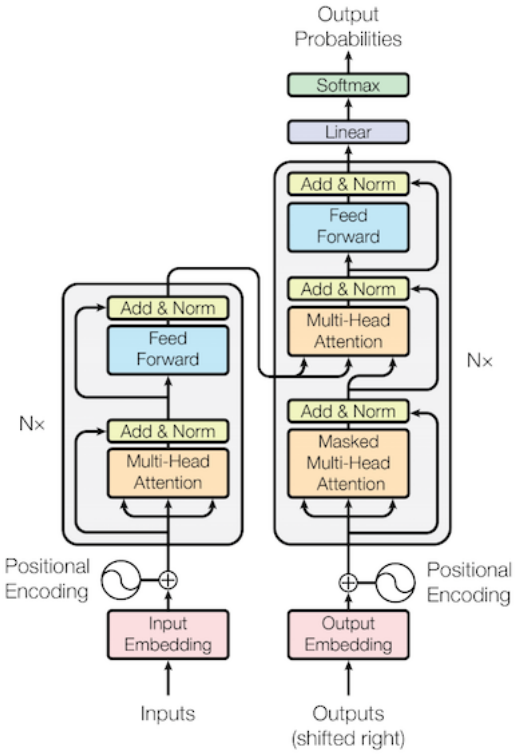
1️⃣ Transformer Code 1

Transformer의 구조

이론은 공부했다 가정하에 포스팅



위와 같이 Encoder와 Decoder 부분으로 나누어져 있음

일단 Encoder와 Decoder 어디든 들어가는 Attention에 대해 코딩 하겠음

1. 한국 위키피티아 크롤링 해서 만들어진 Vocab을 Load

```
import torch
import torch.nn as nn
import sentencepiece as spm
import numpy as np
import matplotlib.pyplot as plt
# vocab loading
vocab_file = "/home/cschoi/Ko_ViLT/Transformer/Vocab_Sentencepiece/web-crawler/kowiki.model"
vocab = spm.SentencePieceProcessor()
vocab.load(vocab_file)
```

입력 texts를 Tensor로 변환

```
lines = [
  "겨울은 추워요.",
  "감기 조심하세요."
]

# text를 tensor로 변환
inputs = []
for line in lines:
```

```
  pieces = vocab.encode_as_pieces(line)
  ids = vocab.encode_as_ids(line)
  inputs.append(torch.tensor(ids))
  print(pieces)
```

Pieces 출력 결과

```
['▁겨울', '은', '▁추', '워', '요', '.']
['▁감', '기', '▁조', '심', '하', '세', '요', '.']
```

보다 싶이 입력으로 들어갈 길이가 다르기에 입력 최대 길이에 맞춰 padding(0)을 추가 해 줌

```
inputs = torch.nn.utils.rnn.pad_sequence(inputs, batch_first=True, padding_value=0)
# shape
print(inputs.size())
# 값
print(inputs)
```

inputs 출력 결과

```
torch.Size([2, 8])
tensor([[3207, 3634,  197, 3986, 3790, 3620,    0,    0],
        [ 195, 3636,   53, 3865, 3626, 3712, 3790, 3620]])
```

2. Embedding

자! 이제 주어진 inputs을 임베딩해줘야 함.

embedding은 입력 토큰을 vector 형태로 변환하는 것임.

이론에서 배웠다 싶이 Transformer의 Embedding은 **'Input Embedding'**과 **'Postion Embedding'** 두 가지를 합해서 사용함

▼ **트랜스포머(Transformer)의 주요 하이퍼파라미터**

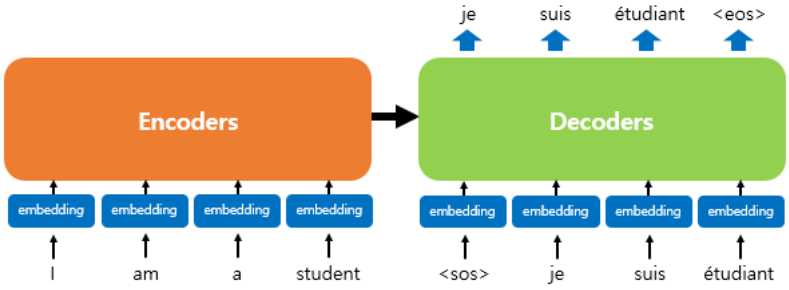
d_{model}(본 코드 **d_hidn**) = 512트랜스포머의 인코더와 디코더에서의 정해진 입력과 출력의 크기를 의미합니다. 임베딩 벡터의 차원 또한 d_{model}이며, 각 인코더와 디코더가 다음 층의 인코더와 디코더로 값을 보낼 때에도 이 차원을 유지합니다. 논문에서는 512입니다. **본 코드에서는 128**

num_layers = 6트랜스포머에서 하나의 인코더와 디코더를 층으로 생각하였을 때, 트랜스포머 모델에서 인코더와 디코더가 총 몇 층으로 구성되었는지를 의미합니다. 논문에서는 인코더와 디코더를 각각 총 6개 쌓았습니다.

num_heads = 8트랜스포머에서는 어텐션을 사용할 때, 한 번 하는 것 보다 여러 개로 분할해서 병렬로 어텐션을 수행하고 결과값을 다시 하나로 합치는 방식을 택했습니다. 이때 이 병렬의 개수를 의미합니다. 본 코드에서는 **n_head = 2**

d_{ff} = 2048트랜스포머 내부에는 피드 포워드 신경망이 존재하며 해당 신경망의 **은닉층의 크기**를 의미합니다. 피드 포워드 신경망의 입력층과 출력층의 크기는 d_{model}입니다. **본 코드에서는 512 =** d_{model}**(본 코드 d_hidn) * 4**

