Transformer

What is a Transformer?

트랜스포머(Transformer)는 2017년 구글이 발표한 논문인 "Attention is all you need"에서 나온 모델로 기존의 seq2seq의 구조인 인코더-디코더를 따르면서도, 논문의 이름처럼 어텐션(Attention)만으로 구현한 모델입니다. 이 모델은 RNN을 사용하지 않고, 인코더-디코더 구조를 설계하였음에도 성능도 RNN보다 우수하다는 특징을 갖고있습니다.

기존의 seq2seq 모델의 한계

기존의 seq2seq 모델은 인코더-디코더 구조로 구성되어져 있었습니다. 여기서 인코더는 입력 시퀀스를 하나의 벡터 표현으로 압축하고, 디코더는 이 벡터 표현을 통해서 출력 시퀀스를 만들어냈습니다. 하지만 이러한 구조는 인코더가 입력 시퀀스를 하나의 벡터로 압축하는 과정에서 입력 시퀀스의 정보가 일부 손실된다는 단점이 있었고, 이를 보정하기 위해 어텐션이 사용되었습니다.
그런데 어텐션을 RNN의 보정을 위한 용도가 아니라 아예 어텐션으로 인코더와 디코더를 만들어보면 어떨까요?

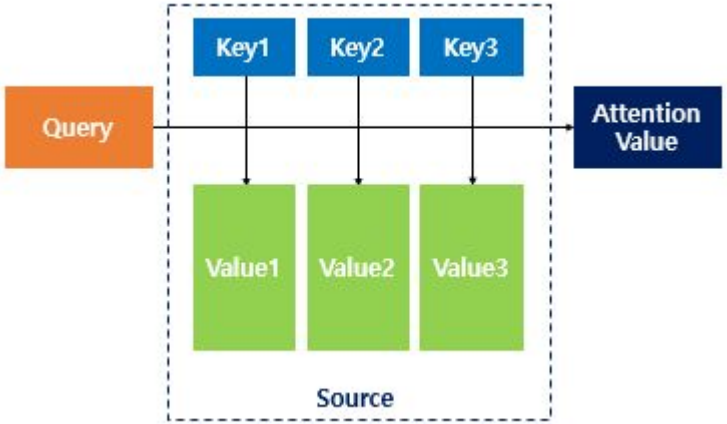
문제점 정리

1. 하나의 고정된 크기의 벡터에 모든 정보를 압축하려고 하니까 정보 손실이 발생한다.
2. RNN의 고질적인 문제인 기울기 소실(Vanishing Gradient) 문제가 존재한다.

-> Transformer의 등장

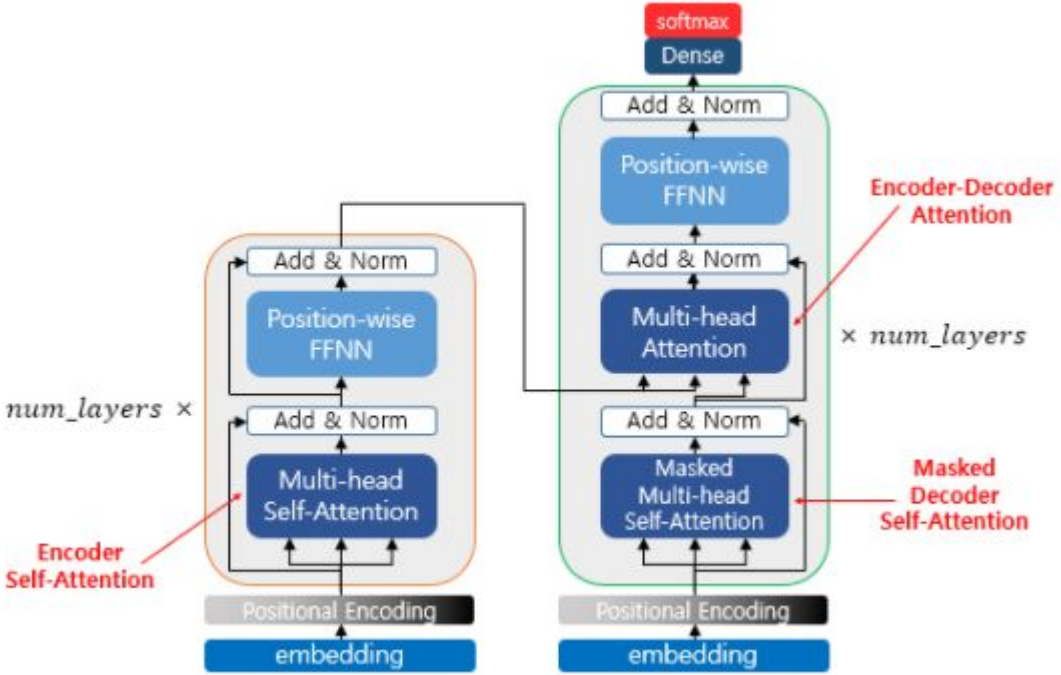
What is an Attention?

