부스트캠프 AI Tech 2기

boostcamp ai tech

Transformer

Attention is All you Need, NIPS 2017

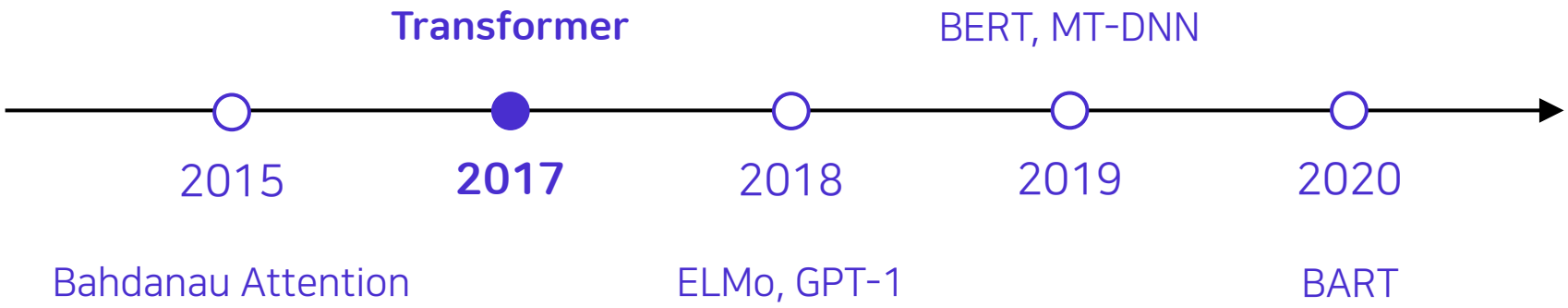
Email : jinmang2@gmail.com

GitHub : github.com/jinmang2

Huggingface Hub: huggingface.co/jinmang2

진명훈_T2216

Boostcamp AI Tech 2 NLP 논문 모임에 오신 여러분 환영합니다!



Quick Research History

Attention is all you need!

Experiment and Results

Code Review

00. 논문 및 저자 소개

Ashish Vaswani
Senior Research Scientist, Google Brain
Relative Inductive Bias, Relative Position repr, ViT, Music Transformer
https://scholar.google.com/citations?user=6rUjwXUAAAAJ&hl=en

Niki Parmar
Google Brain
ViT, Stand-alone self-attn, grammatical error correction
https://www.linkedin.com/in/nikiparmar

Llion Jones
Google Brain
One model to learn them all,
https://scholar.google.com/citations?user=h0zfd-0AAAAJ&hl=en

Lukasz Kaiser
Google Brain
GNMT, ViT, Reformer, Universal Transformer
https://scholar.google.co.kr/citations?user=JWmiQR0AAAAJ&hl=ko

Noam Shazeer
Google Research
T5, ViT, Music Transformer
https://www.linkedin.com/in/noam-shazeer-3b27288

Jakob Uszkoreit
Google Brain
Decomposable Attention Model for NLI
http://research.google.com/intl/en/pubs/author37567.html

Aidan N. Gomez
Univ of Oxford
https://aidangomez.ca/

Illia Polosukhin
NEARProtocol Co-Founder
Blockchain enthusiast, Tensorflow
https://twitter.com/ilblackdragon

Attention Is All You Need

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 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

- ✓ Sequence Transduction문제는 Reccurent or Convolution 기반의 EncDec 모듈 기반 (with attention)
- ✓ Propose Transformer, based solely on attention mechanisms!
- ✓ 성능도 우수, 병렬화로 학습 속도도 빨라짐!
- ✓ 기존 대비 Neural 기반으로 SOTA 달성, Generalize도 잘함!

01. Quick Research History

Sequence Transduction Problem?

By Graves,

- 음역(transliteration): 소스 형식의 예제에 따라 대상 형식으로 단어를 생성
- 철자 수정(spelling correction): 주어진 잘못된 단어 철자에서 올바른 단어 철자를 생성
- 굴절 형태학(inflectional morphology): 소스 시퀀스와 컨텍스트가 주어진 새로운 시퀀스를 생성
- 기계번역: 소스 언어로 된 예제에서 대상 언어로 단어 시퀀스를 생성
- 음성인식: 주어진 오디오 시퀀스로 텍스트 시퀀스 생성
- 단백질 2차 구조 예측: 아미노산의 입력 서열(NLP가 아닌)이 주어진 3D 구조를 예측
- TTS(Test-To-Speech): 또는 음성 합성으로 오디오 주어진 텍스트 시퀀스를 생성

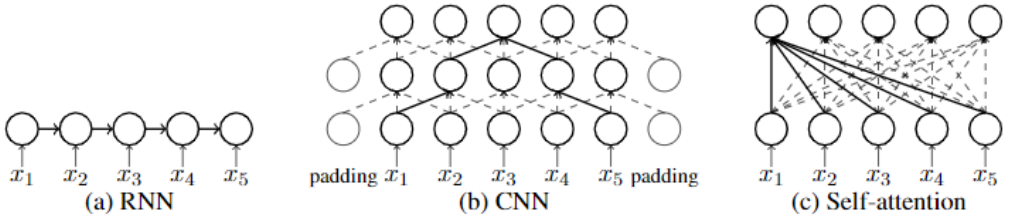


Figure 1: Architectures of different neural networks in NMT.

기존에는 RNN으로 위 문제를 풀었어요! (FAIR의 Convolutional Seq2Seq도 있구요!)

그러나 RNN은 기본적으로 time step을 sequential하게 받는다는 본질적인 문제가 있음
이를 아래와 같은 트릭으로 해결! (그러나 아직도 처리되지 않음...)

- Factorization tricks, Factorization tricks for LSTM networks ICLR 2017
- Conditional Computation, Outrageously Large Neural Networks: The Sparsely-Gated Mixture of Experts Layer ICLR 2017

Attention Mechanism!

어제 다룬 것과 같이 RNN 기반의 Encoder-Decoder 기반 연구에 Attention Mechanism을 추가하여 성능을 PBSMT 비슷하게 올렸었지요!

- RNNSearch (Bahdanau Attention), Neural Machine Translation by Jointly Learning to Align and Translate ICLR 2015
- Structured Attention Network (윤킴 교수님 논문), Structured Attention Networks ICLR 2017

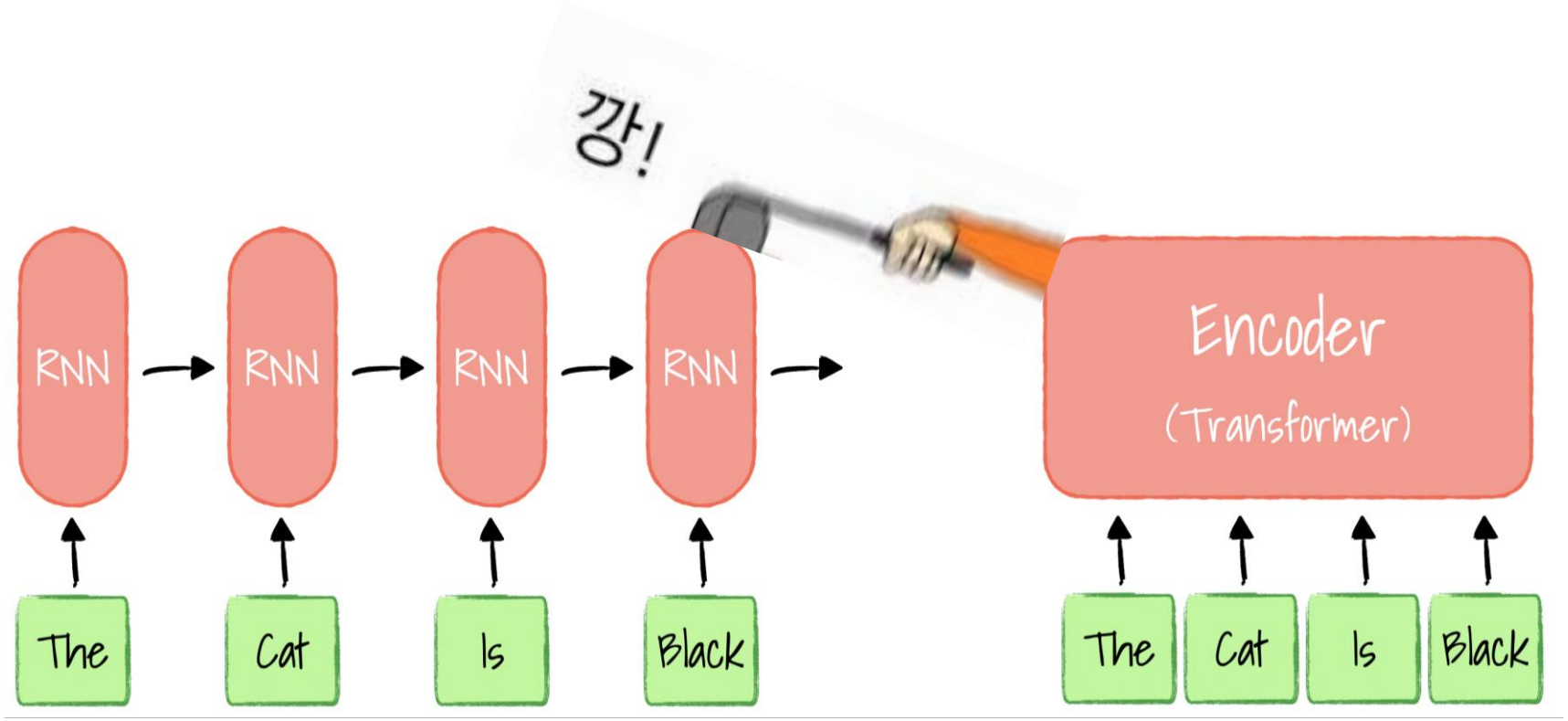
그러나 이전 연구 중 RNN과 Attention을 분리해서 사용한 경우가 있었는데 성능이 좋았음!

- Decomposable Attention (사실상 전신), A Decomposable Attention Model for Natural Language Inference EMNLP 2016

End to End Memory Network라는 논문에서 multi-hop이란 개념을 소개했는데 이것도 성능에 도움이 됐어요!

https://arxiv.org/abs/1211.3711
https://dos-tacos.github.io/translation/transductive-learning/

02. Attention is all you need!



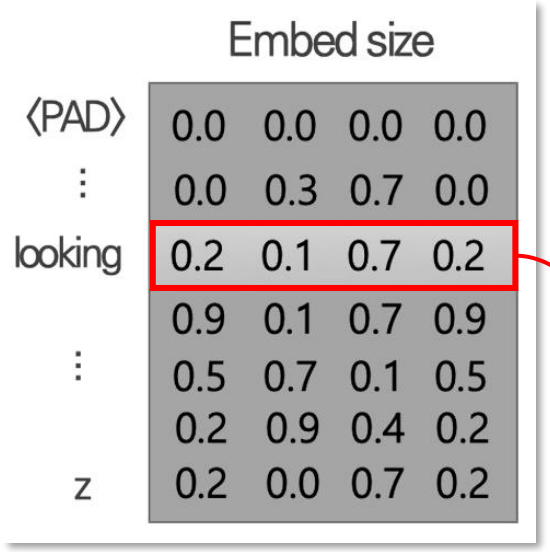
한 밤 중에 일어난 RNN 살인 사건! 끝판왕 Transformer의 등장입니다!

02. Attention is all you need!

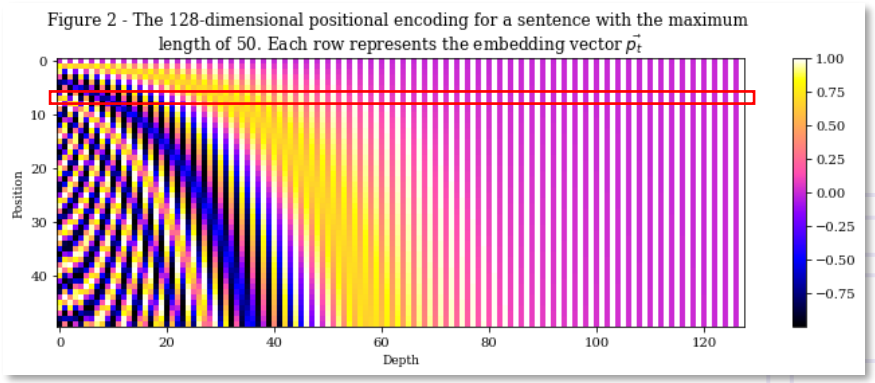
- Input Embeddings + Positional Encodings

Embedding: Learnable
Encoding: 고정된 규칙

이후 논문(BERT)에선 Positional Embedding을 사용해요!
+ Absolute Positional Embedding
+ Relative Positional Embedding
-> Attention Matrix에 직접 주입!