2-1 Input Embedding



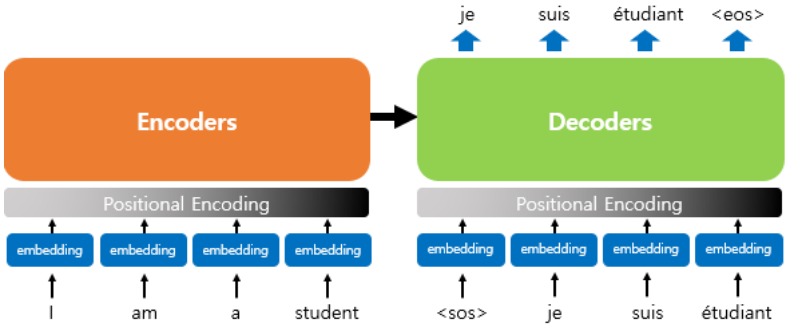
inputs에 대한 embedding 값 input_embs를 구함.

```
n_vocab = len(vocab) # vocab count
# 트랜스포머의 모든 층의 출력 차원을 의미하는 트랜스포머의 하이퍼파라미터
d_hidn = 128 # hidden size
nn_emb = nn.Embedding(n_vocab, d_hidn) # embedding 객체

input_embs = nn_emb(inputs) # input embedding
print(input_embs.size())
print()
```

inputs 임베딩의 차원

```
torch.Size([2, 8, 128])
```

2-2 Position Embedding



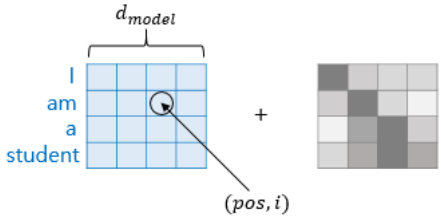
포지셔널 인코딩의 덧셈은 사실 임베딩 벡터가 모여 만들어진 **문장 행렬**과 **포지셔널 인코딩 행렬**의 **덧셈 연산**을 통해 이루어진다는 점을 이해해야 함.

따라서 **두 행렬의 차원이 같아야 할 것**임

Position embedding을 구하는 연습!

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

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



1. 각 position별도 angle 값을 구함.
2. 구해진 angle 중 짝수 index의 값에 대한 sin 값을 구함.
3. 구해진 angle 중 홀수 index의 값에 대한 cos 값을 구함.

```
def get_sinusoid_encoding_table(n_seq, d_hidn):
    def cal_angle(position, i_hidn):
        return position / np.power(10000, 2 * (i_hidn // 2) / d_hidn)
    def get_posi_angle_vec(position):
        return [cal_angle(position, i_hidn) for i_hidn in range(d_hidn)]

    sinusoid_table = np.array([get_posi_angle_vec(i_seq) for i_seq in range(n_seq)])
    sinusoid_table[:, 0::2] = np.sin(sinusoid_table[:, 0::2])  # even index sin
    sinusoid_table[:, 1::2] = np.cos(sinusoid_table[:, 1::2])  # odd index cos

    return sinusoid_table
```

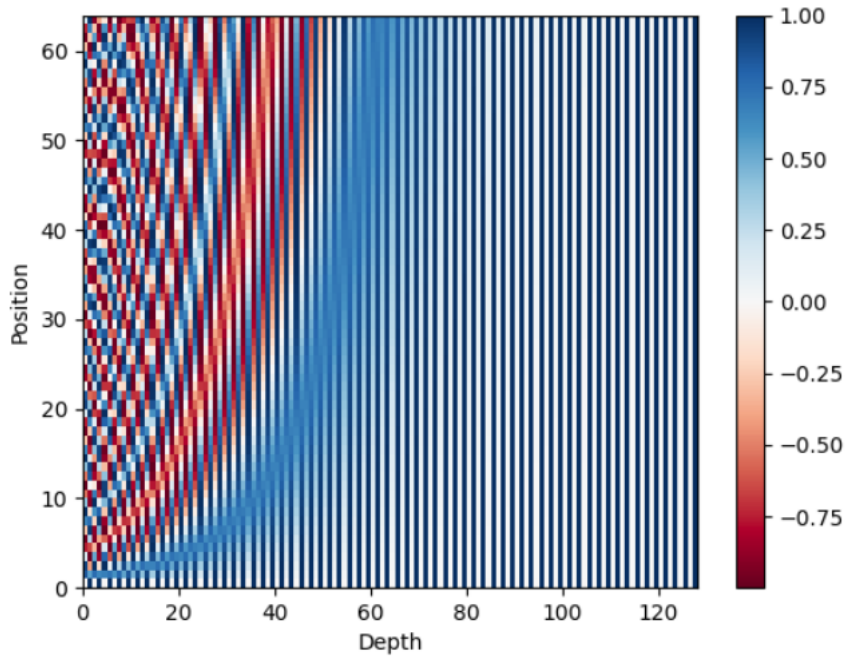
문장의 길이가 64라고 하고 하고 시각화를 한다면

```
n_seq = 64
pos_encoding = get_sinusoid_encoding_table(n_seq, d_hidn)

print("position 임베딩의 차원")
print (pos_encoding.shape) # 크기 출력
print()

plt.pcolormesh(pos_encoding, cmap='RdBu')
plt.xlabel('Depth')
plt.xlim((0, d_hidn))
plt.ylabel('Position')
plt.colorbar()
plt.savefig('./pos_encoding.png')
```

position 임베딩의 차원은

```
(64, 128)
```

(n_seq, d_hidn)
입력 문장의 단어가 64개이면서, 각 단어가 128차원의 임베딩 벡터를 가질 때 사용할 수 있는 행렬입니다.



