

## Attention Is All You Need

- 저자: Ashish Vaswani, 외 7명 (Google Brain)
- NIPS 2017 accepted

# Who is an Author?



Ashish Vaswani

Senior Research Scientist, [Google Brain](#)  
Verified email at google.com

FOLLOW

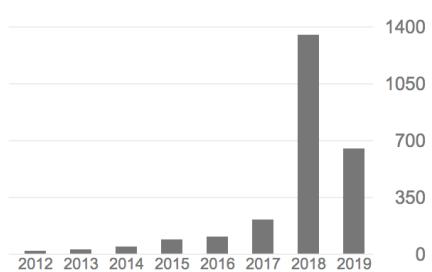
[GET MY OWN PROFILE](#)

Cited by

[VIEW ALL](#)

All Since 2014

Citations	2597	2477
h-index	15	13
i10-index	21	18



TITLE

CITED BY

YEAR

[Attention is all you need](#)

A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, ...  
Advances in neural information processing systems, 5998-6008

1610

2017

[Decoding with large-scale neural language models improves translation](#)

A Vaswani, Y Zhao, V Fossum, D Chiang  
Proceedings of the 2013 Conference on Empirical Methods in Natural Language ...

178

2013

[Relational inductive biases, deep learning, and graph networks](#)

PW Battaglia, JB Hamrick, V Bapst, A Sanchez-Gonzalez, V Zambaldi, ...  
arXiv preprint arXiv:1806.01261

124

2018

[Learning whom to trust with MACE](#)

D Hovy, T Berg-Kirkpatrick, A Vaswani, E Hovy  
Proceedings of the 2013 Conference of the North American Chapter of the ...

101

2013

[One model to learn them all](#)

L Kaiser, AN Gomez, N Shazeer, A Vaswani, N Parmar, L Jones, ...  
arXiv preprint arXiv:1706.05137

97

2017

[Tensor2tensor for neural machine translation](#)

A Vaswani, S Bengio, E Brevdo, F Chollet, AN Gomez, S Gouws, L Jones, ...  
arXiv preprint arXiv:1803.07416

54

2018

## 느낀점

- Multi-Head-Attention을 빠르게 구현하기 위해 Matrix 연산으로 처리하되, Embedding Tensor를 쪼갠 후 합치는게 아닌 reshape & transpose operator로 shape을 변경 후 한꺼번에 행렬곱으로 계산해서 다시 reshape 함으로써 병렬처리가 가능하게끔 구현하는게 인상적이었음
- 행렬곱할 때 weight 와 곱하는건 weight variable 생성 후 MatMul 하는게 아니라 그냥 다 Dense로 처리하는게 구현 티이구나 느꼈음
- Multi-head Attention 구현팁: 쪼갠 다음에 weight 선언 후 매트릭스 곱? No! -> 쪼갠 다음에 Dense! -> 쪼개면 for loop 때문에 병렬처리 안 되잖아! -> 다 계산후에 쪼개자!
- Attention만 구현하면 얼추 끝날 줄 알았는데 Masking 지분이 70~80%였음
  - Masking은 logical 연산 (boolean)으로 padding 체크해서 하는게 일반적임
  - Masking은 input에도 해주고 loss에도 해줌
  - 마스킹 적용할땐 broadcasting 기법을 써서 하는게 일반적임
  - 아래의 두 경우 모두 가능함
    - ex)  $(40, 5, 10, 10) + (40, 1, 1, 10) == (\text{batch}, \text{head}, \text{seq}, \text{seq})$
    - ex)  $(40, 5, 10, 10) + (40, 1, 10, 10) == (\text{batch}, \text{head}, \text{seq}, \text{seq})$

## Abstract

- 대부분의 sequence 모델들은 encoder-decoder 프레임워크를 포함해서, 복잡한 RNN이나 CNN이었음
- 최고의 성능을 내는 모델 역시 encoder-decoder 프레임워크안에서 동작하지만, Attention mechanism으로 모델링했음
- 본 논문에서는 Transformer라는 오직 Attention mechanism에만 기초한 simple neural architecture를 제안함
- ~~RNN이나 CNN 같은거 없음..~~
- 2개의 machine translation task에 대해서 실험했고, 병렬화도 잘되고 학습시간도 짧으면서 퀄리티도 우월함을 확인함
- WMT 2014 English-to-German translation task에서 28.4 BLEU를 기록함 (다른 모델보다 2 BLEU 정도 높음)
- WMT 2014 English-to-French translation task에서는 41.8 BLEU로 SOTA 기록함 (8 GPUs로 3.5days 걸림)
- 다른 Task에서도 Generalize 잘됨

# 1. Introduction

- RNN, LSTM, GRU 등은 sequence modeling (language modeling, machine translation)에서 SOTA를 기록해왔음
- 대부분 노력들은 recurrent language model, encoder-decoder 구조 내에서 이뤄졌음
- RNN 계열의 모델들은 input, output의 sequence 내에서 position을 따라 계산되는 특징이 있는데, 이는 병렬처리를 막고, 시퀀스 길이가 길어지면 문제가 생기는 등의 여러가지 이슈가 있음
- Attention mechanism은 sequence 내에서 거리에 상관 없이 dependency modeling이 가능하기 때문에 sequence modeling을 정복할 수 있음
- 기존엔 RNN 계열에 Attention이 적용되었는데, 본 논문에서 제안하는 Transformer는 recurrence 방식을 피하고, 대신 전적으로 attention mechanism에만 의지해서 input, output 사이의 global dependency를 고려하고자 함
- Transformer는 병렬처리도 잘되고, 성능도 SOTA임 (학습시간은 12시간 걸렸음 with 8 P100 GPUs)

## 2. Background

- Sequential computation 문제를 해결하기 위해 the Extended Neural GPU , ByteNet , ConvS2S 등 CNN을 hidden representations을 병렬처리로 계산하기 위한 basic building block으로 사용하려는 연구들이 있었음
- 위의 연구들은 dependency를 고려하려는 관점에서 볼 때, distance에 대해서 linear (ConvS2S), 혹은 logarithm (ByteNet) 한 고려가 가능했음
- 하지만 position의 거리에 따른 dependency를 학습하는건 더 어렵게 만들었음 (~~이 부분은 잘 이해가 안감~~)
- Transformer에서는 constant 레벨까지 operation 숫자를 줄일 수 있음 (~~N개에 대한 dependency 있지만 병렬처리가 가능해서 그런듯...~~)
- Self-Attention (다른 말로, intra-attention)은 한 시퀀스 내에서 서로 다른 position들의 관계 representation을 계산하기 위한 attention mechanism임
- Self-attention은 이미 여러 task에서 성공했음 (reading comprehension, abstractive summarization, textual entailment, learning task-independent sentence representations)

### 3. Model Architecture

- 대부분 경쟁력있는 sequence transduction model은 encoder-decoder 형태임
- encoder는 input sequence of symbol representations ( $x_1, \dots, x_n$ )을 continuous representations ( $z_1, \dots, z_n$ )로 맵핑하는 역할을 함
- decoder는 주어진  $z$ 로 부터 output sequence ( $y_1, \dots, y_m$ )를 생성함
- 각 스텝에서 모델은 auto-regressive함 (previously generated symbol을 addtional input으로 사용)

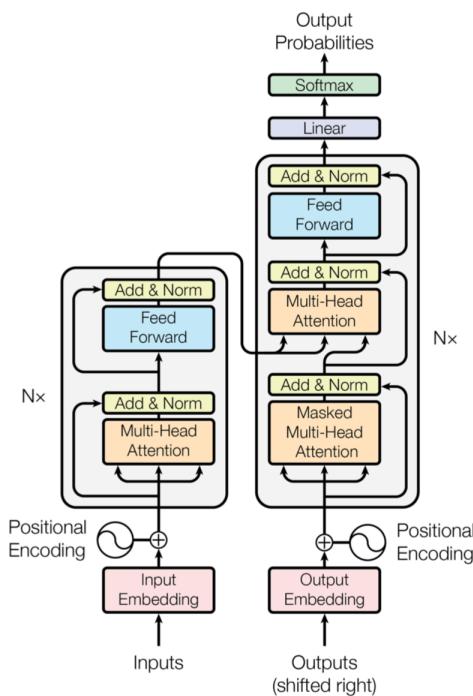


Figure 1: The Transformer - model architecture.

