Attention Is All you need

NeurIPS 2017 (# Citation: 170,153회)

가짜연구소 김 성 은

2025.03.13

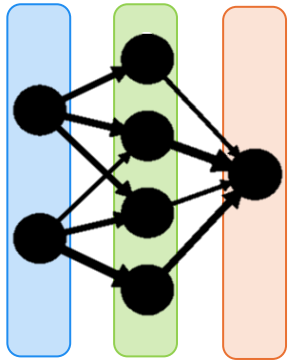
Background

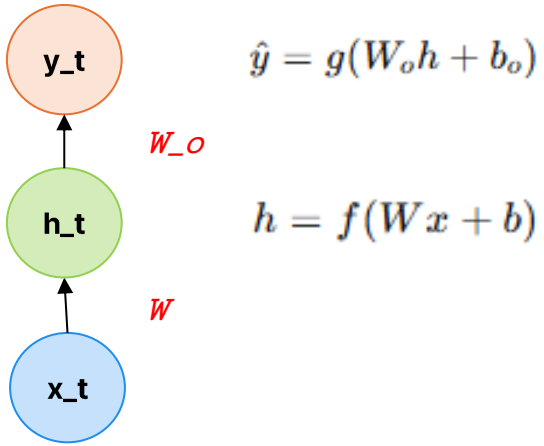
FNN (Feedforward Neural Network)

입력 계층(input layer): 외부에서 데이터를 받아 전달하는 계층

은닉 계층(hidden layer): 데이터의 특징을 추출하는 계층

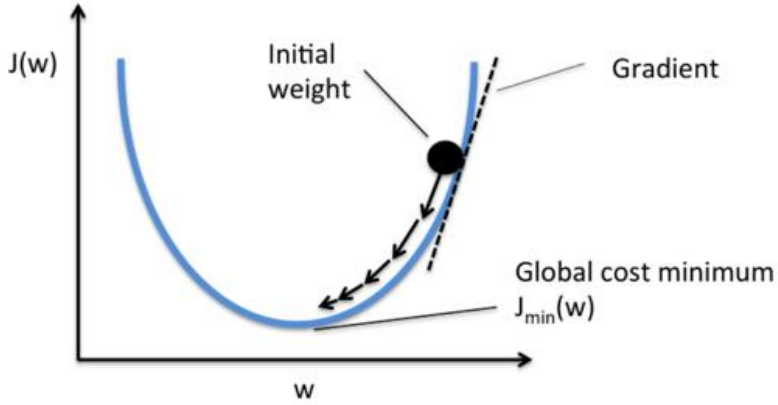
출력 계층(output layer): 추출된 특징을 기반으로 외부에 출력하는 계층





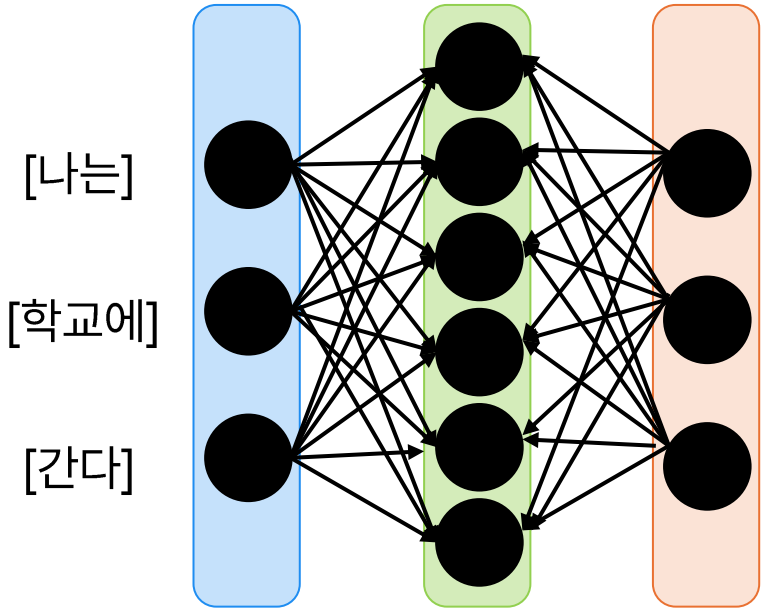
$$\hat{y} = g(W_o h + b_o)$$

$$h = f(Wx + b)$$



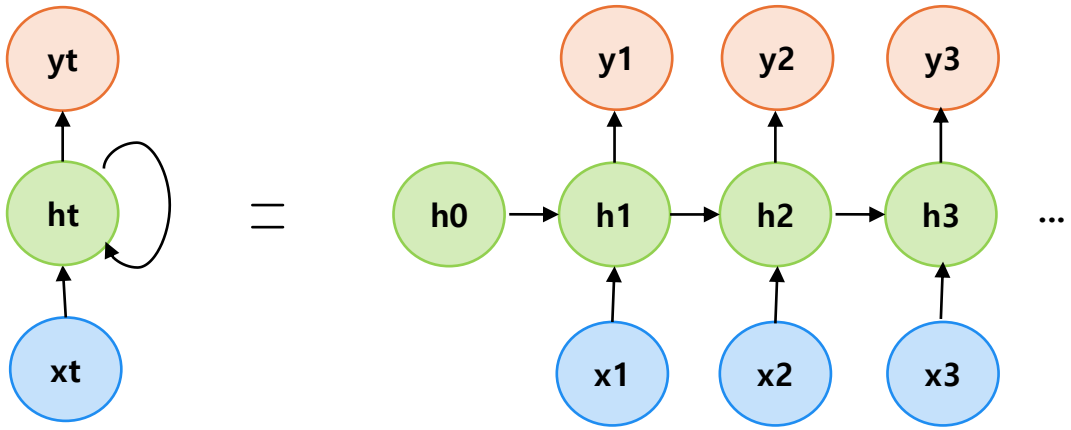
FNN의 한계점

- FNN은 단어 하나하나를 개별적으로 처리하므로, 문장의 흐름을 고려할 수 없음
- 그렇다면, 이전의 정보를 기억해서 문장의 흐름을 고려할 수 있게 할 수는 없을까? -> YES

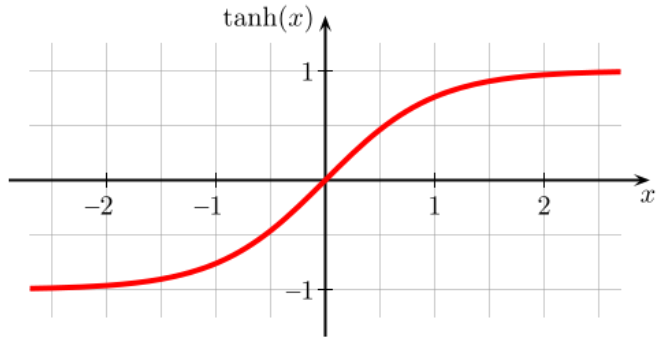


RNN (Recurrent Neural Network)

- 이전 상태(hidden state)를 기억하면서 연속적인 데이터를 처리하는 신경망 -> 이전 정보를 기억 -> 문맥 고려

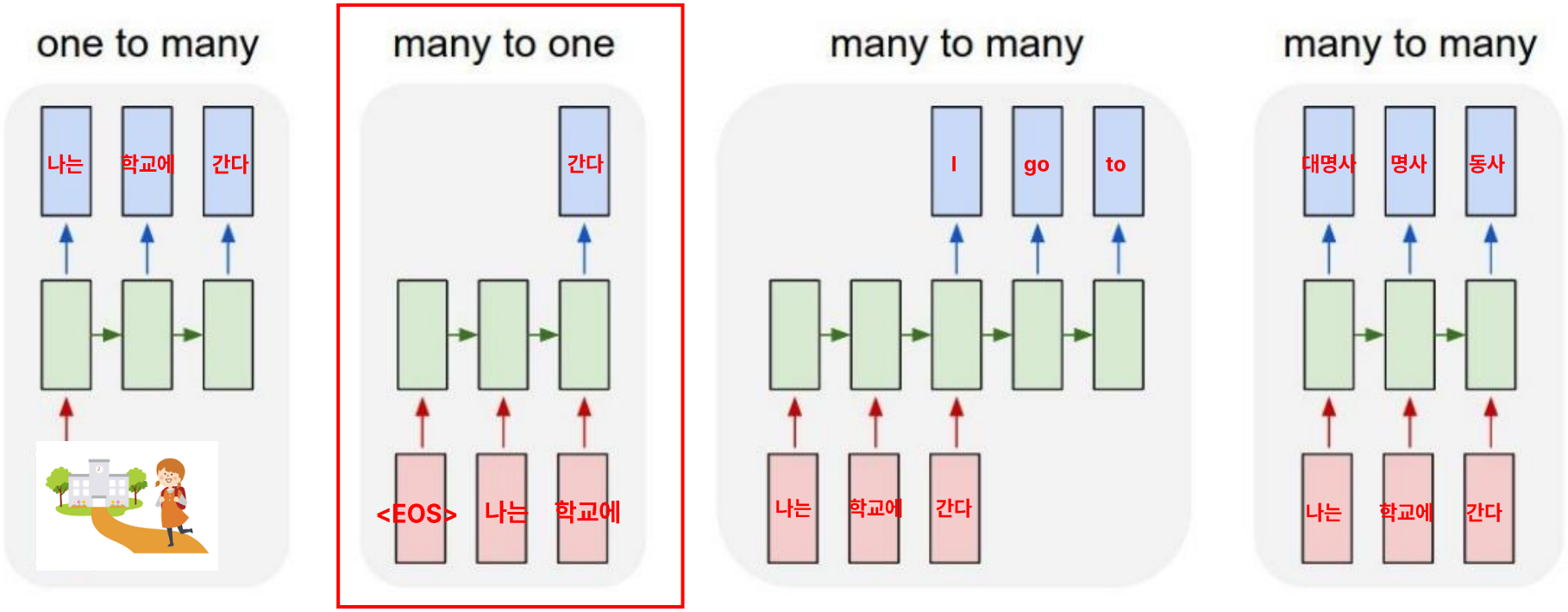


$$h_t = tanh(W_x x_t + W_h h_{t-1} + b)$$
$$y_t = f(W_y h_t + b)$$



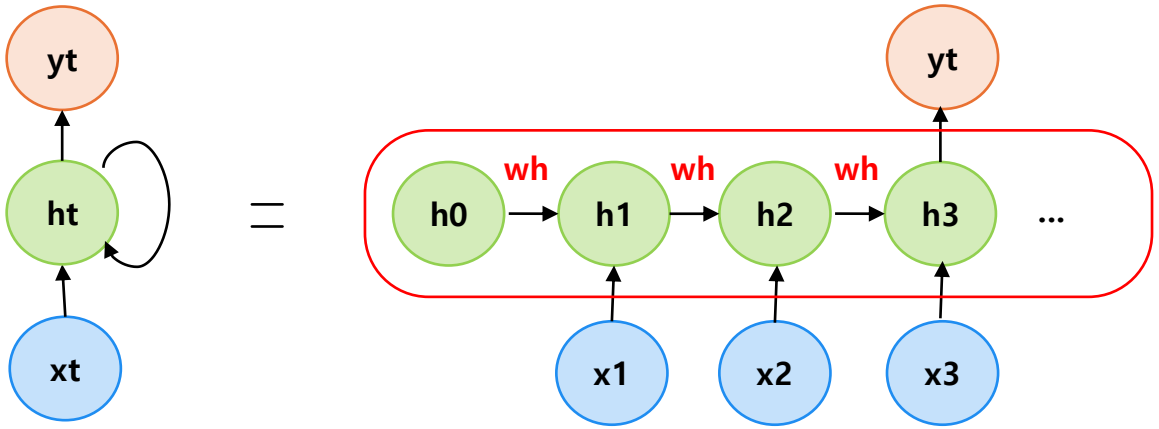
RNN (Recurrent Neural Network)

- RNN은 다양한 구조를 통해 연속적인 데이터를 처리할 때 사용됨



RNN의 한계점

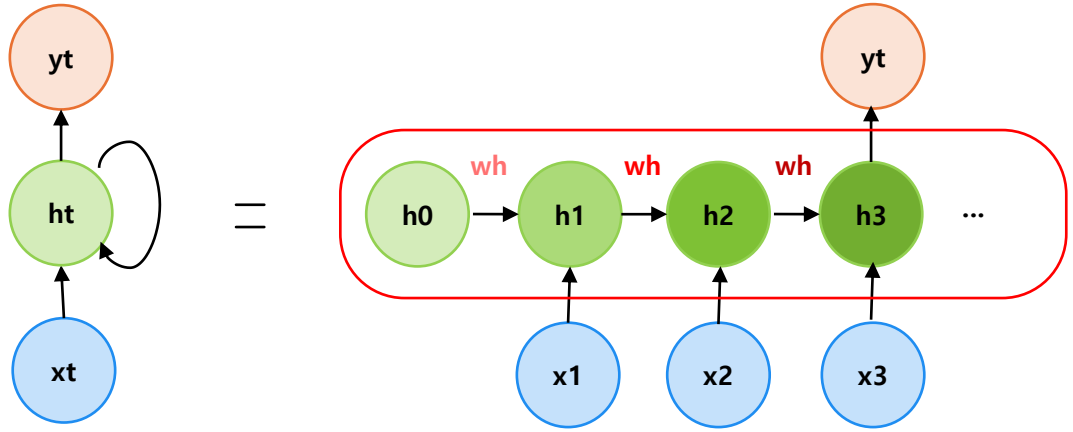
- 나는 아침에 일어나서 운동을 하고, 샤워를 한 뒤, 밥을 먹고, 책을 읽고, [학교에 간다] ?