Idea: 디코더에서 출력 단어를 예측하는 매 시점(time step)마다, 인코더에서의 전체 입력 문장을 다시 한 번 참고한다. 단, 전체 입력 문장을 전부 다 동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야할 단어와 연관이 있는 입력 단어 부분을 좀 더 집중(attention)해서 보게 됩니다.



어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구합니다.
그리고 구해낸 이 유사도를 키와 맵핑되어있는 각각의 '값(Value)'에 반영해줍니다.
그리고 유사도가 반영된 '값(Value)'을 모두 더해서 리턴합니다.
이를 어텐션 값(Attention Value)이라고 합니다.

What is an Attention?



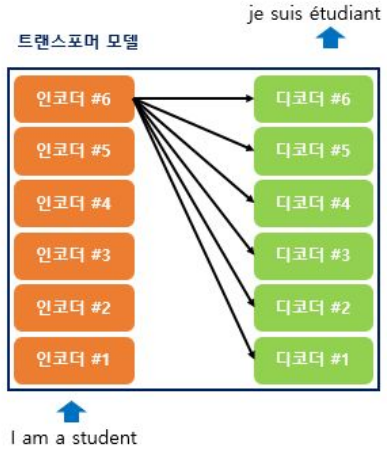
위 그림은 트랜스포머의 아키텍처에서 세 가지 어텐션이 각각 어디에서 이루어지는지를 보여줍니다.

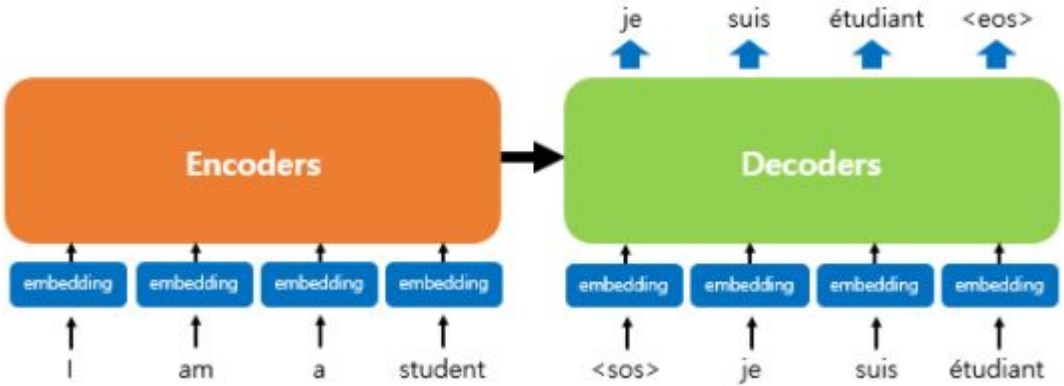
세 개의 어텐션에 추가적으로 '멀티 헤드'라는 이름이 붙어있습니다. 이는 트랜스포머가 어텐션을 병렬적으로 수행하는 방법을 의미합니다.

Transformer Structure

트랜스포머는 RNN을 사용하지 않지만 기존의 seq2seq처럼 인코더에서 입력 시퀀스를 입력받고, 디코더에서 출력 시퀀스를 출력하는 인코더-디코더 구조를 유지하고 있습니다. 다만 다른 점은 인코더와 디코더라는 단위가 N개가 존재할 수 있다는 점입니다.