### 3. Model Architecture

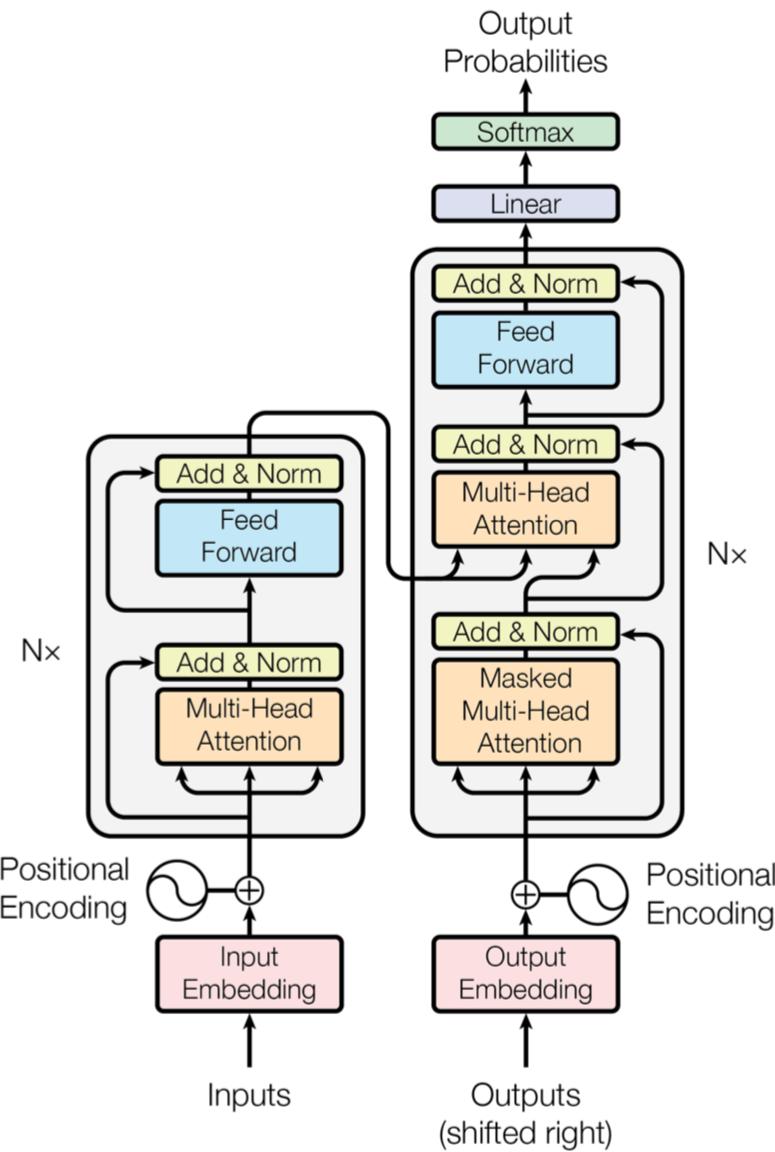
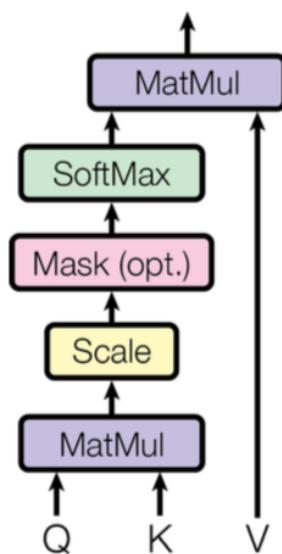


Figure 1: The Transformer - model architecture.

### 3. Model Architecture

Scaled Dot-Product Attention



Multi-Head Attention

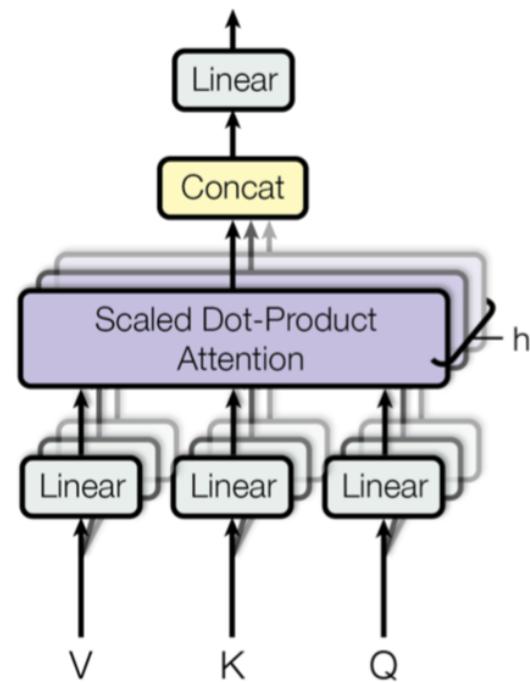
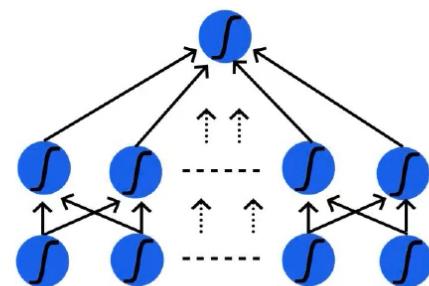
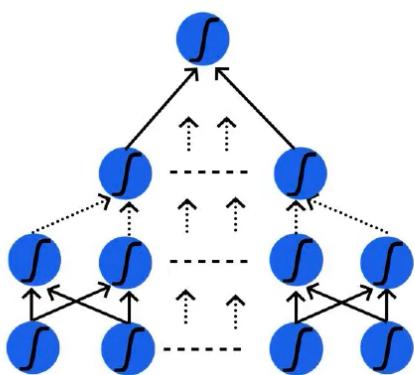
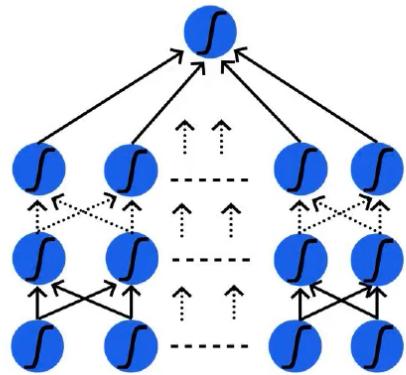
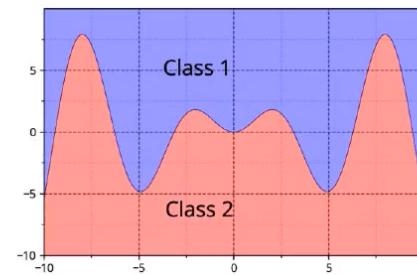
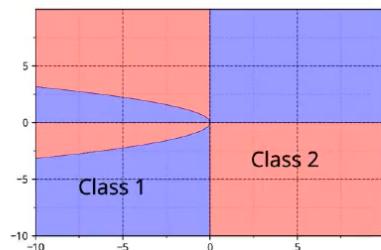
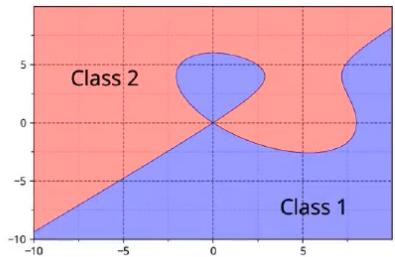


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

# Universal Approximation Theorem



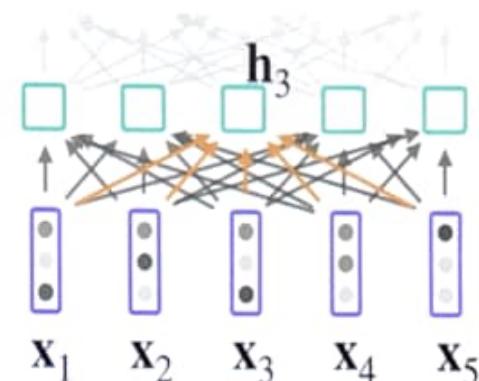
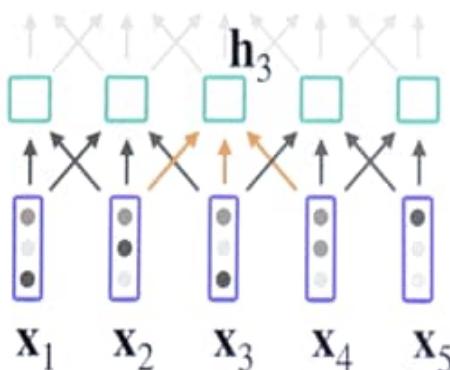
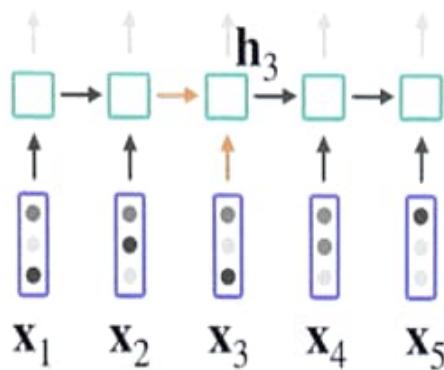
# RNN vs CNN vs Attention?

