



[5주차] Attention, Transformer

Ch 15. 어텐션(Attention)

1. seq2seq의 한계

seq2seq란?

압축에 따른 정보 손실 문제

RNN의 기울기 소실 문제

2. 어텐션(Attention)

어텐션의 기본 아이디어

어텐션 함수

3. 닷-프로덕트 어텐션(Dot-Product Attention)

개요

1. 어텐션 스코어 : Enc-Dec 은닉 상태 간 유사도

2. 어텐션 분포 : 어텐션 가중치의 모음값

3. 어텐션 값을 구하고 연결 및 변환 : 새로운 s_t 만들기

4. 바다나우 어텐션(Bahdanau Attention)

어텐션 스코어

주의할 점

Ch15-3. 양방향 LSTM과 (바다나우)어텐션 : IMDB 리뷰 감성 분류

1. 데이터 전처리

2. 바다나우 어텐션 구현

3. 양방향 LSTM + 어텐션 구현

Ch 16. 트랜스포머 (Transformer)

1. 트랜스포머?

순환 신경망 없는 인코더-디코더

주요 하이퍼파라미터

RNN 없는 시계열 데이터 처리

포지셔널 인코딩

인코더의 구조

인코더의 셀프 어텐션

스케일드-닷-프로덕트-어텐션

행렬 연산 통한 일괄 처리

스케일드-닷-프로덕트-어텐션 구현

서브층 1 : 멀티 헤드 어텐션

멀티 헤드 어텐션 구현

패딩 마스크

서브층 2 : 포지션-와이즈 피드 포워드 신경망

잔차 연결과 층 정규화

디코더의 구조

룩 어헤드 마스크

첫 번째 서브층 : 마스크드 멀티 헤드 셀프 어텐션

두 번째 서브층 : 인코더 - 디코더 멀티 헤드 어텐션

세 번째 서브층 : 포지션 와이즈 피드포워드 신경망

트랜스포머 구현

학습률 스케줄링

Ch 16-2. 트랜스포머를 이용한 일상 대화 챗봇

데이터 로드

Ch 15. 어텐션(Attention)

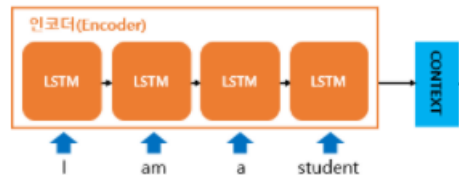
1. seq2seq의 한계

seq2seq란?

- 인코더 - 디코더라 불리는 두 개의 순환 신경망을 활용해 각각 입력 시퀀스를 벡터화하고 새로운 시퀀스를 출력하는 모델. ex) 기계번역

압축에 따른 정보 손실 문제

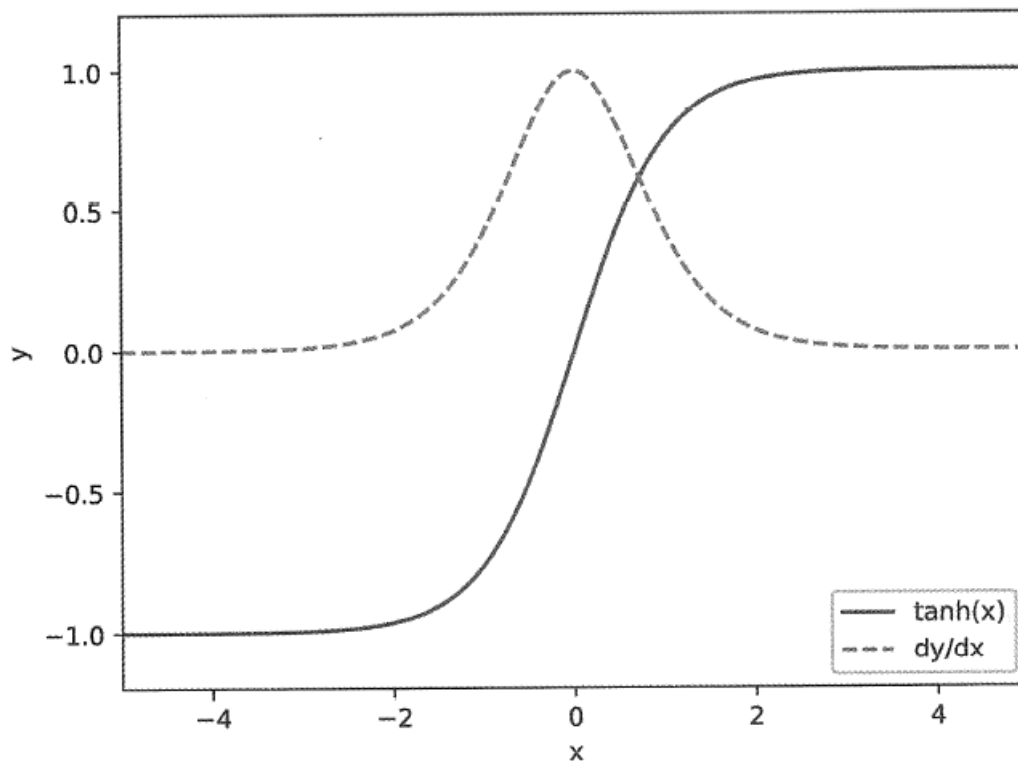
- 인코더가 문장(시퀀스)을 길이에 상관 없이 하나의 고정된 크기의 벡터(Context Vector)에 모든 정보를 압축하여 정보 손실 발생.



RNN의 기울기 소실 문제

- 기울기가 일정 수준 이하로 작아지면 장기 의존 관계 학습이 어려움.
 - ex) 하이퍼볼릭 탄젠트 함수의 미분 → x 가 0에서 멀수록 작아짐.

그림 6-6 $y = \tanh(x)$ 의 그래프(점선은 미분)





입력 시퀀스(문장) 이 길어져도 출력의 정확도를 지킬 수 있는 기법 필요 → 어텐션!

2. 어텐션(Attention)

어텐션의 기본 아이디어

- 매 시점마다 전체 입력 시퀀스를 참고하기.
→ 이 때, 필요한 부분을 집중(**attention**)해서 본다.
: 순환 신경망이 잊은 정보를 리마인드시켜주는 도우미 함수라고 생각하면 편한듯.

