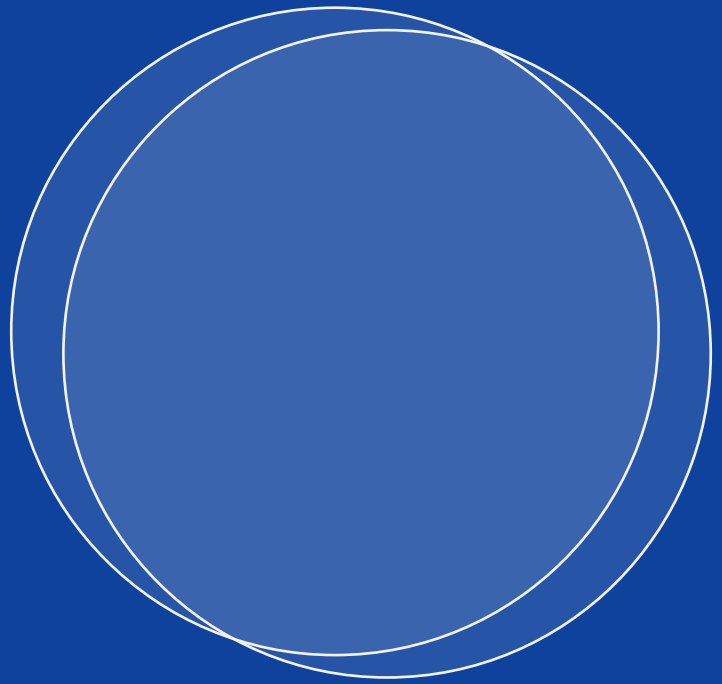
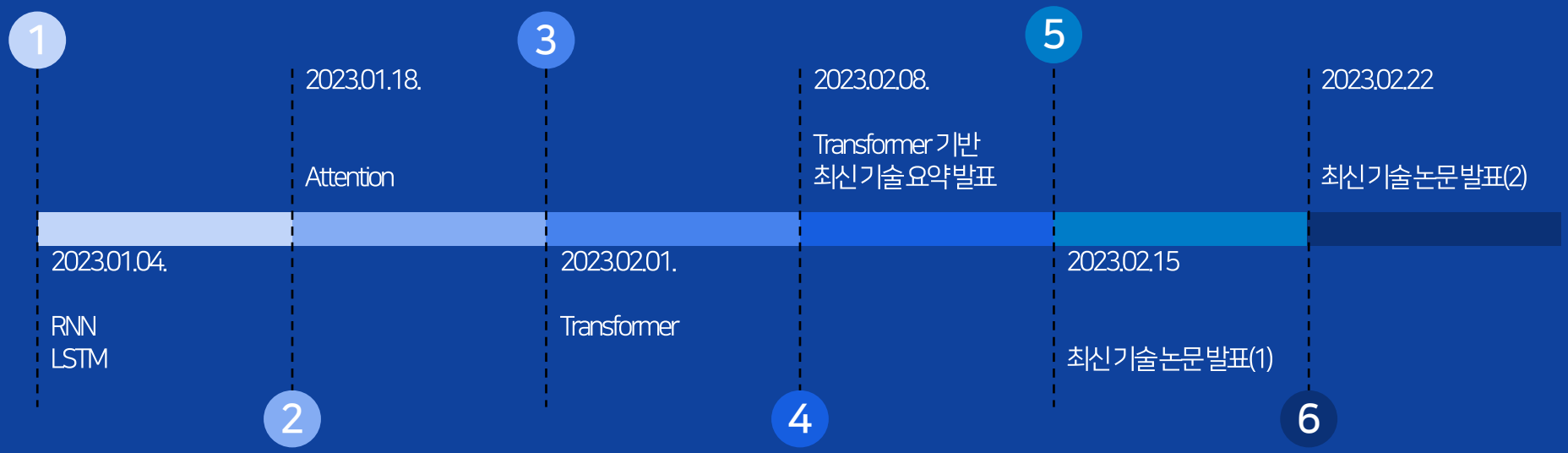


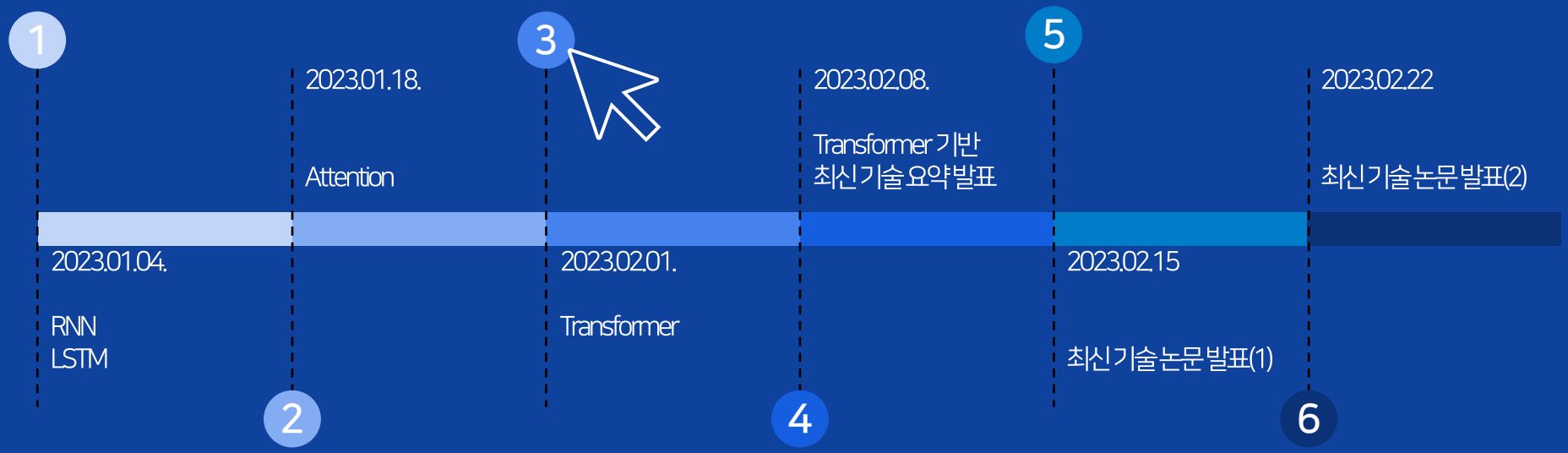
Transformer

202132033 염지현

>> Transformer 주차별 계획



>> Transformer 주차별 계획



>> What we will study

01

Transformer의 등장

- Seq2seq 모델의 한계점
- Transformer 개념 소개

02

Transformer 동작원리

- Positional encoding
- Encoder
- Decoder

03

코드로 보는 Transformer 동작원리

- Positional encoding
- Encoder
- Decoder

1

Transformer의 등장

Part 1 >> 기존 연구의 한계점

1. RNN(seq2seq 모델)

1. 고정된 크기의 context vector에 모든 정보를 압축하기 때문에 정보의 손실 발생
2. Gradient vanishing/exploding 문제 발생

*seq2seq: sequence-to-sequence로 RNN, LSTM과 같이 입력된 시퀀스로부터 다른 도메인의 시퀀스를 출력하는 모델

2. CNN

1. Kernel 간의 정보 공유 불가

Part 1 >> Transformer 등장

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 dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

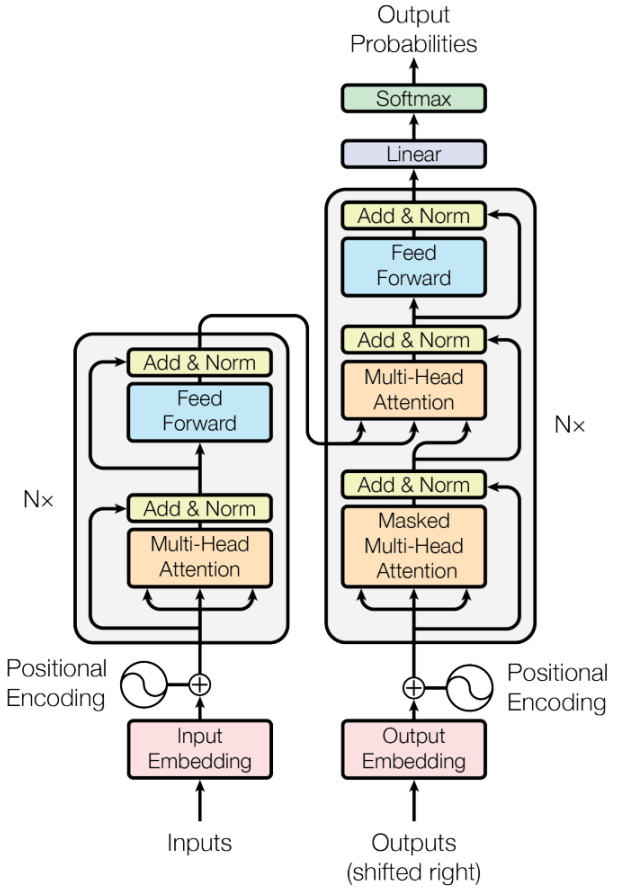


Figure 1: The Transformer - model architecture.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Part 1 >> Transformer 등장

1) 요약

1) "Attention is all you need(NIPS, 2017)" 논문에서 제안한 모델
2) Non-recurrent sequence to sequence encoder-decoder model 생성이 목적
3) 기존 seq2seq encoder-decoder 구조를 따르면서 attention mechanism에만 의존하여 번역 작업 수행

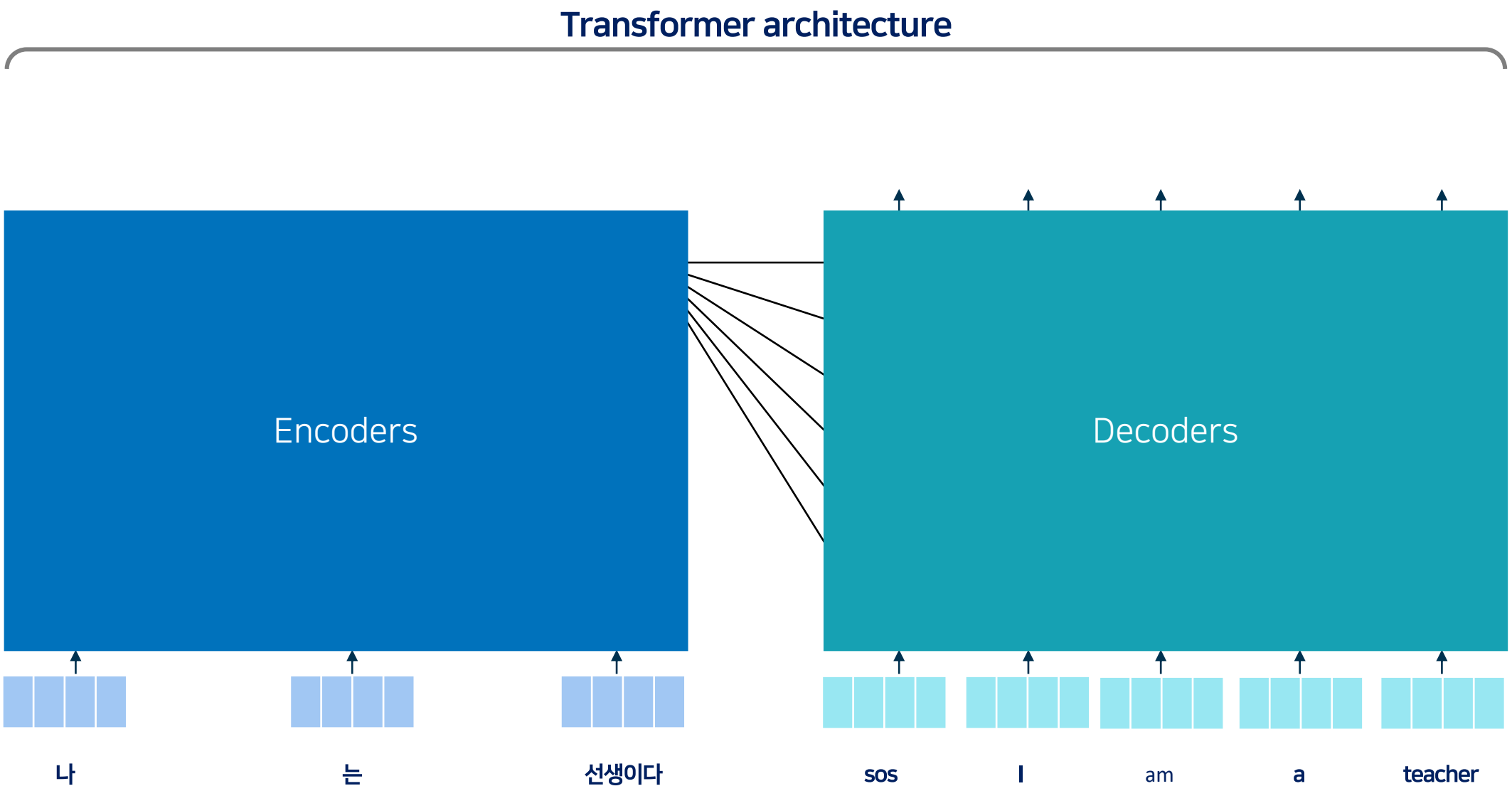
2) 특징

1) 기존 seq2seq 모델에서 고정된 크기의 context vector를 사용함에 따른 정보 손실 해결
2) Attention mechanism을 사용하여 입력과 출력 사이 전역적인 의존성을 끌어냄
3) RNN을 사용하지 않으면서 encoder, decoder 다수 사용
4) 순차적 입력이 아닌 한 번의 입력으로 계산 복잡도 낮춤

