

Paper

- Attention is all you need

Vocab size

- 단어장 크기는 unique word의 수로 결정된다.

$$\text{vocab size} = \text{count}(\text{set}(N))$$

- N을 구하기 위해서 단어를 하나하나 Tokenize 해야 한다.

Encoding

- unique word에 고유의 index 수를 할당해야 한다.

1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game
13	14	15	16	17	18	19	20	21	22	23	
the	win	of	enemy	you	wait	thrones	and	or	die	He	

- 결과

- Word to index
- index to word

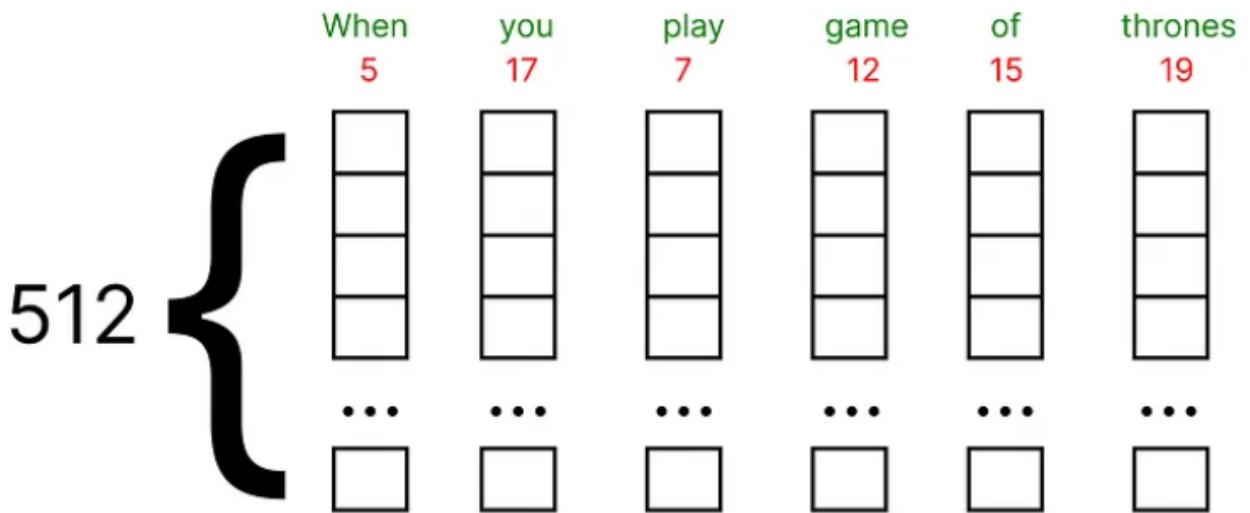
Calculating Embedding

1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game
13	14	15	16	17	18	19	20	21	22	23	
the	win	of	enemy	you	wait	thrones	and	or	die	He	



- 한 문장을 이루는 토큰이 input sequence로 들어갔다고 한다면, 토큰을 벡터로 embedding 해야 한다.
- 논문에서는 각 토큰 당 512 dimensional embedding vector를 사용했다.

Attention Is All You Need paper



- embedding vector 값은 0 과 1 사이의 값으로 구성되어 있고 처음에 랜덤한 값으로 채워진다.
- 이후에 모델이 학습하기 시작하면서 업데이트 된다. 그리고 트랜스포머 모델은 각 단어들 사이에 문맥을 이해하기 시작한다.

Caculating positional Embedding

- embedding matrix가 만들어지고 나서는 positional Embedding을 계산해야 한다.
- formulas
 - 각 위치에 따라 두 가지 공식으로 계산된다.

Embedding vector for any word

even position
odd position
even position
odd position
even position
...

For even position

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

For odd position

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

- 단어의 주기성을 주기 위함으로 바꿔서 해도 상관 차이가 많이 없었다.
- 문장에서 가장 첫 번째 토큰에 해당하는 단어의 positional embedding을 계산한다.