장기 의존성(Long-Term Dependency) 문제

과거 정보가 현재까지 잘 전달되지 않는 문제

RNN의 한계점



$$h_t = tanh(W_x x_t + W_h h_{t-1} + b)$$
$$y_t = f(W_y h_t + b)$$

h_1 = h_0 x w_(h)

h_2 = h_1 x w_(h)

= h_0 x w_(h) x w_(h)

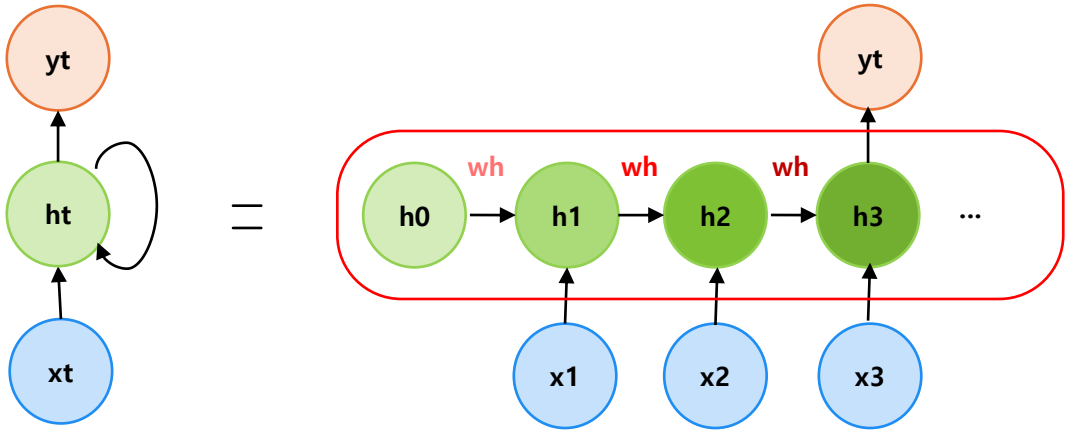
h_3 = h_2 x w_(h)

= h_0 x w_(h) x w_(h) x w_(h)

...

h_t = h_0 x w_(h)^t

RNN의 한계점



$$h_t = h_0 \times w_{(h)}^{t}$$

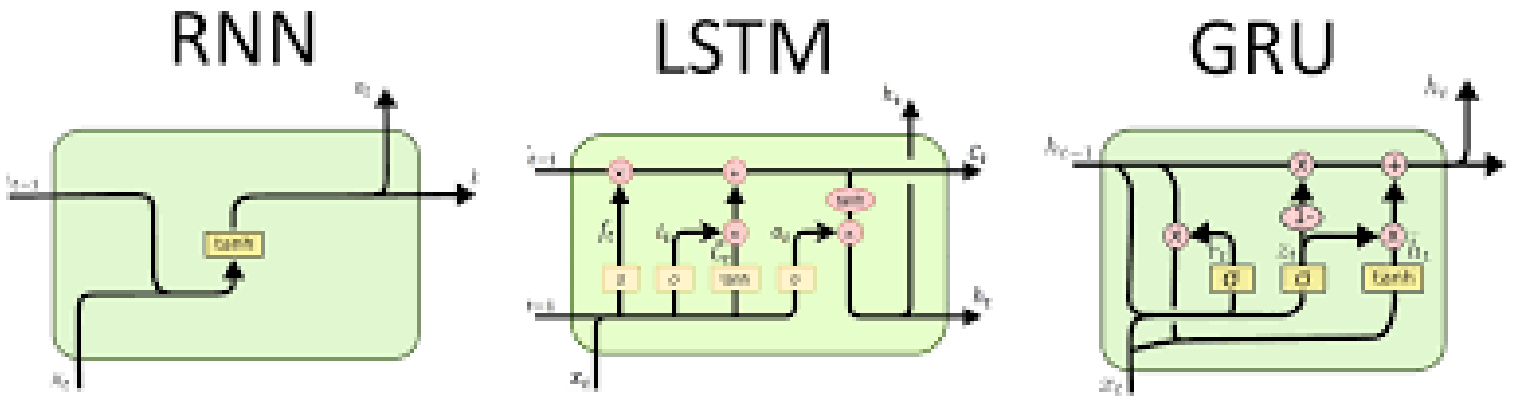
w_(h)

- >1, → ∞ (exploding gradient) → gradient clipping
  // 학습이 더 이상 진행불가능
- <1, → 0 (vanishing gradient) → Gated RNN: LSTM, GRU
  // 학습이 잘 된 것인지, 아닌지 도중 파악 어려움

LSTM, GRU

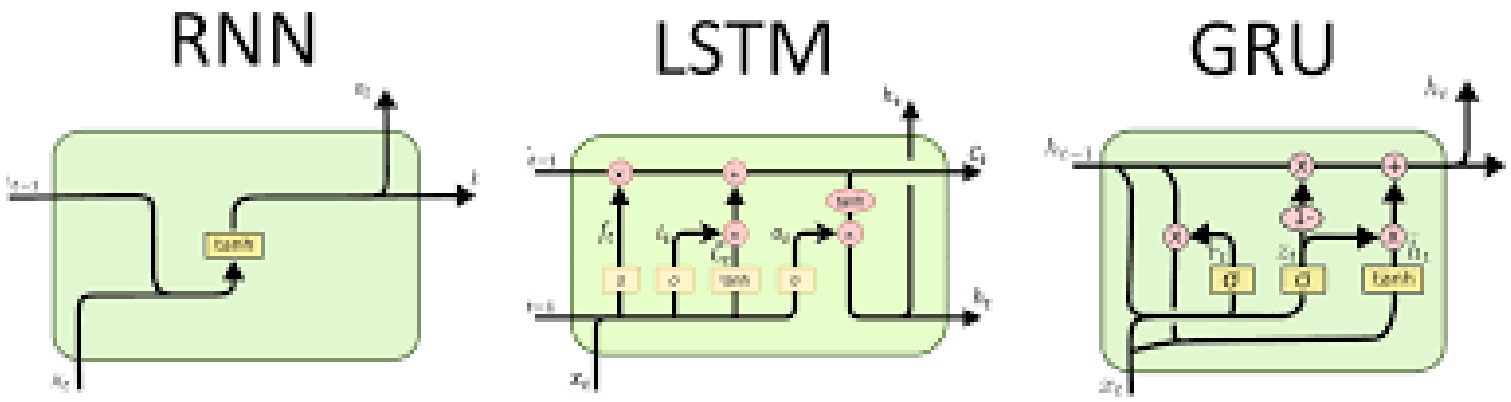
- 기울기 소실 문제 (Vanishing gradient)
- 장기 의존성 문제 (Long-Term Dependency)

해결하기 위해 → LSTM, GRU 등장

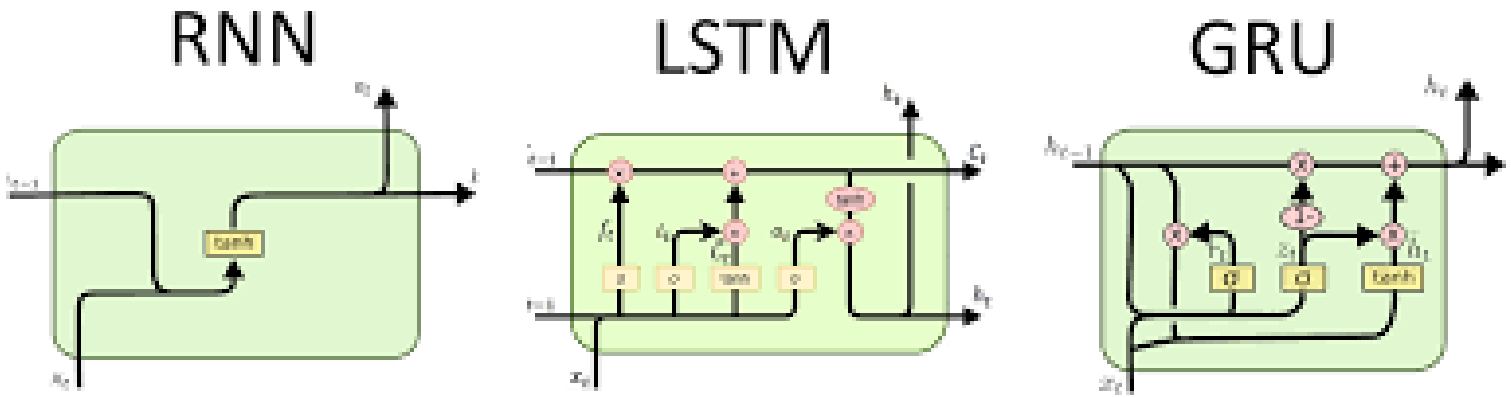


LSTM, GRU

- LSTM: Forget Gate, Input Gate, Output Gate를 추가하여 중요한 정보는 유지하고 불필요한 정보는 잊도록 설계
- GRU: Update Gate, Reset Gate로 더 단순한 구조이며, 효과적으로 정보 보존 가능



