Transformer

Attention Is All You Need

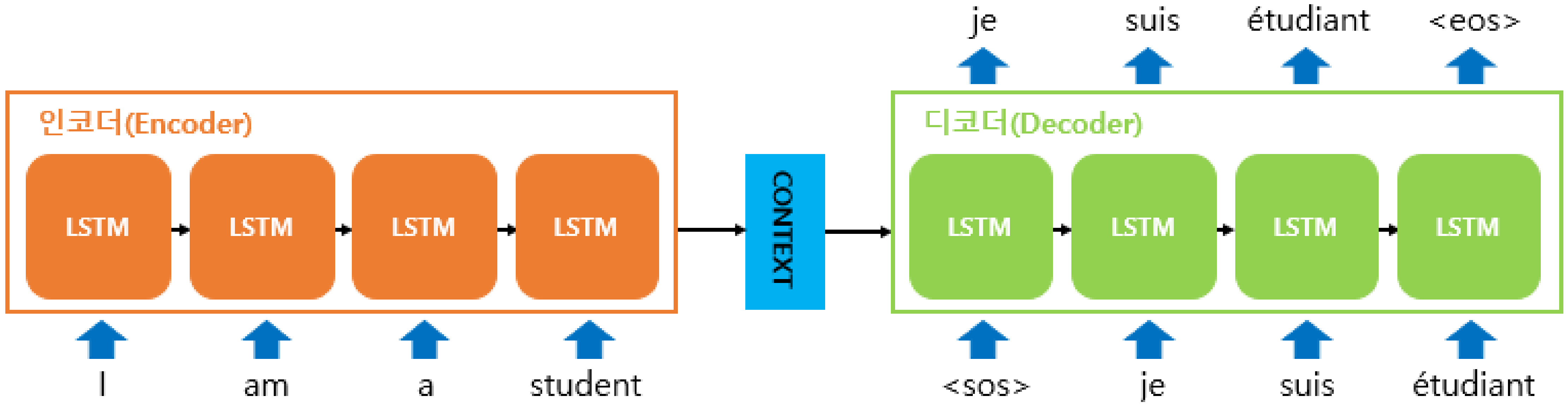
ICLAB 김종현

첫째,

Seq2Seq

Sequence-to-Sequence, Seq2Seq

- 하나의 문백 벡터(context vector)에 Encoder의 정보를 압축
  - Context Vector은 Encoder의 최종 / Decoder의 첫 번째 Hidden State



문제점

1. 하나의 고정된 크기의 Vector에 모든 정보를 압축하려고 하니깐 정보 손실 발생
2. RNN의 고질적 문제인 기울기 소실(Gradient Vanishing) 문제 존재

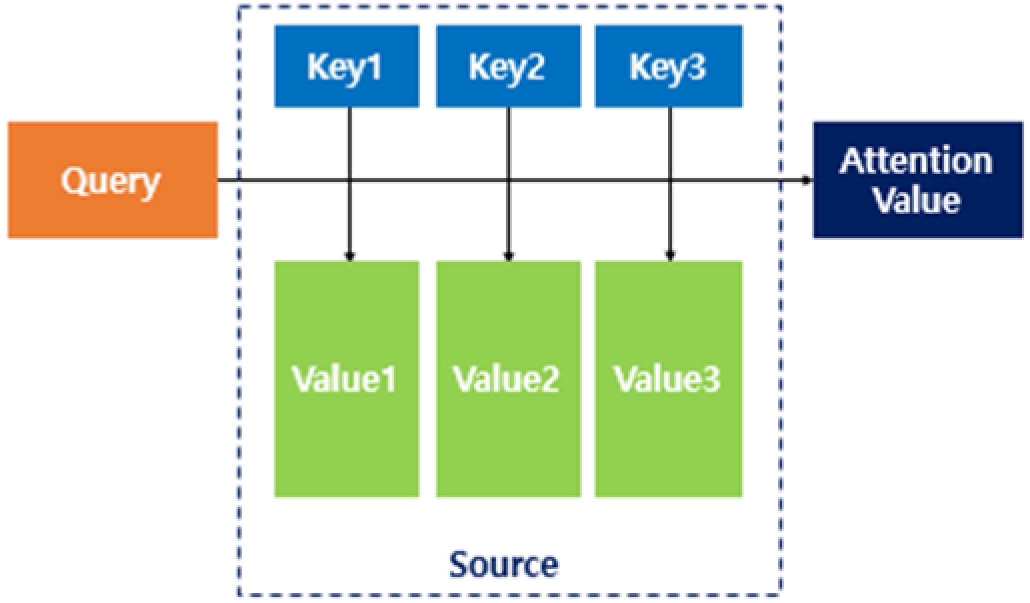
→ 문장의 길이가 길면 길수록 성능 저하

둘째,

Attention

Attention Mechanism

- seq2seq 모델의 단점 보완
- Decoder에서 매 time-step 마다 Encoder의 전체 입력 문장을 참고
  - 동일한 비율로 참고하는 것이 아니라, 해당 시점과 연관된 단어를 집중(attention)해서 참고



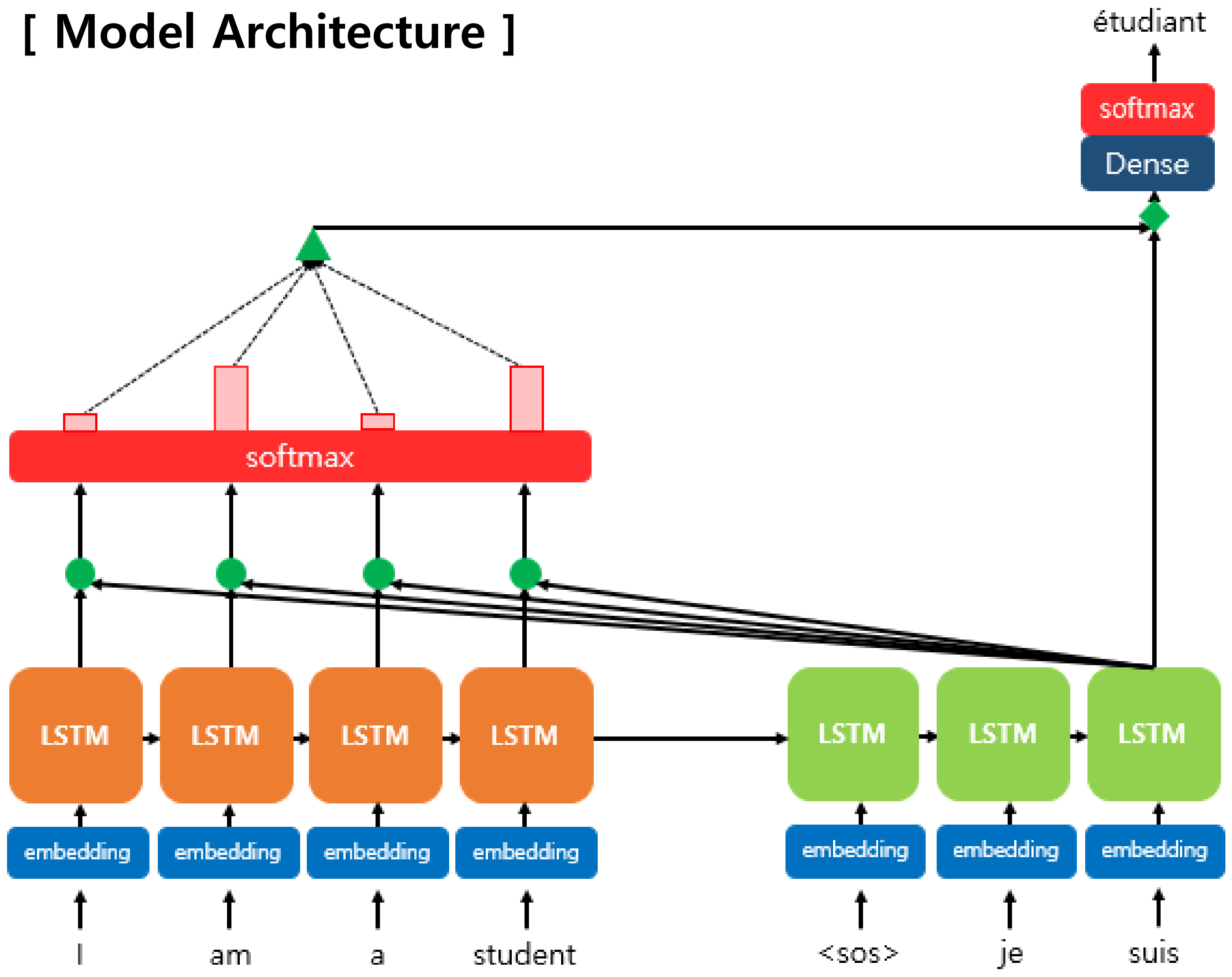
Attention Function

Attention(Q, K, V) = Attention Value

Q(Query) = t 시점의 Decoder 셀에서의 Hidden State
K(Key) = 모든 시점의 Encoder 셀의 Hidden States
V(Values) = 모든 시점의 Encoder 셀의 Hidden States

Dot-Product Attention

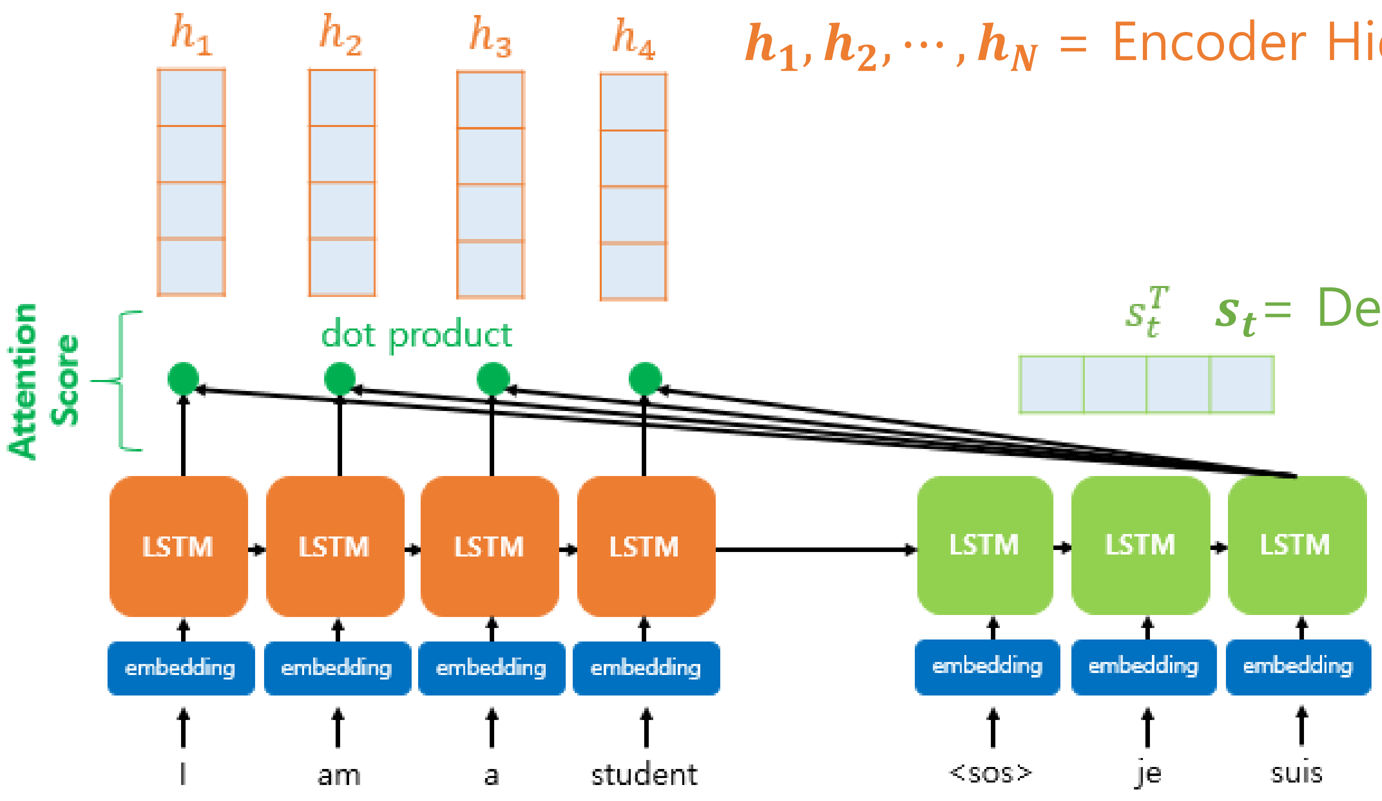
[Model Architecture]