## Architectures for Sequence Modeling

$n$ : sequence length  
 $k$ : kernel size

### Self-Attention

	RNNs	CNNs	Self-Attention
Parallel	✗		
Infinite Context		✗	
Time Complexity	$O(n)$	$O(kn)$	$O(n^2)$
Dynamic Weights	✓	✗	✓



# RNN vs CNN vs Attention?

## Architectures for Sequence Modeling

n: sequence length  
k: kernel size

### RNNs

Elman et al., 1986  
Backpropagation and Schmidhuber, 1997  
Vinyals et al., 2014

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

### CNNs

Waibel et al., 1987  
LeCun et al., 1998  
Sifre et al., 2014

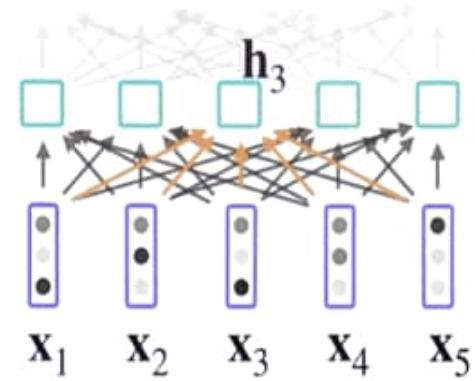
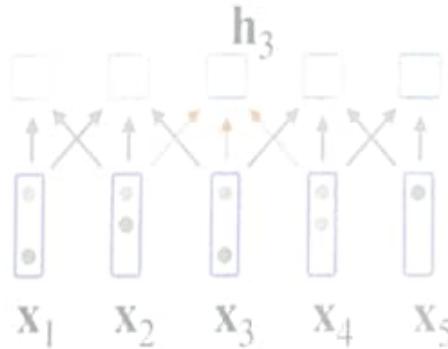
$$\mathbf{h}_t = f(\mathbf{x}_{t-\lfloor k/2 \rfloor}, \dots, \mathbf{x}_{t-\lfloor k/2 \rfloor+k})$$

### Self-Attention

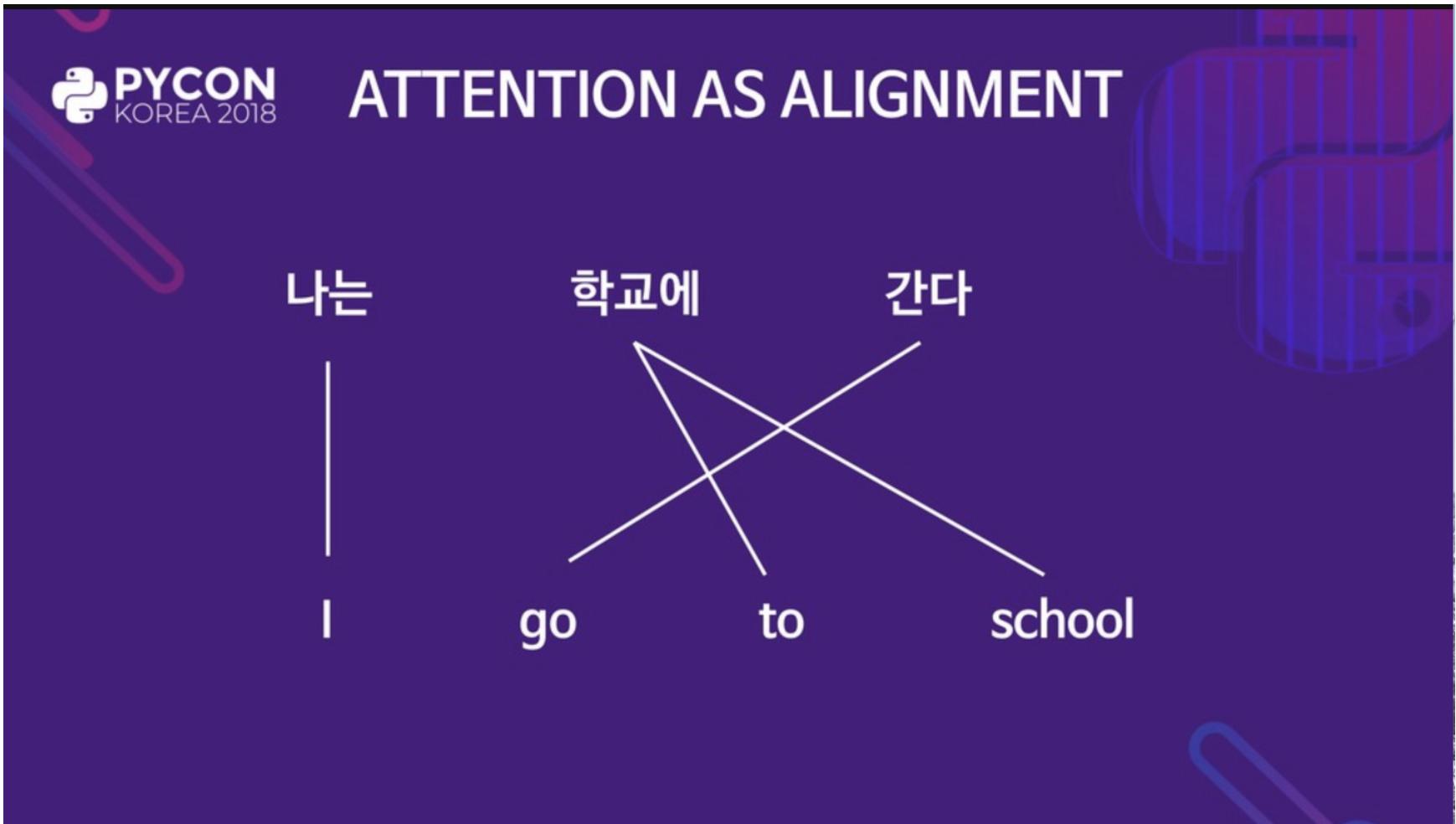
Cheng et al., 2016  
Parikh et al., 2016  
Vaswani et al., 2017

$$\mathbf{h}_t = \sum_{i=1}^n a_{t,i} \cdot \mathbf{x}_i$$

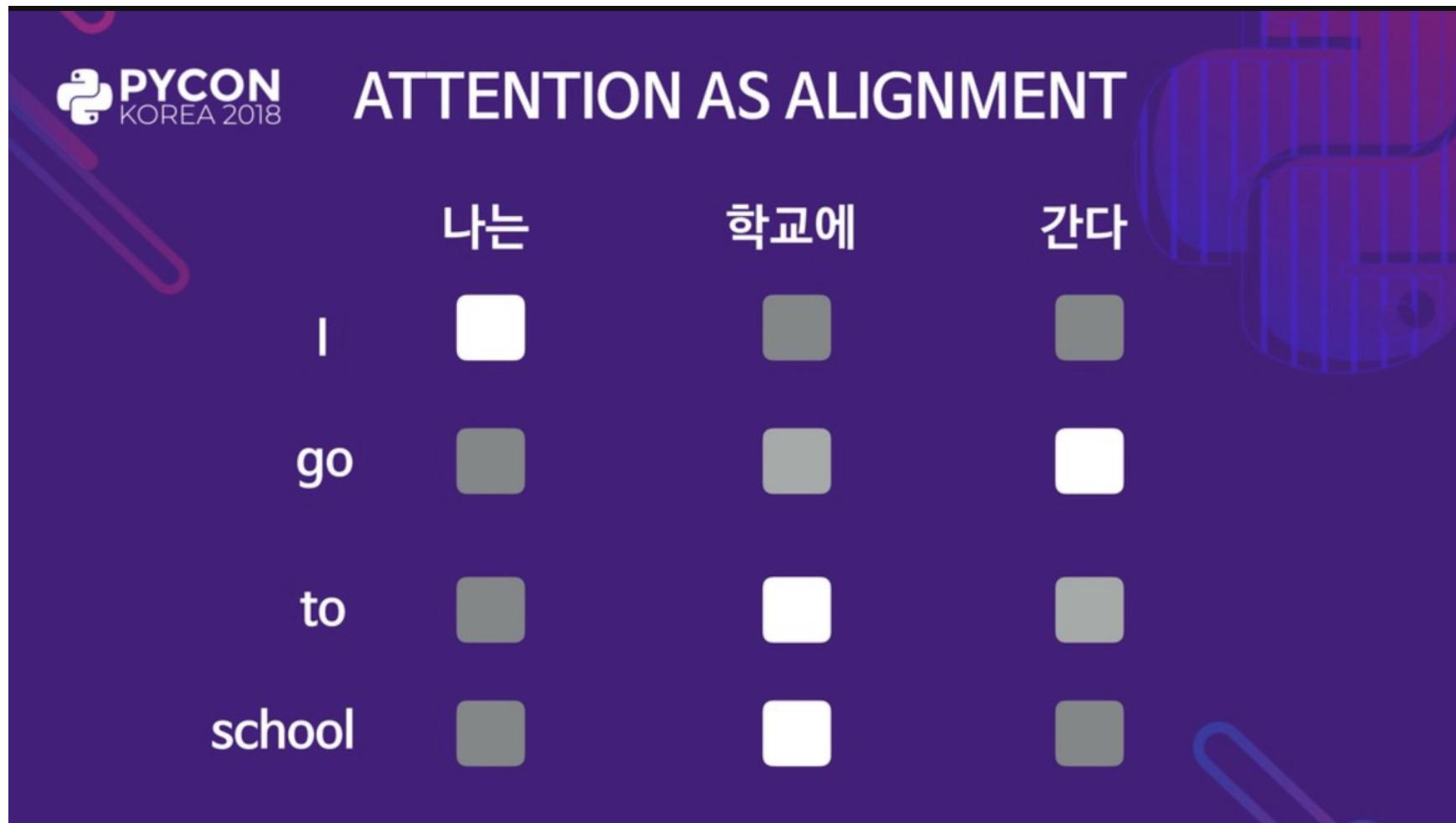
$$a_{t,i} = \frac{\exp(f(x_t, x_i))}{\sum_{j=1}^n \exp(f(x_t, x_j))}$$