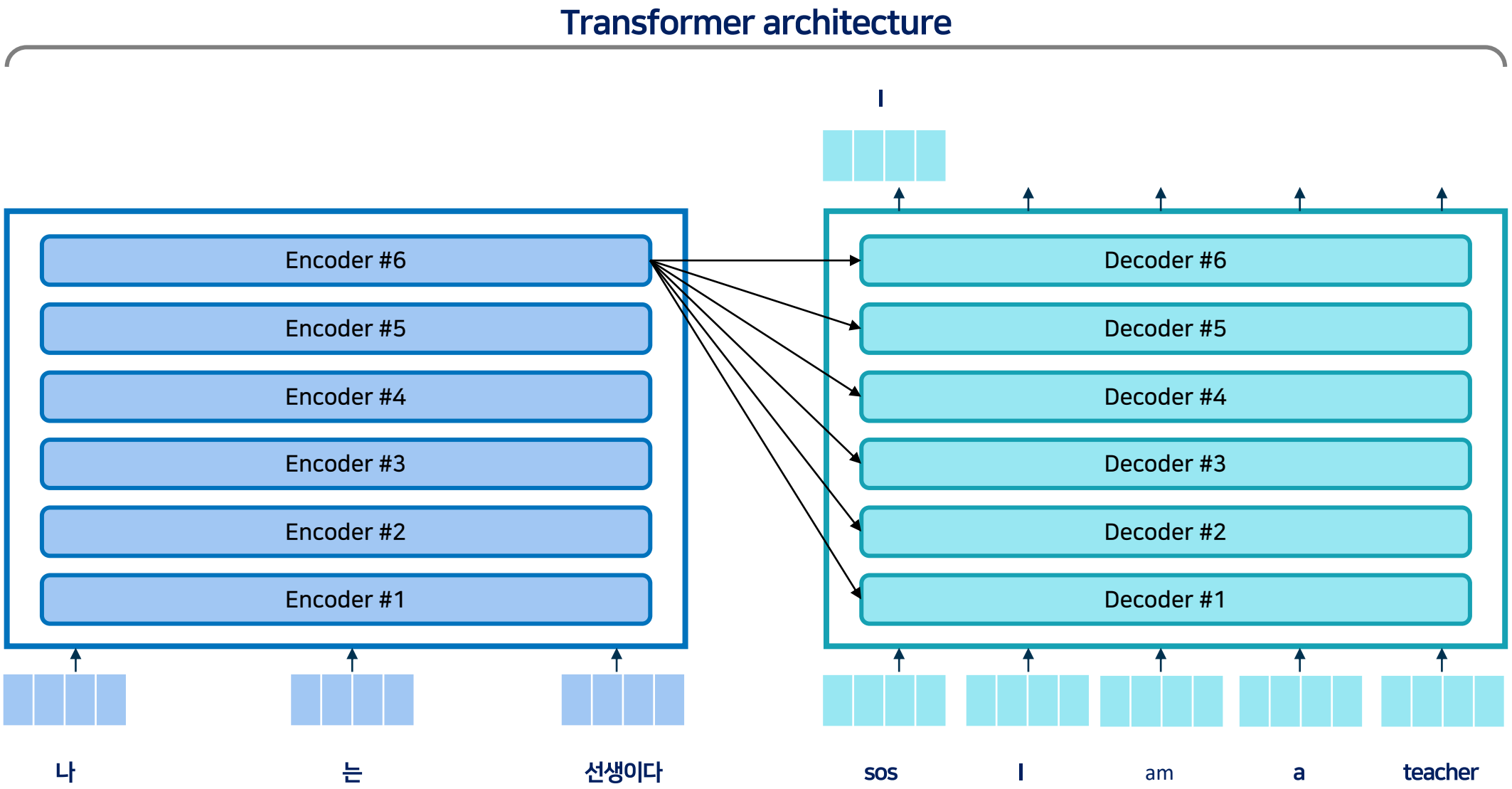
2

Transformer 동작 원리

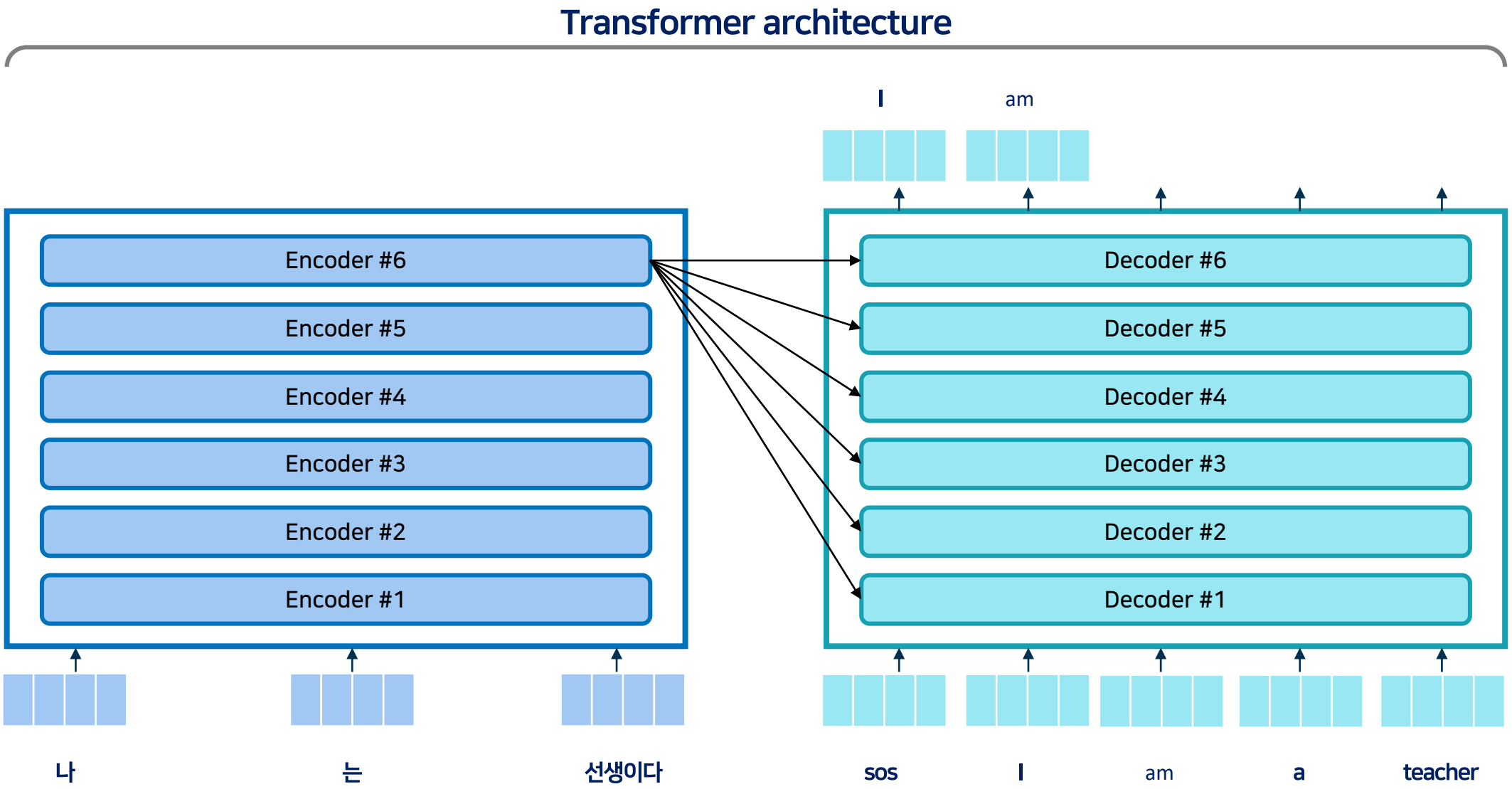
Part 2 >> Transformer 동작 원리(0) 흐름 파악하기



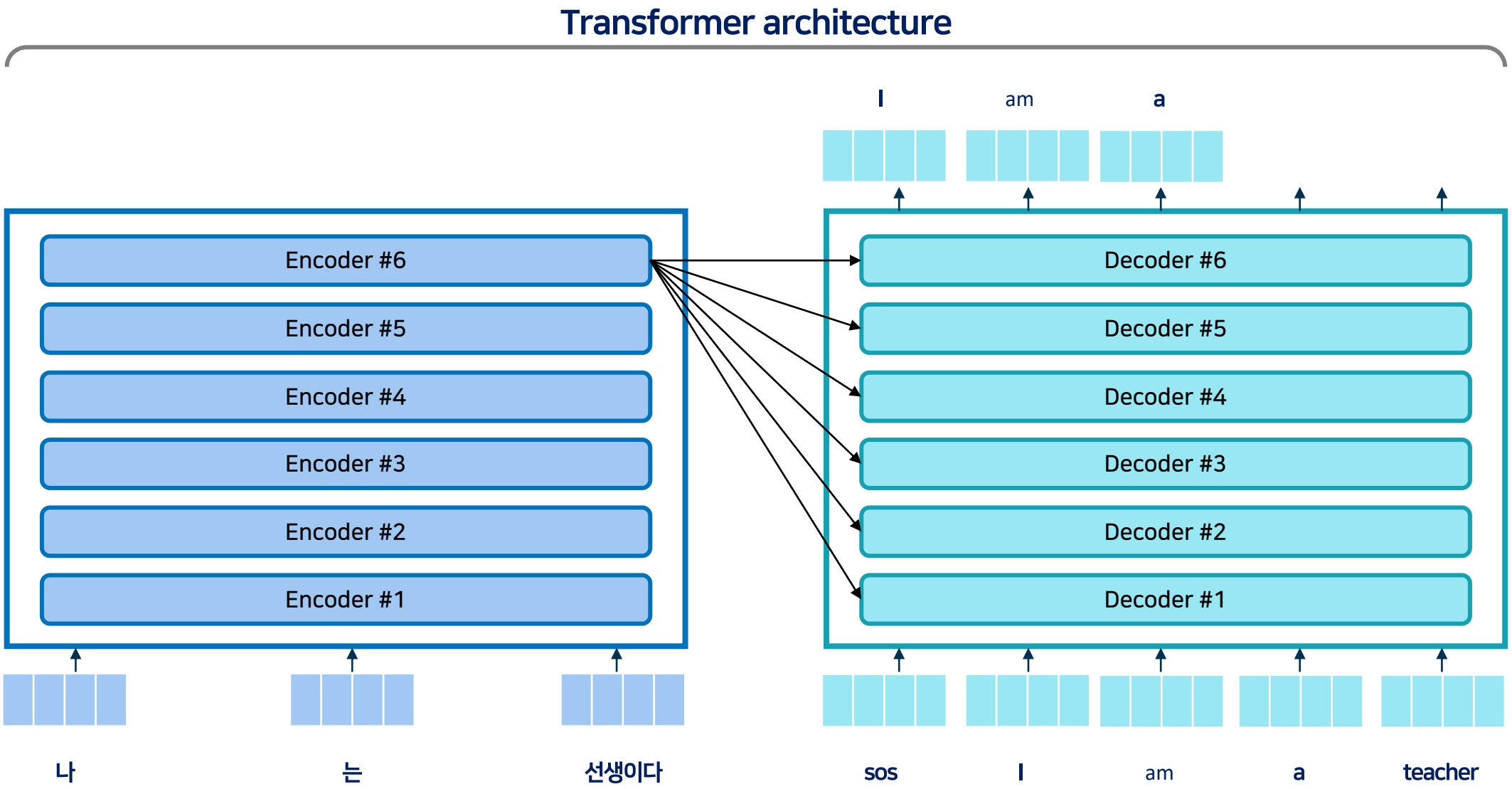
Part 2 >> Transformer 동작 원리(0) 흐름 파악하기



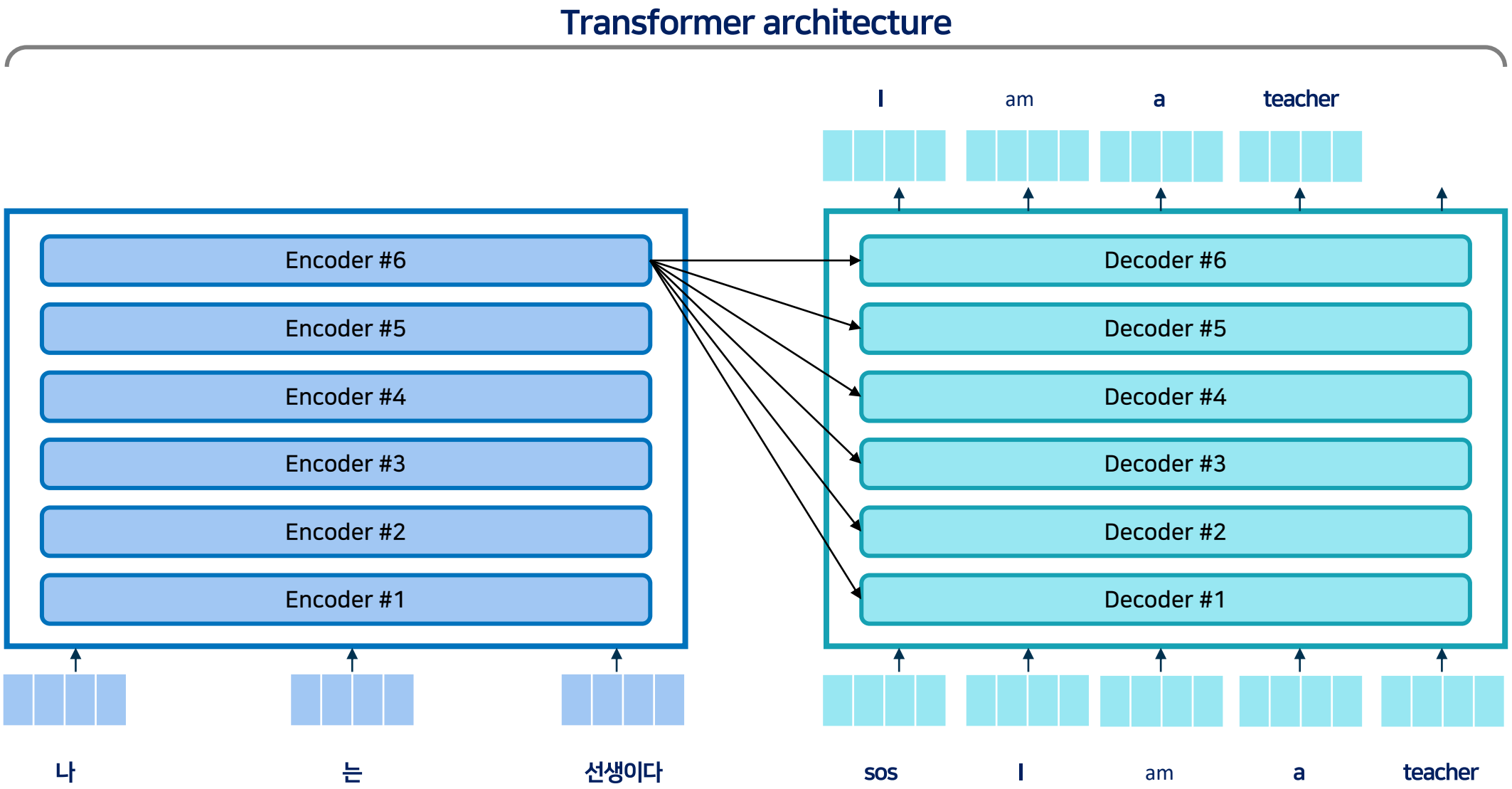
Part 2 >> Transformer 동작 원리(0) 흐름 파악하기



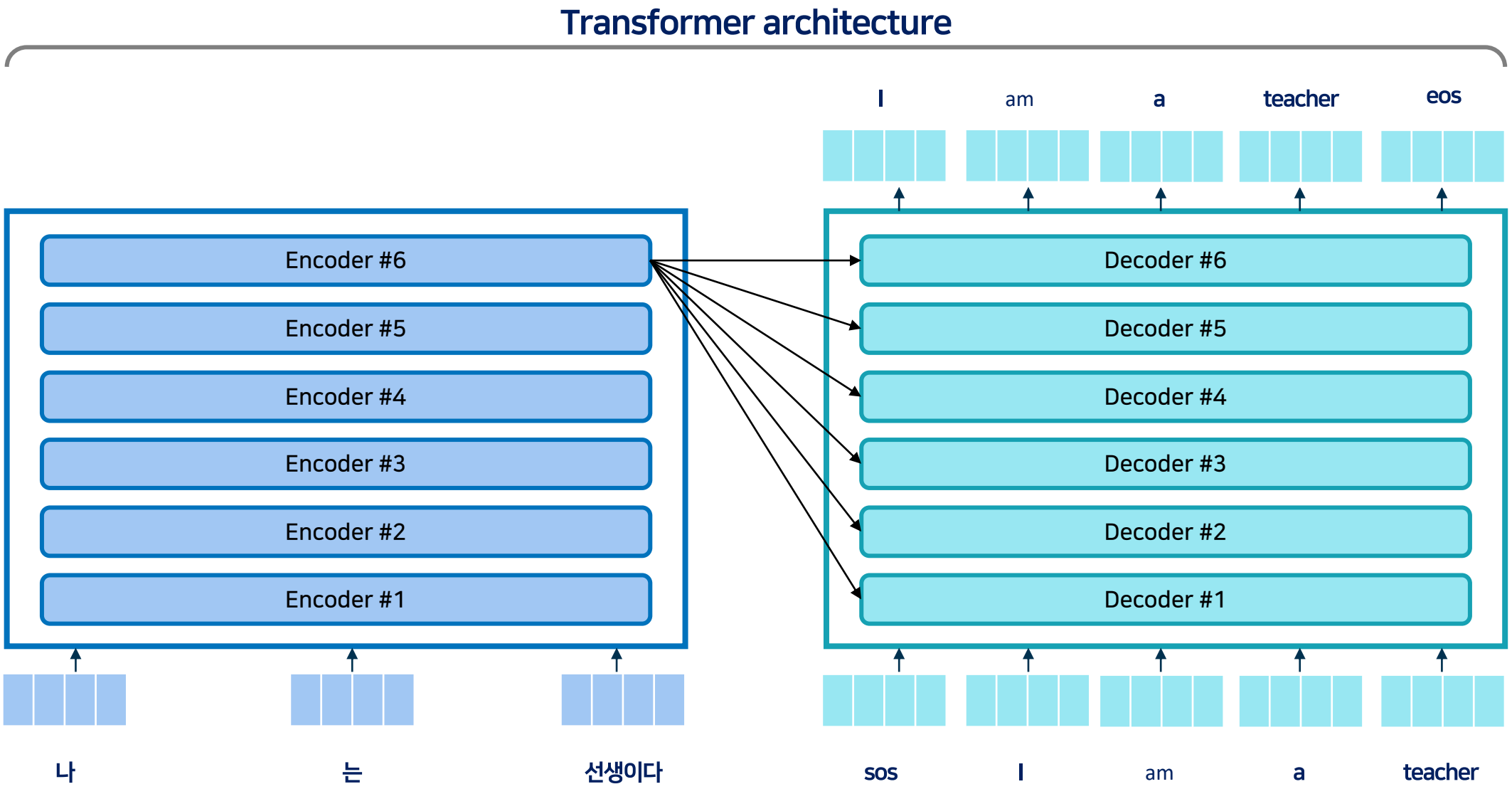
Part 2 >> Transformer 동작 원리(0) 흐름 파악하기



Part 2 >> Transformer 동작 원리(0) 흐름 파악하기



Part 2 >> Transformer 동작 원리(0) 흐름 파악하기

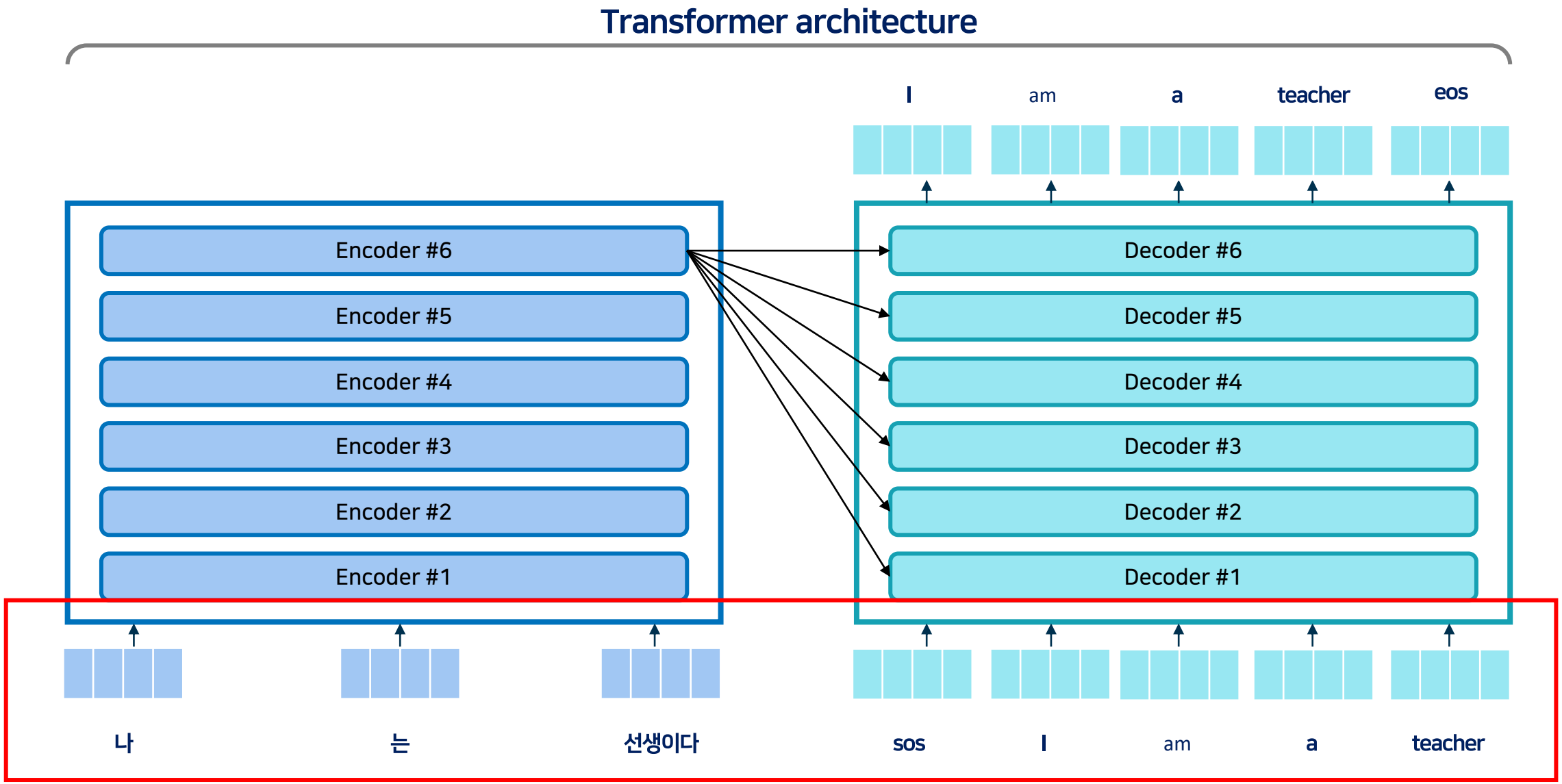


Part 2 >> Transformer 동작 원리(0) 흐름 파악하기

1) 요약

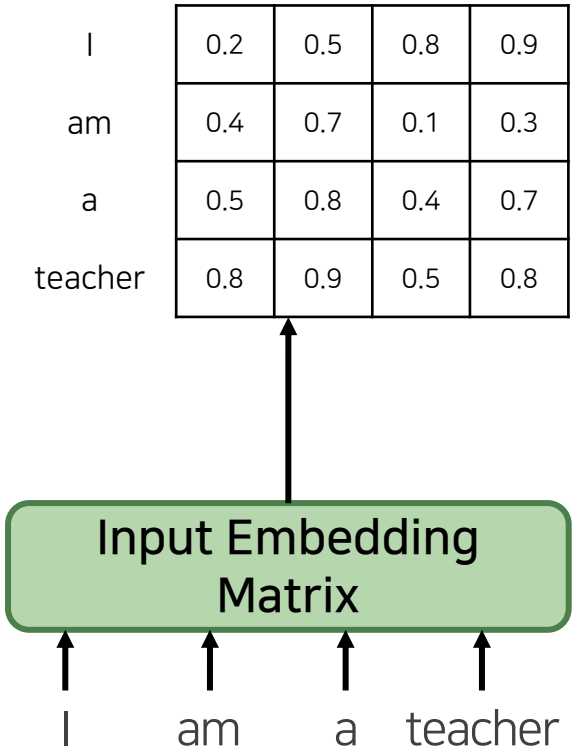
1) Encoders, Decoders로 여러 개의 encoder와 decoder를 사용
    1) 논문 기준 encoder 6개, decoder 6개 사용
2) Seq2seq 모델과 다르게 한 번의 병렬 입력으로 처리
3) Encoders의 마지막 layer에서 얻은 출력은 Decoders의 모든 입력으로 사용
4) Decoder에서 마지막 출력은 병렬적으로 한 번에 출력이 아닌 순차 출력
    1) eos(end of sentence) 시그널을 만날 때까지 출력

Part 2 >> Transformer 동작 원리(1) Positional encoding



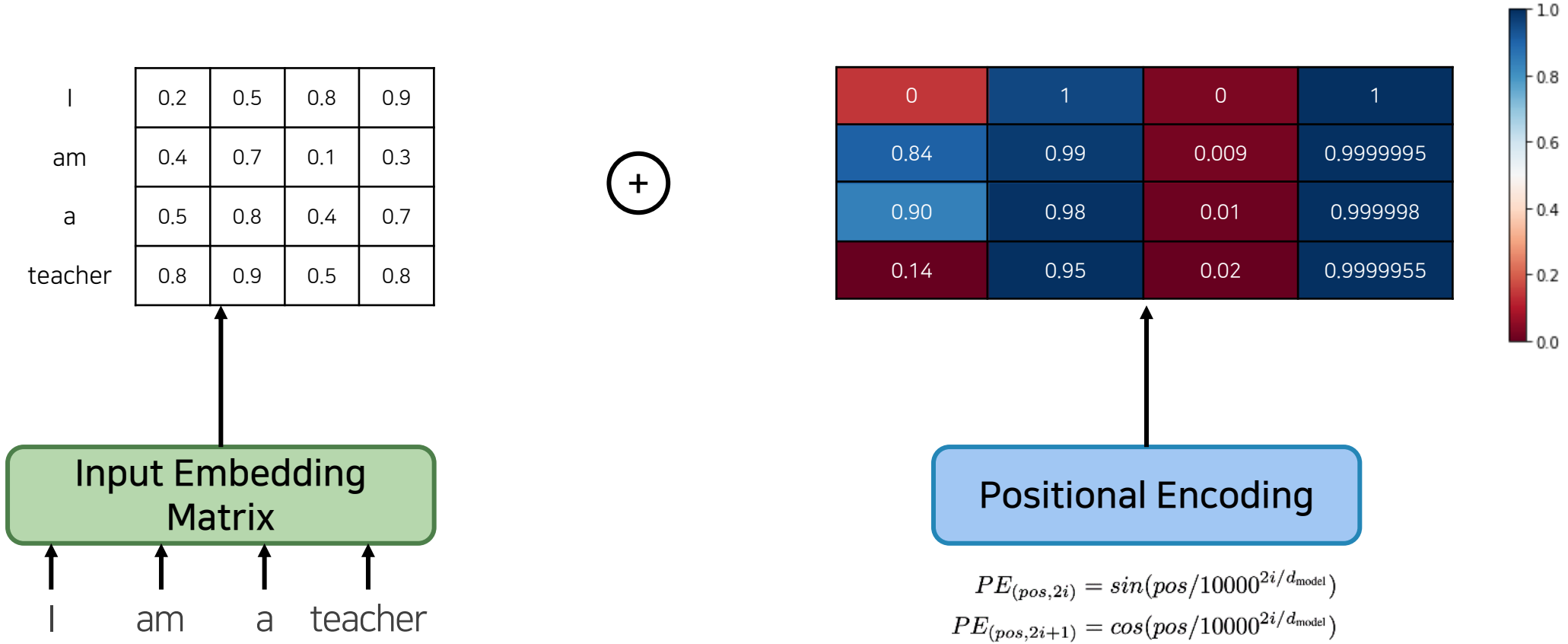
Part 2 >> Transformer 동작 원리(1) Positional encoding

Positional encoding 사용 목적: Transformer에게 입력단어의 위치 정보를 알려주기 위해 사용



Part 2 >> Transformer 동작 원리(1) Positional encoding

Positional encoding 사용 목적: Transformer에게 입력단어의 위치 정보를 알려주기 위해 사용



Part 2 >> Transformer 동작 원리(1) Positional encoding

Positional encoding 사용 목적: Transformer에게 입력단어의 위치 정보를 알려주기 위해 사용

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

pos: 몇 번째 단어인지(행)
i: 임베딩 벡터 내에서 몇 차원인지
d_model: 임베딩 벡터 차원수

i →

pos ↓

	0=i	1=i+1	2=i	3=i+1
0	Sin(0/1)	Cos(0/10)	Sin(0/100)	Cos(0/1000)
1	Sin(1/1)	Cos(1/10)	Sin(1/100)	Cos(1/1000)
2	Sin(2/1)	Cos(2/10)	Sin(3/100)	Cos(2/1000)
3	Sin(3/1)	Cos(3/10)	Sin(3/100)	Cos(3/1000)

Part 2 >> Transformer 동작 원리(1) Positional encoding

Positional encoding 사용 목적: Transformer에게 입력단어의 위치 정보를 알려주기 위해 사용

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

pos: 몇 번째 단어인지(행)
i: 임베딩 벡터 내에서 몇 차원인지
d_model: 임베딩 벡터 차원수

i →

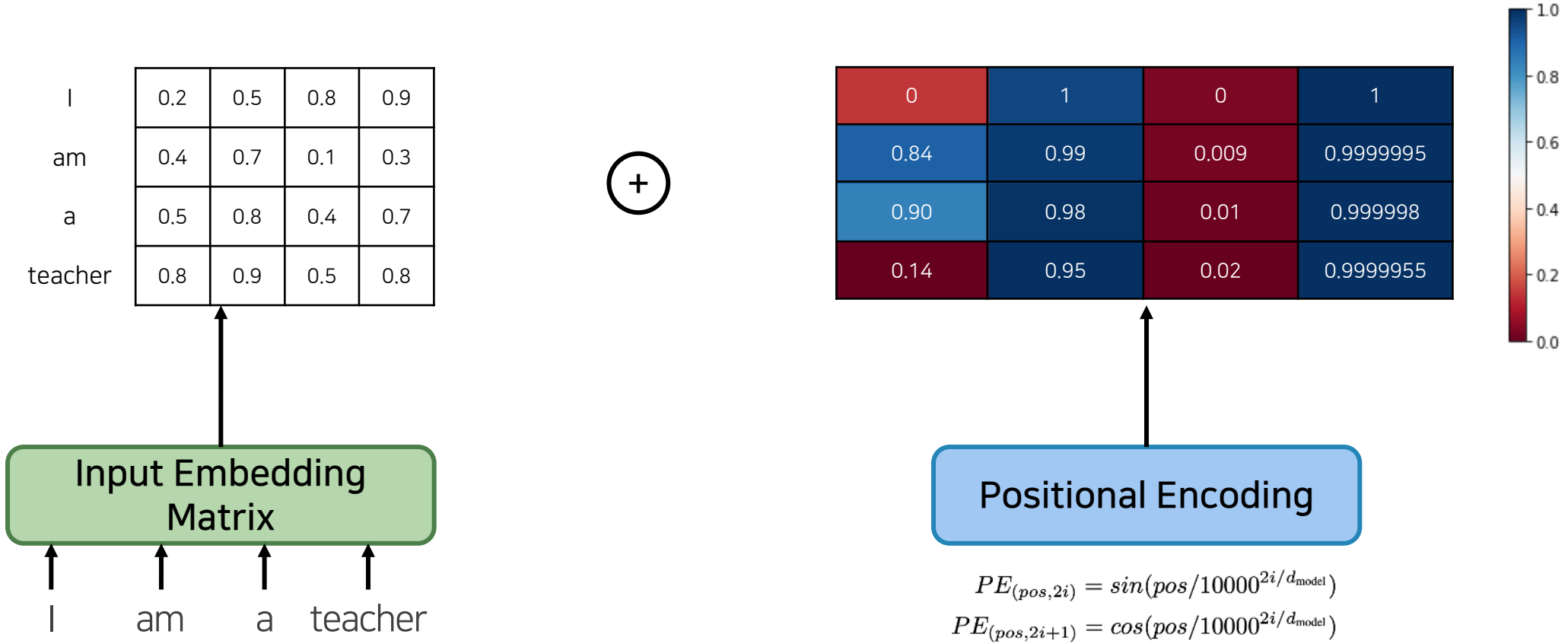
pos ↓

	0=i	1=i+1	2=i	3=i+1
0	Sin(0/1)	Cos(0/10)	Sin(0/100)	Cos(0/1000)
1	Sin(1/1)	Cos(1/10)	Sin(1/100)	Cos(1/1000)
2	Sin(2/1)	Cos(2/10)	Sin(3/100)	Cos(2/1000)
3	Sin(3/1)	Cos(3/10)	Sin(3/100)	Cos(3/1000)

