Homework #11

```
n = 4
dim = 2

def get_angles(pos, i, dim):
  angles = 1/math.pow(10000,(2*(i//2))/dim)
  return pos * angles

def get_positional_encoding(pos,i,dim):
  if i%2==0:
    return math.sin(get_angles(pos,i,dim))
  return math.cos(get_angles(pos,i,dim))

result = [[0] * dim for _ in range(n)]

for i in range(n):
  for j in range(dim):
    result[i][j] = get_positional_encoding(i,j,dim)

import numpy as np
representation = np.array([[-1,3],[2,-2],[2,1],[1,2]])
positional_embedding = np.array(result)

representation + positional_embedding
```

```
array([[-1.        ,  4.        ],
       [ 2.84147098, -1.45969769],
       [ 2.90929743,  0.58385316],
       [ 1.14112001,  1.0100075 ]])
```

*ipynb 참고

1. '봄', '여름', '가을', '겨울'에 대한 representation 값을 각각 (1,1), (2,1), (2,1)이라고 했을 때, 아래의 순서에 따른 positional encoding 값을 더한 representation 값을 구하시오. Transformer의 positional encoding을 참고하시오.
   a. ['여름', '가을', '겨울', '봄']

pos: 단어의 순서

K: 벡터

'봄': (1.141, 1.01) '가을': (2.841, -1.46)

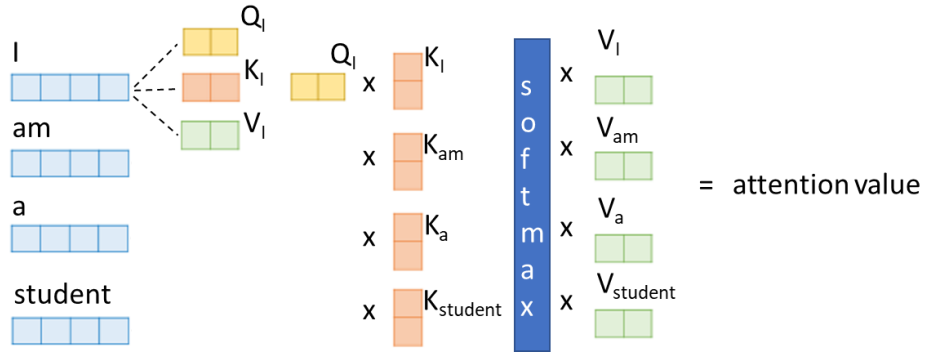
'여름': (-1, 4) '겨울': (1.141, 1.01)

2. Transformer 에서 self-attention을 사용하는 목적 혹은 무엇을 얻고자 사용하는지 이유를 서술하시오.

입력 데이터를 내적하는 것을 통하여, 각 입력 벡터를 입력데이터의 벡터들로 표현할 수 있기 때문이다. 이러한 re-description을 통하여 멀리 떨어진 단어들과의 유사도를 계산할 수 있으며, (long term dependency 구하기), 추후 연산을 위해 각 벡터를 표준화할 수 있다.

*ipynb 참고

3. Transformer 모델에서 vector size = 3 이다.
   [I , am , a , student] 를 Q, K , V 로 한 결과가 아래와 같을 때 Q_I에 대한 attention 출력값을 구하시오.
   - Q_I = [2, 4, 6]
   - K_I = [2, 4, 6], K_{am}=[1, 2, 4], K_a=[0, 1, 2], $K_{student}$=[1, 1, 1]
   - V_I = [0.5, 0.3, 0.2], V_{am}=[0.6, 0.1, 0.3], V_a=[0.1, 0.7, 0.2], $V_{student}$=[0.4, 0.5, 0.1]



$$A(Q,K,V) = \text{softmax}\left(Q_I \cdot \begin{bmatrix} K_I \\ K_{am} \\ K_a \\ K_{student} \end{bmatrix}^T \times \frac{1}{\sqrt{3}}\right) \cdot \begin{bmatrix} V_I \\ V_{am} \\ V_a \\ V_{student} \end{bmatrix} = [0.5, 0.3, 0.2]$$

```
Q = torch.FloatTensor([2,4,6])
Kt = torch.FloatTensor([[2,1,0,1],[4,2,1,1],[6,4,2,1]])
V = torch.FloatTensor([[0.5,0.3,0.2],[0.6,0.1,0.3],[0.1,0.7,0.2],[0.4,0.5,0.1]])
embedding_d = 3
QKt = torch.matmul(Q,Kt)
ad_QKt = QKt/math.sqrt(embedding_d)

expQKt = torch.exp(ad_QKt)
sum_expQKt = torch.sum(expQKt)
softmax = expQKt/sum_expQKt

torch.matmul(softmax,V)
```

```
tensor([0.5000, 0.3000, 0.2000])
```