+



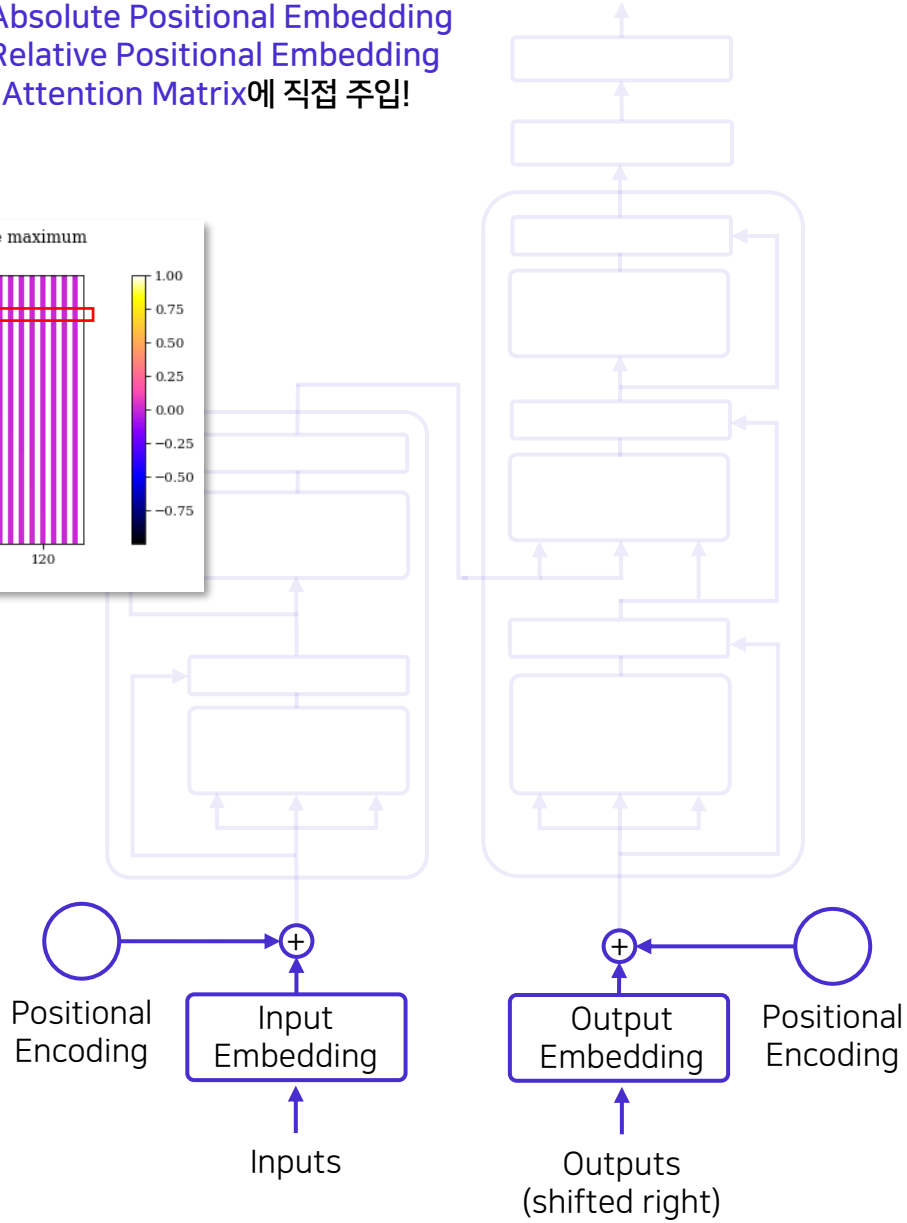
Professor looking for a Ph.D student

Transformer부턴 유독 Word2Vec같은 Word Embedding 기법을 사용하지 않음

대신 주로 nn.Embedding을 사용!

Embedding Layer는 위 처럼 각 단어(토큰)에 대한 vector를 가지고 있는 mapping table이고

Backpropagation으로 각 단어(토큰)의 벡터가 업데이트됨



https://gist.github.com/jinmang2/c2d41939ee9800b9d9d8d485059870cb

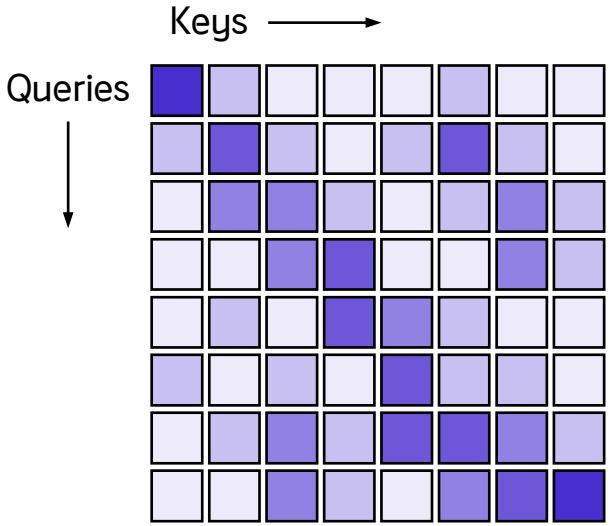
02. Attention is all you need!

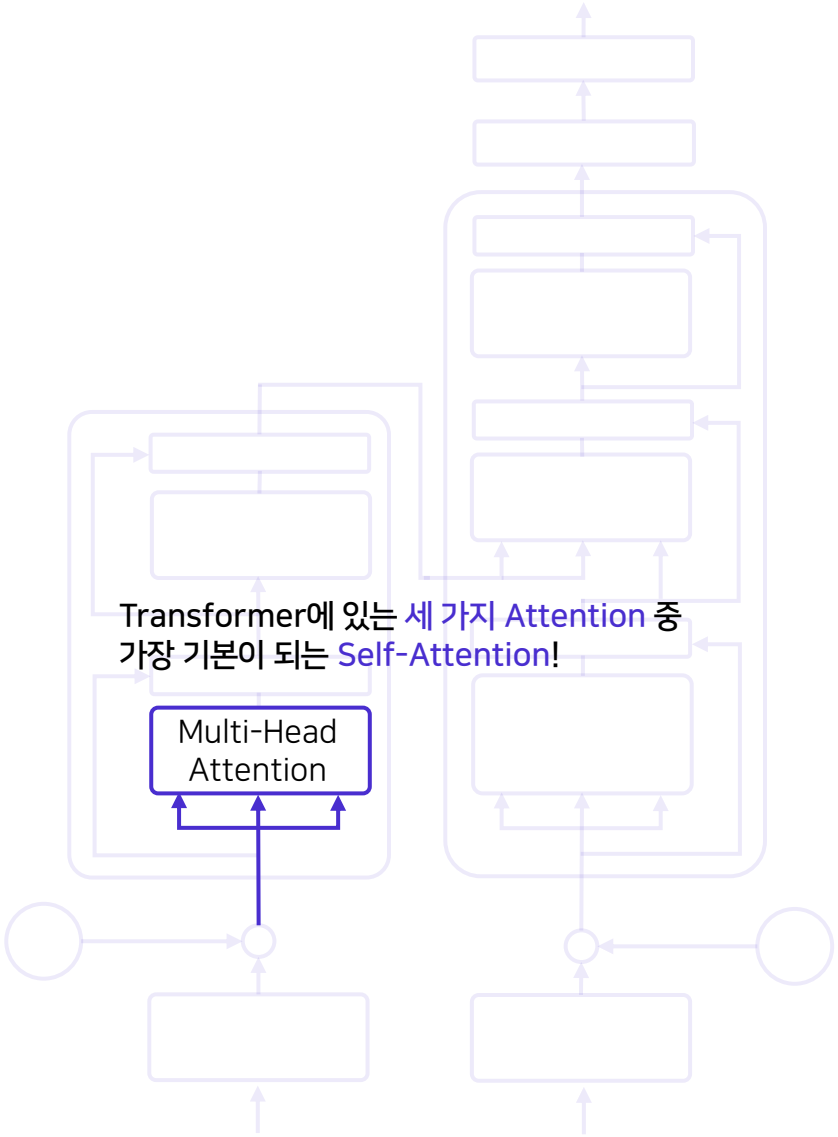
- Self-Attention

같은 문장끼리 스스로 (Self) Attention 계산을 수행!

최종적으로 오른쪽과 같은 Attention Map을 만들면 성공!

자기 자신을 얼마나 잘 아는가?!





https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

02. Attention is all you need!

Multi-Head 연산을 위해 Hidden Dimension의 Tensor를 쪼개줍니다!

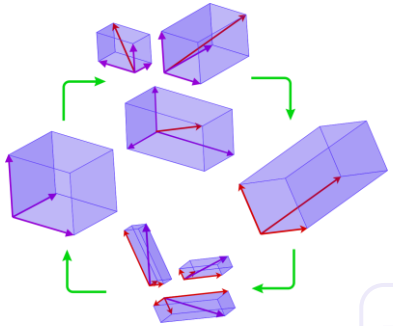
Multi-Head Scale Dot Product Self-Attention

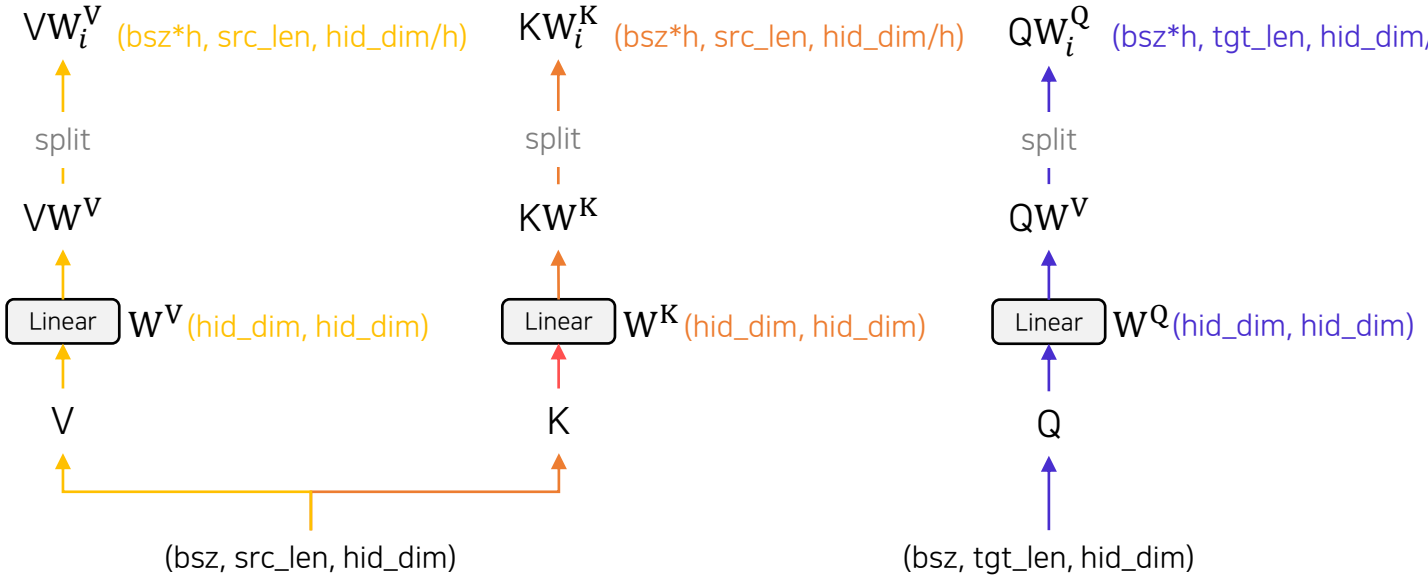
Transformer의 핵심 모듈인 MHSA에 대해 이해해 봅시다!

