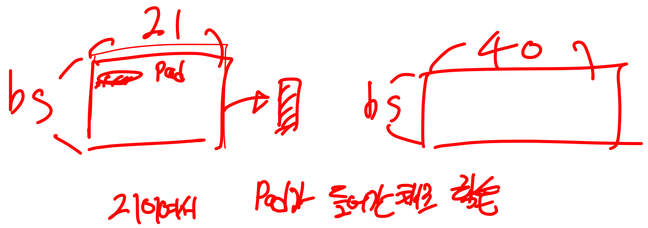
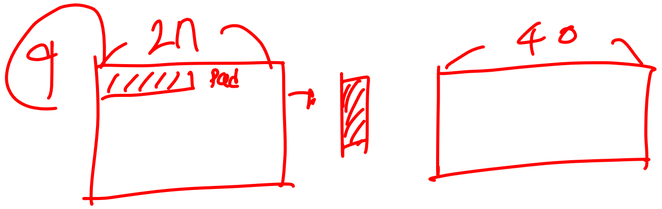
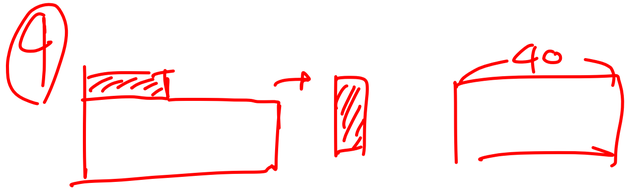
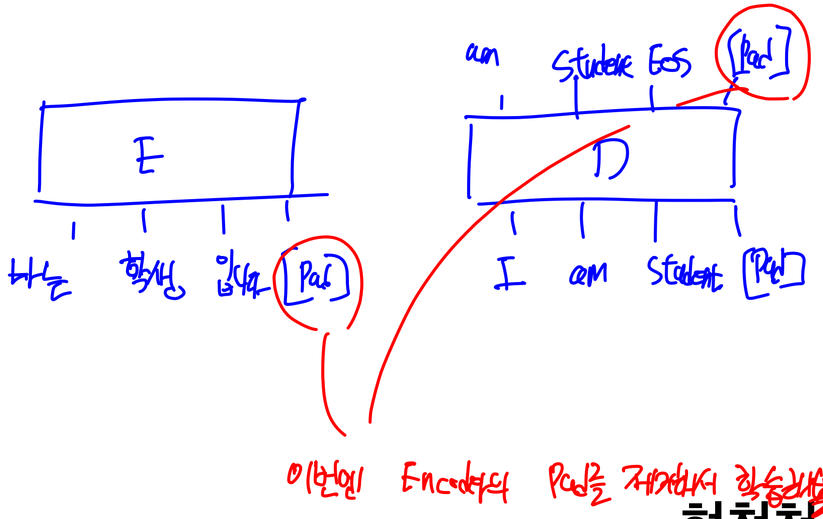


ICT이노베이션스퀘어 AI복합교육 고급 언어과정

자연어처리를 위한 Transformer







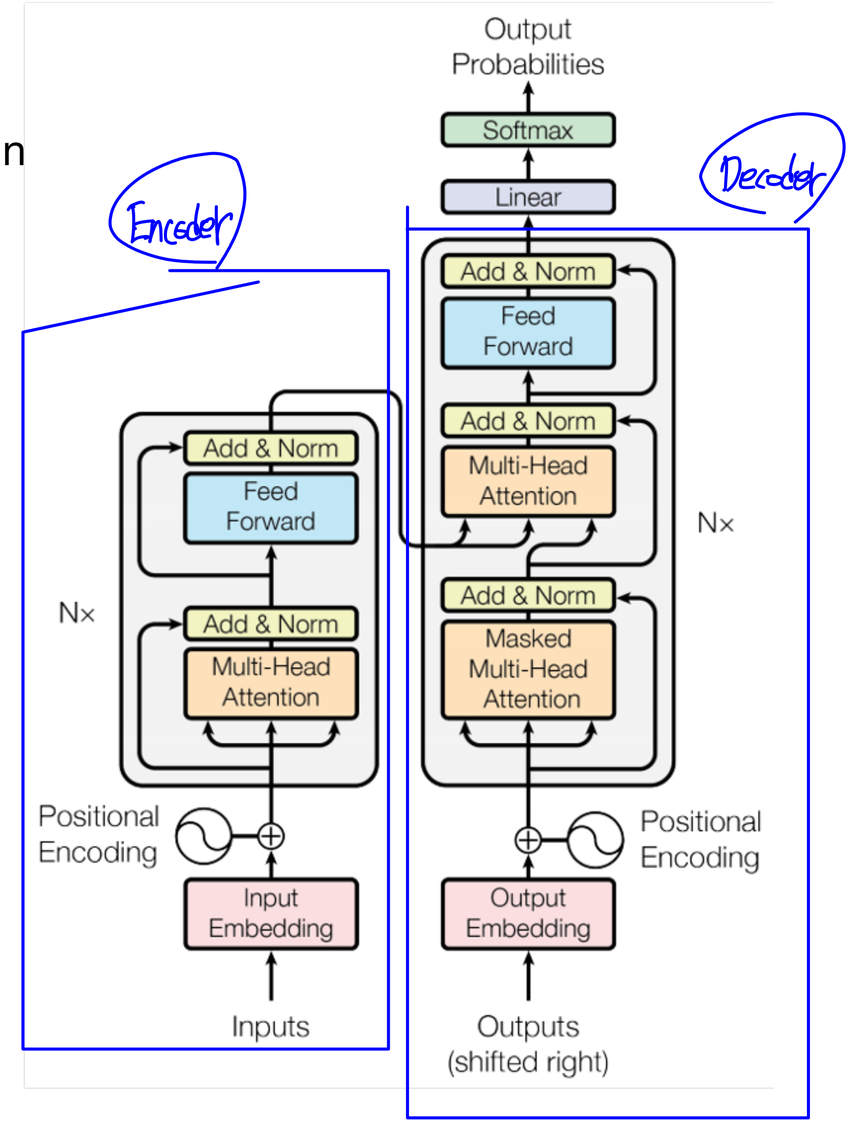
- Pad 있고 없고의 성능차이가 있었음
- 학습할때는 Padding 까지 잘 들어간 data를 학습했지만 실제 위에 학습된 모델을 사용해서 들어간 것 Padding이 들어가지 않은 상태로 들어가서 이전에 없었던 패턴으로 인식해서 잘 안된 상태임.

현청천

2021.04.19

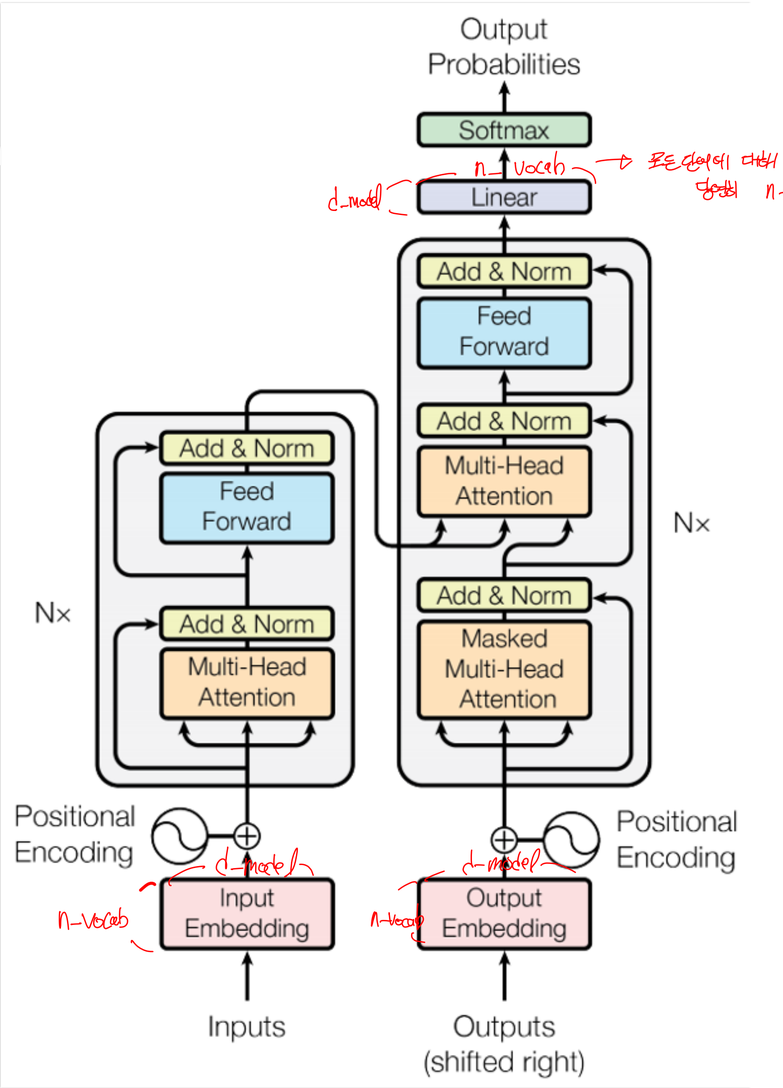
Attention is all you need

- Attention is all you need. 2017 google
- Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin
- https://arxiv.org/abs/1706.03762
  - RNN 없는 attention 만으로 구성된 encoder-decoder모델
  - Parallel corpus 이용한 machine translation task
  - 번역된 단어를 예측
  - NMT의 성능을 한단계 끌어 올림
  - 현재는 자연어 뿐 아니고 여러 분야에서 사용되는 가장 주목받는 모델 임



Attention is all you need

- Attention is all you need. 2017 google
- Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin
- https://arxiv.org/abs/1706.03762
  - RNN 없는 attention 만으로 구성된 encoder-decoder모델
  - Parallel corpus 이용한 machine translation task
  - 번역된 단어를 예측
  - NMT의 성능을 한단계 끌어 올림
  - 현재는 자연어 뿐 아니고 여러 분야에서 사용되는 가장 주목받는 모델 임



Attention is all you need (Abstract)

- RNN, CNN등을 사용하지않고

  attention mechanisms만으로 구성된

  Transformer 네트워크를 제안 함

- 두 번역 task에서 실험한 결과 이 모델이 품질이

  우수하면서도 병렬(parallel)처리로 학습시간이 단축 됨

  - **WMT 2014 English-to-German**:

    28.4 BLEU.

    기존 기록 2BLUE 초과

  - **WMT 2014 English-to-French**: 단일 모델

    SOTA 41.8 BLUE.

    8개 GPU에서 3.5일 학습 함.

- 영어 constituency parsing에서 많은 데이터와 적은 데이터에

  성공적으로 Transformer를 적용하므로 다른 task에도 적용이 가능함을 보임

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

Attention is all you need (Introduction)

- RNN은 문장의 순서대로 연산을 수행 함.

  Hidden state h_t는 이전 hidden state h_{t-1}과 현재 위치 t의 함수.

  이와 같은 순차적인 처리때문에 동시(parallel)처리가 어려움

- Attention mechanisms은 다양한 task에서 주목 받고 있음.

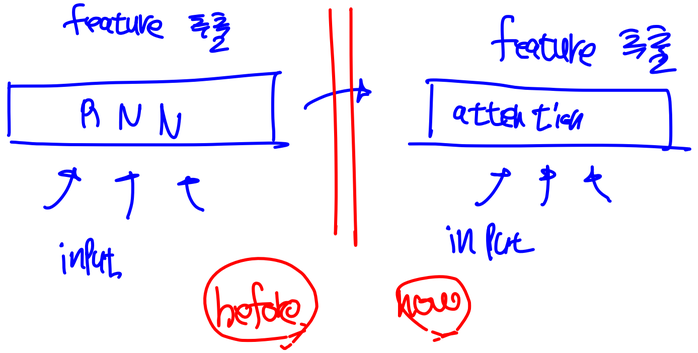
  Attention은 입력의 거리에 영향을 받지 않음

- Transformer는 입력과 출력의 관계를

  attention mechanism을 이용해서 학습함.

  Transformer는 더 많은 동시(parallel)처리를 가능하게 해서

  8개의 P100 GPUs에서 12시간 만에 기존보다 더 좋은 성능을 냄.



1 Introduction

Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation [35, 2, 5]. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures [38, 24, 15].

Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Recent work has achieved significant improvements in computational efficiency through factorization tricks [21] and conditional computation [32], while also improving model performance in case of the latter. The fundamental constraint of sequential computation, however, remains.

Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences [2, 19]. In all but a few cases [27], however, such attention mechanisms are used in conjunction with a recurrent network.

In this work we propose the Transformer, a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality after being trained for as little as twelve hours on eight P100 GPUs.

Attention is all you need (Background)

- 단어의 위치가 먼 경우 관계를 계산하는데 많은 operations이 필요함.
  이런 이유로 거리가 먼 단어의 관계를 계산하는데 어려움이 있음.
  Transformer는 operations 수를 줄이면서
  Multi-Head Attention으로 해결.

- Self-attention (intra-attention)은 단일 문장의
  다른 위치와의 관계를 attention mechanism을 이용해서
  단어의 representations을 계산.

- Transformer는 RNNs or CNN을 사용하지 않고
  self-attention만을 사용해서 representations을 계산하는
  첫번째 번역 모델

2 Background

The goal of reducing sequential computation also forms the foundation of the Extended Neural GPU [16], ByteNet [18] and ConvS2S [9], all of which use convolutional neural networks as basic building block, computing hidden representations in parallel for all input and output positions. In these models, the number of operations required to relate signals from two arbitrary input or output positions grows in the distance between positions, linearly for ConvS2S and logarithmically for ByteNet. This makes it more difficult to learn dependencies between distant positions [12]. In the Transformer this is reduced to a constant number of operations, albeit at the cost of reduced effective resolution due to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention as described in section 3.2.

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations [4, 27, 28, 22].

End-to-end memory networks are based on a recurrent attention mechanism instead of sequence-aligned recurrence and have been shown to perform well on simple-language question answering and language modeling tasks [34].

To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution. In the following sections, we will describe the Transformer, motivate self-attention and discuss its advantages over models such as [17, 18] and [9].

Attention is all you need (Model Architecture)

- Input sequence: $x = (x_1, x_2, \ldots, x_n)$
- Encoder output: $z = (z_1, z_2, \ldots, z_n)$
- Decoder output: $y = (y_1, y_2, \ldots, y_m)$

- Auto-regressive: 이전 step에서 생성된 symbols을 다음 step 생성을 위해 추가 token으로 입력 함

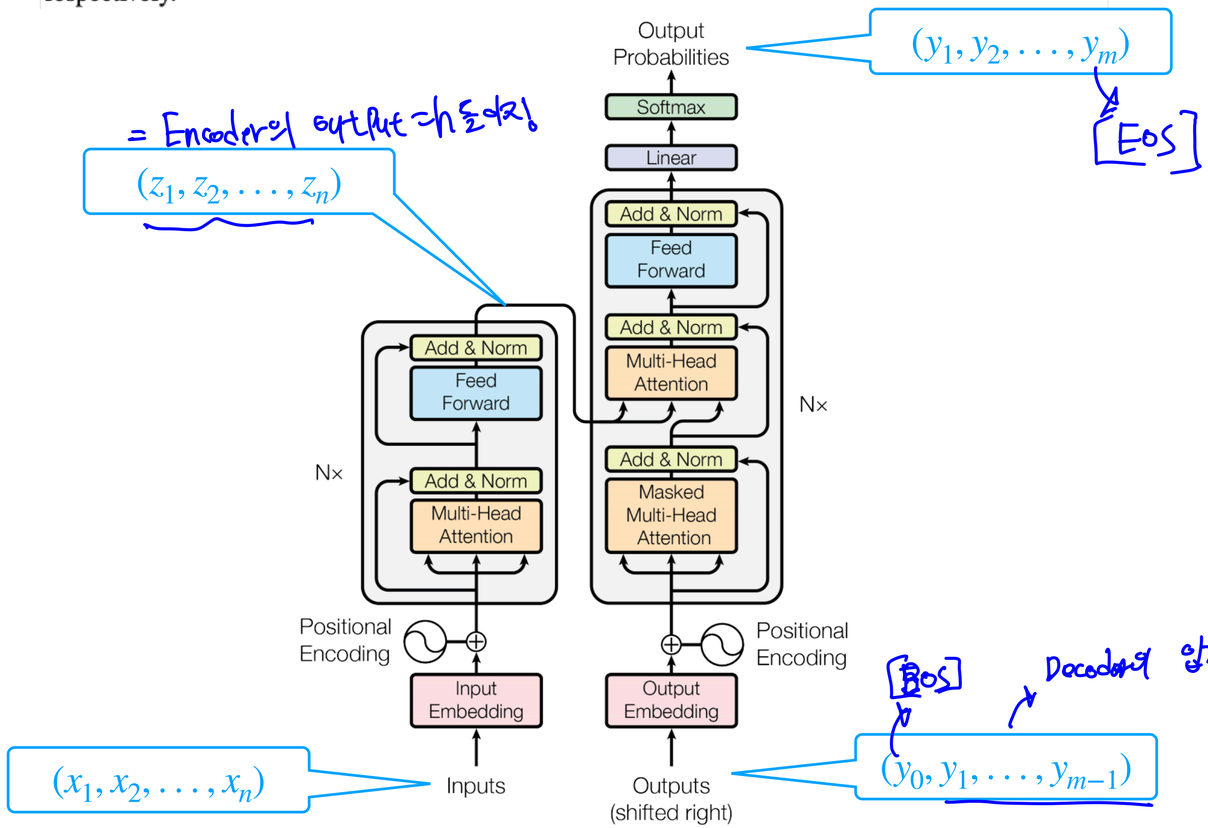
Decoder Input	Decoder Output
(y_0)=[BOS]	(y_1)
(y_0, y_1)	(y_1, y_2)
...	...
$(y_0, y_1, \ldots, y_{m-1})$	$(y_1, y_2, \ldots, y_m)$

[EOS]

3 Model Architecture

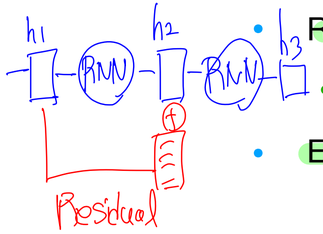
Most competitive neural sequence transduction models have an encoder-decoder structure [5, 2, 35]. Here, the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive [10], consuming the previously generated symbols as additional input when generating the next.

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.



Attention is all you need (Encoder and Decoder Stacks)

- **Encoder**
  - Stack of N=6 identical layers
  - Sublayers
    - Multi-head self-attention
    - Position-wise fully connected feed-forward network
  - Residual and layernorm
    - $LayerNorm(x + Sublayer(x))$
  - Embedding and sublayers output $d_{model} = 512$

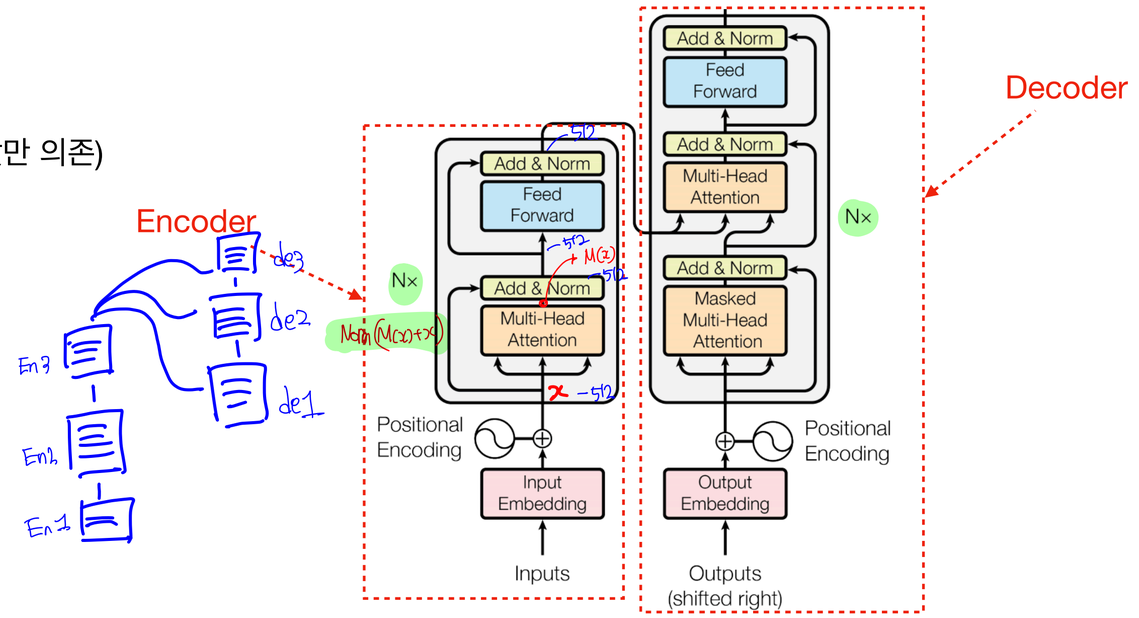


- **Decoder**
  - Stack of N=6 identical layers
  - Sublayers
    - Masked multi-head self-attention (i에 대한 예측은 i이전 값만 의존)
    - Multi-head encoder-decoder attention
    - Position-wise fully connected feed-forward network
  - Residual and layernorm
    - $LayerNorm(x + Sublayer(x))$
  - Embedding and sublayers output $d_{model} = 512$

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.