어텐션 함수

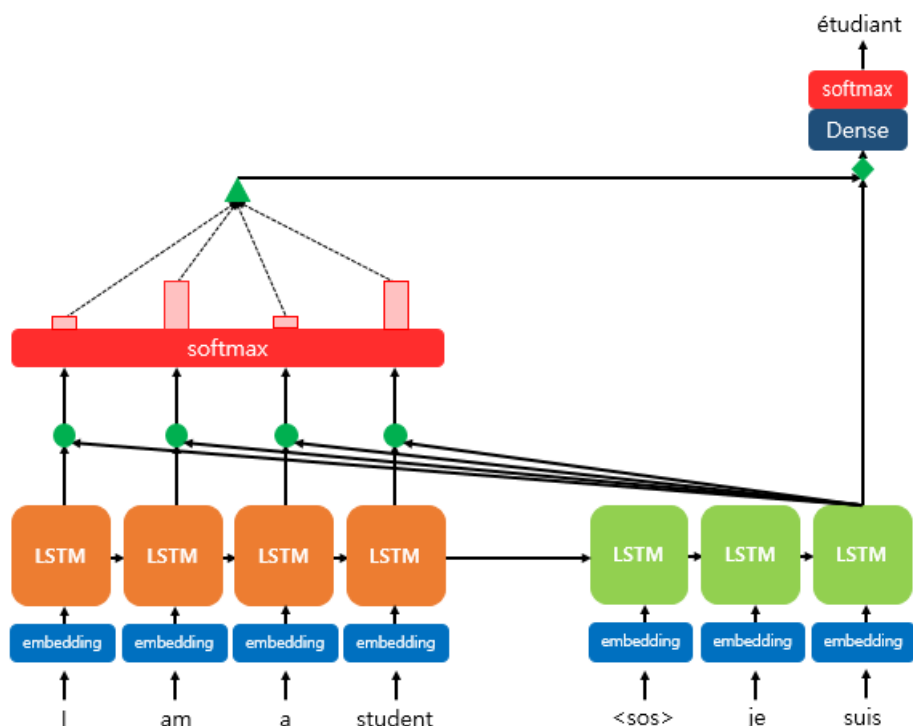
$$\text{Attention}(Q, K, V) = \text{Attention Value}$$

- 주어진 Q에 대해 모든 K의 값 V를 Q-K간 유사도만큼 가중합.
- seq2seq+ 어텐션 모델에서 Q,K,V는..
 - Q(Query) : t 시점에 디코더 셀에서의 은닉 상태
 - K(Key) : 모든 시점의 인코더 셀의 은닉 상태들
 - V(values) : 모든 시점의 인코더 셀의 은닉 상태들 (값)

3. 닷-프로덕트 어텐션(Dot-Product Attention)

개요

- 어텐션 스코어 → 어텐션 분포 → 어텐션 값 → 값 연결

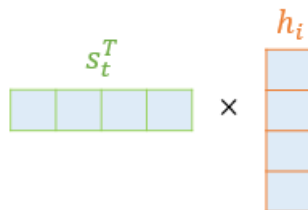


1. 어텐션 스코어 : Enc-Dec 은닉 상태 간 유사도

- 디코더의 시점 t 에서 은닉 상태 & 인코더의 시점 i 에서 은닉 상태를 곱합니다.
 - 출력으로 나오는 스칼라가 인코더-디코더의 은닉 상태 간의 유사도가 된다.
 - e_t 는 어텐션 스코어 모음 for 모든 인코더 시점.

$$\text{score}(s_t, h_i) = s_t^T h_i$$

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

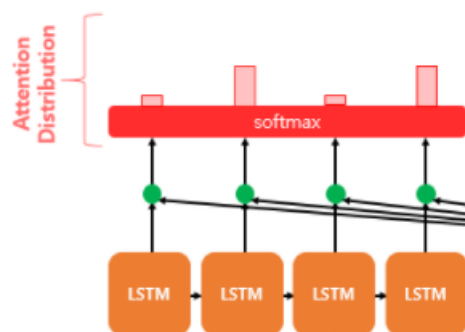


은닉 상태의 차원이
같아야 한다.

2. 어텐션 분포 : 어텐션 가중치의 모음값

- 어텐션 스코어를 softmax 함수에 통과시켜 비율로 변환
 - 각 인코더 시점 i 에서 은닉 상태가 s_t 와 얼마나 관련 있는지 비율로 나타남.
 - 특정 시점에서 인코더의 은닉 상태는 해당 시점에 입력된 단어(x_i)와 이전 은닉 상태(h_{i-1})을 입력으로 받기 때문에 해당 시점 입력 단어의 의미를 크게 담고 있다.
 - → 디코더 t 시점 단어와 인코더 i 시점 단어의 관계를 많이 담고 있다고도 말할 수 있음.

$$\alpha^t = \text{softmax}(e^t)$$

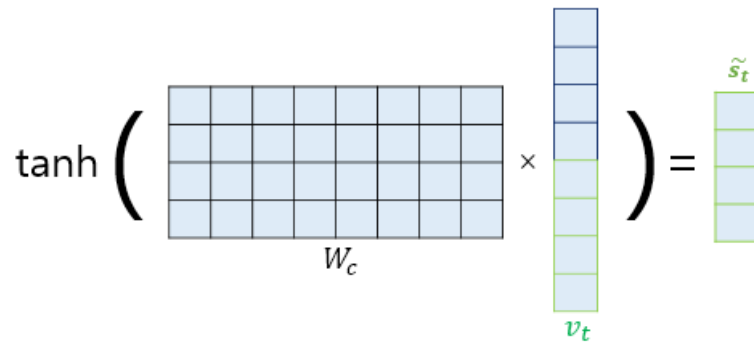


3. 어텐션 값을 구하고 연결 및 변환 : 새로운 s_t 만들기

- 만들어진 어텐션 가중치 모음 a_t 와 디코더의 기존 은닉 상태 s_t 를 연결 (concatenate)

- 출력의 형상 유지를 위해 가중치 행렬을 곱한다.(편향은 그림 생략)
- 이를 tanh함수에 통과시키면 새로운 은닉 상태 s_t 가 된다.

$$a_t = \sum_{i=1}^N \alpha_i^t h_i$$



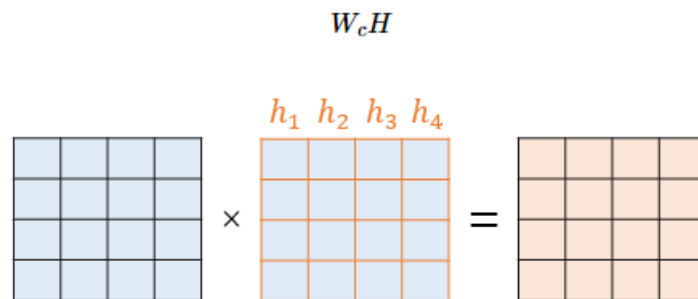
💡 어텐션 스코어를 구하는 방법에 따라 다양한 어텐션이 있다.