- 예시는 다음과 같다

When
5

i	e1	Position	Formula	p1
0	0.79	Even	$\sin(0/10000^{(2*0/6)})$	0
1	0.6	Odd	$\cos(0/10000^{(2*1/6)})$	1
2	0.96	Even	$\sin(0/10000^{(2*2/6)})$	0
3	0.64	Odd	$\cos(0/10000^{(2*3/6)})$	1
4	0.97	Even	$\sin(0/10000^{(2*4/6)})$	0
5	0.2	Odd	$\cos(0/10000^{(2*5/6)})$	1

d (dim) 6

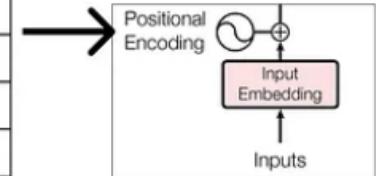
POS 0

- 이러한 과정을 모든 단어에 적용한다.

When	you	play	game	of	thrones
5	17	7	12	15	19

i	p1	p2	p3	p4	p5	p6
0	0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	1	0.0464	0.9957	0.1388	0.1846	0.9732
2	0	0.0022	0.0043	0.0065	0.0086	0.0108
3	1	0.0001	1	0.0003	0.0004	1
4	0	0	0	0	0	0
5	1	0	1	0	0	1

d (dim) 6 POS 0 1 2 3 4 5



Concatenating Positional and Word Embeddings

지금까지 Positional embedding과 Word Embedding을 구했다. 이후에는 word Embedding + Positional embedding을 한 matrix를 구해준다.

When	you	play	game	of	thrones
5	17	7	12	15	19

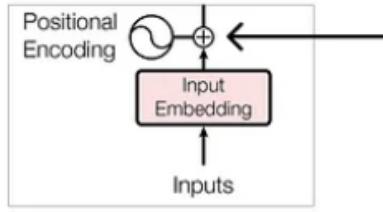
Position Embedding Matrix

p1	p2	p3	p4	p5	p6
0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	0.0464	0.9957	0.1388	0.1846	0.9732
0	0.0022	0.0043	0.0065	0.0086	0.0108
1	0.0001	1	0.0003	0.0004	1
0	0	0	0	0	0
1	0	1	0	0	1