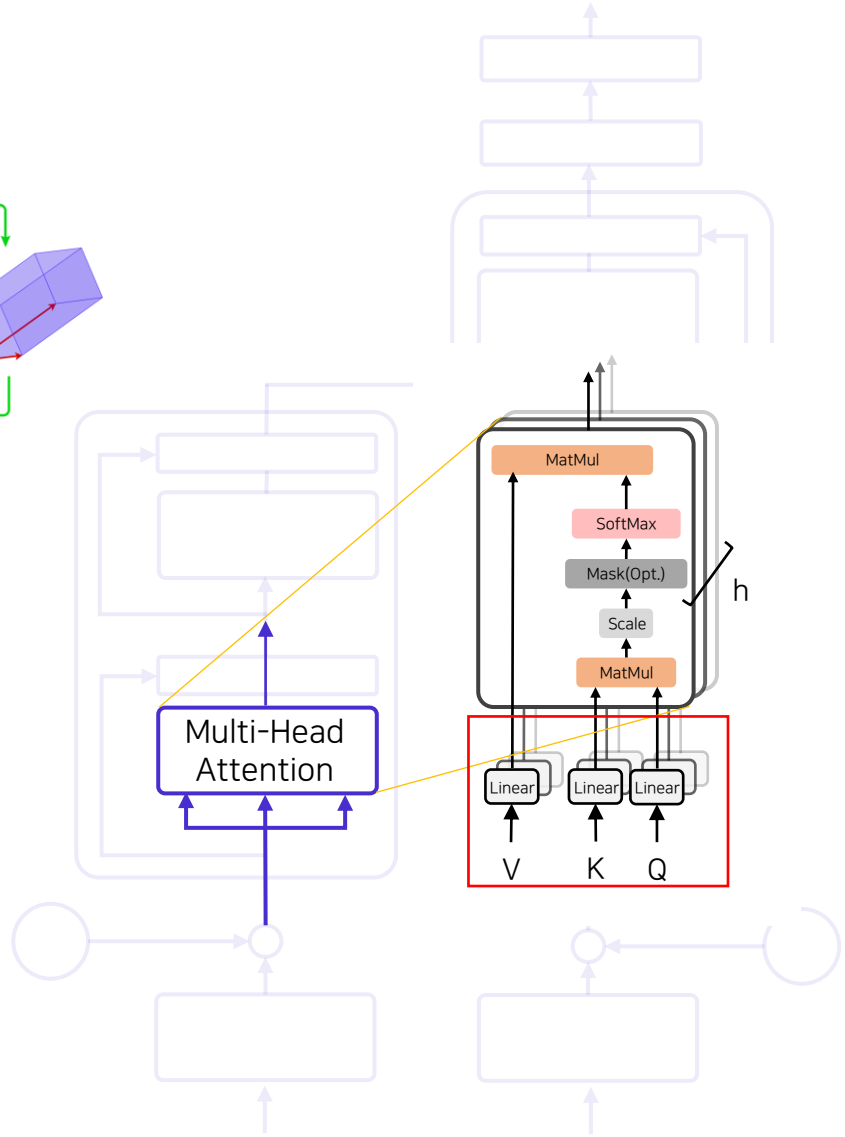
왜 Multi-Head를 해주는 지는 우선 제쳐두고 ㅎㅎ

(직관적인 개념은 여러 aspect를 본다고 이해하시면 될 것 같아요!)

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$





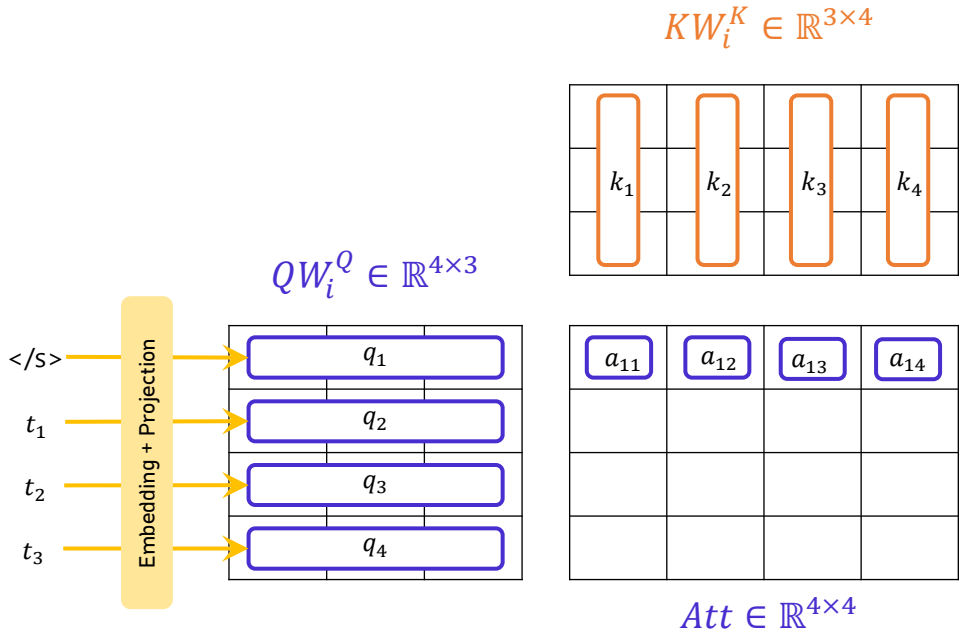


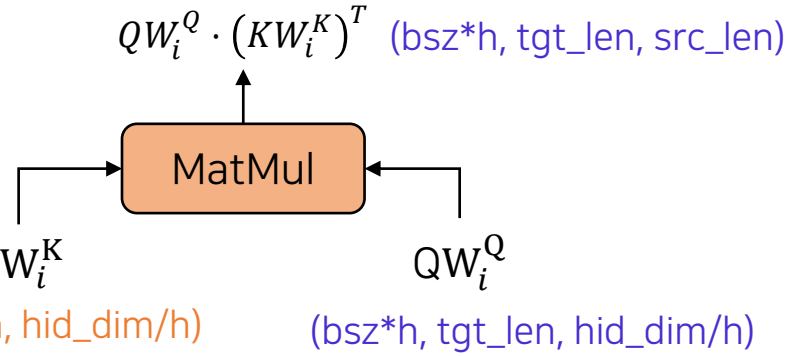
https://www.youtube.com/watch?v=1ocEn9LdkAI&t=1542s

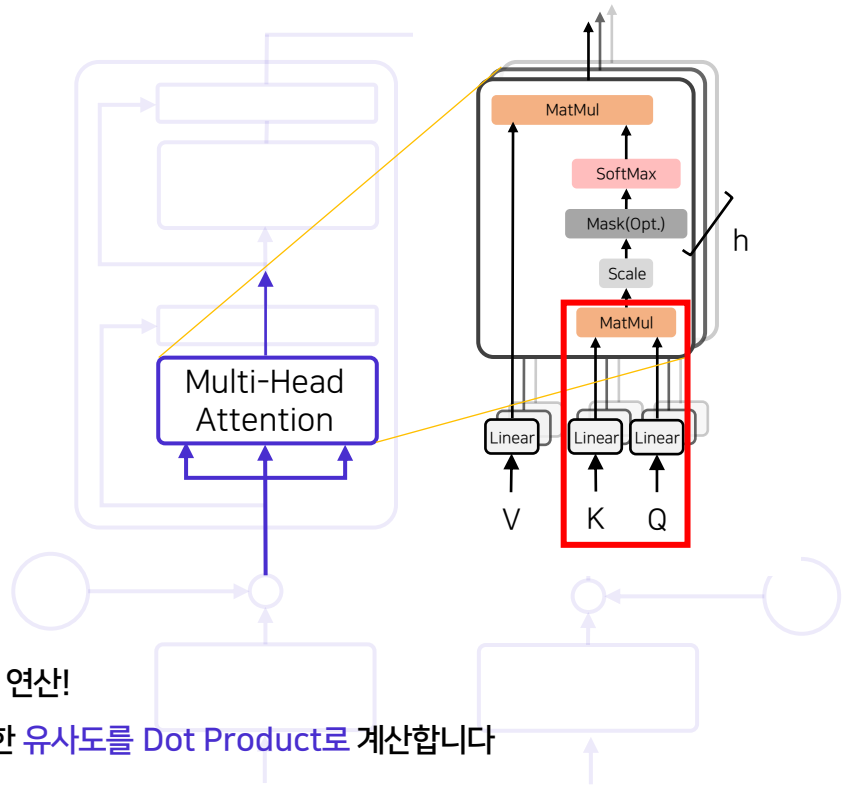
02. Attention is all you need!

Multi-Head Scale Dot Product Self-Attention

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$







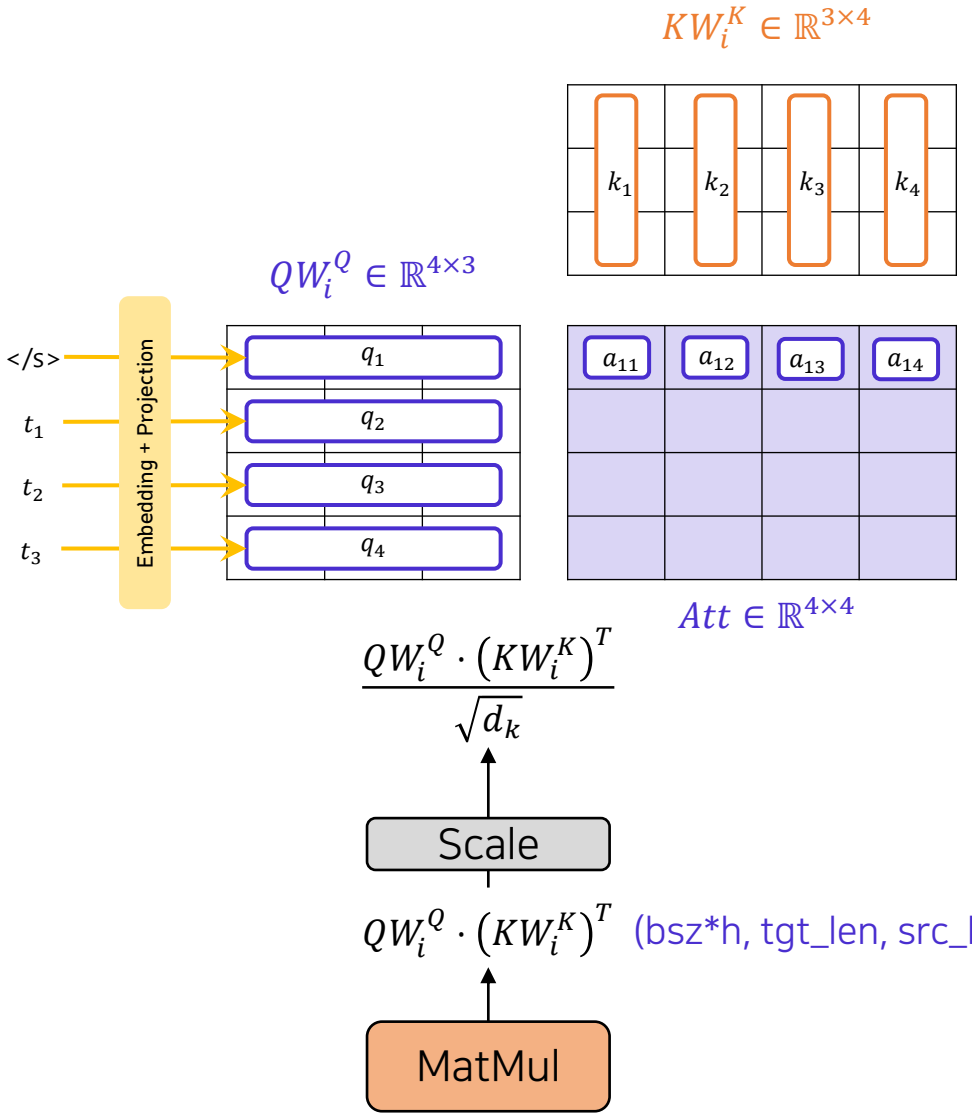
MHSA의 첫 번째 행렬 곱 연산!

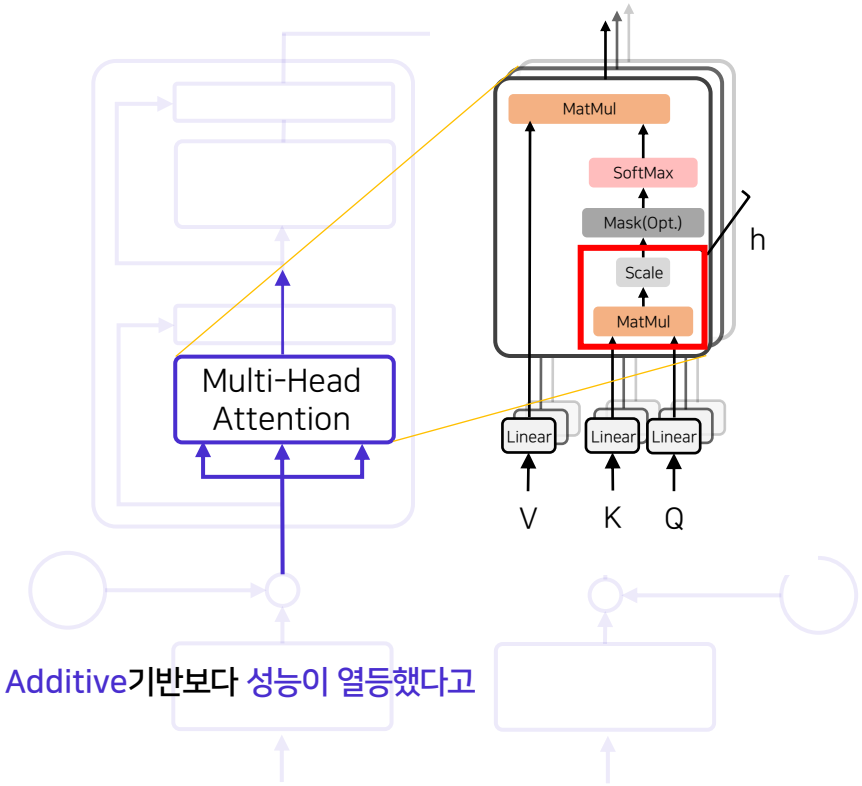
Query에 맞는 Key에 대한 유사도를 Dot Product로 계산합니다

02. Attention is all you need!

Multi-Head Scale Dot Product Self-Attention

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$





Scaling을 하지 않을 시 Additive기반보다 성능이 열등했다고 논문에서 보고합니다!