LSTM, GRU의 한계점



아직 긴 문장을 처리하기에 한계가 있는 것은 사실이고,

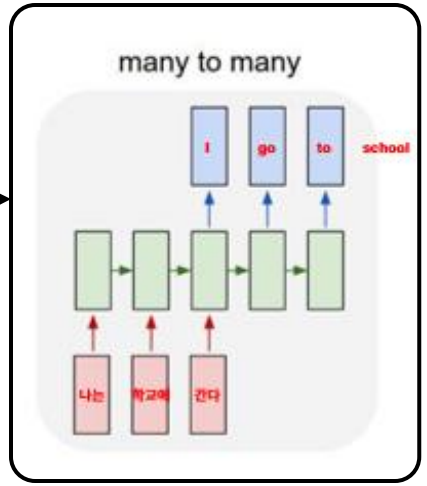
순차적으로 데이터를 처리하므로 병렬 연산 불가 → 속도가 느림

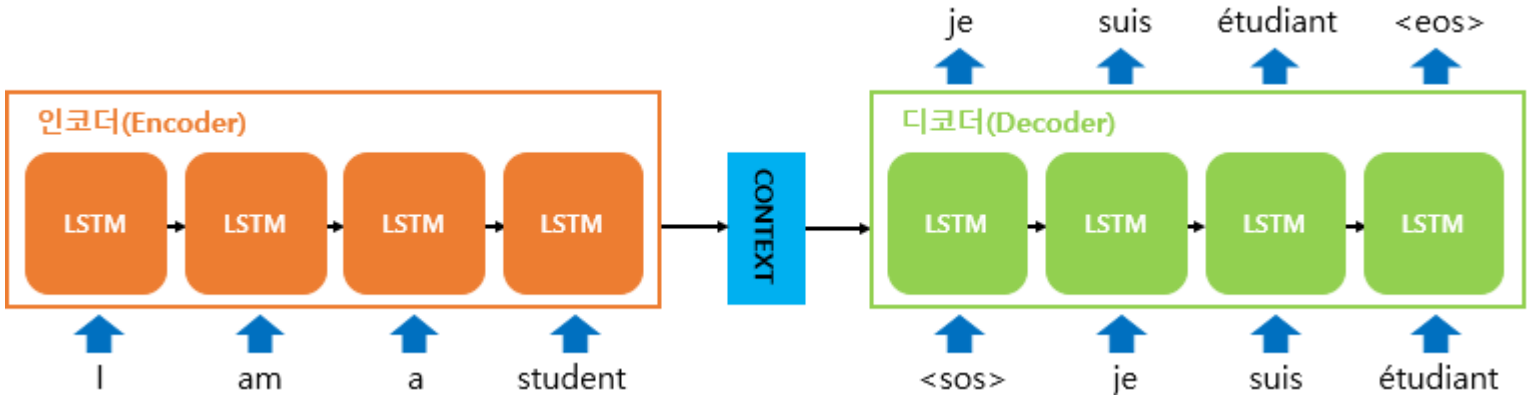
Seq2Seq

- 번역, 문장 요약, 질의 응답 등 다양한 NLP 작업에서 활용됨
- RNN의 입력과 출력의 길이가 다른 문제를 해결하기 위해

  입력 문장을 인코더에서 고정된 컨텍스트 벡터(Context Vector)로 변환 후,

  고정된 컨텍스트 벡터(Context Vector)을 기반으로 새로운 시퀀스를 생성

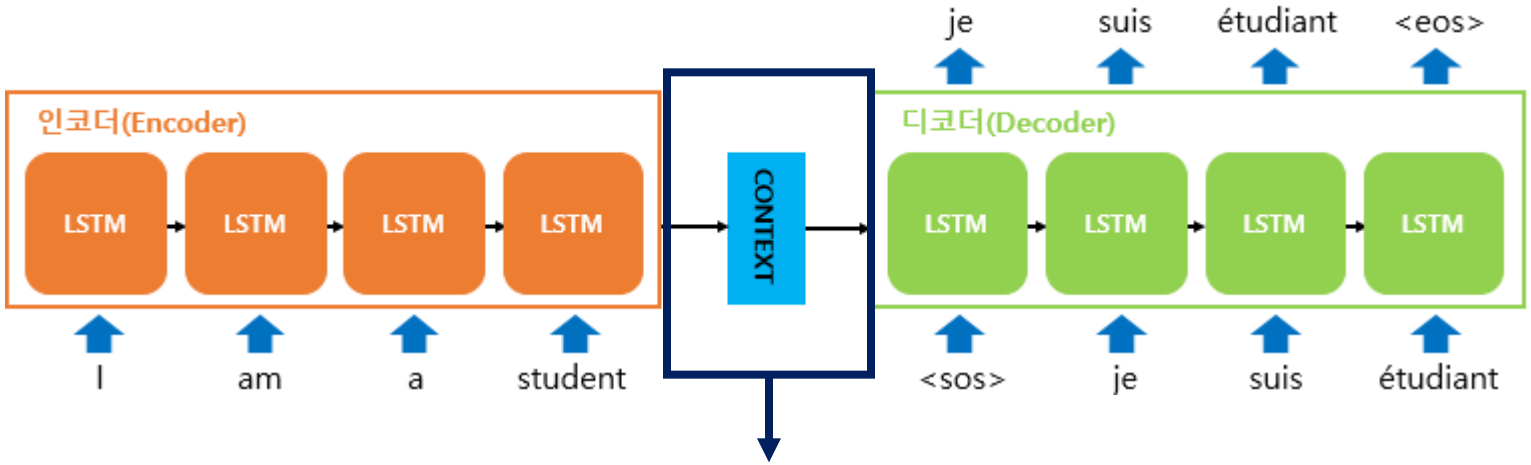




Seq2Seq의 한계점

- 나는 아침에 일어나서 운동을 하고, 샤워를 한 뒤, 밥을 먹고, 책을 읽고, 학교에 간다

  → I wake up in the morning, exercise, take a shower, eat breakfast, read a book, and go to school.



위와 같이 긴 문장에서 정보 손실 발생

문장이 길어질수록 초반의 정보가 희미해짐

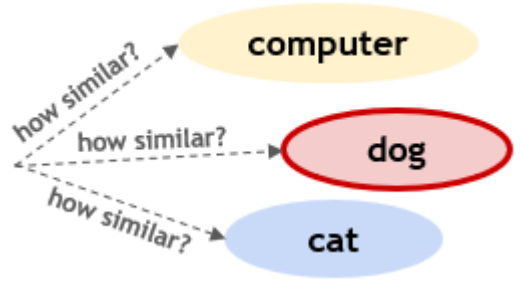
번역 과정에서 뒤쪽 단어만 반영되고 앞쪽 단어가 무시될 가능성이 높아짐

Attention의 등장

- Seq2Seq의 고정된 컨텍스트 벡터(context vector) 때문에,
  문장이 길어질수록 초반의 정보가 희미해지는 문제 발생 → 동적으로 참고할 수 없을까?
- 이러한 문제를 해결하기 위해 **Attention 개념**이 등장함!
- **Attention**이란, Query와 비슷한 값을 가진 Key를 찾아서 Value를 얻는 과정

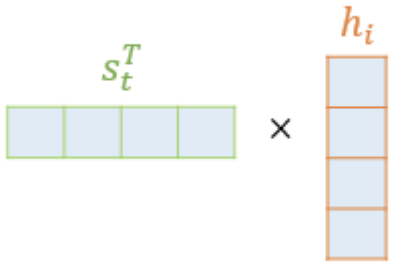
dictionary = {**'computer'**: 9, **'dog'**: 2, **'cat'**: 3}

query: **puppy**



seq2seq with Attention

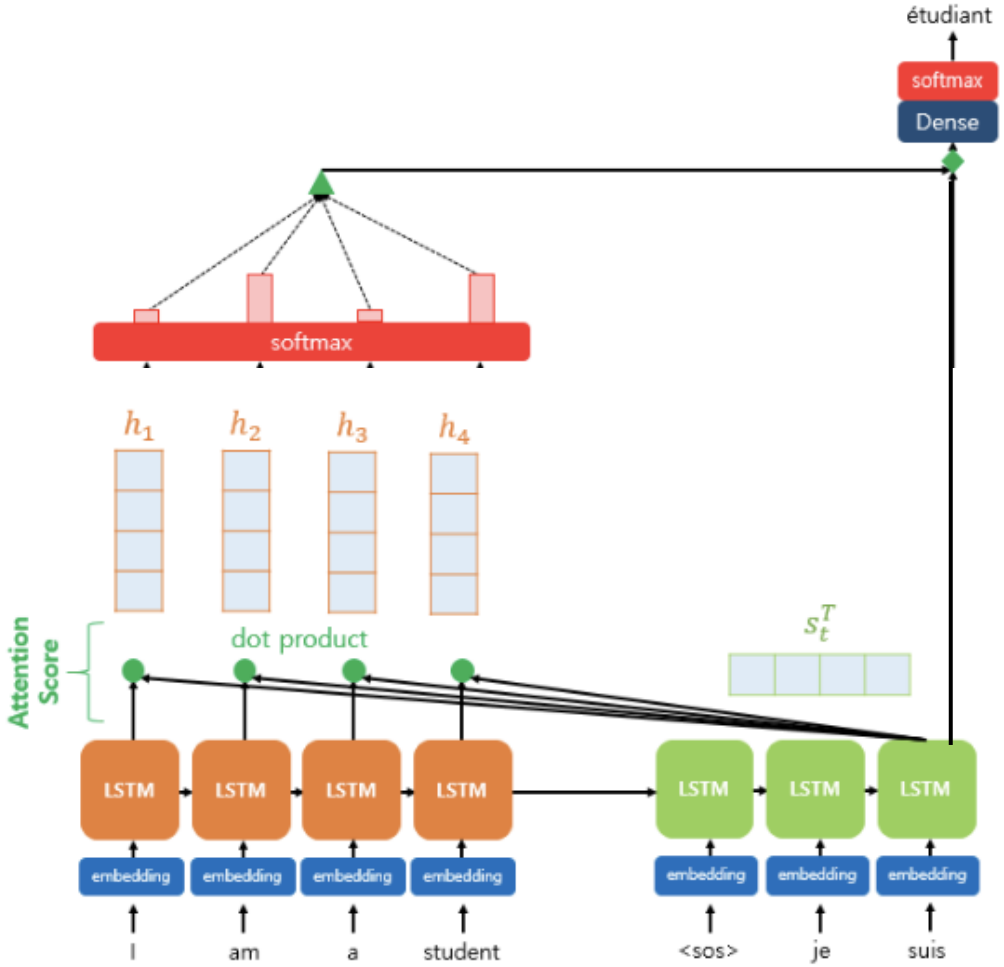
- Attetion은 디코더 내의 타겟 단어를 예측할 때마다,
  인코더의 내의 source 문장들 중
  어느 입력 단어들과 가장 유관한지를 탐색하는 것



$$score(s_t,\ h_i) = s_t^T h_i$$

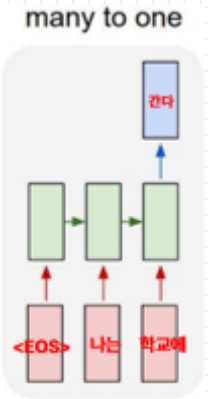
$$e^t = [s_t^T h_1, \ldots, s_t^T h_N]$$

$$\alpha^t = softmax(e^t)$$



- **FNN 한계** → 독립적 -> 문맥 X
- 연속적인 데이터(문맥)를 처리할 수 있는 **RNN 등장!!**

- **다음 문장의 예측 예시**



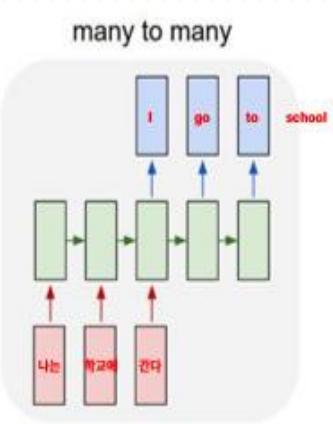
  - 장기 의존성 문제 (Long-Term Dependency)
  - 기울기 소실 문제 (Vanishing gradient)

  → **LSTM, GRU** 등장

  장기 의존성 완전 해결 X

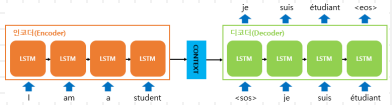
  병렬 처리 X -> 속도 ↓

- **번역 작업 예시**



  - 입력과 출력의 길이가 다름에 따른 오류

  → **Seq2Seq** 등장



  Context vector

  긴 문장 정보 손실

  → **Attention** 등장

  Query와 비슷한 Key를 찾아 Value를 얻는 과정

  → **Seq2Seq + Attention**

Transformer

- 2017년 “Attention Is All You Need”의 논문에서 Transformer라는 모델을 제안

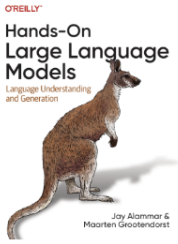
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The Illustrated Transformer

Discussions: Hacker News (65 points, 4 comments), Reddit r/MachineLearning (29 points, 3 comments)
Translations: Arabic, Chinese (Simplified) 1, Chinese (Simplified) 2, French 1, French 2, Italian, Japanese, Korean, Persian, Russian, Spanish 1, Spanish 2, Vietnamese
Watch: MIT's Deep Learning State of the Art lecture referencing this post
Featured in courses at Stanford, Harvard, MIT, Princeton, CMU and others



Update: This post has now become a book! Check out LLM-book.com which contains (Chapter 3) an updated and expanded version of this post speaking about the latest Transformer models and how they've evolved in the seven years since the original Transformer (like Multi-Query Attention and RoPE Positional embeddings).

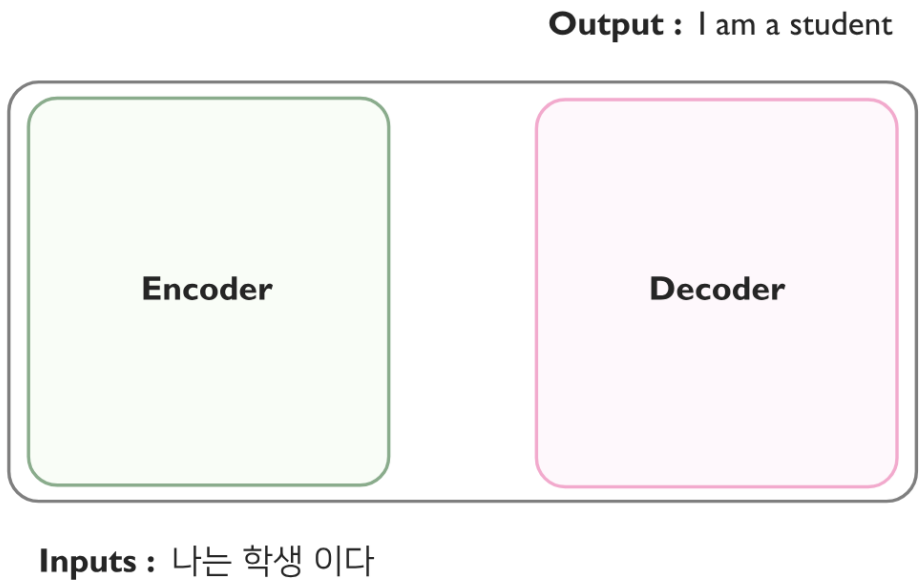
In the previous post, we looked at Attention – a ubiquitous method in modern deep learning models. Attention is a concept that helped improve the performance of neural machine translation applications. In this post, we will look at **The Transformer** – a model that uses attention to boost the speed with which these models can be trained. The Transformer outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation

https://www.dropbox.com/scl/fi/p9vpn7rnrijwnci2wna4m/Transformer-to-LLaMA_.pdf?rlkey=7utoz00ectj7lpet7lslvlsrq&e=3&dl=0

https://github.com/pilsung-kang/Text-Analytics/tree/master/08%20Seq2Seq%20Learning%20and%20Pre-trained%20Models