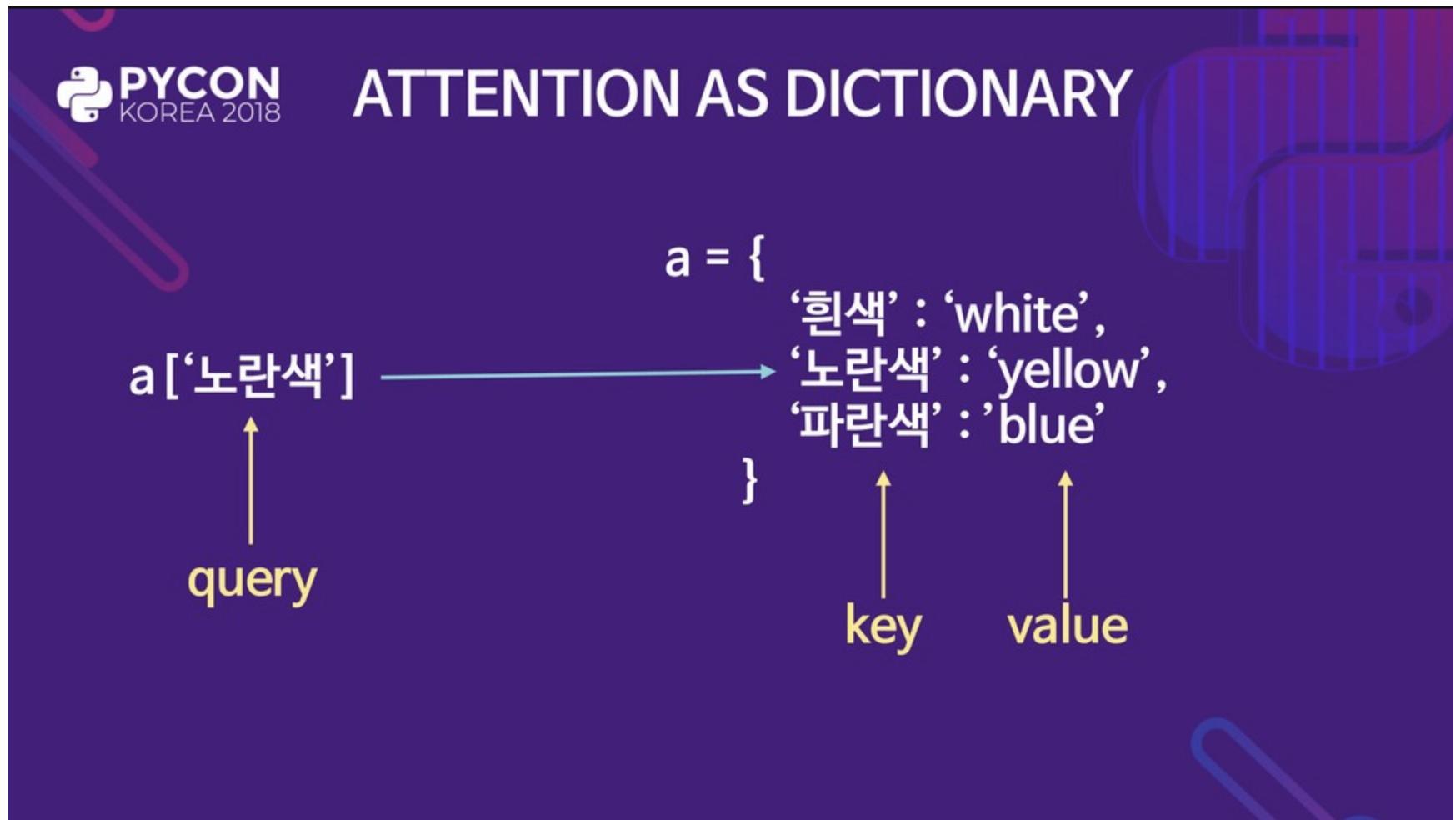
# What is Attention?



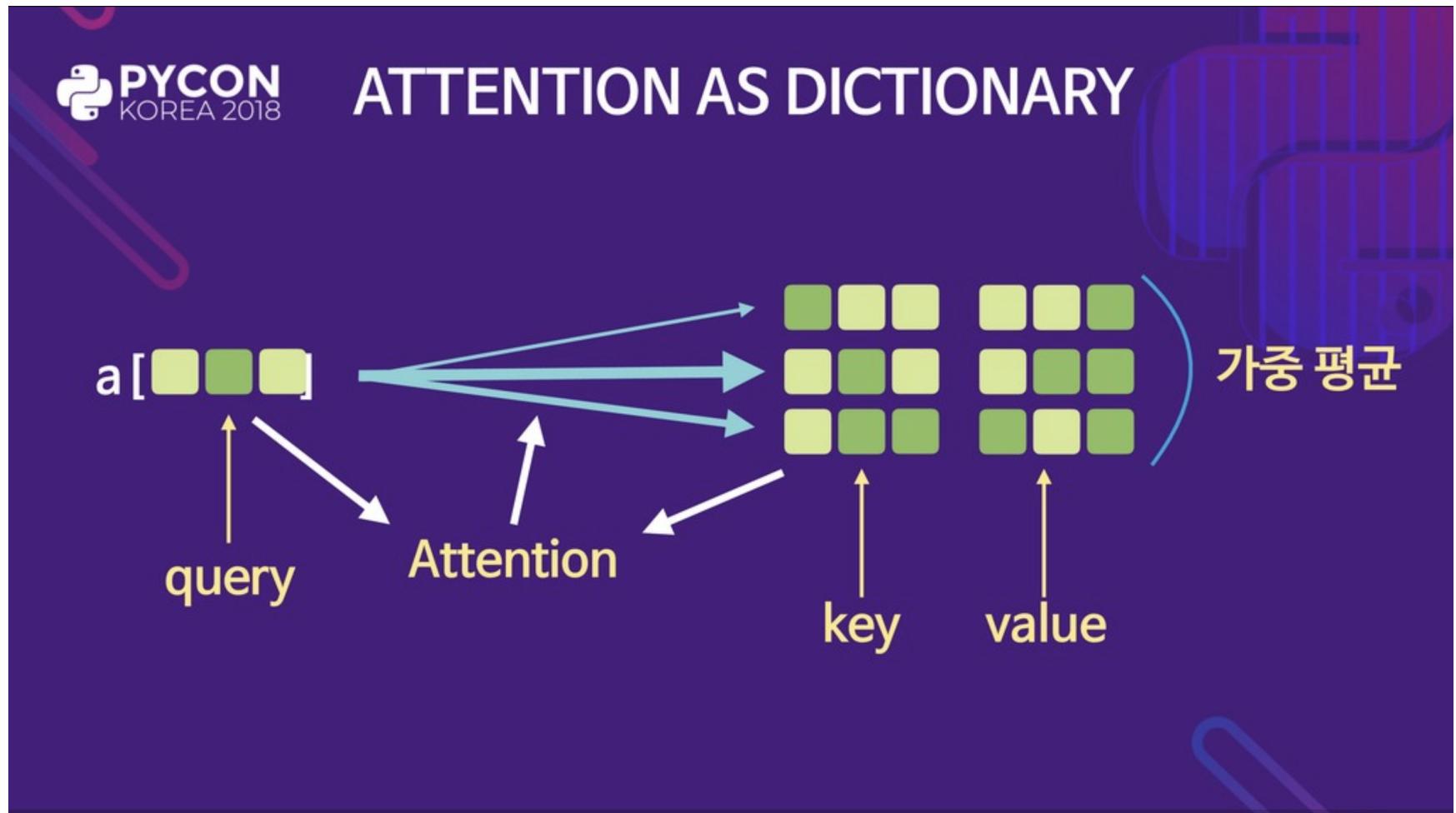
## What is Attention?