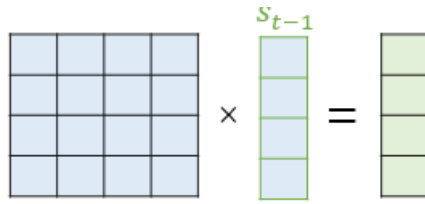
4. 바다나우 어텐션(Bahdanau Attention)

어텐션 스코어

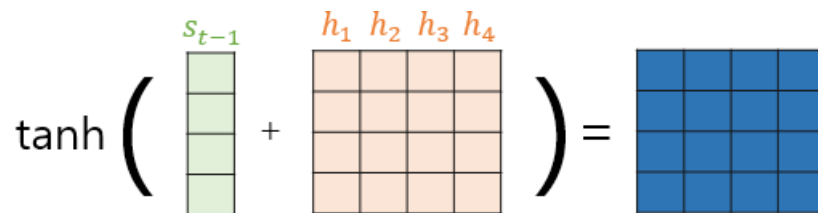
- 이전과는 다르게 s_t 가 아니라 s_{t-1} 을 사용하여 새로운 s_t 를 만드는 데에 사용한다.
- s 와 h 에 가중치 행렬을 곱한다.
 - 가중치 행렬의 크기를 조정할 수 있어 인코더와 디코더의 출력의 길이가 달라도 쓸 수 있을듯.
- 그 둘을 더하여 tanh 통과
- 이후 가중치 행렬 곱하여 어텐션 스코어 도출.

$$score(s_{t-1}, h_i) = W_a^T \tanh(W_b s_{t-1} + W_c h_i)$$

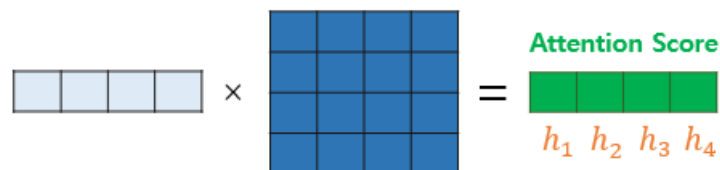




$$\tanh(W_b s_{t-1} + W_c H)$$



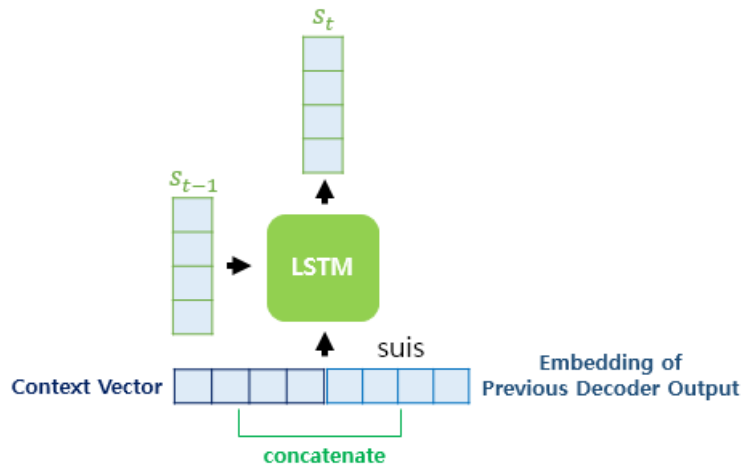
$$e^t = W_a^T \tanh(W_b s_{t-1} + W_c H)$$



→ 이후 동일한 방법으로 어텐션 값을 구한다.

주의할 점

- 닷-프로덕트 어텐션에서 어텐션 값은 디코더의 현재 시점 t에서의 은닉 상태 s_t 와 연결 후 어파인 변환(가중치 행렬 곱하고 편향 더함)을 통해 기존 s_t 를 대체했다.
- 바다나우 어텐션에서 어텐션 값은 디코더의 현재 시점 t에서의 입력 값 x_t 와 concatenate하여 LSTM의 새로운 입력으로 사용한다. s_{t-1} 과 새롭게 생긴 x_t 를 갖고 디코더는 새로운 s_t 를 출력한다.
- 이전 어텐션에서처럼 연결된 벡터를 형상 변환 안하는 이유 : LSTM 안에서 형상 변환(가중치 행렬 통과)가 이루어지기 때문



Ch15-3. 양방향 LSTM과 (바다나우)어텐션 : IMDB 리뷰 감성 분류

1. 데이터 전처리

- Import

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

- 데이터 분리, 최대 단어 개수 10000개로 제한

```
vocab_size = 10000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words = vocab_size)
```

```
print('리뷰의 최대 길이 : {}'.format(max(len(l) for l in X_train)))
print('리뷰의 평균 길이 : {}'.format(sum(map(len, X_train))/len(X_train)))
```

```
-> 리뷰의 최대 길이 : 2494
    리뷰의 평균 길이 : 238.71364
```

- 리뷰의 평균/ 최대 길이를 고려해 길이 500으로 데이터를 패딩.

```
max_len = 500
X_train = pad_sequences(X_train, maxlen=max_len)
X_test = pad_sequences(X_test, maxlen=max_len)
```

2. 바다나우 어텐션 구현

```
class BahdanauAttention(tf.keras.Model):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__() # 객체 초기화하여 부모 생성자부터 실행
```

```

# 부모 클래스인 keras.Model 사용 위함.
self.W1 = Dense(units) # Wa
self.W2 = Dense(units) # Wb
self.V = Dense(1) # Wa

def call(self, values, query):
    hidden_with_time_axis = tf.expand_dims(query, 1) # 행렬 덧셈 위한 벡터의 차원변경

    score = self.V(tf.nn.tanh(
        self.W1(values) + self.W2(hidden_with_time_axis))) # 어텐션 스코어

    attention_weights = tf.nn.softmax(score, axis=1) # 어텐션 가중치 모음

    context_vector = attention_weights * values # 어텐션 값(컨텍스트 벡터)
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights # 컨텍스트 벡터와 어텐션 가중치 모음을 출력

```

3. 양방향 LSTM + 어텐션 구현

- Import

```

from tensorflow.keras.layers import Dense, Embedding, Bidirectional, LSTM, Concatenate, Dropout
from tensorflow.keras import Input, Model
from tensorflow.keras import optimizers
import os

```

- 입력층과 임베딩 층

```

sequence_input = Input(shape=(max_len,), dtype='int32') # 문장의 길이는 500이었음.
embedded_sequences = Embedding(vocab_size, 128, input_length=max_len,
    mask_zero = True)(sequence_input) # 임베딩 벡터의 차원 128
## mask_zero는 패딩된 부분을 마스킹해 학습에 영향 주지 않도록 함.