Attention is all you need (Attention)

- **Attention**
  - 입력 query와 key-value 쌍
  - 출력은 value이 weighted sum 임
  - Weight는 query와 key의 함수로 계산 됨

decoder - S, encoder h, encoder h

$\frac{S^T}{q} \cdot \frac{h}{k}$ = weight

Q S K h V h

Q h K h V h ⇒ self

Q S K S V S ⇒ self

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Attention score $$e = s^T h \in \mathbb{R}^{m \times n}$$

Attention prob $$\alpha = softmax(e) \in \mathbb{R}^{m \times n}$$ ← Soft($S^T \cdot h$)

Attention output $$a = h\alpha^T \in \mathbb{R}^{h \times m}$$ ∈ h · (Soft($S^T \cdot h$))^T

$S^T = Q$

$h^T = k$ and V

a^T = Soft($S^T \cdot h$) · h^T

$a = V^T \cdot (soft(Q \cdot K^T))^T$

$a^T = softmax(Q \cdot K^T) \cdot V$

$$a^T = softmax(QK^T)V$$

이항법칙

where $Q = s^T, K = h^T, V = h^T$

Attention is all you need (Scaled Dot-Product Attention)

- Input
  - Q: d_k
  - K: d_k
  - V: d_v
- $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$
  - Dot-product attention에 scaling factor $\sqrt{d_k}$ 추가
  - Additive가 dot-product attention보다 성능이 좋으나 scaling factor $\frac{1}{\sqrt{d_k}}$가 추가되면 성능이 비슷함.

    Dot-product가 빠르고 메모리를 적게 사용 함
  - d_k가 클경우 dot-product값이 커지고 softmax 결과의 기울기가 줄어 듬.

    이를 해결하기 위해 dot-product의 결과에 scaling factor $\frac{1}{\sqrt{d_k}}$를 곱함

3.2.1 Scaled Dot-Product Attention

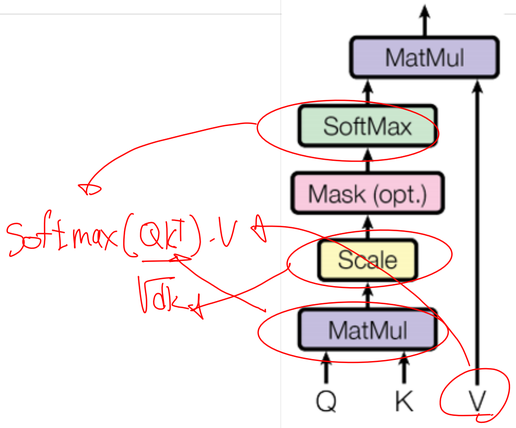
We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

The two most commonly used attention functions are additive attention [2], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.



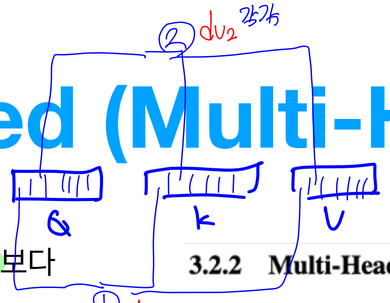
Attention is all you need (Multi-Head Attention)

- d_{model}차원의 query, key, value의 attention을 한번 실행하는 것 보다
- Linear projection으로 여러 작은 d_k, d_v차원의 query, key, value로 나눈 후
- 각 attention을 병렬로 실행하여 여러개의 작은 d_v차원의 출력을 만든 후
- 여러개의 d_v를 concatenate 후 linear projection을 최종 출력을 생성

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^o$$

- $$where \quad head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$
- $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}, W^O \in \mathbb{R}^{hd_v \times d_{model}}$
- $h = 8, \quad d_k = d_v = \frac{d_{model}}{h} = 64$

- Head별로 차원을 줄여서 전체 계산 량은 single-head attention과 비슷함



3.2.2 Multi-Head Attention

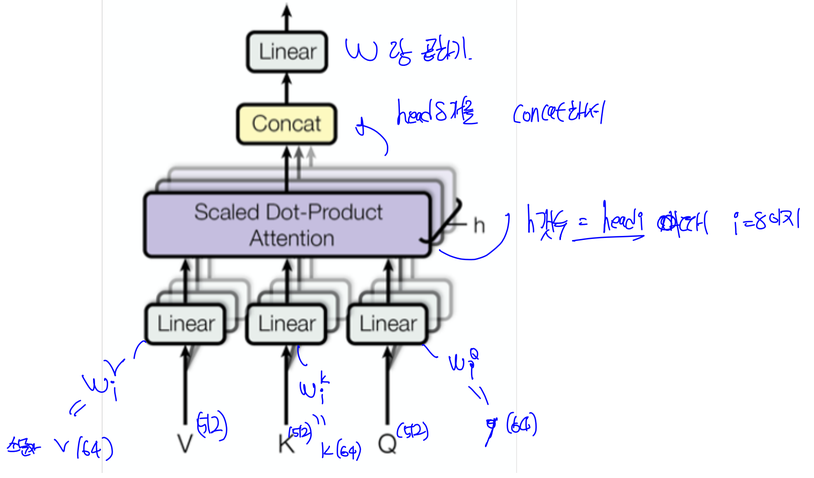
Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.



Attention is all you need (Applications of Attention)

① **encoder-decoder attention**
- Q: 이전 decoder layer 결과
- K, V: encoder 결과
- Encoder 입력 대해서 decoder가 attention을 할 수 있도록 함

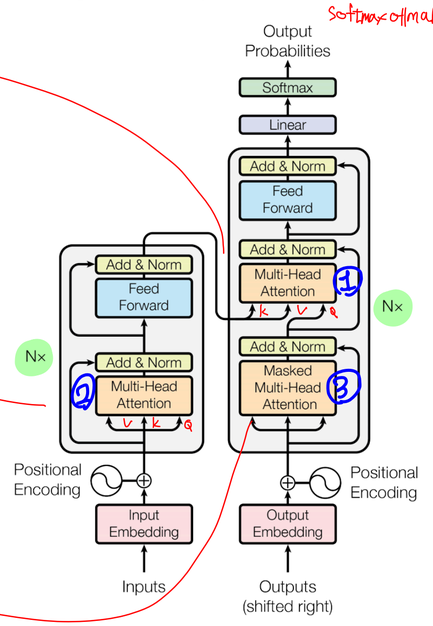
② **encoder self-attention**
- Q, K, V: 이전 encoder layer 결과
- Encoder 입력 대해서 attention을 할 수 있도록 함

③ **decoder self-attention**
- Q, K, V: 이전 decoder layer 결과
- Decoder 입력 대해서 attention을 할 수 있도록 함
- auto-regressive를 위해 mask를 사용 함

3.2.3 Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways:

- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [38, 2, 9].
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 2.



Attention is all you need (Applications of Attention)

- **encoder-decoder attention**
  - Q: 이전 decoder layer 결과
  - K, V: encoder 결과
  - Encoder 입력 대해서 decoder가 attention을 할 수 있도록 함

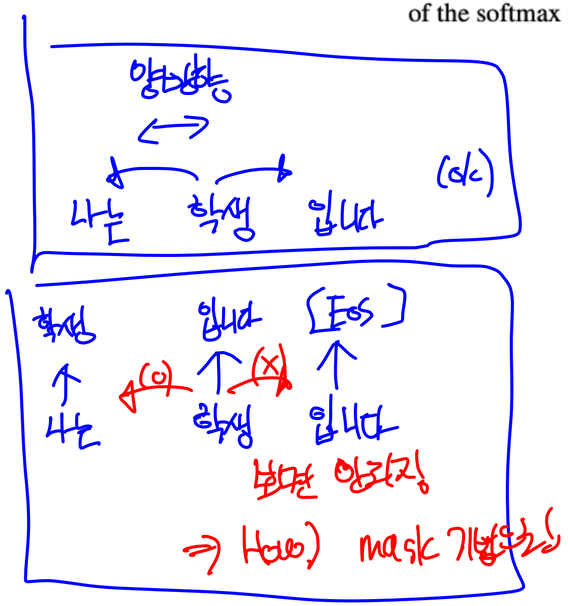
- **encoder self-attention**
  - Q, K, V: 이전 encoder layer 결과
  - Encoder 입력 대해서 attention을 할 수 있도록 함

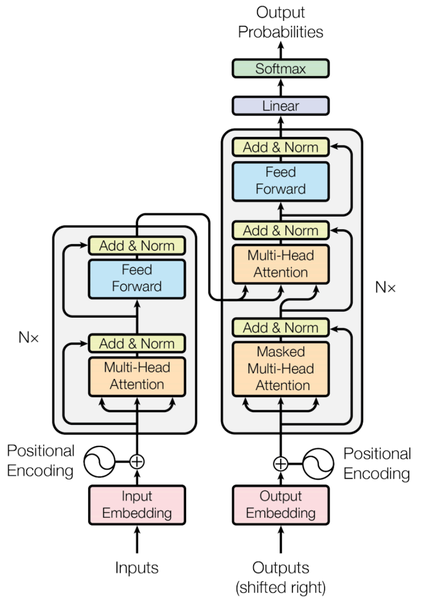
- **decoder self-attention**
  - Q, K, V: 이전 decoder layer 결과
  - Decoder 입력 대해서 attention을 할 수 있도록 함
  - auto-regressive를 위해 mask를 사용 함

3.2.3 Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways:

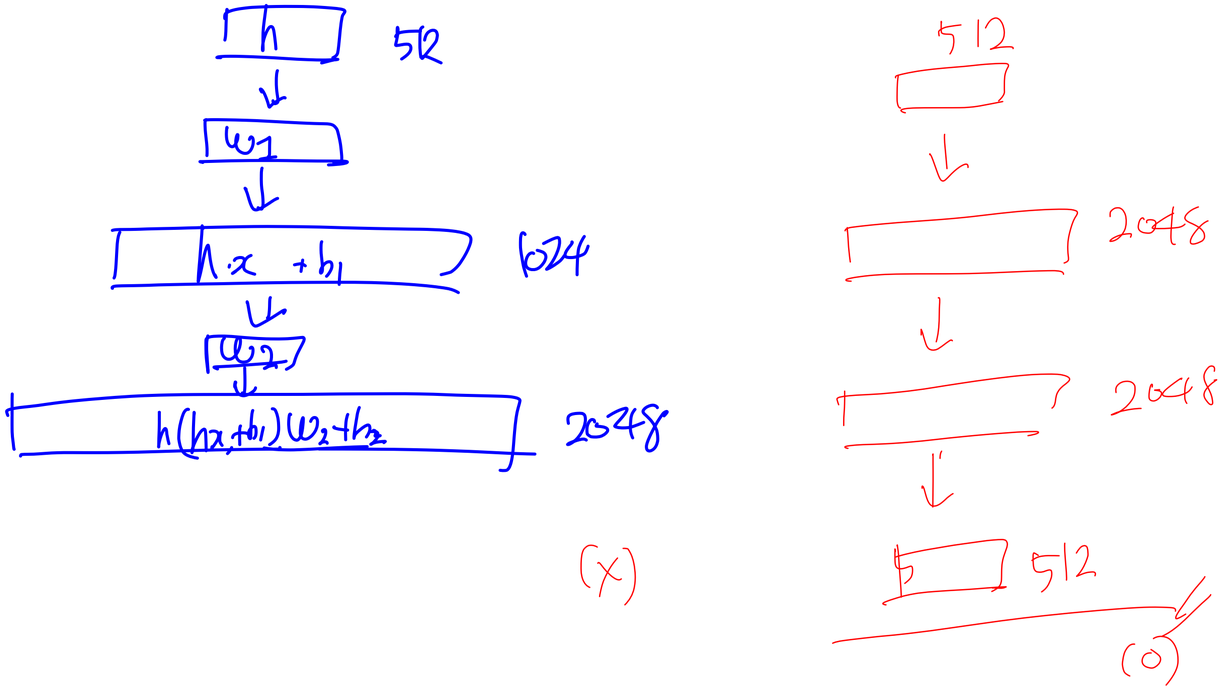
- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [38, 2, 9].
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 2.





Attention is all you need (Applications of Attention)

- 두개의 linear와 사이에 ReLU activation으로 구성 됨
- $FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$
- Input, output: $d_{model} = 512$
- Inner-layer: $d_{ff} = 1024$ → 2048



3.3 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

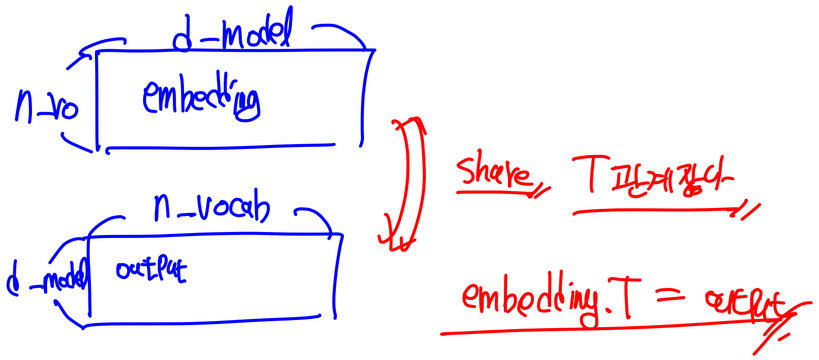
While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

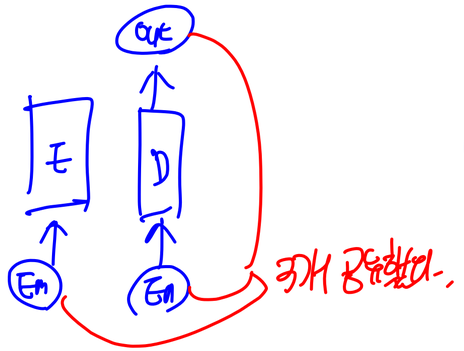
Attention is all you need (Embedding and Softmax)

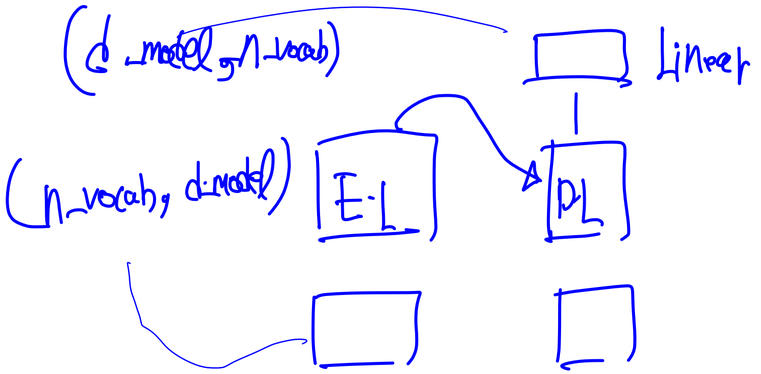
- 입력 token을 d_{model} 차원 vector로 변경하기 위해 embedding을 사용 함
- Decoder 출력을 위해 softmax를 사용 함
- 두개의 embedding과 softmax 이전 linear가 weight matrix를 공유 함
- Embedding layer에 $\sqrt{d_{model}}$을 곱해 줌

3.4 Embeddings and Softmax

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [30]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.







Attention is all you need (Positional Encoding)

- 모델이 sequence 순서를 사용하기 위해 입력 token에 절대(상대) 위치를 포함 (positional encodings)
- Token Embedding과 동일한 d_{model}차원을 가짐 (더하기 가능)
- $$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$
- $$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$
- pos: position, i: dimension
- Sinusoid encoding은 쉽게 상대적 위치를 배울 수 있다는 가설.
  PEpos+k는 PEpos의 선형 함수로 계산 가능
- Position Embedding을 학습하는 것과 sinusoid를 사용하는 것과 결과가 거의 동일함

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

In this work, we use sine and cosine functions of different frequencies:

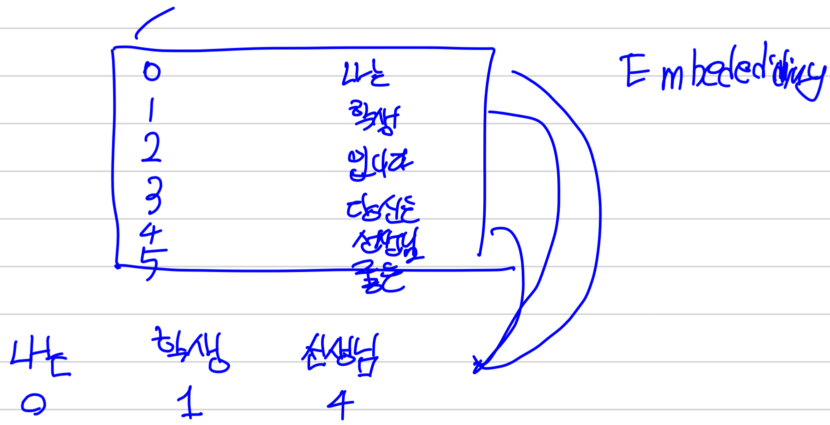
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

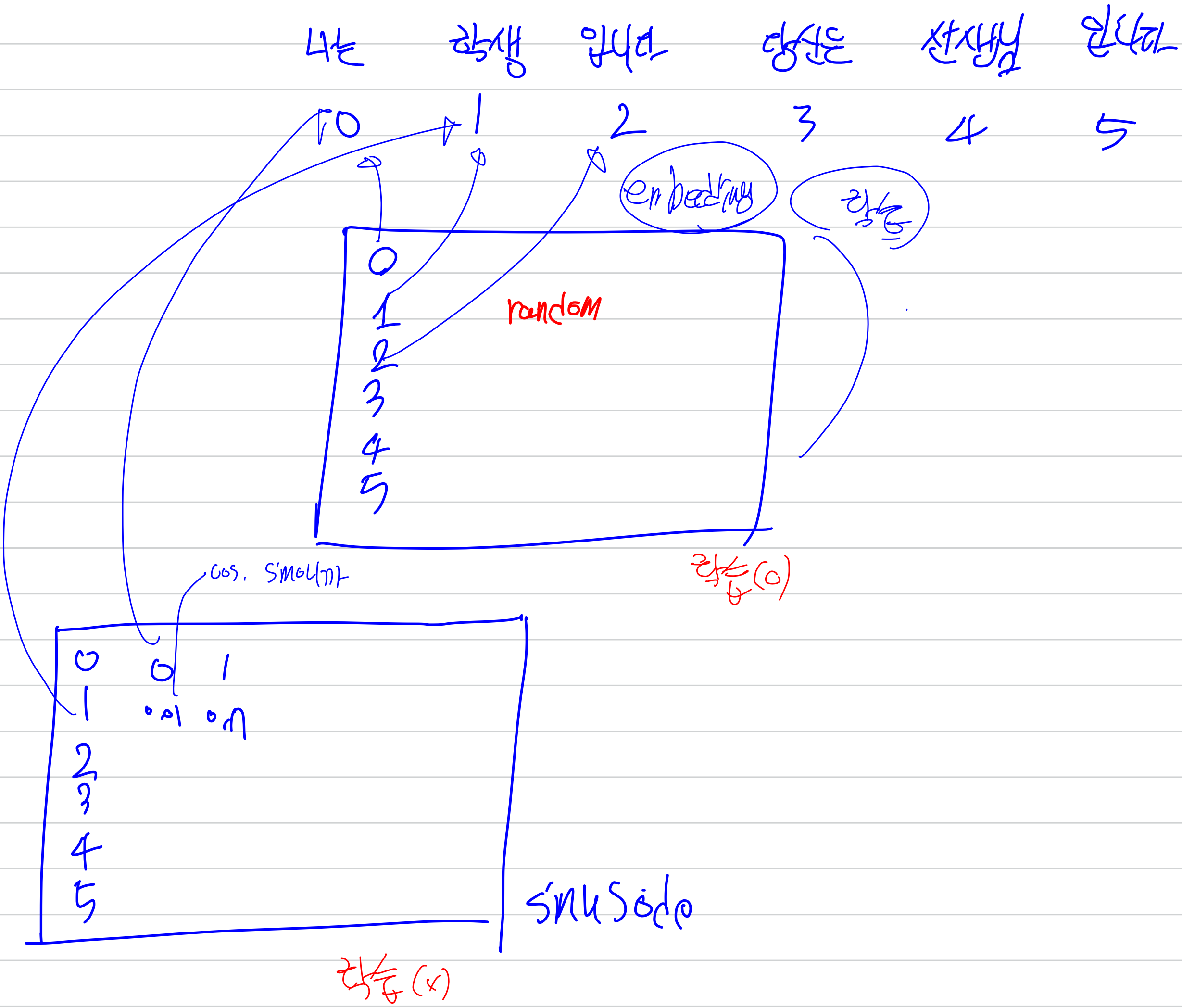
We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

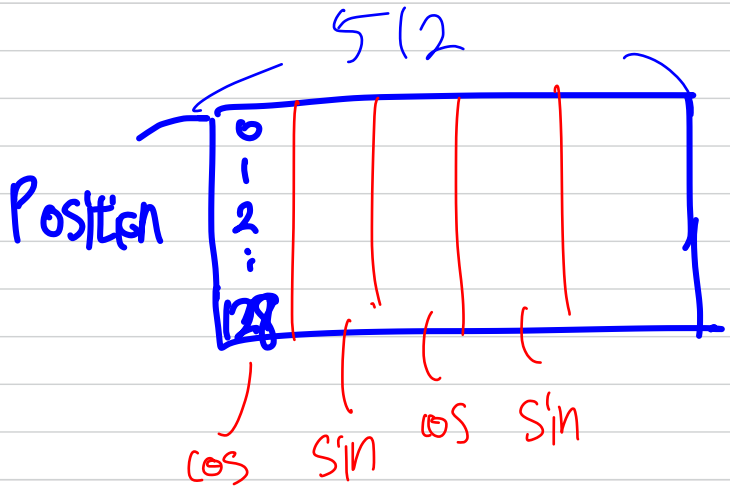
	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(E)	positional embedding instead of sinusoids									4.92	25.7	

6개짜리 vokab



Position




512
Position
0
1
2
:
128
cos
sin
cos
sin

Attention is all you need (Why Self-Attention)

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- Sequential Operation: self attention은 $O(1)$ 번, RNN은 $O(N)$번 연산
- Computational complexity: d보다는 n이 작을 때 빠름
- 긴 문장을 처리하는 성능을 높이기 위해 attention을 주변 r개 단어만 계산하는 방법 가능 (restricted). Maximum path length가 $O(\frac{n}{r})$ 로 증가
- CNN에서 k<n일 경우 Maximum path length가 $O(\log_k(n))$이 됨
- Self-attention은 attention distribution을 통해 좀더 해석이 쉬움

4 Why Self-Attention

In this section we compare various aspects of self-attention layers to the recurrent and convolutional layers commonly used for mapping one variable-length sequence of symbol representations $(x_1, ..., x_n)$ to another sequence of equal length $(z_1, ..., z_n)$, with $x_i, z_i \in \mathbb{R}^d$, such as a hidden layer in a typical sequence transduction encoder or decoder. Motivating our use of self-attention we consider three desiderata.