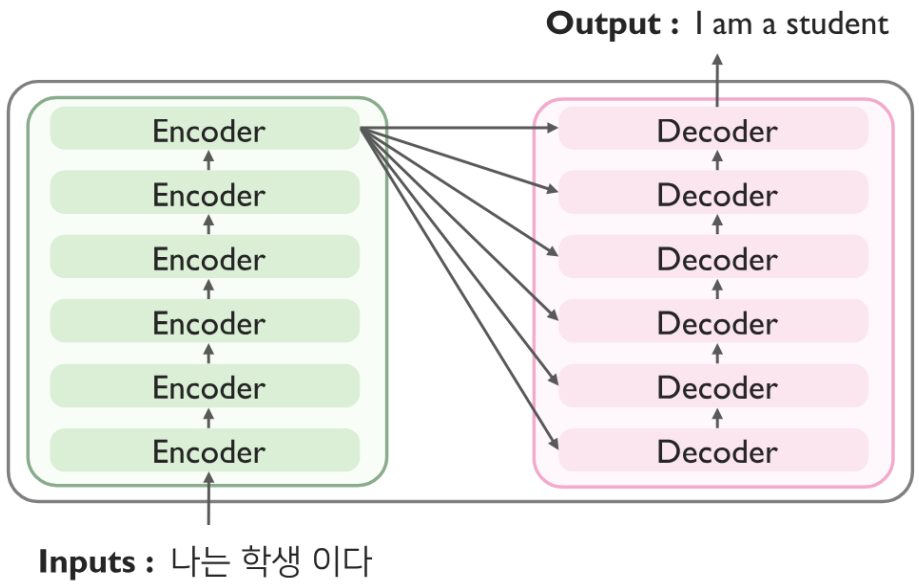
Transformer

- Transformer는 Encoder-Decoder 구조로 구성



Transformer

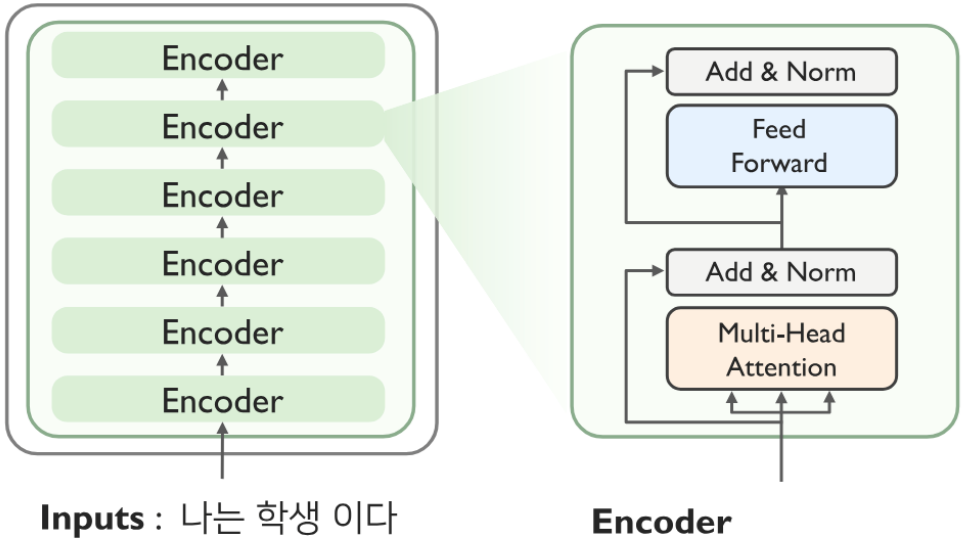
- Transformer는 Encoder-Decoder 구조로 구성
  - 각 N개의 모듈로 구성 (논문에서는 N = 6)



Transformer

- Encoder는

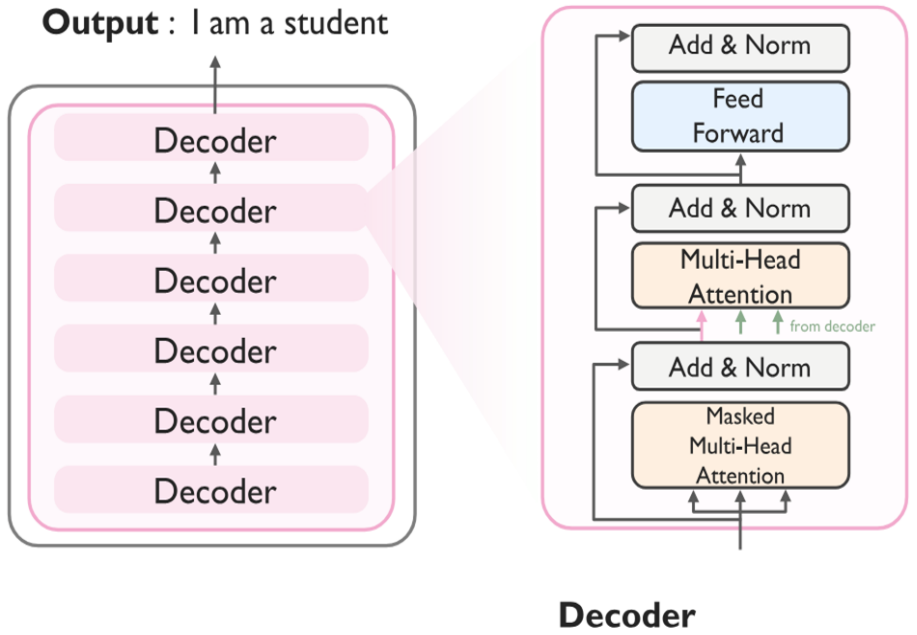
  (1) Mult-Head Attention 과 (2) Feed Forward Network로 구성



Transformer

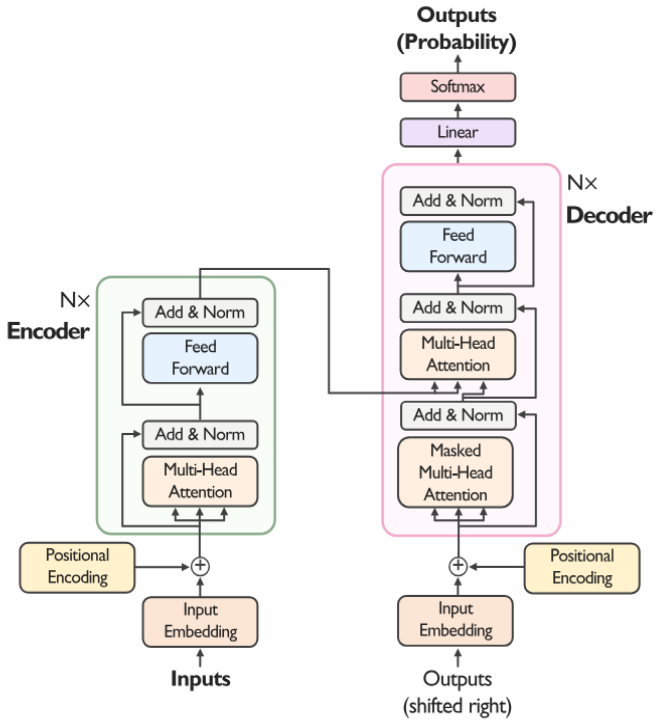
- Decoder는

  (1) Masked Mult-Head Attention , (2) Mult-Head Attention, (3) Feed Forward Network로 구성



Transformer

- 다음 순서에 따라 깊이 이해해봅시다!



① **Embedding**

② **Positional Encoding**

③ **Encoder**

- **Self-Attention**
- **Feed Forward**

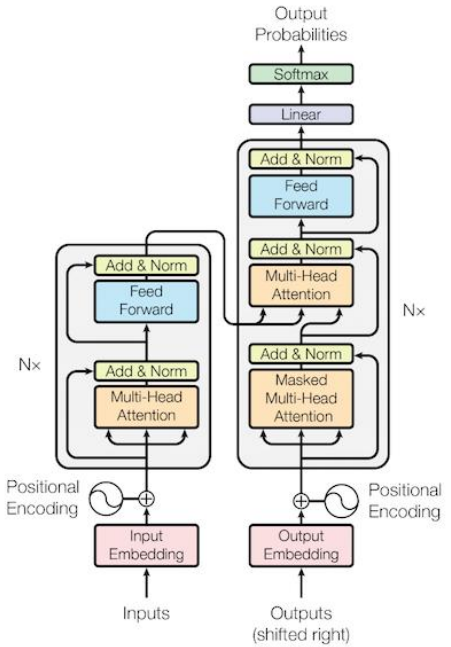
④ **Decoder**

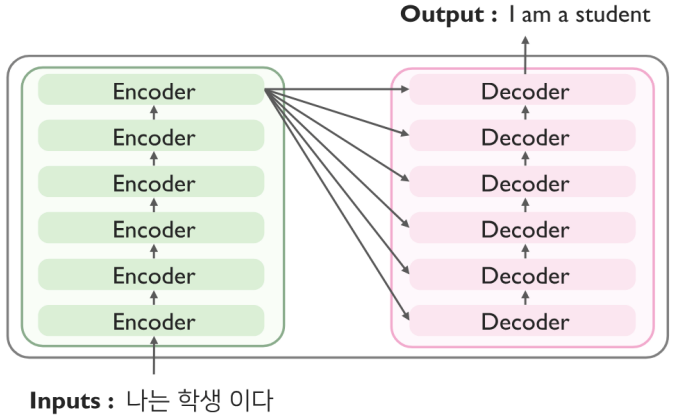
- **Masked Self-Attention**
- **Encoder-Decoder Attention**
- **Feed Forward**

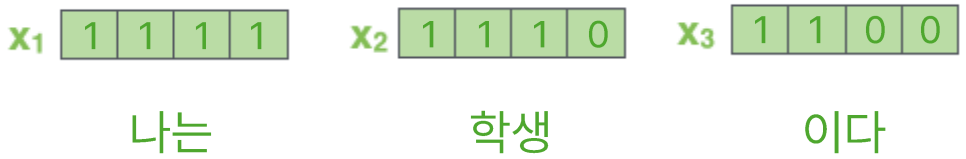
⑤ **Prediction**

Embedding

- 단어(token) 형태의 데이터를 수치로 변환 → 최초의 입력으로 사용
- 논문에서 Base 모델은 512차원의 임베딩을, Big 모델은 1024차원의 임베딩을 사용함
- ~~초기에는 **one-hot vector** 형태로 입력됨~~

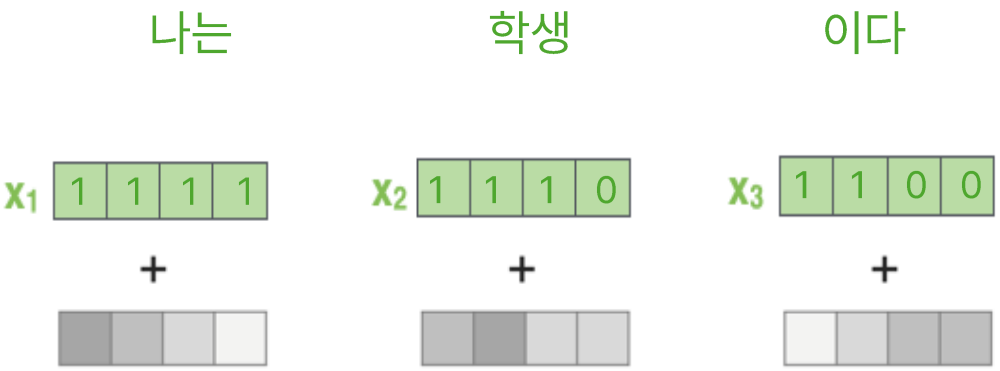


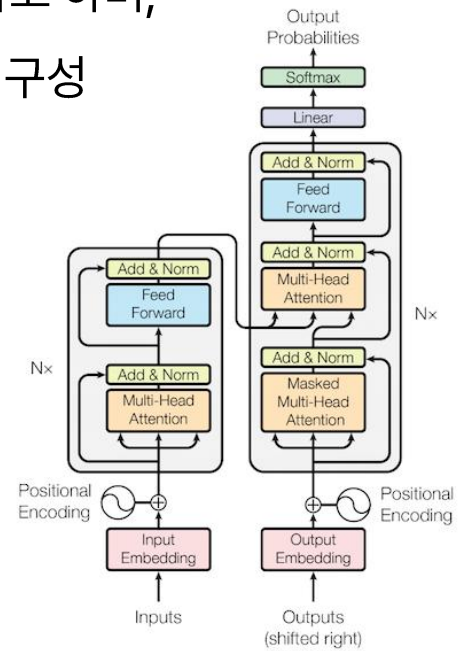




Positional Encoding

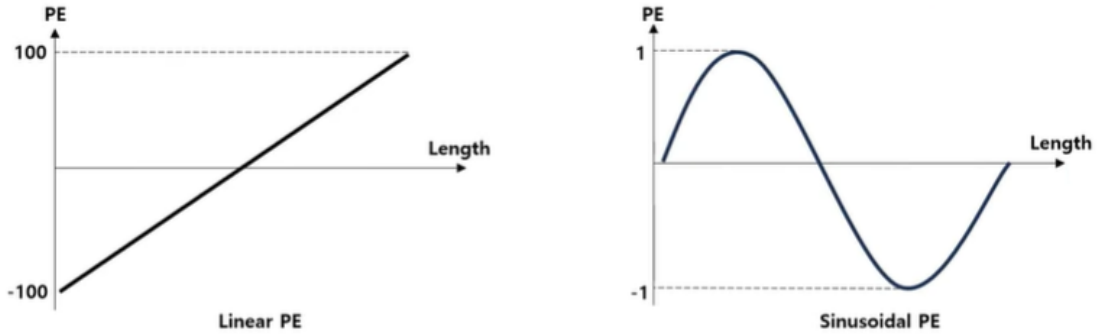
- Transformer는 단어를 순차적으로 받지 않고 한 번에 받는 구조이므로
  입력 시퀀스에서 단어들 간의 위치 관계를 표현해 줄 필요가 있음
- 이러한 역할을 수행하기 위해 설계된 벡터를 Positional Encoding 이라고 하며,
  모든 단어 Embedding에 Positional Encoding을 더해서 입력 벡터를 구성