02. Attention is all you need!

왜 Scaling을 해주는 걸까요?

```python
import torch
import torch.nn as nn

# set arguments
batch_size = 32
source_length = 12
target_length = 12 # assume that self-attention
hidden_dim = 512
num_heads = 8
assert hidden_dim % num_heads == 0
head_dim = hidden_dim // num_heads

# get query, key, value state
encoder_hidden_state = torch.randn(batch_size, source_length, hidden_dim)
decoder_hidden_state = torch.randn(batch_size, target_length, hidden_dim)

Q = decoder_hidden_state
K = encoder_hidden_state
V = K.clone()

print(Q.shape, K.shape, V.shape)
```

```
torch.Size([32, 12, 512]) torch.Size([32, 12, 512]) torch.Size([32, 12, 512])
```

```python
# perform linear transformation on query, key, and value
q_proj = nn.Linear(hidden_dim, hidden_dim)
k_proj = nn.Linear(hidden_dim, hidden_dim)
v_proj = nn.Linear(hidden_dim, hidden_dim)
out_proj = nn.Linear(hidden_dim, hidden_dim)

QW = q_proj(Q)
KW = k_proj(K)
VW = v_proj(V)
print(QW.shape, KW.shape, VW.shape)
```

```
torch.Size([32, 12, 512]) torch.Size([32, 12, 512]) torch.Size([32, 12, 512])
```

```python
def _shape(tensor: torch.Tensor, seq_len: int, batch_size: int, num_heads: int):
    shape = (batch_size, seq_len, num_heads, -1)
    return tensor.view(*shape).transpose(1, 2).contiguous()
```

```python
# get multi-heads
from functools import partial

_multihead_shape = partial(_shape, batch_size=batch_size, num_heads=num_heads)

query_states = _multihead_shape(QW, seq_len=target_length)
key_states = _multihead_shape(KW, seq_len=source_length)
value_states = _multihead_shape(VW, seq_len=source_length)
print(query_states.shape, key_states.shape, value_states.shape)

proj_shape = (batch_size * num_heads, -1, head_dim)

query_states = query_states.view(*proj_shape)
key_states = key_states.view(*proj_shape)
value_states = value_states.view(*proj_shape)
print(query_states.shape, key_states.shape, value_states.shape)
```

```
torch.Size([32, 8, 12, 64]) torch.Size([32, 8, 12, 64]) torch.Size([32, 8, 12, 64])
torch.Size([256, 12, 64]) torch.Size([256, 12, 64]) torch.Size([256, 12, 64])
```

```python
attn_weights = torch.bmm(query_states, key_states.transpose(1, 2))
print(attn_weights.shape)
```

```
torch.Size([256, 12, 12])
```

```python
Q.var(), K.var(), attn_weights.var() # 64 ** 0.5 == 8.
```

```
(tensor(1.0049), tensor(1.0023), tensor(7.1726, grad_fn=<VarBackward0>))
```

02. Attention is all you need!

왜 Scaling을 해주는 걸까요?

```python
import torch
import torch.nn as nn

# set arguments
batch_size = 32
source_length = 12
target_length = 12 # assume that self-attention
hidden_dim = 512
num_heads = 8
assert hidden_dim % num_heads == 0
head_dim = hidden_dim // num_heads

# get query, key, value state
encoder_hidden_state = torch.randn(batch_size, source_length, hidden_dim)
decoder_hidden_state = torch.randn(batch_size, target_length, hidden_dim)

Q = decoder_hidden_state
K = encoder_hidden_state
V = K.clone()

print(Q.shape, K.shape, V.shape)
```

```
torch.Size([32, 12, 512]) torch.Size([32, 12, 512]) torch.Size([32, 12, 512])
```

```python
# perform linear transformation on query, key, and value
q_proj = nn.Linear(hidden_dim, hidden_dim)
k_proj = nn.Linear(hidden_dim, hidden_dim)
v_proj = nn.Linear(hidden_dim, hidden_dim)
out_proj = nn.Linear(hidden_dim, hidden_dim)

QW = q_proj(Q)
KW = k_proj(K)
VW = v_proj(V)
print(QW.shape, KW.shape, VW.shape)
```

```
torch.Size([32, 12, 512]) torch.Size([32, 12, 512]) torch.Size([32, 12, 512])
```

```python
# do scaling
QW *= num_heads ** 0.5
```

```python
def _shape(tensor: torch.Tensor, seq_len: int, batch_size: int, num_heads: int):
    shape = (batch_size, seq_len, num_heads, -1)
    return tensor.view(*shape).transpose(1, 2).contiguous()
```

```python
# get multi-heads
from functools import partial

_multihead_shape = partial(_shape, batch_size=batch_size, num_heads=num_heads)

query_states = _multihead_shape(QW, seq_len=target_length)
key_states = _multihead_shape(KW, seq_len=source_length)
value_states = _multihead_shape(VW, seq_len=source_length)
print(query_states.shape, key_states.shape, value_states.shape)

proj_shape = (batch_size * num_heads, -1, head_dim)

query_states = query_states.view(*proj_shape)
key_states = key_states.view(*proj_shape)
value_states = value_states.view(*proj_shape)
print(query_states.shape, key_states.shape, value_states.shape)
```

```
torch.Size([32, 8, 12, 64]) torch.Size([32, 8, 12, 64]) torch.Size([32, 8, 12, 64])
torch.Size([256, 12, 64]) torch.Size([256, 12, 64]) torch.Size([256, 12, 64])
```

```python
attn_weights = torch.bmm(query_states, key_states.transpose(1, 2))
print(attn_weights.shape)
```

```
torch.Size([256, 12, 12])
```

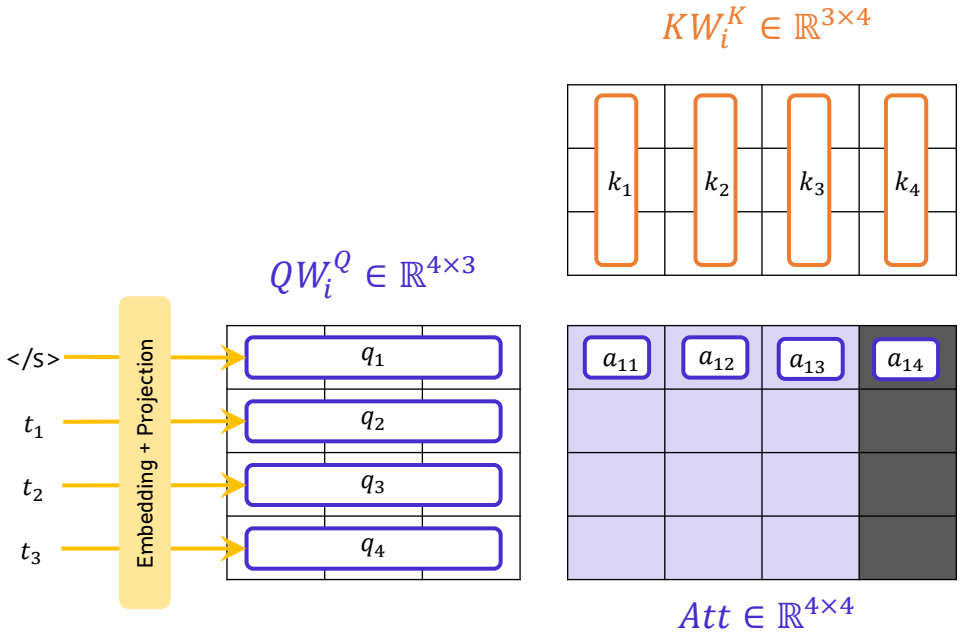
```python
Q.var(), K.var(), attn_weights.var() # 64 ** 0.5 == 8.
```

```
(tensor(1.0049), tensor(1.0023), tensor(0.8966, grad_fn=<VarBackward0>))
```

02. Attention is all you need!

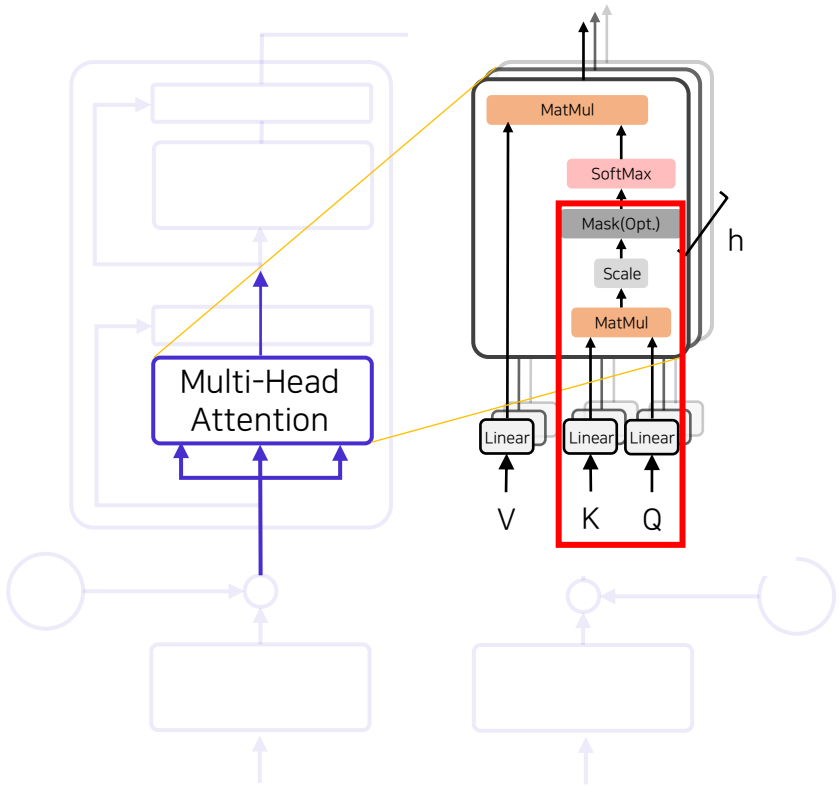
Multi-Head Scale Dot Product Self-Attention

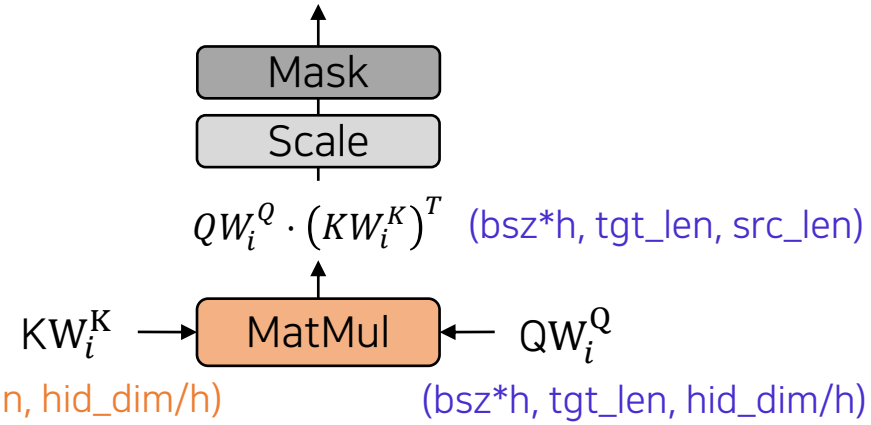
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$



Decoder의 Masked Self-Attention만 masking해주는 건 아니에요!

Encoder에서도 <PAD> 토큰 연산을 하지 않기 위해 마스킹해줍니다!