```

- 첫 번째 양방향 LSTM 계층

```

lstm = Bidirectional(LSTM(64, dropout=0.5, return_sequences = True))(embedded_sequences)
# return_sequences=True 이면 모든 시점의 은닉 상태를 출력한다.
# 64 * 2 = 128 ( 양방향LSTM 고려한 units ) -> 순방향 역방향 출력을 concat하기 때문.

```

- 두 번째 양방향 LSTM 계층

```

lstm, forward_h, forward_c, backward_h, backward_c = Bidirectional \
    (LSTM(64, dropout=0.5, return_sequences=True, return_state=True))(lstm)
# return_state=True 이면 마지막 시점의 은닉 상태 2번 + 마지막 시점의 셀 상태 출력
# return_sequences=True를 같이 쓰면 출력은 다음과 같다.
# 전체 시점의 은닉 상태, 마지막 시점에서 순방향 LSTM의 은닉/셀상태, 역방향 은닉/셀상태

```

- 크기 확인

```

print(lstm.shape, forward_h.shape, forward_c.shape, backward_h.shape, backward_c.shape)
-> (None, 500, 128) (None, 64) (None, 64) (None, 64) (None, 64)

```

- 양방향 LSTM의 은닉 상태와 셀 상태 불러오기 : forward-backward 연결


```
state_h = Concatenate()([forward_h, backward_h]) # 은닉 상태
state_c = Concatenate()([forward_c, backward_c]) # 셀 상태
```

- 컨텍스트 벡터 얻기

```
attention = BahdanauAttention(64) # 가중치 크기 정의
context_vector, attention_weights = attention(lstm, state_h) #계층과 은닉상태 입력으로 하기.
```

- 어텐션 거친 이후 모델 설계

```
dense1 = Dense(20, activation="relu")(context_vector) # 밀집층 통과
# 컨텍스트 벡터와 s_t-1을 연결해 새로운 s_t를 만든 게 아님
# 어텐션 없이 나온 s_t를 어텐션에 넣어 컨텍스트 벡터로 만들어 밀집층에 통과시킴.
dropout = Dropout(0.5)(dense1)
output = Dense(1, activation="sigmoid")(dropout) # 감성(이진) 분류이므로 1개뉴런, sigmoid
model = Model(inputs=sequence_input, outputs=output)
```

- 모델 훈련과 결과 확인

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs = 3, batch_size = 256,
                    validation_data=(X_test, y_test), verbose=1)
...
...
...
Epoch 3/3
25000/25000 [=====] - 543s 22ms/sample - loss: 0.1901
- accuracy: 0.9352 - val_loss: 0.3375 - val_accuracy: 0.8793

print("\n 테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
25000/25000 [=====] - 183s 7ms/sample - loss: 0.1901
- acc: 0.8793
테스트 정확도: 0.8793
# 하나의 LSTM 안에서 이전 은닉 상태를 리마인드 시켜주는 셀프 어텐션?
```

Ch 16. 트랜스포머 (Transformer)

1. 트랜스포머?

순환 신경망 없는 인코더-디코더

- 어텐션을 순환 신경망을 보정해주는 용도로 쓰지 않고 어텐션 자체만으로 인코더, 디코더를 만든다는 것이 트랜스포머의 아이디어.

주요 하이퍼파라미터



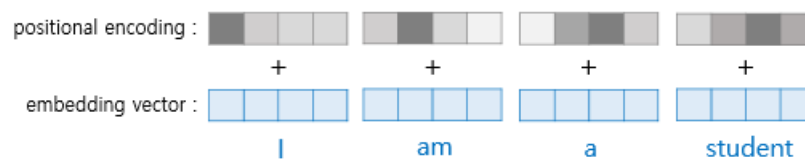
트랜스포머를 제안한 논문에서 사용한 수치 사용

- $d_{\text{model}} = 512$

- 인코더와 디코더의 입/출력 길이. 임베딩 벡터의 차원도 d_{model} 이 되며 차원은 계속 유지된다.
- $num_layers = 6$
 - 인코더-디코더가 몇 층으로 구성되었는지를 의미.
- $num_heads = 8$
 - 어텐션을 한 번 하는 것보다 여러번 작게 나누어 실행하고 그 결과를 합치는 방식을 사용
- $d_{ff} = 2048$
 - 트랜스포머 내부에 존재하는 완전연결 신경망의 은닉층 크기

RNN 없는 시계열 데이터 처리

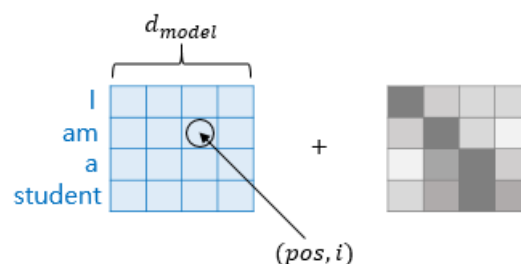
- RNN을 사용하지 않고도 시계열 데이터를 처리하기 위해 트랜스포머는
 - → (각 단어의 임베딩 벡터 + 단어의 위치 정보) 를 모델의 입력으로 사용.
 - 이것이 '포지셔널 인코딩'



포지셔널 인코딩

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

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



- pos = 입력 문장에서 임베딩 벡터의 위치(문장 중 몇 번째 단어인가?)
- i = 임베딩 벡터 내의 차원의 인덱스(단어의 분산표현 중 몇 번째 차원인가?)
 - 홀, 짝에 따라 포지셔널 인코딩 함수에서 \sin , \cos 을 사용.
- PE 구현