+

Word Embedding Matrix

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

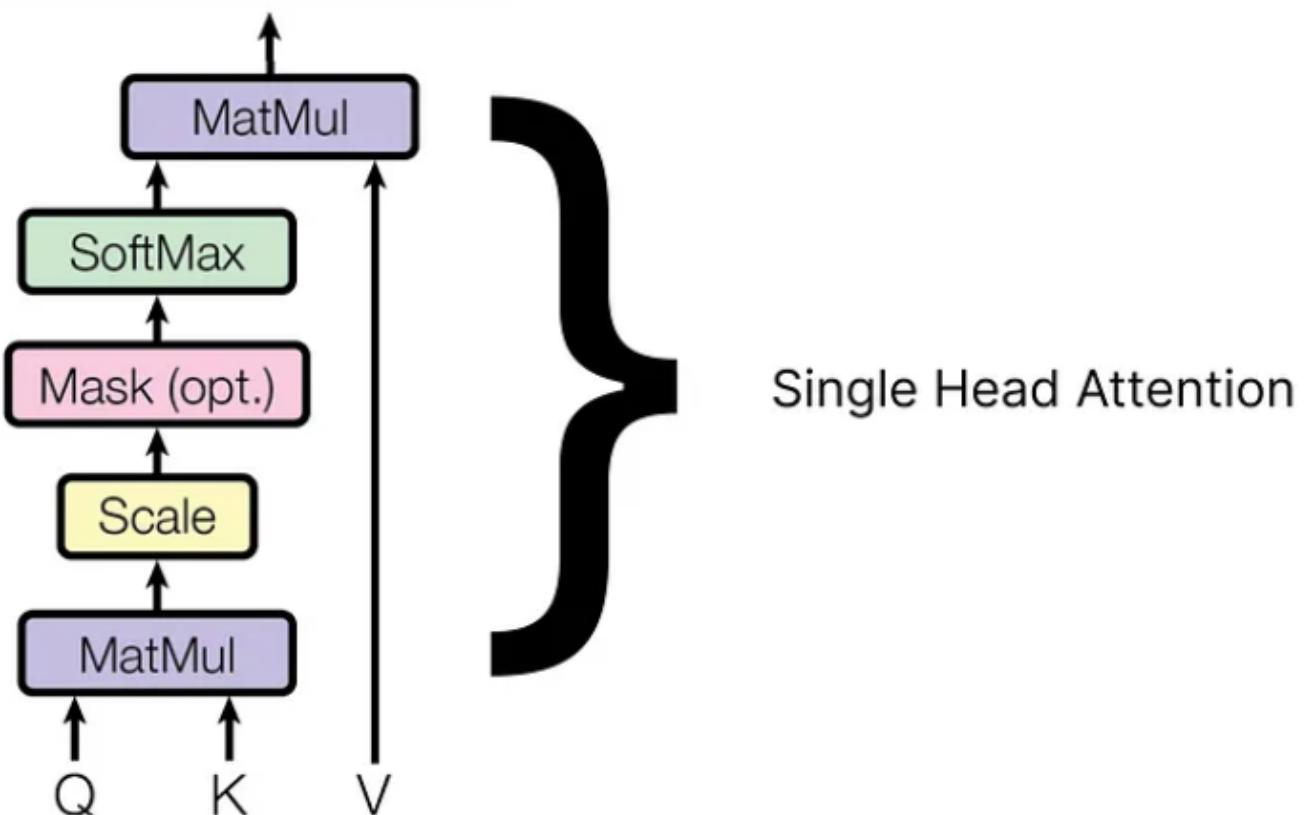


ep1	ep2	ep3	ep4	ep5	ep6
0.79	1.22	0.92	0.26	0.12	-0.36
1.6	0.17	1.51	0.74	0.59	1.3
0.96	0.06	0.27	0.66	0.8	0.76
1.64	0.79	1.31	0.22	0.62	1.48
0.97	0.9	0.56	0.07	0.5	0.94
1.2	0.74	1.59	0.37	0.7	1.21

Multi Head Attention

- multi head attention은 많은 단일 head attention으로 구성되어 있다.
- 단일 attention 얼마나 결합할지는 우리가 결정하게 된다.
 - LLaMA의 경우 32 single head를 사용한다.

Single head Attention



단일 head attention은 query, key, value로 구성되어 있고, 각 값에 weight가 다른 matrix가 곱해져서 각각 고유

한 상태가 있는 상태가 된다.

key, Query, value는 이전에 계산했던 임베딩 행렬의 transpose에 각각 곱해진다.

- **Query**

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6x6

Linear weights for query

0.52	0.45	0.91	0.69
0.05	0.85	0.37	0.83
0.49	0.1	0.56	0.61
0.71	0.64	0.4	0.14
0.76	0.27	0.92	0.67
0.85	0.56	0.57	0.07

6x4

- key
 - Value

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

Linear weights for key

0.74	0.57	0.21	0.73
0.55	0.16	0.9	0.17
0.25	0.74	0.8	0.98
0.8	0.73	0.2	0.31
0.37	0.96	0.42	0.08
0.28	0.41	0.87	0.86

6 x 4

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6x6

Linear weights for value

0.62	0.07	0.7	0.95
0.2	0.97	0.61	0.35
0.57	0.8	0.61	0.5
0.67	0.35	0.98	0.54
0.47	0.83	0.34	0.94
0.6	0.69	0.13	0.98

6x4

결과적으로 세가지 구성 값들은 모두 동일한 형상을 갖는다.

