Self Attention

1. Attention and Self-Attention

◉ **RNN이 해결하지 못한 문제** :

- 멀리 떨어진 벡터(예: 단어)간에 대한 문맥 정보가 약하다

  The **man** who once worked at a famous hotel **was** ,,,,,,

- 시퀀스 벡터들 사이의 문맥적 의미 차이를 구분하지 못한다

  모든 시퀀스 벡터들과의 관계를 나타내기 어렵다

- 동일한 시퀀스 표현에 대한 서로 다른 임베딩/관점(인도, 나는, ...)을 나타태지 못한다

  "인도 위 주차 금지"

  "인도 카레" ==> 3개의 다른 의미의 '인도'가 동일한 임베딩 벡터를 갖는다

  "인도 국민"

- 병렬처리가 불가능 (Lack of Parallelizability)

◉ **Motivation** :

1) 특정 **단어의 문장 내에서의 문맥(의미)**은 문장 내 존재하는 **다른 단어들과의 관계**에 따라 달라진다.

예)

나는 ==> 주변에 다른 단어가 없기 때문에 의미를 특정할 수 없다

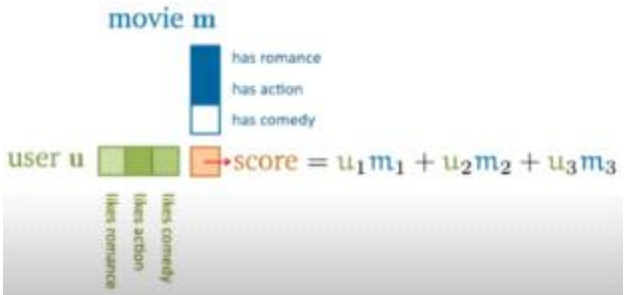
나는 어제 아침을 먹고 ..... (문장내 다른 단어들과의 유사도 계산을 반영하면,

하늘을 나는 자동차를 ... 서로 다른 '나는'에 대한 다른 의미(표현)을 구할 수 있다)

김이 나는 맛있는 만두를 사왔다

2) 벡터들의 연산 (예: dot-product)으로 단어의 문맥(context)을 계산할 수 있다.

영화 추천을 예를 들어, 어떤 user의 취향에 맞는 영화를 고를 때, 아래처럼 각 user와, 각 영화에 대한 정보가 벡터값으로 잘 표현되어 있으면 간단히 벡터의 유사성 계산 (dot-product)로도 적절한 영화를 추천할 수 있다.

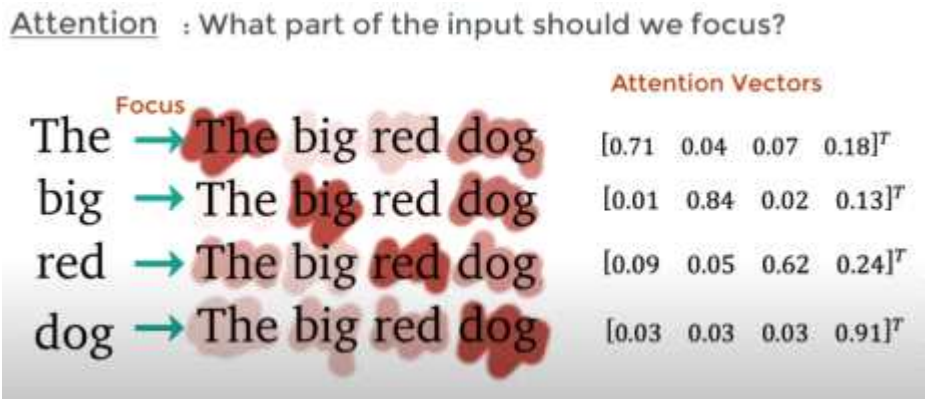


◉ **Attention**: 입력 시퀀스 내 다양한 위치들에 다른 가중치를 부여하여, 현재 생성하려는 출력(단어나 토큰)과 가장 관련이 높은 입력 위치에 주목(attention)하도록 하는 것.

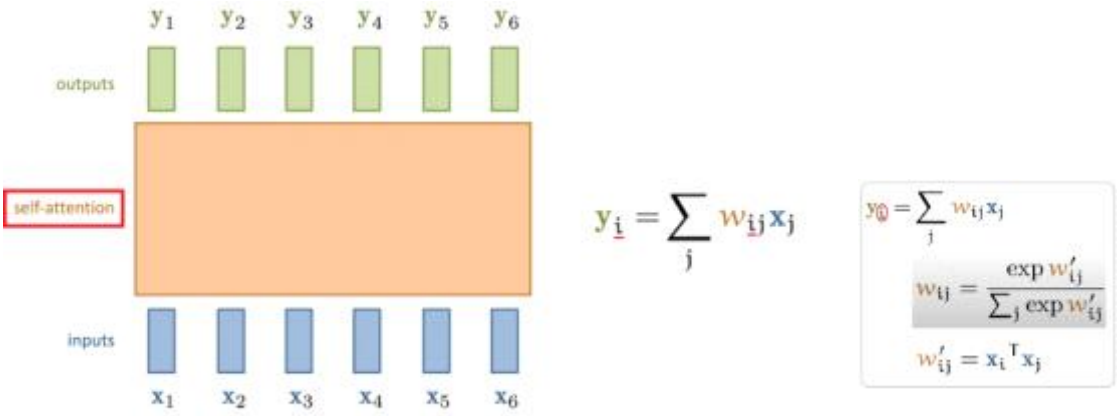
- 사용목적: Sequence-to-Sequence model에서 번역, 요약 QnA 시스템 개발 시 사용.
- 방법: 예를 들어, 번역시에 출력 단어와 동일한 의미의 입력 단어에 더 많은 가중치를 부여

◉ **Self-Attention (Intra-Attention)** : 동일한 시퀀스 내 다양한 위치 벡터들 사이의 상호작용을 모델링 한다.