```

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]) # 0에서 2씩 커지며 짝수만 인덱싱

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

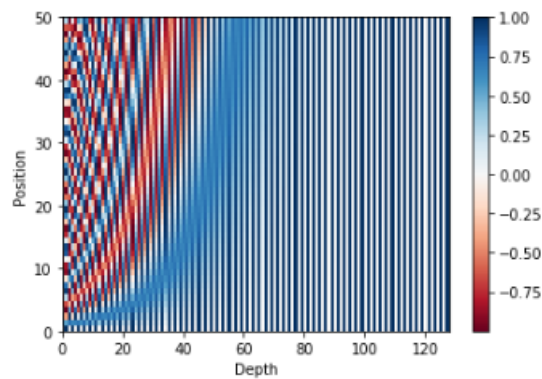
        angle_rads = np.zeros(angle_rads.shape)
        angle_rads[:, 0::2] = sines
        angle_rads[:, 1::2] = cosines
        pos_encoding = tf.constant(angle_rads)
        pos_encoding = pos_encoding[tf.newaxis, ...] #...는 모든 값 불러오기

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

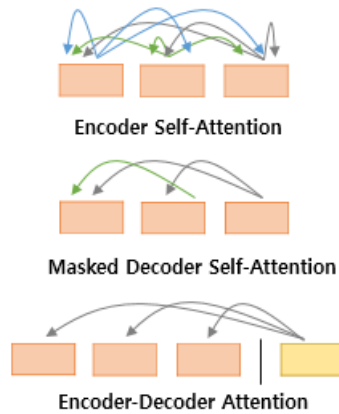
    def call(self, inputs):
        return inputs + self.pos_encoding[:, :tf.shape(inputs)[1], :]

```

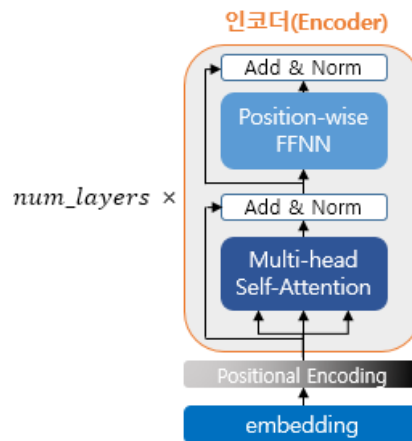
- PE 행렬 시각화



- 트랜스포머에서는 아래처럼 세 종류의 어텐션을 사용한다.




인코더의 구조



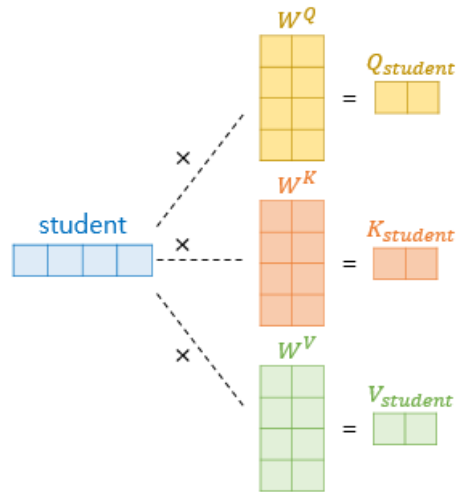
- 하나의 인코더 층은 '멀티 헤드 셀프 어텐션' 과 '포지션 와이즈 FFNN'라는 두 개의 서브층으로 구성된다.

인코더의 셀프 어텐션

- 인코더의 특정 임베딩 벡터가 다른 임베딩 벡터들과 얼마나 연관이 있는가? 를 구하기.
 - Q, K, V는 입력 문장의 단어 벡터들이다.

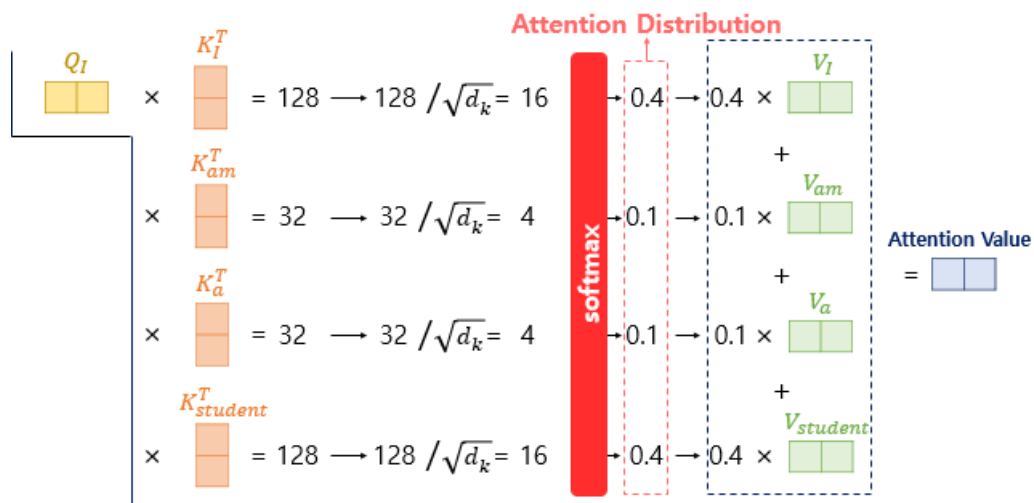
 트랜스포머는 어텐션을 병렬 사용한다. → 여러명이 해석한 다양한 의견을 종합하자.

- 그러므로 Q, K, V의 차원은 원래의 차원 d_model 을 병렬 어텐션 수 num_heads 로 나눈 값이 된다.
- 벡터의 차원 변환은 단어 벡터를 $d_model * (d_model/num_heads)$ 크기의 가중치 행렬에 넣어 실행한다.



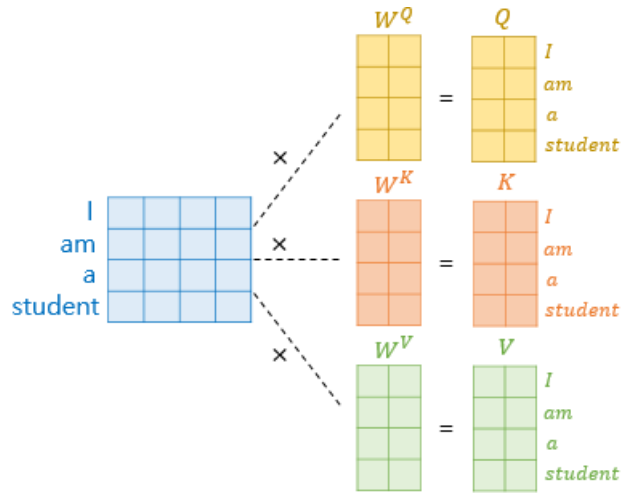
스케일드-닷-프로덕트-어텐션

- 닷-프로덕트 어텐션과 동일하지만 어텐션 스코어를 $\sqrt{d_k}$ 으로 나눠준다.
 - n은 K의 차원 d_k 이다.

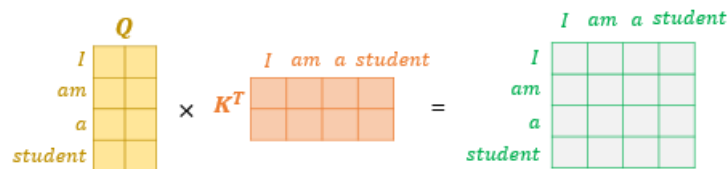


행렬 연산 통한 일괄 처리

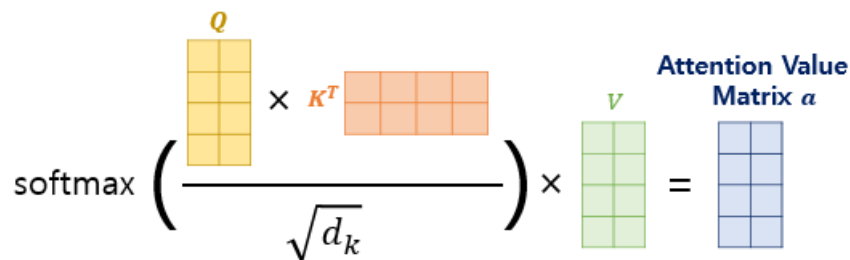
- 어텐션을 위해 모든 단어 벡터의 차원을 줄이는 과정을 반복하는 대신, 문장 행렬의 차원을 줄여(문장 행렬에 가중치 행렬을 곱해) Q, K, V행렬을 만드는 것이 가능하다.



- 그리하면 어텐션 스코어도 각 원소가 Q벡터와 K벡터의 내적이 되는 행렬의 형태로 나온다.



- 그러한 행렬을 $\sqrt{d_k}$ 로 나누고, softmax 함수를 거쳐 value 행렬과 곱해주면 어텐션 값 행렬을 얻는다.



스케일드-닷-프로덕트-어텐션 구현