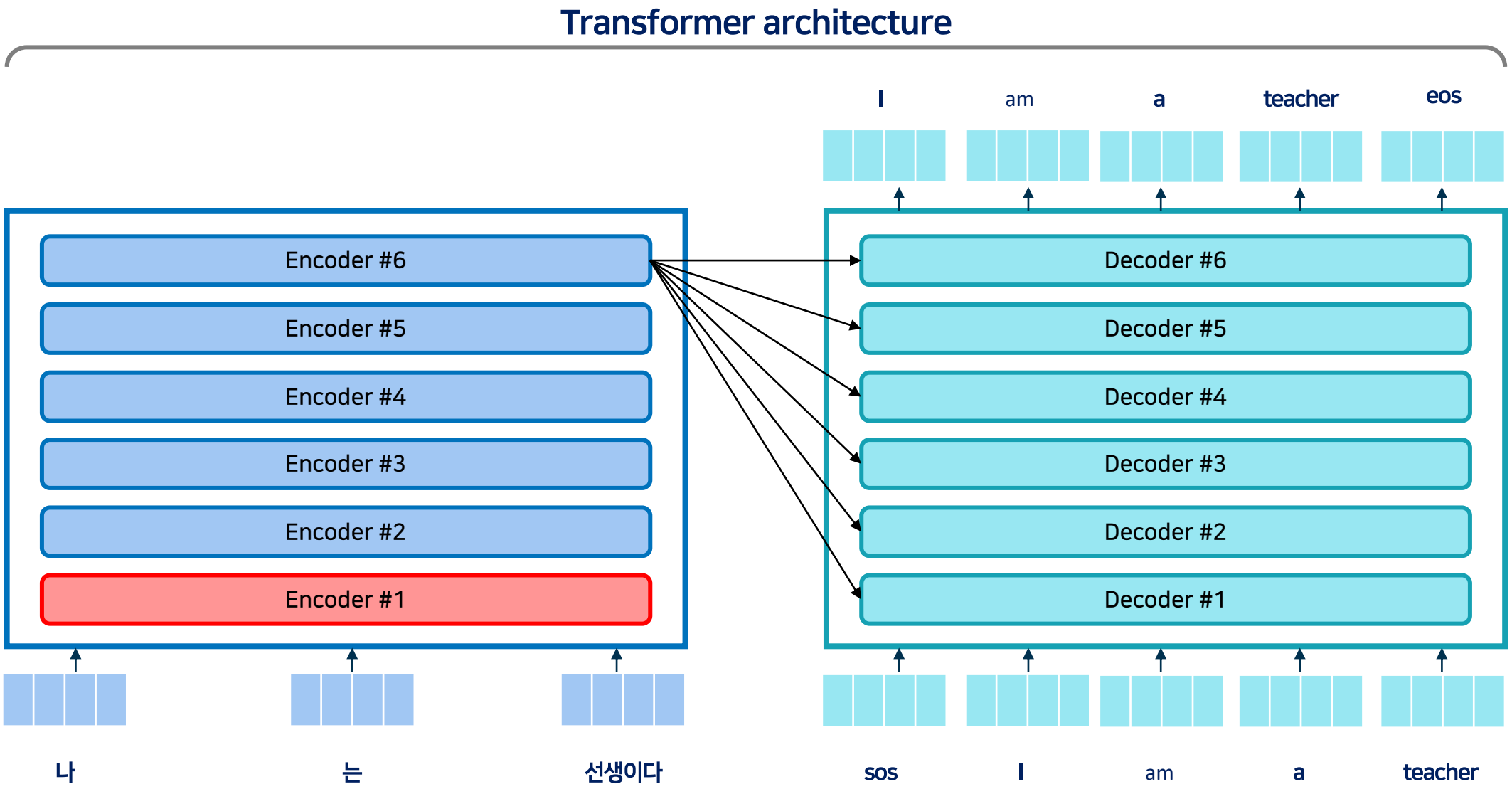
	i	i+1	i	i+1
0	0	1	0	1
1	0.84	0.99	0.009	0.9999995
2	0.90	0.98	0.01	0.999998
3	0.14	0.95	0.02	0.9999955

Part 2 >> Transformer 동작 원리(1) Positional encoding

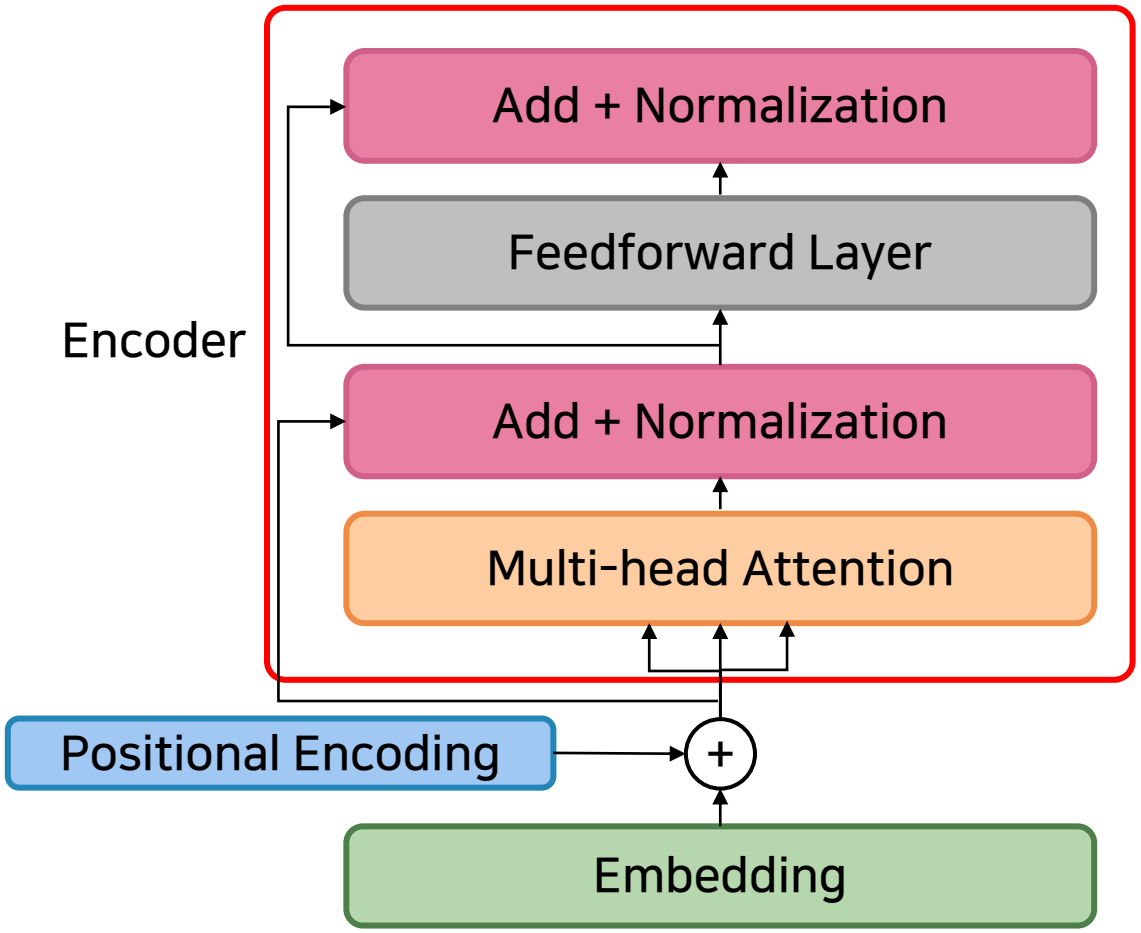
Positional encoding 사용 목적: Transformer에게 입력단어의 위치 정보를 알려주기 위해 사용



Part 2 >> Transformer 동작 원리(2) Encoder

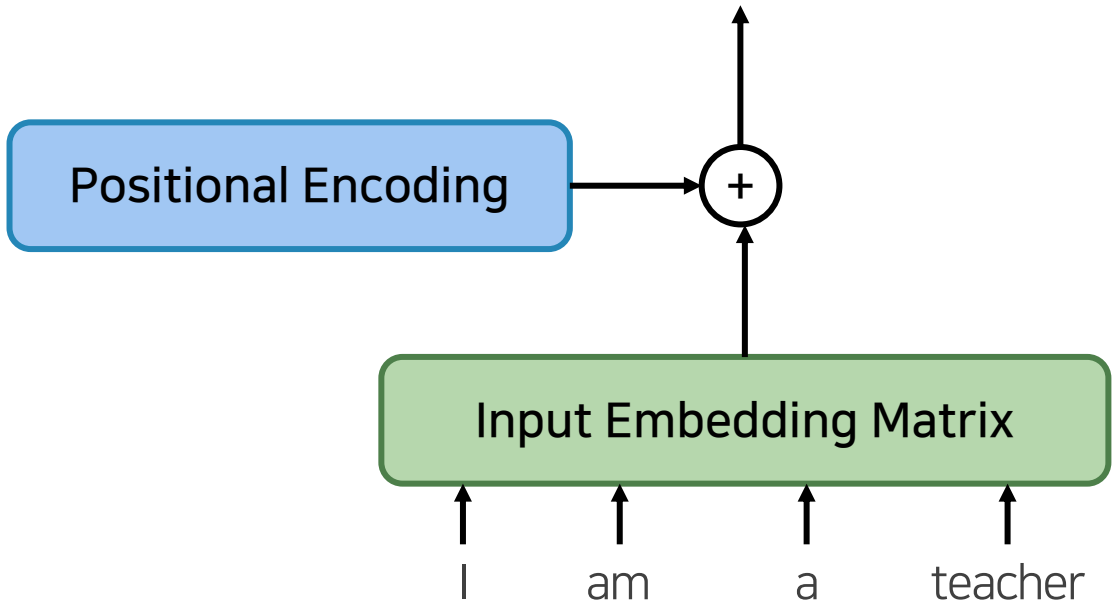


Part 2 >> Transformer 동작 원리(2) Encoder

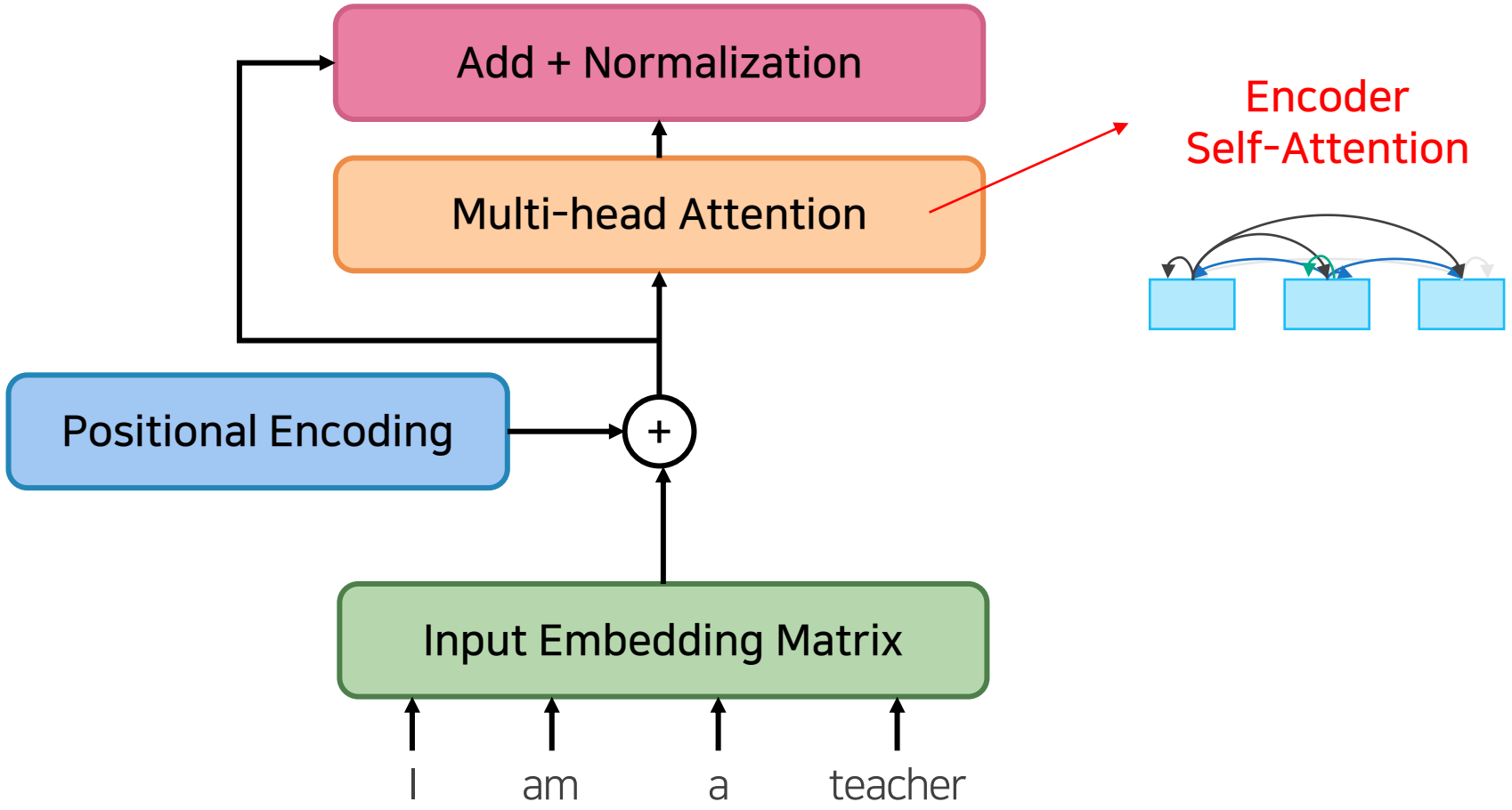


Encoder = Multi-head Attention(Encoder Self-Attention) + FFNN(Feedforward neural network)