- 사용목적: 문장내에서 각 단어의 문맥적 의미를 더 잘 이해하기 위함.
- 방법: 각 입력 위치에서 전체 입력에 대해 가중치를 계산하여,
  각 단어가 문장 내의 다른 단어들과 어떻게 상호 작용하는지를 모델링 한다



(https://www.youtube.com/watch?v=rPFkX5fJdRY)



X1 벡터를 예를 들면, X_1 벡터가 나머지 X_2 ~ X_k 벡터 각각과의 관계(similarity)를 가중치로 고려하여, X_1 벡터에 대한 새로운 벡터 표현 Y_1을 계산하는 것이다.

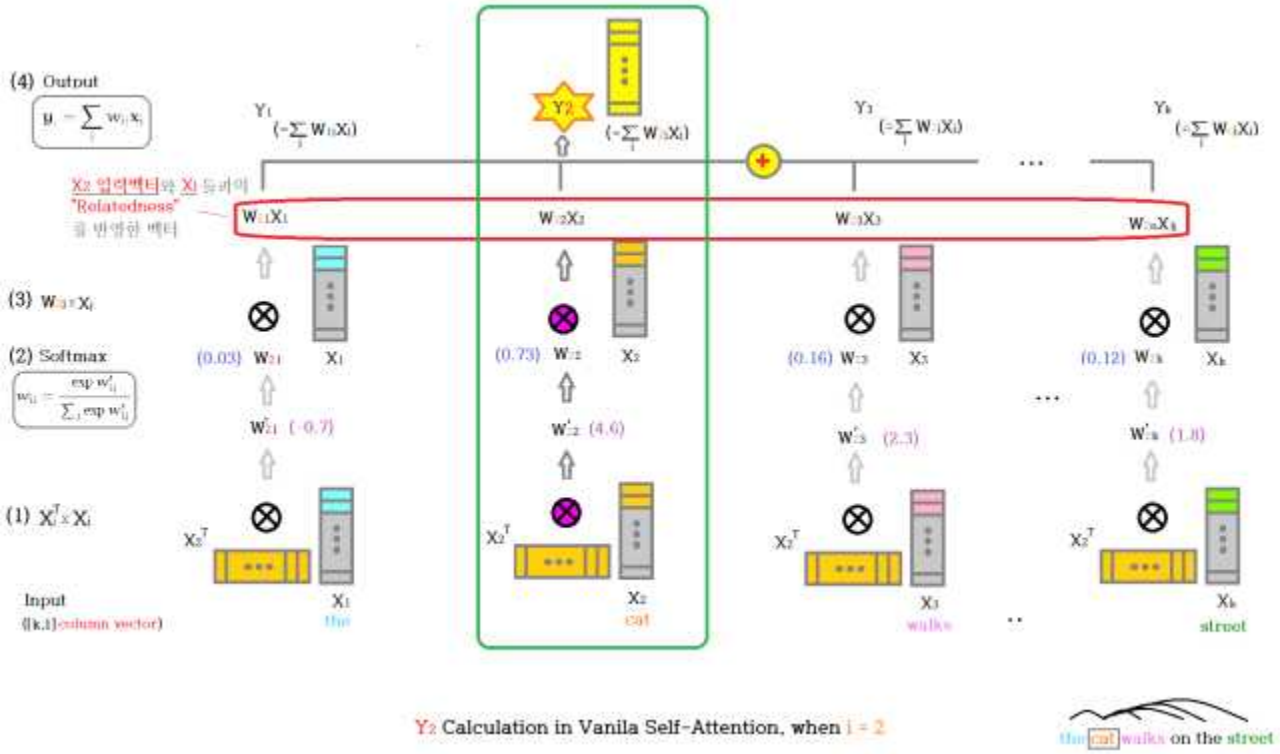
※ Attention과 Self-Attention의 차이점 : **Attention**은 입력 시퀀스와 출력 시퀀스 사이의 관계에 주목하며, 주로 다른 입력 부분에 대한 출력 부분의 의존성을 모델링하는 데 사용되고, **Self-Attention**은 단일 시퀀스 내의 요소들 사이의 관계를 모델링하며, 각 요소가 시퀀스 내의 다른 요소들과 어떻게 상호작용하는지를 학습한다.