Dot-Product Attention

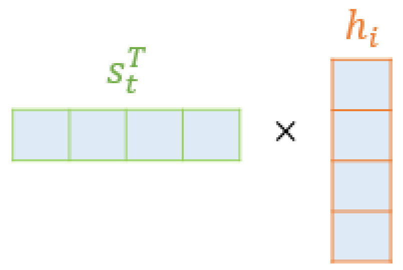
[1. Attention Score 계산 (Dot Product)]

- 현재 Decoder의 Hidden State가 Encoder의 모든 Hidden States와 얼마나 유사 한지 판단



$\boldsymbol{h_1, h_2, \cdots, h_N}$ = Encoder Hidden States (Every Time-steps)

$\boldsymbol{s_t}$= Decoder Hidden State (Present Time-step)



Attention Score

$$score(s_t, h_i) = s_t^T h_i$$

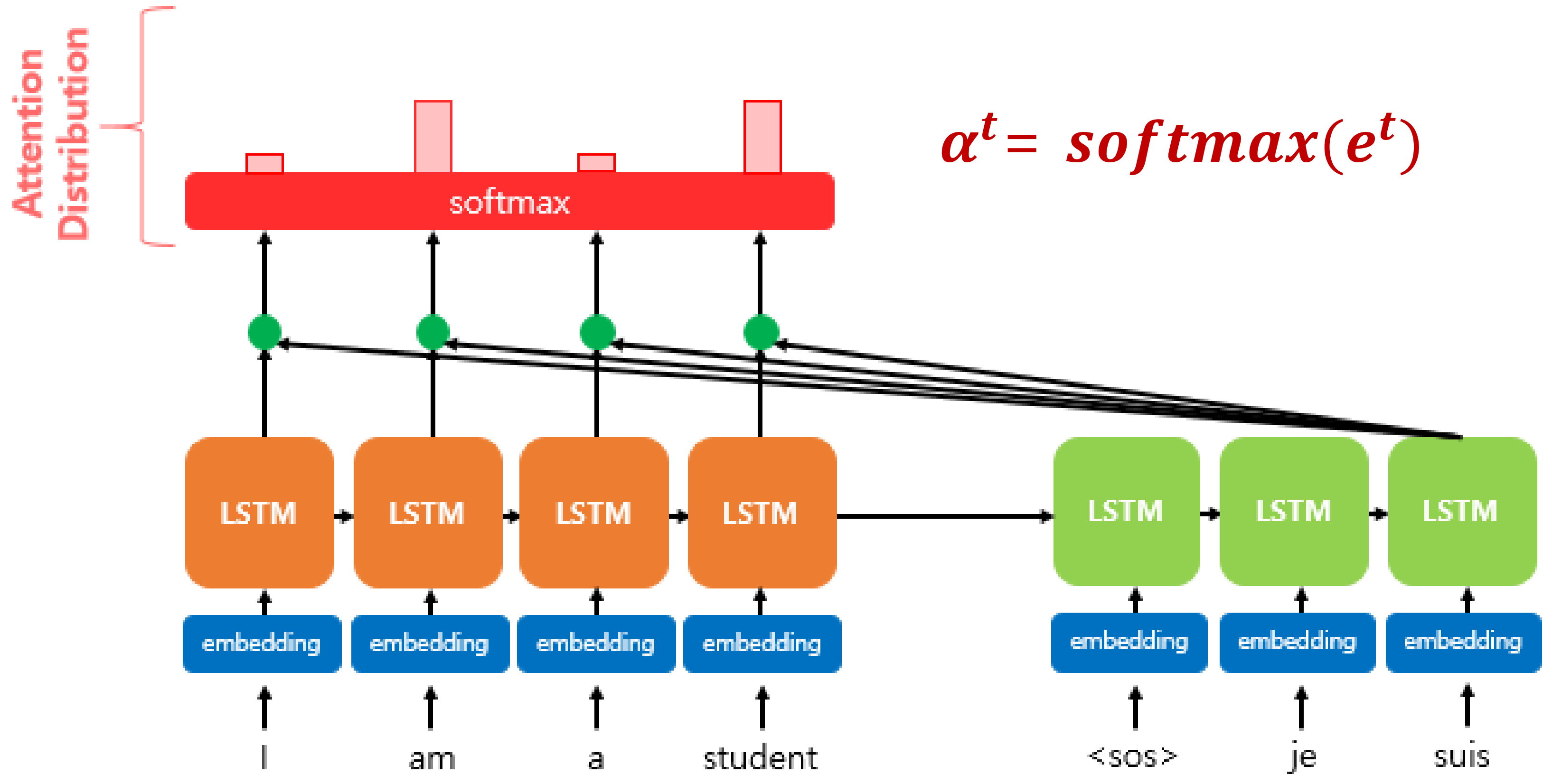
$$e^t = [s_t^T h_1, \cdots, s_t^T h_N]$$

$\boldsymbol{s_t}$를 transpose 해준 뒤 $\boldsymbol{h_t}$와 Dot Product

Dot-Product Attention

[2. Attention Distribution 계산 (Softmax)]

- Attention Score에 Softmax 함수를 적용하여 확률분포를 구함
- 각각의 값은 총합이 1인 Attention Weight

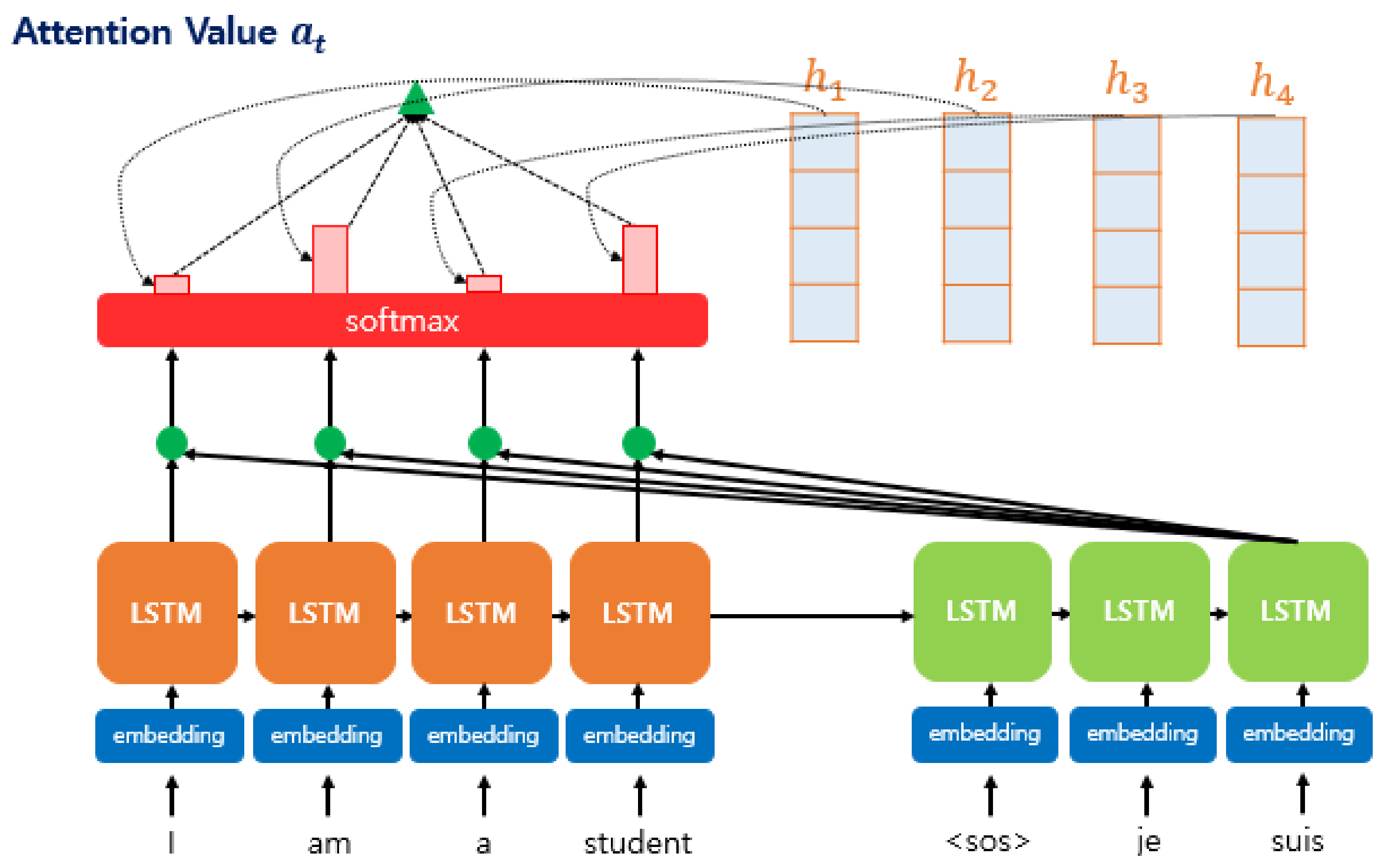


$$\alpha^t = softmax(e^t)$$

Dot-Product Attention

[3. Attention Value 계산 (Weighted Sum)]

- 각 Encoder의 Attention Weight와 Hidden State를 곱하고, 최종적으로 모두 더함

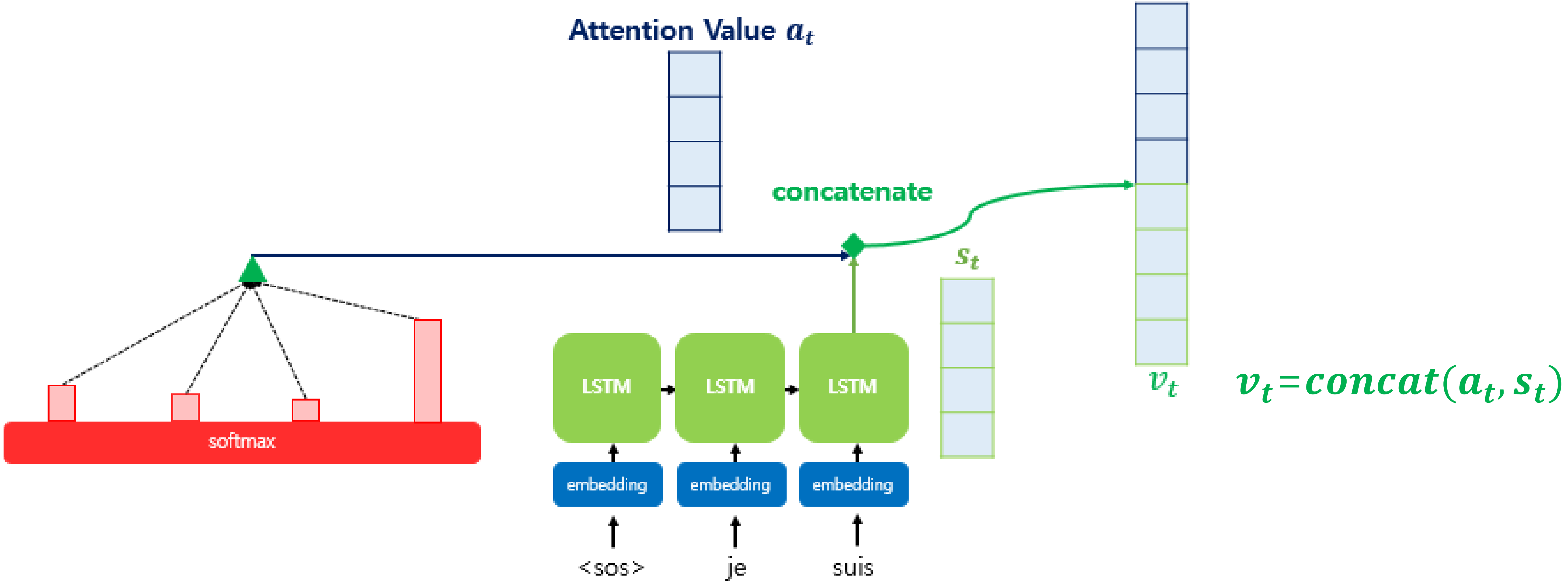


$$a^t = \sum_{i=1}^{N} \alpha_i^t h_i$$

Dot-Product Attention

[4. Concatenate]