Part 2 >> Transformer 동작 원리(2) Encoder

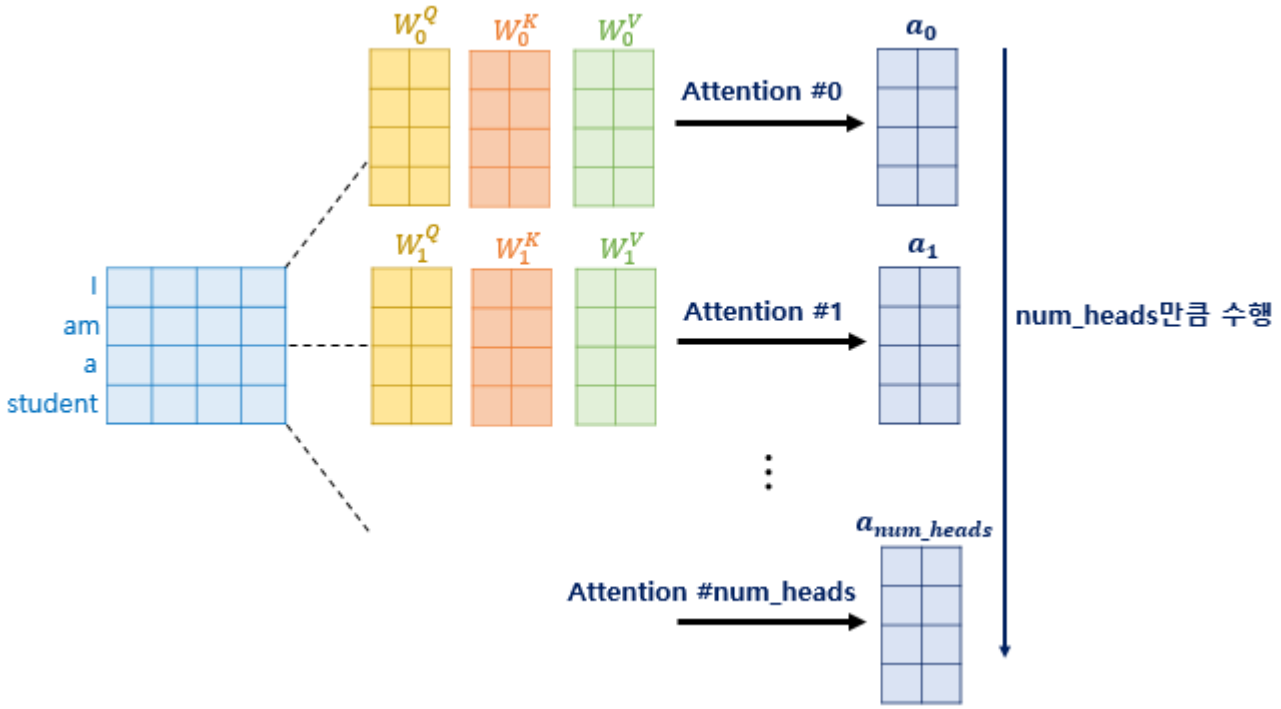


Part 2 >> Transformer 동작 원리(2) Encoder



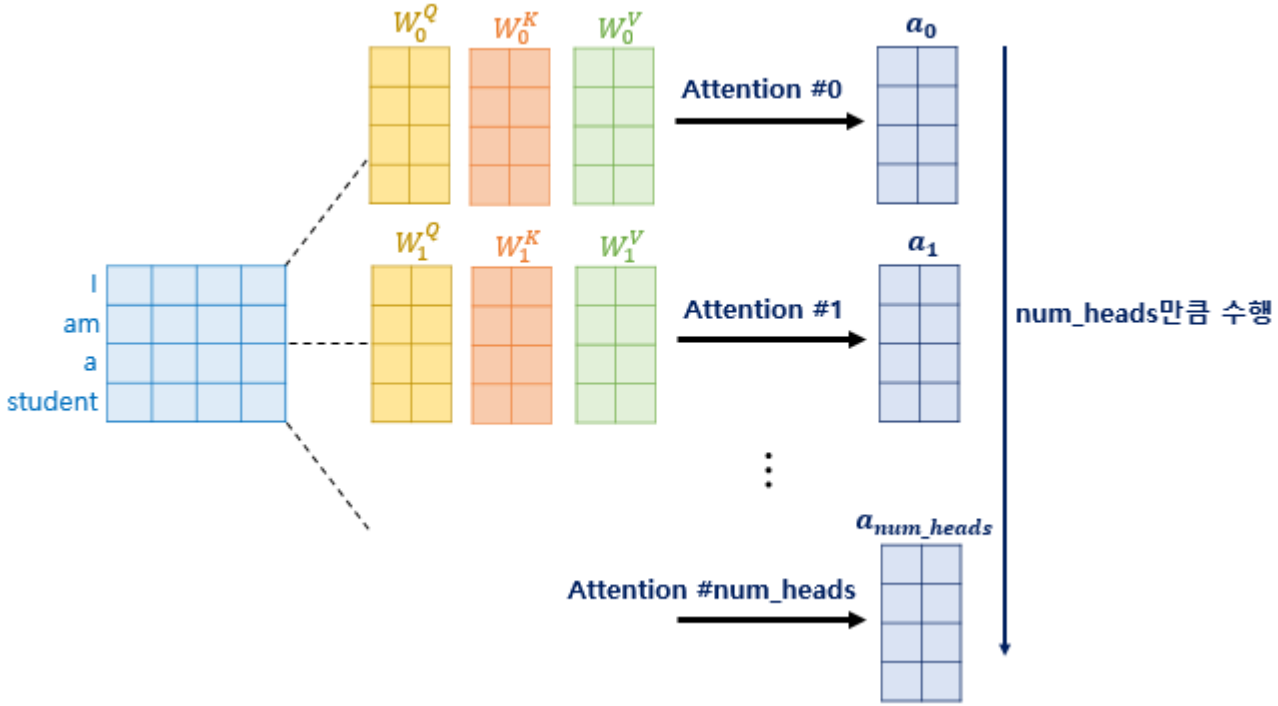
Part 2 >> Transformer 동작 원리(2) Encoder

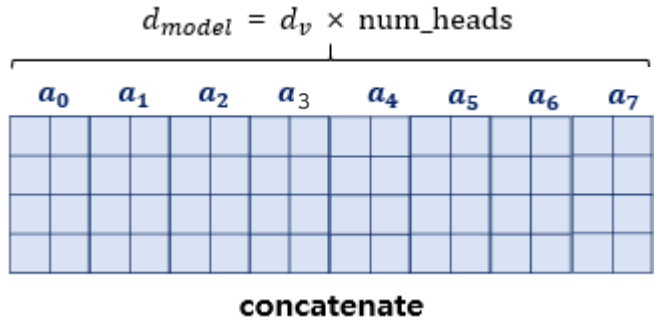
1. Encoder Self-Attention

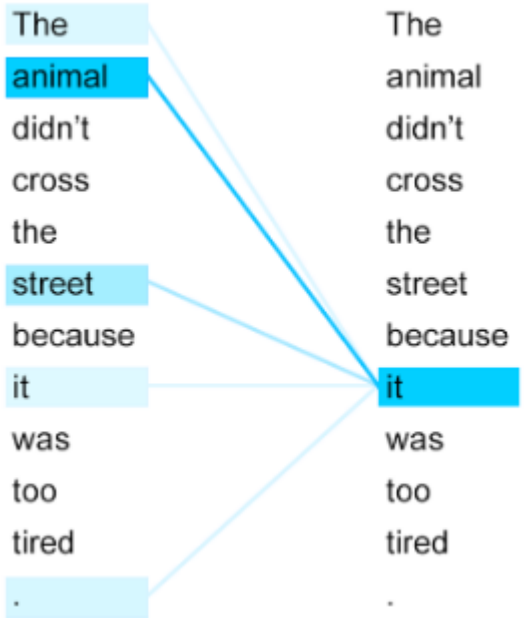


Part 2 >> Transformer 동작 원리(2) Encoder

1. Encoder Self-Attention



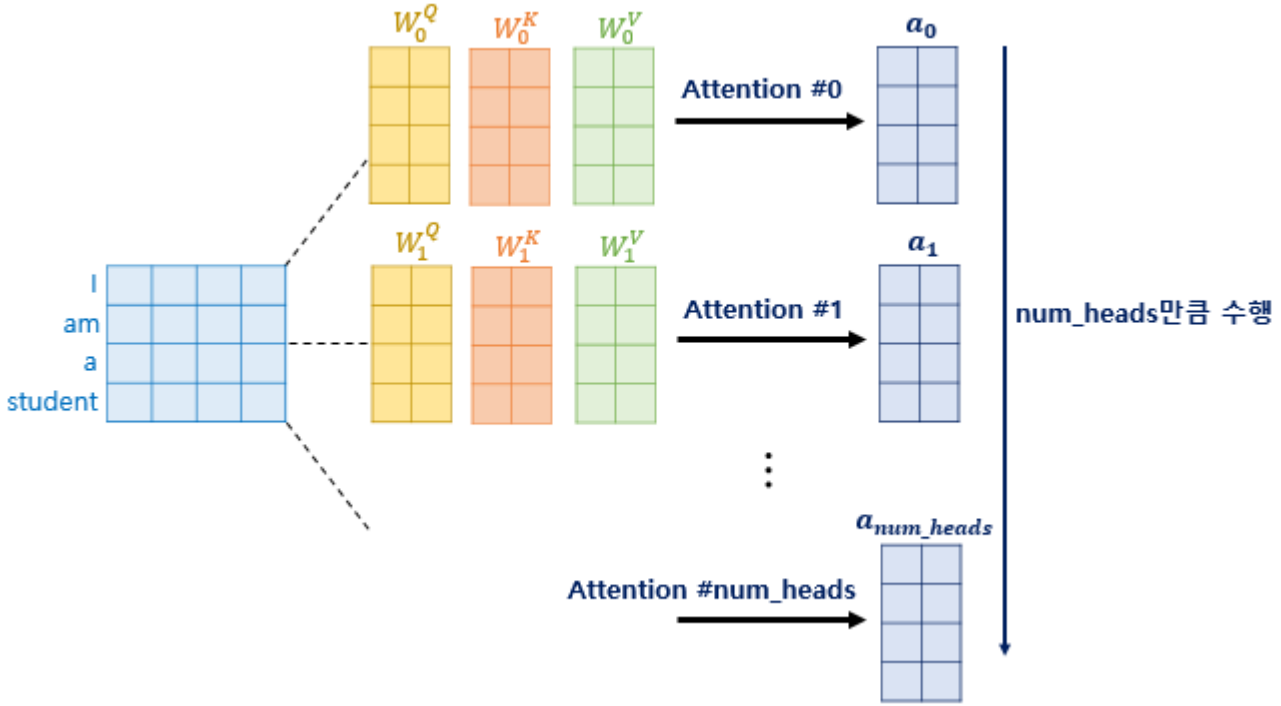


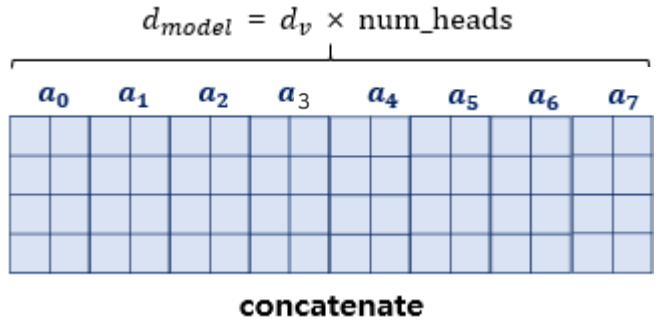


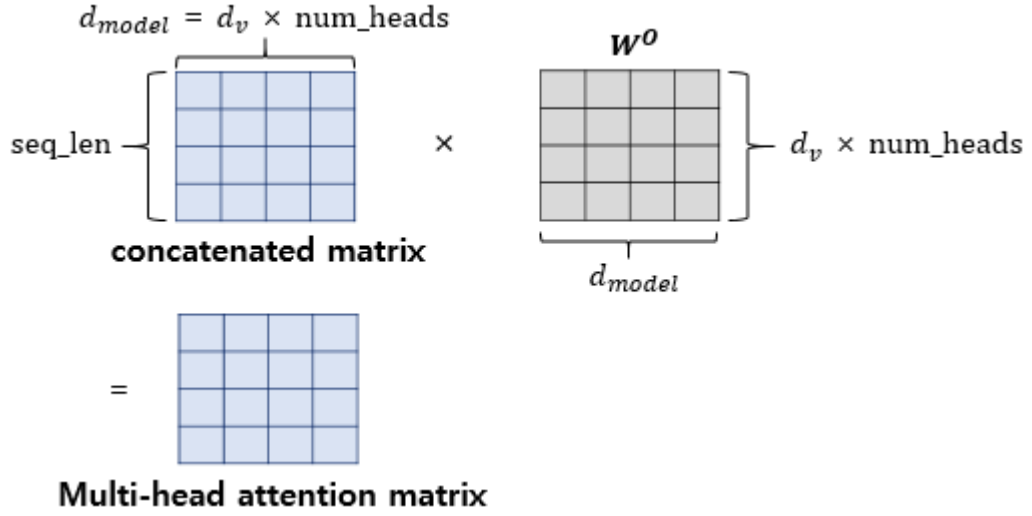
Part 2 >>

Transformer 동작 원리(2) Encoder

1. Encoder Self-Attention



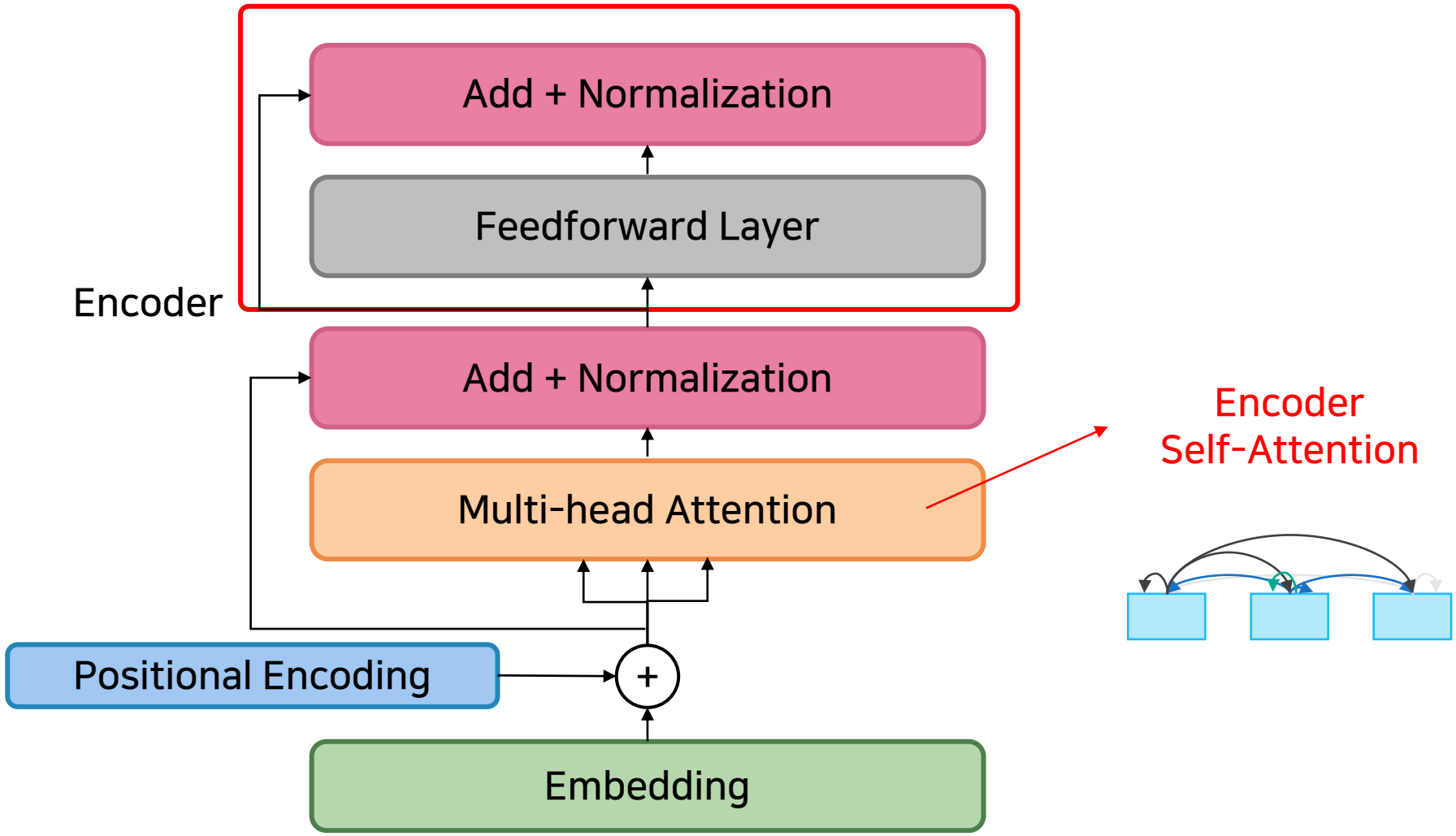




Part 2 >>

Transformer 동작 원리(2) Encoder

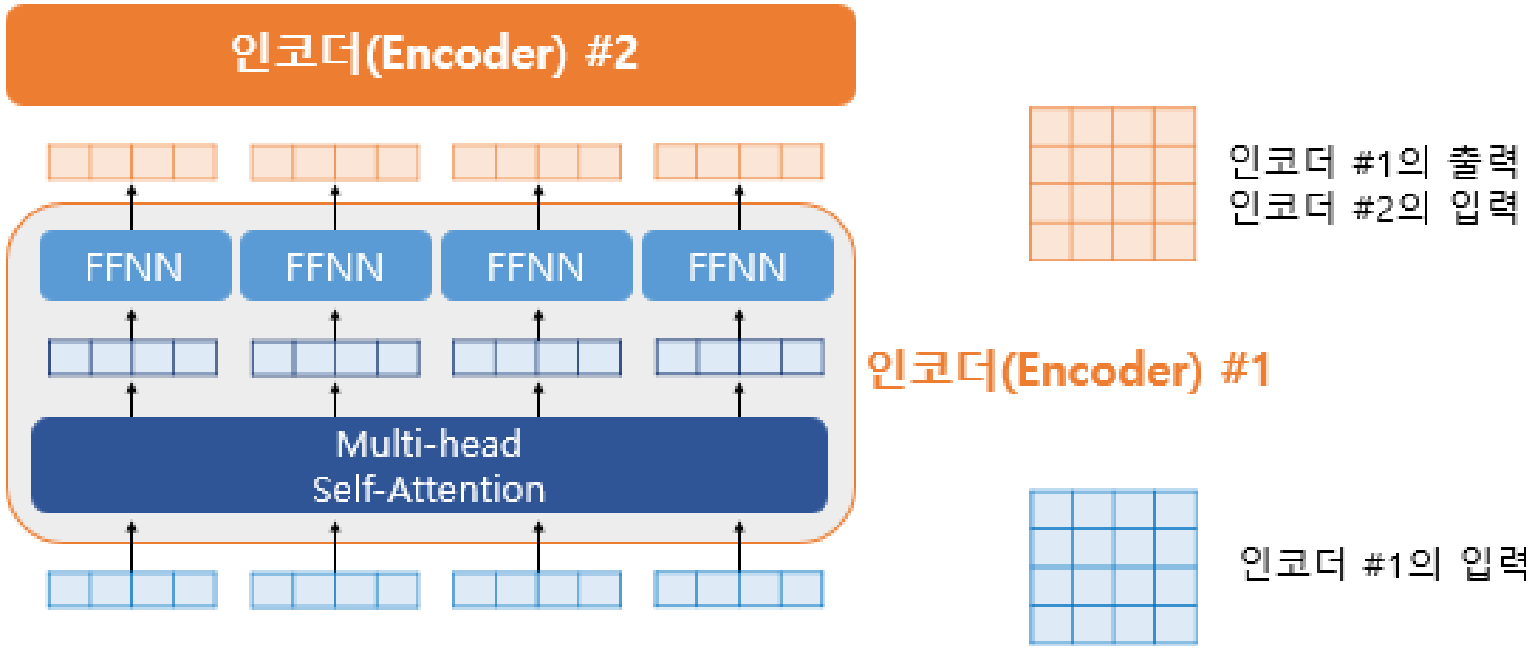
2. FFNN(Feedforward neural network)



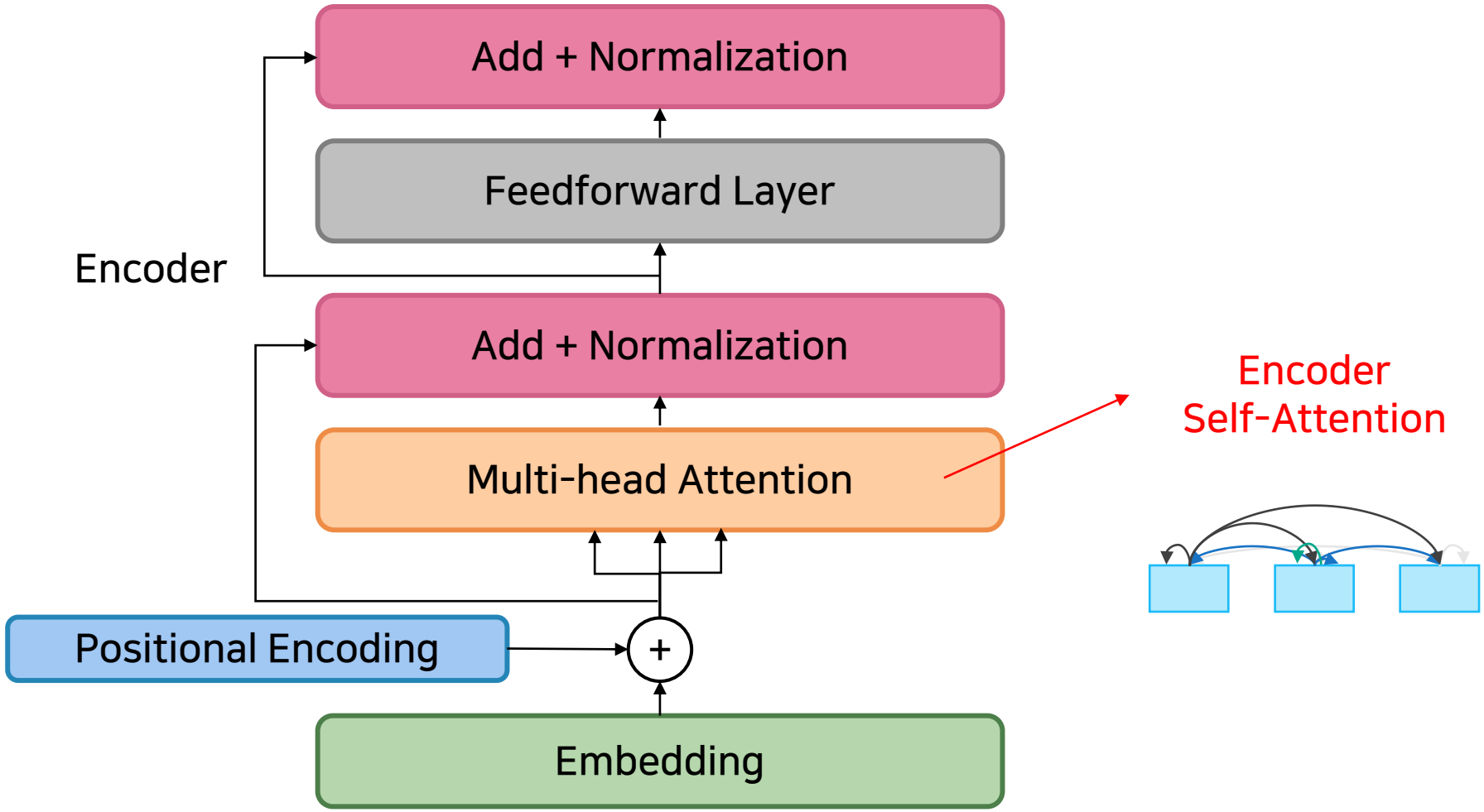
Part 2 >> Transformer 동작 원리(2) Encoder

2. FFNN(Feedforward neural network)

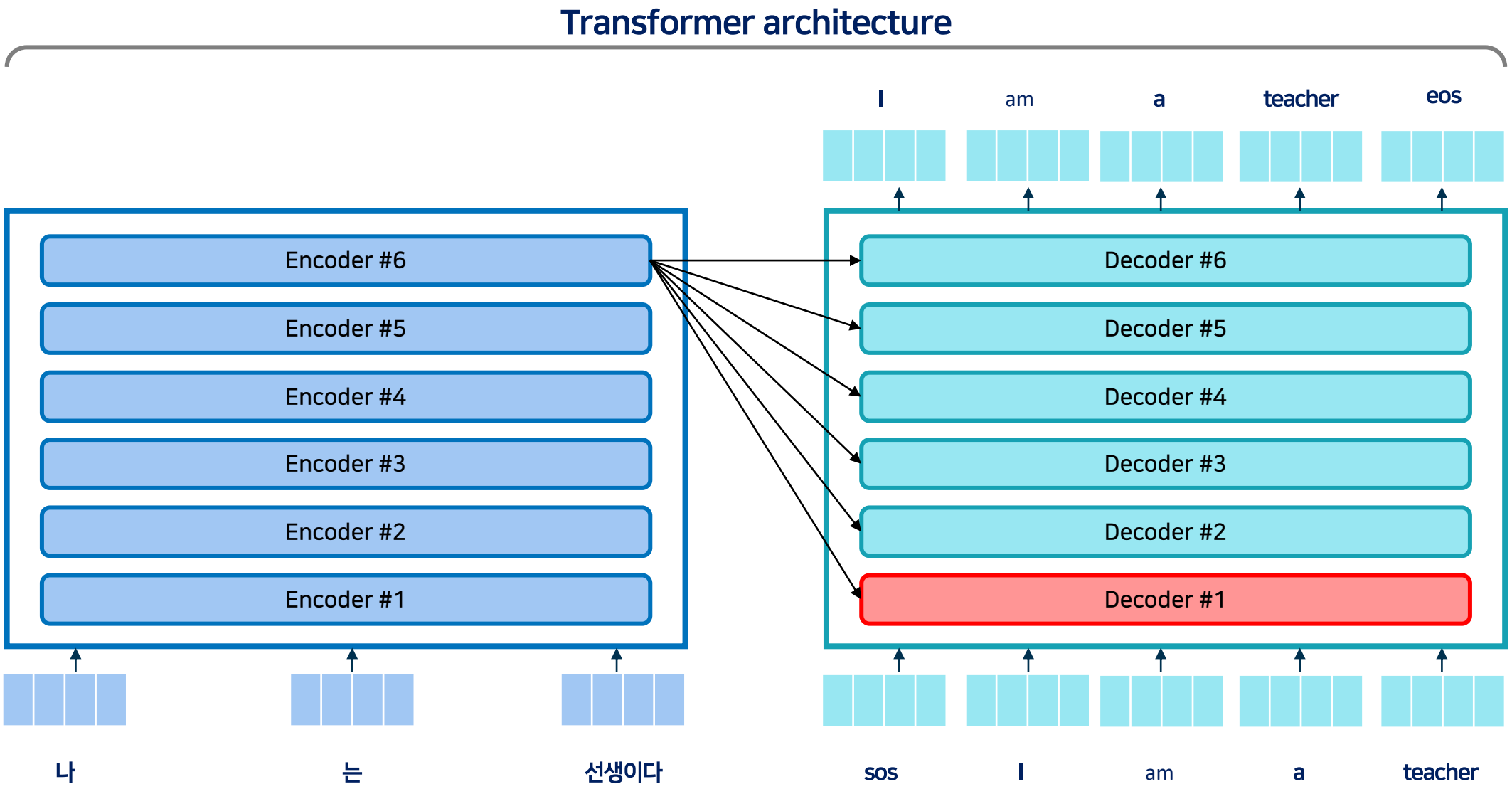
$$FFNN(x) = MAX(0, xW_1 + b_1)W_2 + b_2$$



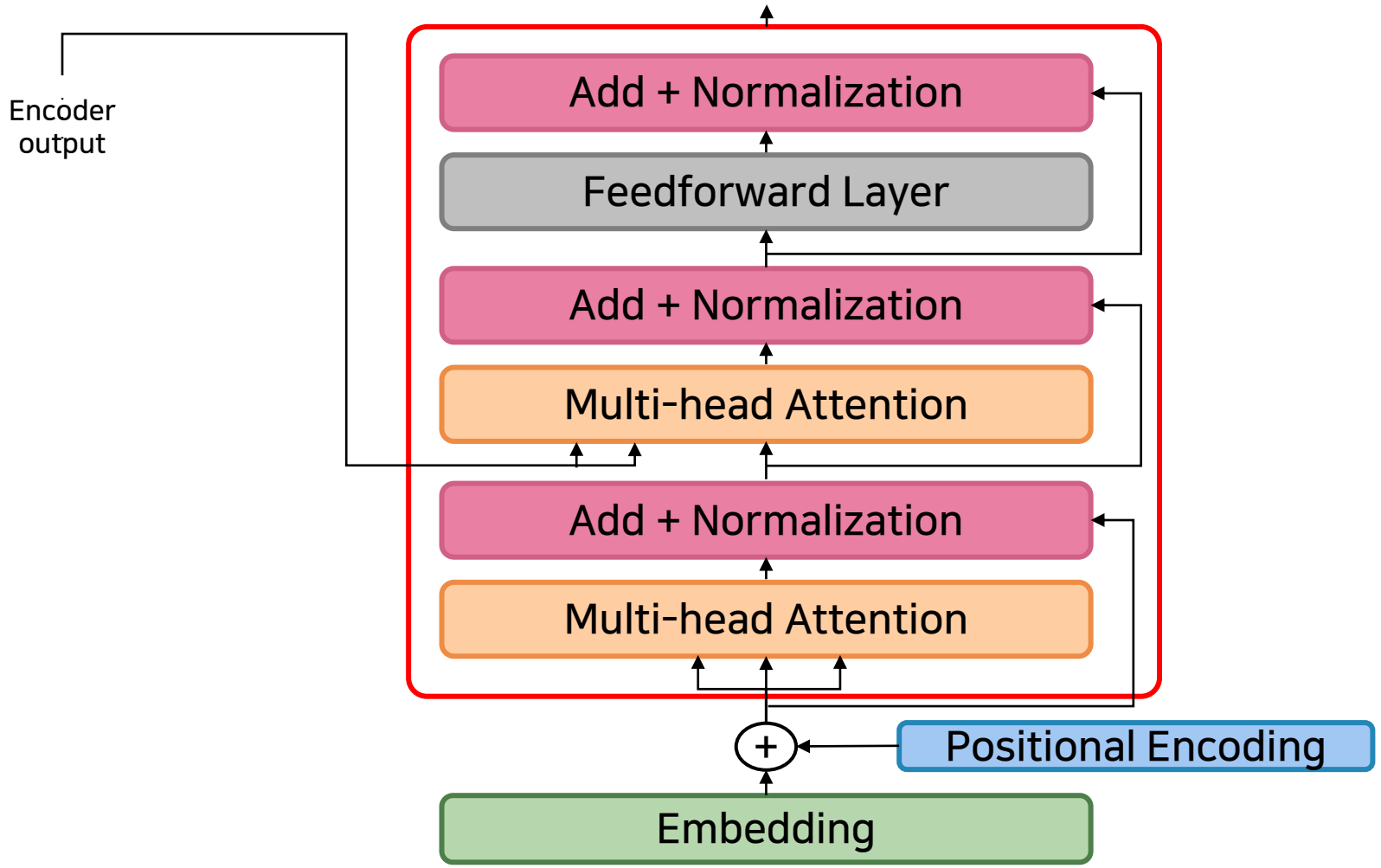
Transformer 동작 원리(2) Encoder



Part 2 >> Transformer 동작 원리(2) Decoder

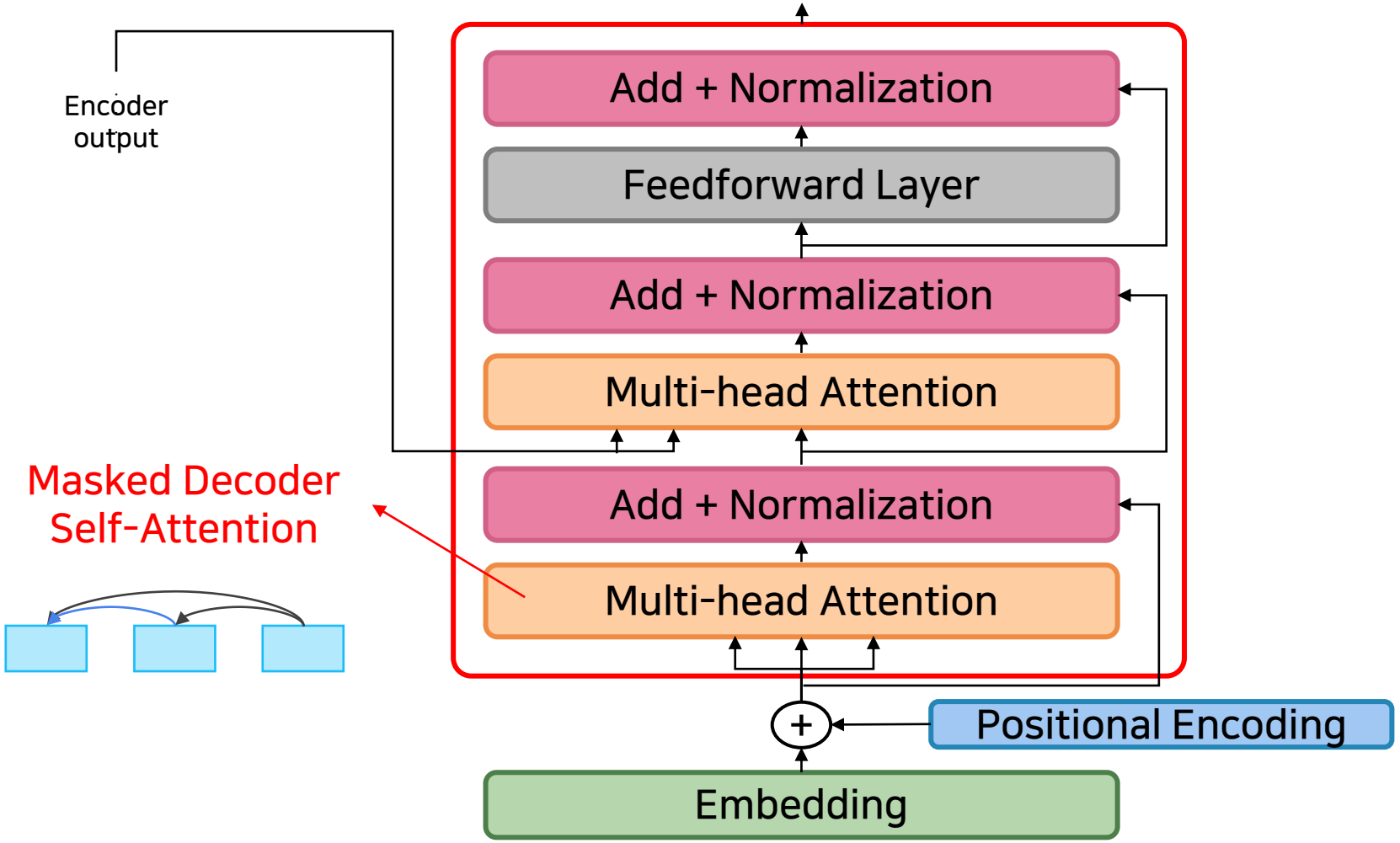


Part 2 >> Transformer 동작 원리(2) Decoder



Decoder = Multi-head Attention(Masked Decoder Self-Attention) + Multi-head attention(Encoder-Decoder Attention) + FFNN(Feedforward neural network)