```
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)

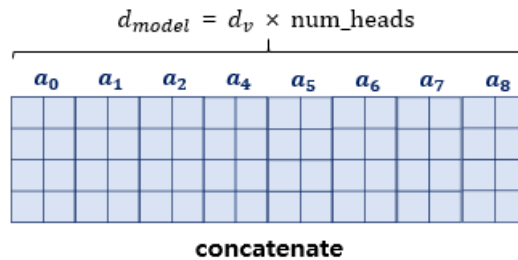
    # 소프트맥스 함수는 마지막 차원인 key의 문장 길이 방향으로 수행된다.
    # attention weight : (batch_size, num_heads, query의 문장 길이, key의 문장 길이)
    attention_weights = tf.nn.softmax(logits, axis=-1)
```

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

return output, attention_weights
```

서브층 1 : 멀티 헤드 어텐션

- Q : 행렬 크기를 줄여서 어텐션 값 행렬 크기도 줄어든 것은 어떡하나요?
- A : 줄어든 만큼 여러번 어텐션을 해서 이어붙입니다.



- num_heads 개의 어텐션 헤드를 연결하면 행렬의 크기는 (seq_len, d_model)이 된다.
- 이 행렬을 또 다른 가중치 행렬 W_o 와 곱한다.

멀티 헤드 어텐션 구현

```
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])
    # 뜬금없이 배치 크기 나오는 이유 : 배치학습 전제로 하는듯
    # batch_size 개의 덩어리 만들
    # 결과 : 배치별 - 어텐션 헤드별 - 문장 길이 * 줄어든 차원 dk(d_model / num_heads)로 데이터를 split

    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)

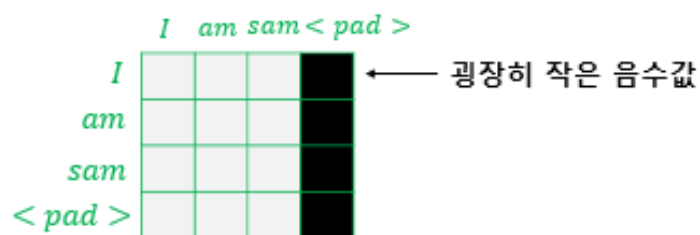
return outputs

# perm은 어텐션 헤드를 나눌 때, 연결할 때 한 번 씩 자료 구조 바꾸기 위해 사용.

```

패딩 마스크

- 어텐션 스코어 행렬에서 key에 패딩 토큰이 있으면(<pad>열이 있으면) 그 열에 매우 작은 음수 할당
 - 리마인드 : q와 T(k)의 행렬곱인 어텐션 스코어 행렬을 softmax에 넣어 만든 어텐션 가중치 행렬과 v를 곱해 어텐션 값을 만드는데, softmax 함수를 지날 때 엄청 작은 값이 있으면 해당 열이 0이 된다.
 - 패딩토큰은 의미가 없으니 가중치를 안 준다!



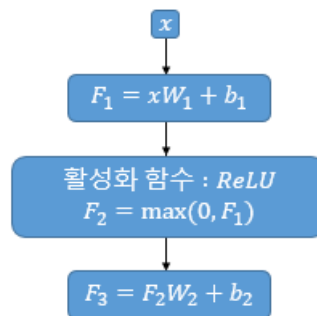
Attention Score Matrix



Attention Score Matrix

서브층 2 : 포지션-와이즈 피드 포워드 신경망

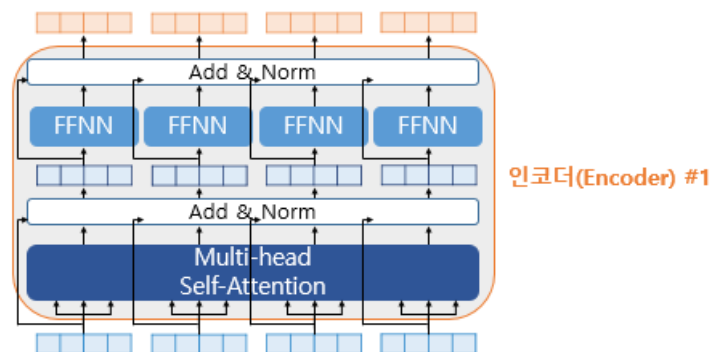
- 두 개의 밀집층의 중첩
 - 밀집층 1 : d_{ff} 개의 뉴런을 가지고 Relu 함수가 활성화 함수.
 - 밀집층 2 : d_{model} 개의 뉴런을 가지고 활성화함수는 따로 없다.
 - → 밀집층 2까지 거치면 input의 형상과 형상 동일.



```
# 두 밀집층의 코드 구현
outputs = tf.keras.layers.Dense(units=dff, activation='relu')(attention)
outputs = tf.keras.layers.Dense(units=d_model)(outputs)
```