Positional Encoding

- sin 함수와 cos 함수를 활용하여 아래의 수식에 따라 Positional Encoding 값이 정해짐
  - 왜? → -1 ~ 1 사이에 있어 안정적임



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

https://www.youtube.com/watch?v=-z2oBUZfL2o

Positional Encoding

Position of word

Location of embedding vector

Size of embedding vector

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

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

나는 

학생 

이다 

=

	0 1 2 3
pos = 0	1 1 1 1
pos = 1	1 1 1 0
pos = 2	1 1 0 0

$$\sin(0/(10000^{(2*0)/4})) = 0$$

$$\cos(0/(10000^{(2*1)/4})) = 1$$

$$\sin(0/(10000^{(2*2)/4})) = 0$$

$$\cos(0/(10000^{(2*3)/4})) = 1$$

[잘문]

- sin과 cos을 번갈아서 하는 이유는?
- d_{model}로 나누는 이유는?

Positional Encoding

Position of v

: embedding vector

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

    def __init__(self, d_hid, n_position=200):
        super(PositionalEncoding, self).__init__()

        # Not a parameter
        self.register_buffer('pos_table', self._get_sinusoid_encoding_table(n_position, d_hid))

    def _get_sinusoid_encoding_table(self, n_position, d_hid):
        ''' Sinusoid position encoding table '''
        # TODO: make it with torch instead of numpy

        def get_position_angle_vec(position):
            return [position / np.power(10000, 2 * (hid_j // 2) / d_hid) for hid_j in range(d_hid)]

        sinusoid_table = np.array([get_position_angle_vec(pos_i) for pos_i in range(n_position)])
        sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2])  # dim 2i
        sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2])  # dim 2i+1

        return torch.FloatTensor(sinusoid_table).unsqueeze(0)