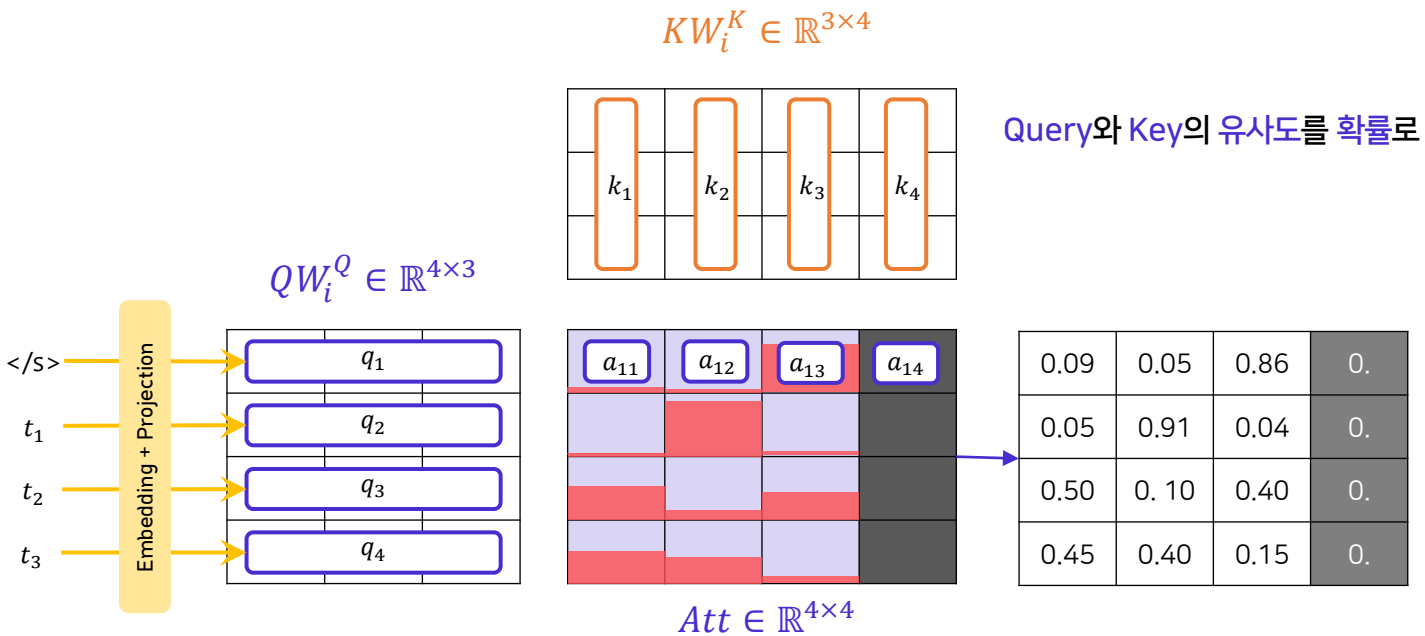




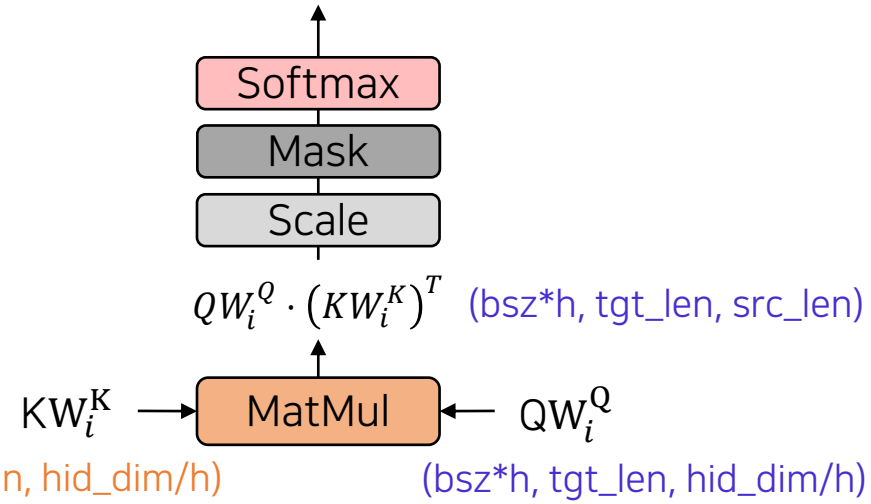
02. Attention is all you need!

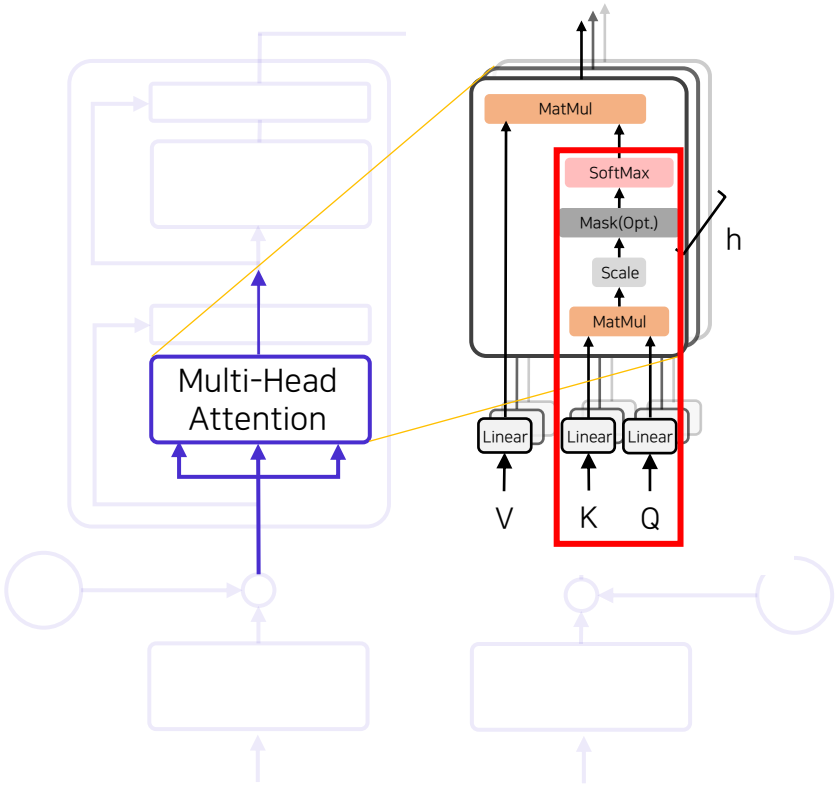
Multi-Head Scale Dot Product Self-Attention

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$



Query와 Key의 유사도를 확률로 만들어줍니다!

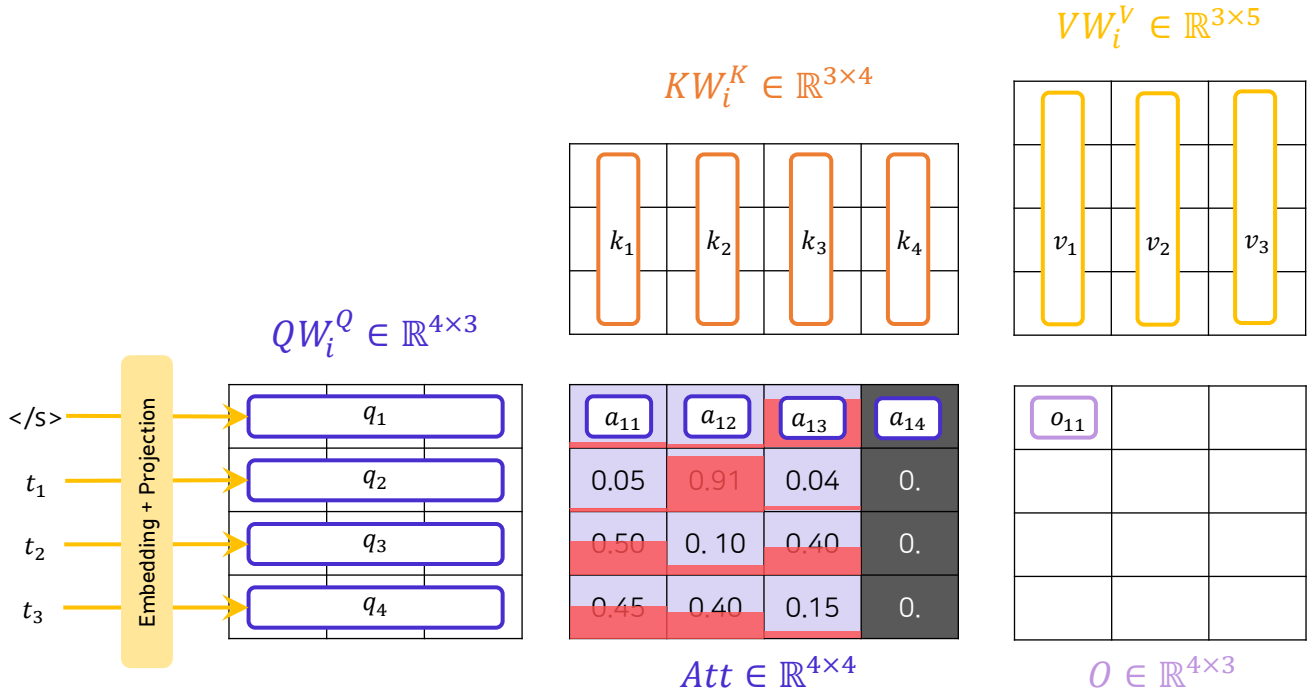


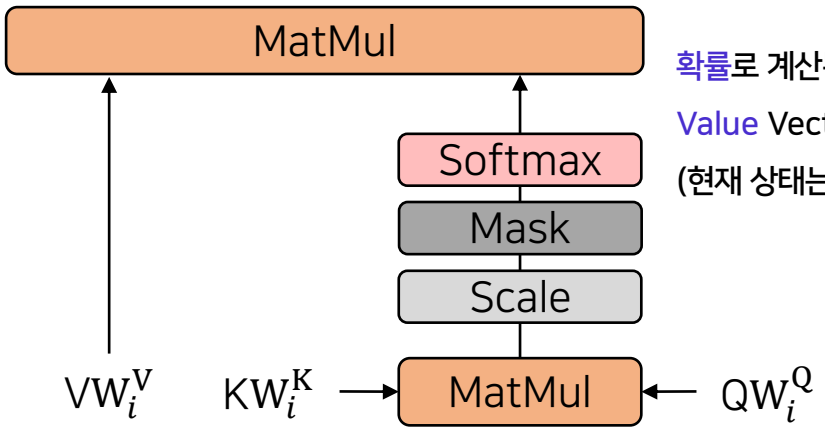


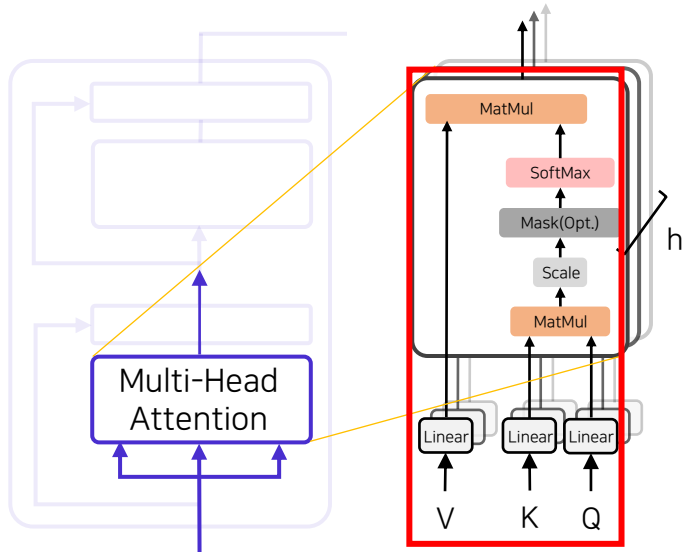
02. Attention is all you need!

Multi-Head Scale Dot Product Self-Attention

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right) VW_i^V$$







확률로 계산된 Query와 Key의 유사도를

Value Vector에 곱해줘서 최종 Hidden State를 반환받습니다!

(현재 상태는 layer별, Multi Head 별, batch 별 딱 한 파트의 연산을 수행한 상태에요!)

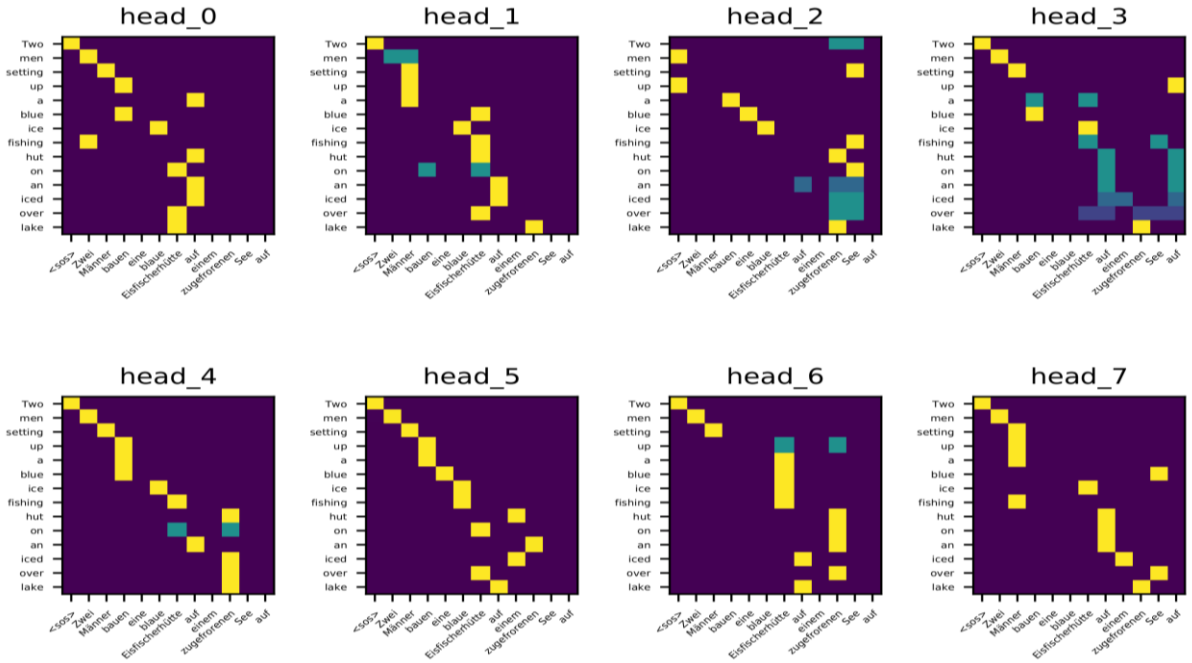
02. Attention is all you need!