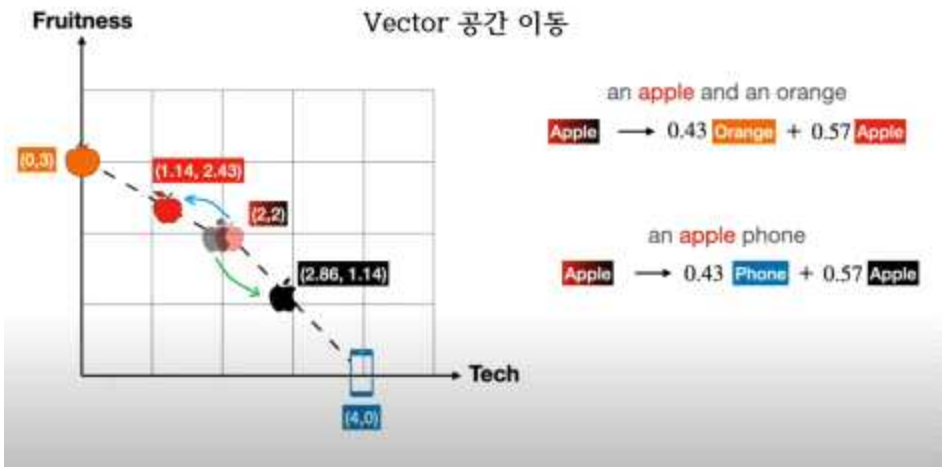
2. Vanilla Self-Attention

◉ Basic Self-Attention 개념도

- 입력벡터(시퀀스)들의 출력값을, 자신을 포함한 전체 입력 시퀀스들의 weighted sum으로 표현
- 입력 벡터값(Xi)과 나머지 벡터값들간의 벡터간 ·(dot product) 결과값들의 확률값(Wij)으로 구한 후, 각 벡터별로 Wij*Xj 로 재계산한 값을 모두 더한 값을 각 입력 Xi 의 Attention Score로 계산
- 즉, 입력 시퀀스 상의 다른 벡터들에 대한 가중치를 고려한 새로운 시퀀스 벡터값을 구함

(https://peterbloem.nl/blog/transformers)





(https://www.youtube.com/watch?v=UPtG_38Oq8o)

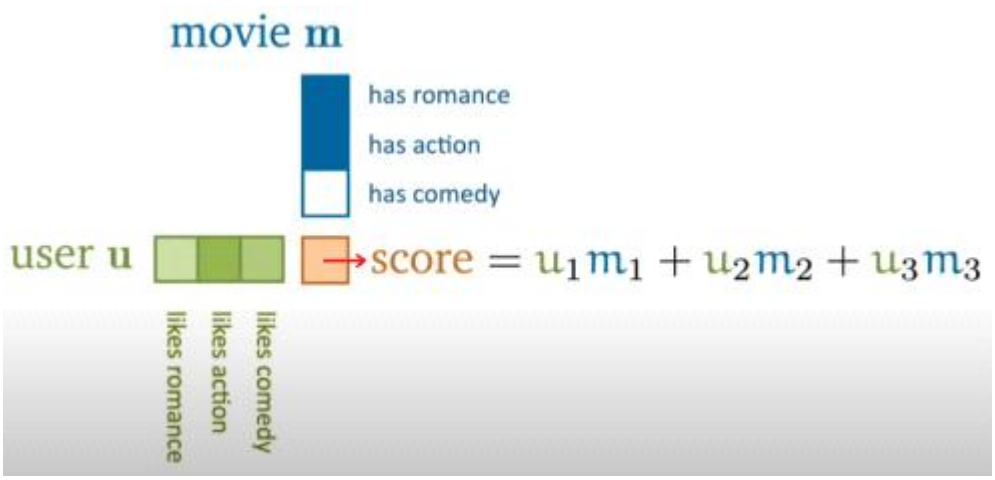
◉ Basic Self-Attention의 특징

- Weights를 학습하는 것이 아니라, 입력 벡터들 사이의 Similarity로만 계산한다

- 입력 벡터들을 집합(set)으로 간주한다. (입력 벡터 간의 sequence(순서) 정보(구분)이 없다.)
  (즉, 입력 X_i의 위치가 바뀌어도 입력값 X_i에 대한 출력값 Y_i는 동일하다.)
  "The cat walks on the street." "The cat on the street walks."

- Permutation Equivalent : p(sa(X)) = sa(p(X))
  "self-attention"과 "입력 벡터의 순서 변환" 간의 선후가 없다. 무엇을 먼저 해도 결과가 같다

- Word2Vec 임베딩에서는 주변 몇 개 단어의 존재를 기준으로 임베딩을 했으나,
  Aelf-Attention에서는 (1) 단른 단어와의 주목도(attention degree)에 따라, (2) 주변 뿐만 아니라 문장내 모든 단어와의 관계를 반영하기 때문에, Word2Vec 방법들보다 표현이 정교해진다고 할 수 있다



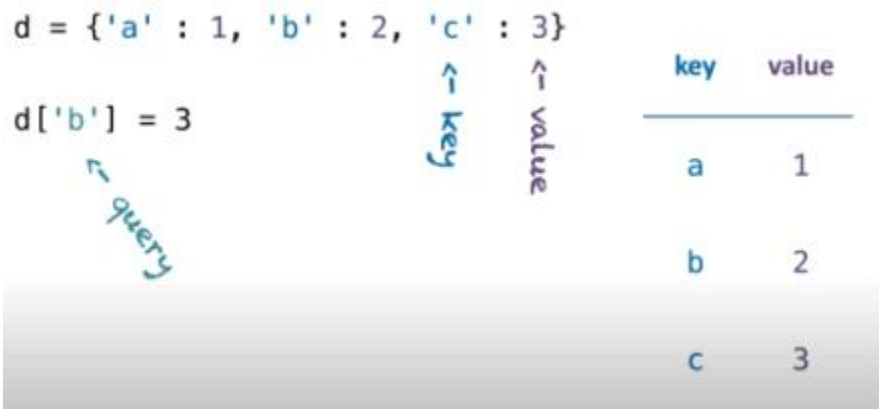
3. query, key, value가 추가된, 학습 가능한 Self-Attention