이제 inputs 임베딩에 더해줄 position embedding 값을 구해보자!

아래 절차로 position embedding 값을 구합니다.

1. 위해서 구해진 position encodong 값을 이용해 position emgedding을 생성합니다. 학습되는 값이 아니므로 freeze옵션을 True로 설정 합니다.
2. 입력 inputs과 동일한 크기를 갖는 positions값을 구합니다.
3. input값 중 pad(0)값을 찾습니다.
4. positions값중 pad부분은 0으로 변경 합니다.
5. positions값에 해당하는 embedding값을 구합니다.

```
pos_encoding = torch.FloatTensor(pos_encoding)
nn_pos = nn.Embedding.from_pretrained(pos_encoding, freeze=True)

positions = torch.arange(inputs.size(1), device=inputs.device, dtype=inputs.dtype).expand(inputs.size(0), inputs.size(1)).contiguous()
pos_mask = inputs.eq(0)
```

```
positions.masked_fill_(pos_mask, 0)
pos_embs = nn_pos(positions) # position embedding

print("inputs의 pad(0) 위치에 positions의 값도 pad(0)으로 변경 되어 있음을 알 수 있음.")
print(inputs)
print(positions)
print(pos_embs.size())
print()
```

```
inputs
tensor([[3207, 3634,  197, 3986, 3790, 3620,    0,    0],
        [ 195, 3636,   53, 3865, 3626, 3712, 3790, 3620]])
positions
tensor([[1, 2, 3, 4, 5, 6, 0, 0],
        [1, 2, 3, 4, 5, 6, 7, 8]])
pos_embs.size()
torch.Size([2, 8, 128])
```

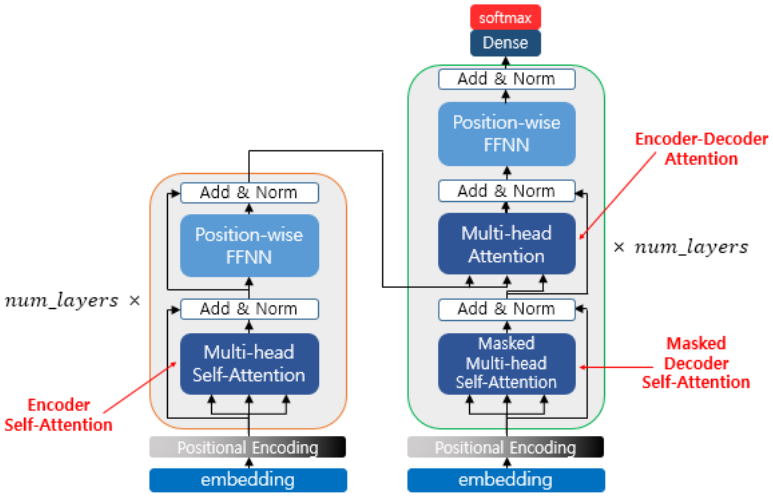
이렇게 inputs embedding와 같은 차원의 Positional embedding이 만들어 졌다! [2, 8, 128]

마지막으로 input임베딩과 position임베딩을 행렬 덧셈 해주기만 하면 된다

```
input_sums = input_embs + pos_embs
```

이러면 문장의 단어의 각 벡터마다 Positon의 정보가 담기는 것

3. Attention



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

어텐션의 입력값은 Q(query), K(key), V(value) 그리고 attention mask로 구성 되어 있음. 약간 정리하자면

> 인코더의 셀프 어텐션 : Query = **Key**= Value
> 디코더의 마스크드 셀프 어텐션 : Query = **Key** = Value
> 디코더의 인코더-디코더 어텐션 : Query : 디코더 벡터 / **Key** = Value : 인코더 벡터

위 그림에 세 개의 어텐션에 추가적으로 '멀티 헤드'라는 이름이 붙어있는데. 트랜스포머가 어텐션을 병렬적으로 수행하는 방법을 의미.

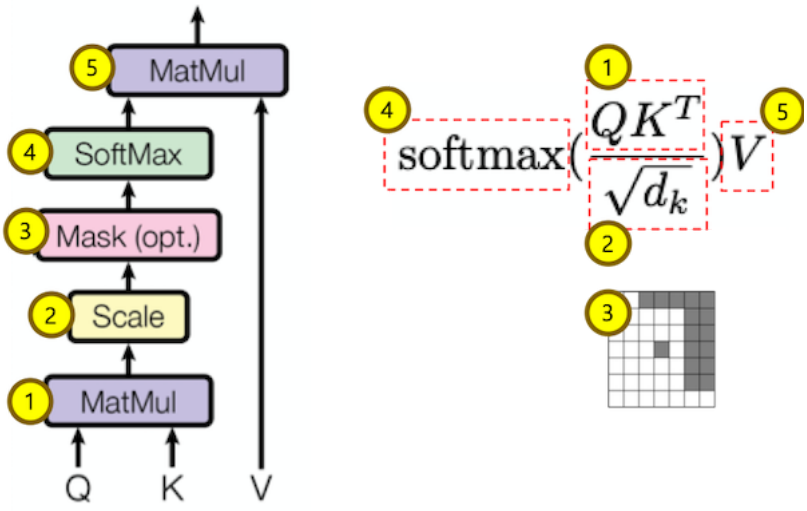
```
Q = input_sums
K = input_sums
V = input_sums
attn_mask = inputs.eq(0).unsqueeze(1).expand(Q.size(0), Q.size(1), K.size(1))
print("attn_mask의 값은 pad(0) 부분만 True")
print(attn_mask.size())
print(attn_mask[0])
print()
```