왜 여러 가지의 Head를 만들어 주는 걸까요? 왜 쪼갠 다음 합쳐서 out project로 정리해줄까요?

여러 가지의 다양한 표현을 볼 수 있기 때문에!

앙상블로도 생각해 볼 수 있어요 ㅎㅎ

그런데, Attention의 Multi-Head들이 서로 다른 부분을 참조한다는 직관은 틀렸을 수 있어요!



The original multi-head attention was defined as:

$$\text{MultiHead}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right)\mathbf{W}^O$$

$$\text{where head}_i = \text{Attention}\left(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\right)$$

Basically, the initial embedding dimension dim is **decomposed** to $h \times d_{head}$, and the computation per head is carried out independently.

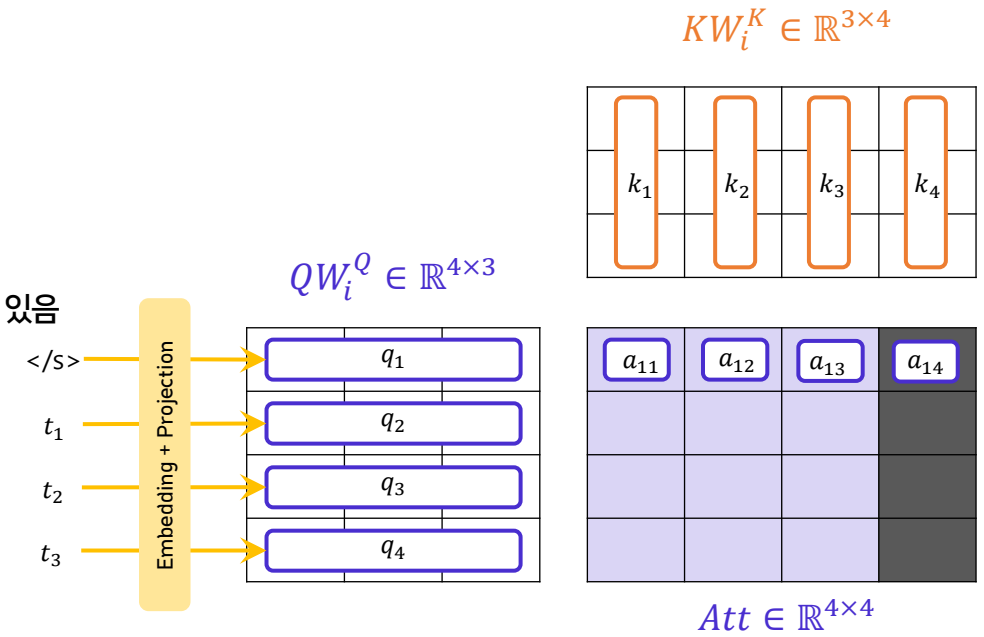
https://docs.dgl.ai/en/0.4.x/tutorials/models/4_old_wines/7_transformer.html

02. Attention is all you need!

Multi-Head Attention에 대하여 (Further Reading 개념! 논문의 가설이라는 점 유의해주세요!)

Insight 0: self-attention은 symmetric하지 않음

- 같은 input repr를 사용하는 경향이 있기 때문에, self-attention이 symmetric하다는 함정에 걸려들지 말기.
- Self-attention이 symmetric이 되기 위해선, $W_Q = W_K$여야 함. (같은 projection matrix)
- 왜? 행렬을 Transpose와 곱하면 대칭 행렬이 생성되기 때문. 그러나 결과 행렬의 rank는 증가하지 않음
- 이에 영감을 받아 key, query에 대해 두 개가 아닌 하나의 shared projection matrix를 사용하는 많은 논문이 있음

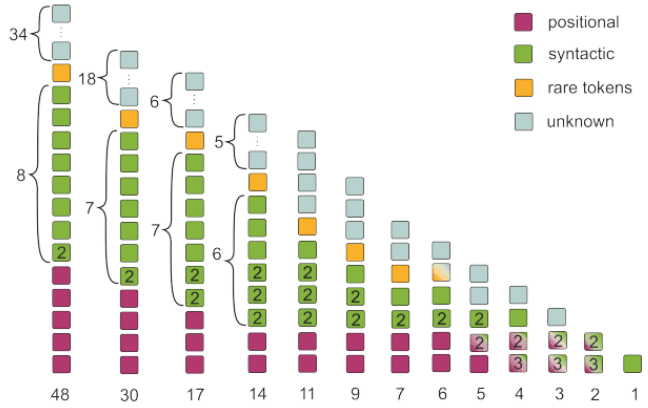


Insight 1: MHSA의 각 head는 해당 state의 subspace 뿐만 아니라 거의 전체 내용을 보존함!

- https://arxiv.org/abs/1910.06611
- 우리의 직관과 실제로 학습되는 과정이 다를 수 있다는 것을 보여준 논문

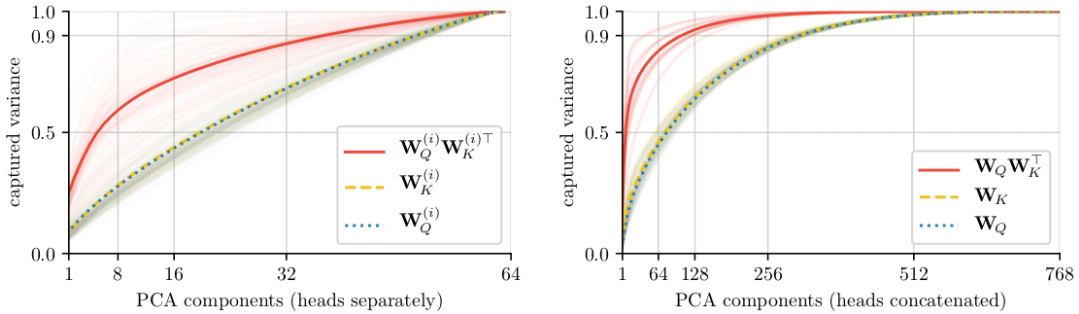
Insight 2: Cross-Attention에서 Decoder의 첫 번째 layer는 LM을 설명, 마지막 Layer는 source 문장에 대한 정보를 요약!

- https://arxiv.org/abs/1905.09418
- 문장의 각 부분을 본다고 서로 다른 subspace를 학습할 거라고 생각하는 것은 틀릴지도 모름!
- 그런데 attention map을 보면 표현이 다르지 않는가? 전체 정보는 보존한다 치고 어떻게 학습하는가?
- Attention Map에서 세 가지 유형의 중요한 유형을 판별
  - Positional Heads: 주변에 집중하는
  - Syntactic Heads: 특수한 문법적인 관계를 가지는 Token을 가리키는
  - Rare Heads: 드물게 등장하는 Token을 가리키는
- 전체 공간의 정보를 가지면서, 어떤 정보에 가중을 더 줄지가 정확한 표현인 듯



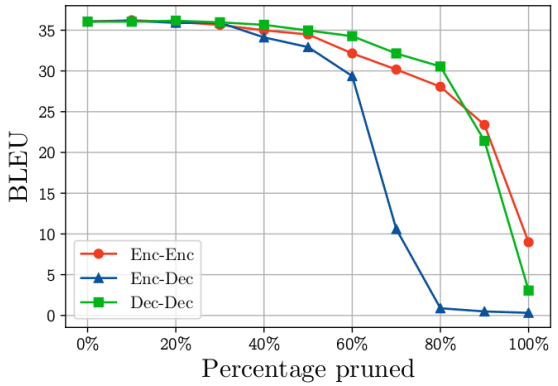
02. Attention is all you need!

Multi-Head Attention에 대하여 (Further Reading 개념! 논문의 가설이라는 점 유의해주세요!)



Insight 3: Multi-Head를 합친 이후의 Attention Matrix는 Low-Rank임

- https://arxiv.org/abs/2006.16362
- 왜 Concat하면 rank가 줄어들까?
- Rank가 줄어든다는 것은 서로 종속인 (다른 독립인 column vector들의 선형 결합으로 표현할 수 있는) 벡터들이 늘어난다는 것
- 즉, 공통된 표현을 배운다는 얘기
- 역설적으로 independently하게 학습되는 각 Head들은 서로 공통된 subspace를 학습한다고 논문에서는 주장



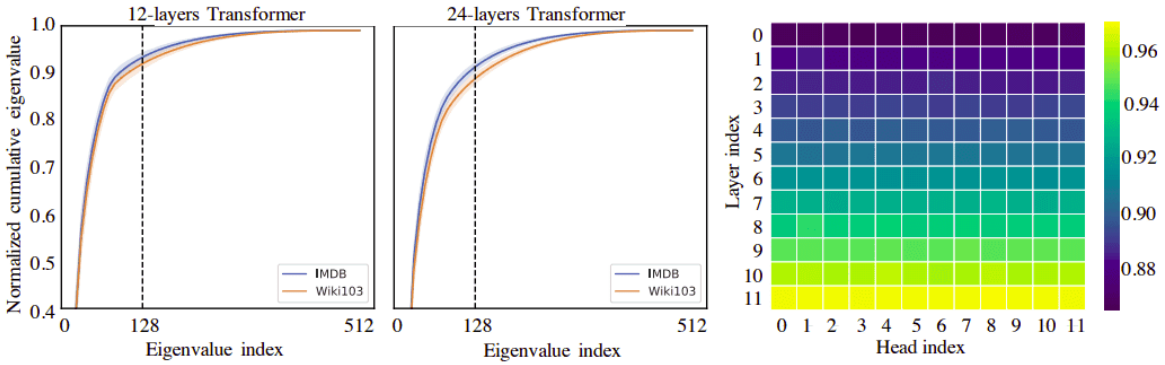
Insight 4: Cross-Attention이 Multi-Head에 더욱 민감함

- https://arxiv.org/abs/1905.10650
- 번역 등에서 Alignment를 어떻게 하느냐가 중요했음, Multi-Head는 그러한 연결 고리 역할을 해주는 것 같다고 보고

Insight 5: Softmax를 취한 이 후의 Attention Matrix는 Low-Rank임. (이를 토대로 linformer 제안)

- https://arxiv.org/abs/2006.04768
- 실제로 KLUE-Roberta-Large로 실험해보니 오른쪽처럼 결과가 나왔음
- 즉, P의 domestic한 eigenvalue들로 Self-Attention Matrix를 복원할 수 있음

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \underbrace{\text{softmax}\left[\frac{QW_i^Q(KW_i^K)^T}{\sqrt{d_k}}\right]}_{P} VW_i^V,$$



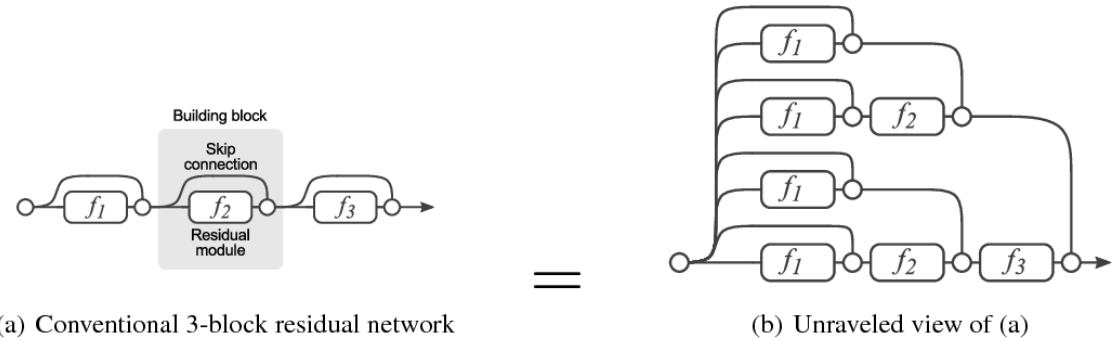
02. Attention is all you need!