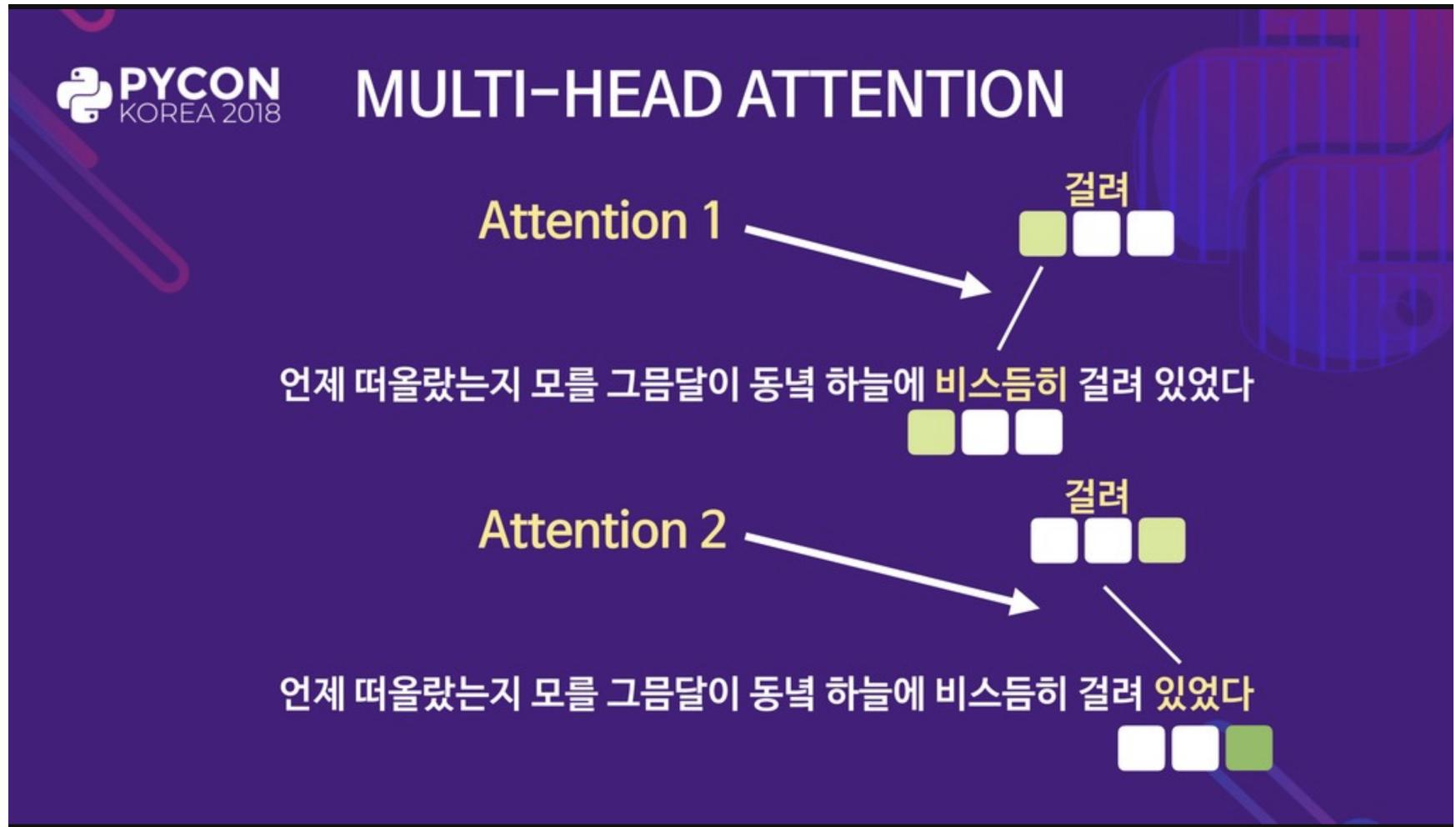
## What is Attention?



# What is Attention?



# What is Multi-head Attention?



- Reference: <https://speakerdeck.com/dreamonfly/soseol-sseuneun-dib-reoning-pycon-korea-2018?slide=21>

# Attention 시각화

## Attention Visualizations

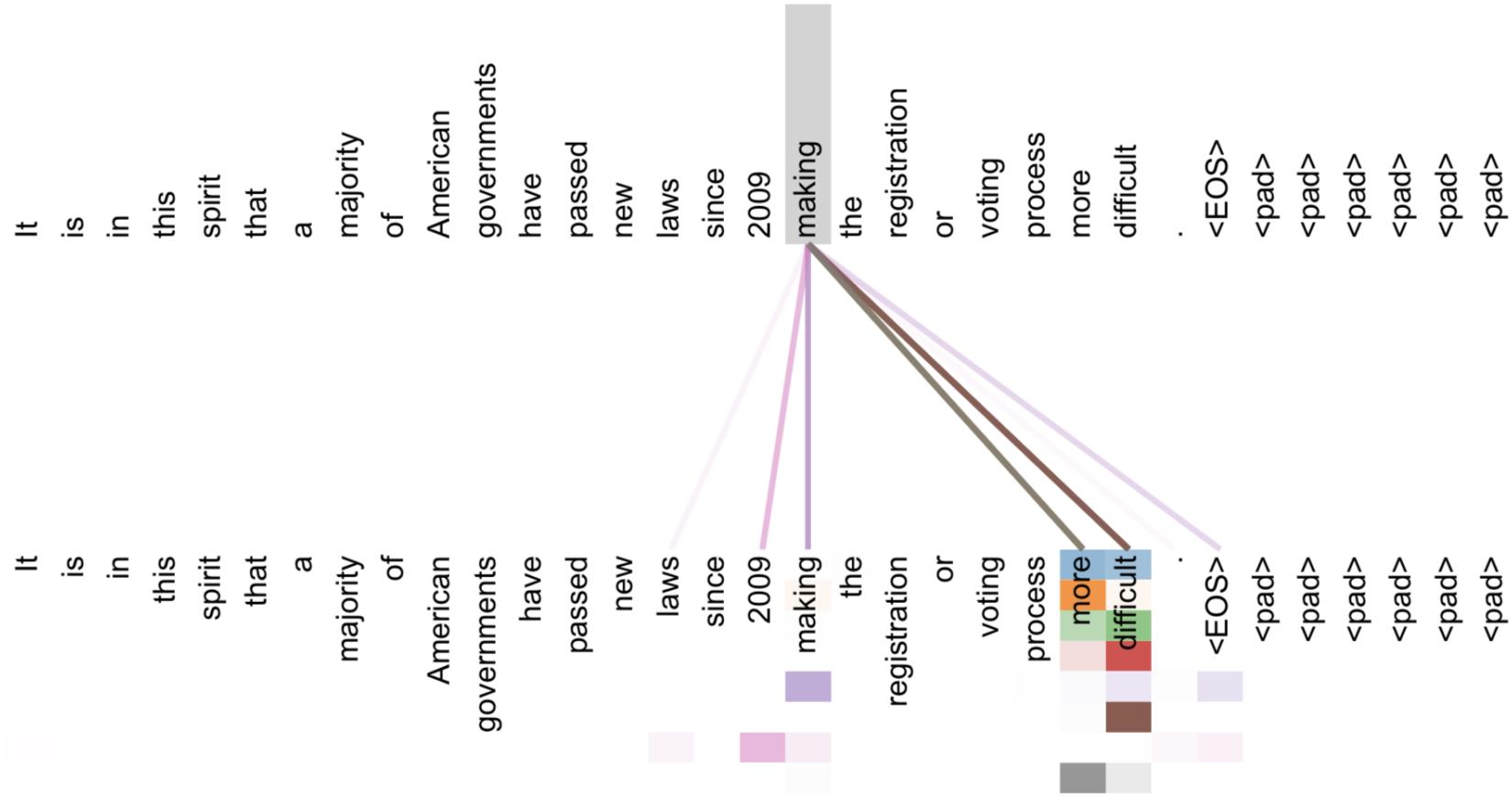


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb ‘making’, completing the phrase ‘making...more difficult’. Attentions here shown only for the word ‘making’. Different colors represent different heads. Best viewed in color.