세 가지 서로 다른 가중치를 갖는 Query key value matrix를 구한 후에 각 single head attention에서 계산된다.

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

3.71	4.04	4.15	3.41
2.18	2.51	1.64	1.93
3.28	3.11	3.65	3.01
1.07	1.13	1.64	1.35
1.49	1.97	2.14	1.81
2.51	3.04	3.45	2.28

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6x4

6x4

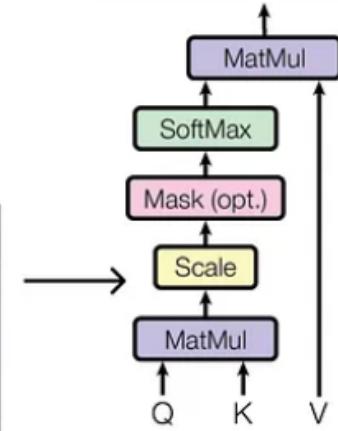
- Query와 key의 transpose 행렬을 곱해서 정방 행렬을 구성해 준다.
 - 구하려는 target Query에 대해서 input 문장에 있는 모든 단어에 대해서 얼마나 mapping이 되는지가 표현된 matrix가 되는 것 같다.

- key와 Query를 구하고 나서 결과 matrix를 scaling 해주기 위해서 차원 크기의 제곱근 만큼 나눠준다.

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265

$$\sqrt{d_k} \quad \text{where } d \text{ (dimension) is 6}$$

23.81721	12.77409	20.30219	8.062904	11.50852	17.62
14.1009	7.434201	12.09231	4.809165	6.781004	10.47973
20.77167	11.186	17.65266	6.976963	10.01433	15.39096
7.401542	3.972256	6.307436	2.535222	3.612997	5.466445
10.76551	5.752218	9.205999	3.64974	5.184753	7.956759
17.10152	9.254989	14.54997	5.715476	8.205791	12.62712

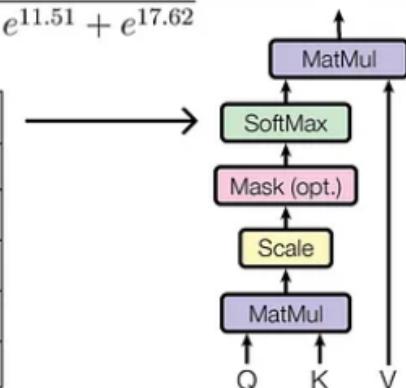


- 다음 층에서 구성되어 있는 masking은 option이다.
 - transformer 모델에서는 decoder에서 masking을 계산해준다.
 - masking은 앞으로 나올 단어들을 masking 처리함으로써 단계적으로 나올 단어를 유추할 수 있도록 돋는다. 마스킹 처리를 해주지 않으면 생성하기도 전에 cheating하는 것과 같은 개념이 된다.
- scaling을 해준 이후에 softmax 함수를 적용해서 각 토큰에 대한 확률값이 출력된다.

23.82	12.77	20.3	8.06	11.51	17.62
14.1	7.43	12.09	4.81	6.78	10.48
20.77	11.19	17.65	6.98	10.01	15.39
7.4	3.97	6.31	2.54	3.61	5.47
10.77	5.75	9.21	3.65	5.18	7.96
17.1	9.25	14.55	5.72	8.21	12.63

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105



- 소프트맥스를 적용한 정방 행렬에 value 값을 곱해준다.