attn_mask의 값을 확인해보면 pad(0) 부분만 True 임

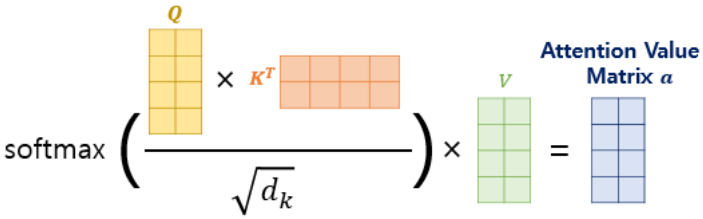
```
torch.Size([2, 8, 8])
tensor([[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True],
[False, False, False, False, False, False,  True,  True]])
```

이제 어텐션 연산에 활용되는 스케일드 닷-프로덕트 어텐션(Scaled dot-product Attention) 연산을 확인 해보자

순서는 아래와 같다



d_{model}의 차원을 num_heads
개로 나누어 d_{model}/num_heads
의 차원을 가지는 Q, K, V에 대해서 num_heads
개의 병렬 어텐션을 수행합니다.



입력값은 Q(query), K(key), V(value) 그리고 attention mask로 구성 되어 있음

```
Q = input_sums
K = input_sums
V = input_sums
attn_mask = inputs.eq(0).unsqueeze(1).expand(Q.size(0), Q.size(1), K.size(1))
```

```
print(attn_mask.size())
print(attn_mask[0])
print()
batch_size = Q.size(0)
n_head = 2
```

배치사이즈 = 2, n_head = 2

attn_mask의 값은 pad(0) 부분만 True

```
attn_mask
torch.Size([2, 8, 8])
tensor([[[False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True],
         [False, False, False, False, False, False,  True,  True]])
```

(1) Q * K-transpose를 하는 코드

```
print("MatMul Q, K-transpose")
scores = torch.matmul(Q, K.transpose(-1, -2))
print(scores.size())
print(scores[0])
```

각 단어상호간에 가중치를 표현하는 테이블을 생성

```
MatMul Q, K-transpose
torch.Size([2, 8, 8])
tensor([[180.7564,  65.8829,  61.2576,  36.8172,  53.6519,  43.9333,  61.2428,
61.2428],
[ 65.8829, 194.3670,  59.5719,  34.1793,  42.4963,  64.2007,  28.5740,
28.5740],
[ 61.2576,  59.5719, 201.9257,  32.9764,  51.7735,  66.5509,  45.4557,
45.4557],
[ 36.8172,  34.1793,  32.9764, 150.4789,  44.6441,  28.1509,  19.1284,
19.1284],
[ 53.6519,  42.4963,  51.7735,  44.6441, 177.7521,  64.5677,  48.8788,
48.8788],
[ 43.9333,  64.2007,  66.5509,  28.1509,  64.5677, 209.3232,  40.8650,
40.8650],
[ 61.2428,  28.5740,  45.4557,  19.1284,  48.8788,  40.8650, 186.2899,
186.2899],
[ 61.2428,  28.5740,  45.4557,  19.1284,  48.8788,  40.8650, 186.2899,
186.2899]], grad_fn=<SelectBackward0>)
```

(2) Scale

k-dimension에 루트를 취한 값으로 나누는 코드.

▼ d_head(d_k)

우선 입력 문장의 길이를 seq_len이라고 해봅시다. 그렇다면 문장 행렬의 크기는 (seq_len, d_{model})입니다. 여기에 3개의 가중치 행렬을 곱해서 Q, K, V 행렬을 만들어야 합니다.
Q행렬과 K행렬의 크기는 (seq_len, d_k)이며, V행렬의 크기는 (seq_len, d_v)가 되어야 합니다. 그렇다면 문장 행렬과 Q, K, V 행렬의 크기로부터 가중치 행렬의 크기 추정이 가능합니다. W^Q와 W^K는 (d_{model}, d_k)
의 크기를 가지며, W^V는 (d_{model}, d_v)의 크기를 가집니다. 단, 논문에서는 d_k와 d_v의 차원은 d_{model}/num_heads와 같습니다. 즉, d_{model}/num_heads= d_k = d_v입니다.

```
print("Scale")
d_head = 64
scores = scores.mul_(1/d_head**0.5)
print(scores.size())
print(scores[0])
```

위 값에 비해서 가중치 편차가 줄어든 것을 확인 할 수 있음.