    def forward(self, x):
        return x + self.pos_table[:, :x.size(1)].clone().detach()
```

나는	0	1	0	1
학생				
이다				

=

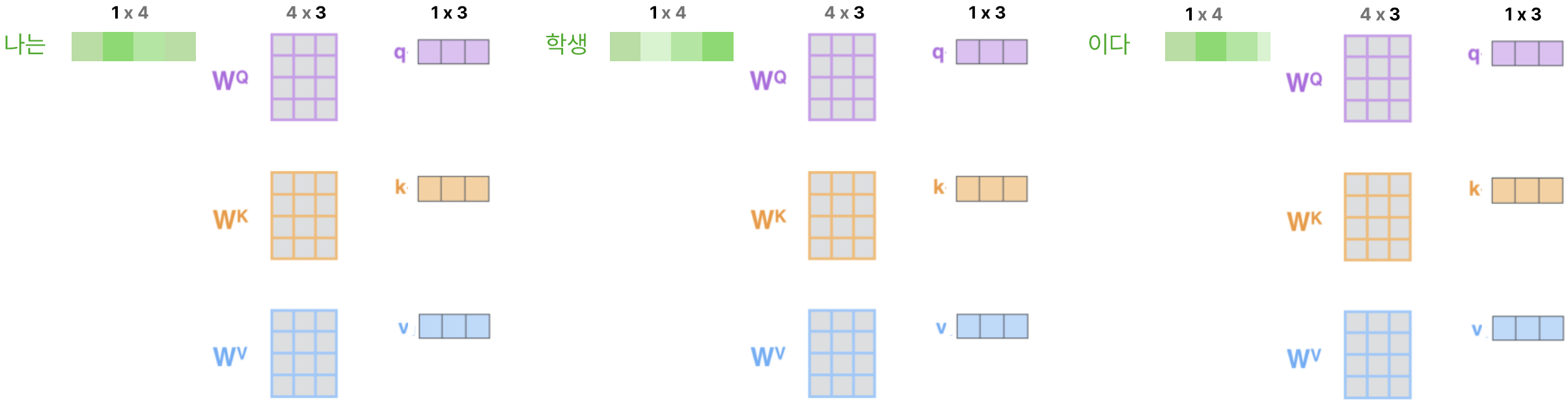
론]

n과 cos을 번갈아서 하는 이유는?

$_{model}$로 나누는 이유는?

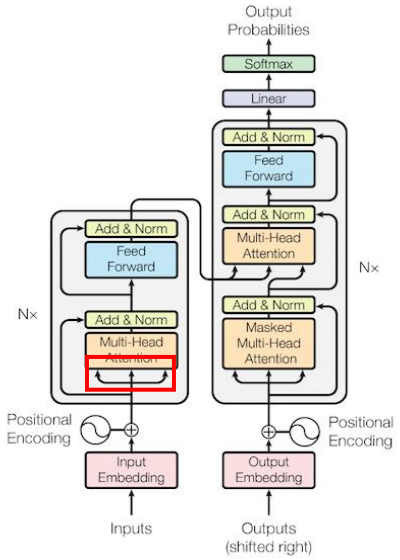
Encoder: Self-Attention

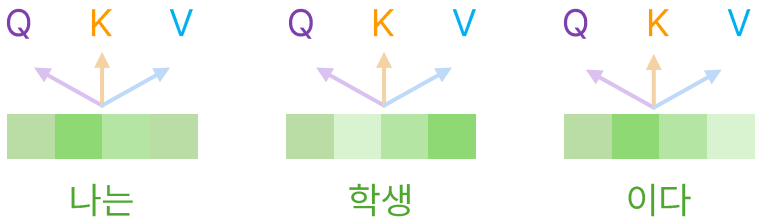
- Step 1: 입력 벡터에 대해 3가지 벡터(Query, Key, Value)를 생성



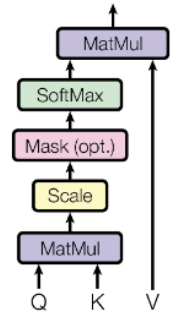
Encoder: Self-Attention

- Step 1: 입력 벡터에 대해 3가지 벡터를 생성
  - Query: 다른 단어들을 고려하여 표현하고자 하는 대상이 되는 현재 단어에 대한 임베딩 벡터
  - Key: Query가 들어왔을 때 다른 단어들과 매칭을 하기 위해 사용되는 레이블로 사용되는 임베딩 벡터
  - Value: Key와 연결된 실제 단어를 나타내는 임베딩 벡터



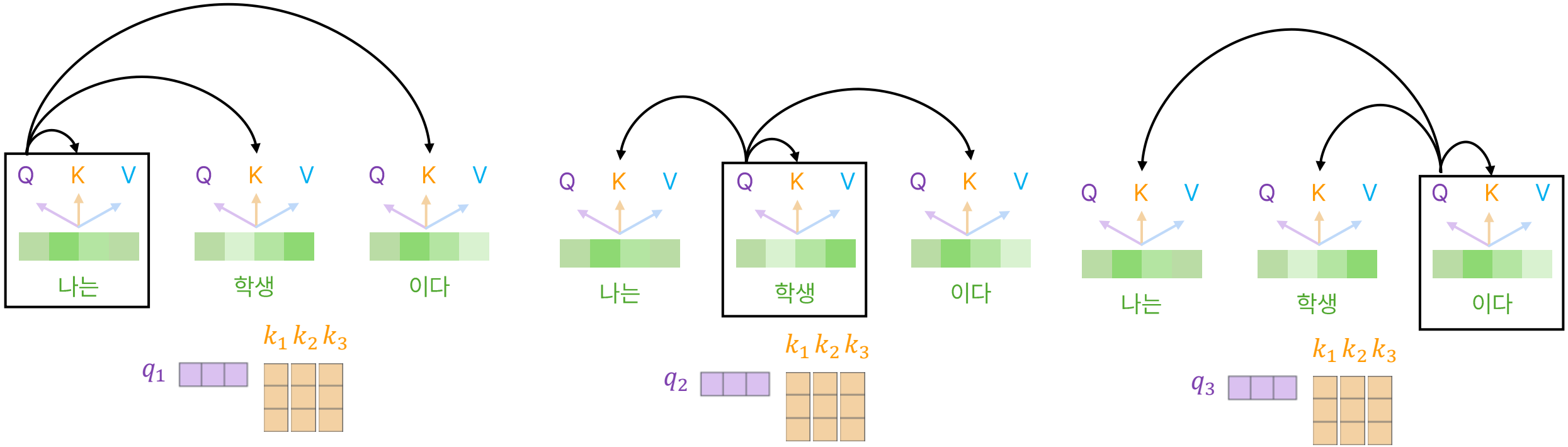


Encoder: Self-Attention

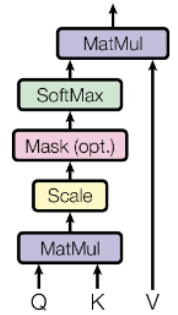


- Step 2: 지금 표현하고자 하는 단어(Q)에 대해 어떤 단어들을 고려해야 하는지(K) 알려주는 스코어 산출

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

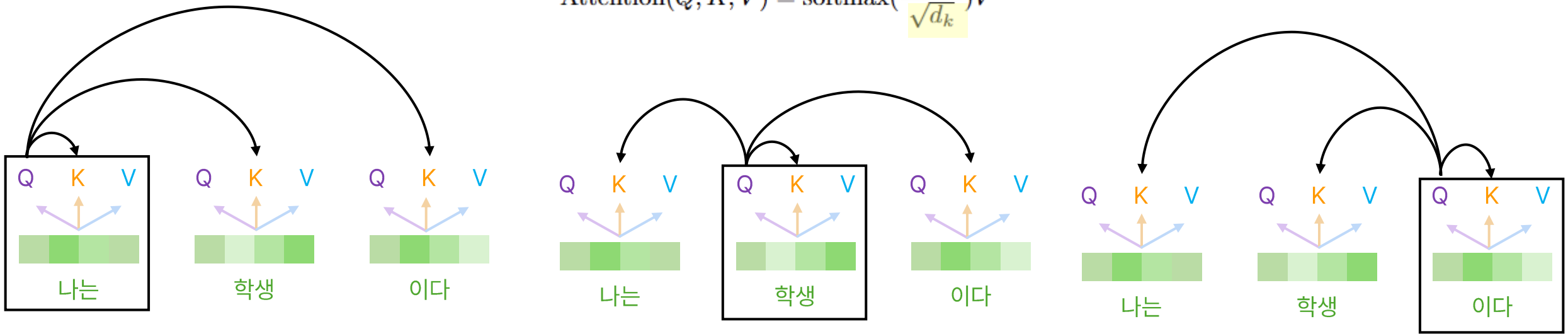


Encoder: Self-Attention

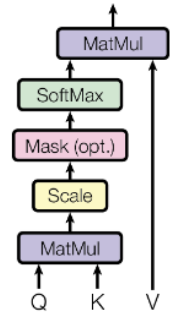


- Step 3: Step 2에서 계산한 스코어를 $\sqrt{d_k}$로 나눠줌 $(d_k = 64, \sqrt{d_k} = 8$ in paper Table 3)
  - Why? Gradient 전파를 안정적으로 수행하기 위함

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

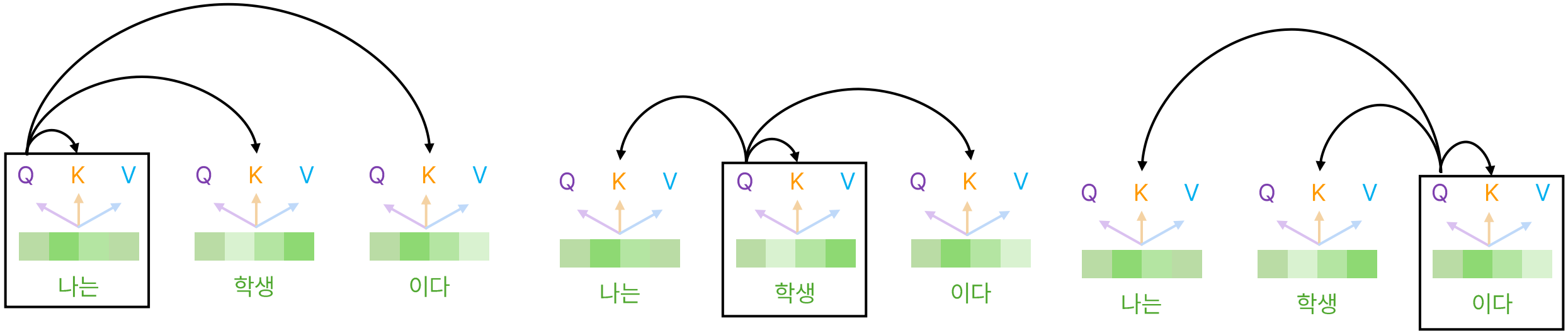


Encoder: Self-Attention

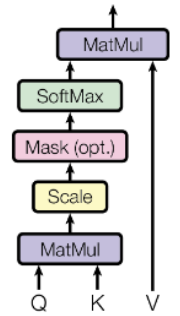


- Step 4: Step 3의 결과물을 이용하여 softmax 함수를 적용하여 해당 단어에 대한 집중도를 산출

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

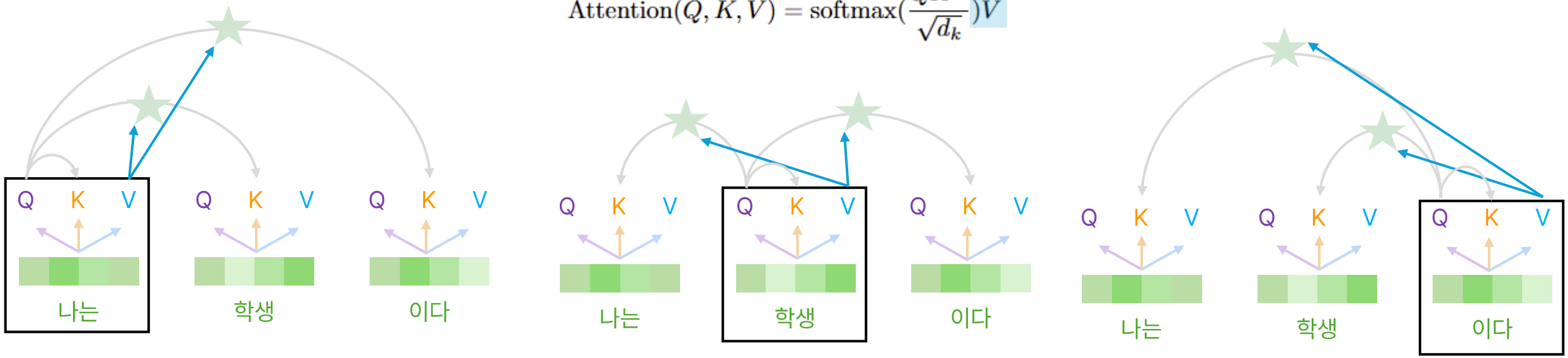


Encoder: Self-Attention

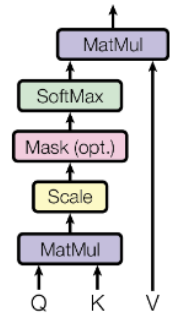


- Step 5: Step 4에서 산출된 확률값과 해당 단어의 **V**의 값을 곱함
  - Why? 집중하고자 하는 토큰의 값을 크게 유지하고, 불필요한 토큰의 값을 매우 작게 하기 위함

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

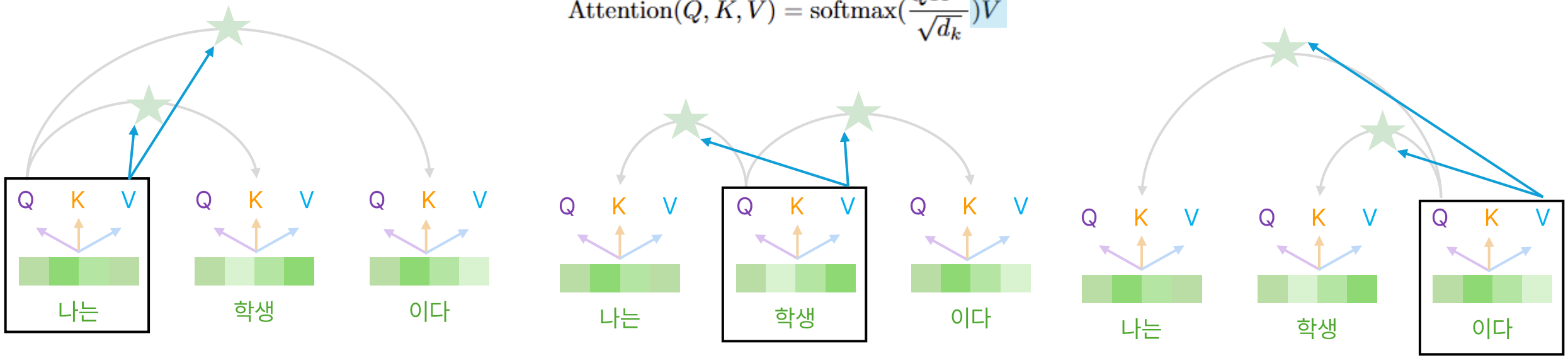


