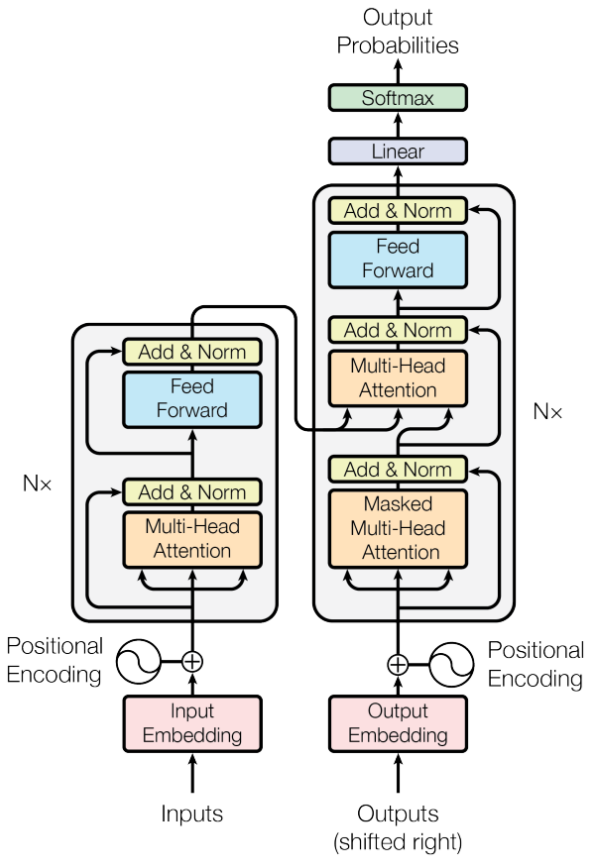
어텐션(Attention)이란 문맥에 따라 집중할 단어를 결정하는 방식을 의미한다.



transformer 구조

transformer는 encoder, decoder가 이루어진 형태로 encoder에 사용하는 attention과 decoder에 사용하는 attention이 조금 다르다.

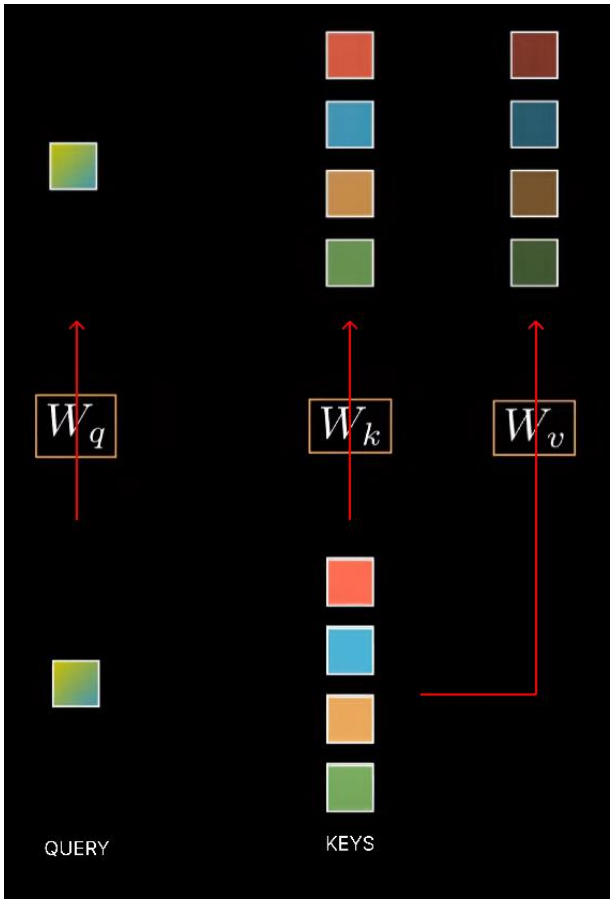
논문에서 존재하는 attention은 multi-head attention, masked multi-head attention이 존재한다.

먼저 논문에선 Scaled Dot-Product Attention라고 쓰인 기본적인 attention에 대해 알아보면,

1.

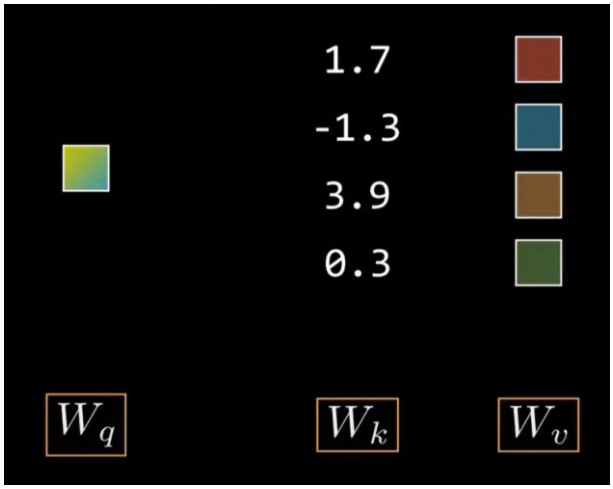
query와 keys들을 각각 weight에 통과시킨다.

weight를 통과해도 각각의 값들의 정보는 변경되지 않는다.