- Value값을 곱해줌으로써 원본 값에 각 토큰의 대한 확률 값이 내적되면서 Query에 해당하는 부분이 Attention된 결과 값이 나오게 된다.
- 여기까지가 Single head attention의 결과 값이다.

$$\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$$

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105

6 x 6

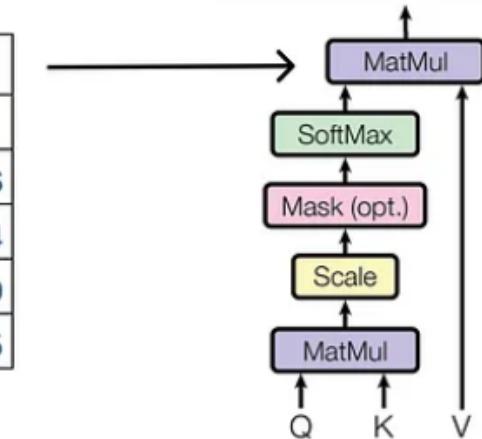
Value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

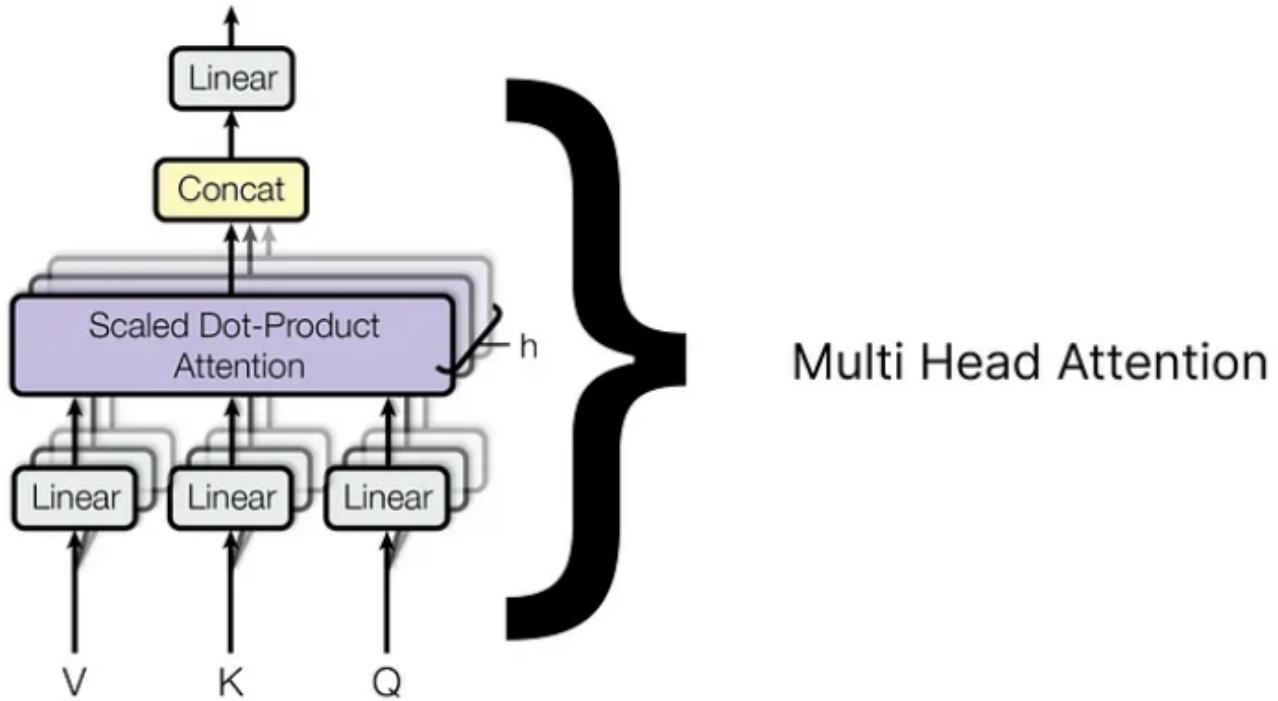
6 x 4

=

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



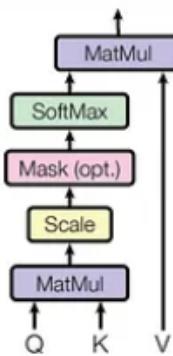
Multi head attention



- multi head attention은 일단 모든 single attention 각각 모두 output을 만들고, 해당 output들이 모두 concatenate 된다.
- 최종적으로 concatenated 한 matrix도 랜덤 값으로 초기화한 가중치 행렬을 곱해서 다시 선형 변환을 하게 된다.
 - 여기서 가중치 행렬은 이후에 훈련되면서 업데이트 된다.