Encoder: Self-Attention



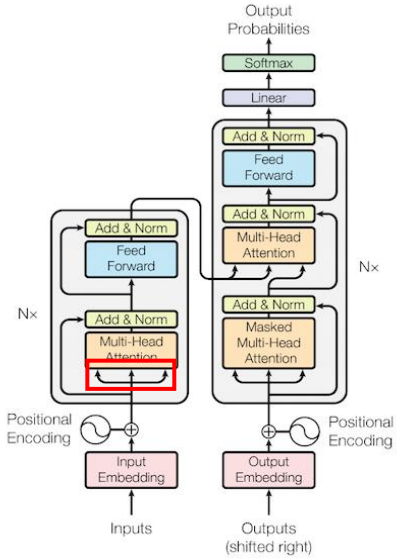
- Step 5: Step 4에서 산출된 확률값과 해당 단어의 **V**의 값을 곱함
  - Why? 집중하고자 하는 토큰의 값을 크게 유지하고, 불필요한 토큰의 값을 매우 작게 하기 위함

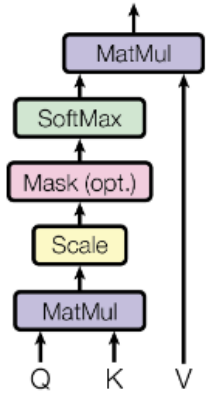
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

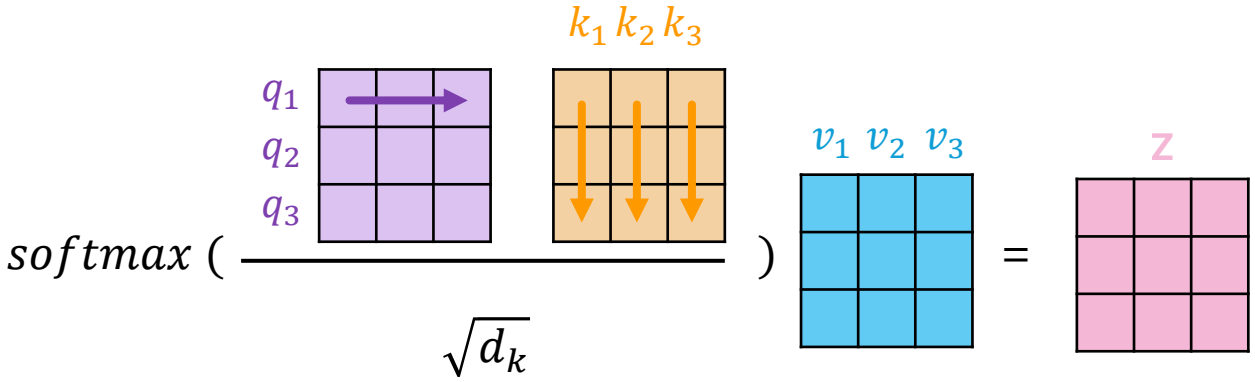


Encoder: Self-Attention

- Step 6: 입력 값을 구성하는 모든 단어 사이 관계를 비교하고, 특징을 추출하여 Z 도출



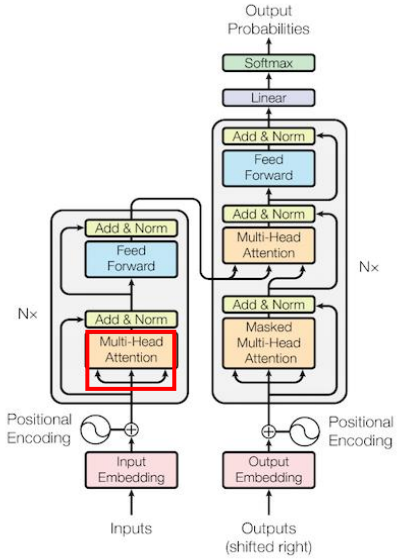


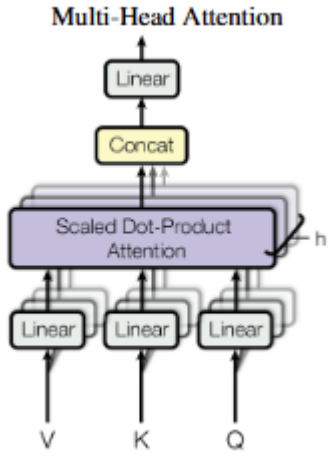


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Encoder: Multi-Head Attention

- 다양한 관계를 학습하기 위해 Muti-Head Attention을 진행

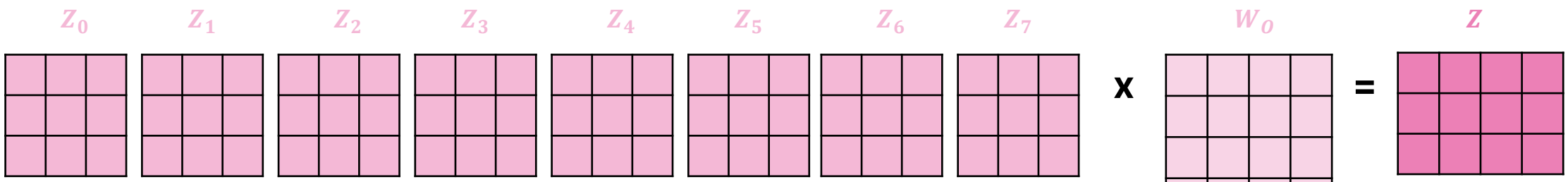




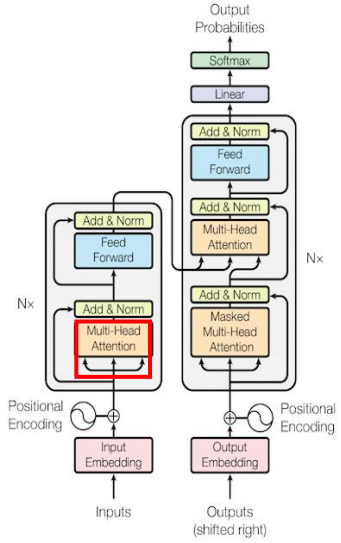
$$softmax\left(\frac{q_1 \; q_2 \; q_3 \quad k_1 \; k_2 \; k_3}{\sqrt{d_k}}\right) v_1 \; v_2 \; v_3 = Z$$

In this work we employ $h = 8$ parallel attention layers, or heads.

Encoder: Multi-Head Attention



3 x 24 (3x8)

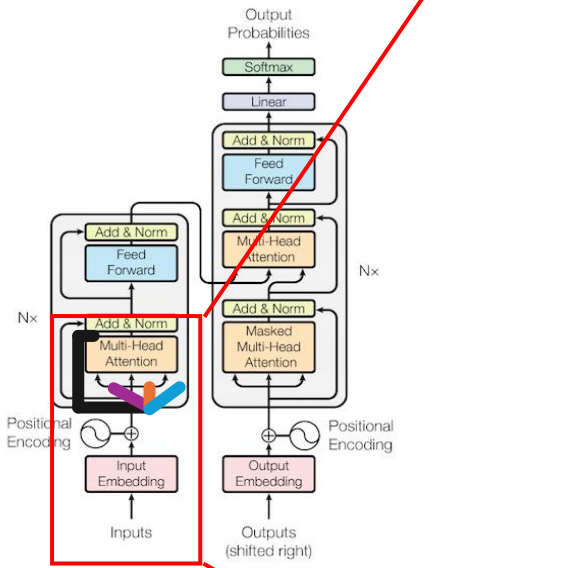


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

...

24 x 4

Encoder: Add & Normalize



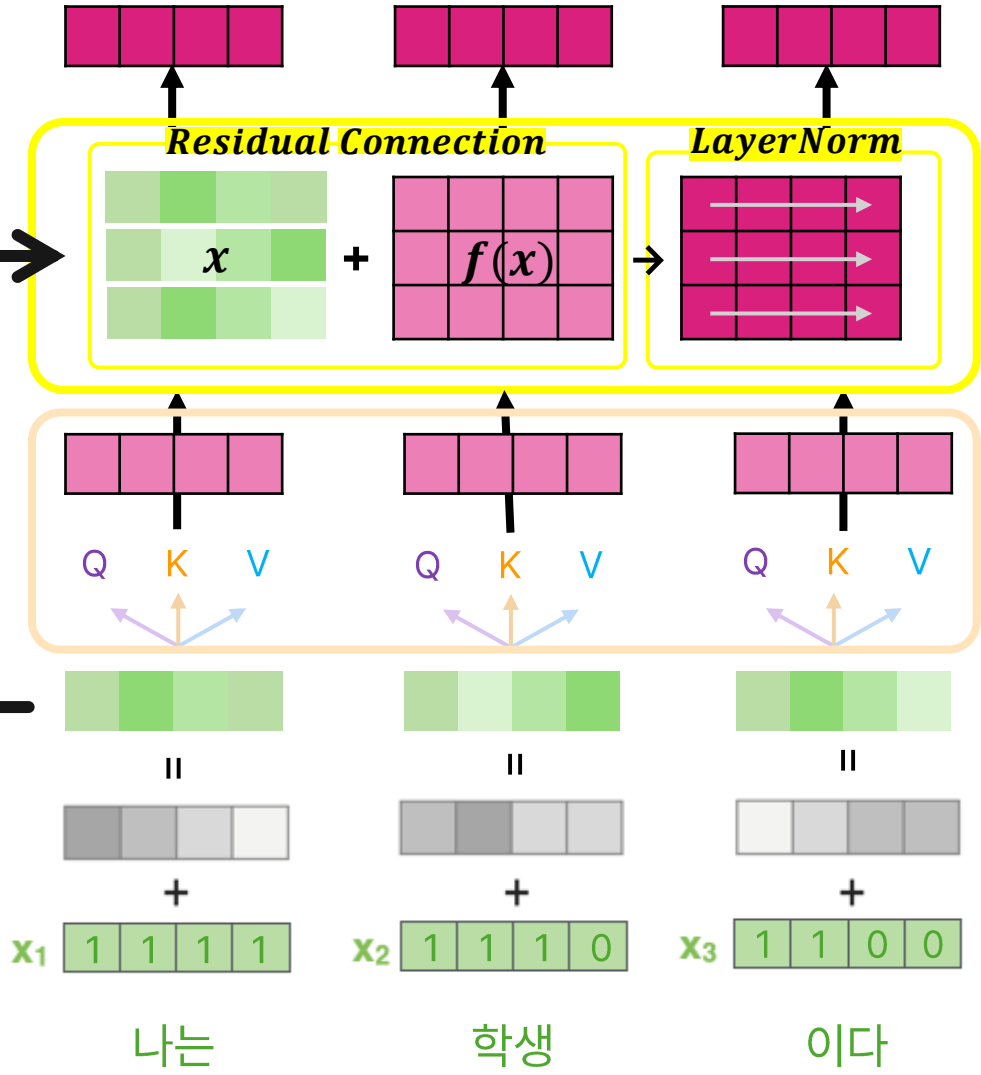
Add & Normalize

Multi-Head Attention

Input

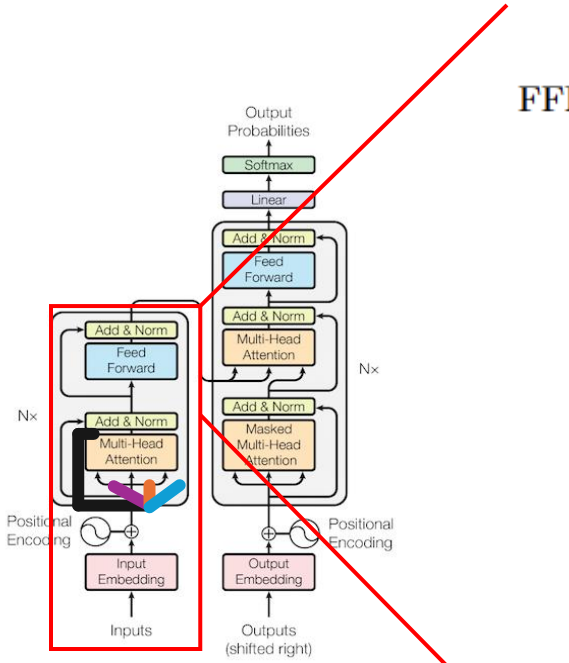
Postional Encoding

Embedding

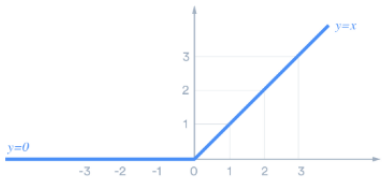


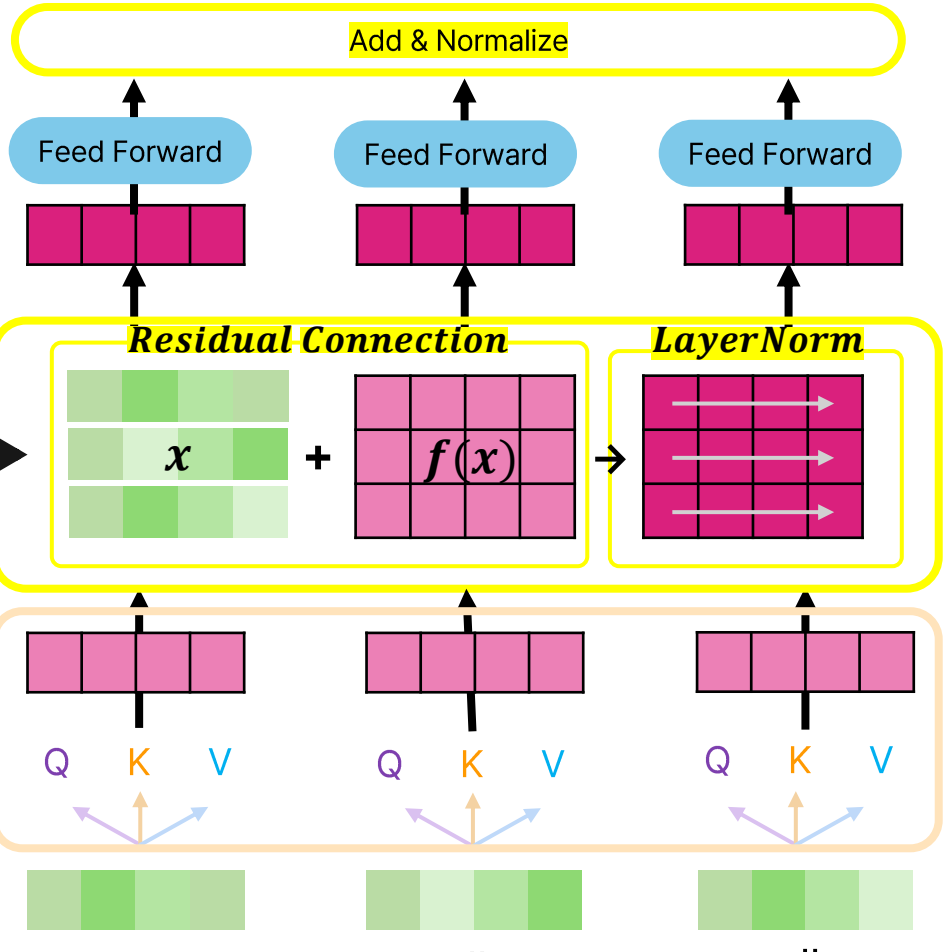
Encoder: Feed Forward Network

- 각 단어에 대응되는 값들에 개별적으로 FFN를 적용하여 비선형성 부여



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



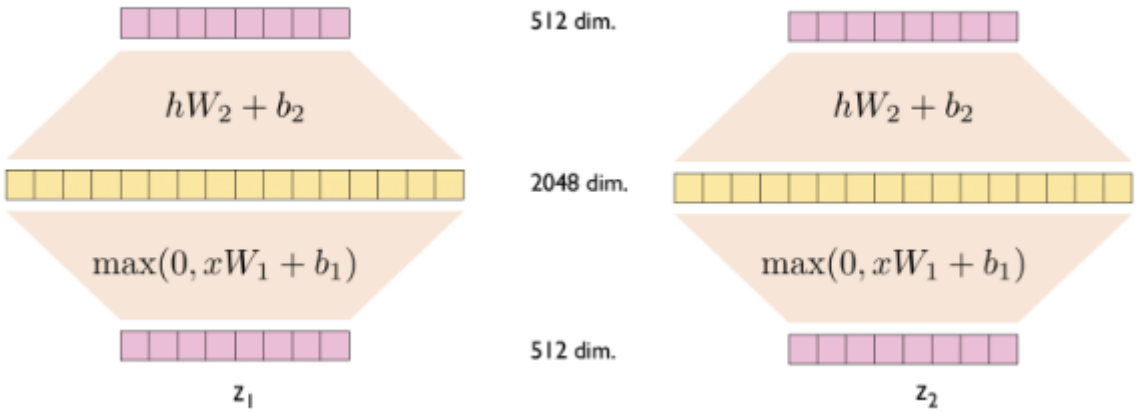


Add & Normalize

Multi-Head Attention

Input

Encoder: Feed Forward Network



In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

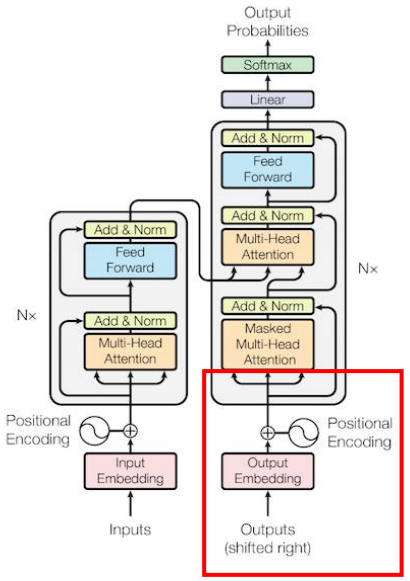
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