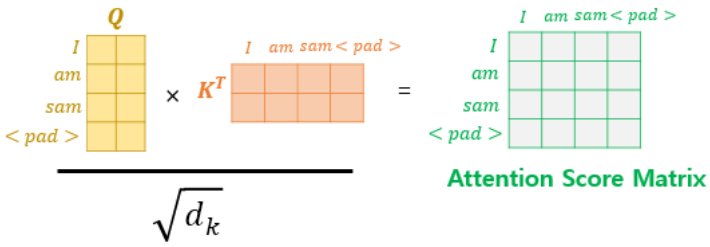
```
Scale
torch.Size([2, 8, 8])
tensor([[22.5946,  8.2354,  7.6572,  4.6022,  6.7065,  5.4917,  7.6553,  7.6553],
[ 8.2354, 24.2959,  7.4465,  4.2724,  5.3120,  8.0251,  3.5718,  3.5718],
[ 7.6572,  7.4465, 25.2407,  4.1221,  6.4717,  8.3189,  5.6820,  5.6820],
[ 4.6022,  4.2724,  4.1221, 18.8099,  5.5805,  3.5189,  2.3911,  2.3911],
[ 6.7065,  5.3120,  6.4717,  5.5805, 22.2190,  8.0710,  6.1098,  6.1098],
[ 5.4917,  8.0251,  8.3189,  3.5189,  8.0710, 26.1654,  5.1081,  5.1081],
[ 7.6553,  3.5718,  5.6820,  2.3911,  6.1098,  5.1081, 23.2862, 23.2862],
[ 7.6553,  3.5718,  5.6820,  2.3911,  6.1098,  5.1081, 23.2862, 23.2862]],
grad_fn=<SelectBackward0>)
```

(3) Mask(Opt.)

mask를 하는 코드

마스킹. 어텐션 스코어 행렬의 마스킹 할 위치에 매우 작은 음수값을 넣음.

매우 작은 값이므로 소프트맥스 함수를 지나면 행렬의 해당 위치의 값은 0이 된다.
즉 <PAD> 토큰이 존재한다면 이에 대해서는 유사도를 구하지 않도록 함




```
print("Mask (Opt.)")
scores.masked_fill_(attn_mask, -1e9)
print(scores.size())
print(scores)
```

mask를 한 부분이 -1e9로 매우 작은 값으로 변경된 것을 확인 할 수 있음.

```
Mask (Opt.)
torch.Size([2, 8, 8])
tensor([[ 2.0172e+01,  7.9554e+00,  7.3960e+00,  6.5239e+00,  4.7410e+00,
2.6578e+00, -1.0000e+09, -1.0000e+09],
[ 7.9554e+00,  2.7101e+01,  7.0413e+00,  3.9789e+00,  6.6717e+00,
5.4765e+00, -1.0000e+09, -1.0000e+09],
[ 7.3960e+00,  7.0413e+00,  2.4798e+01,  6.8370e+00,  5.9531e+00,
6.2476e+00, -1.0000e+09, -1.0000e+09],
[ 6.5239e+00,  3.9789e+00,  6.8370e+00,  2.6690e+01,  9.9979e+00,
7.2079e+00, -1.0000e+09, -1.0000e+09],
[ 4.7410e+00,  6.6717e+00,  5.9531e+00,  9.9979e+00,  2.2829e+01,
7.2579e+00, -1.0000e+09, -1.0000e+09],
[ 2.6578e+00,  5.4765e+00,  6.2476e+00,  7.2079e+00,  7.2579e+00,
2.3626e+01, -1.0000e+09, -1.0000e+09],
[ 6.1388e+00,  8.0745e+00,  5.0602e+00,  6.6181e+00,  3.7223e+00,
4.5345e+00, -1.0000e+09, -1.0000e+09],
[ 6.1388e+00,  8.0745e+00,  5.0602e+00,  6.6181e+00,  3.7223e+00,
4.5345e+00, -1.0000e+09, -1.0000e+09]], grad_fn=<SelectBackward0>)
```

(4) Softmax

```
print("Softmax")
attn_prob = nn.Softmax(dim=-1)(scores)
print(attn_prob.size())
print(attn_prob[0])
```

가중치가 확률로 변환 된 값을 볼 수 있음. mask를 한 부분이 모두 0이 됨.

```
Softmax
torch.Size([2, 8, 8])
tensor([[9.9999e-01, 4.9489e-06, 2.8285e-06, 1.1825e-06, 1.9883e-07, 2.4761e-08,
```

```
         0.0000e+00, 0.0000e+00],
        [4.8427e-09, 1.0000e+00, 1.9412e-09, 9.0803e-11, 1.3415e-09, 4.0599e-10,
         0.0000e+00, 0.0000e+00],
        [2.7696e-08, 1.9425e-08, 1.0000e+00, 1.5835e-08, 6.5427e-09, 8.7832e-09,
         0.0000e+00, 0.0000e+00],
        [1.7464e-09, 1.3704e-10, 2.3883e-09, 1.0000e+00, 5.6345e-08, 3.4608e-09,
         0.0000e+00, 0.0000e+00],
        [1.3948e-08, 9.6171e-08, 4.6874e-08, 2.6764e-06, 1.0000e+00, 1.7282e-07,
         0.0000e+00, 0.0000e+00],
        [7.8311e-10, 1.3121e-08, 2.8369e-08, 7.4112e-08, 7.7911e-08, 1.0000e+00,
         0.0000e+00, 0.0000e+00],
        [9.8287e-02, 6.8103e-01, 3.3424e-02, 1.5873e-01, 8.7704e-03, 1.9759e-02,
         0.0000e+00, 0.0000e+00],
        [9.8287e-02, 6.8103e-01, 3.3424e-02, 1.5873e-01, 8.7704e-03, 1.9759e-02,
         0.0000e+00, 0.0000e+00]], grad_fn=<SelectBackward0>)
```