Add & Layer Normalization

수업 시간에 배운 Residual Connection! y = x + F(x)

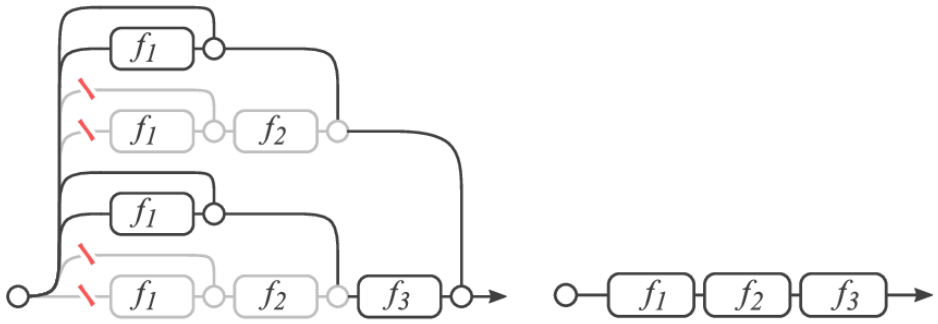
왜 해주는 걸까? (가설)

- 레이어 f1과 f2가 망가져도 f3으로 정상적인 gradient가 흐름!
- 예를 들어, Self-Attention Matrix에 Rank Collapse가 발생하더라도 원래의 rank를 가진 입력 tensor를 더해줌으로써 rank를 보존하는 효과가 있음



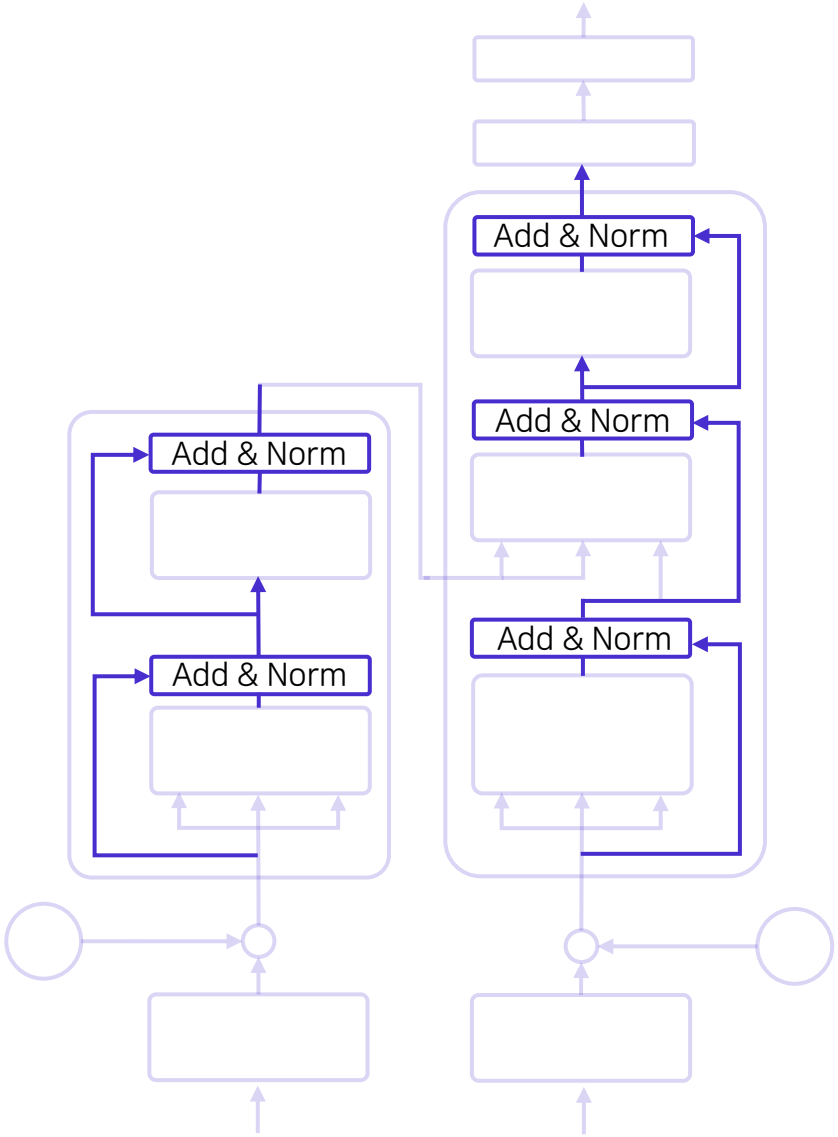
(a) Conventional 3-block residual network

(b) Unraveled view of (a)



(a) Deleting f_2 from unraveled view

(b) Ordinary feedforward network



https://arxiv.org/abs/1605.06431
https://arxiv.org/abs/1605.06431

02. Attention is all you need!

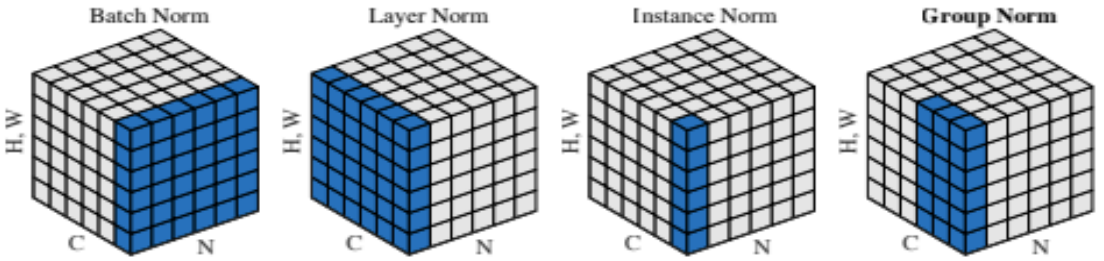
Add & Layer Normalization

Batch Normalization과 동일함! 단지 적용되는 dimension이 다를 뿐

분포를 안정적으로 만들어주고 학습을 빠르게 만들어 줌

Image와 달리 자연어는 input sequence의 길이가 매번 다름

때문에 BatchNorm보단 LayerNorm이 적합함

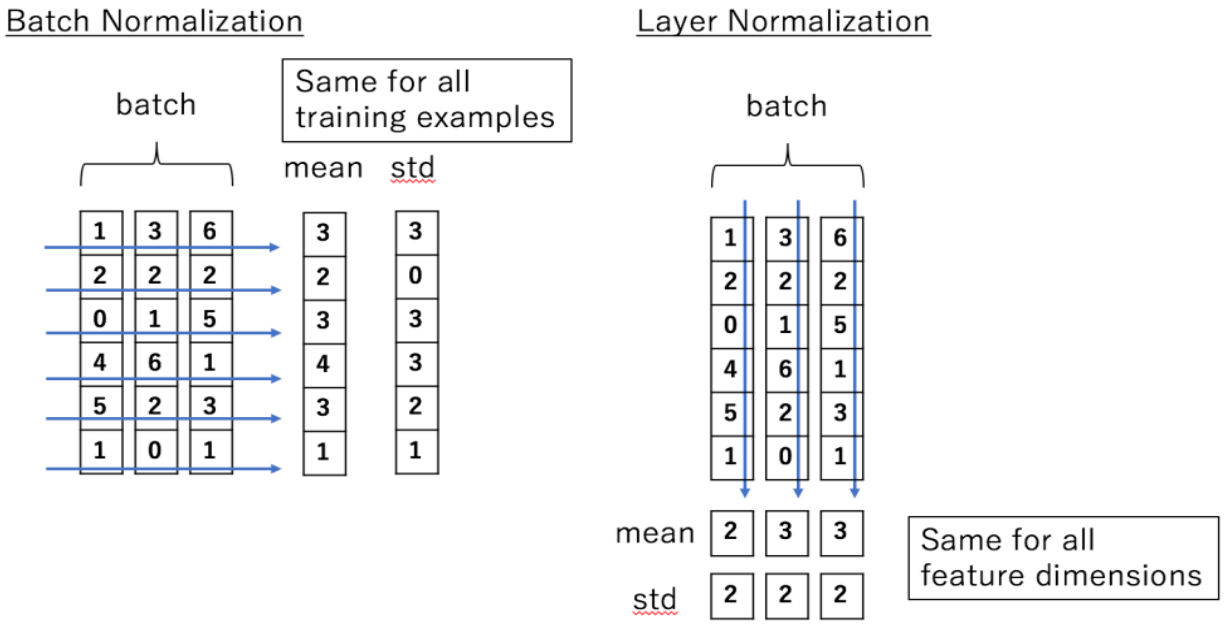


(1, 6, 4)

Batch_normalization

(1, 6, 4)

layer_normalization



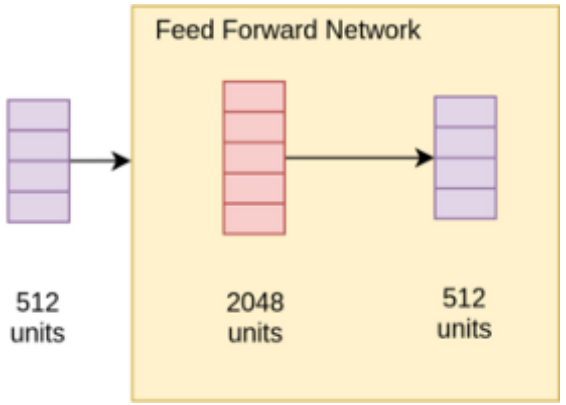
https://arxiv.org/abs/1607.06450

02. Attention is all you need!

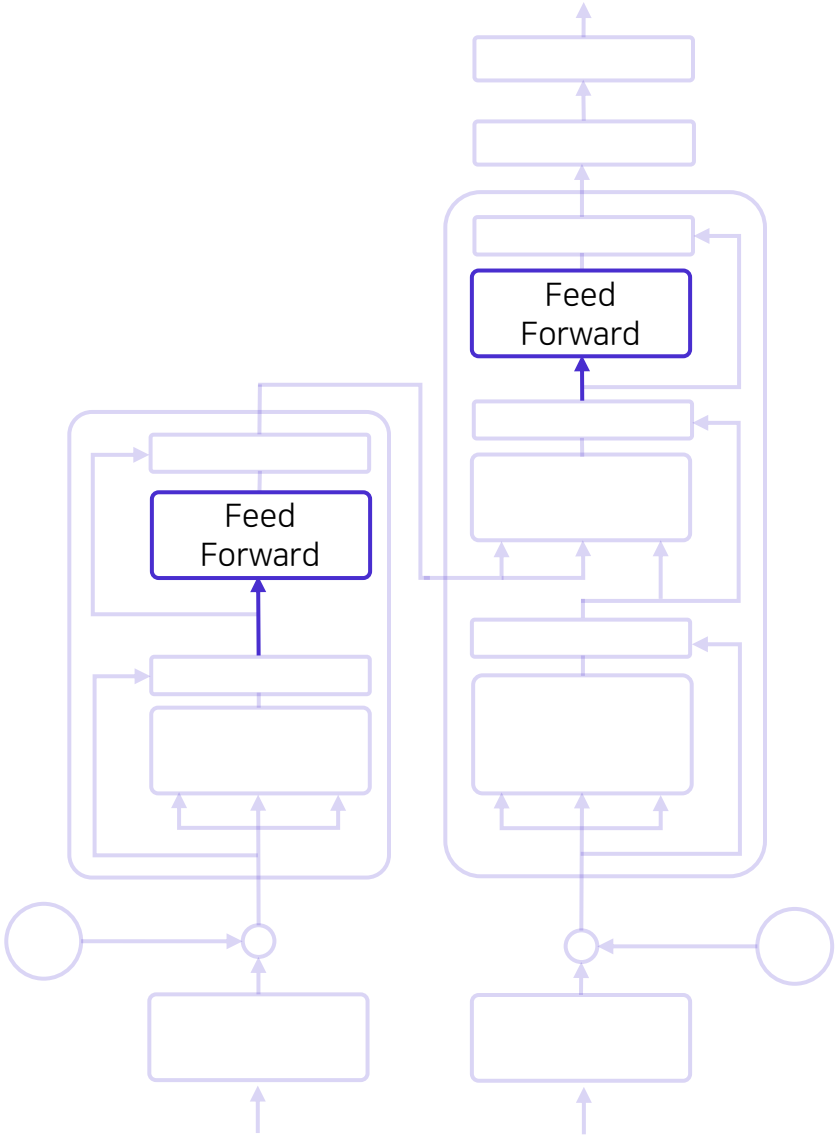
Feed Forward Network

Self-Attention Module의 결과값을 정리해준다. (직관)

근데 왜 4배로 늘려주는걸까? 이유가 있을까?




```
residual = hidden_states
hidden_states = self.activation_fn(self.fc1(hidden_states))
hidden_states = nn.functional.dropout(hidden_states, p=self.activation_dropout, training=self.training)
hidden_states = self.fc2(hidden_states)
hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
hidden_states = residual + hidden_states
hidden_states = self.final_layer_norm(hidden_states)
```



02. Attention is all you need!

Attention is All you Need라면서 왜 LayerNorm과 FFN을 넣어줬을까?

Insight 7: MLP, Skip-Connection이 없으면 Attention Matrix가 기하급수적으로 1-rank matrix로 수렴

- https://arxiv.org/abs/2103.03404
- 위의 현상을 Rank Collapse라고 부름!
- Skip-Connection 중요!
- Feed-Forward-Network도 진짜 중요: 768->3072->768
- Layer Normalization은 Rank Collapse엔 도움이 안 됨
- What is rank collapse? https://calculatedcontent.com/2018/09/21/rank-collapse-in-deep-learning/
- 재밌는 논문 제목, Attention is not all you need!