Part 2 >> Transformer 동작 원리(2) Decoder

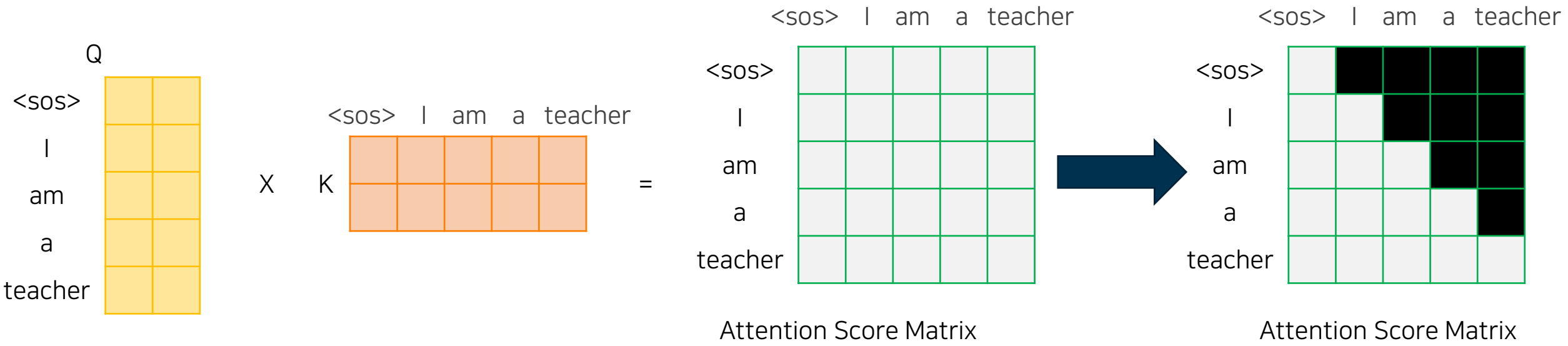


Decoder = Multi-head Attention(Masked Decoder Self-Attention) + Multi-head attention(Encoder-Decoder Attention) + FFNN(Feedforward neural network)

Part 2 >>

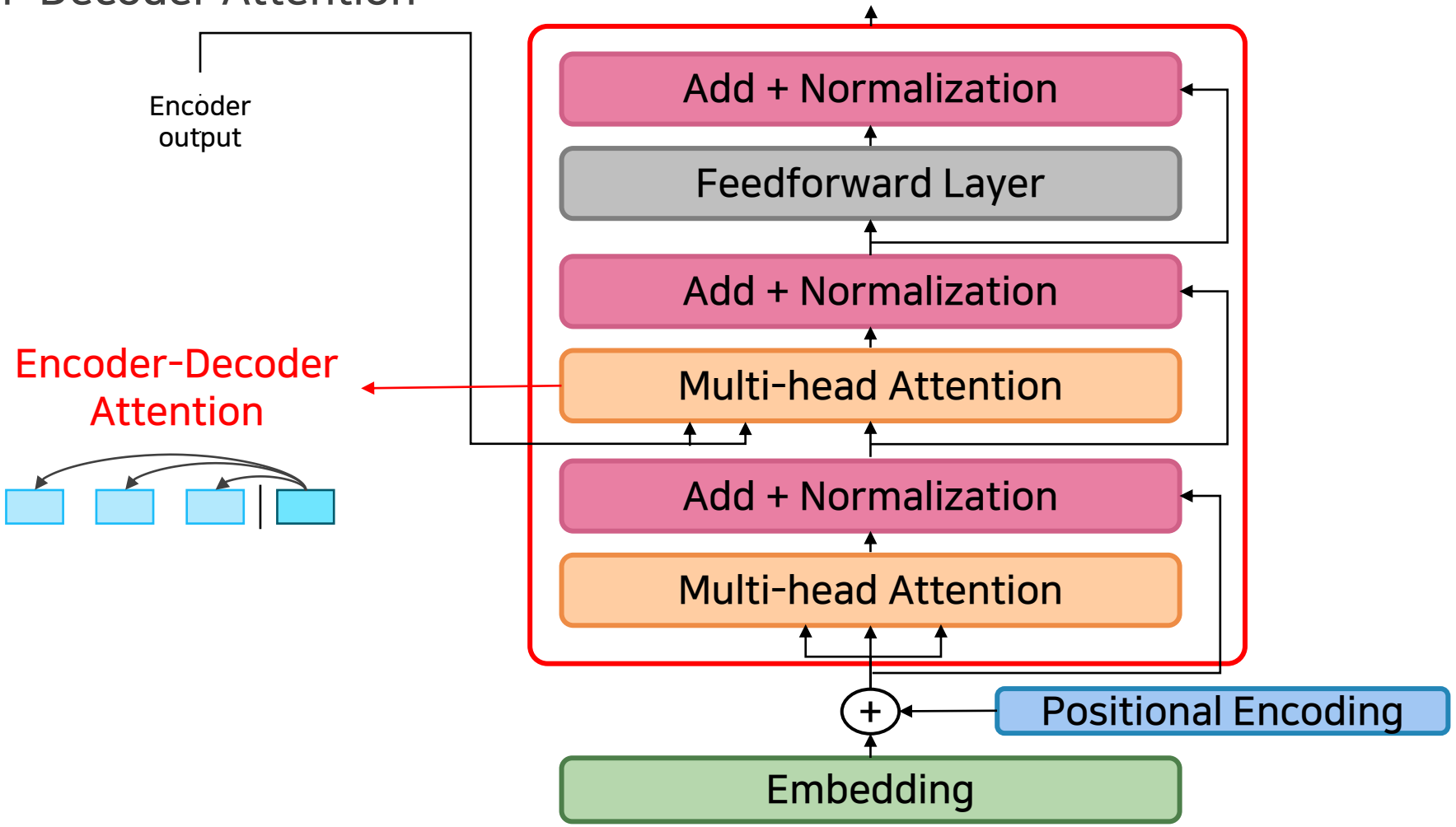
Transformer 동작 원리(2) Decoder

1. Masked Decoder Self-Attention: look-ahead mask



Part 2 >> Transformer 동작 원리(2) Decoder

2. Encoder-Decoder Attention



Part 2 >>

Transformer 동작 원리(2) Decoder

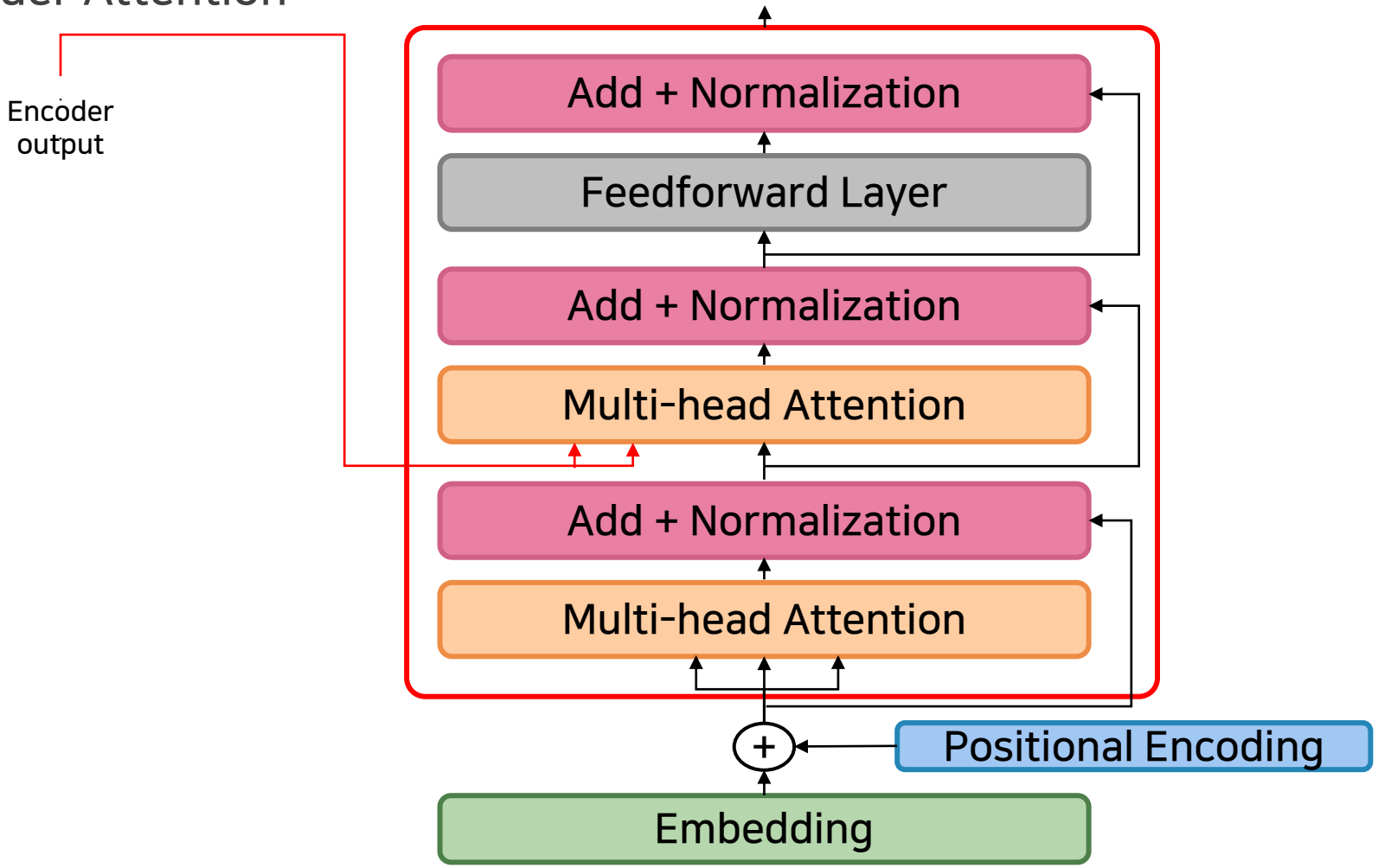
2. Encoder-Decoder Attention

Self-Attention이 아님

Query: Decoder

Key: Encoder

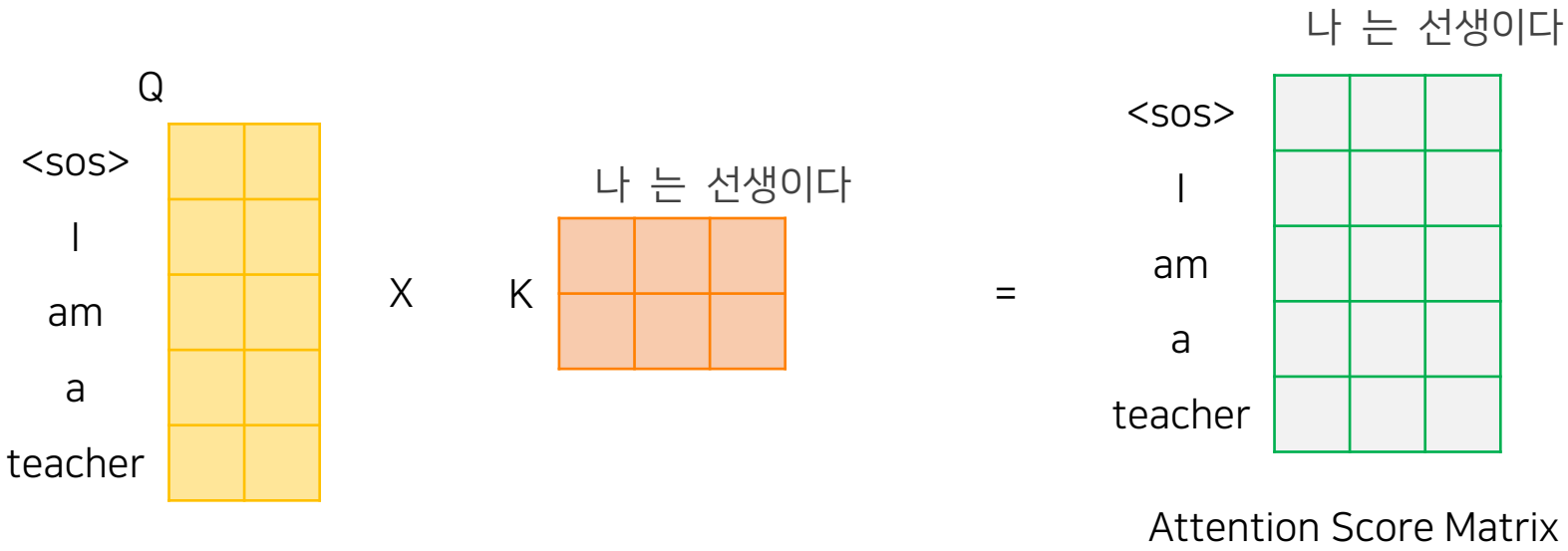
Value: Encoder



Part 2 >>

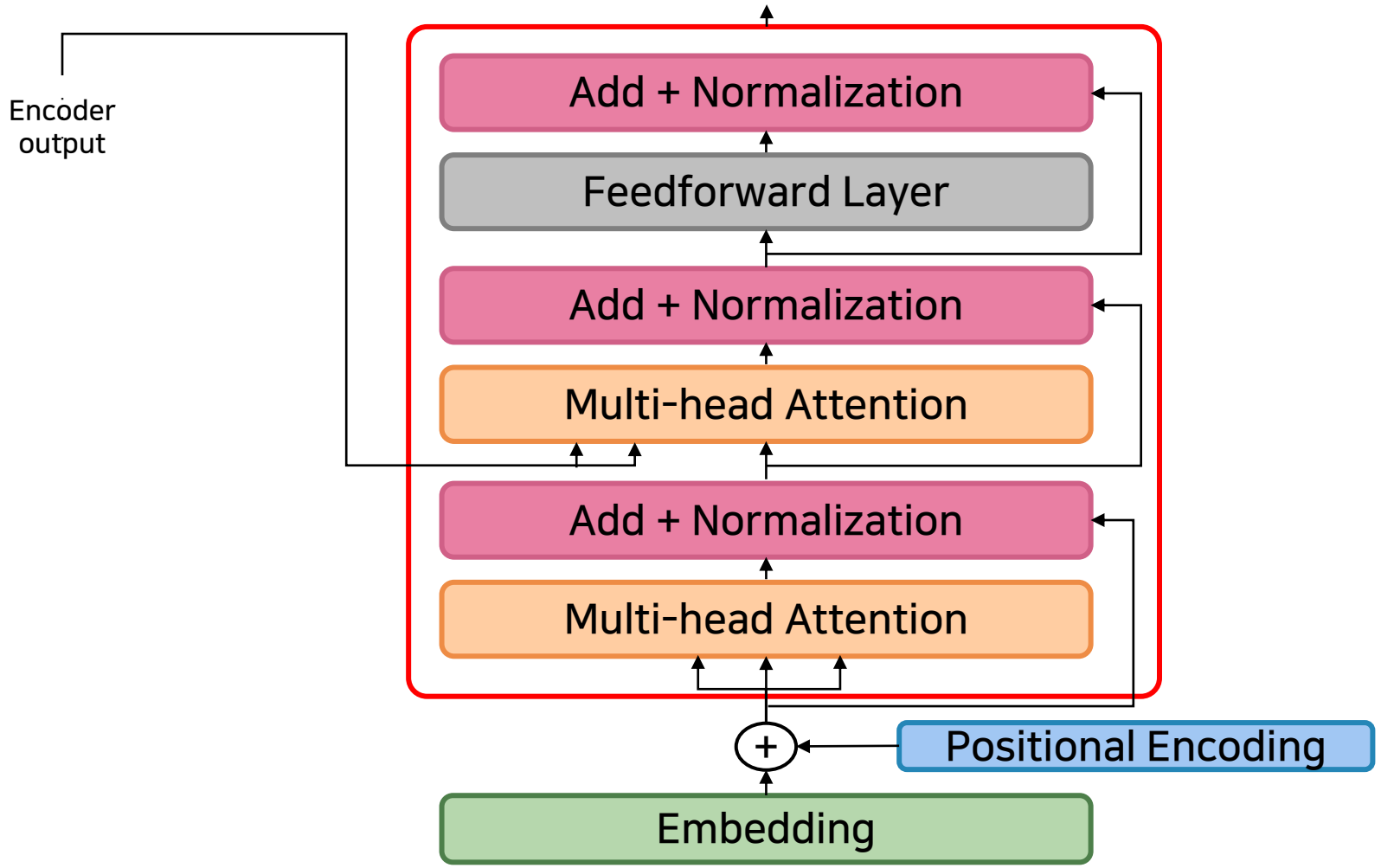
Transformer 동작 원리(2) Decoder

2. Encoder-Decoder Attention



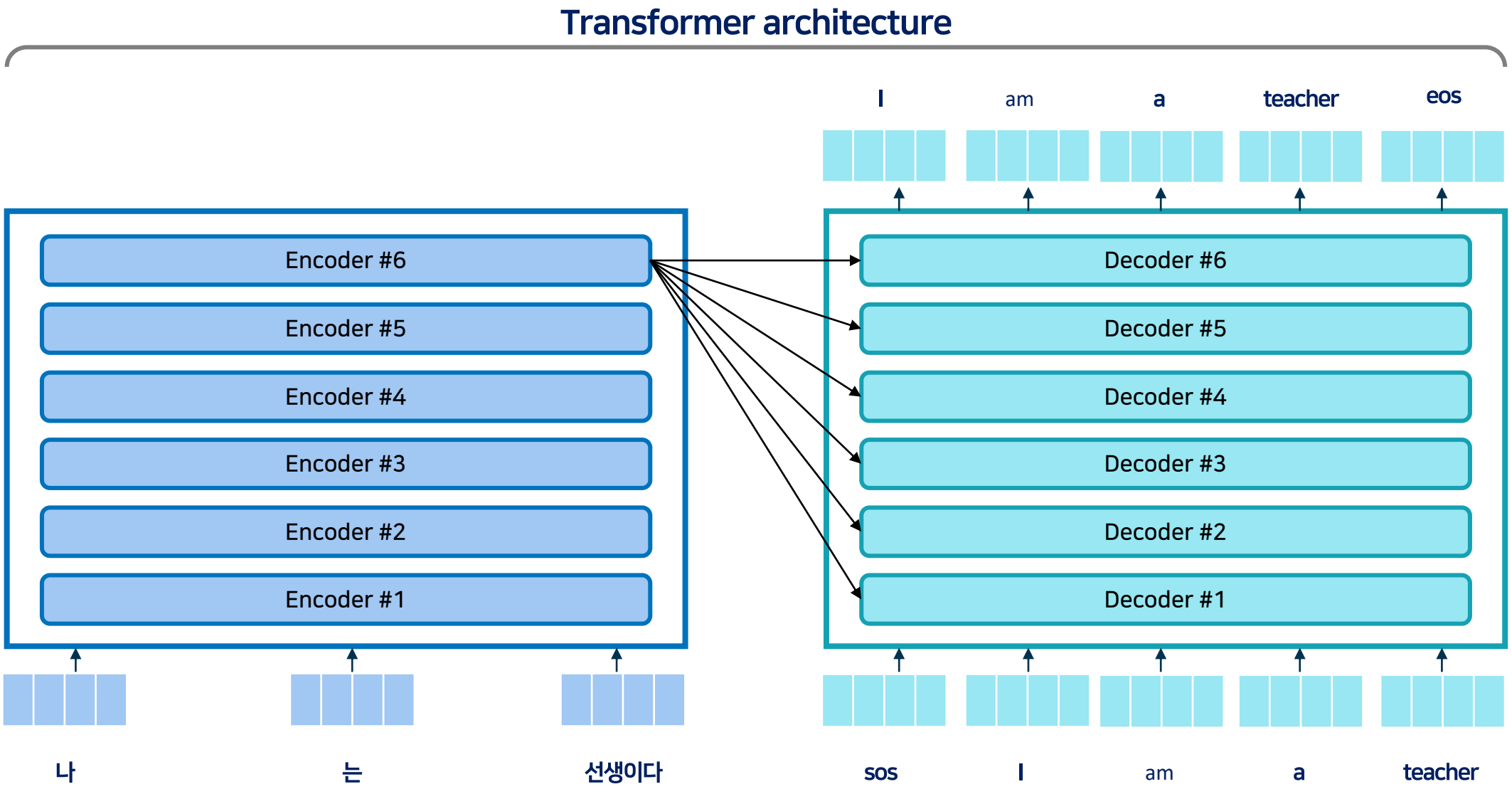
Attention Score Matrix

Part 2 >> Transformer 동작 원리(2) Decoder



Decoder = Multi-head Attention(Masked Decoder Self-Attention) + Multi-head attention(Encoder-Decoder Attention) + FFNN(Feedforward neural network)

Part 2 >> Transformer 동작 원리(4)



3

코드로 살펴보는
Transformer 동작 원리

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

1. Positional encoding

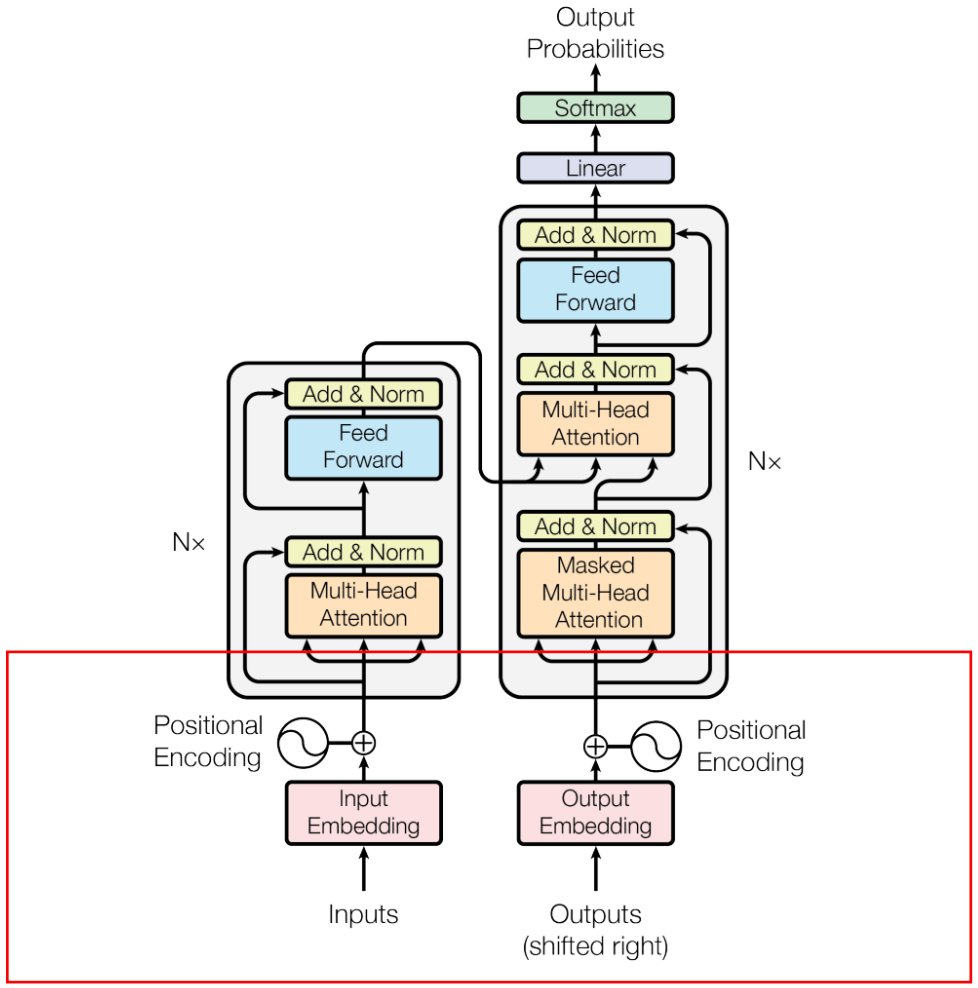


Figure 1: The Transformer - model architecture.