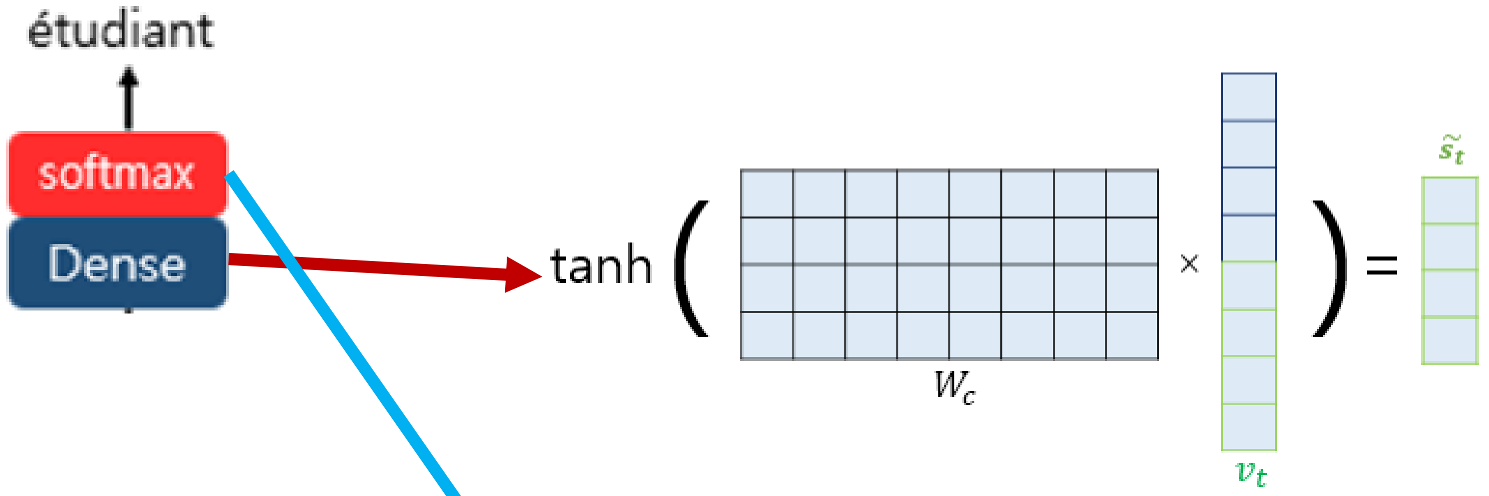
- Attention Value와 Decoder의 현재 시점(t)의 Hidden State를 연결해 하나의 Vector로 만듦



Dot-Product Attention

[5. Output]

- Dense층: 가중치 행렬과 Concatenate 결과를 곱한 후에 하이퍼볼릭탄젠트(Tanh) 함수를 통과
- Softmax층: Dense층의 출력을 입력으로 사용하여 예측 벡터를 얻음



$$\tilde{s}_t = tanh(W_c[a_t; s_t] + b_c)$$

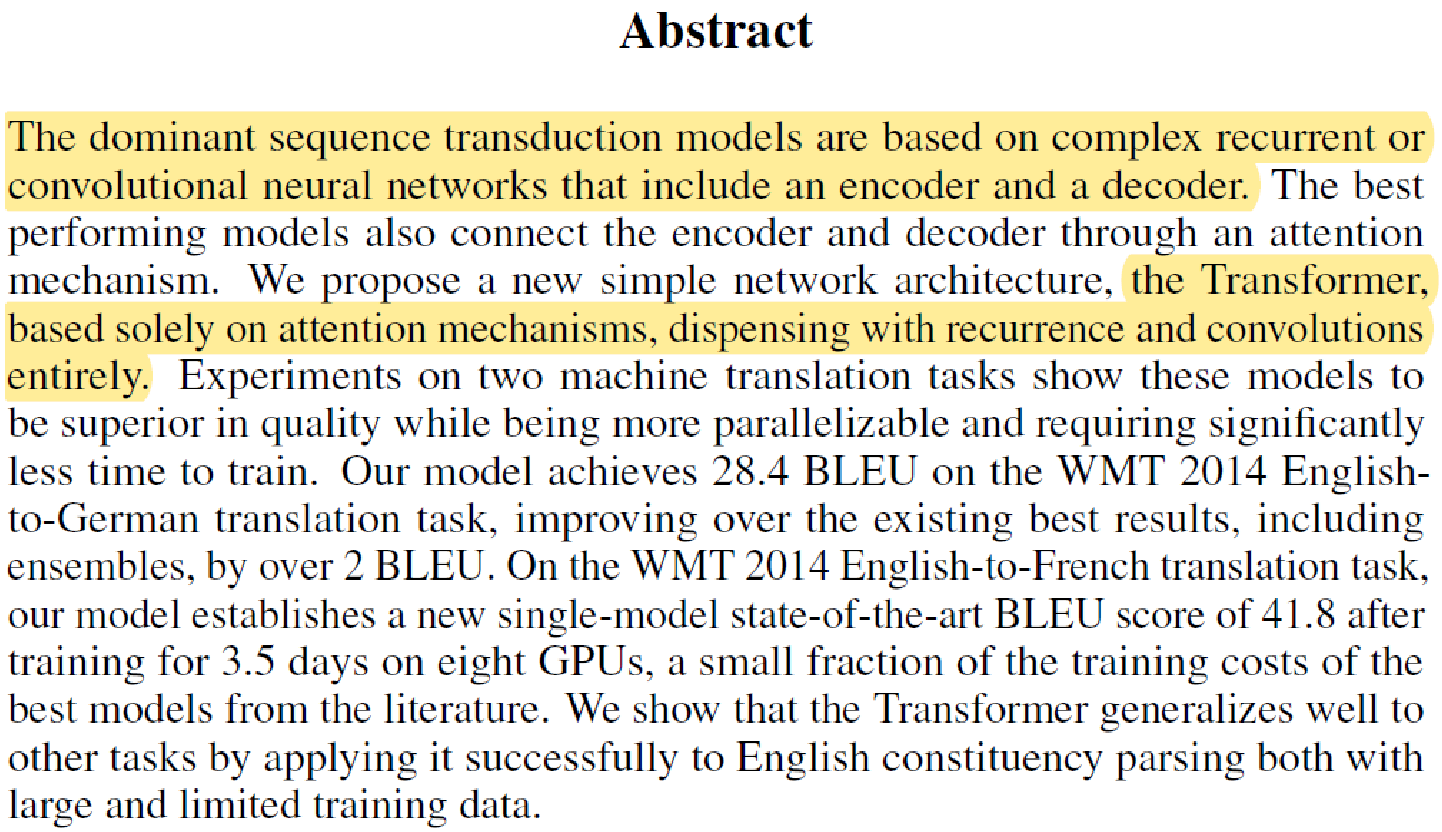
$$\hat{y}_t = Softmax(W_y\tilde{s}_t + b_y)$$

마지막,

Transformer

Transformer (Attention Is All you Need)

[Abstract]



Abstract

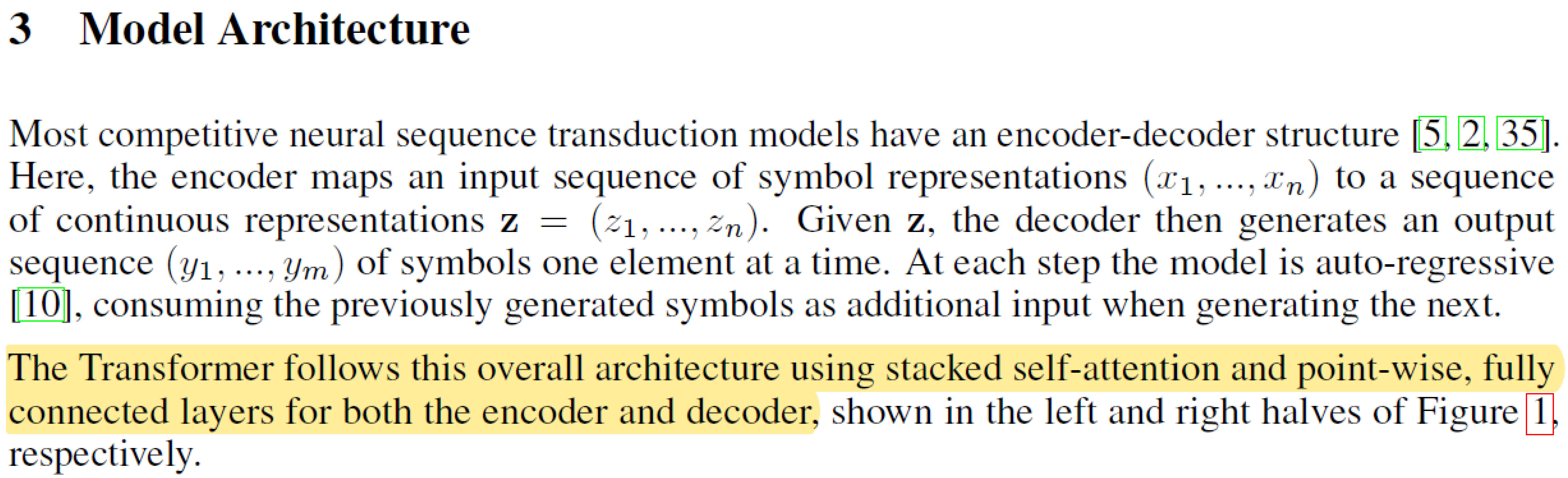
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, 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.