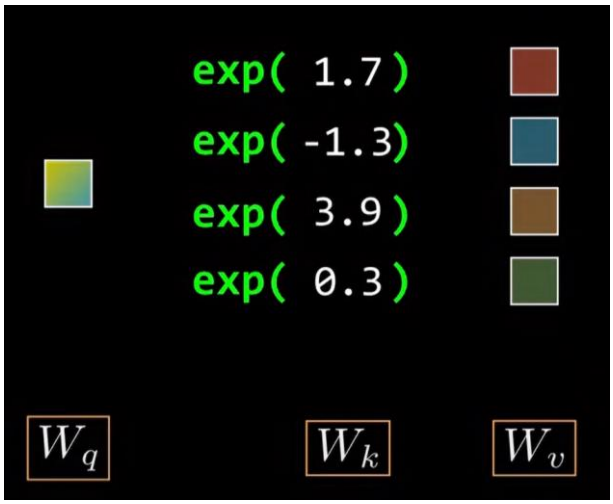
2.

query와 key를 내적 한다. query와 key가 유사하다면 내적 값이 커지며 유사하지 않다면 내적 값이 작아진다.



3.

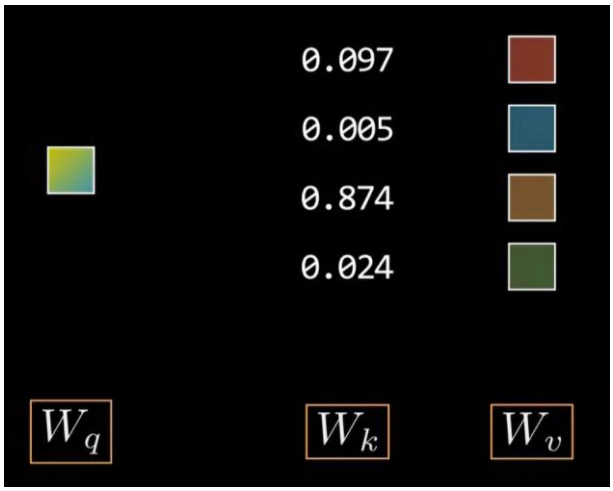
내적 값을 exponential함수에 넣어 모든 값을 양수로, 유사한 값이 큰 값이 되도록 하는 성질을 유지하게 한다.



4.

softmax 함수에 넣어 모든 값의 합을 1로 만든다.

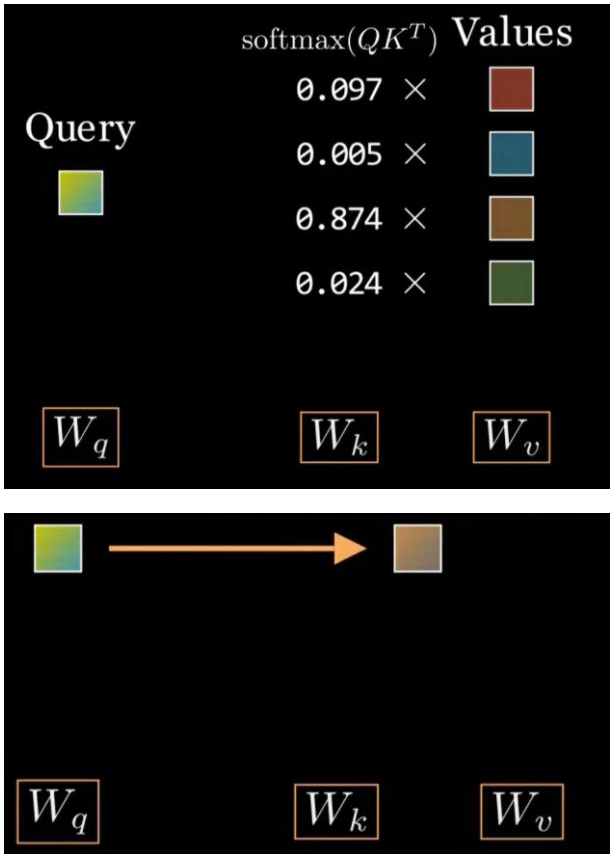
이 값은 쿼리와 키가 유사한 정도를 의미하게 된다.



5.

이후 계산한 keys값과 value값을 weighted sum을 하여 query랑 관련된 key의 혼합된 정보를 얻게 된다.

이 값이 attention의 기본적인 출력 결과이다.



self-attention은 attention 구조에서 자기 자신(sequence)가 input 값으로 들어간다.

self-attention은 동음이의어 같은 단어를 구분하기 위해 자기 자신도 attention의 입력값으로 들어간다.

1.