이전 seq2seq 구조에서는 인코더와 디코더에서 각각 하나의 RNN이 t개의 시점(time-step)을 가지는 구조였다면 이번에는 인코더와 디코더라는 단위가 N개로 구성되는 구조입니다. 트랜스포머를 제안한 논문에서는 인코더와 디코더의 개수를 각각 6개를 사용하였습니다.

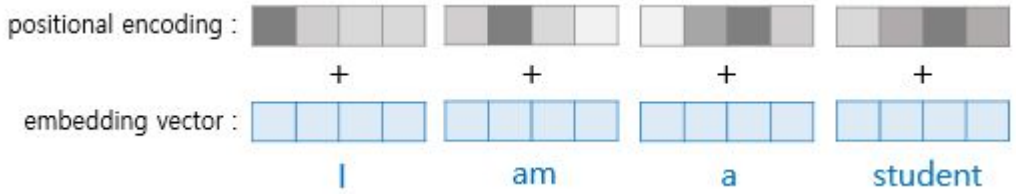




Positional Encoding

트랜스포머는 단어 입력을 순차적으로 받는 RNN과 달리, 단어의 위치 정보를 얻기 위해서 각 단어의 임베딩 벡터에 위치 정보들을 더하여 모델의 입력으로 사용하는데, 이를 포지셔널 인코딩(positional encoding) 이라고 합니다.

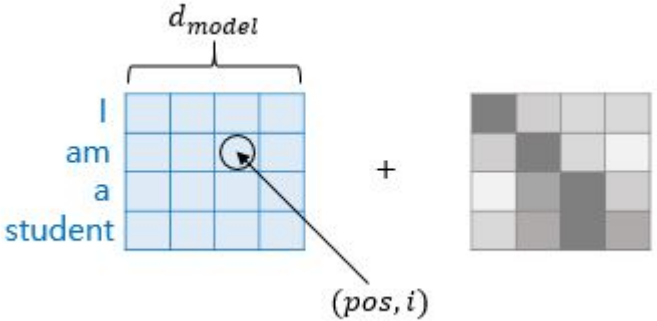
임베딩 벡터가 인코더의 입력으로 사용되기 전에 포지셔널 인코딩값이 더해지는 과정을 시각화하면 아래와 같습니다.



Implement Positional Encoding

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

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



pos는 입력 문장에서의 임베딩 벡터의 위치를 나타내며,
i는 임베딩 벡터 내의 차원의 인덱스를 의미합니다. 위의 식에 따르면 임베딩 벡터 내의 각 차원의 인덱스가 짝수인 경우에는 사인 함수의 값을 사용하고 홀수인 경우에는 코사인 함수의 값을 사용합니다.

dmodel은 트랜스포머의 모든 층의 출력 차원을 의미하는 트랜스포머의 하이퍼파라미터입니다.

그림에서는 마치 4로 표현되었지만 실제 논문에서는 512의 값을 가집니다.

