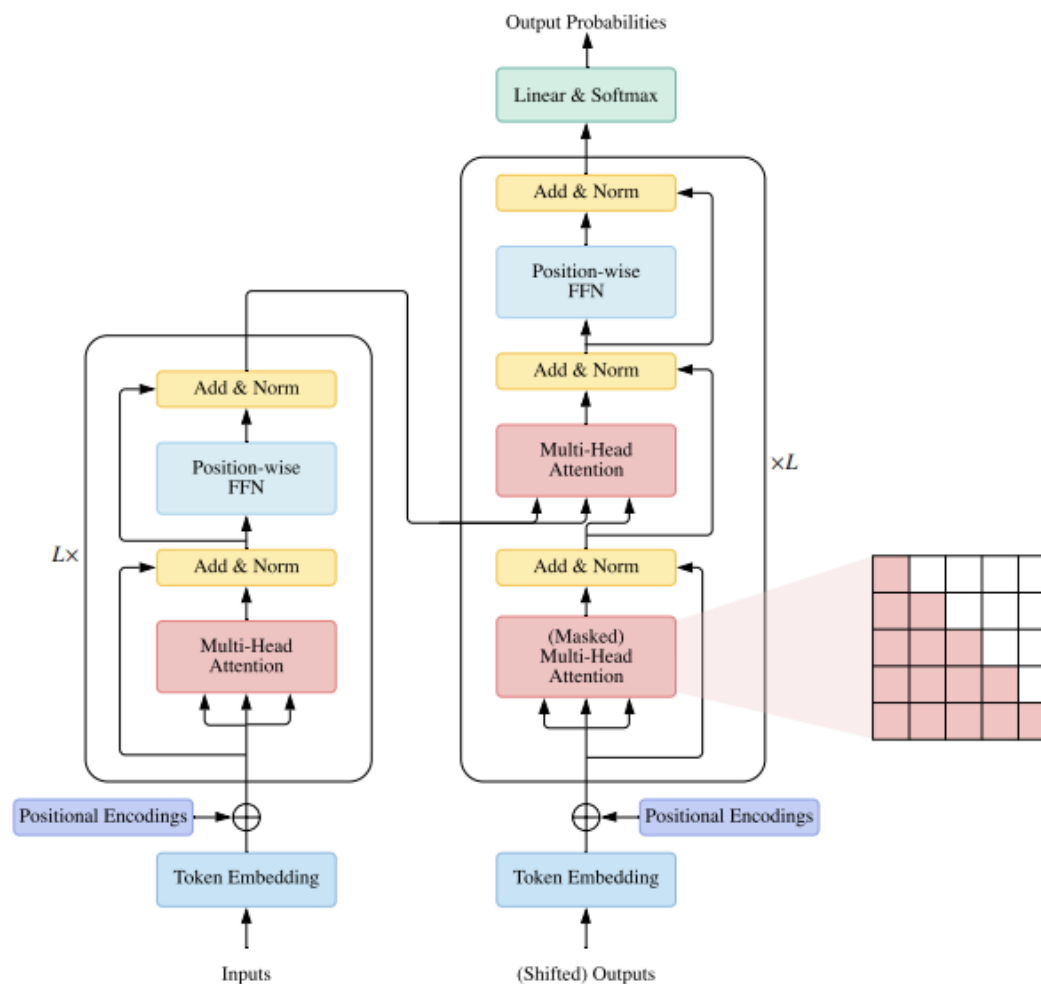


Fig. 1. Overview of vanilla Transformer architecture