(5) MatMul attn_prov, V

```
print("MatMul attn_prov, V")
context = torch.matmul(attn_prob, V)
print(context.size())
```

attn_prov * V를 하는 코드
Q와 동일한 shape 값이 구해짐. 이 값은 V값들이 attn_prov의 가중치를 이용해서 더해진 값.

```
MatMul attn_prov, V
torch.Size([2, 8, 128])
```

위 절차를 하나의 클래스 형태로 구성하면 아래와 같음

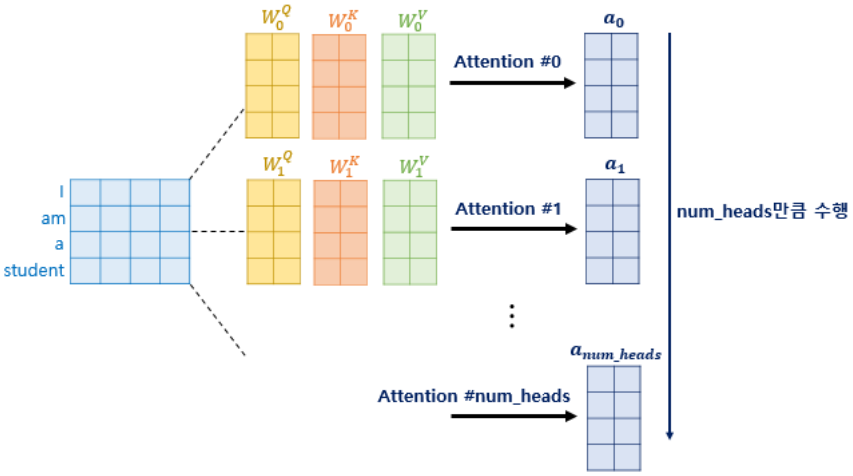
`Scaled Dot Product Attention Class`

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, d_head):
        super().__init__()
        self.scale = 1 / (d_head ** 0.5)

    def forward(self, Q, K, V, attn_mask):
        # (bs, n_head, n_q_seq, n_k_seq)
        scores = torch.matmul(Q, K.transpose(-1, -2)).mul_(self.scale)
        scores.masked_fill_(attn_mask, -1e9)
        # (bs, n_head, n_q_seq, n_k_seq)
        attn_prob = nn.Softmax(dim=-1)(scores)
        # (bs, n_head, n_q_seq, d_v)
        context = torch.matmul(attn_prob, V)
        # (bs, n_head, n_q_seq, d_v), (bs, n_head, n_q_seq, n_v_seq)
        return context, attn_prob
```

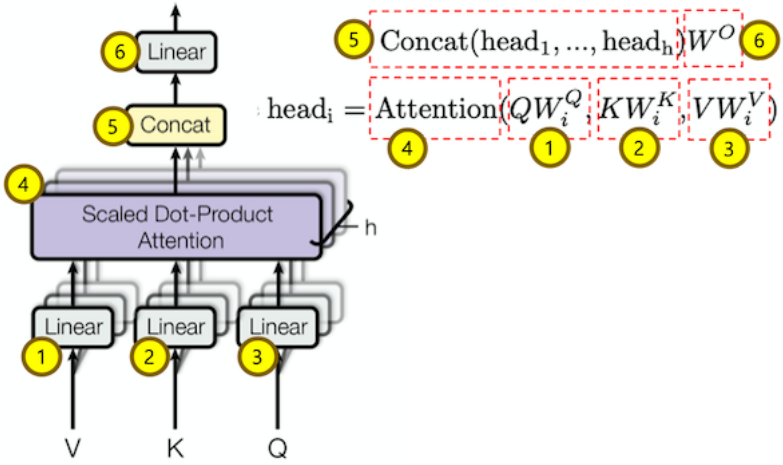
어탠션 연산을 알았으니 여러번의 어텐션을 병렬로 사용하는

Multi-Head Attention

을 해보자



d_{model}(본 코드 **d_hidn**)의 차원을 num_heads 개로 나누어 d_{model}/num_heads의 차원을 가지는 Q, K, V에 대해서 num_heads 개의 병렬 어텐션을 수행함.



(1~3) Multi Head Q, K, V

Q를 여러개의 head로 나누는 과정(2개).

```
# d_hidn = 128, n_head = 2, d_head = 64
W_Q = nn.Linear(d_hidn, n_head * d_head)
W_K = nn.Linear(d_hidn, n_head * d_head)
W_V = nn.Linear(d_hidn, n_head * d_head)

# (bs, n_seq, n_head * d_head)
q_s = W_Q(Q)
print(q_s.size())
# (bs, n_seq, n_head, d_head)
q_s = q_s.view(batch_size, -1, n_head, d_head)
print(q_s.size())
# (bs, n_head, n_seq, d_head)
q_s = q_s.transpose(1,2)
print(q_s.size())
```

보다 싶이 n_head(2) 개로 나누어 졌음

```
q_s
torch.Size([2, 8, 128])
torch.Size([2, 8, 2, 64])
torch.Size([2, 2, 8, 64])
```

위의 과정을 한줄로 일반화 한다면

```
# (bs, n_head, n_seq, d_head)
q_s= W_Q(Q).view(batch_size,-1, n_head, d_head).transpose(1,2)
# (bs, n_head, n_seq, d_head)
k_s= W_K(K).view(batch_size,-1, n_head, d_head).transpose(1,2)
# (bs, n_head, n_seq, d_head)
v_s= W_V(V).view(batch_size,-1, n_head, d_head).transpose(1,2)
print(q_우
```

Q, K, V 모두 Multi Head로 나눠짐

```
torch.Size([2, 2, 8, 64]) torch.Size([2, 2, 8, 64]) torch.Size([2, 2, 8, 64])
```

Attention Mask도 마찬가지로 Multi Head로 변경 해야 함.