◉ DB 개념을 차용한 "Soft Dictionary"로서의 Self-Attention의 특징

- query, key, value 값들은 vector값이다
  입력 벡터의 3가지 서로 다른 역할(쓰임)에 따라 query, key, value로 표현

- 모든 key 값들이 query 값과 "어느정도씩" 매치된다(상관있다)고 본다 (dot-product 값으로)

- return 값은 "모든 value값들이 mix된 값"이다



(https://www.youtube.com/watch?v=KmAISyVvE1Y)

◉ q, k, v를 이용한 Self-Attention : 입력 벡터간의 상관관계(Similarity)를 이용하여, 특정 입력 시퀀스(X_i, q_i, query)에 대하여 다른 입력 시퀀스값들(X_j, k_j, keys)과의 관계를 확률로 반영한(**weighted**), 시퀀스들의 새로운 출력 시퀀스값들(v_j, values)의 **sum**.

$$\text{Self\_Attention}(q_i,k,v) = \sum_j \text{similarity}(q_i, k_j) * v_j$$

즉, 입력 시퀀스 X_i(혹은 q_i) 대한 새로운 X_i 값인 V_i들의 weighted sum 계산

◉ query와 key값들 간의 유사도(Similarity) 계산 방법들

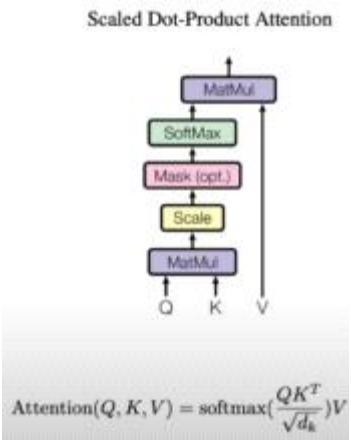
similarity(q, k_j) :