Single Head Attention Our Case

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



$$\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \times \text{Value}$$

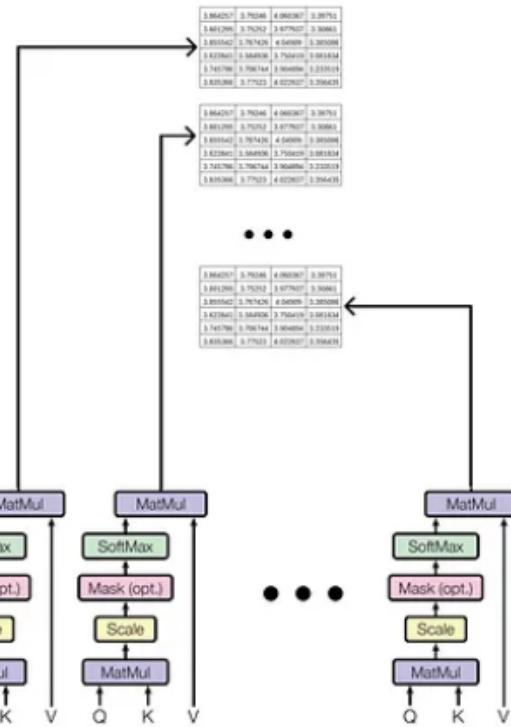
3.86	3.79	4.06	3.4
3.8	3.75	3.98	3.31
3.86	3.79	4.05	3.39
3.62	3.58	3.75	3.08
3.75	3.71	3.9	3.23
3.84	3.78	4.02	3.36

6 x 4

=

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

Multi Head Attention (N Heads) Real world Case Concatenation



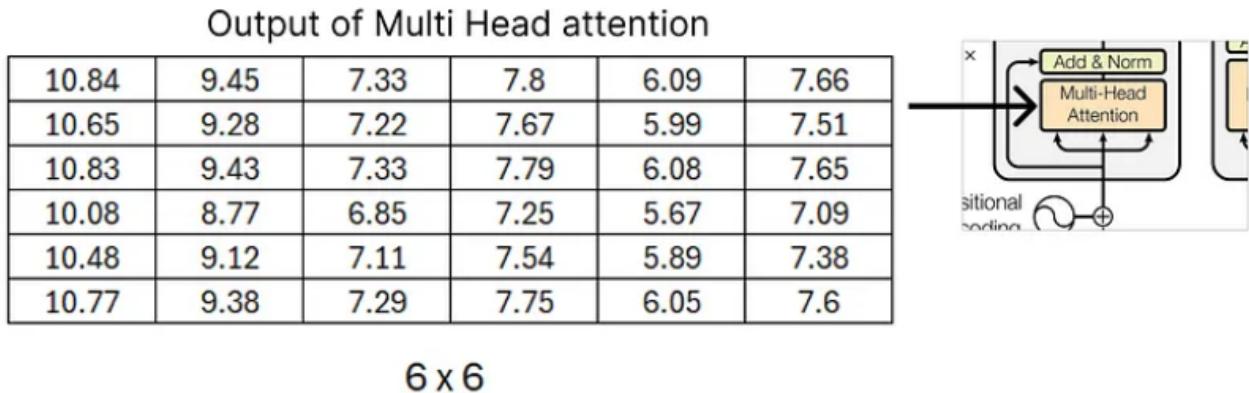
Linear weights
columns length must be
(embedding+positional) matrix columns length

0.8	0.34	0.45	0.54	0.07	0.53
0.85	0.74	0.78	0.5	0.75	0.55
0.53	0.81	0.55	0.59	0.49	0.14
0.7	0.6	0.12	0.42	0.29	0.87

4 x 6

- 여기서 곱해주는 선형 가중치 열의 길이는 반드시 이전에 계산했던 임베딩 행렬 열의 크기와 같아야 한다.