One is the total computational complexity per layer. Another is the amount of computation that can be parallelized, as measured by the minimum number of sequential operations required.

The third is the path length between long-range dependencies in the network. Learning long-range dependencies is a key challenge in many sequence transduction tasks. One key factor affecting the ability to learn such dependencies is the length of the paths forward and backward signals have to traverse in the network. The shorter these paths between any combination of positions in the input and output sequences, the easier it is to learn long-range dependencies [12]. Hence we also compare the maximum path length between any two input and output positions in networks composed of the different layer types.

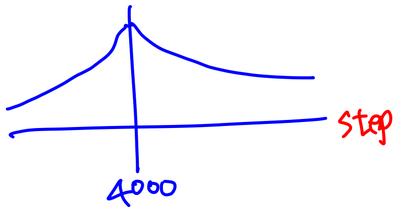
As noted in Table 1, a self-attention layer connects all positions with a constant number of sequentially executed operations, whereas a recurrent layer requires $O(n)$ sequential operations. In terms of computational complexity, self-attention layers are faster than recurrent layers when the sequence length n is smaller than the representation dimensionality d, which is most often the case with sentence representations used by state-of-the-art models in machine translations, such as word-piece [38] and byte-pair [31] representations. To improve computational performance for tasks involving very long sequences, self-attention could be restricted to considering only a neighborhood of size r in the input sequence centered around the respective output position. This would increase the maximum path length to $O(n/r)$. We plan to investigate this approach further in future work.

A single convolutional layer with kernel width $k < n$ does not connect all pairs of input and output positions. Doing so requires a stack of $O(n/k)$ convolutional layers in the case of contiguous kernels, or $O(log_k(n))$ in the case of dilated convolutions [18], increasing the length of the longest paths between any two positions in the network. Convolutional layers are generally more expensive than recurrent layers, by a factor of k. Separable convolutions [6], however, decrease the complexity considerably, to $O(k \cdot n \cdot d + n \cdot d^2)$. Even with $k = n$, however, the complexity of a separable convolution is equal to the combination of a self-attention layer and a point-wise feed-forward layer, the approach we take in our model.

As side benefit, self-attention could yield more interpretable models. We inspect attention distributions from our models and present and discuss examples in the appendix. Not only do individual attention heads clearly learn to perform different tasks, many appear to exhibit behavior related to the syntactic and semantic structure of the sentences.

Attention is all you need (Training)

- **Training Data and Batching**
  - WMT 2014 English-German (4.5M sentence pair)
  - BPE 37,000 vocabulary
  - WMT 2014 English-French (36M sentence pair)
  - Word-piece 32,000 vocabulary
- **Hardware and Schedule**
  - 8 NVIDIA P100 GPU
  - Base model: 100,000 steps or 12 hours (0.4sec/step)
  - Big model: 300,000 steps 3.5days (1.0sec/step)
- **Optimizer**
  - Adam optimizer: $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$
  - $lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$
  - $lrate$가 처음 warmup_steps 까지는 선형 증가 이후 step의 역 제곱근에 비례해서 감소 (warmup_steps = 4000)



5 Training

This section describes the training regime for our models.

5.1 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary [38]. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

5.2 Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

5.3 Optimizer

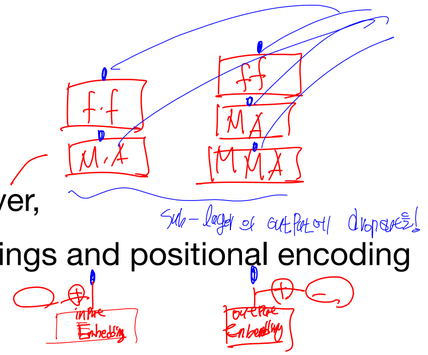
We used the Adam optimizer [20] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first $warmup\_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$.

Attention is all you need (Training)

- **Regularization**
  - Residual Dropout
    - Output of sub-layer, sums of embeddings and positional encoding
    - $P_{drop} = 0.1$
  - Label Smoothing
    - $\epsilon_{ls} = 0.1$
    - Perplexity가 증가하지만 BLEU score가 높아짐

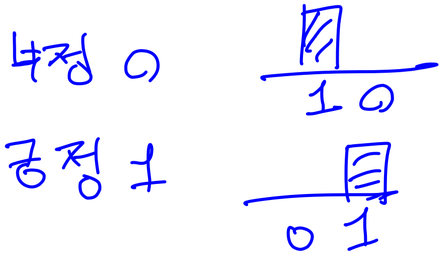


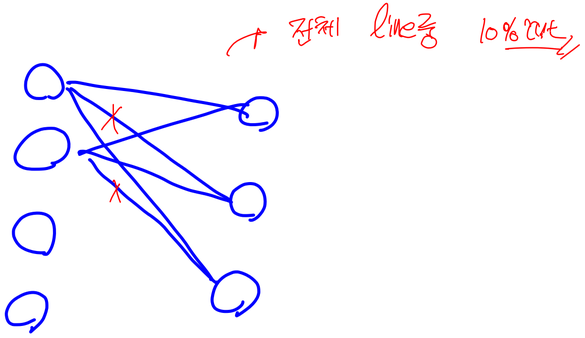
5.4 Regularization

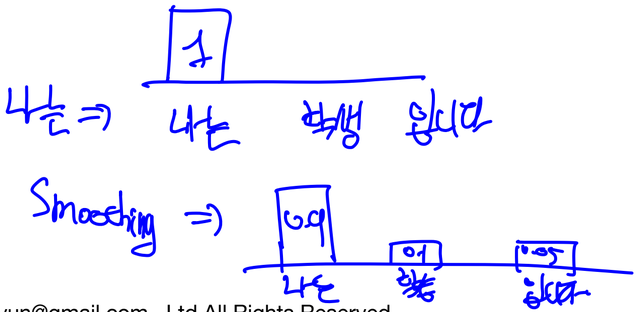
We employ three types of regularization during training:

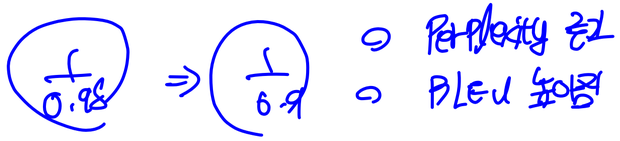
Residual Dropout We apply dropout [33] to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$.

Label Smoothing During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ [36]. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.









Attention is all you need (Results)

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- **Machine Translation**
  - WMT 2014 English-to-German
    - Transformer(big)
    - SOTA 28.4 BLEU (over 2.0 previous)
    - Training 8 p100 GPUs, 3.5 days
    - Transformer(base)는 training cost가 작으면서 성능이 좋음
  - WMT 2014 English-to-French
    - Transformer(big)
    - SOTA 41.0 BLEU
    - $\frac{1}{4}$training cost of previous SOTA
    - $P_{drop} = 0.1$
    - BEAM search 사용 beam size: 4, length penalty $\alpha = 0.6$

6 Results

6.1 Machine Translation

On the WMT 2014 English-to-German translation task, the big transformer model (Transformer (big) in Table 2) outperforms the best previously reported models (including ensembles) by more than 2.0 BLEU, establishing a new state-of-the-art BLEU score of 28.4. The configuration of this model is listed in the bottom line of Table 3. Training took 3.5 days on 8 P100 GPUs. Even our base model surpasses all previously published models and ensembles, at a fraction of the training cost of any of the competitive models.

On the WMT 2014 English-to-French translation task, our big model achieves a BLEU score of 41.0, outperforming all of the previously published single models, at less than $1/4$ the training cost of the previous state-of-the-art model. The Transformer (big) model trained for English-to-French used dropout rate $P_{drop} = 0.1$, instead of 0.3.

For the base models, we used a single model obtained by averaging the last 5 checkpoints, which were written at 10-minute intervals. For the big models, we averaged the last 20 checkpoints. We used beam search with a beam size of 4 and length penalty $\alpha = 0.6$ [38]. These hyperparameters were chosen after experimentation on the development set. We set the maximum output length during inference to input length + 50, but terminate early when possible [38].

Table 2 summarizes our results and compares our translation quality and training costs to other model architectures from the literature. We estimate the number of floating point operations used to train a model by multiplying the training time, the number of GPUs used, and an estimate of the sustained single-precision floating-point capacity of each GPU [5].

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Attention is all you need (Results)

- **Model Variations**

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

6.2 Model Variations

To evaluate the importance of different components of the Transformer, we varied our base model in different ways, measuring the change in performance on English-to-German translation on the development set, newstest2013. We used beam search as described in the previous section, but no checkpoint averaging. We present these results in Table 3.

In Table 3 rows (A), we vary the number of attention heads and the attention key and value dimensions, keeping the amount of computation constant, as described in Section 3.2.2. While single-head attention is 0.9 BLEU worse than the best setting, quality also drops off with too many heads.

In Table 3 rows (B), we observe that reducing the attention key size d_k hurts model quality. This suggests that determining compatibility is not easy and that a more sophisticated compatibility function than dot product may be beneficial. We further observe in rows (C) and (D) that, as expected, bigger models are better, and dropout is very helpful in avoiding over-fitting. In row (E) we replace our sinusoidal positional encoding with learned positional embeddings [9], and observe nearly identical results to the base model.

Attention is all you need (Results)

- **English Constituency Parsing**
  - WSJ Penn Treebank 40K
    - 16K vocabulary
  - BerkleyParser 17M (semi-supervised)
    - 32K vocabulary
  - 4-layer, $d_{model} = 1024$
  - Output length: input length + 300
  - Bean size: 21, α = 0.3

Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

6.3 English Constituency Parsing

To evaluate if the Transformer can generalize to other tasks we performed experiments on English constituency parsing. This task presents specific challenges: the output is subject to strong structural constraints and is significantly longer than the input. Furthermore, RNN sequence-to-sequence models have not been able to attain state-of-the-art results in small-data regimes [37].

We trained a 4-layer transformer with $d_{model} = 1024$ on the Wall Street Journal (WSJ) portion of the Penn Treebank [25], about 40K training sentences. We also trained it in a semi-supervised setting, using the larger high-confidence and BerkleyParser corpora from with approximately 17M sentences [37]. We used a vocabulary of 16K tokens for the WSJ only setting and a vocabulary of 32K tokens for the semi-supervised setting.

We performed only a small number of experiments to select the dropout, both attention and residual (section 5.4), learning rates and beam size on the Section 22 development set, all other parameters remained unchanged from the English-to-German base translation model. During inference, we increased the maximum output length to input length + 300. We used a beam size of 21 and $\alpha = 0.3$ for both WSJ only and the semi-supervised setting.

Our results in Table 4 show that despite the lack of task-specific tuning our model performs surprisingly well, yielding better results than all previously reported models with the exception of the Recurrent Neural Network Grammar [8].

In contrast to RNN sequence-to-sequence models [37], the Transformer outperforms the Berkeley-Parser [29] even when training only on the WSJ training set of 40K sentences.

참고: http://nlpprogress.com/english/constituency_parsing.html
https://github.com/nikitakit/self-attentive-parser

Attention is all you need (Conclusion)

- Transformer는 RNN을 사용하지 않고 attention만을 사용한 첫 번역 모델
- Encoder-Decoder 구조에 multi-head self-attention 사용

- 번역 task에서 Transformer는 RNN, CNN에 비해 빨리 학습
- WMT 2014 English-to-German, English-to-French 번역에서 SOTA 기록

- https://github.com/tensorflow/tensor2tensor > tf. 공식

illustrated transformer 한국어

SuperGLUE — Leaderboard

7 Conclusion

In this work, we presented the Transformer, the first sequence transduction model based entirely on attention, replacing the recurrent layers most commonly used in encoder-decoder architectures with multi-headed self-attention.

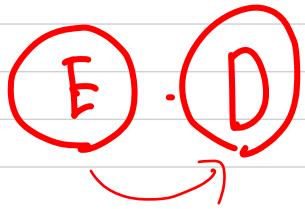
For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. On both WMT 2014 English-to-German and WMT 2014 English-to-French translation tasks, we achieve a new state of the art. In the former task our best model outperforms even all previously reported ensembles.

We are excited about the future of attention-based models and plan to apply them to other tasks. We plan to extend the Transformer to problems involving input and output modalities other than text and to investigate local, restricted attention mechanisms to efficiently handle large inputs and outputs such as images, audio and video. Making generation less sequential is another research goals of ours.

The code we used to train and evaluate our models is available at `https://github.com/tensorflow/tensor2tensor`.

Acknowledgements We are grateful to Nal Kalchbrenner and Stephan Gouws for their fruitful comments, corrections and inspiration.

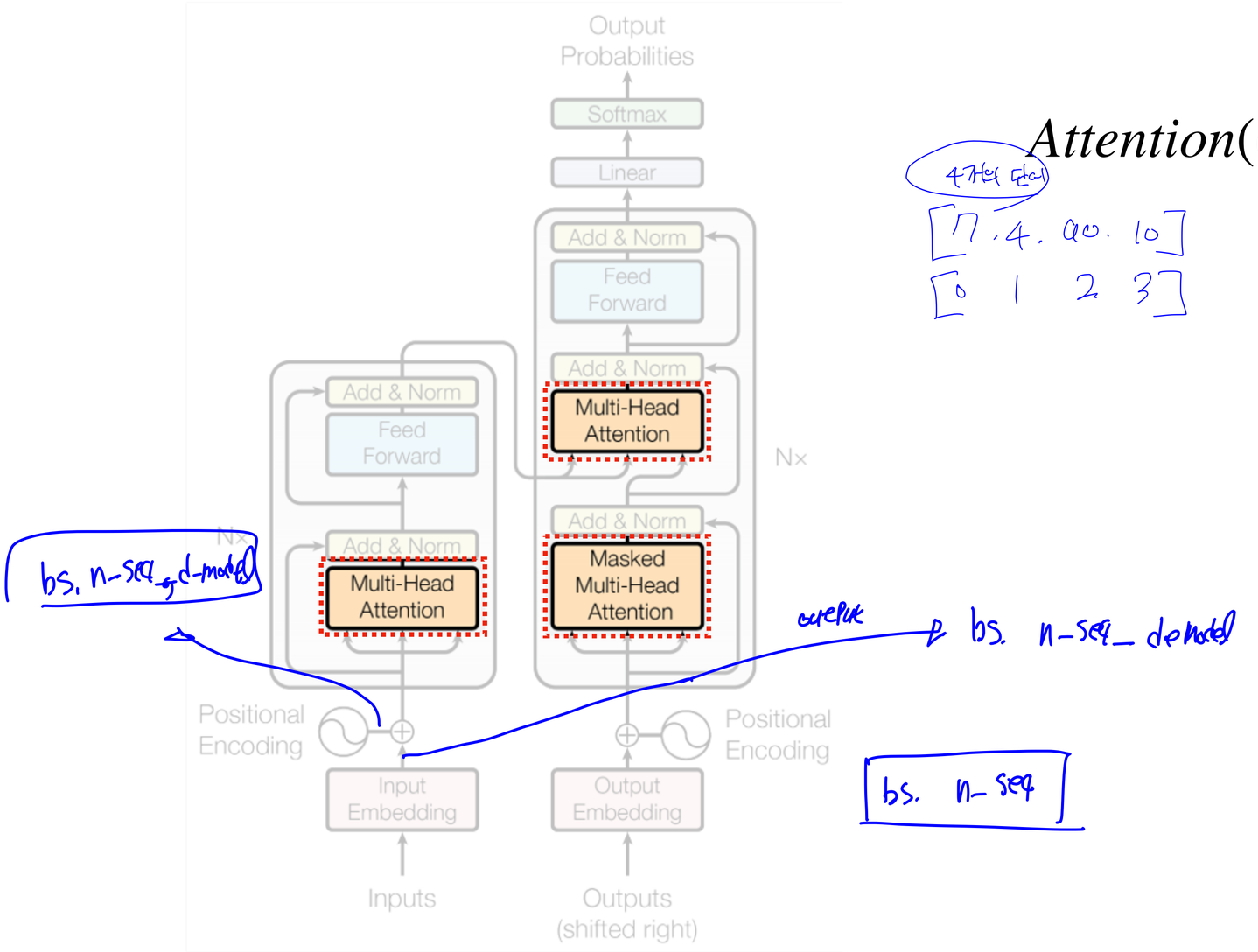
NMT



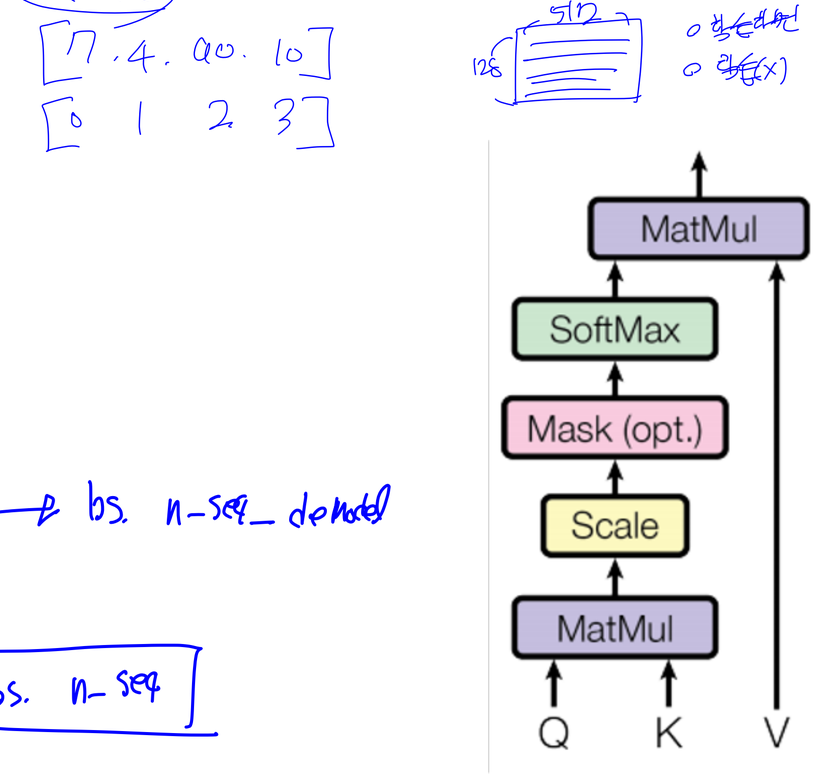
condition

Transformer Tutorial

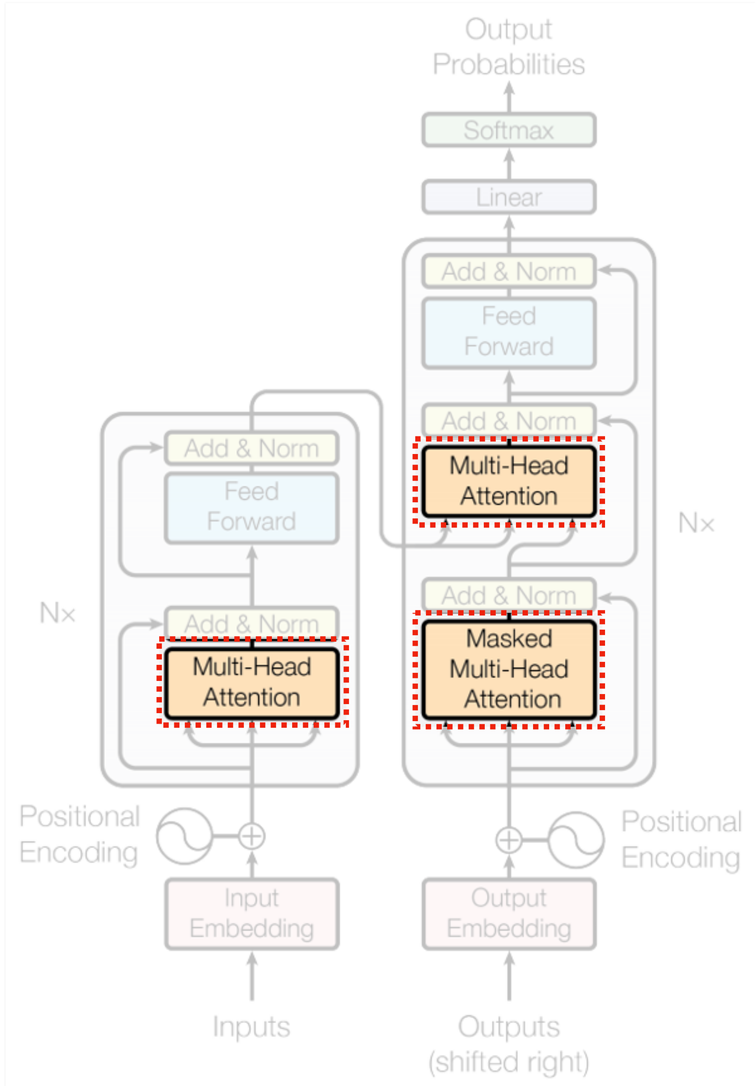
Transformer Tutorial (Scaled Dot-Product Attention)