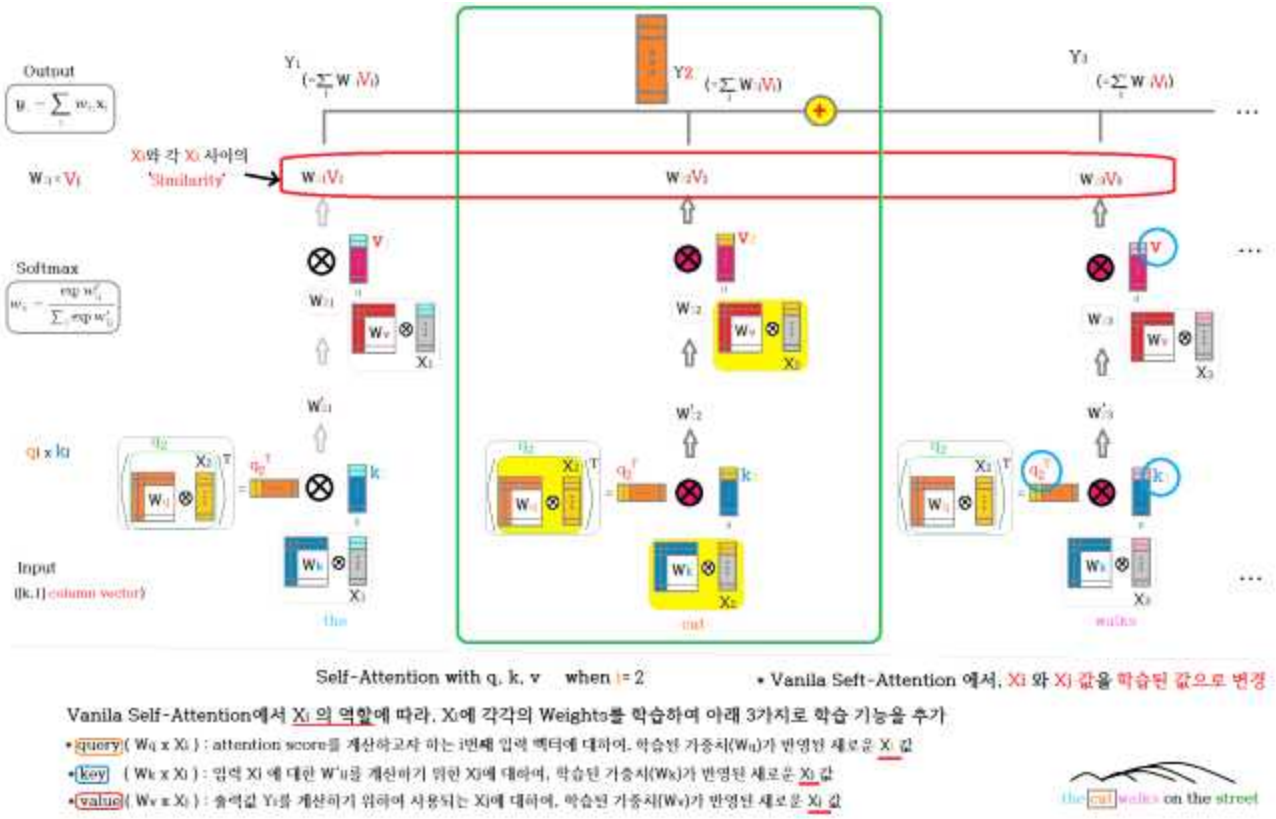
(1) $q^T * k_j$ dot product

(2) $(q^T * k_j) / \sqrt{\dim(k)}$ scaled dot product --->

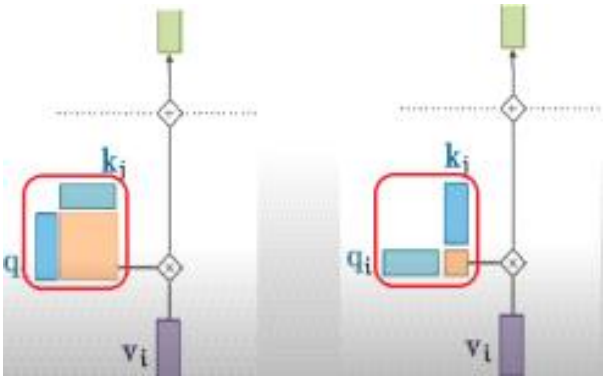
(3) $q^T * (W * k_j)$ general dot product

(4) $(W_q^T * q) + (W_k^T * k_j)$ additive similarity





※ 사용하는 유사도 계산 방법과 입력벡터의 표현([k x 1], [1 x k])에 따라 에 따라 q, k의 similarity 값은 scalar 일수도, matrix 일수도 있다

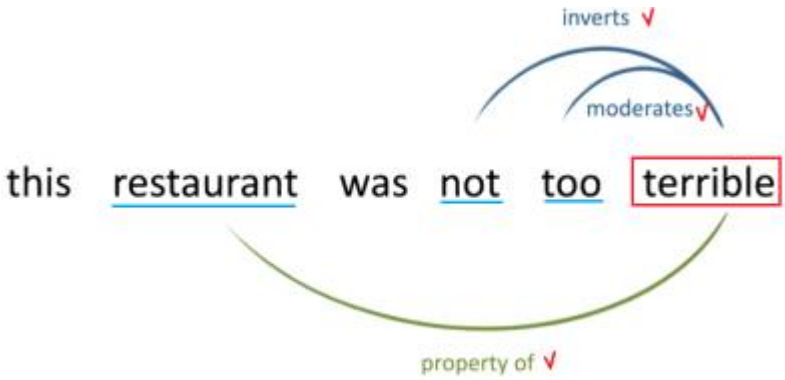


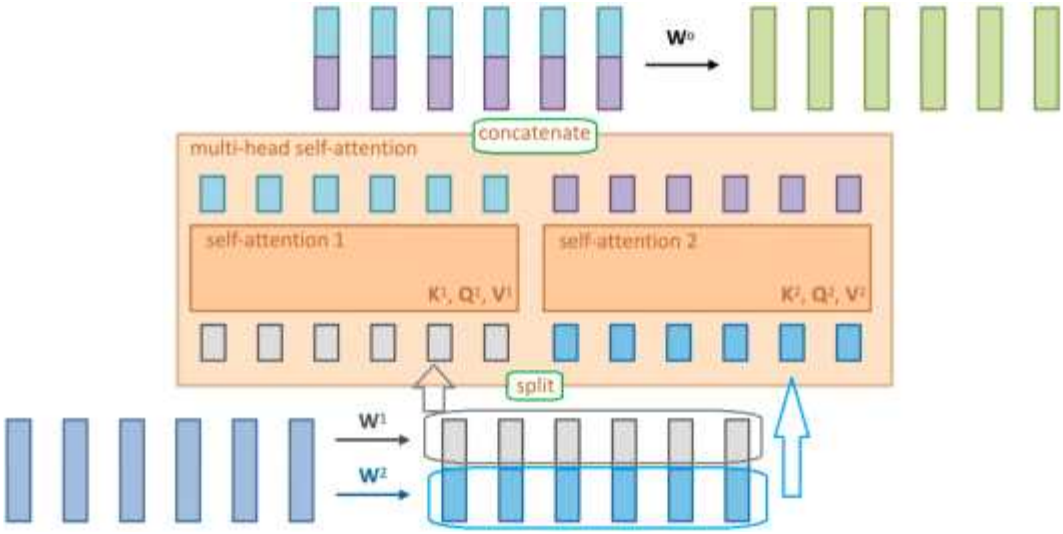
※ q, k, v 계산에
bias 값도 사용할 수 있다.

$$k_i = Kx_i + b_k$$
$$q_i = Qx_i + b_q$$
$$v_i = Vx_i + b_v$$

◉ Multi-head Attention

- 어떤 단어가 다른 단어와 서로 다른 관계(relation)으로 연관되어 있다는 점(perspective)을 나타내기 위해, head를 여러개로 나눈 것
- 적절한 헤드 수는 실험과 성능평가로 경험적으로 찾는다





◉ Self-Attention with q,k,v의 특징

- Long-term Memory 로서 가능
- 병렬처리 가능
- RNN과 마찬가지로 여전히 입력 벡터간 Sequential 정보의 구분이 불가능
  (positional embedding이 필요)