```
print("\nattn_mask")
print(attn_mask.size())
attn_mask = attn_mask.unsqueeze(1).repeat(1, n_head, 1, 1)
print(attn_mask.size())
```

```
attn_mask
torch.Size([2, 8, 8])
torch.Size([2, 2, 8, 8])
```

(4) Scaled Dot-Product Attention

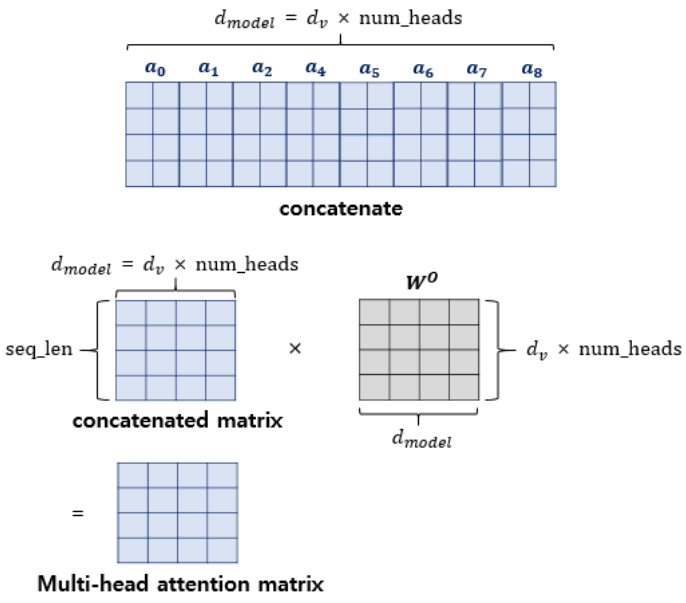
분리된 Q, K, S와 Ateention Mask로 위에 설명한 ‘Scaled Dot Product Attention’을 사용

```
context, attn_prob= scaled_dot_attn(q_s, k_s, v_s, attn_mask)
print(context.size())
print(attn_prob.size())
```

Multi Head에 대한 Attention이 구해짐

```
context
torch.Size([2, 2, 8, 64])
attn_prob
torch.Size([2, 2, 8, 8])
```

(5) Concat



병렬 어텐션을 모두 수행하였다면 모든 어텐션 헤드를 concatenate 함. 모두 연결된 어텐션 헤드 행렬의 크기는 (seq_len, d_{model})가 됨

```
# (bs, n_seq, n_head * d_head)
context = context.transpose(1, 2).contiguous().view(batch_size, -1, n_head * d_head)
print("\ncontext")
print(context.size())
```

Multi Head를 한개로 합침

```
context
torch.Size([2, 8, 128])
```

(6) Linear

```
linear = nn.Linear(n_head * d_head, d_hidn)
# (bs, n_seq, d_hidn)
output = linear(context)
print("Linear")
print(output.size())
```

입력 Q와 동일한 shape을 가진 Multi Head Attention이 구해짐

```
Linear
torch.Size([2, 8, 128])
```

위 절차를 하나의 클래스 형태로 구성하면 아래와 같음

`Multi Head Attention Class`

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_hidn, n_head, d_head):
        super().__init__()
        self.d_hidn = d_hidn
        self.n_head = n_head
        self.d_head = d_head

        self.W_Q = nn.Linear(d_hidn, n_head * d_head)
        self.W_K = nn.Linear(d_hidn, n_head * d_head)
        self.W_V = nn.Linear(d_hidn, n_head * d_head)
        self.scaled_dot_attn = ScaledDotProductAttention(d_head)
        self.linear = nn.Linear(n_head * d_head, d_hidn)

    def forward(self, Q, K, V, attn_mask):
        batch_size = Q.size(0)
        # (bs, n_head, n_q_seq, d_head)
        q_s = self.W_Q(Q).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)
        # (bs, n_head, n_k_seq, d_head)
        k_s = self.W_K(K).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)
        # (bs, n_head, n_v_seq, d_head)
        v_s = self.W_V(V).view(batch_size, -1, self.n_head, self.d_head).transpose(1,2)

        # (bs, n_head, n_q_seq, n_k_seq)
        attn_mask = attn_mask.unsqueeze(1).repeat(1, self.n_head, 1, 1)

        # (bs, n_head, n_q_seq, d_head), (bs, n_head, n_q_seq, n_k_seq)
        context, attn_prob = self.scaled_dot_attn(q_s, k_s, v_s, attn_mask)
        # (bs, n_head, n_q_seq, h_head * d_head)
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.n_head * self.d_head)
        # (bs, n_head, n_q_seq, e_embd)
        output = self.linear(context)
        # (bs, n_q_seq, d_hidn), (bs, n_head, n_q_seq, n_k_seq)
        return output, attn_prob
```