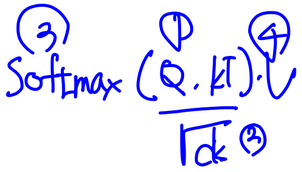
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

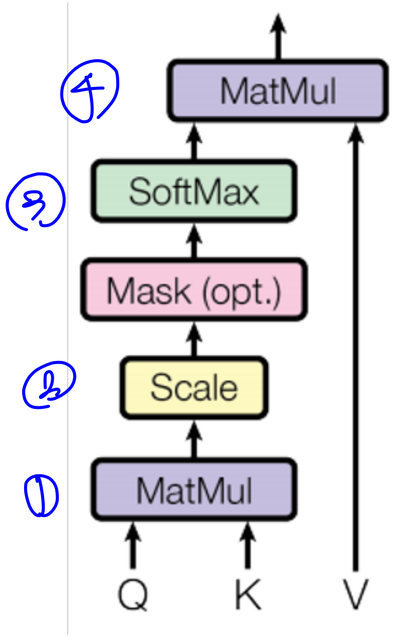


Transformer Tutorial (Scaled Dot-Product Attention)

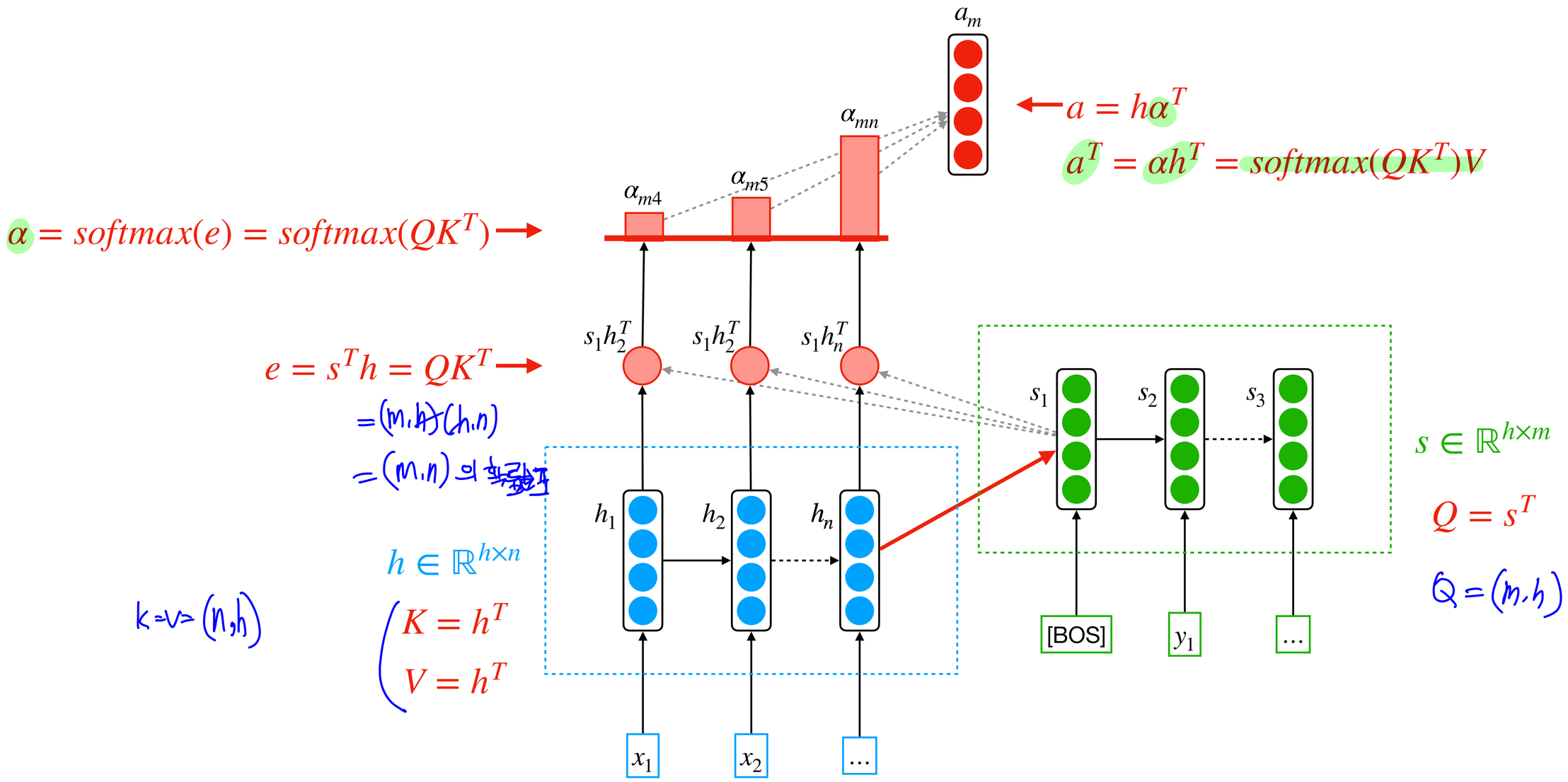


$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$





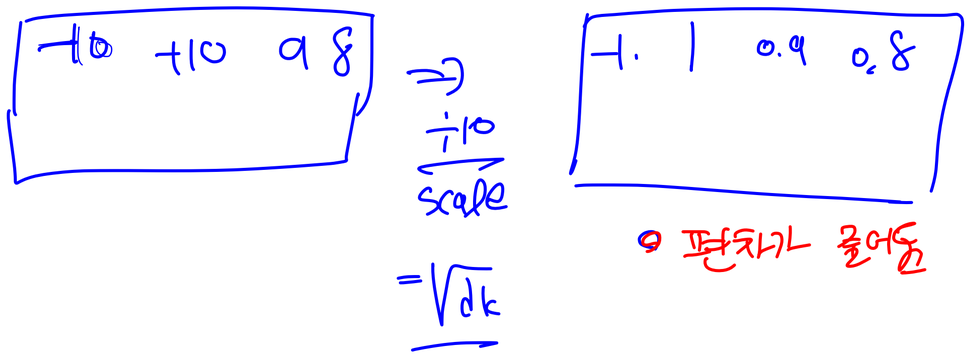
Transformer Tutorial (Scaled Dot-Product Attention)



Transformer Tutorial (Scaled Dot-Product Attention)

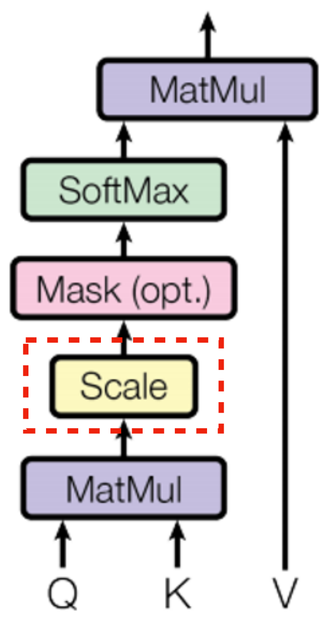
- **Dot-product Attention**
  - $Attention(Q, K, V) = softmax(QK^T)V$
    - $Q \in \mathbb{R}^{|Q| \times d_k}$
    - $K \in \mathbb{R}^{|K| \times d_k}$
    - $V \in \mathbb{R}^{|K| \times d_v}$
  - d_k가 커지면 QK^T의 결과값의 편차가 커짐
  - $softmax(QK^T)$의 결과 값이 편차가 커짐
  - Gradient가 작아짐
  - 학습이 잘 안됨

- **Scaled Dot-product Attention**
  - $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$
  - QK^T의 결과를 $\sqrt{d_k}$로 나눔
  - 값의 편차가 줄어듬



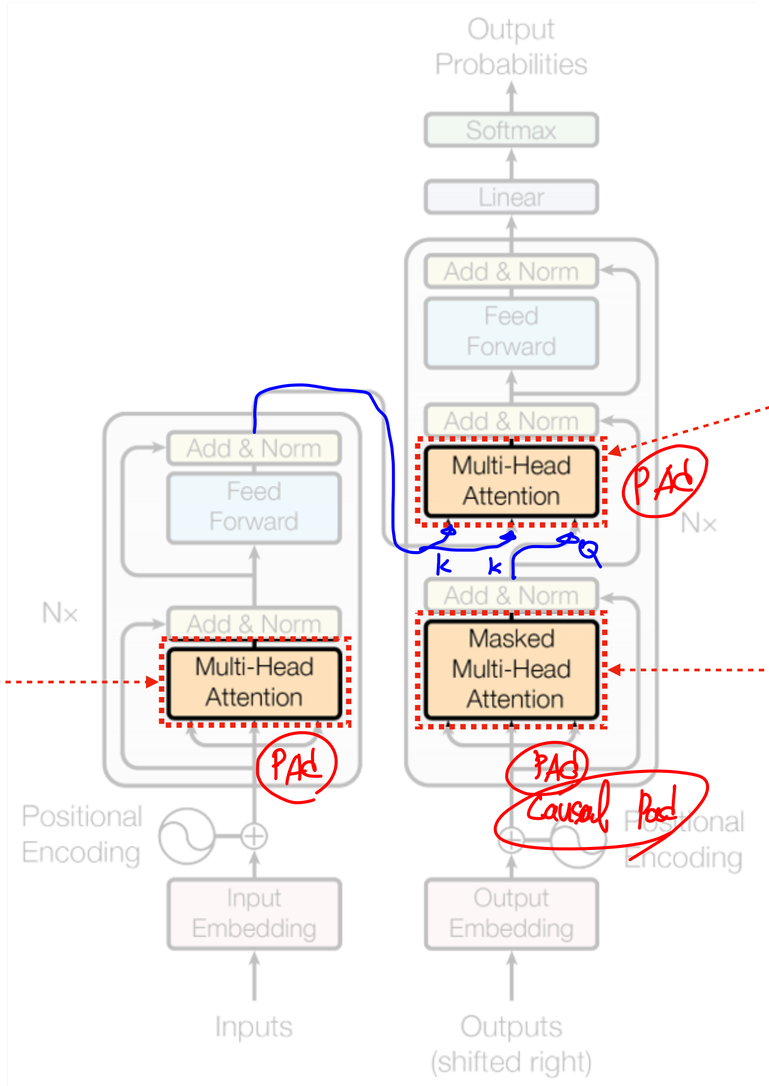
Transformer Tutorial (Scaled Dot-Product Attention)

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



Transformer Tutorial (Scaled Dot-Product Attention)

- **Encoder Self-Attention**
  - Q: encoder hidden
  - K: encoder hidden
  - V: ~~decoder~~ hidden

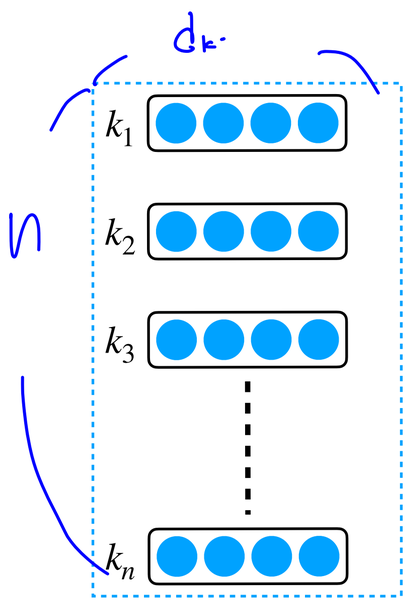


- **Encoder-Decoder Attention**
  - Q: decoder hidden
  - K: encoder output
  - V: encoder output

- **Decoder Self-Attention**
  - Q: decoder hidden
  - K: decoder hidden
  - V: decoder hidden

Transformer Tutorial (Scaled Dot-Product Attention)

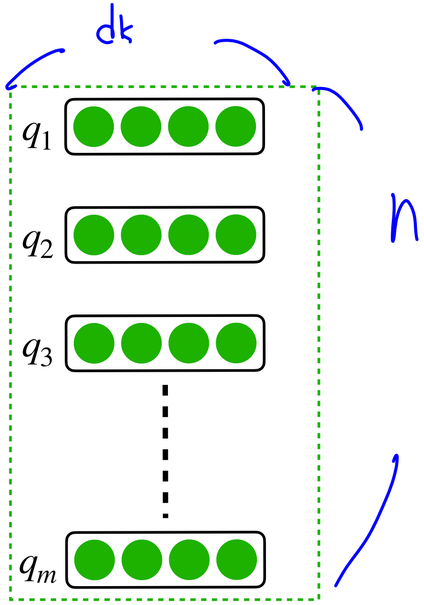
Encoder



$K \in \mathbb{R}^{n \times d_k}$

$V \in \mathbb{R}^{n \times d_k}$

Decoder



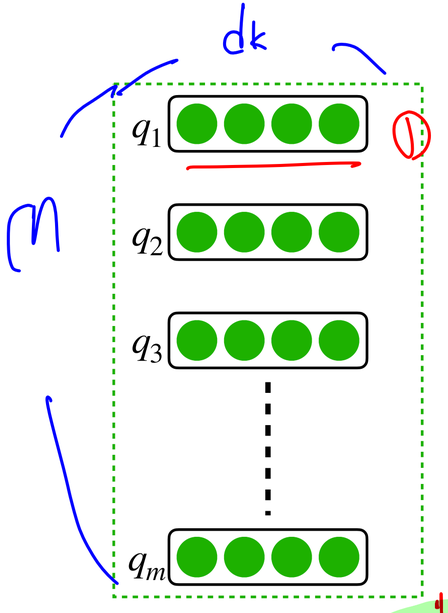
$Q \in \mathbb{R}^{m \times d_k}$

Transformer Tutorial (Scaled Dot-Product Attention)

I am a student Pad

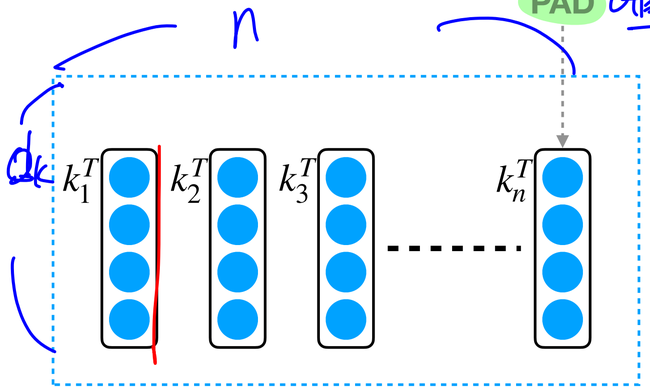
나는 학생 입니다 Pad

$QK^T \in \mathbb{R}^{m \times n}$



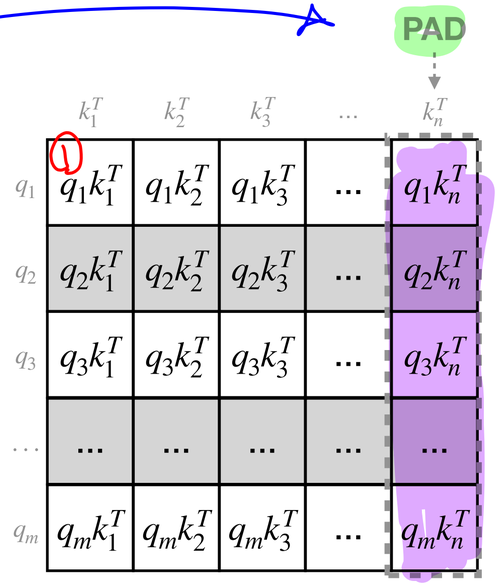
$Q \in \mathbb{R}^{m \times d_k}$ ◎ decoder

×

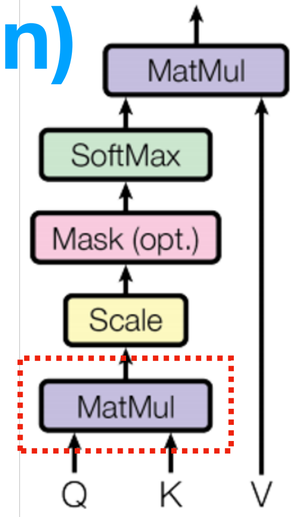


$K^T \in \mathbb{R}^{d_k \times n}$ ◎ Encoder

=



↳ 다 Pad 라이야!!



(m.dk) (dk.n)

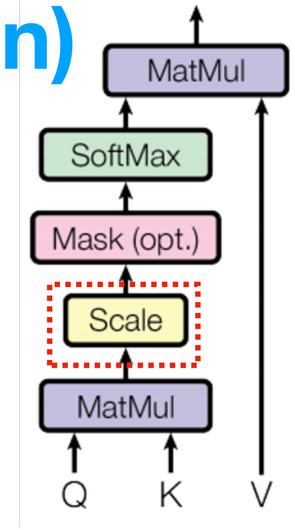
=(m,n)

Transformer Tutorial (Scaled Dot-Product Attention)

= Q k^T

= Decoder × Encoder^T
(S) (ch)

$$\frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{m \times n}$$



PAD

	k_1^T	k_2^T	k_3^T	...	k_n^T
q_1	$q_1k_1^T$	$q_1k_2^T$	$q_1k_3^T$	...	$q_1k_n^T$
q_2	$q_2k_1^T$	$q_2k_2^T$	$q_2k_3^T$	...	$q_2k_n^T$
q_3	$q_3k_1^T$	$q_3k_2^T$	$q_3k_3^T$	...	$q_3k_n^T$
...	...	...	...	...	...
q_m	$q_mk_1^T$	$q_mk_2^T$	$q_mk_3^T$	...	$q_mk_n^T$

/ $\sqrt{d_k}$ =

Scale 해주기.

PAD

	k_1^T	k_2^T	k_3^T	...	k_n^T
q_1	$\frac{q_1k_1^T}{\sqrt{d_k}}$	$\frac{q_1k_2^T}{\sqrt{d_k}}$	$\frac{q_1k_3^T}{\sqrt{d_k}}$	...	$\frac{q_1k_n^T}{\sqrt{d_k}}$
q_2	$\frac{q_2k_1^T}{\sqrt{d_k}}$	$\frac{q_2k_2^T}{\sqrt{d_k}}$	$\frac{q_2k_3^T}{\sqrt{d_k}}$	...	$\frac{q_2k_n^T}{\sqrt{d_k}}$
q_3	$\frac{q_3k_1^T}{\sqrt{d_k}}$	$\frac{q_3k_2^T}{\sqrt{d_k}}$	$\frac{q_3k_3^T}{\sqrt{d_k}}$	...	$\frac{q_3k_n^T}{\sqrt{d_k}}$
...	...	...	...	...	...
q_m	$\frac{q_mk_1^T}{\sqrt{d_k}}$	$\frac{q_mk_2^T}{\sqrt{d_k}}$	$\frac{q_mk_3^T}{\sqrt{d_k}}$	...	$\frac{q_mk_n^T}{\sqrt{d_k}}$

ex) 10 10 20 80 ..
:

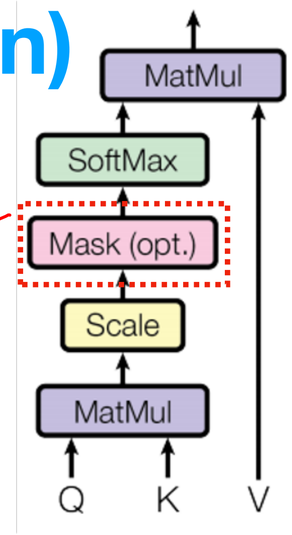
/ 10 =

1 1 2 3 ...

0 패딩가 좋아보이죠.