- 그 이유는 최종 결과로 나온 값에 결과적으로 나온 normalized matrix를 더해줄 것이기 때문이다.



Adding and Normalizing

Adding

- Output of Multi head Attention + Embedding matrix

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6×6

+

=

Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6×6

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

- 원본 값을 더해주는 residual을 add 해주는 과정으로 보인다.

Normalize

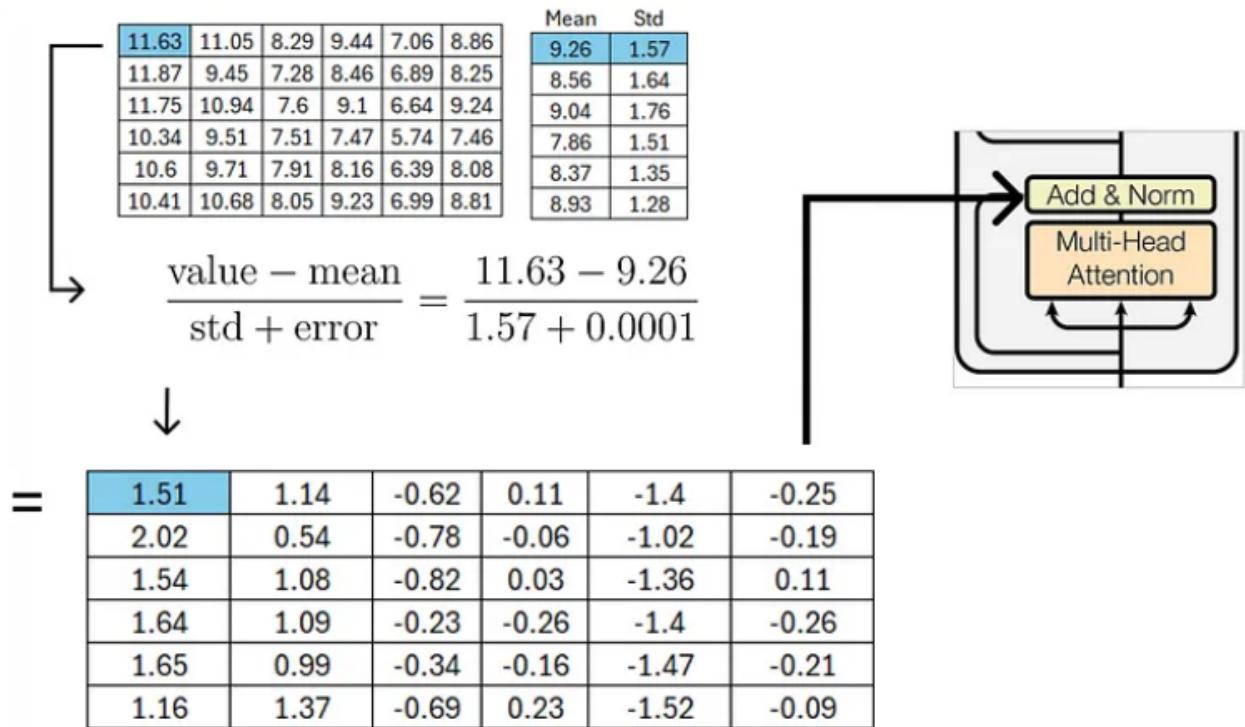
- 정규화 해주기 위해서 우리는 행방향으로 평균과 표준편차를 계산해준다.