1. 입력: word embedding vector
2. 출력: 위치 정보가 추가된 word embedding vector
3. 동작

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

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

1. Word embedding vector 의 위치별로 위와 같은 식을 적용하여 positional encoding 진행
2. Word embedding vector + positional encoding하여 위치 정보가 추가된 embedding vector return

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

1. Positional encoding

```
class PositionalEncoding(tf.keras.layers.Layer):
  def __init__(self, position, d_model):
    super(PositionalEncoding, self).__init__()
    self.pos_encoding = self.positional_encoding(position, d_model)

  def get_angles(self, position, i, d_model):
    angles = 1 / tf.pow(10000, (2 * (i // 2)) / tf.cast(d_model, tf.float32))
    return position * angles

  def positional_encoding(self, position, d_model):
    angle_rads = self.get_angles(
        position=tf.range(position, dtype=tf.float32)[:, tf.newaxis],
        i=tf.range(d_model, dtype=tf.float32)[tf.newaxis, :],
        d_model=d_model)
```

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

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

	0=i	1=i+1	2=i	3=i+1
0	Sin(0/1)	Cos(0/10)	Sin(0/100)	Cos(0/1000)
1	Sin(1/1)	Cos(1/10)	Sin(1/100)	Cos(1/1000)
2	Sin(2/1)	Cos(2/10)	Sin(3/100)	Cos(2/1000)
3	Sin(3/1)	Cos(3/10)	Sin(3/100)	Cos(3/1000)

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

1. Positional encoding

```
class PositionalEncoding(tf.keras.layers.Layer):
  def __init__(self, position, d_model):
    super(PositionalEncoding, self).__init__()
    self.pos_encoding = self.positional_encoding(position, d_model)

  def get_angles(self, position, i, d_model):
    angles = 1 / tf.pow(10000, (2 * (i // 2)) / tf.cast(d_model, tf.float32))
    return position * angles

  def positional_encoding(self, position, d_model):
    angle_rads = self.get_angles(
        position=tf.range(position, dtype=tf.float32)[:, tf.newaxis],
        i=tf.range(d_model, dtype=tf.float32)[tf.newaxis, :],
        d_model=d_model)
```

1. get_angles 함수에 인자를 주어 호출하는 곳
    1. position: (단어 길이, 1)
    2. i: (1, d_model)

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

1. Positional encoding

```
  # 배열의 짝수 인덱스(2i)에는 사인 함수 적용
  sines = tf.math.sin(angle_rads[:, 0::2])

  # 배열의 홀수 인덱스(2i+1)에는 코사인 함수 적용
  cosines = tf.math.cos(angle_rads[:, 1::2])

  angle_rads = np.zeros(angle_rads.shape)
  angle_rads[:, 0::2] = sines
  angle_rads[:, 1::2] = cosines
  pos_encoding = tf.constant(angle_rads)
  pos_encoding = pos_encoding[tf.newaxis, ...]

  print(pos_encoding)
  return tf.cast(pos_encoding, tf.float32)

def call(self, inputs):
  return inputs + self.pos_encoding[:, :tf.shape(inputs)[1], :]
```

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

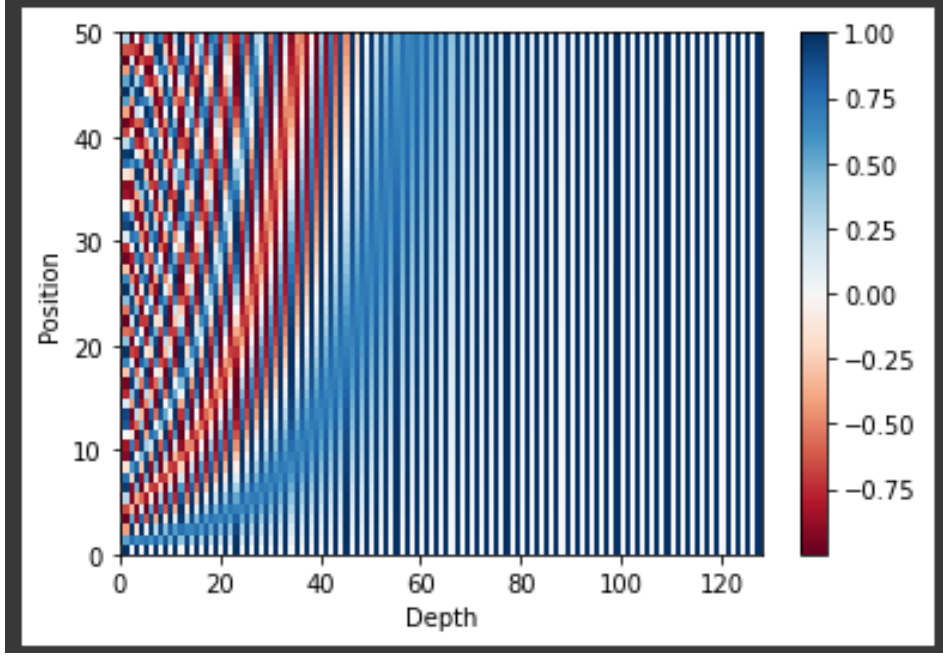
$$PE_{(pos,\,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

	0=i	1=i+1	2=i	3=i+1
0	Sin(0/1)	Cos(0/10)	Sin(0/100)	Cos(0/1000)
1	Sin(1/1)	Cos(1/10)	Sin(1/100)	Cos(1/1000)
2	Sin(2/1)	Cos(2/10)	Sin(3/100)	Cos(2/1000)
3	Sin(3/1)	Cos(3/10)	Sin(3/100)	Cos(3/1000)

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

1. Positional encoding

```
1 sample_pos_encoding = PositionalEncoding(50, 128)
2 plt.pcolormesh(sample_pos_encoding.pos_encoding.numpy()[0], cmap='RdBu')
3 plt.xlabel('Depth')
4 plt.xlim((0, 128))
5 plt.ylabel('Position')
6 plt.colorbar()
7 plt.show()
```



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

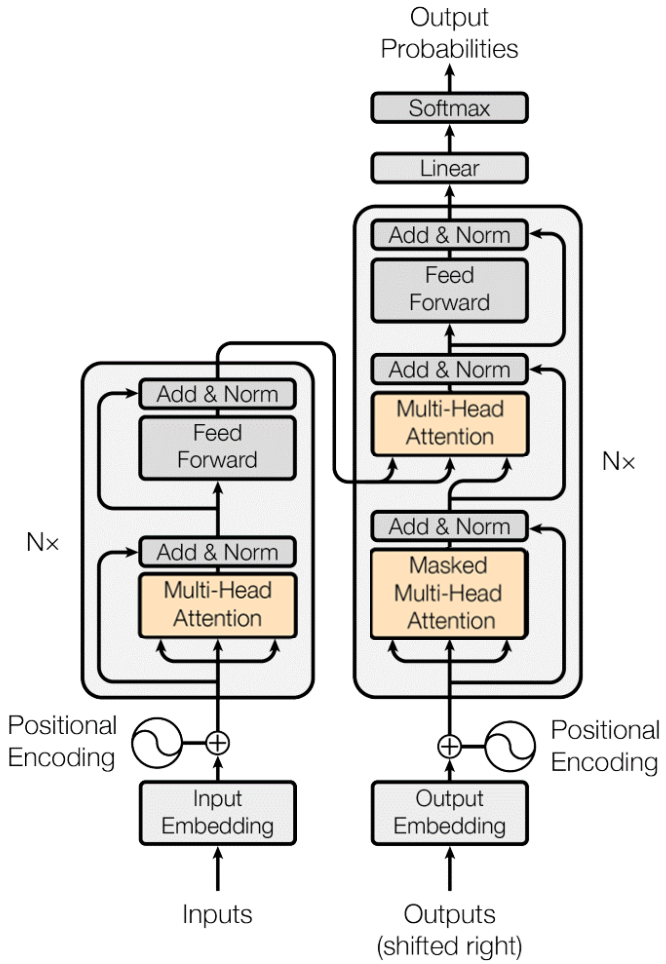
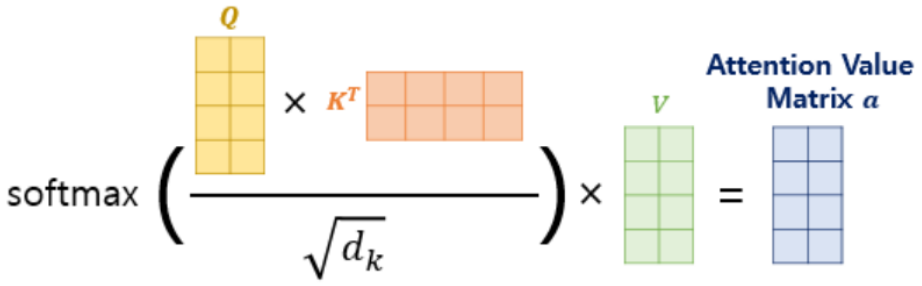


Figure 1: The Transformer - model architecture.

1. 입력: q, k, v
2. 출력: q, k 기반으로 구한 attention score matrix 에 v를 곱한 값
3. 동작
    1. Attention mechanism

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



    2. Scaled_dot_product를 num_head 만큼 수행

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

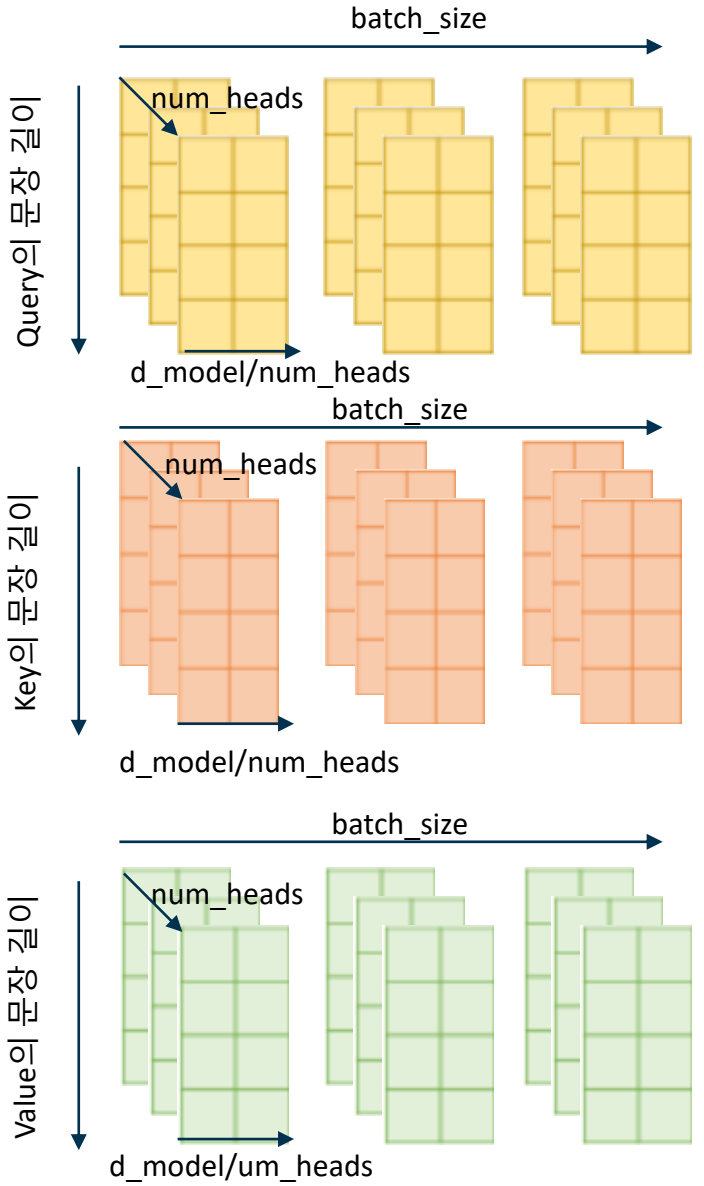
2. Multi-head attention

```
def scaled_dot_product_attention(query, key, value, mask):
  # query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # key 크기 : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # value 크기 : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  # padding_mask : (batch_size, 1, 1, key의 문장 길이)

  # Q와 K의 곱. 어텐션 스코어 행렬.
  matmul_qk = tf.matmul(query, key, transpose_b=True)

  # 스케일링
  # dk의 루트값으로 나눠준다.
  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = matmul_qk / tf.math.sqrt(depth)

  # 마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣는다.
  # 매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
  if mask is not None:
    logits += (mask * -1e9)

  # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
  # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  # output : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  output = tf.matmul(attention_weights, value)

  return output, attention_weights
```

Query

Key

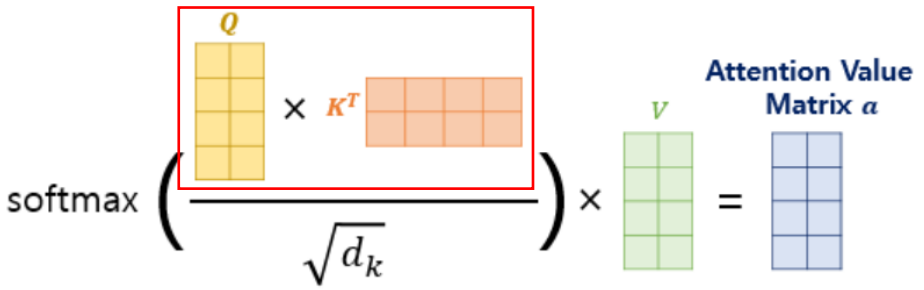
Value



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

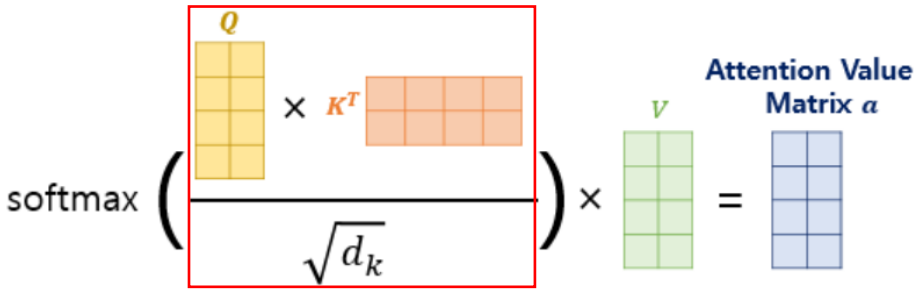
```
def scaled_dot_product_attention(query, key, value, mask):
  # query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # key 크기 : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # value 크기 : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  # padding_mask : (batch_size, 1, 1, key의 문장 길이)

  # Q와 K의 곱. 어텐션 스코어 행렬.
  matmul_qk = tf.matmul(query, key, transpose_b=True)

  # 스케일링
  # dk의 루트값으로 나눠준다.
  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = matmul_qk / tf.math.sqrt(depth)

  # 마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣는다.
  # 매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
  if mask is not None:
    logits += (mask * -1e9)

  # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
  # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  # output : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  output = tf.matmul(attention_weights, value)

  return output, attention_weights
```



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

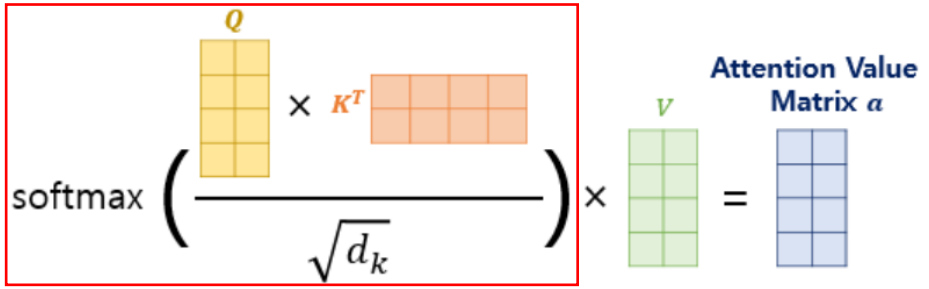
```python
def scaled_dot_product_attention(query, key, value, mask):
  # query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # key 크기 : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # value 크기 : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  # padding_mask : (batch_size, 1, 1, key의 문장 길이)

  # Q와 K의 곱. 어텐션 스코어 행렬.
  matmul_qk = tf.matmul(query, key, transpose_b=True)

  # 스케일링
  # dk의 루트값으로 나눠준다.
  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = matmul_qk / tf.math.sqrt(depth)

  # 마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣는다.
  # 매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
  if mask is not None:
    logits += (mask * -1e9)

  # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
  # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  # output : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  output = tf.matmul(attention_weights, value)

  return output, attention_weights
```



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

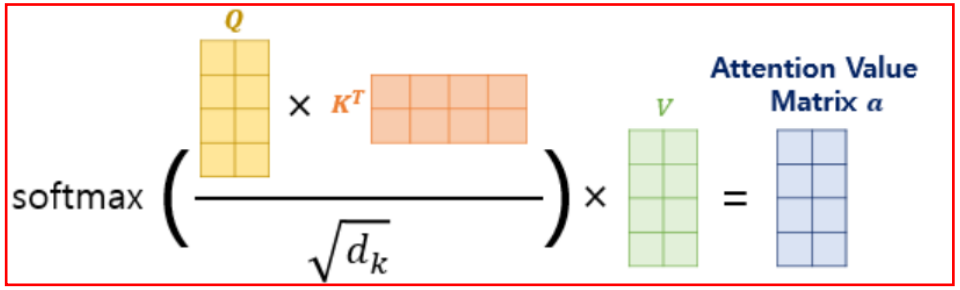
```
def scaled_dot_product_attention(query, key, value, mask):
  # query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # key 크기 : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # value 크기 : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  # padding_mask : (batch_size, 1, 1, key의 문장 길이)

  # Q와 K의 곱. 어텐션 스코어 행렬.
  matmul_qk = tf.matmul(query, key, transpose_b=True)

  # 스케일링
  # dk의 루트값으로 나눠준다.
  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = matmul_qk / tf.math.sqrt(depth)

  # 마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣는다.
  # 매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
  if mask is not None:
    logits += (mask * -1e9)

  # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
  # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  # output : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  output = tf.matmul(attention_weights, value)

  return output, attention_weights
```



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def scaled_dot_product_attention(query, key, value, mask):
  # query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # key 크기 : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # value 크기 : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  # padding_mask : (batch_size, 1, 1, key의 문장 길이)

  # Q와 K의 곱. 어텐션 스코어 행렬.
  matmul_qk = tf.matmul(query, key, transpose_b=True)

  # 스케일링
  # dk의 루트값으로 나눠준다.
  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = matmul_qk / tf.math.sqrt(depth)

  # 마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣는다.
  # 매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
  if mask is not None:
    logits += (mask * -1e9)

  # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
  # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  # output : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  output = tf.matmul(attention_weights, value)

  return output, attention_weights
```



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads, name="multi_head_attention"):
    super(MultiHeadAttention, self).__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    # d_model을 num_heads로 나눈 값.
    # 논문 기준 : 64
    self.depth = d_model // self.num_heads

    # WQ, WK, WV에 해당하는 밀집층 정의
    self.query_dense = tf.keras.layers.Dense(units=d_model)
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    # WO에 해당하는 밀집층 정의
    self.dense = tf.keras.layers.Dense(units=d_model)

  # num_heads 개수만큼 q, k, v를 split하는 함수
  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

d_model % num_heads == 0
임베딩 벡터 차원을 헤드 수로 나눌 때 딱 나누어 떨어지는지 체크

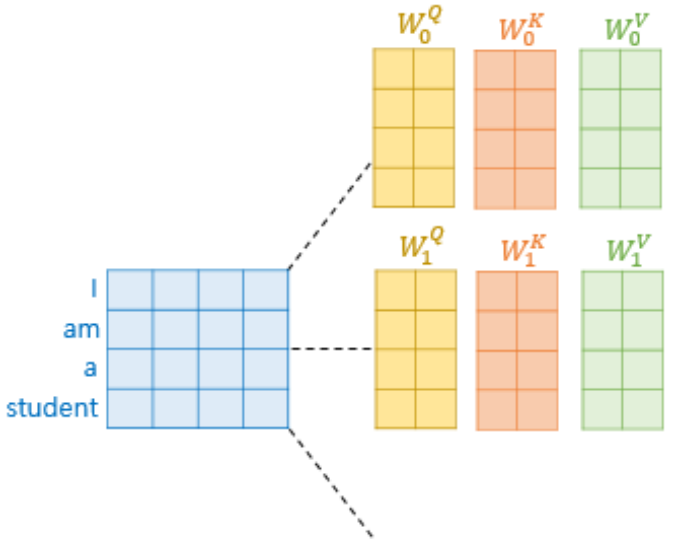
d_model % num_heads == 0
나누어 떨어지면 임베딩 베터 차원//헤드 수

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads, name="multi_head_attention"):
    super(MultiHeadAttention, self).__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    # d_model을 num_heads로 나눈 값.
    # 논문 기준 : 64
    self.depth = d_model // self.num_heads

    # WQ, WK, WV에 해당하는 밀집층 정의
    self.query_dense = tf.keras.layers.Dense(units=d_model)
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    # WO에 해당하는 밀집층 정의
    self.dense = tf.keras.layers.Dense(units=d_model)

  # num_heads 개수만큼 q, k, v를 split하는 함수
  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

임베딩 단어 벡터에 일련의 가중치를 곱하여 Query, key, value 생성

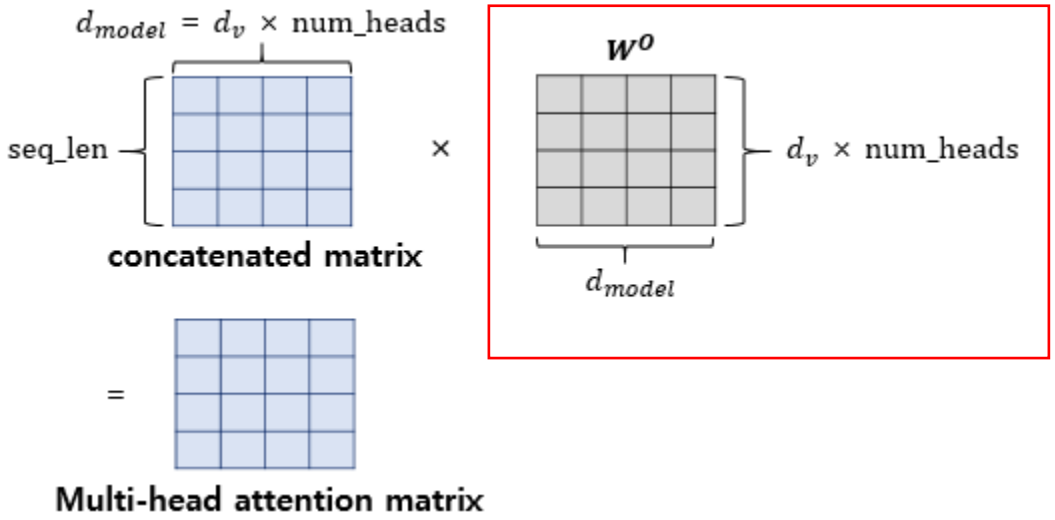


Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads, name="multi_head_attention"):
    super(MultiHeadAttention, self).__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    # d_model을 num_heads로 나눈 값.
    # 논문 기준 : 64
    self.depth = d_model // self.num_heads

    # WQ, WK, WV에 해당하는 밀집층 정의
    self.query_dense = tf.keras.layers.Dense(units=d_model)
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    # WO에 해당하는 밀집층 정의
    self.dense = tf.keras.layers.Dense(units=d_model)

  # num_heads 개수만큼 q, k, v를 split하는 함수
  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

Concatenated matrix에 곱해지는 weight 학습



Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

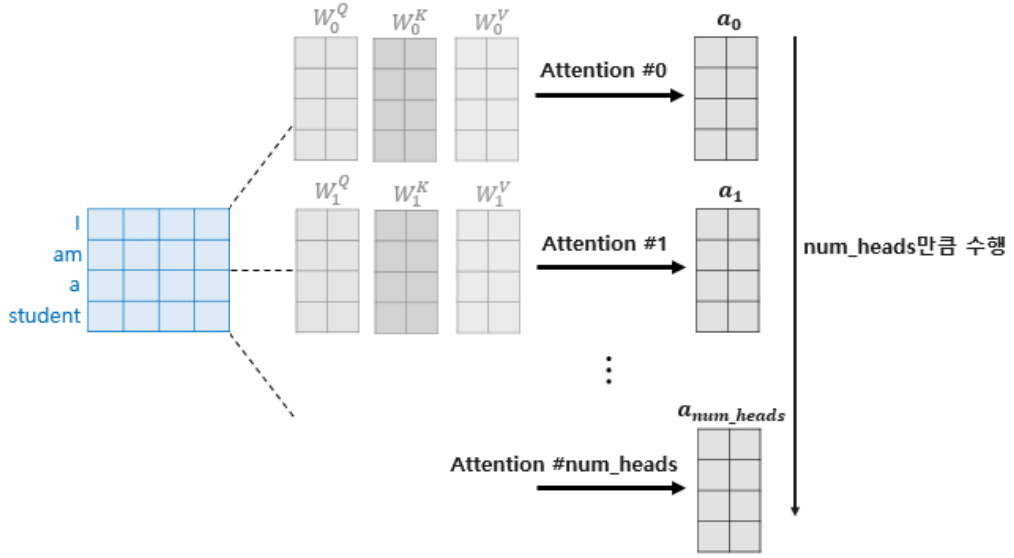
```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads, name="multi_head_attention"):
    super(MultiHeadAttention, self).__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    # d_model을 num_heads로 나눈 값.
    # 논문 기준 : 64
    self.depth = d_model // self.num_heads

    # WQ, WK, WV에 해당하는 밀집층 정의
    self.query_dense = tf.keras.layers.Dense(units=d_model)
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    # WO에 해당하는 밀집층 정의
    self.dense = tf.keras.layers.Dense(units=d_model)

  # num_heads 개수만큼 q, k, v를 split하는 함수
  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

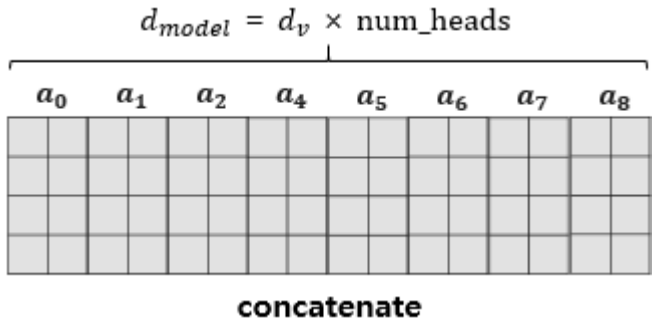
Query, key, value를 헤드 수만큼 split
논문기준: 512차원 임베딩 벡터를 8개의 헤드수로 나눔
→ 한 헤드당 64차원

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'], inputs[
      'value'], inputs['mask']
  batch_size = tf.shape(query)[0]

  # 1. WQ, WK, WV에 해당하는 밀집층 지나기
  # q : (batch_size, query의 문장 길이, d_model)
  # k : (batch_size, key의 문장 길이, d_model)
  # v : (batch_size, value의 문장 길이, d_model)
  # 참고) 인코더(k, v)-디코더(q) 어텐션에서는 query 길이와 key, value의 길이는 다를 수 있다.
  query = self.query_dense(query)
  key = self.key_dense(key)
  value = self.value_dense(value)

  # 2. 헤드 나누기
  # q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # k : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # v : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  query = self.split_heads(query, batch_size)
  key = self.split_heads(key, batch_size)
  value = self.split_heads(value, batch_size)

  # 3. 스케일드 닷 프로덕트 어텐션. 앞서 구현한 함수 사용.
  # (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  scaled_attention, _ = scaled_dot_product_attention(query, key, value, mask)
  # (batch_size, query의 문장 길이, num_heads, d_model/num_heads)
  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

  # 4. 헤드 연결(concatenate)하기
  # (batch_size, query의 문장 길이, d_model)
  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))

  # 5. WO에 해당하는 밀집층 지나기
  # (batch_size, query의 문장 길이, d_model)
  outputs = self.dense(concat_attention)

  return outputs
```

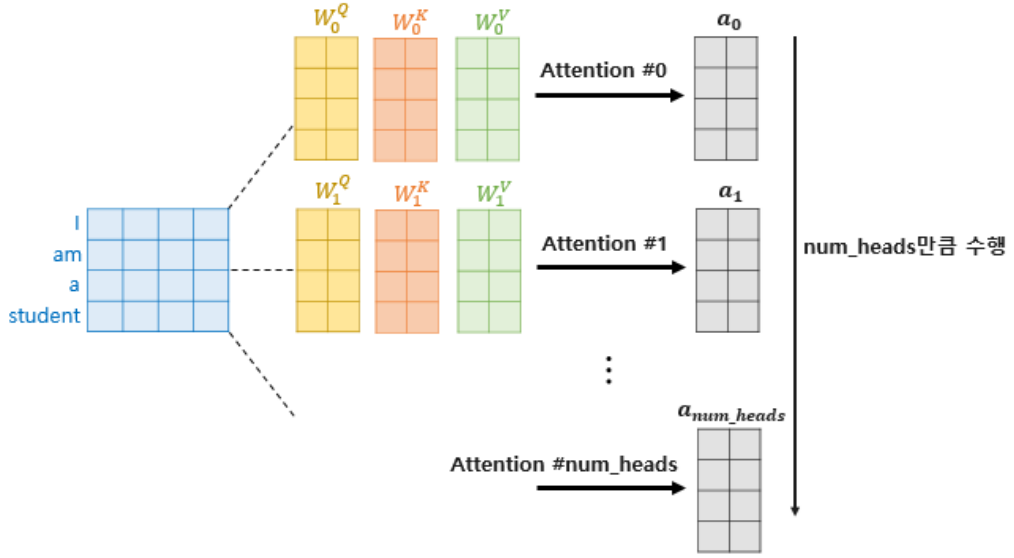


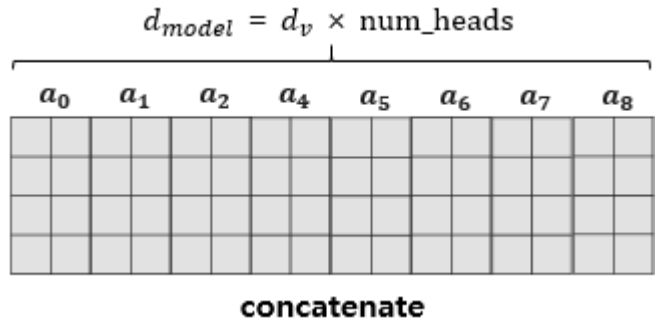


Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'], inputs[
      'value'], inputs['mask']
  batch_size = tf.shape(query)[0]

  # 1. WQ, WK, WV에 해당하는 밀집층 지나기
  # q : (batch_size, query의 문장 길이, d_model)
  # k : (batch_size, key의 문장 길이, d_model)
  # v : (batch_size, value의 문장 길이, d_model)
  # 참고) 인코더(k, v)-디코더(q) 어텐션에서는 query 길이와 key, value의 길이는 다를 수 있다.
  query = self.query_dense(query)
  key = self.key_dense(key)
  value = self.value_dense(value)

  # 2. 헤드 나누기
  # q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # k : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # v : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  query = self.split_heads(query, batch_size)
  key = self.split_heads(key, batch_size)
  value = self.split_heads(value, batch_size)

  # 3. 스케일드 닷 프로덕트 어텐션. 앞서 구현한 함수 사용.
  # (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  scaled_attention, _ = scaled_dot_product_attention(query, key, value, mask)
  # (batch_size, query의 문장 길이, num_heads, d_model/num_heads)
  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

  # 4. 헤드 연결(concatenate)하기
  # (batch_size, query의 문장 길이, d_model)
  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))

  # 5. WO에 해당하는 밀집층 지나기
  # (batch_size, query의 문장 길이, d_model)
  outputs = self.dense(concat_attention)

  return outputs
```

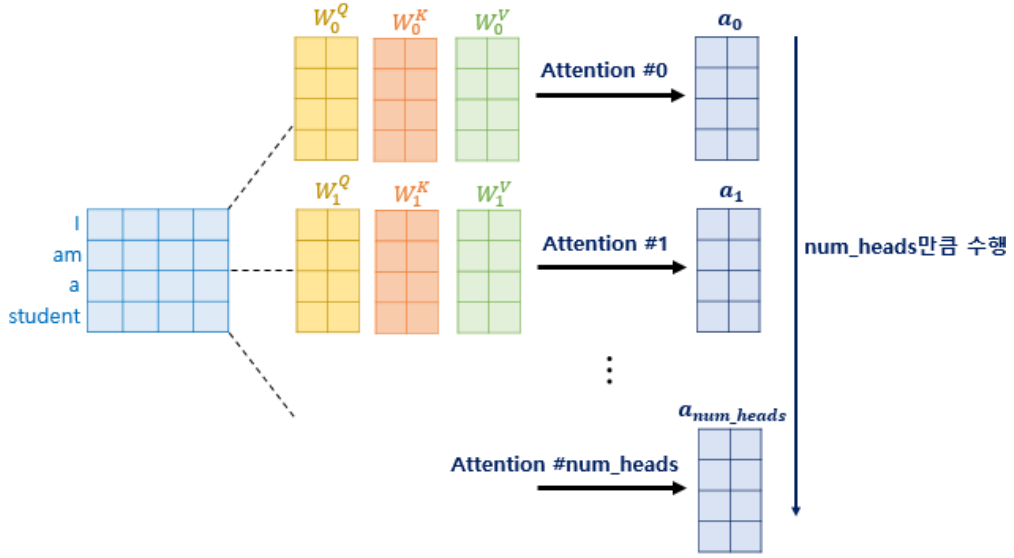


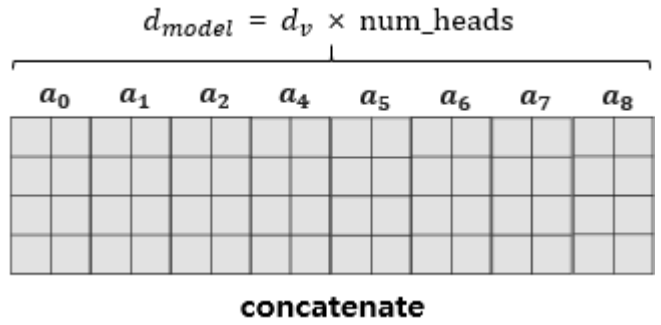


Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'], inputs[
      'value'], inputs['mask']
  batch_size = tf.shape(query)[0]

  # 1. WQ, WK, WV에 해당하는 밀집층 지나기
  # q : (batch_size, query의 문장 길이, d_model)
  # k : (batch_size, key의 문장 길이, d_model)
  # v : (batch_size, value의 문장 길이, d_model)
  # 참고) 인코더(k, v)-디코더(q) 어텐션에서는 query 길이와 key, value의 길이는 다를 수 있다.
  query = self.query_dense(query)
  key = self.key_dense(key)
  value = self.value_dense(value)

  # 2. 헤드 나누기
  # q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # k : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # v : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  query = self.split_heads(query, batch_size)
  key = self.split_heads(key, batch_size)
  value = self.split_heads(value, batch_size)

  # 3. 스케일드 닷 프로덕트 어텐션. 앞서 구현한 함수 사용.
  # (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  scaled_attention, _ = scaled_dot_product_attention(query, key, value, mask)
  # (batch_size, query의 문장 길이, num_heads, d_model/num_heads)
  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

  # 4. 헤드 연결(concatenate)하기
  # (batch_size, query의 문장 길이, d_model)
  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))

  # 5. WO에 해당하는 밀집층 지나기
  # (batch_size, query의 문장 길이, d_model)
  outputs = self.dense(concat_attention)

  return outputs
```

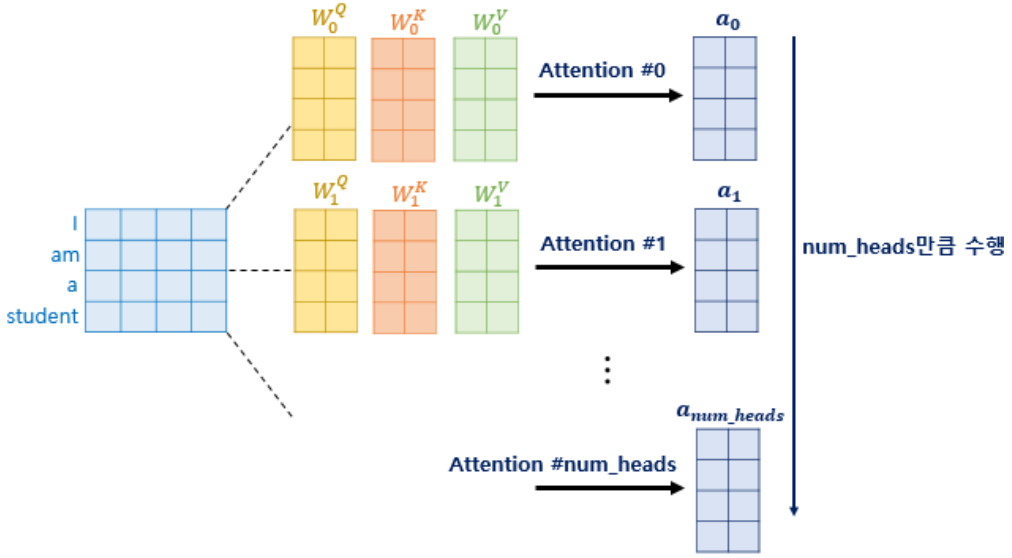


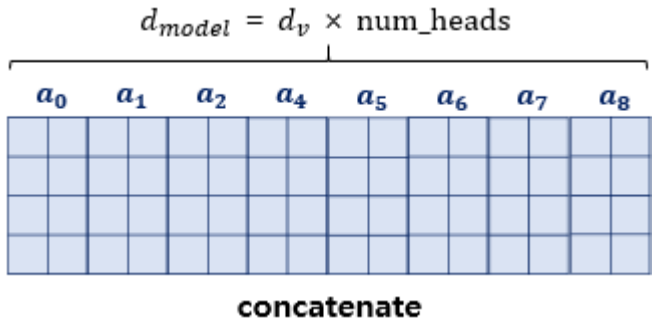


Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'], inputs[
      'value'], inputs['mask']
  batch_size = tf.shape(query)[0]

  # 1. WQ, WK, WV에 해당하는 밀집층 지나기
  # q : (batch_size, query의 문장 길이, d_model)
  # k : (batch_size, key의 문장 길이, d_model)
  # v : (batch_size, value의 문장 길이, d_model)
  # 참고) 인코더(k, v)-디코더(q) 어텐션에서는 query 길이와 key, value의 길이는 다를 수 있다.
  query = self.query_dense(query)
  key = self.key_dense(key)
  value = self.value_dense(value)

  # 2. 헤드 나누기
  # q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # k : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # v : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  query = self.split_heads(query, batch_size)
  key = self.split_heads(key, batch_size)
  value = self.split_heads(value, batch_size)

  # 3. 스케일드 닷 프로덕트 어텐션. 앞서 구현한 함수 사용.
  # (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  scaled_attention, _ = scaled_dot_product_attention(query, key, value, mask)
  # (batch_size, query의 문장 길이, num_heads, d_model/num_heads)
  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

  # 4. 헤드 연결(concatenate)하기
  # (batch_size, query의 문장 길이, d_model)
  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))

  # 5. WO에 해당하는 밀집층 지나기
  # (batch_size, query의 문장 길이, d_model)
  outputs = self.dense(concat_attention)

  return outputs
```

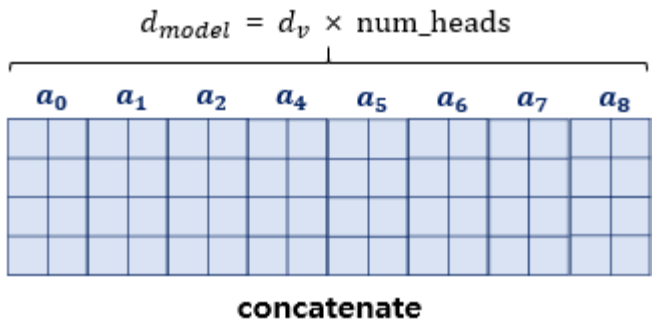


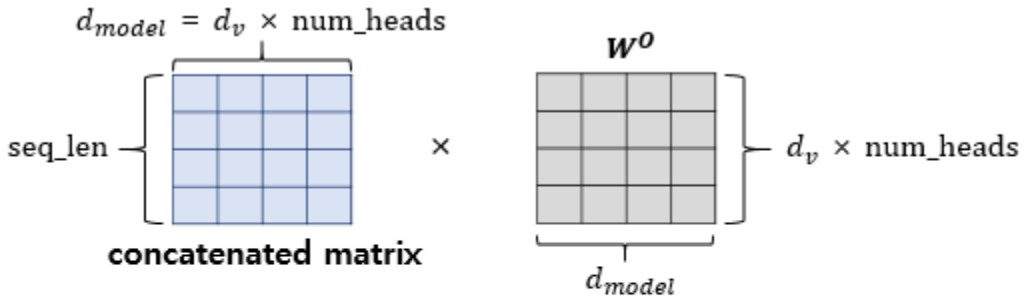


Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

2. Multi-head attention

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'], inputs[
      'value'], inputs['mask']
  batch_size = tf.shape(query)[0]

  # 1. WQ, WK, WV에 해당하는 밀집층 지나기
  # q : (batch_size, query의 문장 길이, d_model)
  # k : (batch_size, key의 문장 길이, d_model)
  # v : (batch_size, value의 문장 길이, d_model)
  # 참고) 인코더(k, v)-디코더(q) 어텐션에서는 query 길이와 key, value의 길이는 다를 수 있다.
  query = self.query_dense(query)
  key = self.key_dense(key)
  value = self.value_dense(value)

  # 2. 헤드 나누기
  # q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  # k : (batch_size, num_heads, key의 문장 길이, d_model/num_heads)
  # v : (batch_size, num_heads, value의 문장 길이, d_model/num_heads)
  query = self.split_heads(query, batch_size)
  key = self.split_heads(key, batch_size)
  value = self.split_heads(value, batch_size)

  # 3. 스케일드 닷 프로덕트 어텐션. 앞서 구현한 함수 사용.
  # (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
  scaled_attention, _ = scaled_dot_product_attention(query, key, value, mask)
  # (batch_size, query의 문장 길이, num_heads, d_model/num_heads)
  scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

  # 4. 헤드 연결(concatenate)하기
  # (batch_size, query의 문장 길이, d_model)
  concat_attention = tf.reshape(scaled_attention,
                                (batch_size, -1, self.d_model))

  # 5. WO에 해당하는 밀집층 지나기
  # (batch_size, query의 문장 길이, d_model)
  outputs = self.dense(concat_attention)

  return outputs
```





Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Encoder

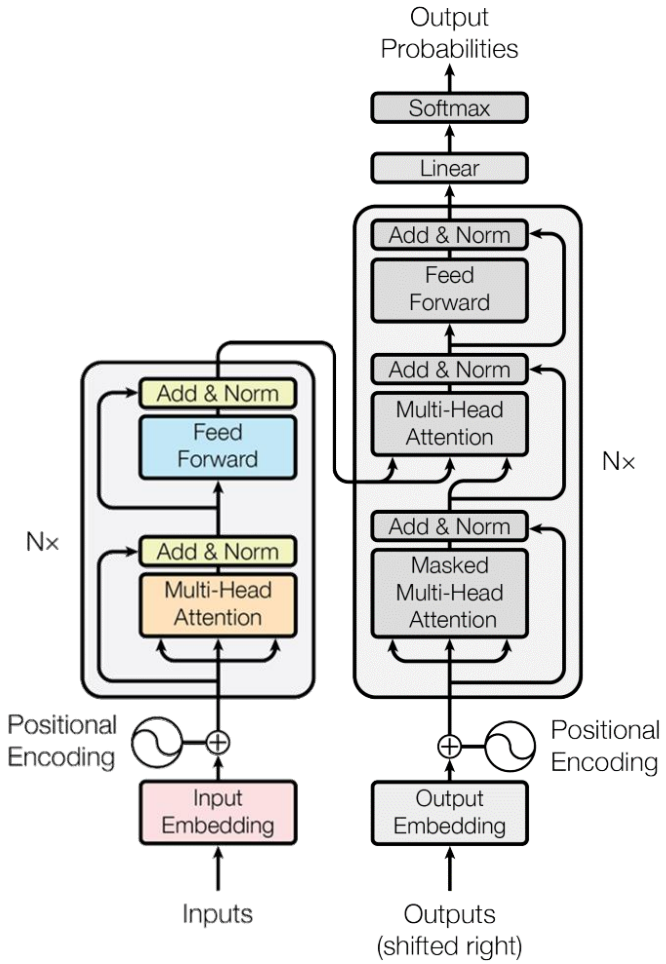


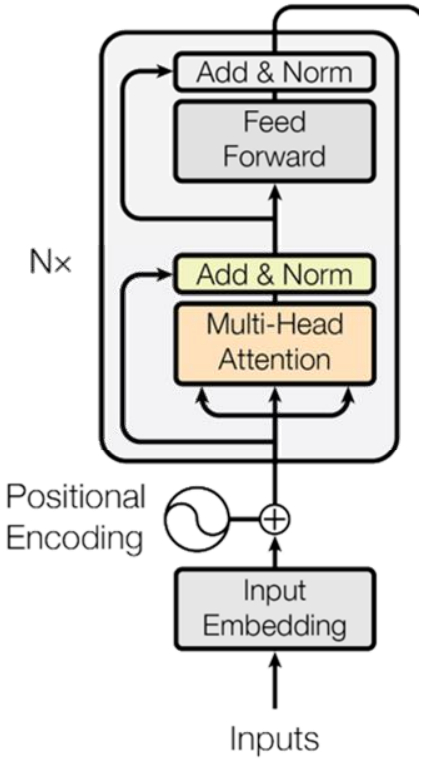
Figure 1: The Transformer - model architecture.

1. 입력: q, k, v
2. 출력: attention mechanism이 적용된 q, k, v
3. 동작
    1. Encoder self-attention
    2. Residual connection, normalization
    3. FFNN
    4. Residual connection, normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Encoder