# Attention 시각화

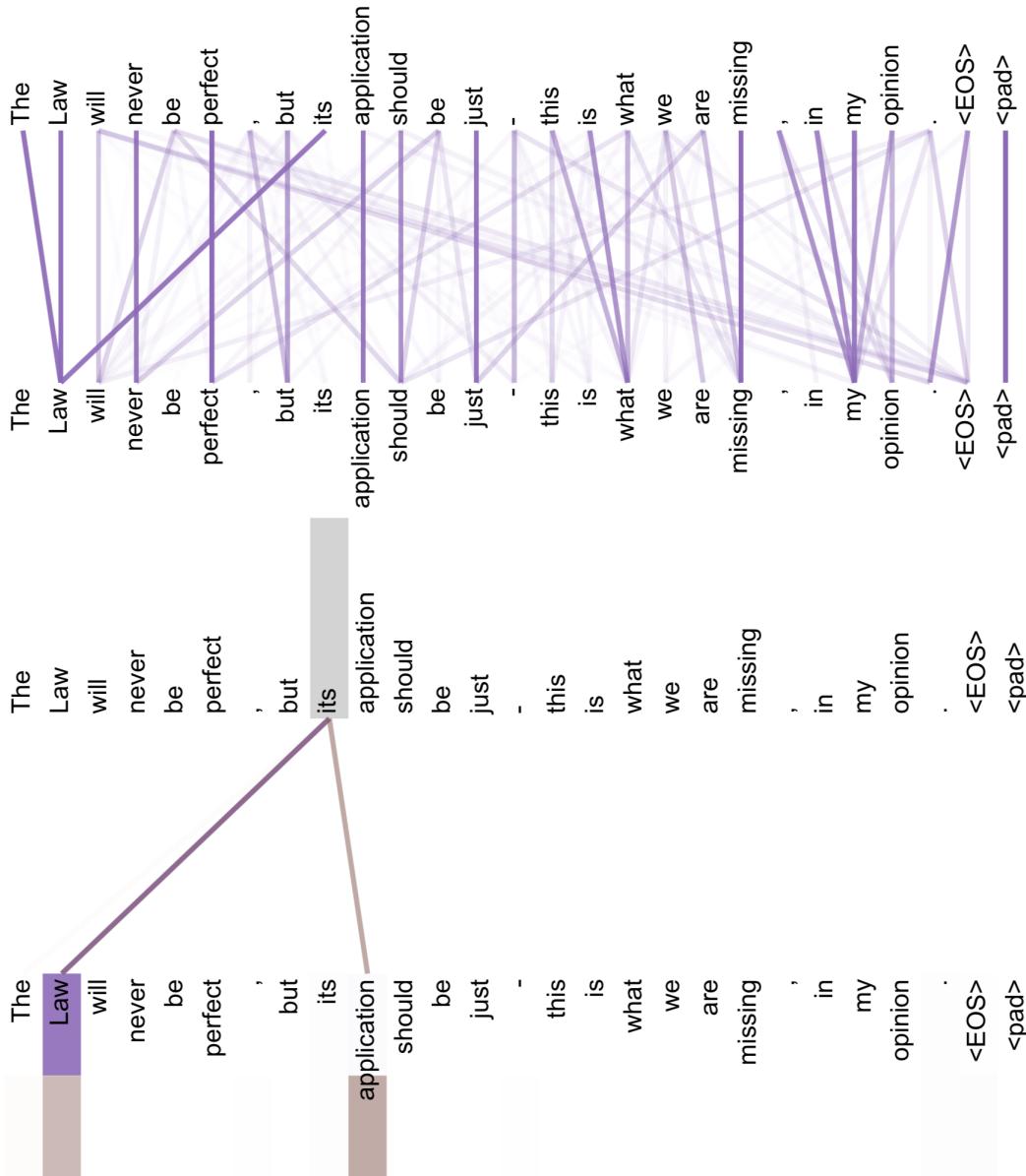


Figure 4: Two attention heads, also in layer 5 of 6, apparently involved in anaphora resolution. **Top:** Full attentions for head 5. **Bottom:** Isolated attentions from just the word 'its' for attention heads 5 and 6. Note that the attentions are very sharp for this word.

# Attention 시각화

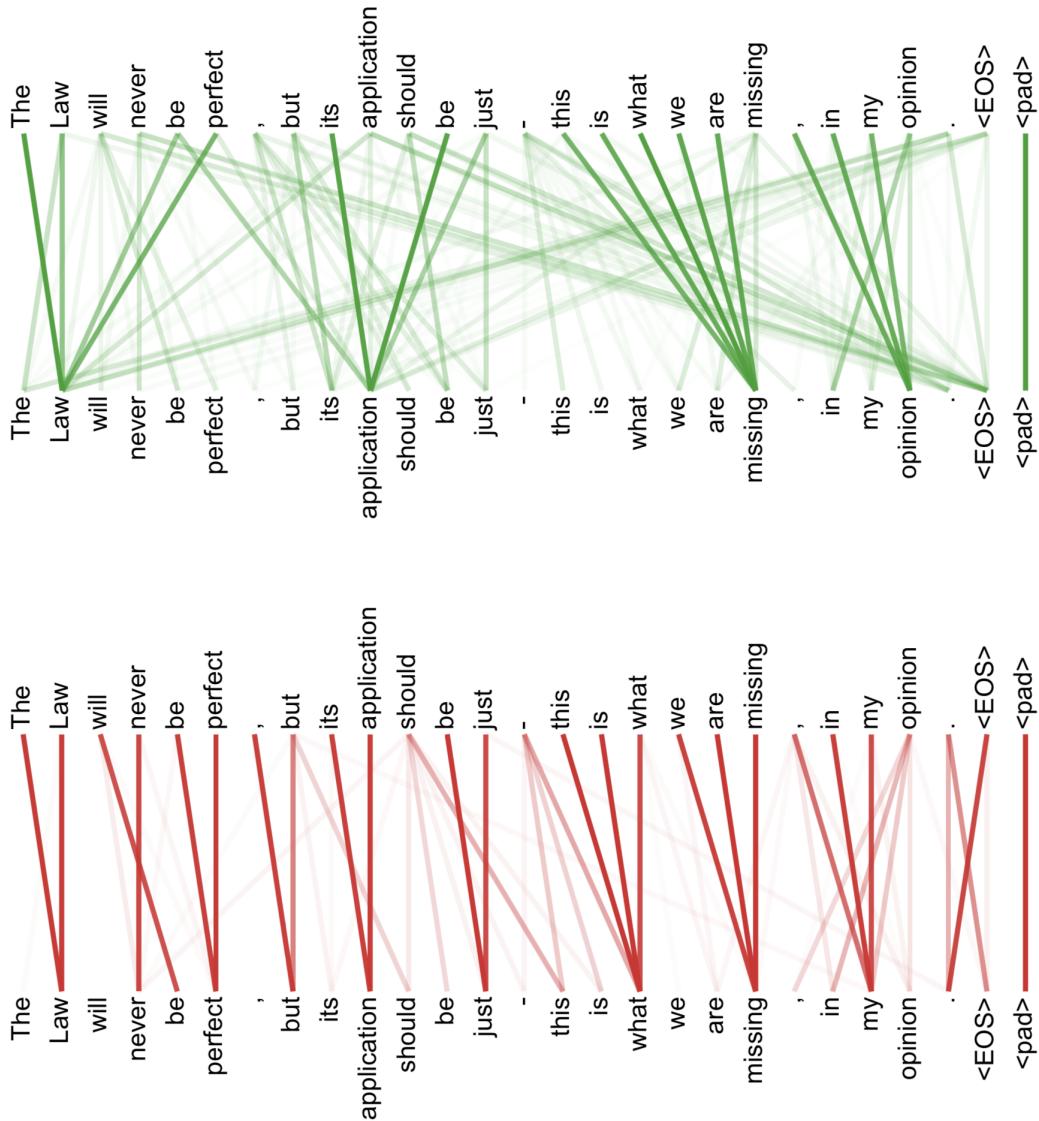


Figure 5: Many of the attention heads exhibit behaviour that seems related to the structure of the sentence. We give two such examples above, from two different heads from the encoder self-attention at layer 5 of 6. The heads clearly learned to perform different tasks.

### 3.1. Encoder and Decoder Stacks

- Encoder
  - 인코더는  $N = 6$  개의 identical layer의 스택으로 이루어져 있음
  - 각 layer는 두 개의 sub-layer로 이루어져 있음
    - 첫번째는 multi-head self-attention mechanism
    - 두번째는 position-wise fully connected feed-forward network

```
def sub_layer(self, x, training=False, padding_mask=None):
    out_1, attention_weight = self.mha(x, K = x, V = x, mask=padding_mask, flag="encoder_mask")
    out_1 = self.dropout1(out_1, training=training)
    out_2 = self.layer_norm_1(out_1 + x)
    out_3 = self.position_wise_fc(out_2)
    out_3 = self.dropout2(out_3, training=training)
    out_4 = self.layer_norm_2(out_2 + out_3)
    return out_4, attention_weight
```

### 3.1. Encoder and Decoder Stacks

- Encoder
  - 두 개의 레이어에 각각 residual connection & layer normalization을 적용함
  - 각 sub-layer의 output은 **LayerNorm( $x + \text{Sublayer}(x)$ )** 형태임
    - layerNorm은 `tf.keras.layers.LayerNormalization` API로 쉽게 구현 가능함
    - Hidden units들에 대해 Norm을 계산하기 때문에 Batch Norm과 다르다고함 (추가로 공부 필요)
  - where Sublayer( $x$ ) is the function implemented by the sub-layer itself
  - residual connection을 하기 위해서 모델에 있는 모든 sub-layer는(embedding layer 까지 포함) output의 dimension  $d_{\text{model}} = 512$ 로 셋팅함

```
for i in range(self.layer_num):
    x, attention_block1, attention_block2 = self.sub_layer(x, encoder_output, training, look_ahead_mask, padding_mask)
    attention_weights['decoder_layer{}_block1'.format(i + 1)] = attention_block1
    attention_weights['decoder_layer{}_block2'.format(i + 1)] = attention_block2
```