같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라진다

-> 순서 정보가 보존된다.

```
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)

    # 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.shape)

    return tf.cast(pos_encoding, tf.float32)


  def call(self, inputs):

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

Encoder

트랜스포머는 하이퍼파라미터인 num_layers 개수의 인코더 층을 쌓습니다. 논문에서는 총 6개의 인코더 층을 사용하였습니다. 인코더를 하나의 층이라는 개념으로 생각한다면, 하나의 인코더 층은 크게 총 2개의 서브층(multi-head self-attention, 피드 포워드 신경망 FFNN) 으로 나뉘어집니다.

Self Attention

셀프 어텐션은 단지 어텐션을 자기 자신에게 수행한다는 의미입니다(Query, Key, Value가 같은 경우)

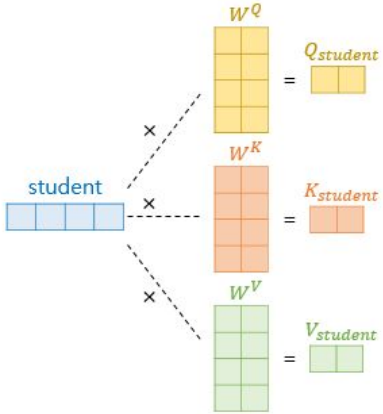
입력 문장 내의 단어들끼리 유사도를 구합니다.

셀프 어텐션은 각 단어 벡터들로부터 Q벡터, K벡터, V벡터를 얻는 작업을 거칩니다. 이때 이 Q벡터, K벡터, V벡터들은 초기 입력인

dmodel의 차원을 가지는 단어 벡터들보다 더 작은 차원을 가지는데, 논문에서는 dmodel=512의 차원을 가졌던 각 단어 벡터들을 64의 차원을 가지는 Q벡터, K벡터, V벡터로 변환하였습니다.

논문에서 사용한 num_heads은 8, 따라서 dmodel / num_heads = 64를 각 단어 벡터들의 차원으로 사용하였습니다.

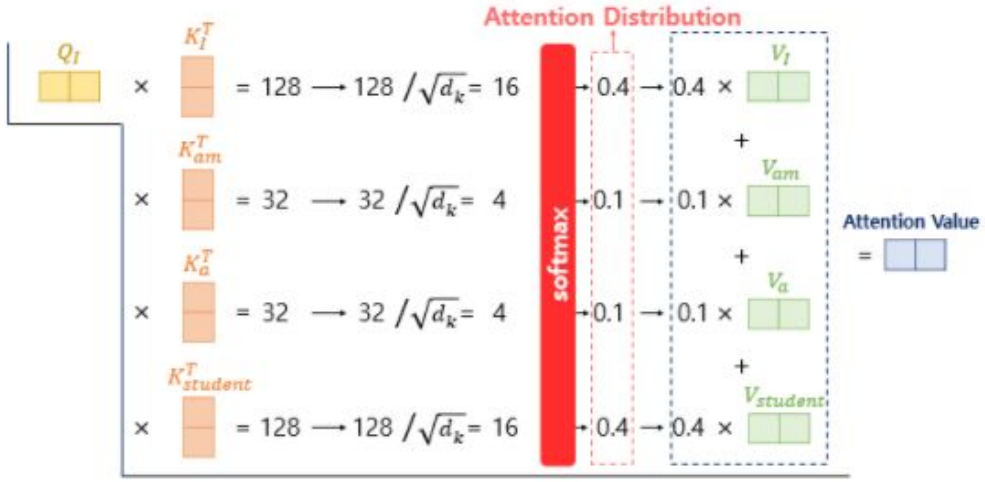
기존의 벡터로부터 더 작은 벡터는 가중치 행렬을 곱하므로서 완성됩니다.
각 가중치 행렬은 dmodel×(dmodel/num_heads)의 크기를 가집니다.



Scaled dot-product Attention

Q, K, V 벡터를 얻었다면 각 Q 벡터는 모든 K 벡터에 대해서 어텐션 스코어를 구하고, 어텐션 분포를 구한 뒤에 이를 사용하여 모든 V 벡터를 가중합하여 어텐션 값 또는 컨텍스트 벡터를 구하게 됩니다. 그리고 이를 모든 Q 벡터에 대해서 반복합니다.

여기서는 어텐션 함수로
score(q,k)=q·k/√n를 사용합니다.
여기서 n은 Q, K, V 벡터들의 차원을 의미합니다.
이러한 함수를 사용하는 어텐션을
scaled dot-product attention이라고 합니다.



어텐션 스코어에 소프트맥스 함수를 사용하여 어텐션
어텐션 값(Attention Value)을 구합니다.
-> 행렬 연산으로 일괄 처리

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

Scaled dot-product 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
```

Padding Mask

```
logits += (mask * -1e9)
```

이는 입력 문장에 <PAD> 토큰이 있을 경우 어텐션에서 사실상 제외하기 위한 연산입니다.