4. Multi-head Self-Attention (with q, k, v) Source Code 및 실행 결과

4.1 Source Code

(Similarity 함수 : outer product 사용)

```
# Vanilla Self Attention
# conda activate word2vec
# python self_attention.py
importtorch
importtorch.nn.functional as F
import numpy as np
import math
#=====================================================
# Input Target batch 데이터 생성 함수
# #=====================================================
def make_batch(sentences,forTrain=True):
  input_batch =[]

  if forTrain :
    target_batch =[]
    for sen in sentences:
      words =sen.split()
      inputNdx =[wordDictionary[n]for n in words[:-1]]
      target =wordDictionary[words[-1]]
          # input번째 자리만 1이 되게 One-Hot Encoding
          # input list 가 2개 값만 가지면 아래는 2개의 one-hot vectors 생성
      input_batch.append(np.eye(noOfClasses)[inputNdx])

      target_batch.append(target)

    return input_batch,target_batch

  else :# for Prediction
    sentence =sentences

    words =sentence.split()
    #print(f'words: {words}')
    inputNdx =[wordDictionary[n]for n in words[:-1]]
    input_batch.append(np.eye(noOfClasses)[inputNdx])  # One-Hot Encoding

    return input_batch
#=====================================================
# Self-Attention Class
# #=====================================================
class SelfAttention(torch.nn.Module):
    def __init__(self,embeddingSize,headSize):
        super(SelfAttention,self).__init__()
        self.embeddingSize =embeddingSize
        self.headSize =headSize
        self.headDimension =embeddingSize //headSize
        assert (
            self.headDimension *headSize ==embeddingSize
        ),"Embedding size needs to be divisible by headSize"
        #  query, key, value를 scalar값이 아닌 2차원 행렬 형태로 처리
        # 그래서 이 샘플 코드에서는 2차원 weighr표현에 좋은 nn.Linear 형식을 사용
        # 만약, outer product 가 아닌 dot product를 할 경우엔 vector가 필요함.
        self.values =torch.nn.Linear(self.headDimension,self.headDimension,bias=False)
        print(f'\nvalues 벡터 입출력 크기: {self.values}')
        self.keys =torch.nn.Linear(self.headDimension,self.headDimension,bias=False)
```

```
        self.queries =torch.nn.Linear(self.headDimension,self.headDimension,bias=False)
        self.fc_out =torch.nn.Linear(headSize *self.headDimension,embeddingSize)
    def forward(self,valuesBatch,keysBatch,queriesBatch):
        batchSize =queriesBatch.shape[0]   # sentence 갯수와 동일 => 12
                                                        valueLength,keyLength,queryLength
=valuesBatch.shape[1],keysBatch.shape[1],queriesBatch.shape[1]
        print(f'\nbatchSize: {batchSize}\n[query, key, value] 의 시퀀스 길이: \
[{valueLength}, {keyLength}, {queryLength}] \n')
        # Split the embedding into self.headSize different pieces
                                                                queriesBatchMultiHead
=queriesBatch.reshape(batchSize,queryLength,self.headSize,self.headDimension)
                                                                  keysBatchMultiHead
=keysBatch.reshape(batchSize,keyLength,self.headSize,self.headDimension)
                                                                valuesBatchMultiHead
=valuesBatch.reshape(batchSize,valueLength,self.headSize,self.headDimension)
        queriesBatch =self.queries(queriesBatchMultiHead)
        keysBatch =self.keys(keysBatchMultiHead)
        valuesBatch =self.values(valuesBatchMultiHead)
        # q * t 를 하여 similarity(attention 값)을 구함,
        # simililarity 계산법은 정하기 나름인데, 여기서는 outer product(행렬곱)를 수행하므로
        # attention 결과는 4차원 tensor [batch, head, queryDimension, keyDimension]
        attention =torch.einsum("nqhd,nkhd->nhqk",[queriesBatch,keysBatch])
        # queries shape: (N, queryLength, headSize, head_dim),
        # keysBatch shape: (N, keyLength, headSize, head_dim),
        # attention: (N, headSize, queryLength, keyLength)
        # Scale attention scores
        attention =attention /(self.embeddingSize **(1/2))
        attention =F.softmax(attention,dim=-1)
        # 방금 계산한 attention과 values를 곱하여 출력값 계산,
        # 그 후 embeddingSize 사이즈를 갖도록 한 차원 축소된 shape 변경
        out =torch.einsum("nhql,nlhd->nqhd",[attention,valuesBatch]).reshape(
            batchSize,queryLength,self.headSize *self.headDimension
        )
        # attention shape: (N, headSize, queryLength, keyLength)
        # valuesBatch shape: (N, valueLength, headSize, head_dim)
          # after einsum: (N, queryLength, headSize, head_dim) then we flatten the last two
dimensions

        out =self.fc_out(out)  # (N, queryLength, embeddingSize) 크기로 리턴
        print(f'attention 후 출력 y: {out}')
        return out
#=============================================================================
sentences =["i like my dogs","i love hot coffee","i hate cold milk","i enjoy hot tea",
  "you like cute cats","you love hot milk","you hate cold coffee","you want rainy days",
   "he expects big promotion","he wants nice gifts","he wants my dogs","i expect rainy days"]
dtype =torch.float
wordList =list(set("".join(sentences).split()))
wordList.sort()    # 매 실행시마다 동일한 word index 번호를 갖게
wordDictionary ={w:i for i,w in enumerate(wordList)}
print('\n','='*80)
print("word dictionary:",wordDictionary,'\n')
number_dict ={i:w for i,w in enumerate(wordList)}
windowSize =3
headSize =4
noOfClasses =int(math.ceil(len(wordDictionary)/headSize)*headSize)
embeddingSize =noOfClasses
print(f'\nheadSize: {headSize}, noOfClasses: {noOfClasses}')
# make one-hot encoded input batch
input_batch,target_batch =make_batch(sentences,True)
valueLength,keyLength,queryLength =windowSize,windowSize,windowSize # sequence lengths => 3
queries =torch.tensor(np.array(input_batch),dtype=torch.float32)
```

```
keys =torch.tensor(np.array(input_batch),dtype=torch.float32)
values =torch.tensor(np.array(input_batch),dtype=torch.float32)
print('\n','='*80)
print(f"Self-Attention용 [{len(queries)}]개 input batch:",input_batch)
attention =SelfAttention(embeddingSize,headSize)
# Self-attention(Forward Pass only) 실행
out =attention(values,keys,queries)
print(f'\noutput의   shape[batchSize,   sequenceLength,   VectorDimension]:   {out.shape}\n')#
Expected shape: (batchSize, sequenceLength,embeddingSize)
print('첫 번째 batch의 앞 쪽 2개의 시퀀스에 대한 attention 결과값 출력해보기')
for i in range(2):
    print('-'*80)
    for j in range(out.shape[1]):
        print(f'sequence #[{i}][{j}] {sentences[i].split()[j]}\n')
        for k in range(out.shape[2]):
            print(out[i,j,k].item(),end='')
        print('\n')
    print('\n')
```