```python
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

    def forward(self, src_seq, src_mask, return_attns=False):

        enc_slf_attn_list = []

        # -- Forward
        enc_output = self.src_word_emb(src_seq)
        if self.scale_emb:
            enc_output *= self.d_model ** 0.5
        enc_output = self.dropout(self.position_enc(enc_output))
        enc_output = self.layer_norm(enc_output)

        for enc_layer in self.layer_stack:
            enc_output, enc_slf_attn = enc_layer(enc_output, slf_attn_mask=src_mask)
            enc_slf_attn_list += [enc_slf_attn] if return_attns else []

        if return_attns:
            return enc_output, enc_slf_attn_list
        return enc_output,
```

```python
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
```

Decoder

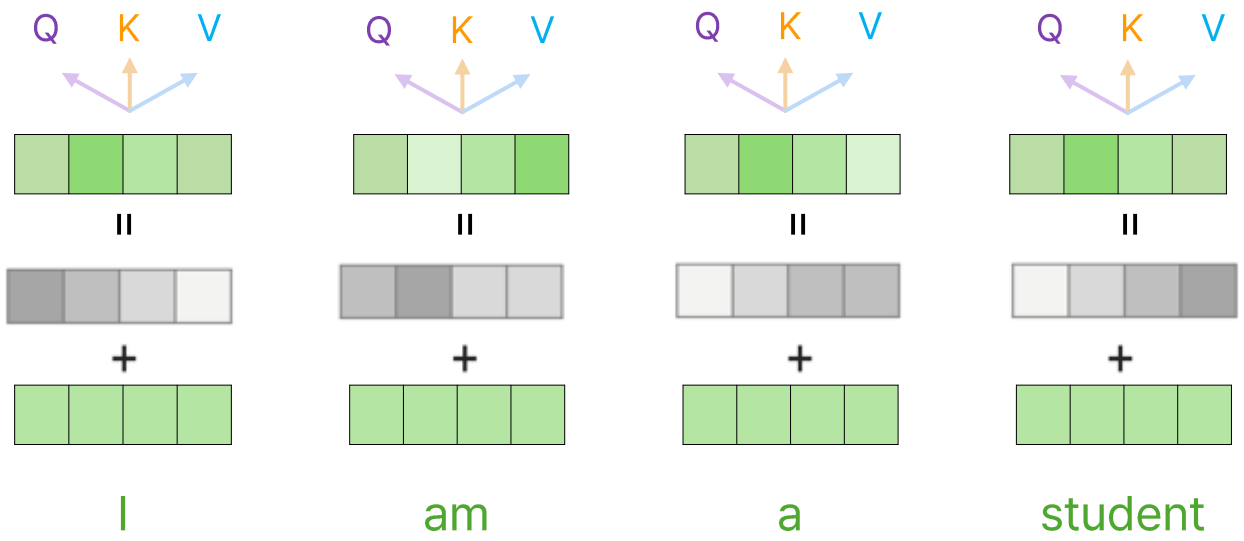


Input

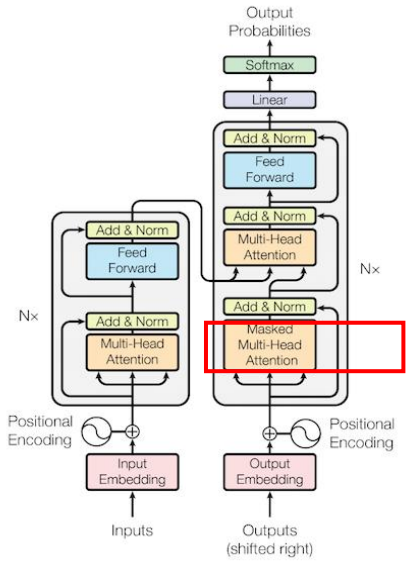
Postional Encoding

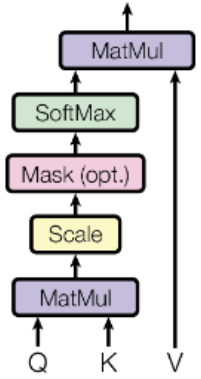
Embedding

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



Decoder: Masked Multi-Head Attention





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

	I	am	a	student
I	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$
am	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$
a	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$
student	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$

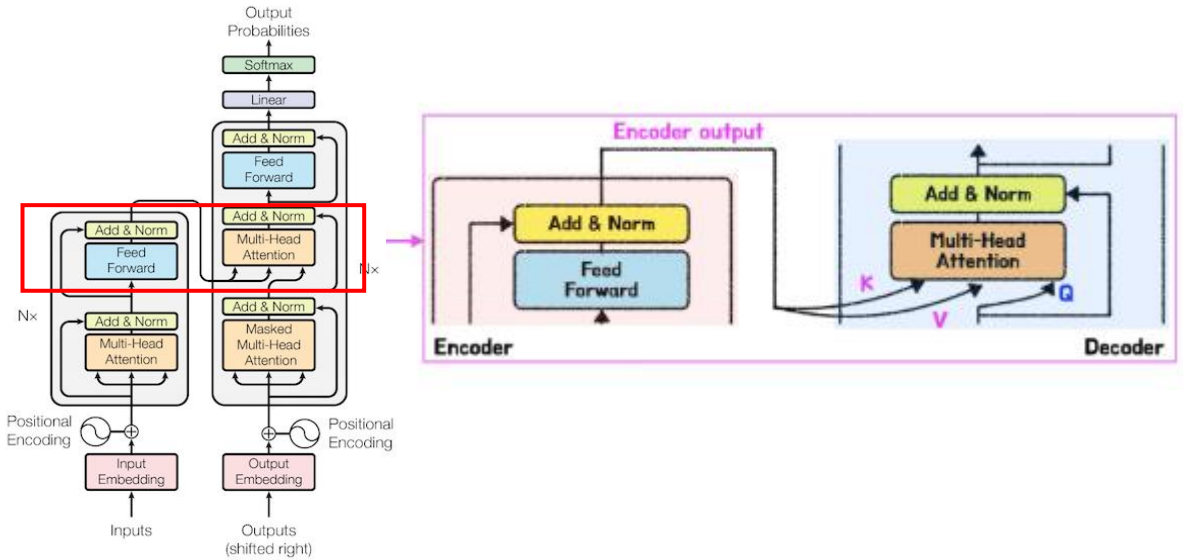
	I	am	a	student
I	$\frac{QK^T}{\sqrt{d_k}}$	$-\infty$	$-\infty$	$-\infty$
am	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$-\infty$	$-\infty$
a	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$-\infty$
student	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$

	I	am	a	student
I	$\frac{QK^T}{\sqrt{d_k}}$	0	0	0
am	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	0	0
a	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	0
student	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$	$\frac{QK^T}{\sqrt{d_k}}$

순차적으로 수행할 필요 없이 한번에 수행가능

Decoder: Multi-Head Attention

- 디코더는 인코더의 출력에서 중요한 정보를 찾아 다음 출력 단어를 생성해야 함
- Q는 현재 디코더 상태에서 생성되며, K와 V는 인코더에서 가져와야 입력의 정보를 참고할 수 있음
- 즉, 디코더는 인코더에서 생성된 K, V를 기반으로 다음 출력을 예측



```python
class Decoder(nn.Module):
    ''' A decoder model with self attention mechanism. '''

    def __init__(
            self, n_trg_vocab, d_word_vec, n_layers, n_head, d_k, d_v,
            d_model, d_inner, pad_idx, n_position=200, dropout=0.1, scale_emb=False):

        super().__init__()

        self.trg_word_emb = nn.Embedding(n_trg_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([
            DecoderLayer(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

    def forward(self, trg_seq, trg_mask, enc_output, src_mask, return_attns=False):

        dec_slf_attn_list, dec_enc_attn_list = [], []

        # -- Forward
        dec_output = self.trg_word_emb(trg_seq)
        if self.scale_emb:
            dec_output *= self.d_model ** 0.5
        dec_output = self.dropout(self.position_enc(dec_output))
        dec_output = self.layer_norm(dec_output)

        for dec_layer in self.layer_stack:
            dec_output, dec_slf_attn, dec_enc_attn = dec_layer(
                dec_output, enc_output, slf_attn_mask=trg_mask, dec_enc_attn_mask=src_mask)
            dec_slf_attn_list += [dec_slf_attn] if return_attns else []
            dec_enc_attn_list += [dec_enc_attn] if return_attns else []

        if return_attns:
            return dec_output, dec_slf_attn_list, dec_enc_attn_list
        return dec_output,
```

```python
class DecoderLayer(nn.Module):
    ''' Compose with three layers '''

    def __init__(self, d_model, d_inner, n_head, d_k, d_v, dropout=0.1):
        super(DecoderLayer, self).__init__()
        self.slf_attn = MultiHeadAttention(n_head, d_model, d_k, d_v, dropout=dropout)
        self.enc_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, dec_input, enc_output,
            slf_attn_mask=None, dec_enc_attn_mask=None):
        dec_output, dec_slf_attn = self.slf_attn(
            dec_input, dec_input, dec_input, mask=slf_attn_mask)
        dec_output, dec_enc_attn = self.enc_attn(
            dec_output, enc_output, enc_output, mask=dec_enc_attn_mask)
        dec_output = self.pos_ffn(dec_output)
        return dec_output, dec_slf_attn, dec_enc_attn
```

Why Self-Attention

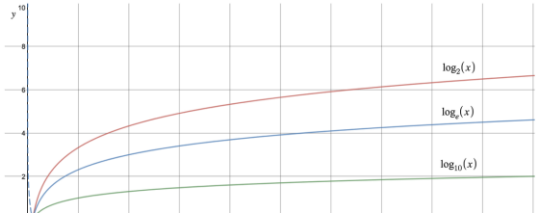


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- RNN → 병렬화 불가 → 학습 속도 느림
- CNN → 국소적 관계 학습 강함 → 멀리 떨어진 토큰 간 관계 학습 어려움
- Self-Attention → 병렬화 가능 + 전역 의존성 학습 가능

(1) Machine Translation

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Transformer (Base) 모델도 기존 모든 모델 및 앙상블을 능가,

Transformer (Big) 모델이 이전 최고 성능 모델(앙상블 포함) 대비 2.0 BLEU 이상 향상

(1) Machine Translation

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Transformer (Base) 모델도 기존 모든 모델 및 앙상블을 능가,

Transformer (Big) 모델이 이전 최고 성능 모델(앙상블 포함) 대비 2.0 BLEU 이상 향상

(2) Model Variations

- 실험 목표: Transformer의 각 구성 요소가 성능에 미치는 영향 분석

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

(A) Head가 적당히 많은 경우 성능 향상을 기대할 수 있으나,
지나치게 많아지면 오히려 성능이 하락

(B) Key의 크기를 줄이면 모델의 성능이 하락할 수 있음

(C)&(D) 모델의 사이즈가 클수록 성능이 향상되는 경향이 있으며,
Drop-out과 Label Smoothing이 성능 개선에 효과적임

(E) 사인파(Sinusoidal) → 학습된(Learned): 성능 X
즉, 사인파(Sinusoidal) 방식 그대로 사용해도 충분하다는 의미

+ English Constituency parsing

- **실험 목표**: Transformer가 기계번역 이외에도 다른 작업에서 일반화될 수 있는지 확인하기 위함
- 영어 구문 분석이란?

The cat sat on the mat.

명사구 동사구 전치사구

+ English Constituency parsing

- **데이터셋**

	WSJ	semi-supervised
데이터의 문장 수	WSJ에 나온 기사 문장 40K (= 4만 개)	WSJ에 나온 기사 문장 40K (= 4만 개) + BerkleyParser 코퍼스 17M (1700만)
Vocabulary size	16K 토큰 (= 16,000 토큰)	32K 토큰 (= 32,000 토큰)

- **모델 구조 및 하이퍼파라미터 설정**
  - 4층(4- layer) Transformer 사용, d model = 1024
  - Dropout, Attention, Residual, Learning Rate, Beam Size 등 최소한의 실험만 수행
  - 대부분의 설정은 **영어-독일어 번역 모델과 동일**
  - 추론 시, 최대 출력 길이 = 입력 길이 + 300
  - Beam Size = 21, α = 0.3

+ English Constituency parsing

- 실험 결과

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)

Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

최고의 성능을 제공하지는 못하지만, 의외로 꽤 좋은 성능을 제공한다는 것을 확인할 수 있음

Conclusion

- **연구 개요**
  - Transformer는 Attention을 전적으로 기반한 최초의 모델
  - 기존의 RNN 기반 인코더-디코더 구조 X, Multi-Head Self Attention 제안
  - 기계 번역에서 기존 방법보다 훈련속도가 크게 향상됨

- **연구 성과**
  - WMT 2014 영어-독일어, 영어-프랑스어 번역 성능 각각 모두 SOTA를 달성
  - 특히, 영어-독일어 번역에서 기존 앙상블 모델보다 우수한 성과를 거둠

Conclusion

- **향후 연구 방향**
  - 다양한 입력/출력 모달리티 적용하고자 함.

    이미지, 오디오 및 비디오와 같은 큰 입력 및 출력을 효율적으로 처리하기 위해 local restricted attention 기법
  - 생성 과정을 덜 순차적으로 만드는 것

local restricted attention기법은 이미지, 오디오, 비디오에 따라 제한한다는 건가?

→ X, 효율적인 계산을 위해 모든 토큰을 참조하는 대신,

(local attention) 이미지는 인접한 픽셀만, 오디오는 특정 시간 구간만 참고하거나,

(restricted attention) 비디오의 경우 중요한 프레임이나 특정 시간 구간만 어텐션 계산

Conclusion

- **향후 연구 방향**
  - 다양한 입력/출력 모달리티 적용하고자 함.

    이미지, 오디오 및 비디오와 같은 큰 입력 및 출력을 효율적으로 처리하기 위해 local restricted attention 기법
  - 생성 과정을 덜 순차적으로 만드는 것

생성 과정을 덜 순차적으로 만드는 이유는 무엇인가?

→ Transformer의 디코더는 이전까지 생성된 단어를 보고 다음 단어를 하나씩 예측하는데,

문장을 한꺼번에 만들지 않고 토큰 하나씩 생성하므로 속도 느리지는 원인이 되므로 병렬적으로 만들겠다는 의미