마스킹을 하는 방법은 어텐션 스코어 행렬의 마스킹 위치에 매우 작은 음수값을 넣어주는 것입니다. 그러면 소프트 맥스 함수를 지난 후에는 0에 굉장히 가까운 값이 되어 단어 간 유사도를 구하는 일에 <PAD>가 반영되지 않게 됩니다.

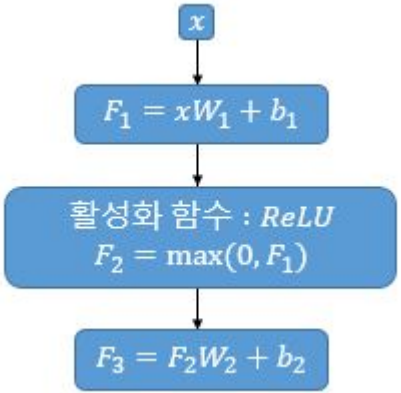
패딩 마스크를 구현하는 방법은 입력된 정수 시퀀스에서 패딩 토큰의 인덱스인지, 아닌지를 판별하는 함수를 구현하는 것입니다. 아래의 함수는 정수 시퀀스에서 0인 경우에는 1로 변환하고, 그렇지 않은 경우에는 0으로 변환하는 함수입니다.

```
def create_padding_mask(x):

  mask = tf.cast(tf.math.equal(x, 0), tf.float32)

  # (batch_size, 1, 1, key의 문장 길이)

  return mask[:, tf.newaxis, tf.newaxis, :]
```

Position-wise feed forward neural network (FFNN)

포지션 와이즈 FFNN은 인코더와 디코더에서 공통적으로 가지고 있는 서브층입니다.

x는 앞서 multi head attention의 결과로 나온 (seq_len, dmodel)의 크기를 가지는 행렬입니다.

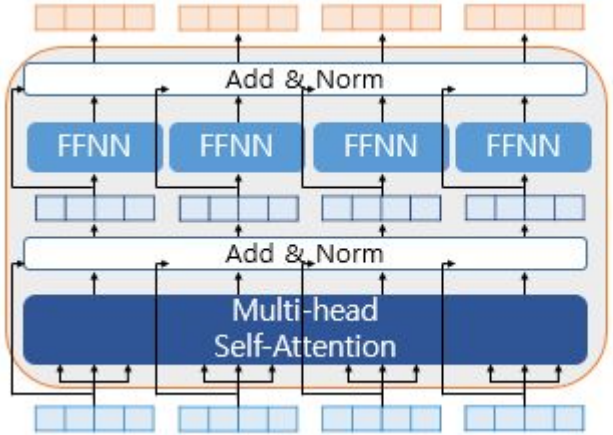


Residual connection & Layer normalization

트랜스포머에서는 이러한 두 개의 서브층(multi-head self-attention, 피드 포워드 신경망 FFNN)을 가진 인코더에 추가적으로 사용하는 기법이 있는데, 바로 Residual connection & Layer normalization 입니다

- Residual connection:
트랜스포머에서 서브층의 입력과 출력은 동일한 차원을 갖고 있으므로, 서브층의 입력과 서브층의 출력은 덧셈 연산을 할 수 있습니다.
즉, H(x) = x + Multi_head_attention(x)

- Layer Normalization:
Residual connection의 입력을 x라고 하면
LN = LayerNorm(x + Sublayer(x))