Transformer Tutorial (Scaled Dot-Product Attention)

$$mask\left(\frac{QK^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{m \times n}$$



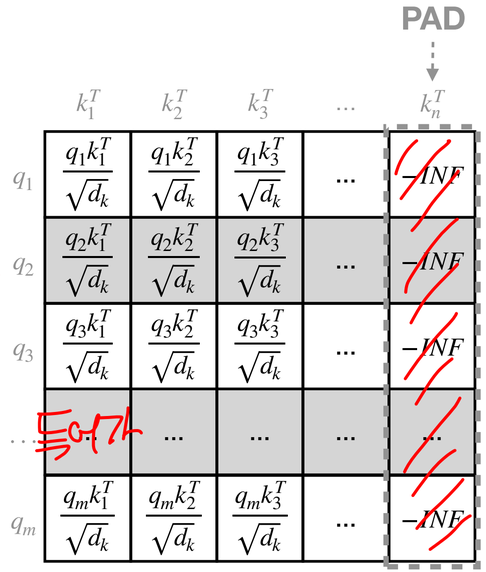
○ E-self

○ E-D

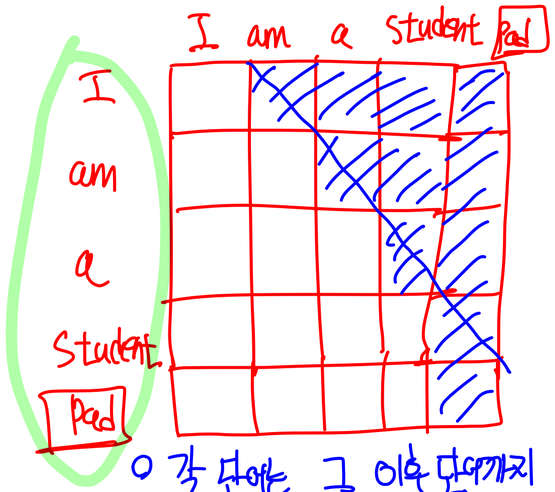
= Q K^T

= Decoder × $Encoder^T$
(S) (h)

○ K^T에다만 mask 등등



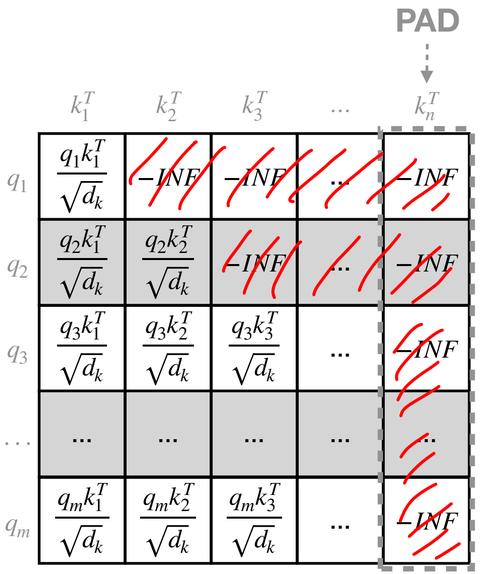
→ 이로 mask



○ 각 단어는 그 이후 단어까지 보지 않게 가려야까지

○ 다시봐야 하는 지점들.
⇒ mask

○ SxS



Masked Attention

○ Self-Decoded

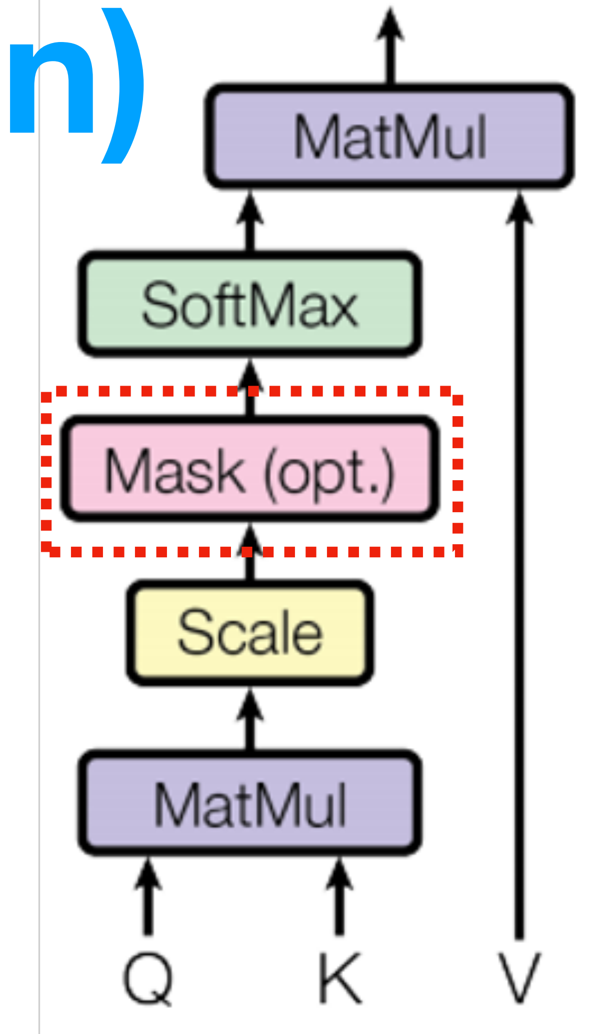
○ Decoder × $Encoder^T$
(S) (S)

○ 같은 거끼리만 [Pad]가 행, 열에 다있다!

SxS 인데
계산할 줄이야지

Soft (Q K^T) V
mask
loss

Transformer Tutorial (Scaled Dot-Product Attention)



$$mask\left(\frac{QK^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{m \times n}$$

loss : 0 → 0
O(x) → 1

Padd mask

expand_dim(mask, axis=1)

PAD

	k_1^T	k_2^T	k_3^T	…	k_n^T
q_1	$\frac{q_1k_1^T}{\sqrt{d_k}}$	$\frac{q_1k_2^T}{\sqrt{d_k}}$	$\frac{q_1k_3^T}{\sqrt{d_k}}$	…	$-INF$
q_2	$\frac{q_2k_1^T}{\sqrt{d_k}}$	$\frac{q_2k_2^T}{\sqrt{d_k}}$	$\frac{q_2k_3^T}{\sqrt{d_k}}$	…	$-INF$
q_3	$\frac{q_3k_1^T}{\sqrt{d_k}}$	$\frac{q_3k_2^T}{\sqrt{d_k}}$	$\frac{q_3k_3^T}{\sqrt{d_k}}$	…	$-INF$
…	…	…	…	…	…
q_m	$\frac{q_mk_1^T}{\sqrt{d_k}}$	$\frac{q_mk_2^T}{\sqrt{d_k}}$	$\frac{q_mk_3^T}{\sqrt{d_k}}$	…	$-INF$

(m, n)

(1, n)

PAD

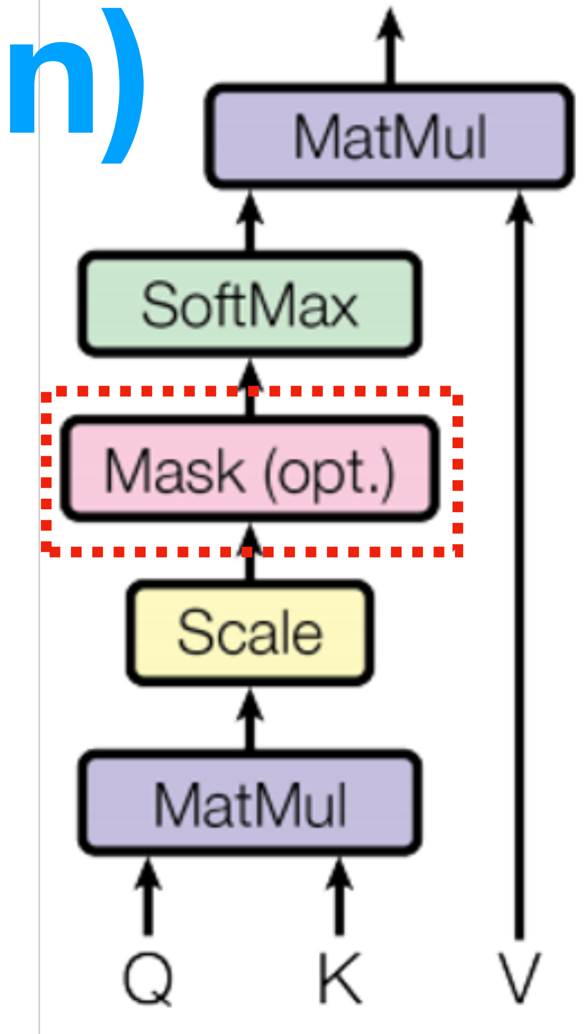
	k_1^T	k_2^T	k_3^T	…	k_n^T
q_1	$\frac{q_1k_1^T}{\sqrt{d_k}}$	$-INF$	$-INF$	…	$-INF$
q_2	$\frac{q_2k_1^T}{\sqrt{d_k}}$	$\frac{q_2k_2^T}{\sqrt{d_k}}$	$-INF$	…	$-INF$
q_3	$\frac{q_3k_1^T}{\sqrt{d_k}}$	$\frac{q_3k_2^T}{\sqrt{d_k}}$	$\frac{q_3k_3^T}{\sqrt{d_k}}$	…	$-INF$
…	…	…	…	…	…
q_m	$\frac{q_mk_1^T}{\sqrt{d_k}}$	$\frac{q_mk_2^T}{\sqrt{d_k}}$	$\frac{q_mk_3^T}{\sqrt{d_k}}$	…	$-INF$

Masked Attention

Attention mask

0 → 1
O(x) → 0

Transformer Tutorial (Scaled Dot-Product Attention)



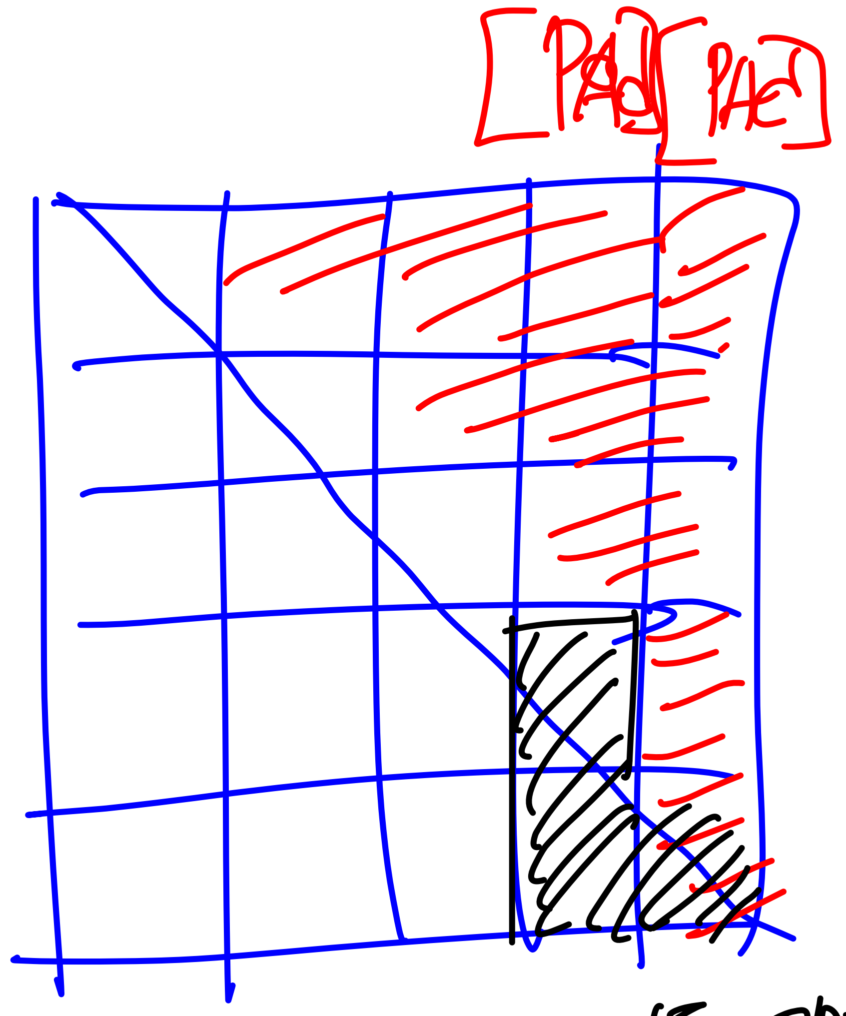
$$mask\left(\frac{QK^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{m \times n}$$

causal_mask

- decoder_self attention에 적용
- → weight sum에서 0처리
- → 미래 단방향을 보지 않게

PAD

	k_1^T	k_2^T	k_3^T	…	k_n^T
q_1	$\frac{q_1k_1^T}{\sqrt{d_k}}$	$\frac{q_1k_2^T}{\sqrt{d_k}}$	$\frac{q_1k_3^T}{\sqrt{d_k}}$	…	$-INF$
q_2	$\frac{q_2k_1^T}{\sqrt{d_k}}$	$\frac{q_2k_2^T}{\sqrt{d_k}}$	$\frac{q_2k_3^T}{\sqrt{d_k}}$	…	$-INF$
q_3	$\frac{q_3k_1^T}{\sqrt{d_k}}$	$\frac{q_3k_2^T}{\sqrt{d_k}}$	$\frac{q_3k_3^T}{\sqrt{d_k}}$	…	$-INF$
…	…	…	…	…	…
q_m	$\frac{q_mk_1^T}{\sqrt{d_k}}$	$\frac{q_mk_2^T}{\sqrt{d_k}}$	$\frac{q_mk_3^T}{\sqrt{d_k}}$	…	$-INF$



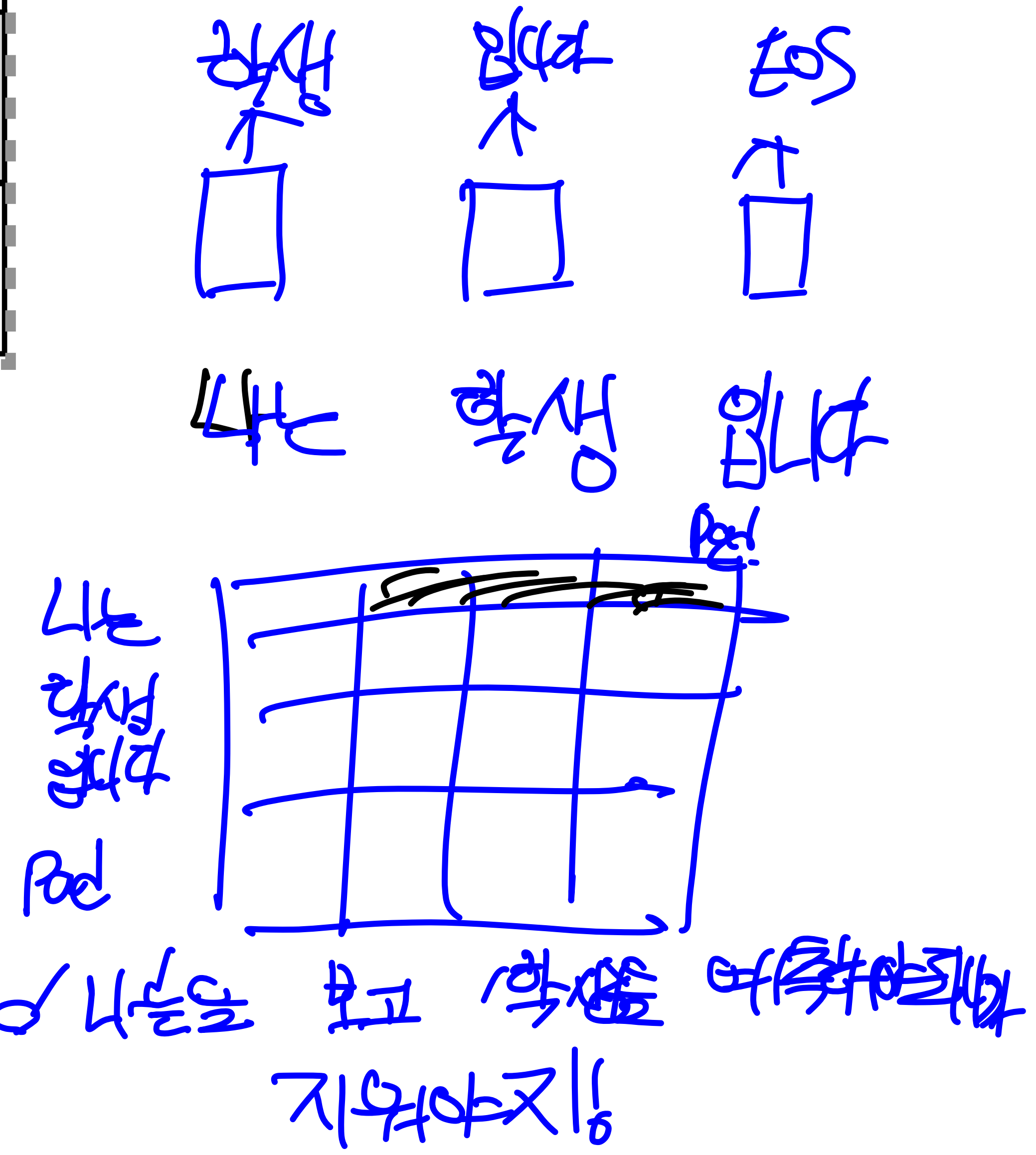
PAD

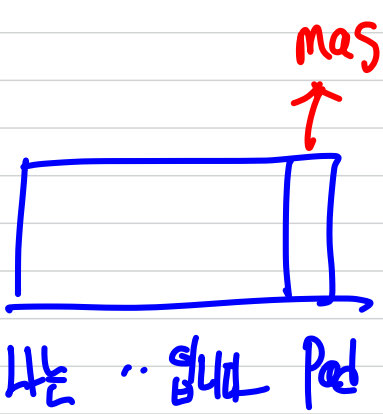
	k_1^T	k_2^T	k_3^T	…	k_n^T
q_1	$\frac{q_1k_1^T}{\sqrt{d_k}}$	$-INF$	$-INF$	…	$-INF$
q_2	$\frac{q_2k_1^T}{\sqrt{d_k}}$	$\frac{q_2k_2^T}{\sqrt{d_k}}$	$-INF$	…	$-INF$
q_3	$\frac{q_3k_1^T}{\sqrt{d_k}}$	$\frac{q_3k_2^T}{\sqrt{d_k}}$	$\frac{q_3k_3^T}{\sqrt{d_k}}$	…	$-INF$
…	…	…	…	…	…
q_m	$\frac{q_mk_1^T}{\sqrt{d_k}}$	$\frac{q_mk_2^T}{\sqrt{d_k}}$	$\frac{q_mk_3^T}{\sqrt{d_k}}$	…	$-INF$

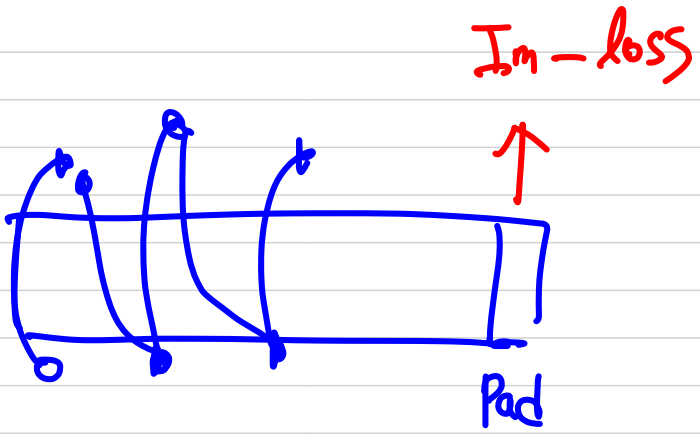
Masked Attention

causal mask

- → weight sum에서 0처리
- → 미래 단방향을 보지 않게

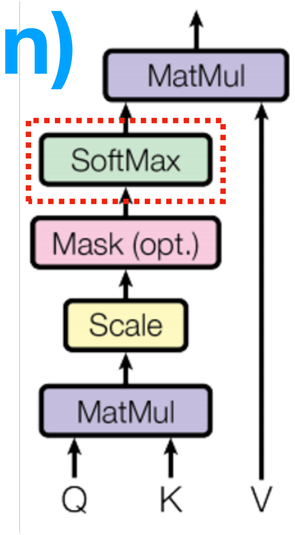


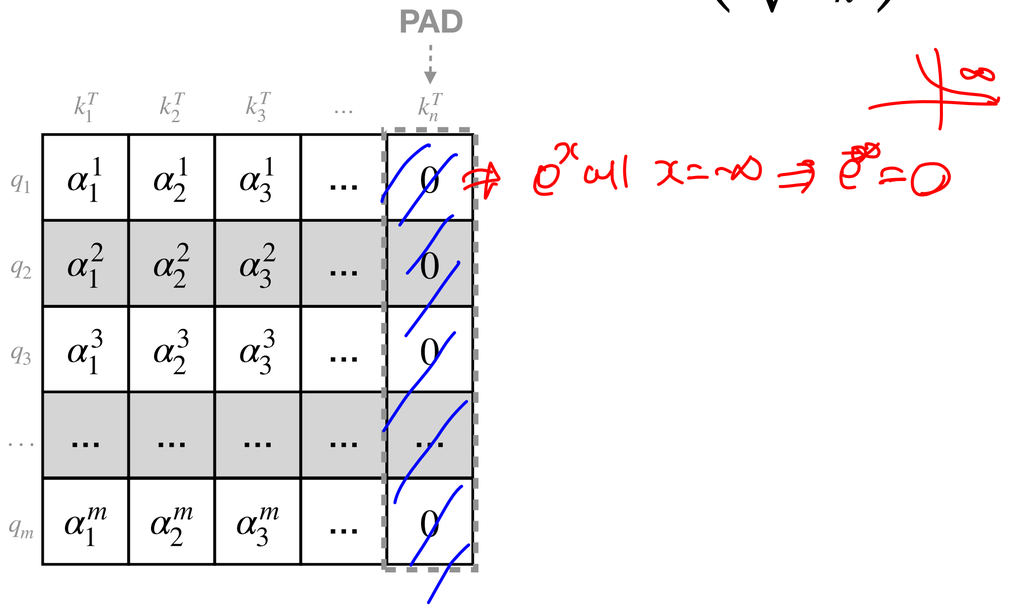

mask
나는 .. 실제 Pad

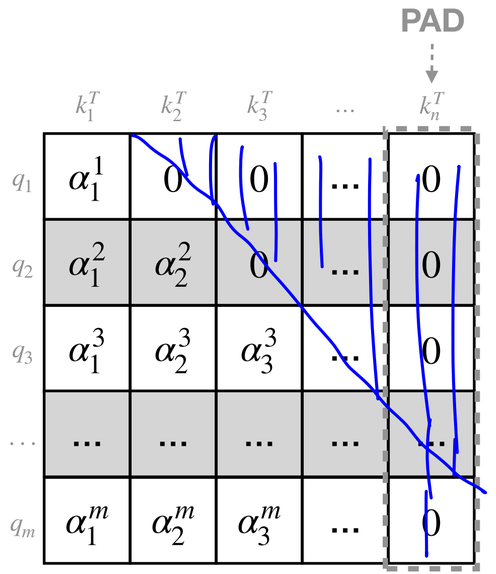

Im - loss
Pad

Transformer Tutorial (Scaled Dot-Product Attention)