```
def encoder_layer(dff, d_model, num_heads, dropout, name="encoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model), name="inputs")

  # 인코더는 패딩 마스크 사용
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  # 멀티-헤드 어텐션 (첫번째 서브층 / 셀프 어텐션)
  attention = MultiHeadAttention(
      d_model, num_heads, name="attention")({
          'query': inputs, 'key': inputs, 'value': inputs, # Q = K = V
          'mask': padding_mask # 패딩 마스크 사용
      })

  # 드롭아웃 + 잔차 연결과 층 정규화
  attention = tf.keras.layers.Dropout(rate=dropout)(attention)
  attention = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(inputs + attention)

  # 포지션 와이즈 피드 포워드 신경망 (두번째 서브층)
  outputs = tf.keras.layers.Dense(units=dff, activation='relu')(attention)
  outputs = tf.keras.layers.Dense(units=d_model)(outputs)

  # 드롭아웃 + 잔차 연결과 층 정규화
  outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
  outputs = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(attention + outputs)

  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
```



input: q, k, v

padding mask: 가리고 싶은 단어를 가리는 용도로 <pad> 토큰 단어 위치의 임베딩 벡터 값을 아주 작은 음수로 설정

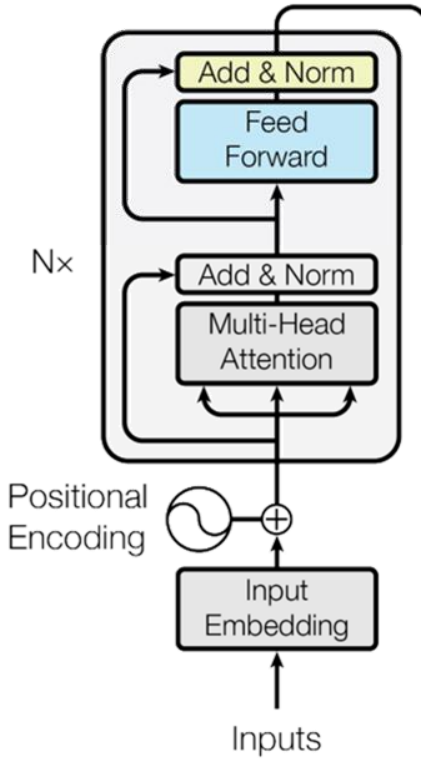
attention: q, k, v를 넣어 multi-head attention 진행
→ 인코더: encoder self-attention

drop out 설정
residual connection, layer normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Encoder

```
def encoder_layer(dff, d_model, num_heads, dropout, name="encoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model), name="inputs")

  # 인코더는 패딩 마스크 사용
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  # 멀티-헤드 어텐션 (첫번째 서브층 / 셀프 어텐션)
  attention = MultiHeadAttention(
      d_model, num_heads, name="attention")({
          'query': inputs, 'key': inputs, 'value': inputs, # Q = K = V
          'mask': padding_mask # 패딩 마스크 사용
      })

  # 드롭아웃 + 잔차 연결과 층 정규화
  attention = tf.keras.layers.Dropout(rate=dropout)(attention)
  attention = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(inputs + attention)

  # 포지션 와이즈 피드 포워드 신경망 (두번째 서브층)
  outputs = tf.keras.layers.Dense(units=dff, activation='relu')(attention)
  outputs = tf.keras.layers.Dense(units=d_model)(outputs)

  # 드롭아웃 + 잔차 연결과 층 정규화
  outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
  outputs = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(attention + outputs)

  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
```



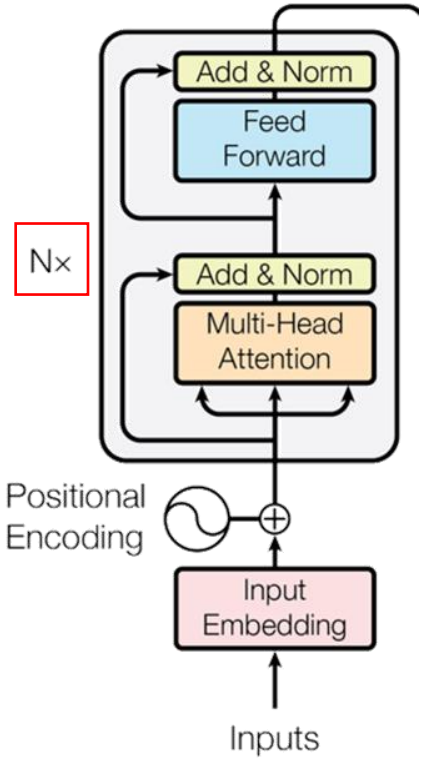
FFNN: Dense
w1: (d_model, dff)
we: (dff, d_model)

drop out 설정
residual connection, layer normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Encoder

```
def encoder(vocab_size, num_layers, dff,
            d_model, num_heads, dropout,
            name="encoder"):
  inputs = tf.keras.Input(shape=(None,), name="inputs")

  # 인코더는 패딩 마스크 사용
  padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")

  # 포지셔널 인코딩 + 드롭아웃
  embeddings = tf.keras.layers.Embedding(vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))
  embeddings = PositionalEncoding(vocab_size, d_model)(embeddings)
  outputs = tf.keras.layers.Dropout(rate=dropout)(embeddings)

  # 인코더를 num_layers개 쌓기
  for i in range(num_layers):
    outputs = encoder_layer(dff=dff, d_model=d_model, num_heads=num_heads,
        dropout=dropout, name="encoder_layer_{}".format(i),
    )([outputs, padding_mask])

  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs, name=name)
```



inputs: q, k, v

padding_mask: 가릴 곳을 가려라

positional encoding

encoder 여러 개 쌓기
이전 인코더 레이어의 output이 곧 다음 인코더 레이어의 입력

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Decoder

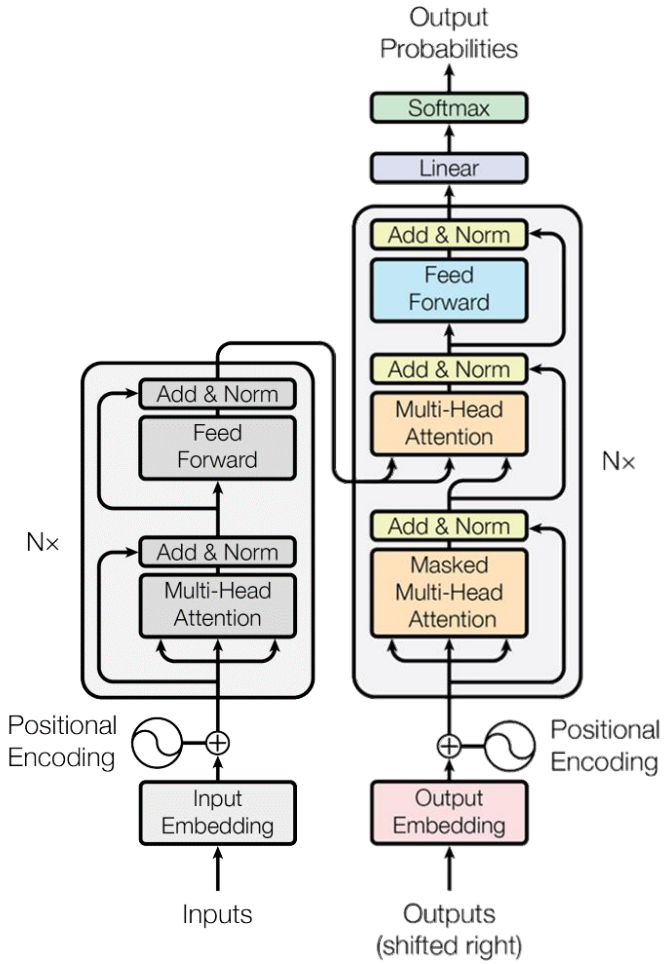


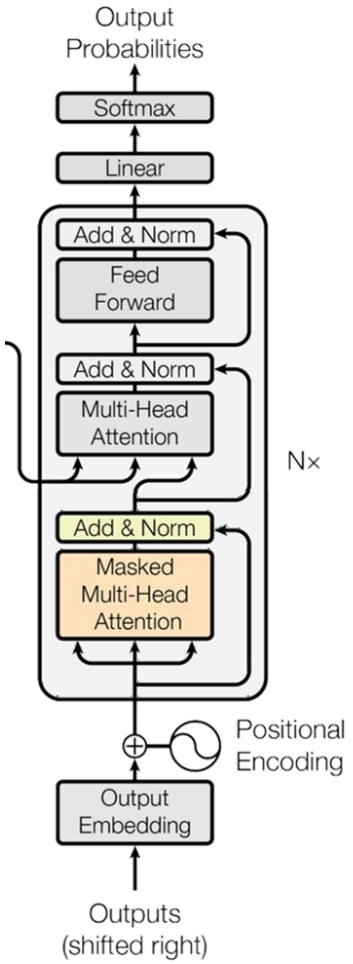
Figure 1: The Transformer - model architecture.

1. 입력: 인코더에서 추출한 k, v / 디코더에서 추출한 q, k, v
2. 출력: 단어의 확률
3. 동작
    1. Masked self-attention
    2. Residual connection, normalization
    3. Encoder-Decoder attention
    4. Residual connection, normalization
    5. FFNN
    6. Residual connection, normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Decoder

```
def decoder_layer(dff, d_model, num_heads, dropout, name="decoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model), name="inputs")
  enc_outputs = tf.keras.Input(shape=(None, d_model), name="encoder_outputs")

  # 디코더는 룩어헤드 마스크(첫번째 서브층)와 패딩 마스크(두번째 서브층) 둘 다 사용.
  look_ahead_mask = tf.keras.Input(
      shape=(1, None, None), name="look_ahead_mask")
  padding_mask = tf.keras.Input(shape=(1, 1, None), name='padding_mask')

  # 멀티-헤드 어텐션 (첫번째 서브층 / 마스크드 셀프 어텐션)
  attention1 = MultiHeadAttention(
      d_model, num_heads, name="attention_1")(inputs={
          'query': inputs, 'key': inputs, 'value': inputs, # Q = K = V
          'mask': look_ahead_mask # 룩어헤드 마스크
      })

  # 잔차 연결과 층 정규화
  attention1 = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(attention1 + inputs)
```



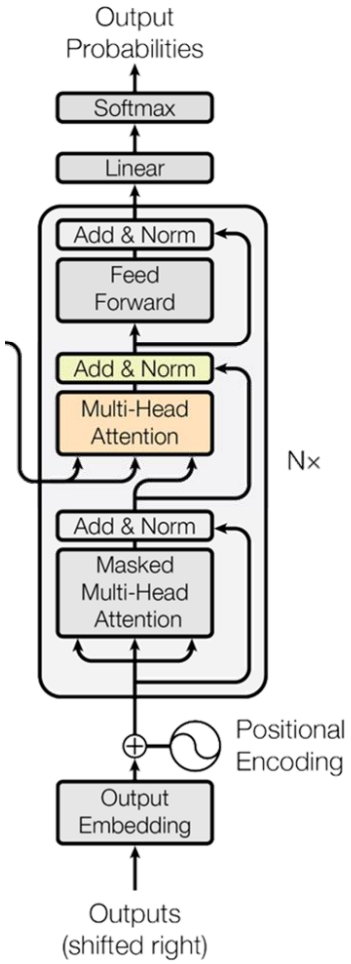
1. Input
    1. decoder q, k, v
    2. encoder output
2. look-ahead-mask
    1. 현재 위치보다 뒤의 있는 단어를 참고하지 못하도록 look-ahead mask를 적용
3. Masked self-attention
    1. 입력: 디코더 q, k, v

4. Residual connection, layer normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Decoder

```
# 멀티-헤드 어텐션 (두번째 서브층 / 디코더-인코더 어텐션)
attention2 = MultiHeadAttention(
    d_model, num_heads, name="attention_2")(inputs={
        'query': attention1, 'key': enc_outputs, 'value': enc_outputs, # Q != K = V
        'mask': padding_mask # 패딩 마스크
    })

# 드롭아웃 + 잔차 연결과 층 정규화
attention2 = tf.keras.layers.Dropout(rate=dropout)(attention2)
attention2 = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(attention2 + attention1)
```

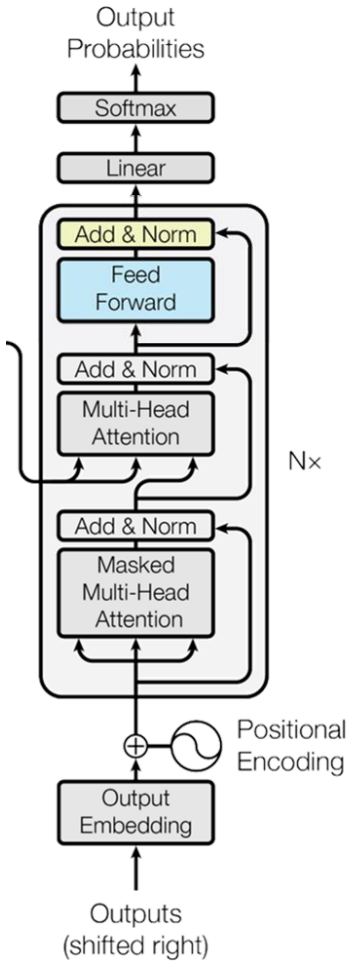


1. encoder-decoder attention
    1. 입력:
       decoder q, encoder outputs k, v
2. Drop out
3. Residual connection, layer normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Decoder

```
# 포지션 와이즈 피드 포워드 신경망 (세번째 서브층)
outputs = tf.keras.layers.Dense(units=dff, activation='relu')(attention2)
outputs = tf.keras.layers.Dense(units=d_model)(outputs)

# 드롭아웃 + 잔차 연결과 층 정규화
outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
outputs = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(outputs + attention2)

return tf.keras.Model(
    inputs=[inputs, enc_outputs, look_ahead_mask, padding_mask],
    outputs=outputs,
    name=name)
```

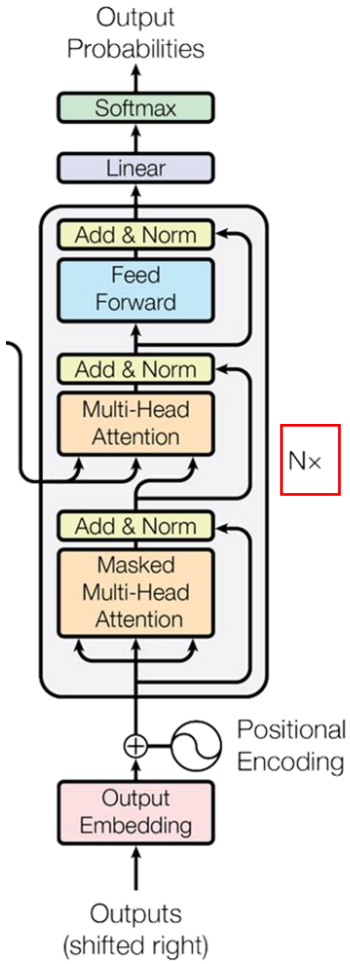


FFNN: Dense
w1: (d_model, dff)
we: (dff, d_model)
drop out 설정
residual connection, layer normalization

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

3. Decoder

```
def decoder(vocab_size, num_layers, dff,
            d_model, num_heads, dropout,
            name='decoder'):
  inputs = tf.keras.Input(shape=(None,), name='inputs')
  enc_outputs = tf.keras.Input(shape=(None, d_model), name='encoder_outputs')

  # 디코더는 룩어헤드 마스크(첫번째 서브층)와 패딩 마스크(두번째 서브층) 둘 다 사용
  look_ahead_mask = tf.keras.Input(
      shape=(1, None, None), name='look_ahead_mask')
  padding_mask = tf.keras.Input(shape=(1, 1, None), name='padding_mask')

  # 포지셔널 인코딩 + 드롭아웃
  embeddings = tf.keras.layers.Embedding(vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast(d_model, tf.float32))
  embeddings = PositionalEncoding(vocab_size, d_model)(embeddings)
  outputs = tf.keras.layers.Dropout(rate=dropout)(embeddings)

  # 디코더를 num_layers개 쌓기
  for i in range(num_layers):
    outputs = decoder_layer(dff=dff, d_model=d_model, num_heads=num_heads,
        dropout=dropout, name='decoder_layer_{}'.format(i),
    )(inputs=[outputs, enc_outputs, look_ahead_mask, padding_mask])
  print(outputs)
  return tf.keras.Model(
      inputs=[inputs, enc_outputs, look_ahead_mask, padding_mask],
      outputs=outputs,
      name=name)
```



inputs: decoder q, k, v/encoder outputs k, v

mask self-attention을 위한 look-ahead mask

positional encoding

decoder 여러 개 쌓기
이전 디코더 레이어의 output이 곧 다음 디코더 레이어의 입력

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

4. Transformer

```
def transformer(vocab_size, num_layers, dff,
                d_model, num_heads, dropout,
                name="transformer"):

  # 인코더의 입력
  inputs = tf.keras.Input(shape=(None,), name="inputs")

  # 디코더의 입력
  dec_inputs = tf.keras.Input(shape=(None,), name="dec_inputs")

  # 인코더의 패딩 마스크
  enc_padding_mask = tf.keras.layers.Lambda(
      create_padding_mask, output_shape=(1, 1, None),
      name='enc_padding_mask')(inputs)

  # 디코더의 룩어헤드 마스크(첫번째 서브층)
  look_ahead_mask = tf.keras.layers.Lambda(
      create_look_ahead_mask, output_shape=(1, None, None),
      name='look_ahead_mask')(dec_inputs)

  # 디코더의 패딩 마스크(두번째 서브층)
  dec_padding_mask = tf.keras.layers.Lambda(
      create_padding_mask, output_shape=(1, 1, None),
      name='dec_padding_mask')(inputs)
```

encoder 입력

decoder 입력

encoder self-attention에서 사용할 패딩 마스크

decoder mask self-attention에서 사용할 look ahead mask

decoder encoder-decoder attention에서 사용할 패딩 마스크

Part 3 >> 코드로 살펴보는 트랜스포머 동작 원리

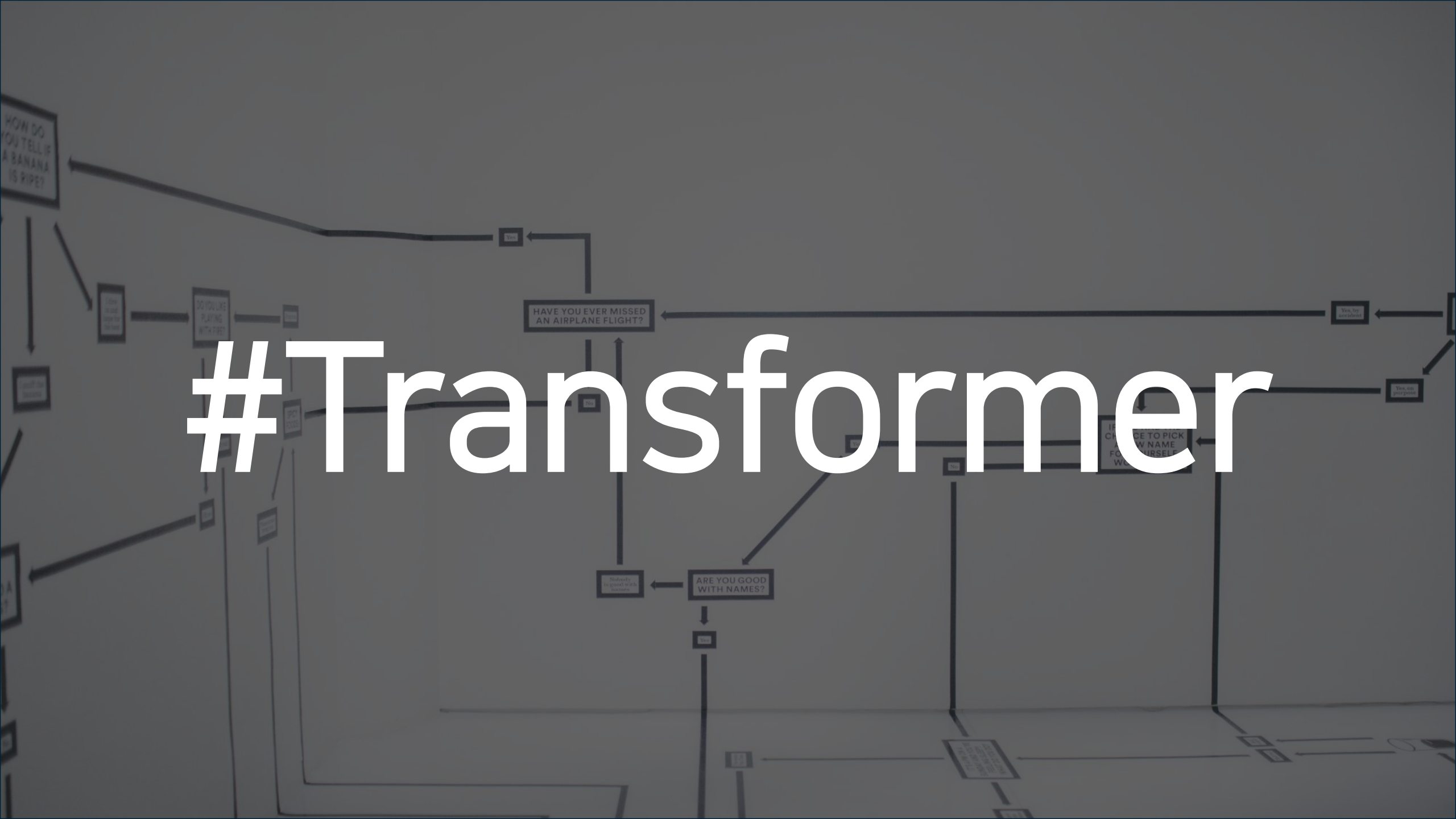
4. Transformer

```
# 인코더의 출력은 enc_outputs. 디코더로 전달된다.
enc_outputs = encoder(vocab_size=vocab_size, num_layers=num_layers, dff=dff,
    d_model=d_model, num_heads=num_heads, dropout=dropout,
)(inputs=[inputs, enc_padding_mask]) # 인코더의 입력은 입력 문장과 패딩 마스크

# 디코더의 출력은 dec_outputs. 출력층으로 전달된다.
dec_outputs = decoder(vocab_size=vocab_size, num_layers=num_layers, dff=dff,
    d_model=d_model, num_heads=num_heads, dropout=dropout,
)(inputs=[dec_inputs, enc_outputs, look_ahead_mask, dec_padding_mask])

# 다음 단어 예측을 위한 출력층
outputs = tf.keras.layers.Dense(units=vocab_size, name="outputs")(dec_outputs)
print(dec_outputs, outputs)

return tf.keras.Model(inputs=[inputs, dec_inputs], outputs=outputs, name=name)
```

encoder

decoder

단어 예측을 위한 출력층



#Transformer