$mean = \frac{\sum_{i=1}^N X_i}{N}$	$standard dev. = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$
-------------------------------------	---

Row Wise Implementation

Mean	Standard Deviation
9.26	1.57
8.56	1.64
9.04	1.76
7.86	1.51
8.37	1.35
8.93	1.28

- 정규화해주는 과정에서 std를 나눠주게 되는데 여기에 아주 작은 error값을 추가해준다.
- 그 이유는 zero값으로 division을 하지 않기 위함이다.



Feed Forward Network

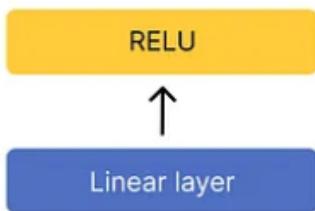
- 정규화를 마친 후에 정방향 네트워크 과정을 거친다.

- 단순한 선형 layer와 activation으로 relu를 사용한다.

$$\text{ReLU}(x) = \max(0, x)$$

$$\text{Linear Layer} = X \cdot W + b$$

our case (one linear layer)



Real world case
(multiple layers)

Linear layer

• • •

RELU



Linear layer



RELU



Linear layer

Matrix after add and norm step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

6 x 6

W

0.5	0.05	0.97	0.22	0.56	0.02
0.17	0.52	0.63	0.48	0.06	0.6
0.53	0.87	0.47	0.1	0.31	0.79
0.83	0.58	0.38	0.09	0.64	0.25
0.81	0.85	0.74	0.35	0.31	0.53
0.25	0.31	0.22	0.77	0.57	0.85

6 x 6

X · W

0.49	1.07	0.84	0.14	0.22	0.7
0.24	1.26	1.11	0.12	0.46	0.97
0.53	1.18	-0.82	0.39	0.33	0.59
0.53	0.97	0.98	0.15	0.16	0.52
0.56	1.11	-0.87	0.11	0.2	0.64
0.62	1.02	0.61	0.26	0.14	0.52

6 x 6

Bias

+

b1	b2	b3	b4	b5	b6
0.42	0.18	0.25	0.42	0.35	0.45

=

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

$$\text{ReLU}(x) = \max(0, x)$$

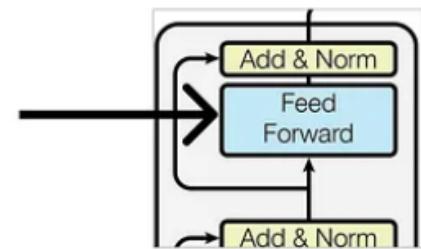
0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

$$\rightarrow \max(0, 0.91)$$



0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97



Adding and Normalizing Again

- 정방향 네트워크도 거친 후에 잔차 연결을 해주고 Normalizing을 해준다.

Matrix from Feed Forward Network

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

Matrix from Previous Add and Norm Step

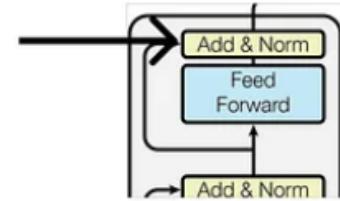
1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09



Mean	Std
1.0033	1.103534
1.1233	1.214349
0.9033	1.301837
0.9933	1.289055
0.8167	1.306016
0.95	1.320773



1.28	1.26	-0.48	-0.3	-1.66	-0.09
1.28	0.71	-0.45	-0.53	-1.1	0.09
1.22	1.18	-1.32	-0.05	-1.22	0.19
1.24	0.97	0.01	-0.53	-1.46	-0.22
1.39	1.12	-0.89	-0.34	-1.33	0.05
0.95	1.23	-0.59	-0.03	-1.5	-0.05



- 이전에 거쳤던 add, norm block에서 나온 output을 더해준다.
- 최종 output matrix는 이후에 decoder part에서 query key matrix로 제공될 것이다.

Decoder part

Reference

- <https://levelup.gitconnected.com/understanding-transformers-from-start-to-end-a-step-by-step-math-example-16d4e64e6eb1>