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{m \times n}$$



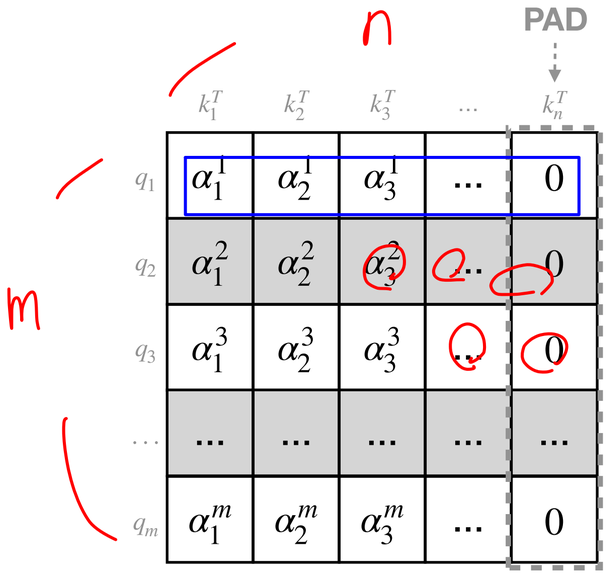




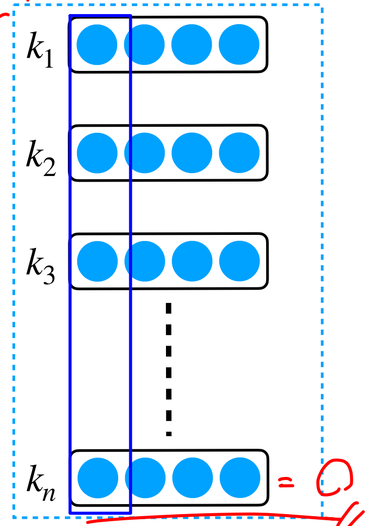
Masked Attention

Transformer Tutorial (Scaled Dot-Product Attention)

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in \mathbb{R}^{m \times d_k}$$

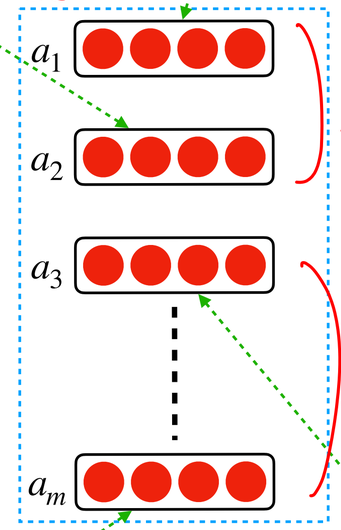


×



$V \in \mathbb{R}^{n \times d_k}$

=



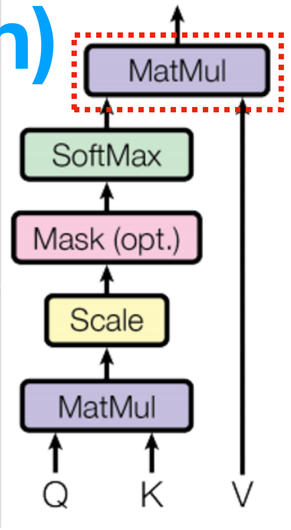
$A \in \mathbb{R}^{m \times d_k}$

$\alpha_1^1 k_1 + \alpha_2^1 k_2 + \alpha_3^1 k_3 + \ldots + \alpha_n^1 k_n$

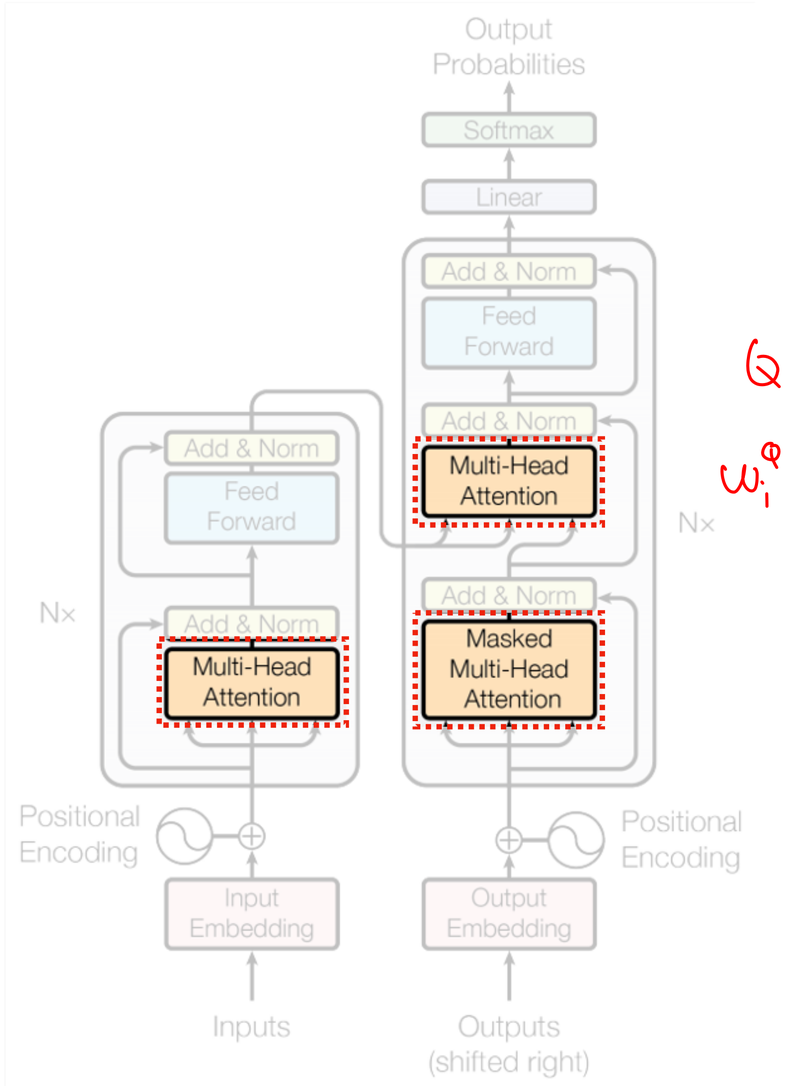
$\alpha_1^2 k_1 + \alpha_2^2 k_2 + \alpha_3^2 k_3 + \ldots + \alpha_n^2 k_n$

$\alpha_1^3 k_1 + \alpha_2^3 k_2 + \alpha_3^3 k_3 + \ldots + \alpha_n^3 k_n$

$\alpha_1^m k_1 + \alpha_2^m k_2 + \alpha_3^m k_3 + \ldots + \alpha_n^m k_n$

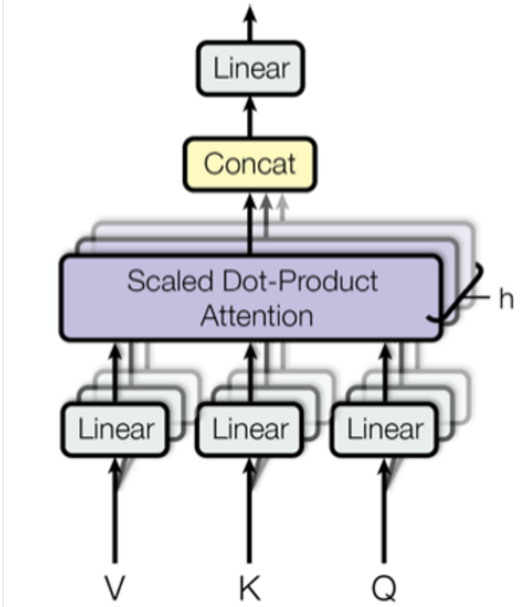


Transformer Tutorial (Multi-Head Attention)

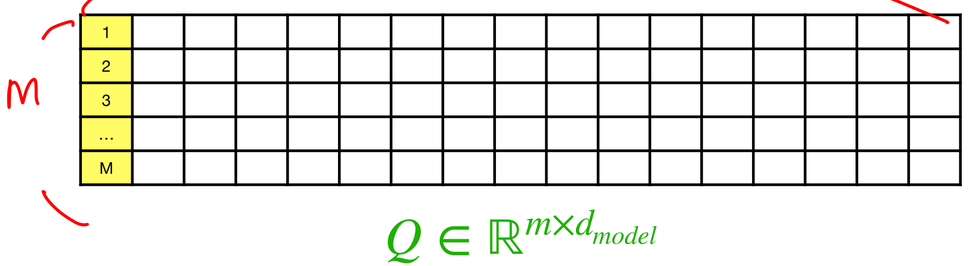


$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^o$$

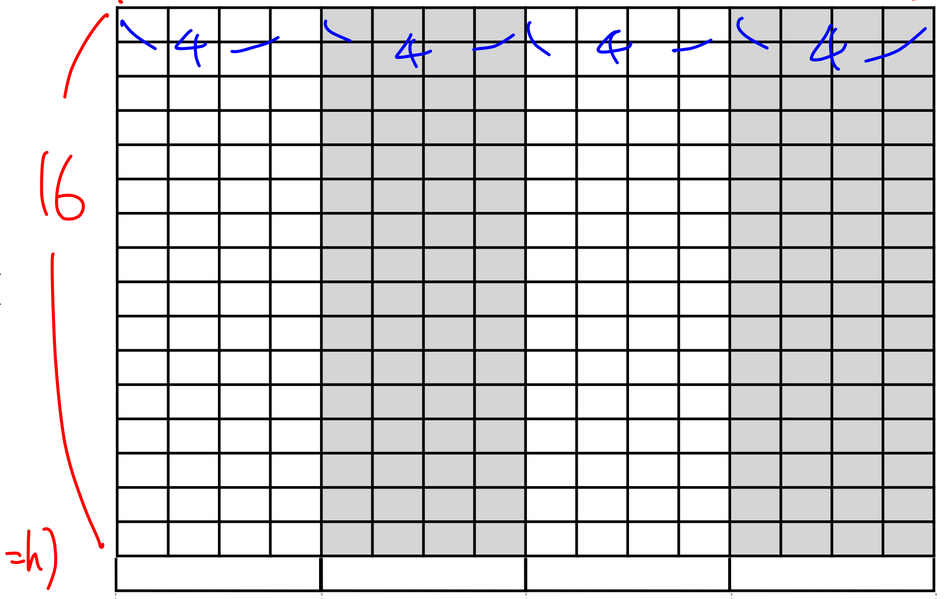
$$where \quad head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$



Transformer Tutorial (Multi-Head Attention)

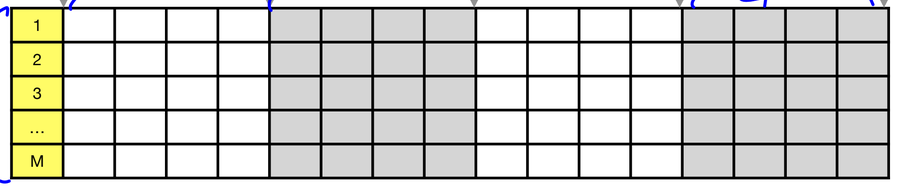


$Q \in \mathbb{R}^{m \times d_{model}}$

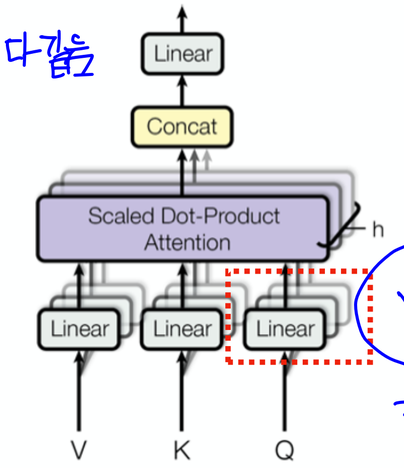


W_1^Q W_2^Q ... W_h^Q $\in \mathbb{R}^{d_{model} \times d_k}$

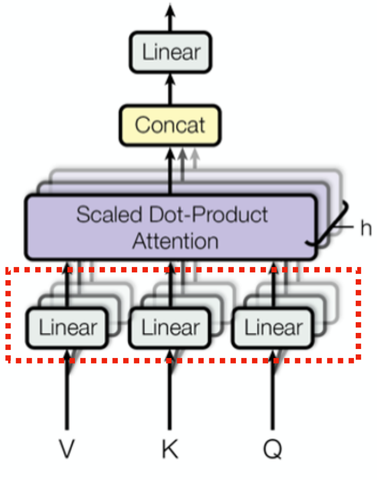
=



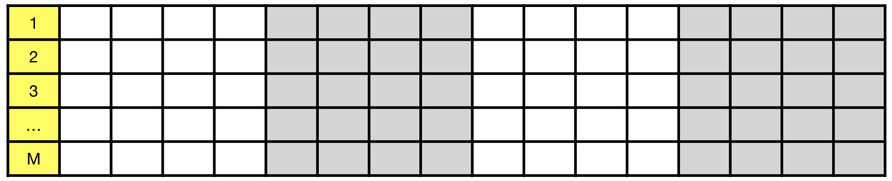
$Q_m \in \mathbb{R}^{m \times hd_k}$



Transformer Tutorial (Multi-Head Attention)

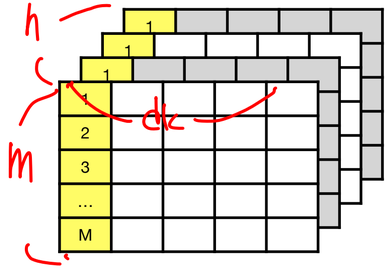


$h=4$.
$d_{k\_head} = d_k = 4$

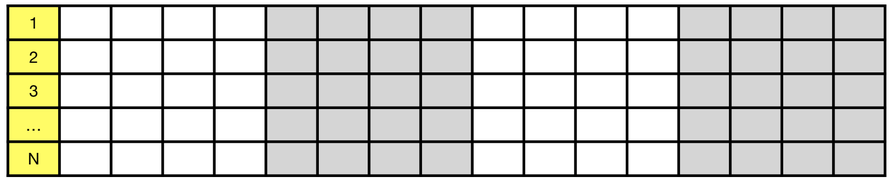


$Q_m \in \mathbb{R}^{m \times hd_k}$

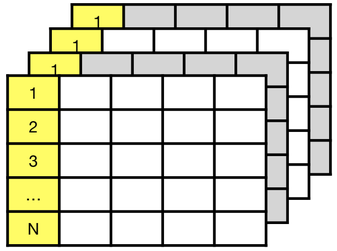
(m. 4x4 = 16)
(m. hx dk) → (m. h. dk) → (h, m. dk)



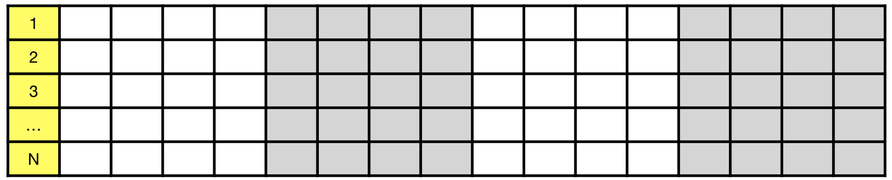
$Q_m \in \mathbb{R}^{h \times m \times d_k}$



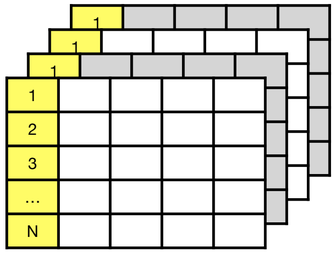
$K_m \in \mathbb{R}^{n \times hd_k}$



$K_m \in \mathbb{R}^{h \times n \times d_k}$



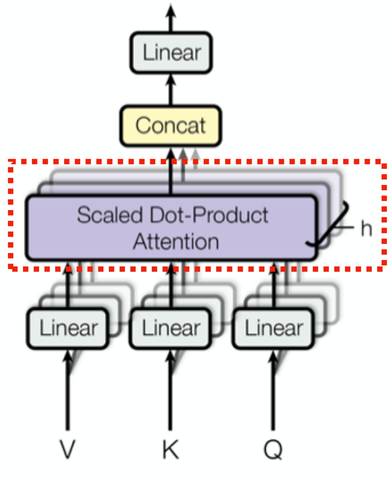
$V_m \in \mathbb{R}^{n \times hd_k}$

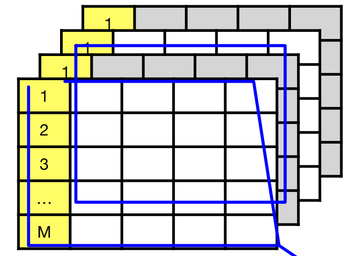


$V_m \in \mathbb{R}^{h \times n \times d_k}$

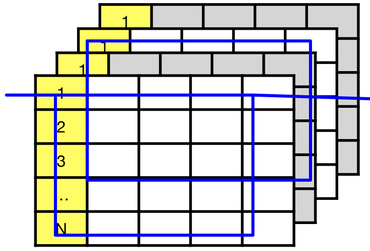
Transformer Tutorial (Multi-Head Attention)

9 Multi-Head 된 Attention

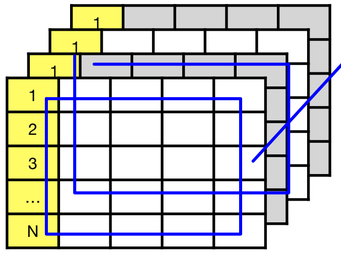




$Q_m \in \mathbb{R}^{h \times m \times d_k}$

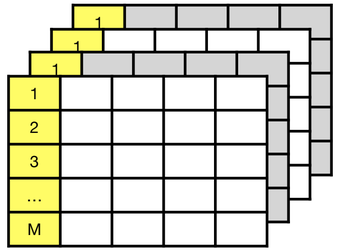


$K_m \in \mathbb{R}^{h \times n \times d_k}$



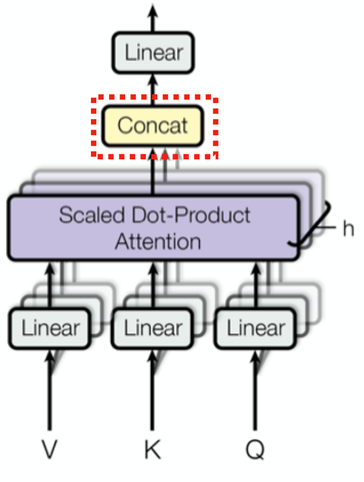
$V_m \in \mathbb{R}^{h \times n \times d_k}$

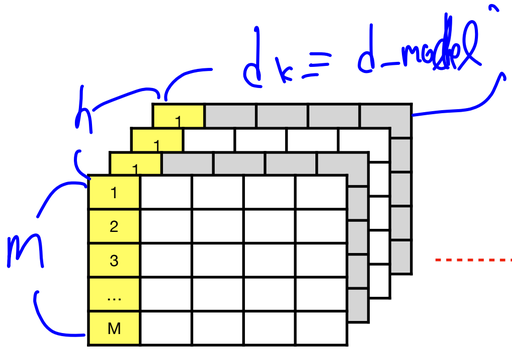
$$Attention(Q_m, K_m, V_m)$$



$H_m \in \mathbb{R}^{h \times m \times d_k}$

Transformer Tutorial (Multi-Head Attention)