- 기존의 sequence 모델은 Encoder와 Decoder을 포함한 RNN 또는 CNN 기반
- Transformer은 RNN이나 CNN을 사용하지 않고 오직 Attention Mechanism으로만 설계

Transformer (Attention Is All you Need)

[Model Architecture]

- Transformer은 중첩된 Self-attention, Positional-Wise, Fully-Connected Layer를 인코더와 디코더에서 사용



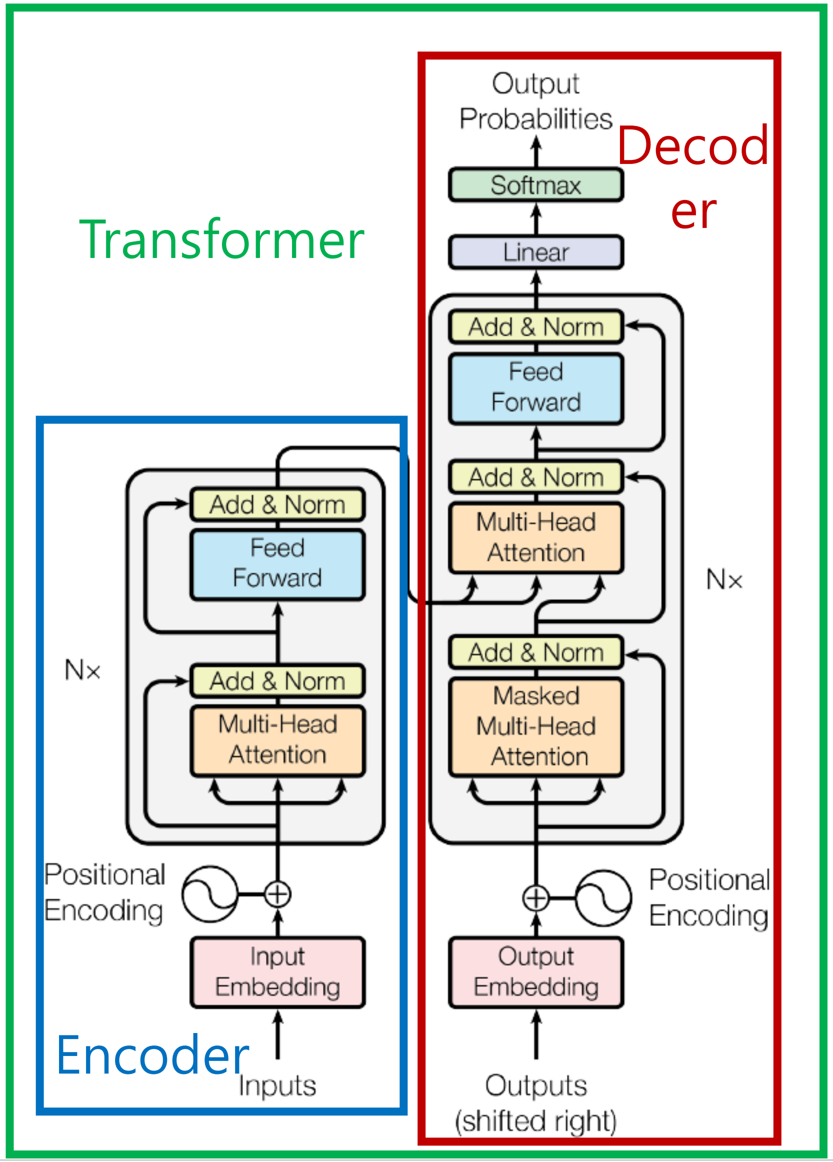
3 Model Architecture

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.

Transformer (Attention Is All you Need)

[Model Architecture]



Transformer (Attention Is All you Need)

[Hyper Parameters]

1. d_{model}: Embedding Dimension (= 512) $d_{\text{model}} = 512$
2. num_layers: Encoder Layer 중첩 개수 (=6) $N = 6$
3. num_heads: 병렬적으로 Attention을 수행할 횟수 (=8) $h = 8$
4. d_k: $\mathrm{d}_{\mathrm{model}}$ / num_heads. 한 Attention의 Query, Key, Value dim(=64) $d_k = d_v = d_{\text{model}}/h = 64$
5. dff: FFNN Layer의 dim(=2048) $d_{ff} = 2048$
6. P_{drop} = 0.1: Dropout Rate $P_{drop} = 0.1$

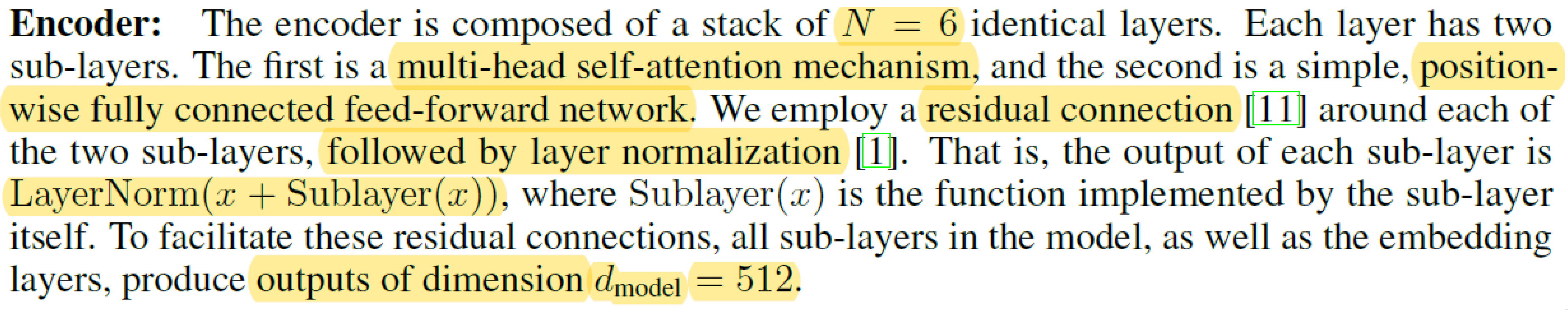
Transformer (Attention Is All you Need)

[Encoder]

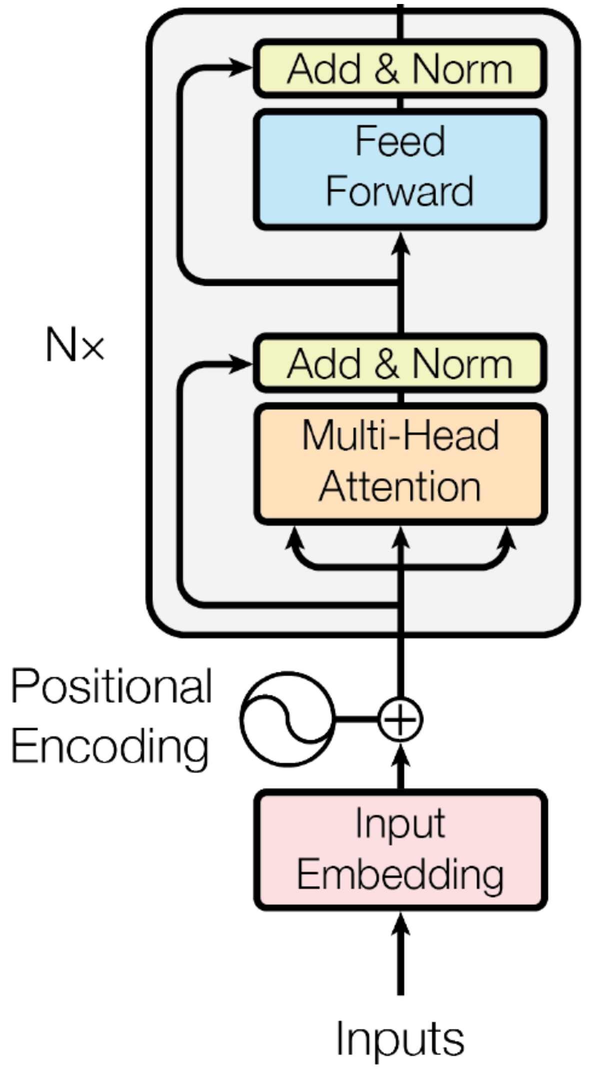
- 6개의 Layer Stack
- 2개의 Sub-Layers

  1. Multi-Head Self-Attention
  2. Position-Wise Fully Connected Feed-Forward

- 각 Sub Layer은 Residual Connection 후 Layer Normalization
  → overfitting 방지
- 최종 Output Dimension은 Input Embedding Dim과 동일 (d_{model} = 512)



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 $d_{\text{model}} = 512$.

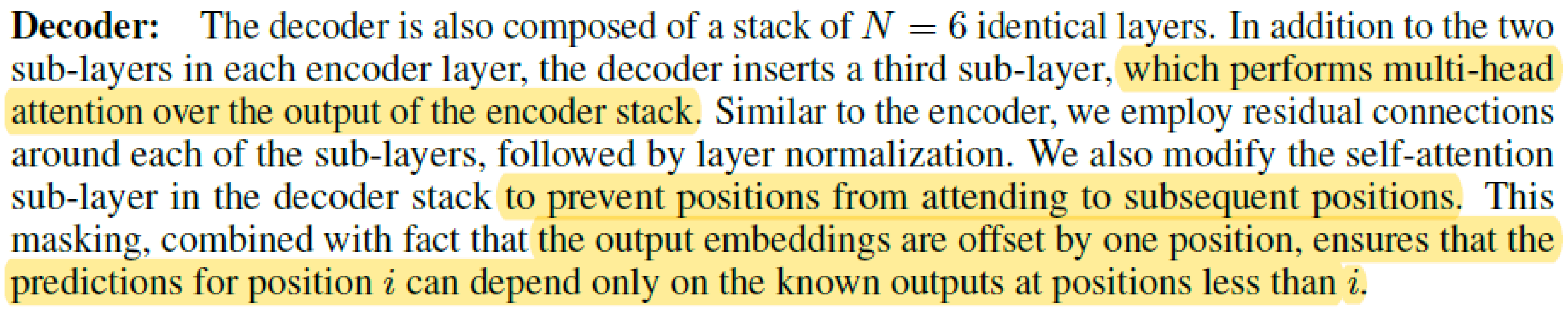


Transformer (Attention Is All you Need)

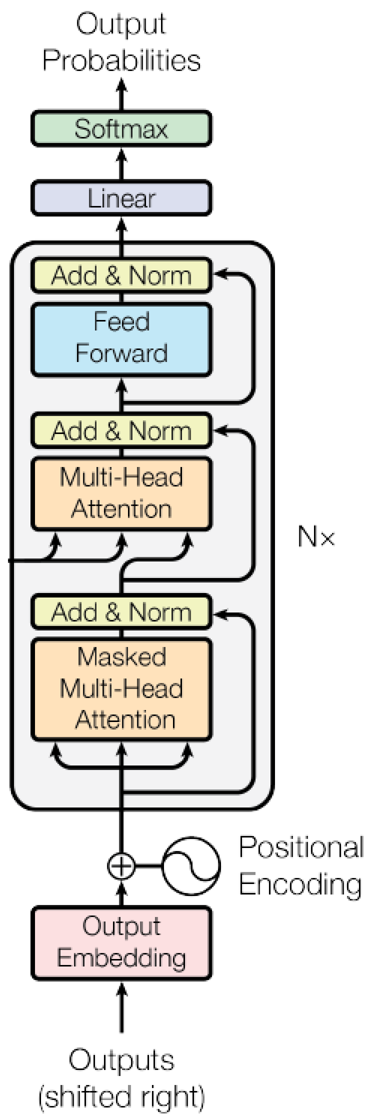
[Decoder]

- Encoder에서 Sub-Layer의 개수와 그 기능만 다르고 나머지는 동일하다.
- 3개의 Sub-Layers

  1. Masked Multi-Head Self-Attention
     → 이전에 등장한 단어들만 참고할 수 있는 형태로 Mask를 씌움
  2. Multi-Head Self-Attention
  3. Position-Wise Fully Connected Feed-Forward



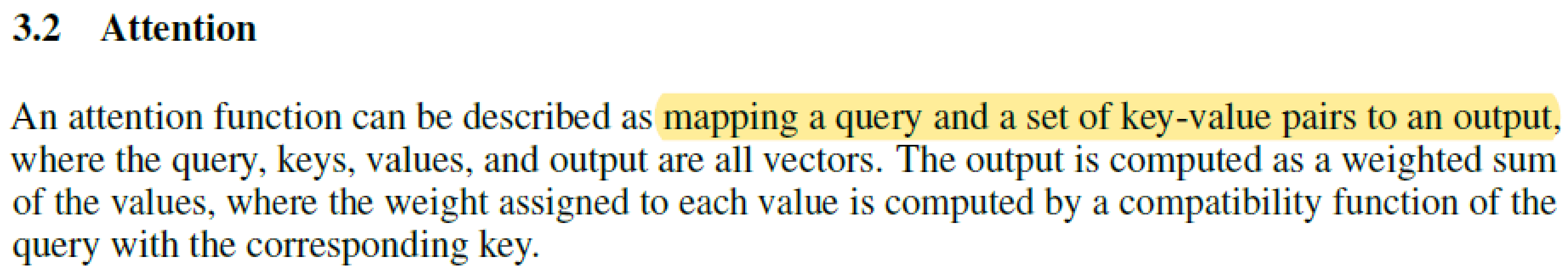
Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections 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.