Implement an 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)
```

인코더 쌓기 (num_layers만큼)

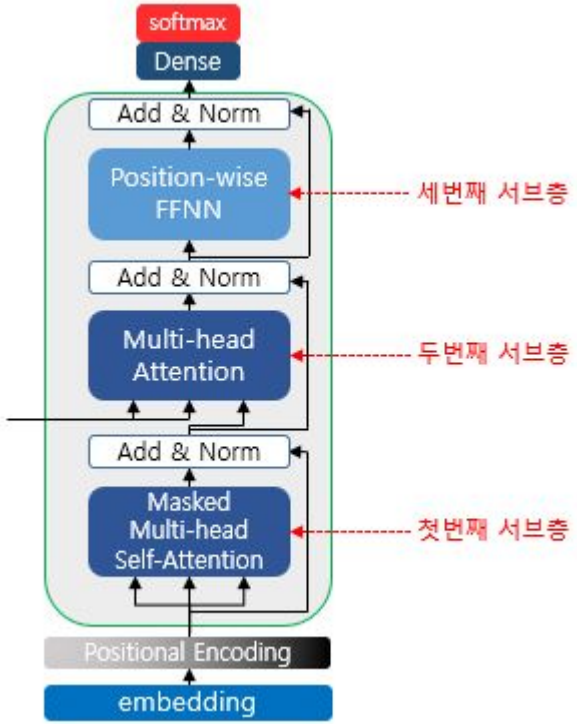
```
def encoder(vocab_size, num_layers, dff,
            d_model, num_heads, dropout,
            name="encoder"):
  inputs = tf.keras.Input(shape=(None,), name="inputs")
  # 인코더는 패딩 마스크 사용
  padding_mask = tf.keras.Input(shape=(1, 1, None),
name="padding_mask")
  # 포지셔널 인코딩 + 드롭아웃
  embeddings = tf.keras.layers.Embedding(vocab_size,
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)
```

Decoder

트랜스포머에는 총 세 가지 어텐션(인코더 1개, 디코더 2개)이 존재하며, 모두 멀티 헤드 어텐션을 수행하고, 멀티 헤드 어텐션 함수 내부에서 스케일드 닷 프로덕트 어텐션 함수를 호출하는데 각 어텐션 시 함수에 전달하는 마스킹은 다음과 같습니다.

- 인코더의 셀프 어텐션 : 패딩 마스크를 전달
- 디코더의 첫번째 서브층인 마스크드 셀프 어텐션 : 룩-어헤드 마스크를 전달
- 디코더의 두번째 서브층인 인코더-디코더 어텐션 : 패딩 마스크를 전달



Decoder 1st sub-layer

디코더의 첫번째 서브층인 멀티 헤드 셀프 어텐션 층은 인코더의 첫번째 서브층인 멀티 헤드 셀프 어텐션 층과 동일한 연산을 수행합니다. 오직, 다른 점은 어텐션 스코어 행렬에서 마스킹을 적용한다는 점만 다릅니다.

이때 룩-어헤드 마스크를 사용합니다.

룩-어헤드 마스크 (look ahead mask)

```
# 디코더의 첫번째 서브층(sublayer)에서 미래 토큰을 Mask하는 함수

def create_look_ahead_mask(x):
  seq_len = tf.shape(x)[1]
  look_ahead_mask = 1 - tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)
  padding_mask = create_padding_mask(x) # 패딩 마스크도 포함
  return tf.maximum(look_ahead_mask, padding_mask)
```

Decoder 2nd sub-layer

디코더의 두번째 서브층은 멀티 헤드 어텐션을 수행한다는 점에서는 이전의 어텐션들(인코더와 디코더의 첫번째 서브층)과 같지만, 셀프 어텐션(Q, K, V가 같은 것)은 아닙니다.

그외 나머지 멀티 헤드 어텐션을 수행하는 과정은 다른 어테션들과 같습니다.

*정리

인코더의 첫번째 서브층 : Query = Key = Value

디코더의 첫번째 서브층 : Query = Key = Value

디코더의 두번째 서브층 : Query : 디코더 행렬 / Key = Value : 인코더 행렬(from 마지막 층)

Implement a Decoder (1)

```
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")

  # 룩어헤드 마스크(첫번째 서브층)

  look_ahead_mask = tf.keras.Input(

      shape=(1, None, None), name="look_ahead_mask")

  # 패딩 마스크(두번째 서브층)

  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)

  # 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

      })
```

Implement a Decoder (2)

```
  # 드롭아웃 + 잔차 연결과 층 정규화

    attention2 =
tf.keras.layers.Dropout(rate=dropout)(attention2)

    attention2 = tf.keras.layers.LayerNormalization(

        epsilon=1e-6)(attention2 + attention1)


    # 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)
```

디코더 쌓기 (num_layers만큼)

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

Implement a Transformer (1)

```
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)
  # 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
```

Implement a Transformer (2)

```
# 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)

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