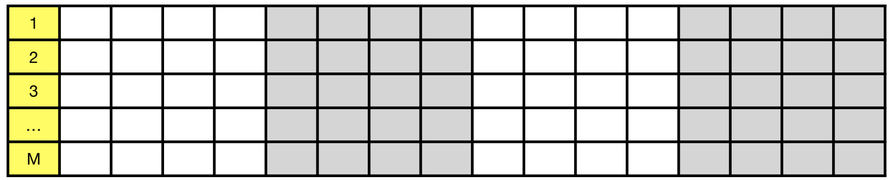




$d_k = d\_model$

$H_m \in \mathbb{R}^{h \times m \times d_k}$

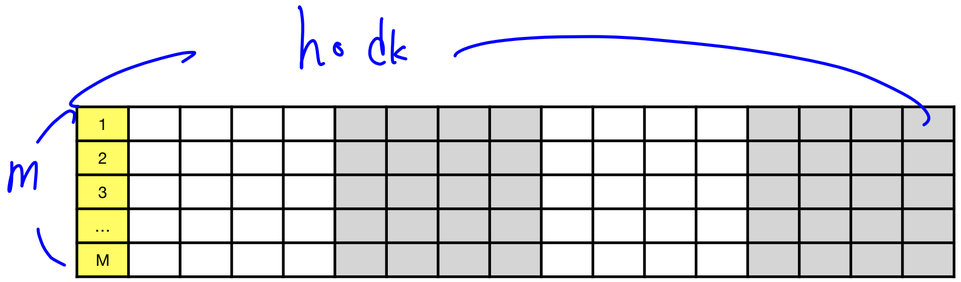
$$Concat(head_1, \ldots, head_h)$$



$H_m \in \mathbb{R}^{m \times hd_k}$

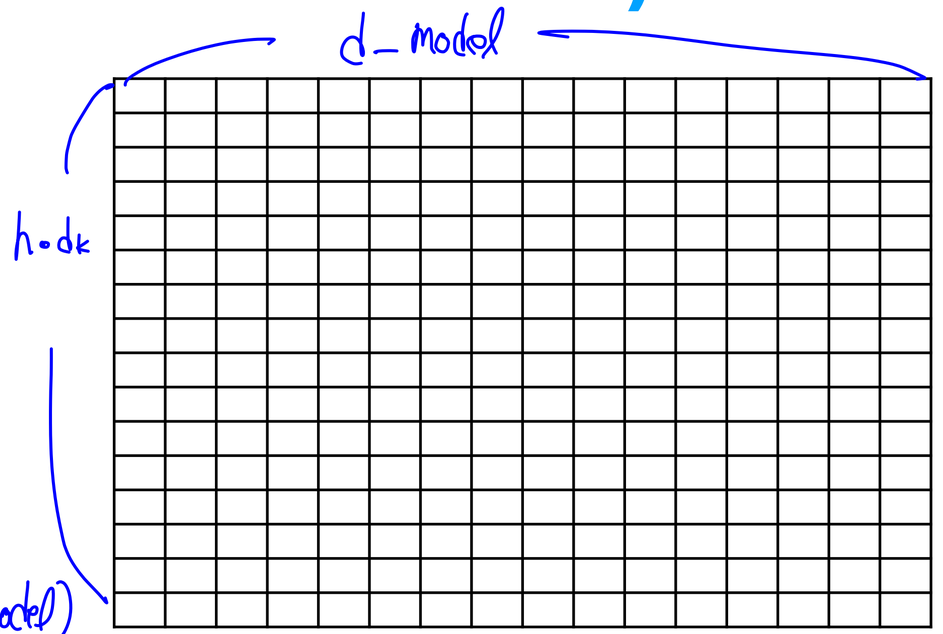
$(h, m, d_k) \rightarrow (m, hd_k) \rightarrow (m, h \times d_k)$

Transformer Tutorial (Multi-Head Attention)



$H_m \in \mathbb{R}^{m \times hd_k}$

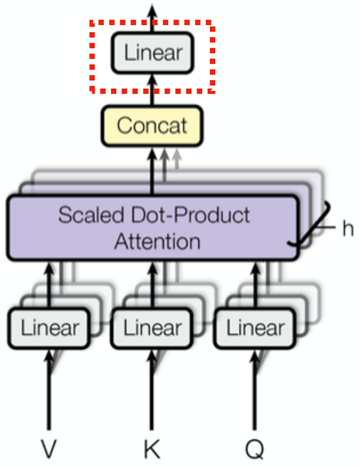
×



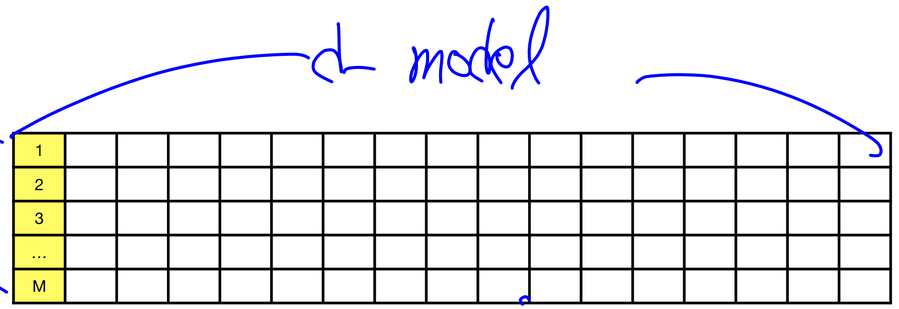
$W^O \in \mathbb{R}^{hd_k \times d_{model}}$

$(m, hd_k)(hd_k, d\_model)$

$= (m, d\_model)$

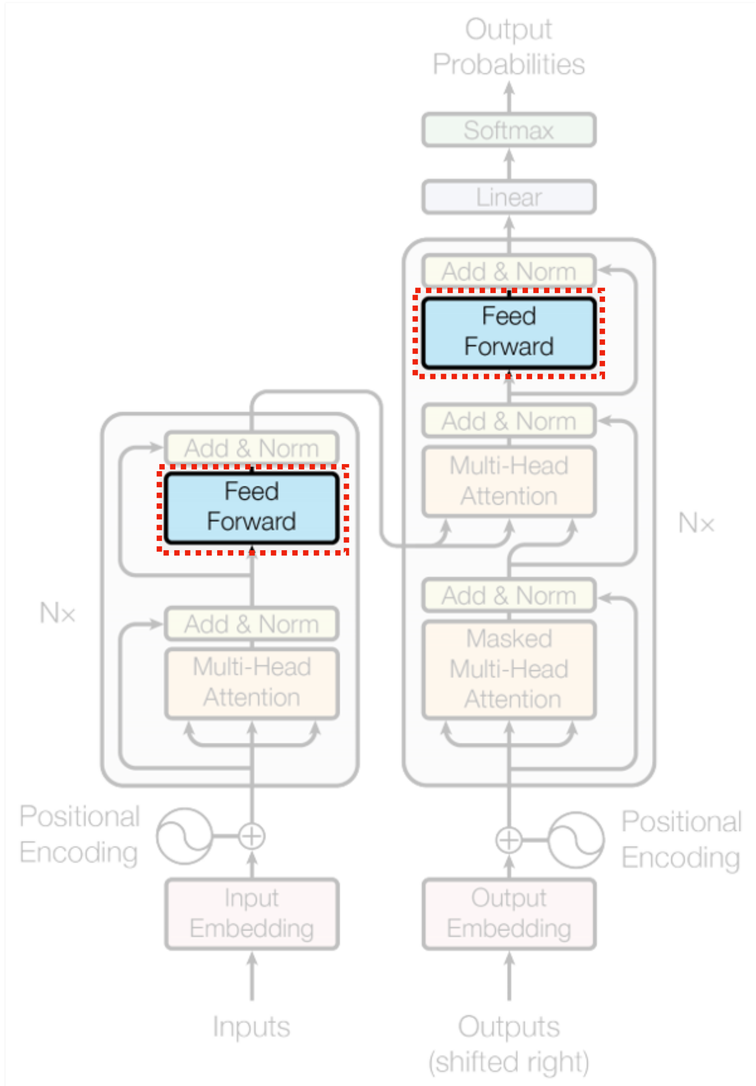


=

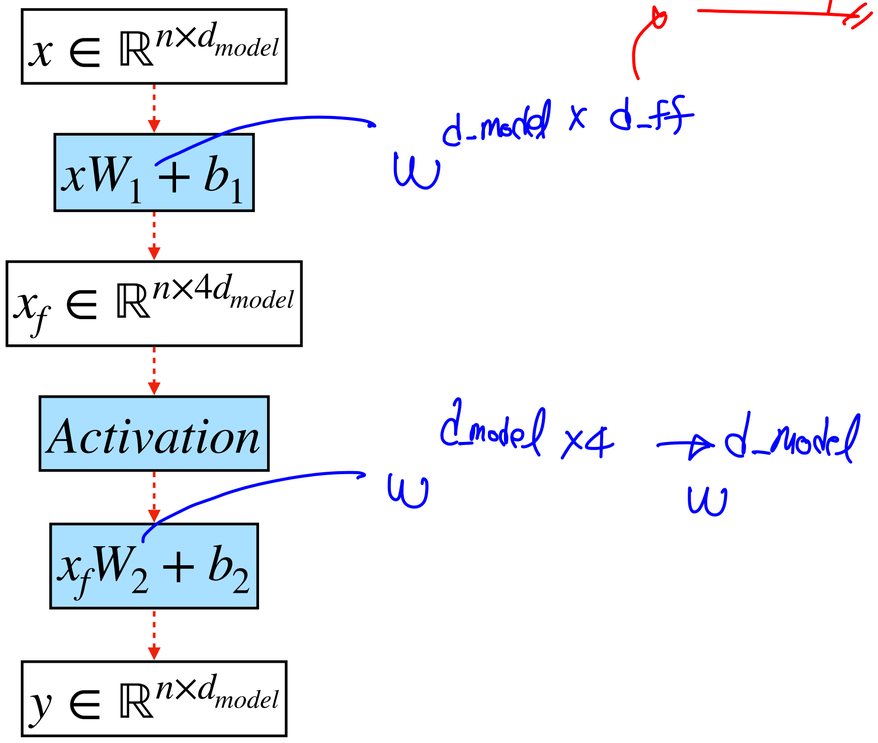


$A \in \mathbb{R}^{m \times d_{model}}$

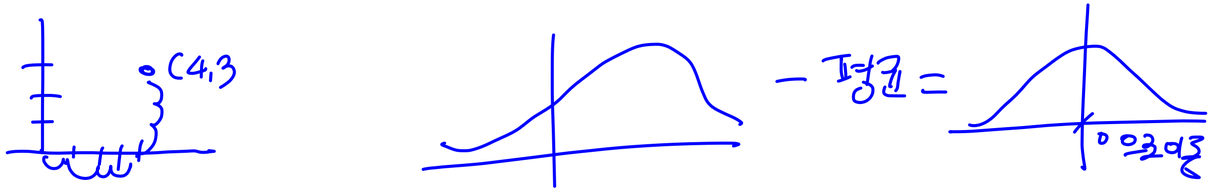
Transformer Tutorial (Feed-Forward)



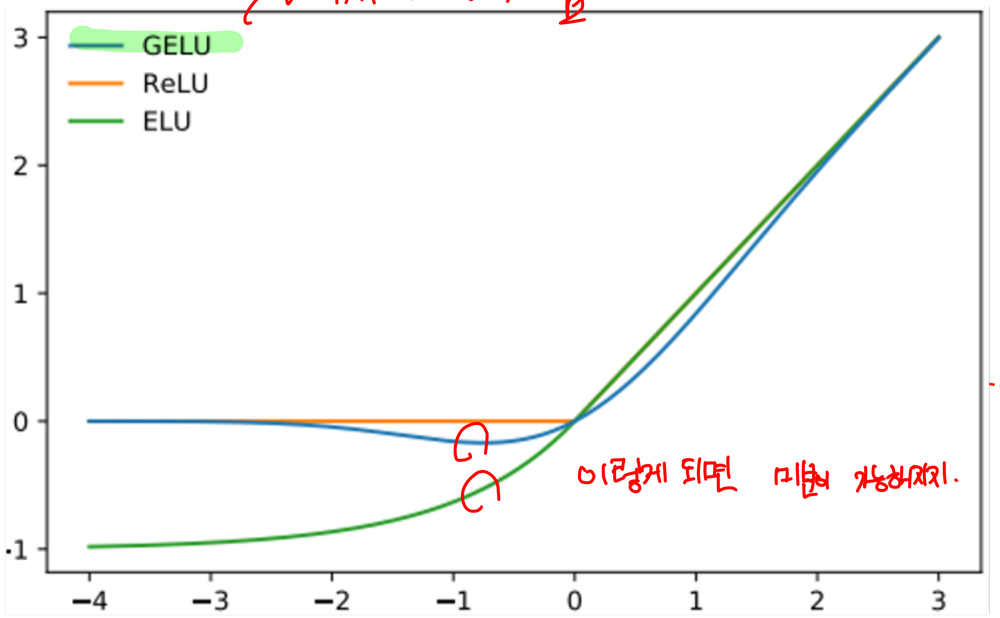
$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

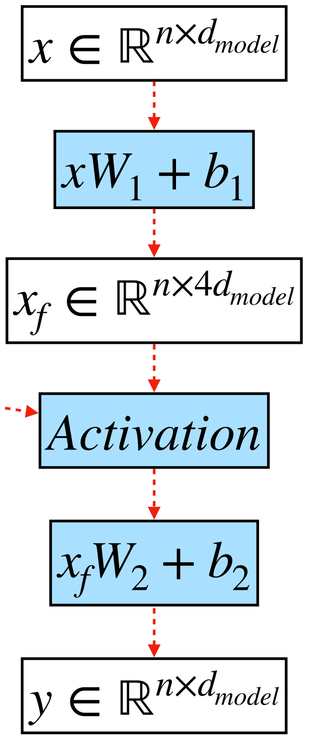


Transformer Tutorial (Feed-Forward)

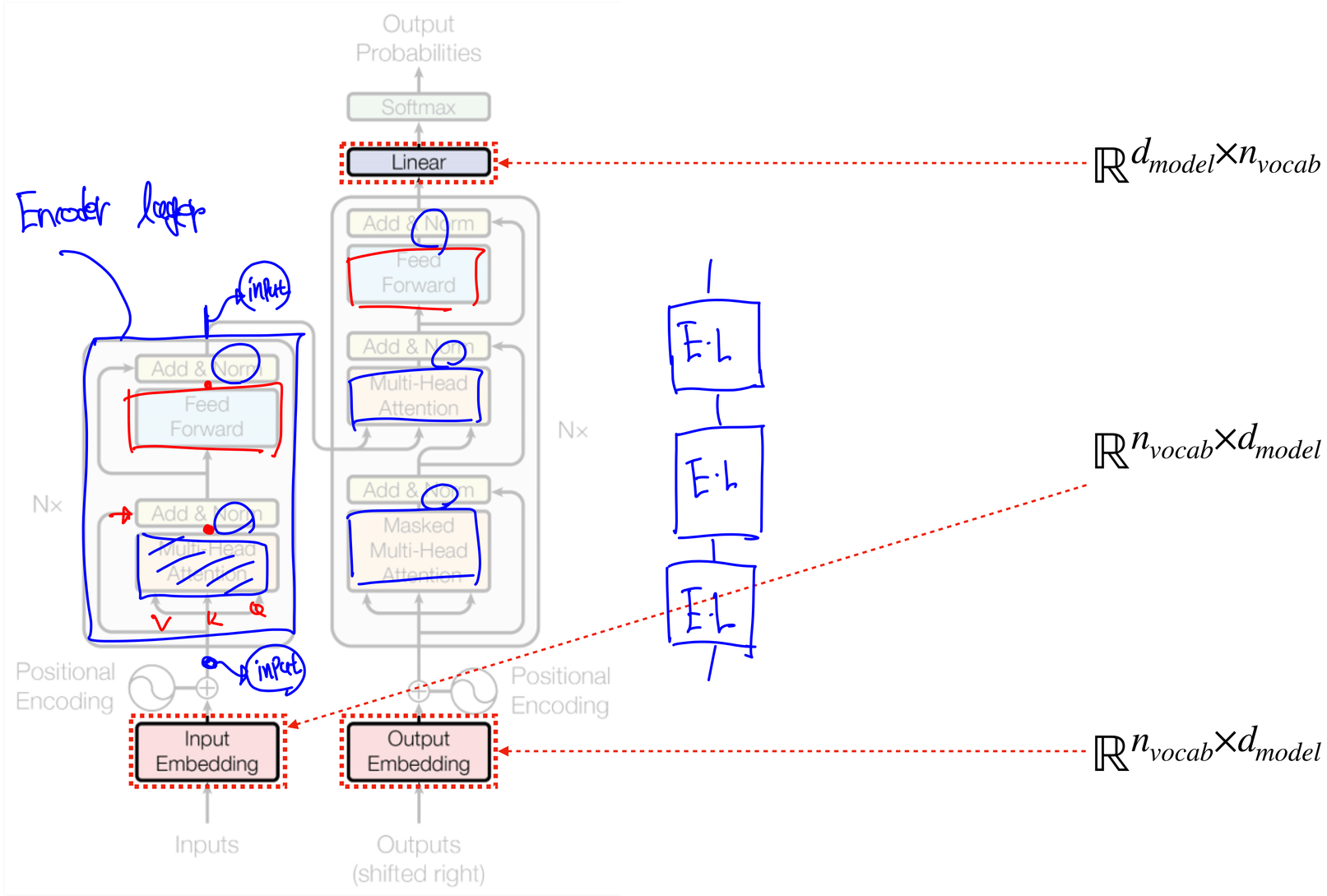


$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

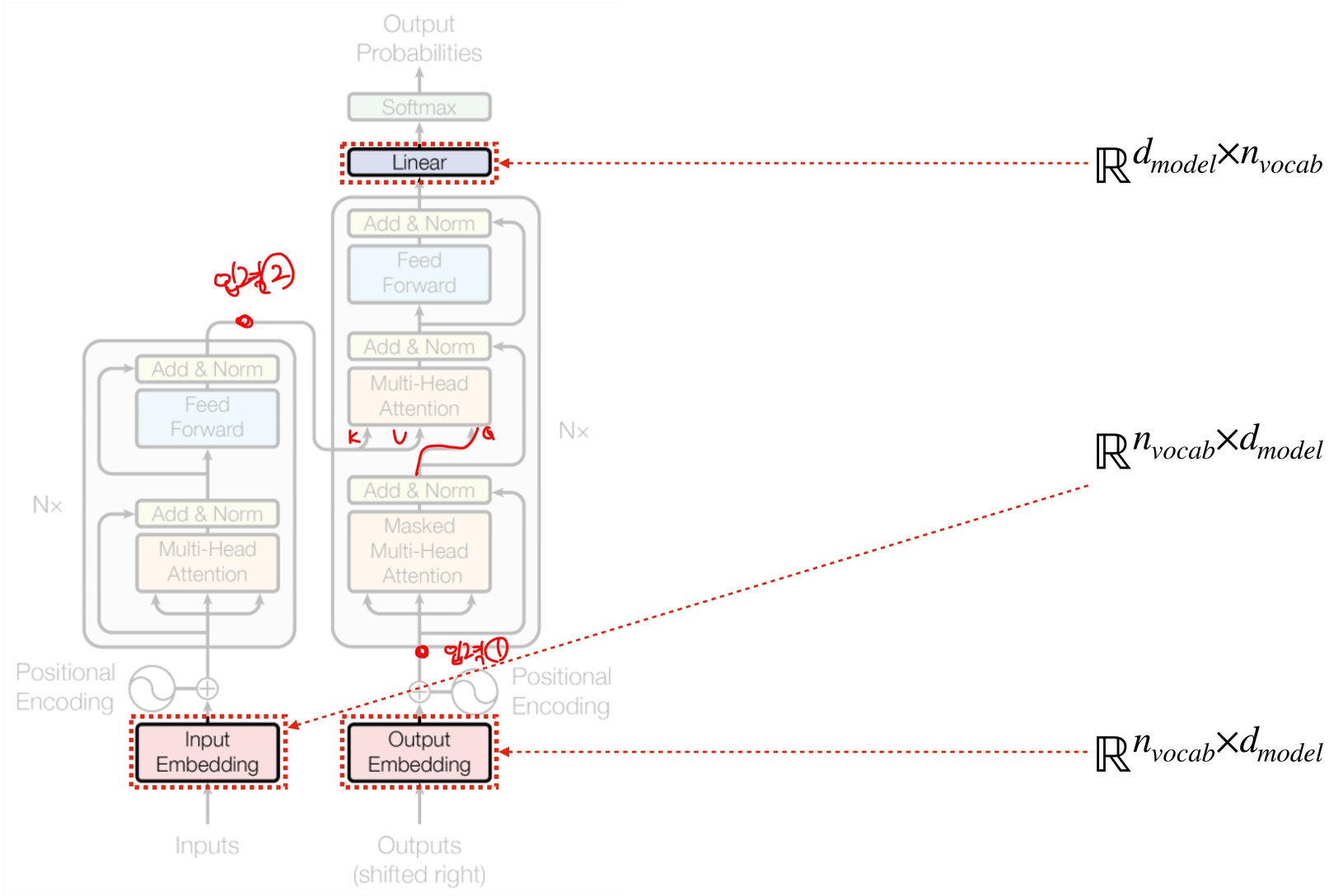




Transformer Tutorial (Weight Shared Embedding)



Transformer Tutorial (Weight Shared Embedding)