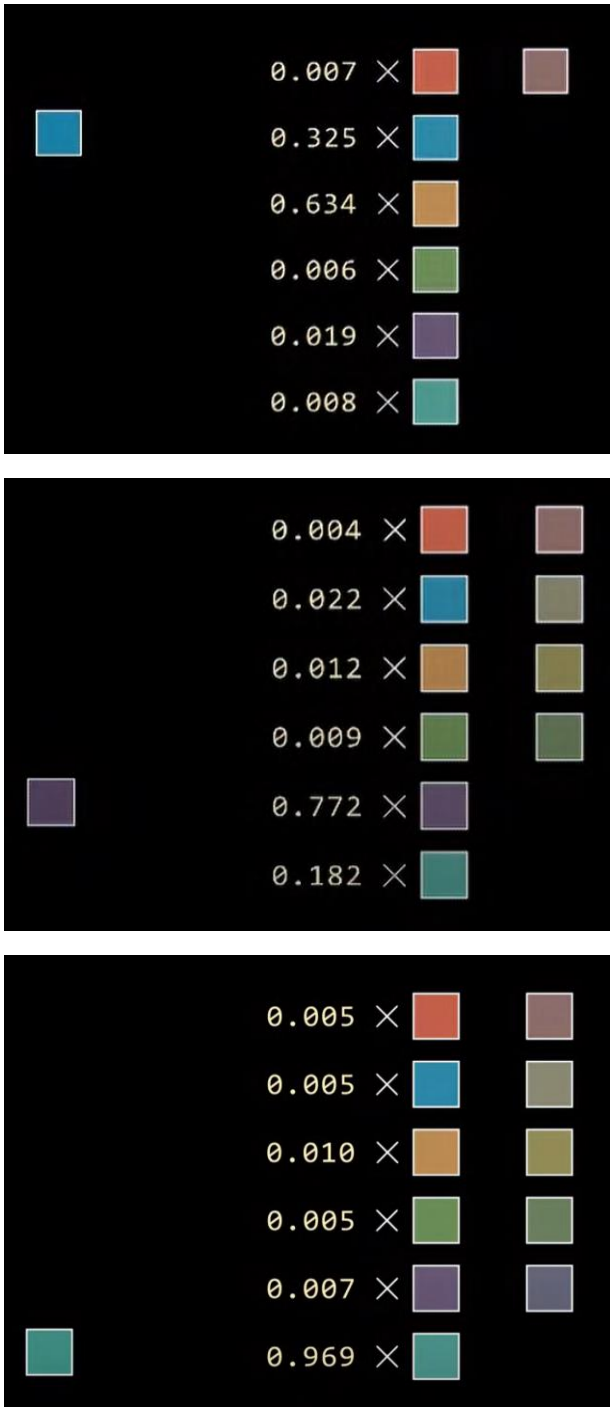
기본 attention 구조는 입력값이 query에 해당하는 단어와 key, value에 해당하는 문장이 들어가는데 self-attention은 문장이 query, key, value에 들어간다.

2.

이후 각각의 단어별로 모든 문장에 대한 attention 출력값을 구하면

해당 문장에 대한 attention 출력값이 나온다.

이 출력값을 다음 layer의 input 값으로 다시 사용한다.



self-attention의 출력값은 해당 단어와 다른 모든 단어들과의 관련된 정보를 포함하게 되어 맥락 정보를 얻을 수 있게 되었다.

이후 multi-head attention은 이 self-attention을 여러 개로 병렬 수행한 뒤 출력값을 concat 하여 얻는 방식이다.

decoder에 존재하는 masked multi-head attention은 multi-head attention에 masking 작업을 한 것이다.

다음 단어를 예측하는 모델이 다음 단어가 입력값에 있으면 안 되기 때문에 n 번째 단어를 예측한다면 n번째 단어부턴 masking하여 attention 구조에 넣는 방식이다.

multi head-attention의 코드 구조

```
1. import torch
2. import torch.nn as nn
3. import torch.nn.functional as F
4. import math
5.
6. class MultiHeadAttention(nn.Module):
7.     def __init__(self, d_model, num_heads):
8.         super().__init__()
9.         assert d_model % num_heads == 0, "d_model must be divisible by
   num_heads"
10.
11.         self.d_model = d_model
12.         self.num_heads = num_heads
13.         self.depth = d_model // num_heads
14.
15.         self.wq = nn.Linear(d_model, d_model)
16.         self.wk = nn.Linear(d_model, d_model)
17.         self.wv = nn.Linear(d_model, d_model)
18.         self.dense = nn.Linear(d_model, d_model)
19.
20.     def split_heads(self, x, batch_size):
21.         x = x.view(batch_size, -1, self.num_heads, self.depth)
22.         return x.permute(0, 2, 1, 3)
23.
24.     def forward(self, q, k, v):
25.         batch_size = q.shape[0]
26.
27.         q = self.split_heads(self.wq(q), batch_size)
28.         k = self.split_heads(self.wk(k), batch_size)
29.         v = self.split_heads(self.wv(v), batch_size)
30.
31.         scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.depth)
32.         attention_weights = F.softmax(scores, dim=-1)
33.
34.         context_vector = torch.matmul(attention_weights, v)
35.         context_vector = context_vector.permute(0, 2, 1, 3).contiguous()
36.         context_vector = context_vector.view(batch_size, -1, self.d_model)
37.
38.         output = self.dense(context_vector)
39.         return output, attention_weights
```