Transformer (Attention Is All you Need)

[Attention]

- Query(질문)와 Key(Attention을 수행할 대상)를 Mapping해 Output으로 전달
- Query, Key, Value, Output은 모두 Vector
- Transformer에서는 병렬적으로 Attention 수행 → 여러 개의 Attention Value를 참조하며 성능 향상



3.2 Attention

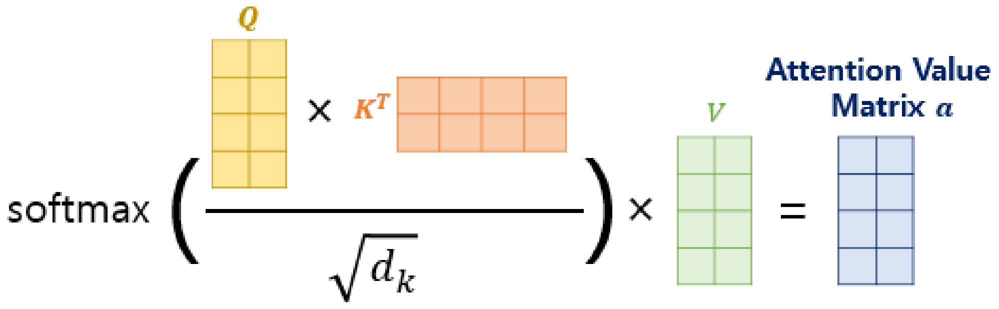
An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the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

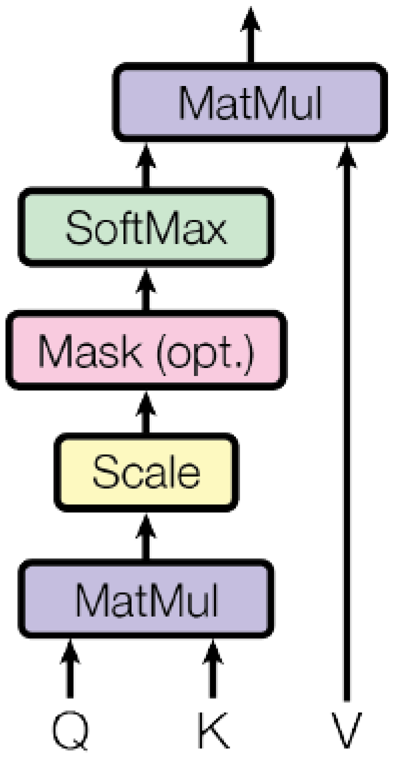
Transformer (Attention Is All you Need)

[Scaled Dot-Product Attention]

1. Query 행렬과 Key 행렬의 Transpose를 MatMul → 특정 단어(Query)가 어떤 단어(Key)와 관련되어 있는지?
2. Scaling (Key Dimension의 Root값 논문에서는 d_k)
3. Masking(Option, Decoder에서 수행)
4. Softmax 함수 통과 (Attention Score를 구함)
5. Attention Score, Value 행렬을 MatMul하여 Attention Value값 계산

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

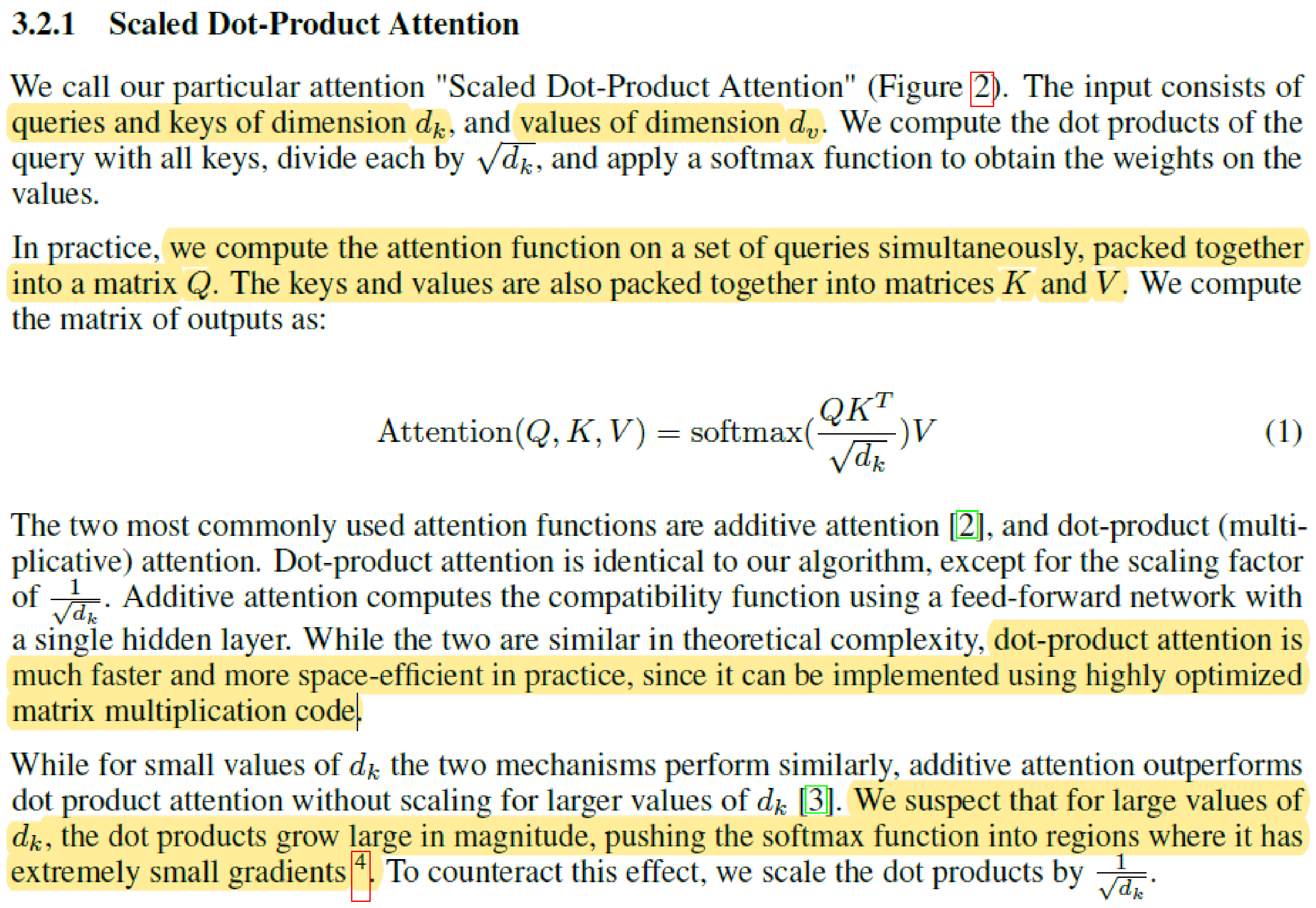




Scaling을 하는 이유?
d_k(Key dim)가 커질수록 내적의 값이 커지므로 Softmax 통과시 기울기 편중 발생
→ 이를 방지하기 위해 Scaling

Transformer (Attention Is All you Need)

[Scaled Dot-Product Attention]



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:

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

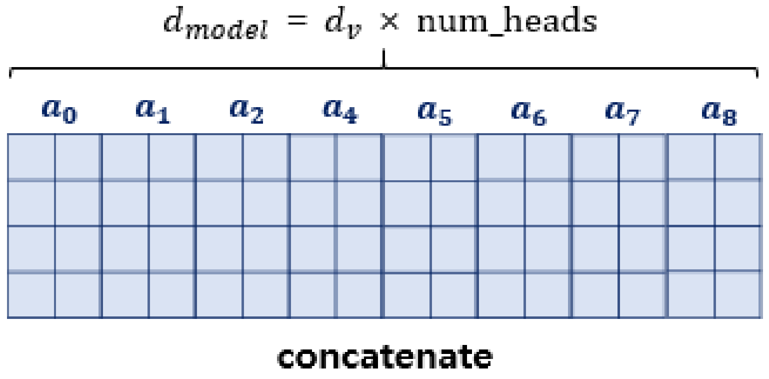
The two most commonly used attention functions are additive attention [2], and dot-product (multi-plicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward 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.

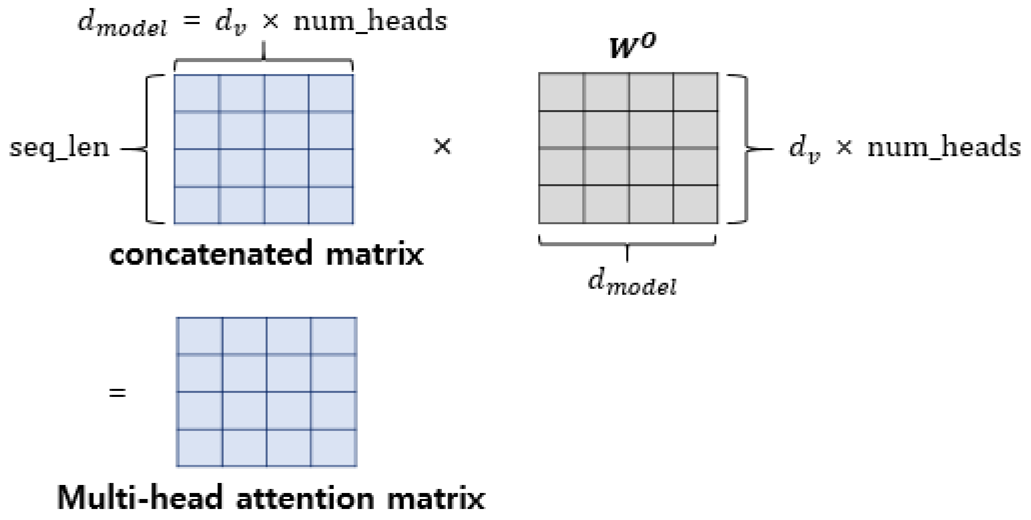
While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it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

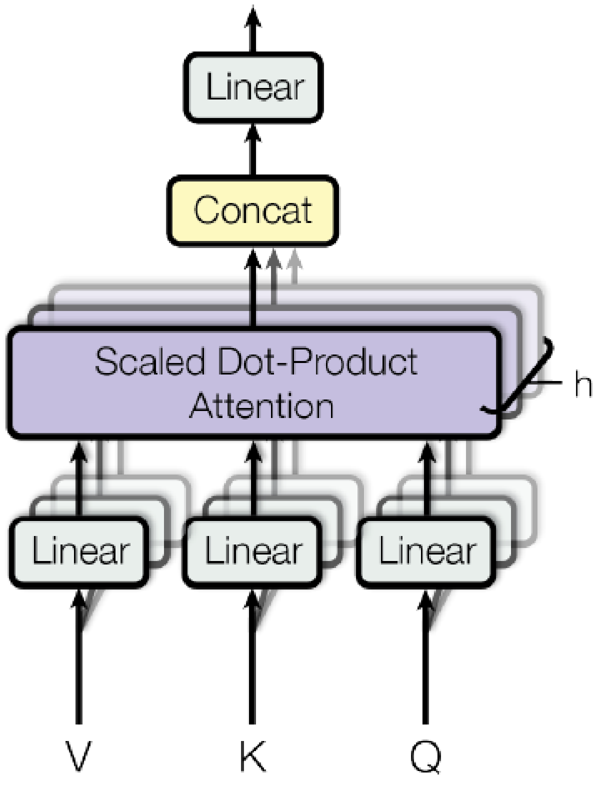
Transformer (Attention Is All you Need)

[Multi-Head Attention]