### 3.1. Encoder and Decoder Stacks

- Decoder
  - 디코더 또한  $N = 6$  개의 identical layer의 스택으로 이루어져있음
  - 디코더에는 2개가 아닌 3개의 sub-layer로 구성됨
    - 첫째는 Masked Multi-Head self-Attention임. 입력 포지션 상에서 이어서 나오는 것들을 마스킹해버려서 position  $i$ 를 예측할때 known outputs at position less than  $i$ 만 사용 가능하게 함
    - 두번째는 Multi-Head Attention임 얘는 encoder의 output에 적용됨
    - 세번째는 Feed Forward Network임
    - 결국 첫번째 sub-layer가 좀 특이한거고 두번째 sub-layer의 인풋에 encoder의 output이 들어가는 게 차이임

```
def sub_layer(self, x, encoder_ouput, training=False, look_ahead_mask=None, padding_mask=None):
    out_1, attention_weight_lah_mha_in_decoder = self.look_ahead_mha(x, K = x, V = x, mask = look_ahead_mask, flag="look_ahead_mask")
    out_1 = self.dropout1(out_1, training=training)
    out_2 = self.layer_norm_1(out_1 + x)
    out_3, attention_weight_pad_mha_in_decoder = self.mha(out_2, K = encoder_ouput, V = encoder_ouput, mask = padding_mask, flag="padding_mask")
    out_3 = self.dropout2(out_3, training=training)
    out_4 = self.layer_norm_2(out_3 + out_2)
    out_5 = self.position_wise_fc(out_4)
    out_6 = self.layer_norm_3(out_4 + out_5)

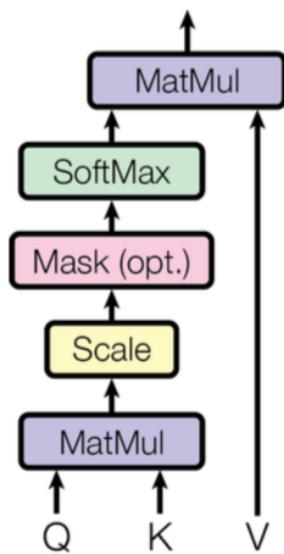
    return out_6, attention_weight_lah_mha_in_decoder, attention_weight_pad_mha_in_decoder
```

## 3.2. Attention

- Attention function은 query와 key-value pair를 output에 매핑하는 것으로 설명 가능함
- 여기서 말하는 query, key, value는 모두 vector를 의미함
- output은 value에 대한 weighted sum으로 계산되는데, 이 value에 할당되는 이 weight는 query에 대응되는 key의 compatibility function에 의해 계산됨
- 결과적으로 Query와 Key의 유사도로 weight 결정되고 이걸 적용하겠다는 것임
- key와 value는 같은 벡터를 의미함
  - key는 weight 뽑는 용
  - value는 weight를 적용할 때 실제 곱해지는 용

## 3.2. Attention

Scaled Dot-Product Attention



Multi-Head Attention

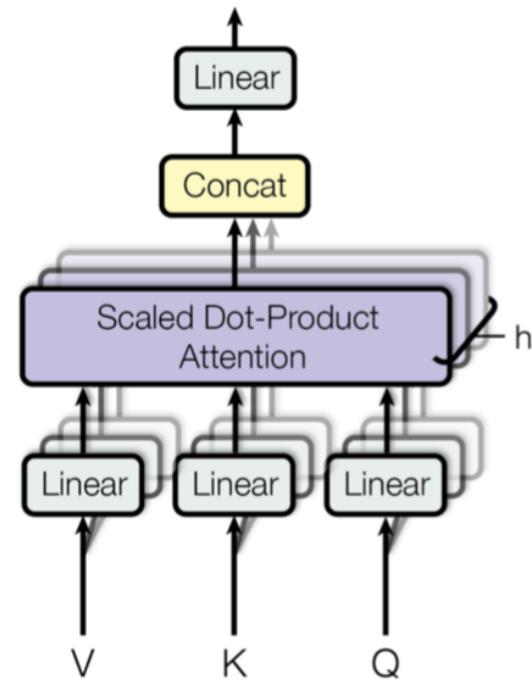


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

### 3.2.1. Scaled Dot-Product Attention

- 본 논문에서 쓰는 어텐션을 "Scaled Dot-Product Attention"이라 칭함
  - input은 queries, keys 라고 보면 되고 key의 dimension은  $d_k$  value의 dimension은  $d_v$  임
1. 먼저 query와 모든 key에 대해서 dot product를 계산함
  2. 계산한 값에 대해서 각각에 대해  $\sqrt{d_k}$ 로 나눠줌 (여기서 Scaled라는 단어가 나온게 아닌가 싶음, 근데 왜  $\sqrt{d_k}$ 로 나눠줄까? 다른것들도 어차피 똑같이 나눠주면 softmax에 영향 없을거 같은데 ->  $e^x$ 의 input 스케일에 따라 값이 차이나서 그런 것임, 다음 슬라이드 참고)
  3. softmax function으로 value에 적용할 weight를 얻음
- 실제로 쓸 땐, A set of queries에 대한 Attention은 동시에 계산하기 때문에 Matrix 형태로 사용함

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

### 3.2.1. Scaled Dot-Product Attention

- 많이 쓰이는 Attention은 주로 additive attention & dot-product attention인데 본 논문에서 쓴건 dot-product쪽임,  $1/\sqrt{d_k}$ 로 스케일링 해줬다는 차이가 있긴 함
- 이론적으로 둘 다 복잡도는 비슷하나, dot-product attention이 훨씬 빠르고 space-efficient한 이유는 highly optimized matrix multiplication code로 구현되어있기 때문임
- $1/\sqrt{d_k}$ 로 스케일링한 이유는  $d_k$  값이 클 경우 dot product 값이 커지고, 이는 softmax function이 small gradients를 갖게 만드는 것이 아닌지 의심이 되었고 이러한 효과를 막기 위해 스케일링 한 것임 ( $d_k$ 가 작은 경우 스케일링을 해주지 않으면 additive attention 성능이 더 좋다고 함)

```
def scaled_dot_product_attention(self, Q, K, V, mask=None, flag=None):
    # (batch, head_num, seq, split_embed_dim) * (batch, head_num, split_embed_dim, seq) = (batch, head_num, seq, seq)
    matmul_qk = tf.matmul(Q, K, transpose_b=True)
    dk = tf.cast(tf.shape(K)[-1], tf.float32) # dk dim
    scaled_dot_product_qk = matmul_qk / tf.math.sqrt(dk)

    if mask is not None:
        minus_infinity = -1e9
        scaled_dot_product_qk += mask * minus_infinity # broadcasting, masking에서 seq은 마지막자리
        # mask와 scaled_dot_product_qk의 차원은 다르지만, 마지막 차원이 같기 때문에 broadcasting이 가능함
    attention_weight = tf.nn.softmax(scaled_dot_product_qk, axis=-1)
    # (batch, head_num, seq, seq) * (batch, head_num, seq, split_embed_dim) = (batch, head_num, seq, split_embed_dim)
    scaled_attention_output = tf.matmul(attention_weight, V)

    return scaled_attention_output, attention_weight
```

### 3.2.2. Multi-Head Attention

- Single Attention function을  $d_{model}$  차원의 keys, values, queries에 적용하기보다 다른  $d_k, d_k, d_v$  차원을 갖는 queries, keys, values에  $h$  times 적용하는 것이 더 좋다는 걸 알게됨
- 한 마디로하면, 그냥 한번만 Attention function 쓰는게 아니라, 기존 Dim을 쪼개서 여러개로 나누고 거기에 여러번 Attention function 적용하면 더 다양한 Attention이 적용되고(여기엔 살짝 랜덤한.. 부분이 있겠지) 더 다양한 representation을 얻을 수 있게 된다는 말임
- 차원을 나눈 상태에서 Attention function은 병렬적으로 계산되고  $d_v$  차원의 output vectors 가 생성됨

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$
$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

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.

- dim of Q == num of tokens  $\times d_{model}$  로 생각하면 될 듯
- 본 논문에서는 multi-head를 8개로 나눠서, 전체 512 차원을 64 차원의 8개 유닛으로 만듬

### 3.2.2. Multi-Head Attention

- 원래는 input 임베딩을 쪼개고, 거기에 맞는 fc를 넣어주면 된다고 생각했는데,  $W_i^Q$ 를  $Q$ 에 곱해 주는거 자체가 input 임베딩을 쪼개는 것임 결과 차원이 num of toknes X  $d_k$ 로 나오기 때문에

```
def split_head(self, vector):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, num_heads, seq_len, depth)
    """
    batch_size = tf.shape(vector)[0]

    # (batch, seq, embed_dim) -> (batch, seq, head_num, split_embed_dim)
    x = tf.reshape(vector, (batch_size, -1, self.head_num, self.split_embed_dim))
    return tf.transpose(x, perm=[0, 2, 1, 3]) # (batch, head_num, seq, split_embed_dim)

def call(self, Q, K, V, mask=None, flag=None):
    # Query, Key 꺼낼 필요 없이 3개 복사해서 쓰면 됨
    # 쪼갠 다음에 weight 선언 후 매트릭스 곱? -> 쪼갠 다음에 Dense -> 쪼개면 for loop 때문에 병렬처리 안되잖아 -> 다 계산후에 쪼개자 -> 쪼개지말고 reshape으로 하면 더 깔끔하다
    multi_head_Q = self.split_head(self.Wq(Q))
    multi_head_K = self.split_head(self.Wk(K))
    multi_head_V = self.split_head(self.Wv(V))

    self.scaled_attention_output, self.attention_weight = self.scaled_dot_product_attention(multi_head_Q, multi_head_K, multi_head_V, mask, flag)

    # (batch, head_num, seq, split_embed_dim) -> (batch, seq, split_embed_dim)
    self.concat_scaled_attention = tf.reshape(self.scaled_attention_output, (tf.shape(Q)[0], -1, self.embed_dim))

    return self.concat_scaled_attention, self.attention_weight
```