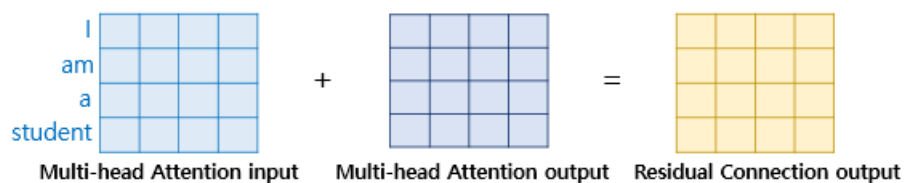
잔차 연결과 층 정규화

- 잔차 연결 : 입력 x 가 서브층을 거칠 때마다 그 출력에 입력을 더해준 것을 출력으로 한다.



- 아래는 멀티 헤드 어텐션에서의 잔차 연결

$$H(x) = x + \text{Multi-head Attention}(x)$$

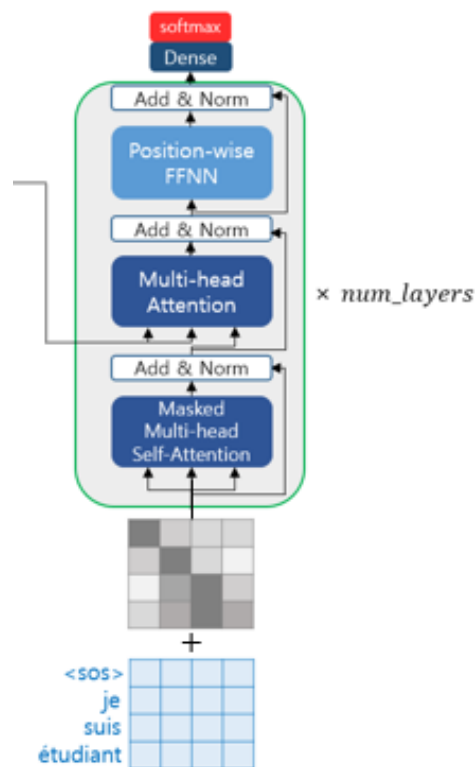


- 층 정규화 : 잔차 연결의 출력을 마지막 차원(d_model)을 기준으로 평균, 분산을 구해 정규화.
 - 역전파시에 미분값 안정화시키는 효과 있다 함.
 - 분모가 0이 되지 않도록 충분히 작은 양수 epsilon 설정하기
 - 이후 초기값이 각각 1, 0인 벡터 감마와 베타를 설정하여 다음과 같이 계산
 - 케라스에서 LayerNormalization() 제공.

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$ln_i = \gamma \hat{x}_i + \beta = LayerNorm(x_i)$$

디코더의 구조



- 트랜스포머의 디코더는 3개의 서브층이 있다.
 - 셀프어텐션 - 인코더/디코더 어텐션 - 포지션 와이즈 FFNN

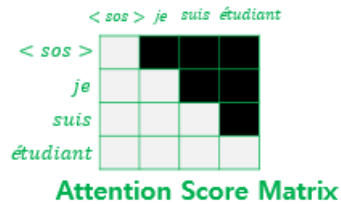
그 중 셀프 어텐션에서는 특이한 마스크가 필요하다.

룩 어헤드 마스크



in 번역, 기존 seq2seq 디코더는 '현재 시점까지 나온 단어' 만 참고 가능했다.
 트랜스포머는 문장 행렬을 한 번에 입력받으므로 미래의 단어도 참고하는 현상 발생(좋은거아님)
 → 못 보게 해야한다 : 룩 어헤드 마스크.

- Q와 K (감소된 차원의 문장 행렬)를 행렬곱한 어텐션 스코어 행렬을 보자.
- 각 행을 보면, 자신의 순서 이하의 단어만 참고하도록 마스크를 하면 디코더가 미래의 단어(정답)을 훑쳐보고 단어를 예측하지 않도록 할 수 있다.



- band_part()를 이용해 행렬의 대각 부분을 기준으로 0을 만들어주고 1에서 빼면 마스크가 된다.

```
look_ahead_mask = 1 - tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)
# https://www.tensorflow.org/api_docs/python/tf/linalg/band_part 에서 자세히 이해해보자..
# 위에서 band_part()의 출력은 왼쪽 아래 삼각형 부분이 1임. -> 1에서 빼면 오른쪽 위 부분이 1로 마스크됨.
```

첫 번째 서브층 : 마스크드 멀티 헤드 셀프 어텐션

- 디코더는 인코더에서와 마찬가지로 셀프 어텐션 서브층을 갖는다.
- 다만 위와 같은 문제의 해결을 위해 룩 어헤드 마스크를 사용한다는 점에서 다르다.

두 번째 서브층 : 인코더 - 디코더 멀티 헤드 어텐션

- 디코더의 문장 행렬이 인코더의 문맥을 참고하도록 병렬 어텐션 진행.

세 번째 서브층 : 포지션 와이즈 피드포워드 신경망

- 인코더에서와 같다.



인코더에서와 같이 각 서브층별로 잔차 연결 & 층 정규화를 거친다.

트랜스포머 구현

```
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

    # 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)

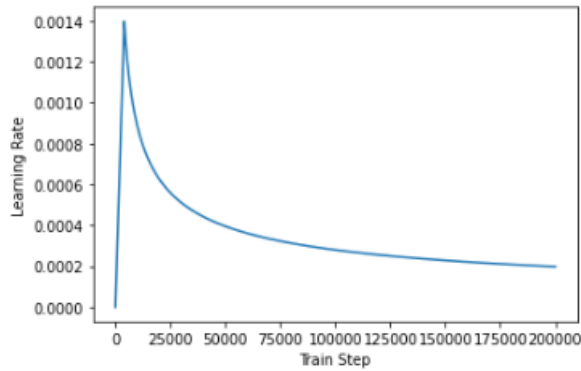
    return tf.keras.Model(inputs=[inputs, dec_inputs], outputs=outputs, name=name)
```

- (마지막) 인코더의 출력은 디코더로 전달된다.
- (마지막) 디코더의 출력은 다중 분류(예시에서는 번역)를 위해 vocab_size만큼의 뉴런을 갖는 출력층으로 전달된다.