1. Query, Key, Value를 num_heads(8) 만큼 나누어서 Linear Projection → 다른 시각으로 학습
   - 서로 다른 8개의 Parallel Attention 수행
   - 가중치 행렬 W^Q, W^K, W^v의 값은 8개의 Attention Head 마다 다름
2. 모든 Attention Head를 Concat
3. Linear(W^O)층 통과


$d_{model} = d_v \times$ num_heads
a_0 a_1 a_2 a_4 a_5 a_6 a_7 a_8
concatenate





Transformer (Attention Is All you Need)

[Multi-Head Attention]

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.

$$\begin{aligned}\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O \\ \text{where}\ \mathrm{head_i} &= \mathrm{Attention}(QW^Q_i, KW^K_i, VW^V_i)\end{aligned}$$

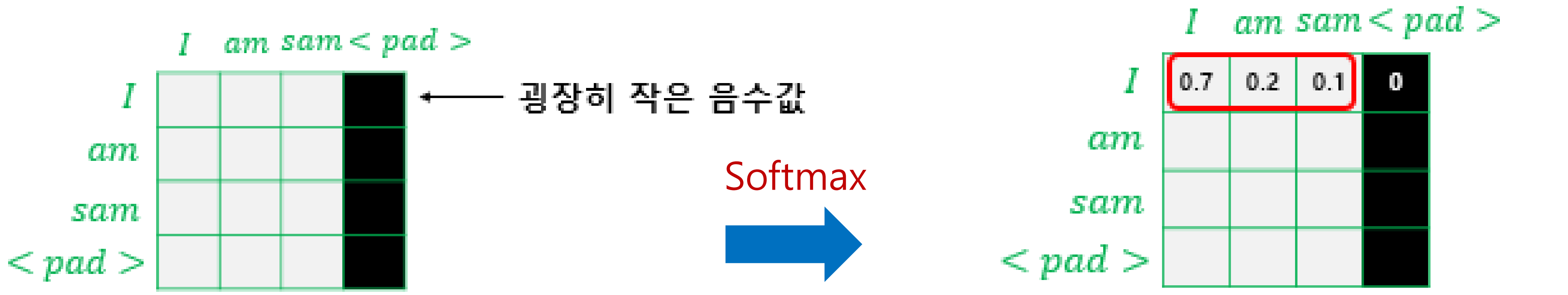
Where the projections are parameter matrices $W^Q_i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W^K_i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W^V_i \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

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.

Transformer (Attention Is All you Need)

[Applications of Attention in Transformer]

- Encoder-Decoder Attention은 Decoder Layer에서 사용
  - Query값은 Decoder Layer에서, Key와 Value 값은 Encoder Layer의 Output
  - 출력 단어를 만들기 위해 source 문장의 단어들 중에서 어떤 단어에 초점을 맞출지 계산
- Self-Attention은 기본적으로 Query, Key, Value가 동일. Encoder Layer에 해당
- Decoder Layer에서 Self-Attention은 inf 값으로 마스크를 씌운다. → Softmax 통과시 inf는 0
  - 앞 부분에 대한 단어의 정보만 참고하도록 하기 위해

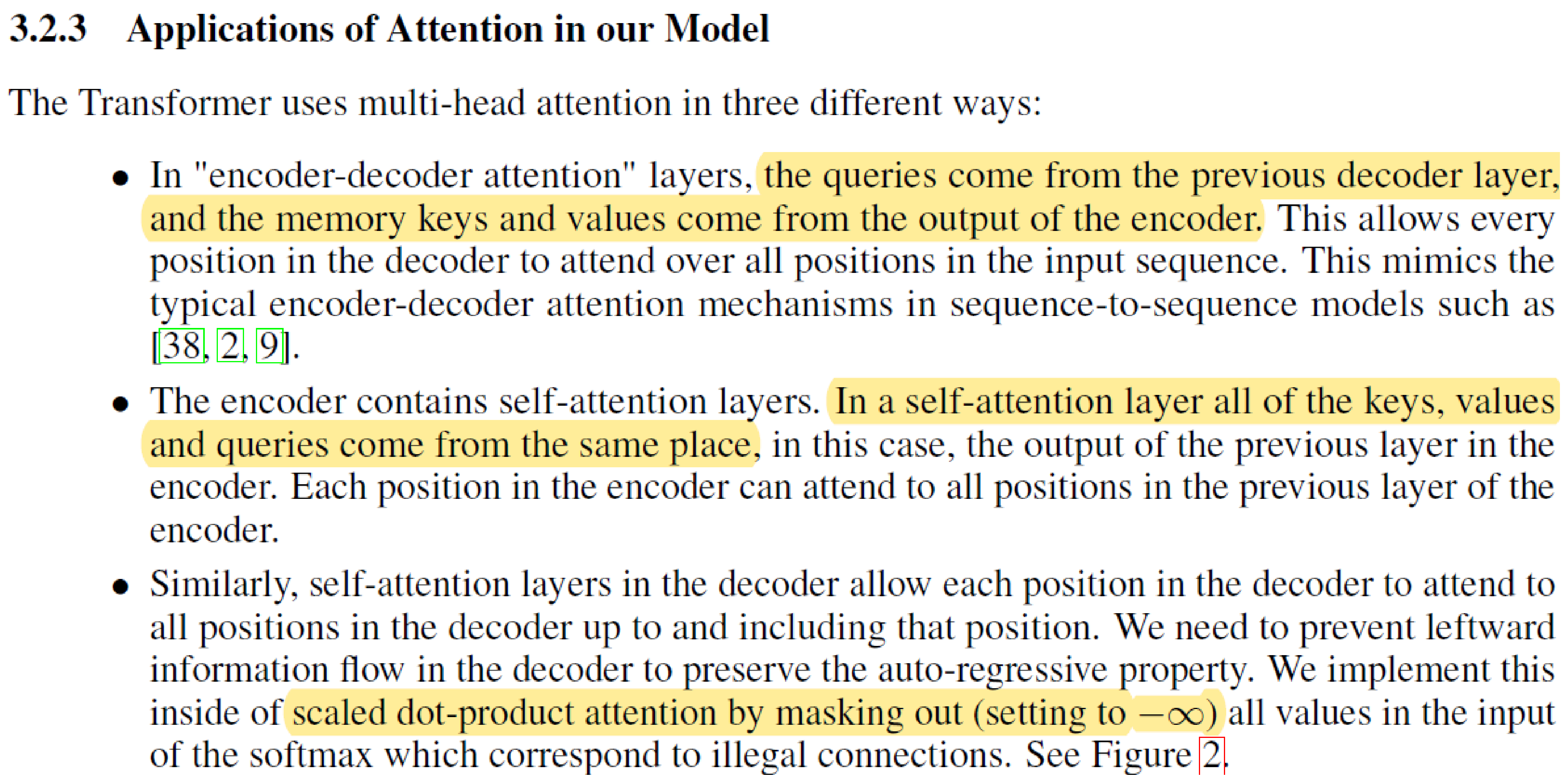


Attention Score Matrix

Attention Score Matrix

Transformer (Attention Is All you Need)

[Applications of Attention in Transformer]



3.2.3 Applications of Attention in our Model

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

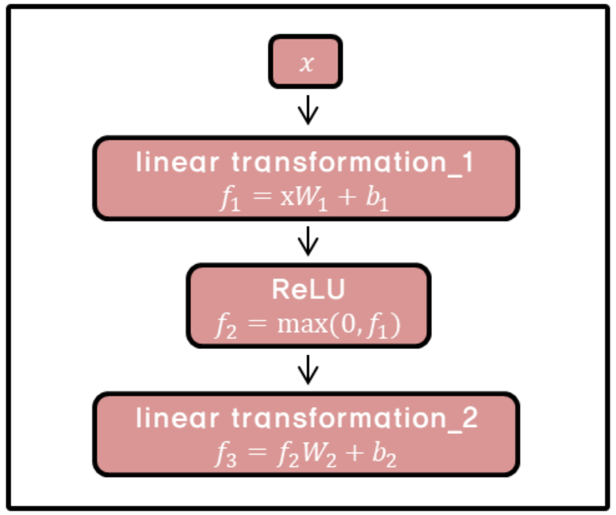
- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention 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.

Transformer (Attention Is All you Need)

[Position-Wise FeedForward Networks, FFNN]

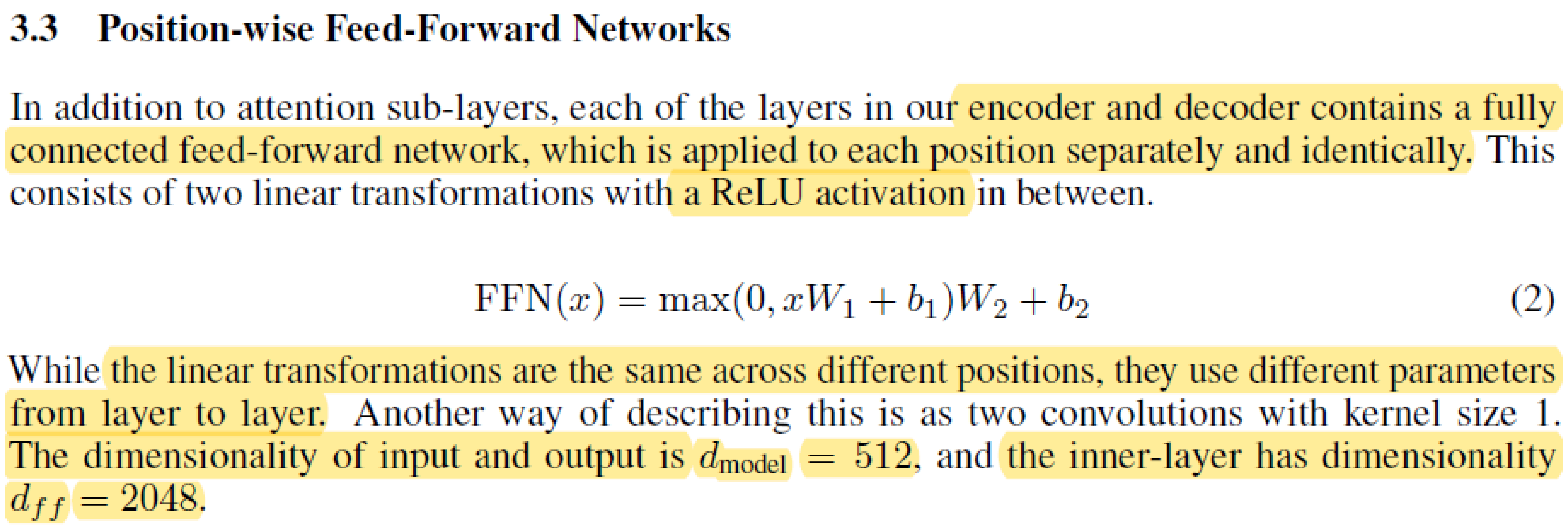
- Attention의 모든 Sub-Layer에는 FFNN이 존재
  - Attention Value들은 2개의 Linear층을 통과
  - Layer(num_layers)마다 서로 다른 파라미터를 사용.
    - 단, 하나의 Encoder Layer 내에서는 다른 문장, 다른 단어들마다 동일하게 사용됨

- 최종적으로 FFNN을 통과함으로써 Embedding dim과 Attention dim이 동일해짐
- 활성화 함수로는 ReLU 함수 사용



Transformer (Attention Is All you Need)

[Position-Wise FeedForward Networks, FFNN]



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.

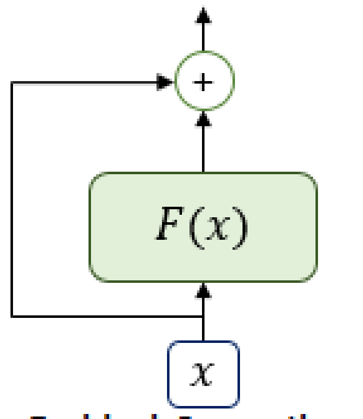
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

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 $d_{ff} = 2048$.