### 3.2.3. Applications of Attention in our Model

- 트랜스포머에서는 멀티헤드 어텐션을 3곳에 적용함
- 첫번째, "**encoder-decoder Attention**" layer 에 적용함
  - decoder의 input에 대해서 Attention 적용하고 그 결과를 Query로 만든 다음 레이어에 적용할때 encoder의 output을 key,value로 사용함
  - 이렇게 하면 decoder의 input도 모두 고려하면서, encoder의 output도 모두 고려하는 seq2seq 모델에 attention을 적용한것과 비슷하게 됨
- 두번째, "**self-attention layer**" in encoder 에 적용함, 이 역시도 sequence 내의 모든 position을 다 고려할 수 있음
- 세번째, "**self-attention layer**" in decoder 에 적용함, 보지못한 정보를 보는 것을 막기 위해 ( to prevent leftward information flow in the decoder to preserve the auto-regressive property ) scaled dot-product attention안에 마스킹을 적용함 (minus infinity) (Figure 2)

### 3.2.3. Applications of Attention in our Model

```
def create_padding_mask(seq):
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

    # add extra dimensions so that we can add the padding
    # to the attention logits.
    return seq[:, tf.newaxis, tf.newaxis, :, :] # (batch_size, 1, 1, seq_len)

def create_look_ahead_mask(step_size):
    """
    - decoder에서 각 상태에 대한 self-attention이 inference step에 맞게 future token을 보지 못하게 해야됨
    - 각 step이 소유하고 있는 attention은 step개수 만큼임
    - future token보지 못하게 하려면 각 step에서 future step에 대해서 마스킹 해야함
    - 1 step에서는 나머지 n-1개 masking, 2번째 스텝에서는 앞에 두개 빼고 나머지 n-2개 마스킹
    - 이렇게 하면 역삼각형 모양의 마스킹 매트릭스가 나옴
    - step * step 을 대각선으로 나눈 모양임

    example)
    x = tf.random.uniform((1, 3))
    temp = create_look_ahead_mask(x.shape[1])
    temp:
    <tf.Tensor: id=311521, shape=(3, 3), dtype=float32, numpy=
    array([[ 0.,  1.,  1.],
           [ 0.,  0.,  1.],
           [ 0.,  0.,  0.]], dtype=float32)>

    Special usecase:
    tf.matrix_band_part(input, 0, -1) ==> Upper triangular part.
    tf.matrix_band_part(input, -1, 0) ==> Lower triangular part.
    tf.matrix_band_part(input, 0, 0) ==> Diagonal.

:param step_size:
:return:

"""
mask = 1 - tf.linalg.band_part(tf.ones((step_size, step_size)), -1, 0)
return mask # (seq_len, seq_len)

def create_masks(inp, tar):
    # Encoder padding mask
    enc_padding_mask = create_padding_mask(inp)

    # Used in the 2nd attention block in the decoder.
    # This padding mask is used to mask the encoder outputs.
    dec_padding_mask = create_padding_mask(inp)

    # Used in the 1st attention block in the decoder.
    # It is used to pad and mask future tokens in the input received by
    # the decoder.
    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    dec_target_padding_mask = create_padding_mask(tar)
    combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)

    return enc_padding_mask, combined_mask, dec_padding_mask
```

### 3.3. Position-wise Feed-Forward Networks

- Attention sub-layers 다음엔 FC(Fully connected feed-forward network)가 붙게 됨
- two linear transformation with ReLU가 적용됨

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

- input and output dim,  $\dim_{model} = 512$
- inner-layer dim,  $\dim_{ff} = 2048$

### 3.4. Embeddings and Softmax

- 다른 sequence transduction model과 같이 여기서도  $d_{model}$  차원을 갖는 learned embeddings을 사용함
- learned linear transformation & softmax function을 사용함
- two embedding layers, pre-softmax linear transformation에 대해서 weight matrix를 공유함
- embedding layer에서 weights에  $\sqrt{d_{model}}$ 를 곱해줌 (스케일링)

```
x = self.embed(inputs) # (batch, seq, word_embedding_dim)
x *= tf.math.sqrt(tf.cast(self.embed_dim, tf.float32))
x = self.add_positional_encoding(x)
```

### 3.5. Positional Encoding

- 본 모델에서는 recurrence도 convolution도 없기 때문에 position 정보를 알 수가 없음
- 그렇기 때문에 position information을 inject해줘야함
- "positional encodings"를 input embedding에 더하겠음 ( input embedding +  
positional encodings )

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

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

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  $2\pi$  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  $PE_{pos}$ .

## 3.5. Positional Encoding

```
def add_positional_encoding(self, embed):
    # ref: https://colab.research.google.com/github/tensorflow/docs/blob/master/si
    def get_angles(pos, i, d_model):
        angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
        return pos * angle_rates

    def positional_encoding(position, d_model):
        angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                               np.arange(d_model)[np.newaxis, :],
                               d_model)

        # apply sin to even indices in the array; 2i
        sines = np.sin(angle_rads[:, 0::2])
        # apply cos to odd indices in the array; 2i+1
        cosines = np.cos(angle_rads[:, 1::2])
        pos_encoding = np.concatenate([sines, cosines], axis=-1)
        pos_encoding = pos_encoding[np.newaxis, ...]

        return tf.cast(pos_encoding, dtype=tf.float32)

    pos_encoding = positional_encoding(self.vocab_size, self.embed_dim)
    seq_len = tf.shape(embed)[1]
    return embed + pos_encoding[:, :seq_len, :]
```

## 4. Why Self-Attention

- self-attention과 다른 알고리즘 비교하겠음
- 대부분은 Self-Attention이 좋음 Complexity 빼고! 이 부분은 주변의 r개만 보는 restricted self-attention 버전으로 해결할수 있을듯

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

## 5. Training

### 5.1. Training Data and Batching

- Data1: WMT 2014 English-German dataset
  - 4.5 million sentence pairs
  - byte-pair encoding
  - source-target vocabulary of about 37,000 tokens
- Data2: larger WMT 2014 English-French dataset
  - 36M sentences
  - split tokens in a 32,000 word-piece vocabulary
  - sentence length가 비슷한 애들끼리 batch 처리함
  - 각 배치당 25,000 source tokens, 25,000 target tokens 정도를 포함함

## 5.2. Hardware and Schedule

- 8 NVIDIA P100 GPUs 사용
- base model
- each training step은 0.4 초 걸림
- 학습에 사용한 steps or time: 100,000 steps or 12 hours
- big models
- 스텝당 1.0 초 걸림, 300,000steps (3.5 days) 소요

## 5.3. Optimizer

- Adam
- $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$
- learning rate 바꿔줌
- *warmup\_steps* 에서는  $|lr0|$  linearly 증가함
- 그 후에는 inverse square root of the step number 비율로 감소함
- *warmup\_steps* = 4,000으로 셋팅함

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

## 5.4. Regularization

- 3가지 기법 적용함 (왜 논문에는 근데 레벨이 2개밖에 없지..)
  - Residual Dropout
    - 각 sub-layer의 output에 미리 적용해서 나중에 sub-layer input에 더해주고 정규화함
    - input embedding과 positional embedding을 더한 결과에 대해서도 적용함 (encoder & decoder 모두)
    - $P_{drop} = 0.1$
  - Label Smoothing
    - label smoothing 적용함  $\epsilon_{ls} = 0.1$
    - This hurts perplexity, as the model learns to be more unsure
    - 하지만 Accuracy와 BLEU score는 올라감
    - 출처: [36] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. CoRR, abs/1512.00567, 2015.

# 6. Results

## 6.1. Machine Translation

- Beam search 적용함
  - beam size = 4
  - length penalty  $\alpha = 0.6$
- Hyper params는 Development set 기준으로 실험적으로 선택함
- Maximum output length = input length + 50

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

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	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	<b><math>3.3 \cdot 10^{18}</math></b>	
Transformer (big)	<b>28.4</b>	<b>41.8</b>	$2.3 \cdot 10^{19}$	

## 6.2. Model Variations

- 모델 컴포넌트들의 중요도를 평가하기 위해 varied model에 대해서 평가함

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	$N$	$d_{\text{model}}$	$d_{\text{ff}}$	$h$	$d_k$	$d_v$	$P_{\text{drop}}$	$\epsilon_{ls}$	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

## 6.3. English Constituency Parsing

- Transformer가 generalize 잘 되는지 평가함
- 생각보다 잘됨

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)

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

## 7. Conclusion

- Attention만 의존하는 모델 처음으로 발표함
- recurrent layer를 multi-headed self-attention을 쓰는 encoder-decoder 구조로 대체함
- rnn, cnn보다 학습 빨리됨
- NMT에서 SOTA 찍음
- 다른 도메인에도 적용 될수 있을거라 생각함

## Reference

- <https://jalammar.github.io/illustrated-transformer/>
- <https://brunch.co.kr/@kakao-it/155>
- <https://pozalabs.github.io/transformer/>