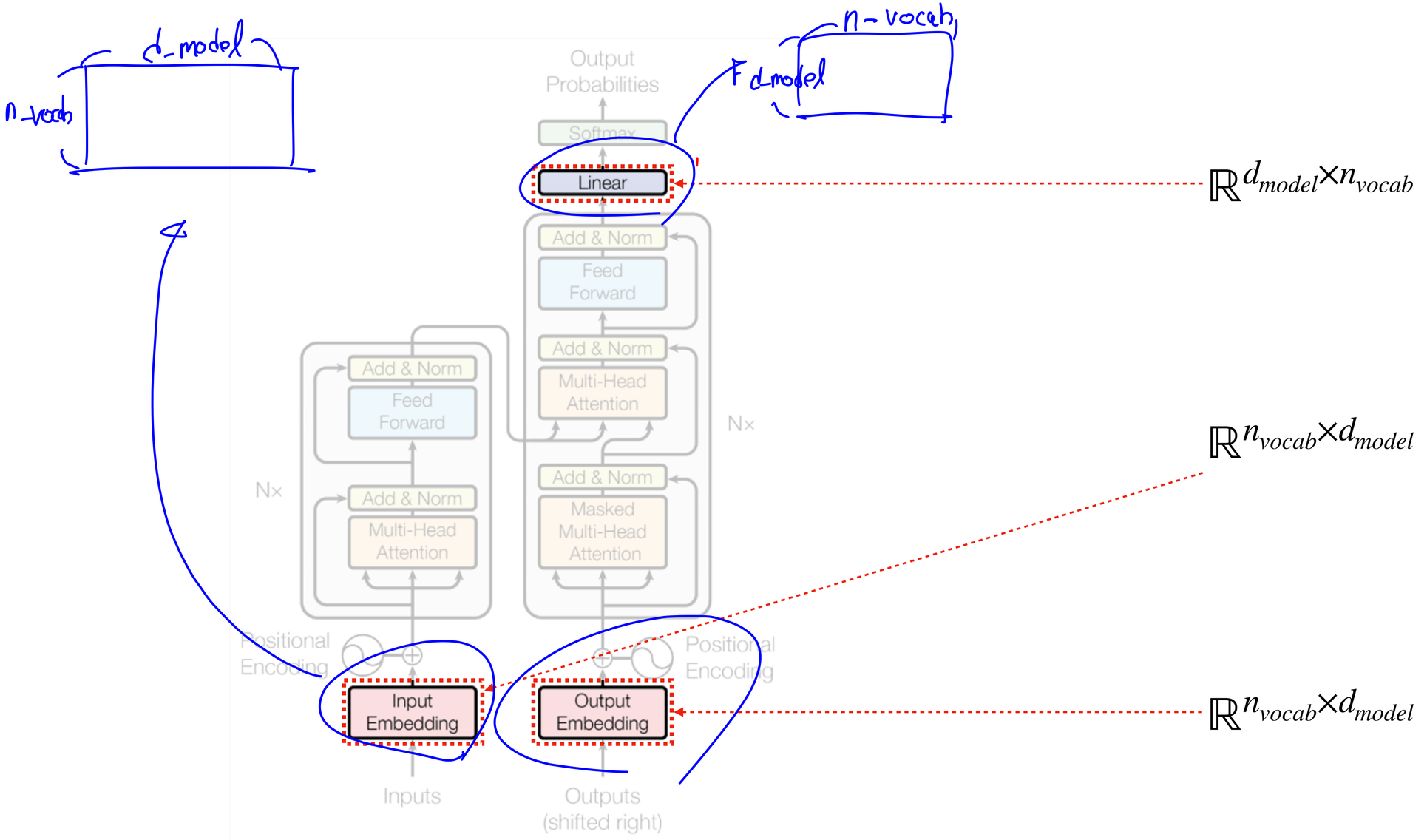


$\mathbb{R}^{d_{model} \times n_{vocab}}$

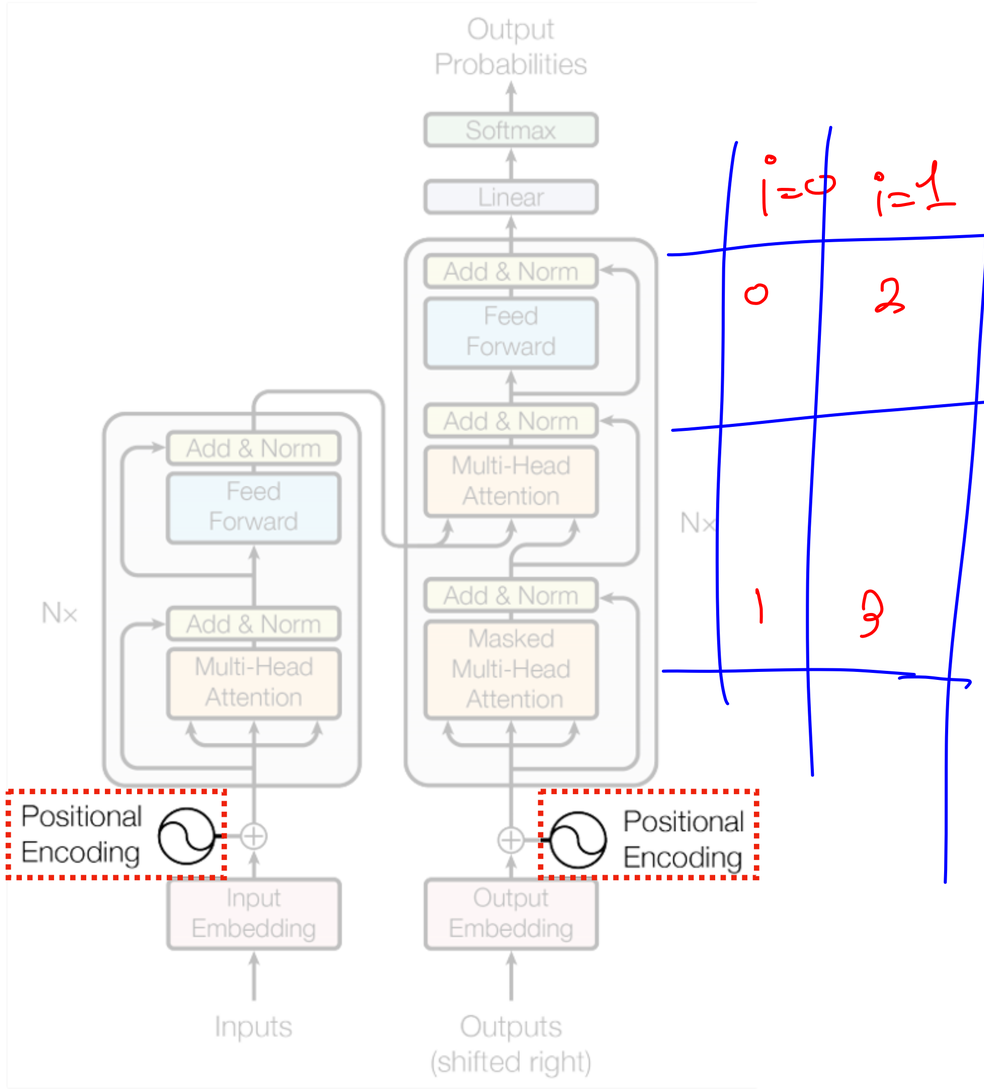
$\mathbb{R}^{n_{vocab} \times d_{model}}$

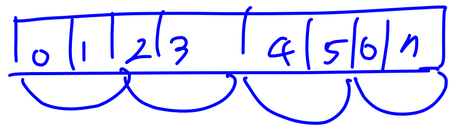
$\mathbb{R}^{n_{vocab} \times d_{model}}$

Transformer Tutorial (Weight Shared Embedding)



Transformer Tutorial (Positional Encoding)





짝수 0

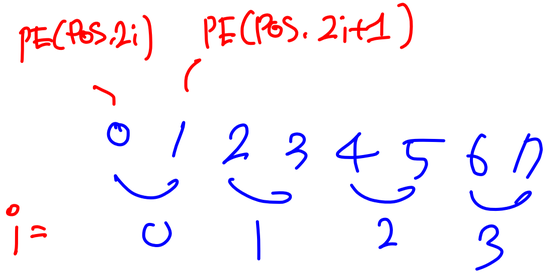
$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

- i//2 ⇒ 몫만 나열
- x2 ⇒ 0.0.2.2.4.4

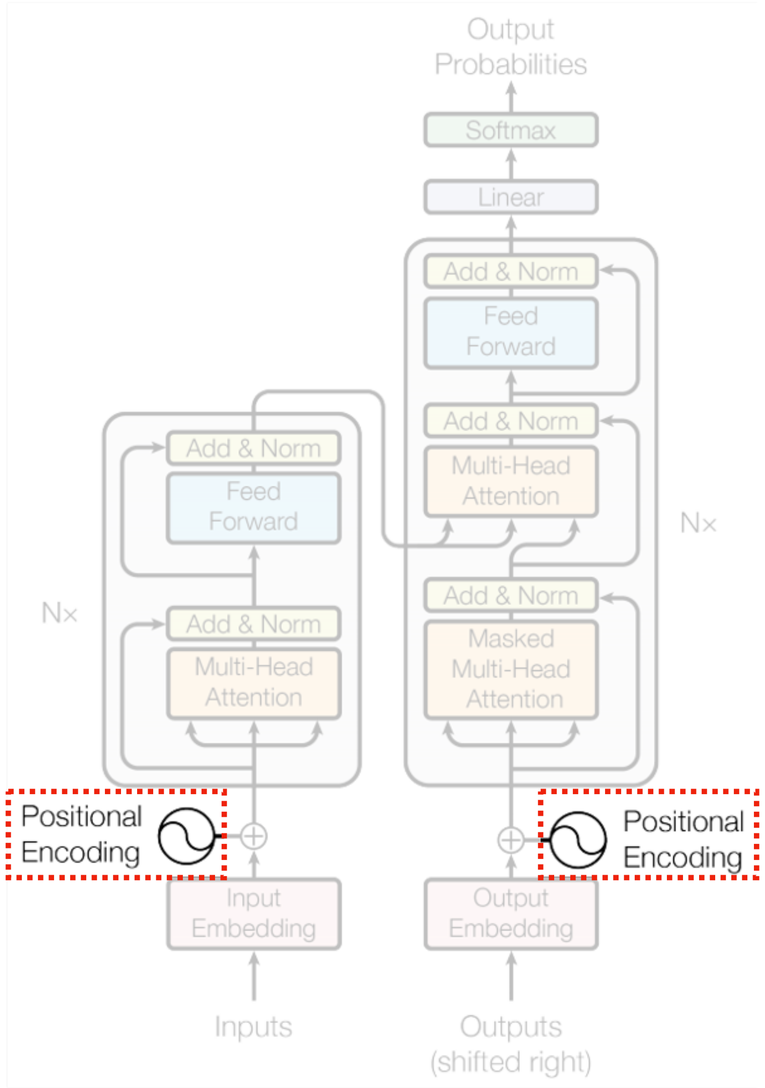
홀수 1

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

- 위와 d_{model} 나눈 2번째부터.



Transformer Tutorial (Positional Encoding)



$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Transformer Tutorial (Positional Encoding)

$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

i $\quad$ $where \;\; d_{model} = 8$

pos	0	1	2	3	4	5	6	7
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1.0	1.0	0.1	0.1	0.01	0.01	0.001	0.001
2	2.0	2.0	0.2	0.2	0.02	0.02	0.002	0.002
3	3.0	3.0	0.3	0.3	0.03	0.03	0.003	0.003
4	4.0	4.0	0.4	0.4	0.04	0.04	0.004	0.004
5	5.0	5.0	0.5	0.5	0.05	0.05	0.005	0.005

Transformer Tutorial (Positional Encoding)

$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

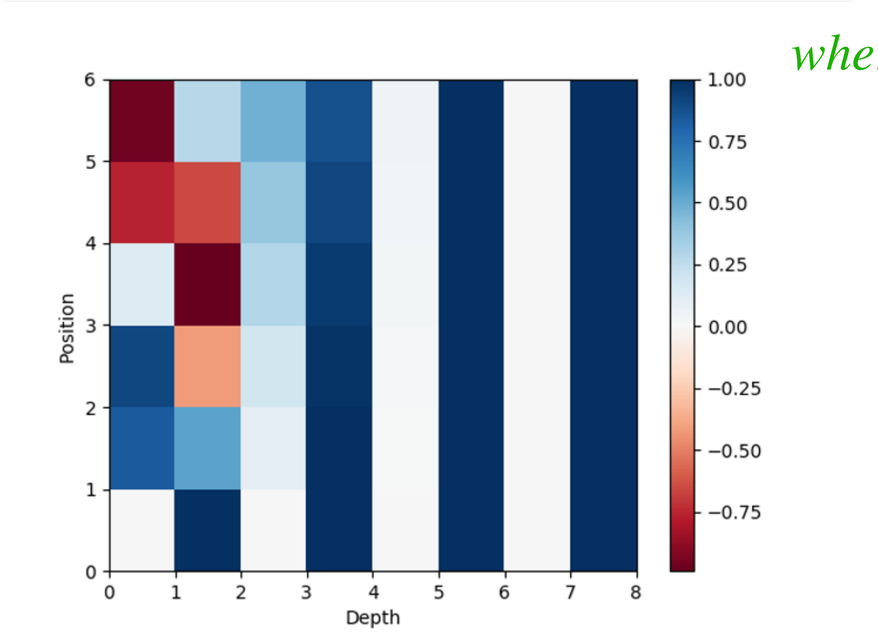
i $\quad$ $where \;\; d_{model} = 8$

pos	0	1	2	3	4	5	6	7
0	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000
1	0.8415	0.5403	0.9008	0.9950	0.0100	1.000	0.0010	1.0000
2	0.9903	-0.4161	0.1987	0.9801	0.0200	0.9998	0.0020	1.0000
3	0.1411	-0.9900	0.2955	0.9553	0.0300	0.9996	0.0030	1.0000
4	-0.756	-0.6536	0.3894	0.9211	0.0400	0.9992	0.0040	1.0000
5	-0.959	0.2837	0.4794	0.8776	0.0500	0.9988	0.0050	1.0000

Transformer Tutorial (Positional Encoding)

$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$



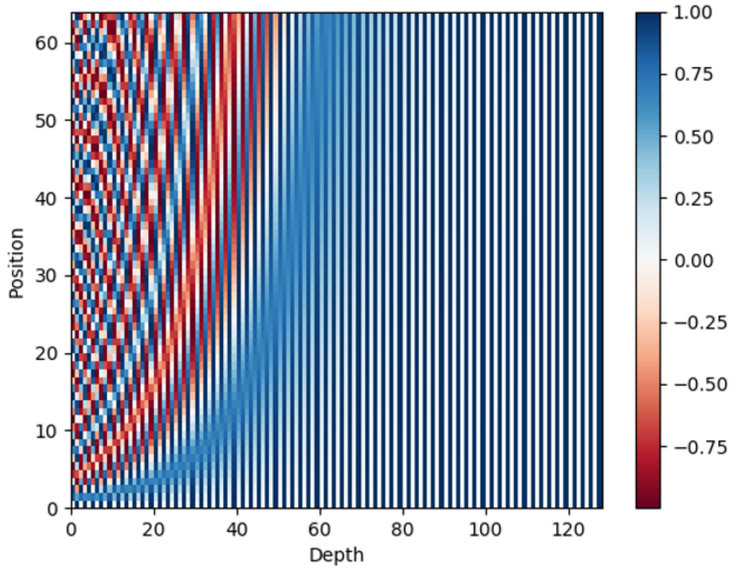
$where \;\; d_{model} = 8$

Transformer Tutorial (Positional Encoding)

$$PE(pos,2i) = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos,2i+1) = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

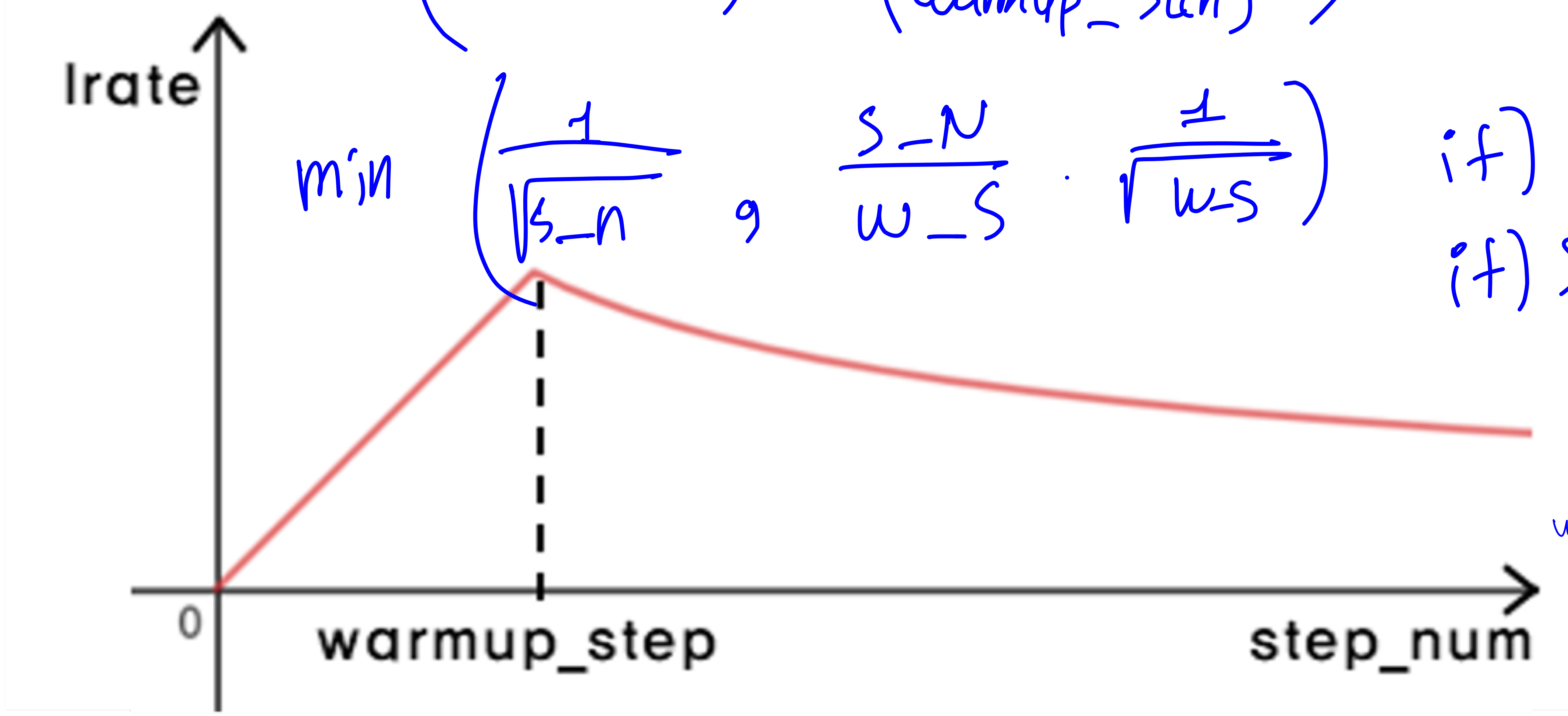
$where \;\; d_{model} = 8$



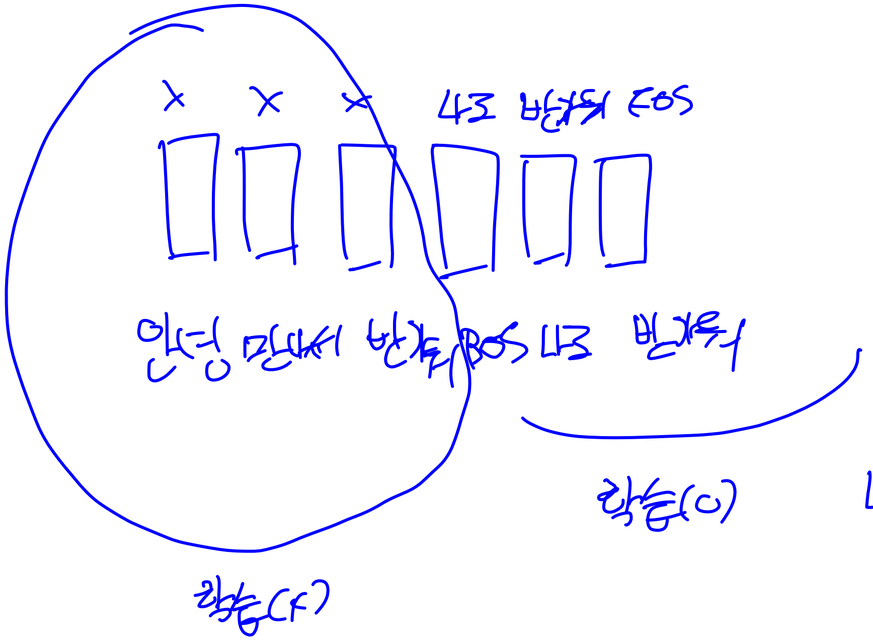
Transformer Tutorial (Optimizer)

- Use Adam Optimizer with $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$
- Varied learning rate over course of training

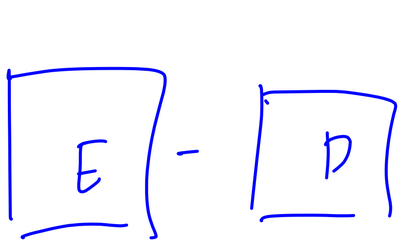
$$lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

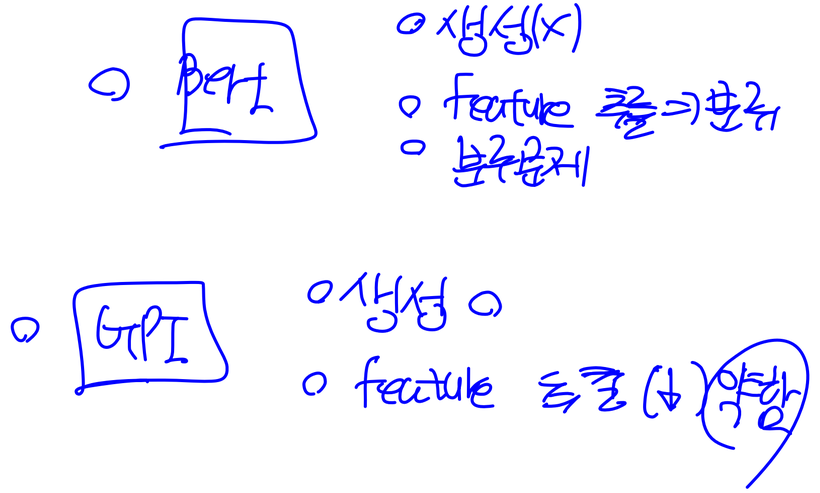


참고: https://hiddenbeginner.github.io/deeplearning/2019/09/22/optimization_algorithms_in_deep_learning.html



감사합니다.





weigh shared Embedding ← Linear input en input de

Position Embedding

E-Lay x 6

De-Lay x 6