Transformer (Attention Is All you Need)

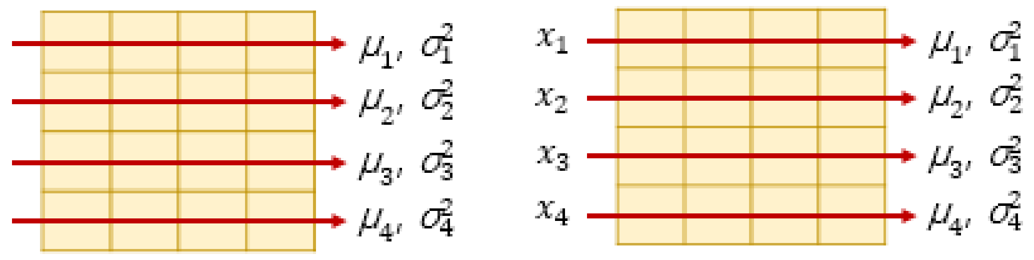
[Add & Norm (Residual Connection and Layer Normalization)]

$$H(x) = x + F(x)$$



Residual Connection

- Residual Connection
  - 서브층의 입력과 출력을 더하는 것
  - 모든 층을 지날 때마다 잔차(Residual)를 균일하게 유지
  - Gradient가 출력층의 비용함수에서 입력층까지 쉽게 흐를 수 있게 함

- Layer Normalization
  - Tensor의 마지막 차원(d_{model})에 대해서 평균과 분산을 구한 후 정규화
  - 

Residual Connection output   Residual Connection output

(seq_len, d_model)

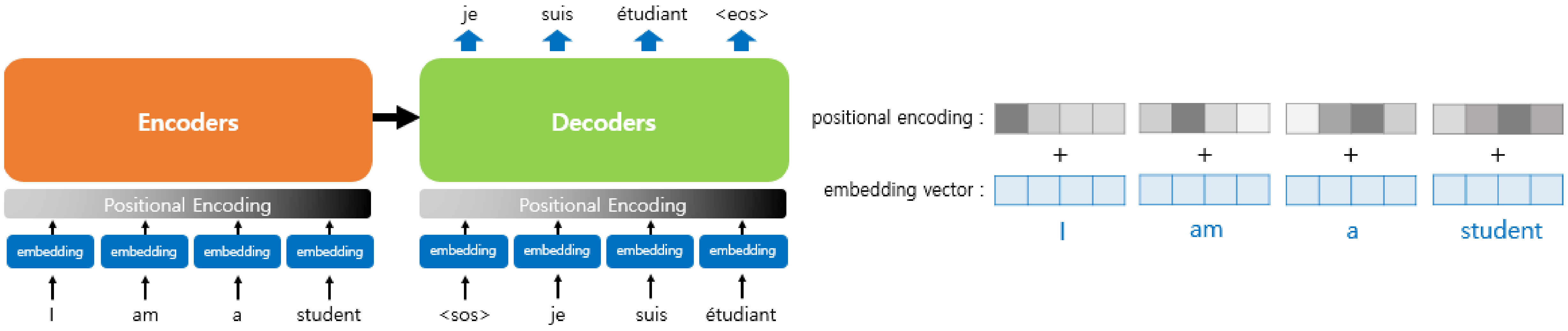
  - x_i: $Vector$, k: x_i의 각 차원, μ: 평균, σ: 분산, ϵ:분모가 0이 되는 것을 방지

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Transformer (Attention Is All you Need)

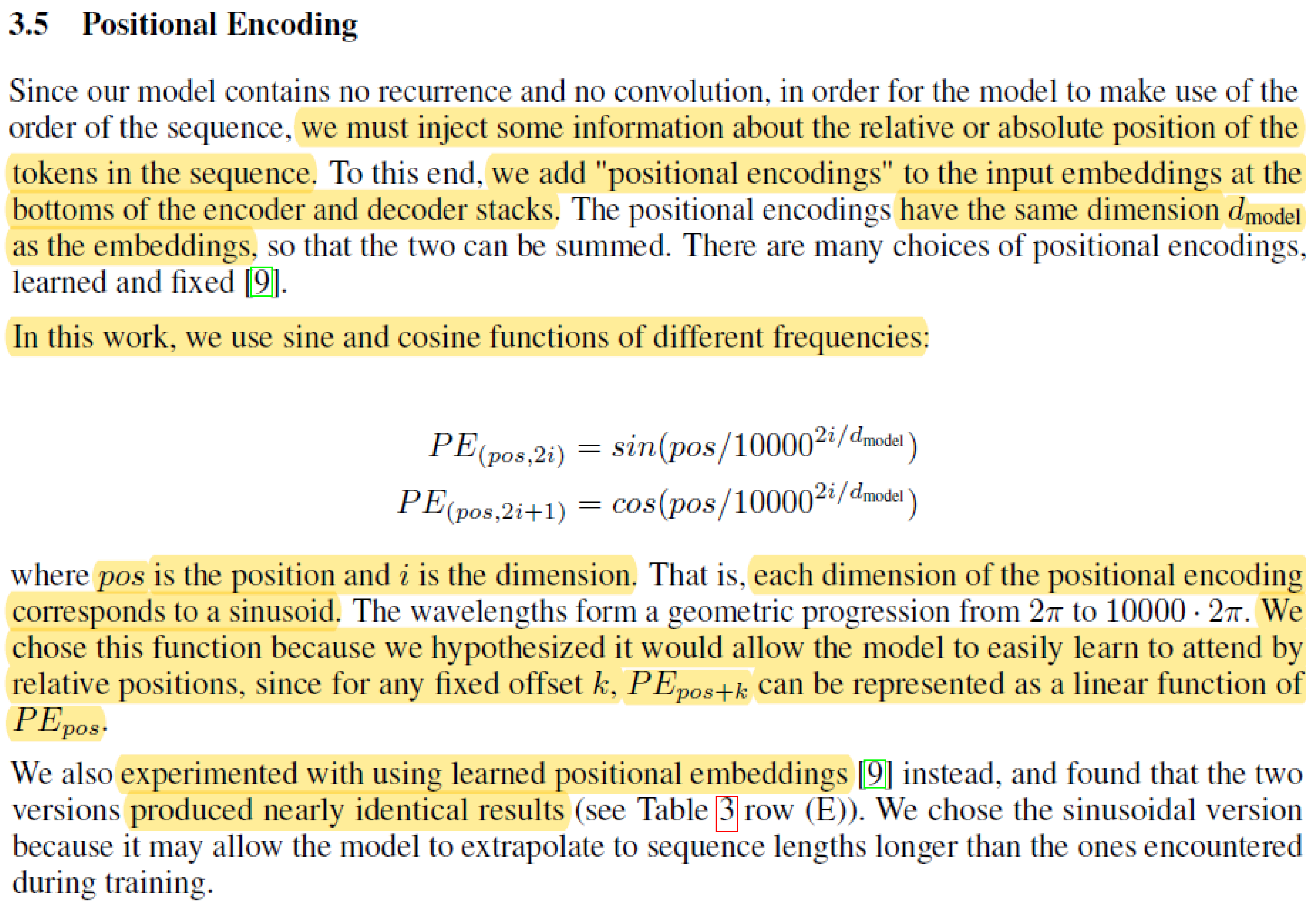
[Positional Encoding]

- RNN이나 CNN을 사용하지 않기 때문에, 위치에 대한 정보 값 필요
- 주기함수(sin, cos)를 통해 위치정보를 알려줌
  - 보다 긴 sequence의 문장이 입력 되었을 때 성능이 잘 나옴
- Positional Enconding은 사전에 정의해도 되고, Train도 가능
  - 논문에서는 두 가지 방법 모두 사용해 보았지만 성능 차이는 크게 나지 않음
- Embedding Vector 각 내 차원의 index가 짝수이면 sin함수를, 홀수이면 cos함수를 이용



Transformer (Attention Is All you Need)

[Positional Encoding]



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.

Transformer (Attention Is All you Need)

[Why Self-Attention ?]

1. 각 Layer마다 Computational Complexity 감소

2. 병렬 연산 가능

3. Sequence가 길어져도 기울기 소실이 일어나지 않음(RNN 대신 Attention만 사용하기 때문)

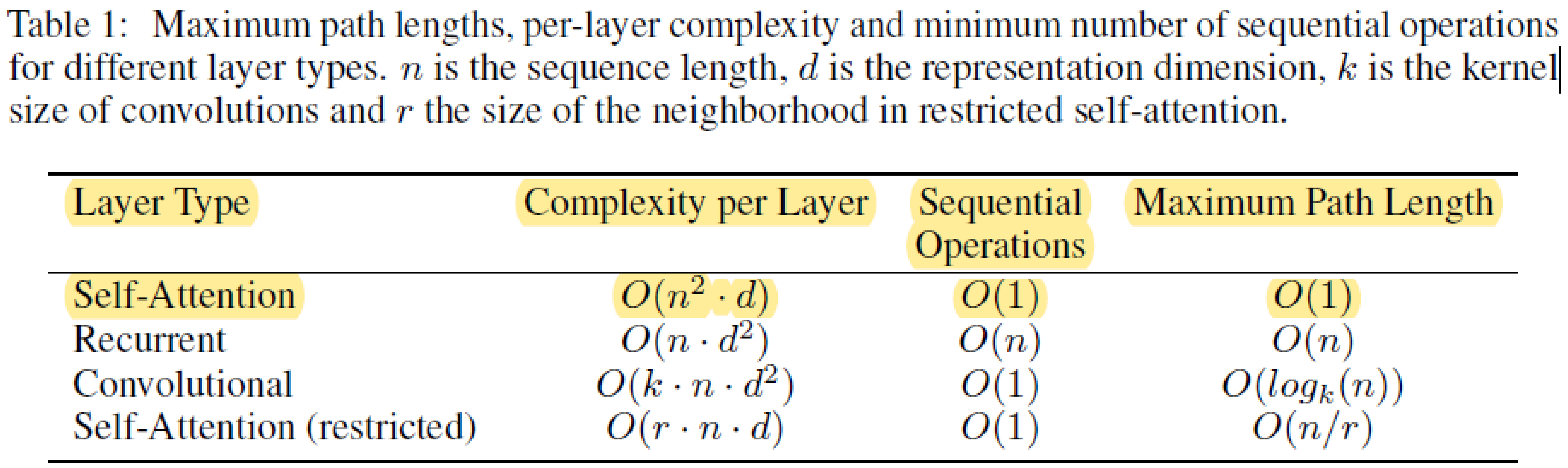


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

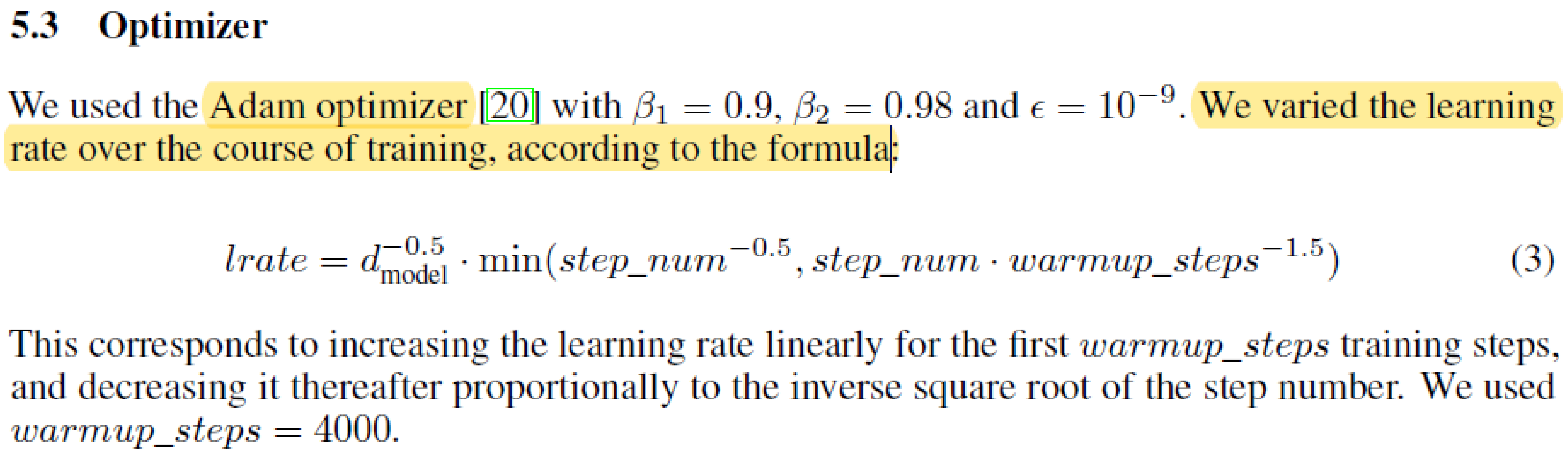
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