학습률 스케줄링

- 큰 학습률로 시작하고 학습 속도가 느려질 때 학습률을 낮추면 고정 학습률보다 효율이 높을 수 있다.
- warmup_steps 를 기준으로 num_step이 작으면 높은 학습률을, 크면 낮은 학습률을 적용한다.
 - 식은 아래와 같다.

$$lr_{rate} = d_{model}^{-0.5} \times \min(\text{step_num}^{-0.5}, \text{step_num} \times \text{warmup_steps}^{-1.5})$$



Ch 16-2. 트랜스포머를 이용한 일상 대화 챗봇

데이터 로드

- 질문과 대답 쌍으로 이루어진 데이터.

	Q	A	label
0	12시 땡!	하루가 또 가네요.	0
1	1지망 학교 떨어졌어	위로해 드립니다.	0
2	3박4일 놀러가고 싶다	여행은 언제나 좋죠.	0
3	3박4일 정도 놀러가고 싶다	여행은 언제나 좋죠.	0
4	PPL 심하네	눈살이 찌푸려지죠.	0

전처리 : 학습 기반 토크나이저 사용

- 서브워드 텍스트 인코더 사용 : BPE와 비슷한 wordpiece model을 사용한다고 한다.

```
tokenizer = tfds.deprecated.text.SubwordTextEncoder.build_from_corpus(
    questions + answers, target_vocab_size=2**13)
```

- 이후 시작 토큰과 종료 토큰에 대한 정수를 부여하고, 전체 데이터를 정수 인코딩 + 패딩한다.

```
# 정수 인코딩 -> 문장 시작, 종료 토큰 추가
sentence1 = START_TOKEN + tokenizer.encode(sentence1) + END_TOKEN
sentence2 = START_TOKEN + tokenizer.encode(sentence2) + END_TOKEN
# 패딩
tokenized_inputs = tf.keras.preprocessing.sequence.pad_sequences(
    tokenized_inputs, maxlen=MAX_LENGTH, padding='post')
tokenized_outputs = tf.keras.preprocessing.sequence.pad_sequences(
    tokenized_outputs, maxlen=MAX_LENGTH, padding='post')
```

- 학습을 위한 디코더 입력 데이터 만들기

```
BATCH_SIZE = 64
```

```

BUFFER_SIZE = 20000 # 를 기준으로 섞음

## 디코더의 실제값 시퀀스에서는 시작 토큰을 제거해야 한다.
dataset = tf.data.Dataset.from_tensor_slices((
    {
        'inputs': questions,
        'dec_inputs': answers[:, :-1] # 디코더의 입력. 마지막 패딩 토큰이 제거된다.
    },
    {
        'outputs': answers[:, 1:] # 맨 처음 토큰이 제거된다. 다시 말해 시작 토큰이 제거된다.
    },
))

dataset = dataset.cache()
dataset = dataset.shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE)
dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)

```

```

tf.keras.backend.clear_session() ## 루프에서 많은 모델 생성에 따른 메모리 부하 방지.

# 하이퍼 파라미터
D_MODEL = 256
NUM_LAYERS = 2
NUM_HEADS = 8
DFF = 512
DROPOUT = 0.1

model = transformer(
    vocab_size=VOCAB_SIZE,
    num_layers=NUM_LAYERS,
    dff=DFF,
    d_model=D_MODEL,
    num_heads=NUM_HEADS,
    dropout=DROPOUT
)

```

```

learning_rate = CustomSchedule(D_MODEL) ## 학습률 조정

optimizer = tf.keras.optimizers.Adam(
    learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)

def accuracy(y_true, y_pred):
    y_true = tf.reshape(y_true, shape=(-1, MAX_LENGTH - 1))
    return tf.keras.metrics.sparse_categorical_accuracy(y_true, y_pred)

model.compile(optimizer=optimizer, loss=loss_function, metrics=[accuracy])

```

```

EPOCHS = 50
model.fit(dataset, epochs=EPOCHS)

-> Epoch 49/50
185/185 [=====] - 22s 116ms/step - loss: 0.0058 - accuracy: 0.1737
Epoch 50/50
185/185 [=====] - 21s 115ms/step - loss: 0.0062 - accuracy: 0.1736
<keras.callbacks.History at 0x7f526b651350>

```

• 챗봇 평가

```

def preprocess_sentence(sentence):
    # 단어와 구두점 사이에 공백 추가.
    # ex) 12시 땡! -> 12시 땡 !
    sentence = re.sub(r"([?.!,])", r" \1 ", sentence)

```

```

sentence = sentence.strip()
return sentence

def evaluate(sentence):
    # 입력 문장에 대한 전처리
    sentence = preprocess_sentence(sentence)

    # 입력 문장에 시작 토큰과 종료 토큰을 추가
    sentence = tf.expand_dims(
        START_TOKEN + tokenizer.encode(sentence) + END_TOKEN, axis=0)

    output = tf.expand_dims(START_TOKEN, 0)

    # 디코더의 예측 시작
    for i in range(MAX_LENGTH):
        predictions = model(inputs=[sentence, output], training=False)

        # 현재 시점의 예측 단어를 받아온다.
        predictions = predictions[:, -1:, :]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)

        # 만약 현재 시점의 예측 단어가 종료 토큰이라면 예측을 중단
        if tf.equal(predicted_id, END_TOKEN[0]):
            break

        # 현재 시점의 예측 단어를 output(출력)에 연결한다.
        # output은 for문의 다음 루프에서 디코더의 입력이 된다.
        output = tf.concat([output, predicted_id], axis=-1)

    # 단어 예측이 모두 끝났다면 output을 리턴.
    return tf.squeeze(output, axis=0)

def predict(sentence):
    prediction = evaluate(sentence)

    # prediction == 디코더가 리턴한 챗봇의 대답에 해당하는 정수 시퀀스
    # tokenizer.decode()를 통해 정수 시퀀스를 문자열로 디코딩.
    predicted_sentence = tokenizer.decode(
        [i for i in prediction if i < tokenizer.vocab_size])

    print('Input: {}'.format(sentence))
    print('Output: {}'.format(predicted_sentence))

    return predicted_sentence

```

```

output = predict("영화 볼래?")
-> Input: 영화 볼래?
    Output: 최신 영화가 좋을 것 같아요 .
## 답변 그럴싸함.

```