Masked Multi-Head Attention

Masked Multi-Head Attention은 Multi-Head Attention과 attention mask를 제외한 부분은 모두 동일.

```
""" attention decoder mask """
def get_attn_decoder_mask(seq):
    subsequent_mask = torch.ones_like(seq).unsqueeze(-1).expand(seq.size(0), seq.size(1), seq.size(1))
    subsequent_mask = subsequent_mask.triu(diagonal=1) # upper triangular part of a matrix(2-D)
    return subsequent_mask


Q = input_sums
K = input_sums
V = input_sums

attn_pad_mask = inputs.eq(0).unsqueeze(1).expand(Q.size(0), Q.size(1), K.size(1))
print("attn_pad_mask")
print(attn_pad_mask[1])
attn_dec_mask = get_attn_decoder_mask(inputs)
print(attn_dec_mask[1])
attn_mask = torch.gt((attn_pad_mask + attn_dec_mask), 0)
print(attn_mask[1])
print(attn_mask.size())
batch_size = Q.size(0)
n_head = 2
```

pad mask, decoder mask 그리고 이 둘을 합한 attention mask를 확인 할 수 있음.

```
tensor([[False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False],
        [False, False, False, False, False, False, False, False]])
tensor([[0, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0]])
tensor([[False,  True,  True,  True,  True,  True,  True,  True],
        [False, False,  True,  True,  True,  True,  True,  True],
        [False, False, False,  True,  True,  True,  True,  True],
        [False, False, False, False,  True,  True,  True,  True],
        [False, False, False, False, False,  True,  True,  True],
        [False, False, False, False, False, False,  True,  True],
        [False, False, False, False, False, False, False,  True],
        [False, False, False, False, False, False, False, False]])
torch.Size([2, 8, 8])
```

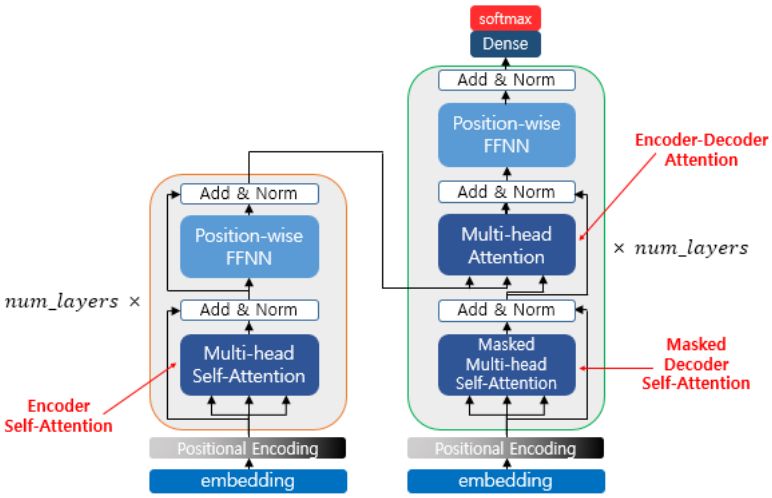
Multi-Head Attention과 동일하므로 위에서 선언한 MultiHeadAttention 클래스를 바로 호출.

```
attention = MultiHeadAttention(d_hidn, n_head, d_head)
output, attn_prob = attention(Q, K, V, attn_mask)
print("\nMultiHeadAttention output, attn_prob")
print(output.size(), attn_prob.size())
```

출력

```
MultiHeadAttention output, attn_prob
torch.Size([2, 8, 128]) torch.Size([2, 2, 8, 8])
```

포지션-와이즈 피드 포워드 신경망(Position-wise FFNN)

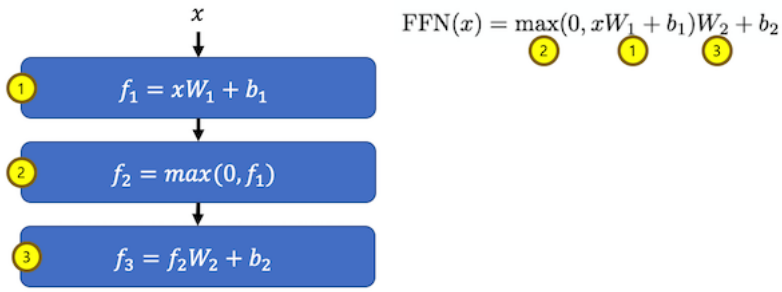


포지션 와이즈 FFNN은 인코더와 디코더에서 공통적으로 가지고 있는 서브층임. 포지션-와이즈 FFNN는 쉽게 말하면 완전 연결 FFNN(Fully-connected FFNN)이라고 해석할 수 있음.

아래는 포지션 와이즈 FFNN의 수식을 보여줌.

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

식을 그림과 순서를 표시하면 다음과 같음



여기서 x는 앞서 멀티 헤드 어텐션의 결과로 나온 (seq_len, dmodel)
의 크기를 가지는 행렬을 말함.

1. f_1(Linear)

```
conv1 = nn.Conv1d(in_channels=d_hidn, out_channels=d_hidn * 4, kernel_size=1)
# (bs, d_hidn * 4, n_seq)
ff_1 = conv1(output.transpose(1, 2))
print("\nff_1")
print(ff_1.size())
```

입력에 비해 hidden dimension이 4배 커짐.

```
ff_1
torch.Size([2, 512, 8])
```

2. Activation (relu or gelu)

논문이 발표 될 당시는 relu를 사용하도록 되어 있었지만 이후 gelu를 사용할 때 더 성능이 좋다는 것이 발견되었습니다.

```
# active = F.relu
active = F.gelu
ff_2 = active(ff_1)
```

3. f_3 (Linear)

```
conv2 = nn.Conv1d(in_channels=d_hidn * 4, out_channels=d_hidn, kernel_size=1)
ff_3 = conv2(ff_2).transpose(1, 2)
print(ff_3.size())
```

입력과 동일한 shape으로 변경됨

```
ff_3
torch.Size([2, 8, 128])
```