4.2 실행결과

```
(word2vec) D:\myprojects\nlp\transformer>python self_attention.py

 ================================================================================
word dictionary: {'big': 0, 'cats': 1, 'coffee': 2, 'cold': 3, 'cute': 4, 'days': 5, 'dogs': 6, 'enjoy': 7, 'expect': 8, 'expects': 9, 'gifts': 10, 'hate': 11, 'he': 12, 'hot': 13, 'i': 14, 'like': 15, 'love': 16, 'milk': 17, 'my': 18, 'nice': 19, 'promotion': 20, 'rainy': 21, 'tea': 22, 'want': 23, 'wants': 24, 'you': 25}


headSize: 4, noOfClasses: 28

 ================================================================================
Self-Attention용 [12]개 input batch: [array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 1., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
```

```
          0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]]), array([[0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]]), array([[0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]]), array([[0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]]), array([[0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
0.,  1.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
         [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]])]

values 벡터 입출력 크기: Linear(in_features=7, out_features=7, bias=False)    <-- head의_dimension 크기

batchSize: 12
 [query, key, value] 의 시퀀스 길이:                    [3, 3, 3]

attention 후 출력 y: tensor([[[ 0.1151,  0.0031,  0.1608,  ...,  0.0824,  0.0418,  0.0665],
          [ 0.1152,  0.0033,  0.1608,  ...,  0.0824,  0.0419,  0.0665],
          [ 0.1146,  0.0010,  0.1630,  ...,  0.0820,  0.0410,  0.0663]],

         [[ 0.0980,  0.0778,  0.1124,  ...,  0.1037,  0.0180,  0.0642],
          [ 0.0979,  0.0783,  0.1124,  ...,  0.1035,  0.0186,  0.0641],
          [ 0.0985,  0.0776,  0.1116,  ...,  0.1037,  0.0184,  0.0645]],

         [[ 0.0416,  0.0694,  0.0984,  ...,  0.1488,  0.0621,  0.0707],
          [ 0.0400,  0.0696,  0.0984,  ...,  0.1500,  0.0618,  0.0705],
          [ 0.0419,  0.0697,  0.0979,  ...,  0.1490,  0.0623,  0.0707]],

         ...,                                                                <-- 12개 문장내 각 입력벡터 3개에 대한
                                                                                 28길이의 새로운 벡터값 들
         [[ 0.0916,  0.0861,  0.1129,  ...,  0.0547,  0.0790,  0.0661],
          [ 0.0908,  0.0860,  0.1127,  ...,  0.0551,  0.0783,  0.0657],
          [ 0.0903,  0.0868,  0.1125,  ...,  0.0555,  0.0787,  0.0652]],

         [[ 0.1252, -0.0109,  0.1545,  ...,  0.0399,  0.0405,  0.0862],
          [ 0.1244, -0.0110,  0.1543,  ...,  0.0403,  0.0397,  0.0857],
          [ 0.1248, -0.0126,  0.1551,  ...,  0.0403,  0.0392,  0.0857]],

         [[ 0.0911,  0.1277,  0.1058,  ...,  0.0807,  0.0874,  0.0328],
          [ 0.0915,  0.1278,  0.1056,  ...,  0.0805,  0.0874,  0.0330],
          [ 0.0914,  0.1274,  0.1053,  ...,  0.0812,  0.0873,  0.0331]]],
        grad_fn=<ViewBackward0>)

output의 shape[batchSize, sequenceLength, VectorDimension]: torch.Size([12, 3, 28])

첫 번째 batch의 앞 쪽 2개의 시퀀스에 대한 attention 결과값 출력해보기
```

```
--------------------------------------------------------------------------------
sequence #[0][0] i

0.1151479035615921 0.003071088343858719 0.1608436107635498 -0.18090710043907166
-8.22991132736206e-05 -0.04824358597397804 0.029503142461180687 -0.02235504426062107
-0.1884058117866516 -0.04054579883813858 -0.23621192574501038 0.09467335045337677
-0.04143090918660164 -0.16817593574523926 0.07039027661085129 0.00943125318735838
0.05982724577188492 0.14963968098163605 0.09289621561765671 -0.147779181599617
0.0973832905292511 -0.09093663096427917 -0.02620057389140129 -0.027777407318353653
-0.23504075407981873 0.08242534846067429 0.04177071154117584 0.06649473309516907