RNN, CNN, Self-Attention 성능 비교

Transformer (Attention Is All you Need)

[Optimizer]

- Adam Optimizer 사용
  - Learning Rate도 Train



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:

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

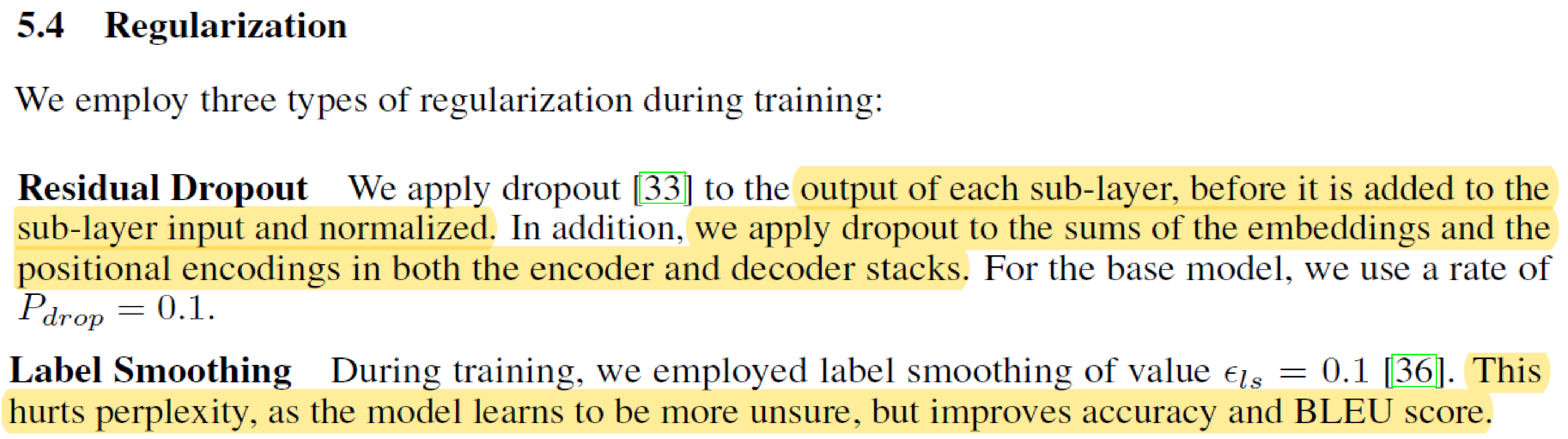
This corresponds to increasing the learning rate linearly for the first *warmup_steps* training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$.

Transformer (Attention Is All you Need)

[Regularization]

- Residual Dropout
  - 정규화 전의 Sub-Layer Input, 임베딩의 합(Attention Score), Positional Encoding에 적용
  - 기본 비율 = 0.1

- Label Smoothing
  - 모델이 특정 출력 값에 대해 확신을 가지지 않도록 함
    → 정규화 효과 극대 (Improve accuracy, BLEU Score)



5.4 Regularization

We employ three types of regularization during training:

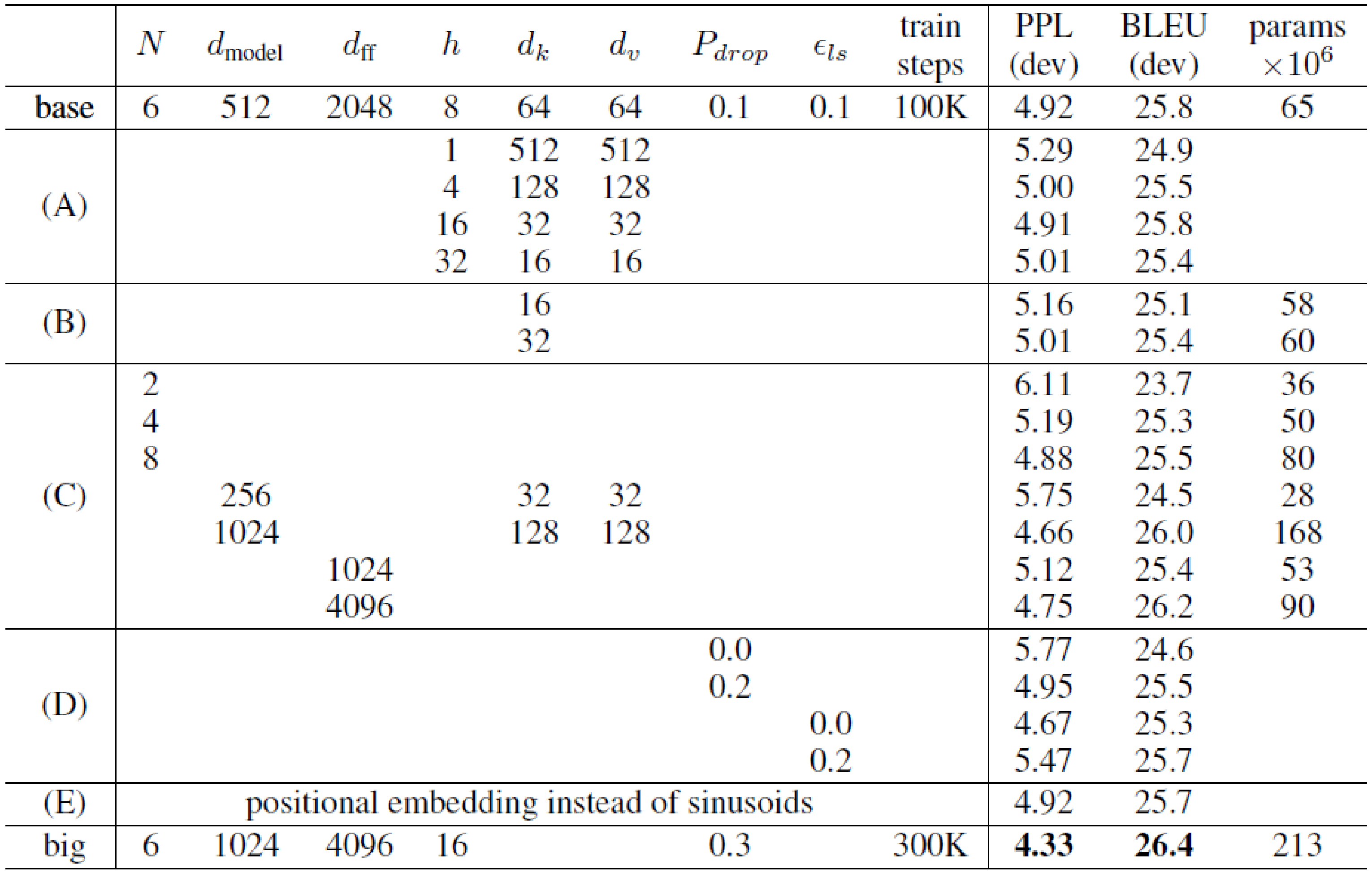
Residual Dropout We apply dropout [33] to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop}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

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.

Transformer (Attention Is All you Need)

[Model Variations]

- Hyper Parameter 값의 변화에 따른 결과 도출

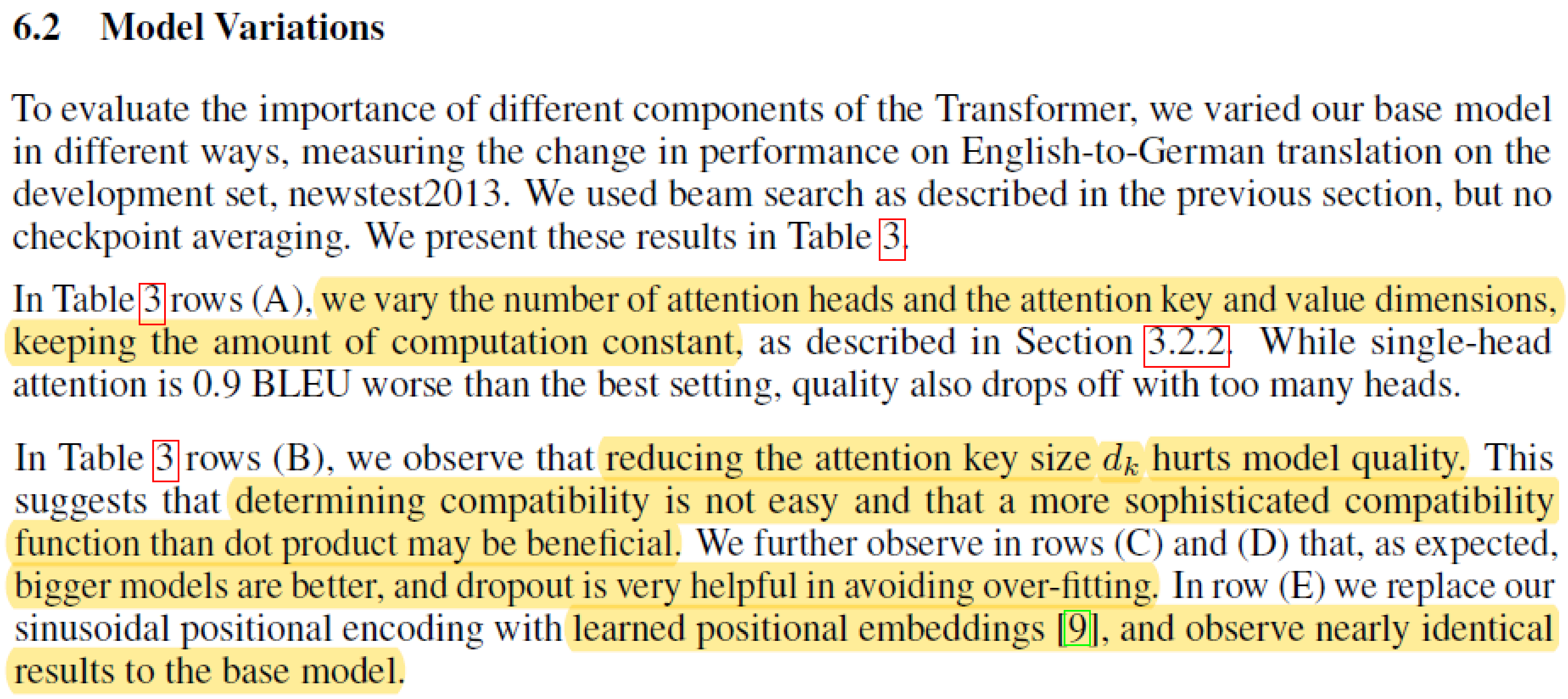


	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					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

Transformer (Attention Is All you Need)

[Model Variations]

- Key Size의 감소는 Capacity 감소로 인해 Model Quality를 낮춤.
- Model의 크기를 키웠을 때 성능 향상
- Dropout 기법은 Overfitting 방지에 효과적



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.