sequence #[0][1] like

0.1151566207408905 0.003255315124988556 0.1607511043548584 -0.18078693747520447
5.2988529205322266e-05 -0.0480973944067955 0.029494956135749817 -0.022331546992063522
-0.18851447105407715 -0.040580540895462036 -0.23612827062606812 0.09478039294481277
-0.04152800887823105 -0.16815431416034698 0.07031996548175812 0.009330022148787975
0.05993542820215225 0.14946940541267395 0.09288780391216278 -0.14772804081439972
0.09712598472833633 -0.09088084101676941 -0.026220005005598068 -0.027843672782182693
-0.2350570112466812 0.08242741227149963 0.04186101630330086 0.06650124490261078

sequence #[0][2] my

0.11457754671573639 0.001003652811050415 0.16302751004695892 -0.18210723996162415
-0.0015854611992835999 -0.048895515501499176 0.02983040176331997 -0.020682843402028084
-0.18523751199245453 -0.03939506411552429 -0.23798206448554993 0.0946304053068161
-0.040980931371450424 -0.1697293519973755 0.07087674736976624 0.0108261089771986
0.058504246175289154 0.1512300968170166 0.09350307285785675 -0.146998330950737
0.0980750322341919 -0.09104740619659424 -0.02622838318347931 -0.027919843792915344
-0.23355022072792053 0.08204780519008636 0.04096003249287605 0.06625881046056747



--------------------------------------------------------------------------------
sequence #[1][0] I <--- 위의 'i'와 다른 벡터값을 가짐 (문맥이 달라진 것을 반영)

0.09795288741588593 0.07784555852413177 0.11239154636859894 -0.17067457735538483
0.029408935457468033 -0.066505067050457 -0.05726855993270874 -0.07056652754545212
-0.26700809597969055 -0.04871722310781479 -0.11723288148641586 0.14266632497310638
-0.05538448691368103 -0.1347925364971161 -0.0029483307152986526 -0.07901907712221146
0.08867290616035461 0.08268306404352188 0.14193429052829742 -0.18763203918933868
0.04427891597151756 -0.02977987378835678 -0.09084992110729218 -0.03940680995583534
-0.28412461280822754 0.10365089029073715 0.018046820536255836 0.06420382112264633

sequence #[1][1] love

0.09794250875711441 0.07827898859977722 0.11241792142391205 -0.17051614820957184
0.02984008938074112 -0.06622977554798126 -0.0568564236164093 -0.07078677415847778
-0.267234206199646 -0.048968516290187836 -0.11726242303848267 0.14250367879867554
-0.055753376334905624 -0.1346895396709442 -0.002770945429801941 -0.07874681055545807
0.0890367329120636 0.08274555206298828 0.14176948368549347 -0.18782582879066467
0.04427953436970711 -0.030234798789024353 -0.0904364287853241 -0.03915885463356972
-0.28411322832107544 0.10348830372095108 0.018637338653206825 0.064115509390831

sequence #[1][2] hot

0.09845006465911865 0.07763826847076416 0.11164907366037369 -0.17027315497398376
0.029730908572673798 -0.06608322262763977 -0.0564747154712677 -0.07055613398551941
-0.2673642635345459 -0.04917823523283005 -0.11775822192430496 0.14215044677257538
-0.05509407818317413 -0.1344359815120697 -0.002571757882833481 -0.07871833443641663
0.08869903534650803 0.08271241188049316 0.14082464575767517 -0.187430739402771
```

```
0.04462708160281181     -0.030377306044101715     -0.09027256071567535     -0.03953283280134201
-0.28399696946144104 0.10373720526695251 0.018373597413301468 0.06450062245130539

(word2vec) D:\myprojects\nlp\transformer>
```

※ [Questions]

1. Self-Attention과 RNN의 차이점에 대하여 설명하시오?

2 self attention에서 query, key, vector 개념을 사용하는 이유가 무엇인가?

3. 벡터간의 similarity를 계산하는 다양한 방법에 대하여 설명하시오

4. Scaled dot product에 대하여 설명하시오

5 이 장에서 설명한 selt-attention은 학습 기능이 있는가?

6. Pytorch의 nn.Linear() 함수의 동작 방법에 대하여 설명하시오

7. Self-Attentioneh 결국 입력 벡터에 대한 새로운 벡터표현을 구하는 것이니 만큼, CBOW와 같은 Word2Vec 알고리즘으로 사용해되 딜 것이다. CBOW와 Aelf-Attention의 성능적 관점에서의 차이점에 대하여 설명하라

※ [실습 문제]

8. 이 장의 소스코드(self_attention.py)에서 입력 시퀀스의 길이를 4로, 한 개의 입력 벡터 길이를 512로 변경하여 실행하시오