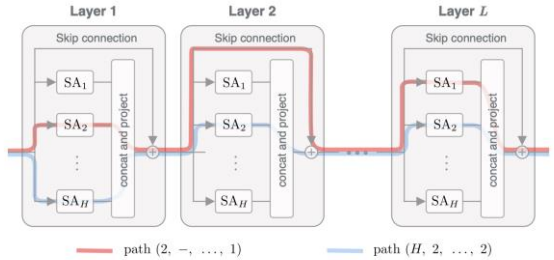


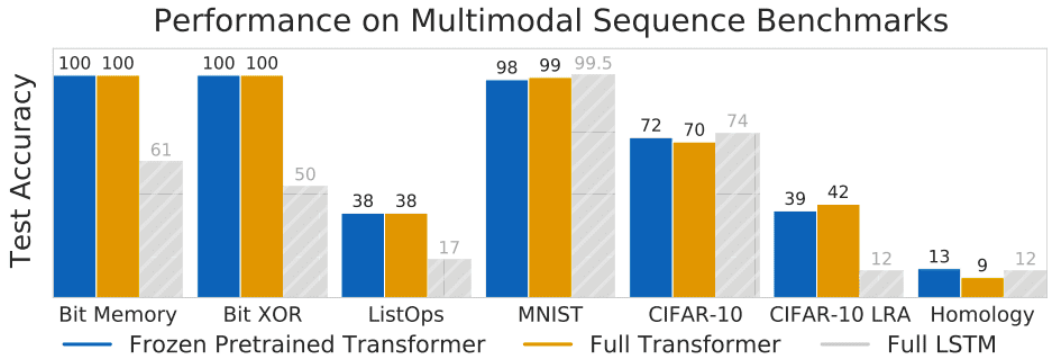
Figure 1: Two paths in a deep Self-Attention Network (SAN) with H heads and L layers. At each layer, a path can go through one of the heads or bypass the layer. Adding an MLP block after each attention layer forms the transformer architecture.

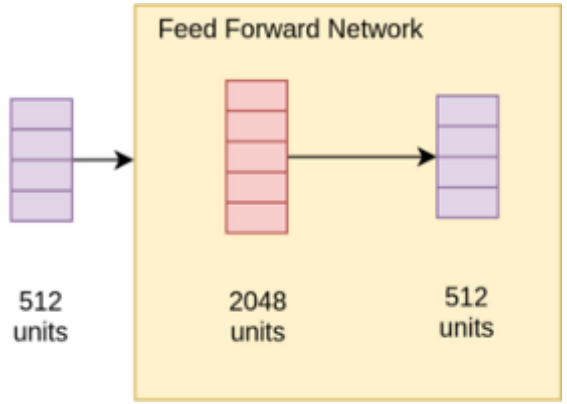
Claim 3.1. *Consider a depth-L and width-H self-attention network with skip connections. There exist infinitely many parameterizations for which $\|res(\boldsymbol{X}^L)\| \geq \|res(\boldsymbol{X})\|$. The preceeding holds even for $L \to \infty$ and β arbitrarily small.*

$$\|res(\boldsymbol{X}^L)\|_{1,\infty} \leq \left(\frac{4\,\beta\,H\,\lambda}{\sqrt{d_{qk}}}\right)^{\frac{3^L-1}{2}} \|res(\boldsymbol{X})\|_{1,\infty}^{3^L},$$

Insight 8: LayerNorm은 PLM transfer learning의 key ingredient!

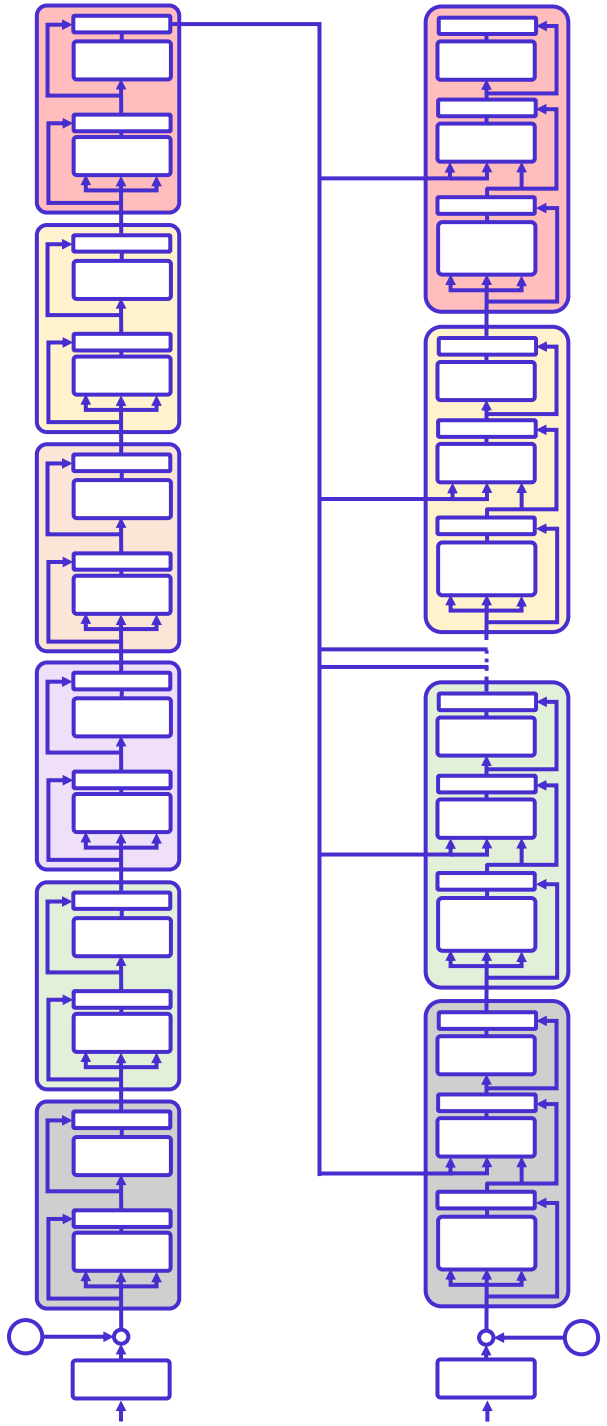
- https://arxiv.org/pdf/2103.05247.pdf
- Fine-Tune할 때 LayerNorm 부분이 성능에 귀결되는 중요한 부분이라고 함!





02. Attention is all you need!

last_hidden_state



Stacked Encoder

총 6개의 layer를 쌓았음 (In transformer, Decoder도 6층 쌓음)

Encoder의 최종 출력 값 (last_hidden_state)가 Decoder 각 layer의 key, value로 넘어감 (For Cross-Attention)

```python
class BartEncoder(BartPretrainedModel):
    """
    Transformer encoder consisting of *config.encoder_layers* self attention layers. Each layer is a
    :class:`BartEncoderLayer`.

    Args:
        config: BartConfig
        embed_tokens (nn.Embedding): output embedding
    """

    def __init__(self, config: BartConfig, embed_tokens: Optional[nn.Embedding] = None):
        super().__init__(config)

        self.dropout = config.dropout
        self.layerdrop = config.encoder_layerdrop

        embed_dim = config.d_model
        self.padding_idx = config.pad_token_id
        self.max_source_positions = config.max_position_embeddings
        self.embed_scale = math.sqrt(embed_dim) if config.scale_embedding else 1.0

        if embed_tokens is not None:
            self.embed_tokens = embed_tokens
        else:
            self.embed_tokens = nn.Embedding(config.vocab_size, embed_dim, self.padding_idx)

        self.embed_positions = BartLearnedPositionalEmbedding(
            config.max_position_embeddings,
            embed_dim,
        )
        self.layers = nn.ModuleList([BartEncoderLayer(config) for _ in range(config.encoder_layers)])
        self.layernorm_embedding = nn.LayerNorm(embed_dim)

        self.init_weights()
```

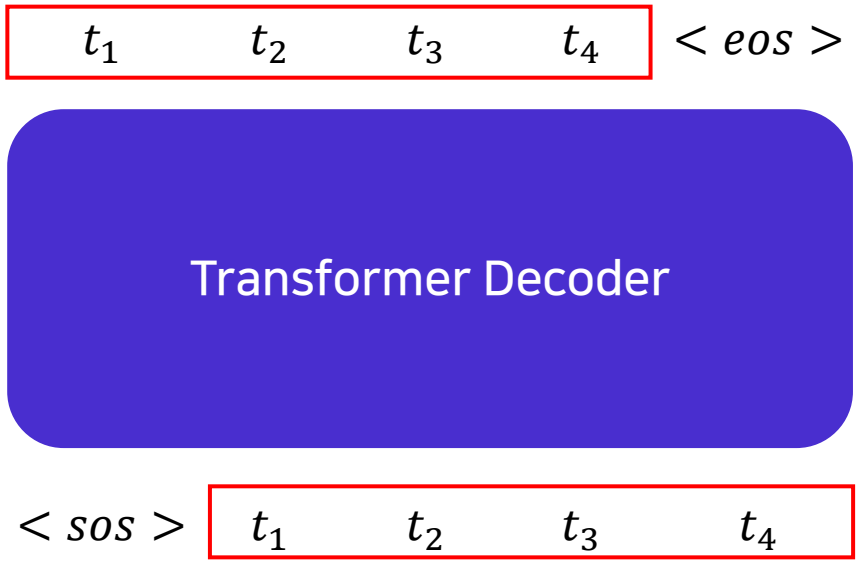
갑자기 BART? 라고 하실 수 있는데
BART는 Seq2Seq 구조로 Transformer와 동일합니다!
(조금 다른 부분이 있긴 해요! 금요일에 나눠봐요 ㅎㅎ)

02. Attention is all you need!

Decoder Input (Output Embedding)

- Transformer의 Decoder는 Autoregressive Model
- 실제 label = [“<sos>”, “t1”, “t2”, “t3”, “t4”, “<eos>”]이라고 가정
- 그러면 input tensor로는 label[:-1]이 들어가게 됨
- Decoder는 input tokens를 기반으로 다음 token을 생성하게 됨
- 생성된 token을 [“~t1”, “~t2”, “~t3”, “~t4”, “<eos>”]라고 하자.
- 이는 기존 label로 치면 label[1:]이 될 것이다.
- Loss함수는 거의 대부분 CE Loss를 사용하며 아래와 같이 정의된다.
- loss_fct = nn.CrossEntropyLoss(ignore_index=pad_token_id)
- Then, loss_fct(logits[1:], label[:-1])로 loss를 계산하여 역전파로 학습하게 된다.
- 위의 logit은 어떻게 계산되는걸까? Encoder에서 넘어온 encoder_hidden_states와
- Input embedding을 Decoder Layer에 태우고
- 최종으로 나온 decoder_hidden_state가 logits가 된다.

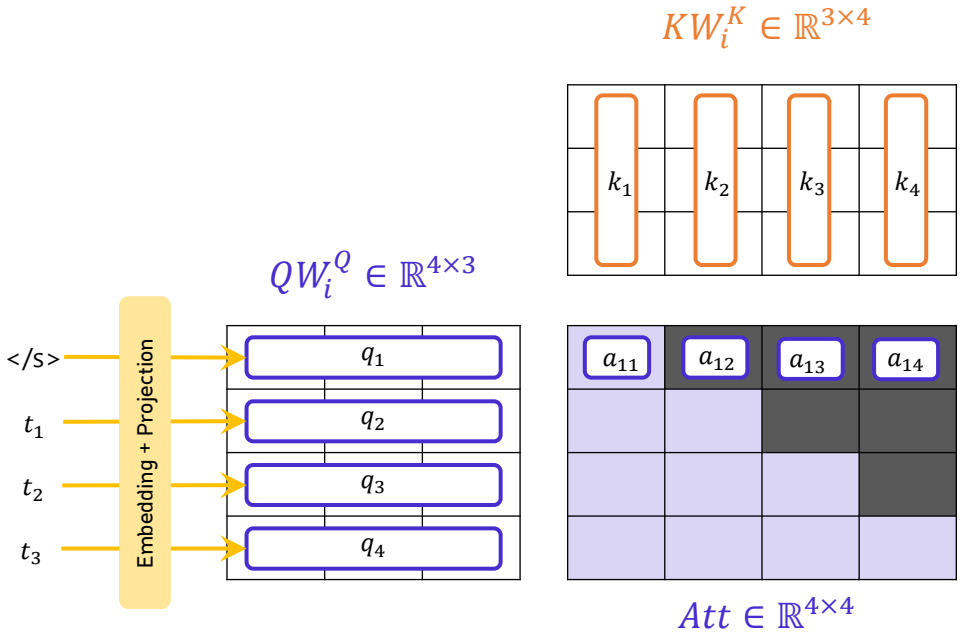
```
def shift_tokens_right(input_ids: torch.Tensor, pad_token_id: int, decoder_start_token_id: int):
    """
    Shift input ids one token to the right.
    """
    shifted_input_ids = input_ids.new_zeros(input_ids.shape)
    shifted_input_ids[:, 1:] = input_ids[:, :-1].clone()
    shifted_input_ids[:, 0] = decoder_start_token_id

    assert pad_token_id is not None, "self.model.config.pad_token_id has to be defined."
    # replace possible -100 values in labels by `pad_token_id`
    shifted_input_ids.masked_fill_(shifted_input_ids == -100, pad_token_id)

    return shifted_input_ids
```

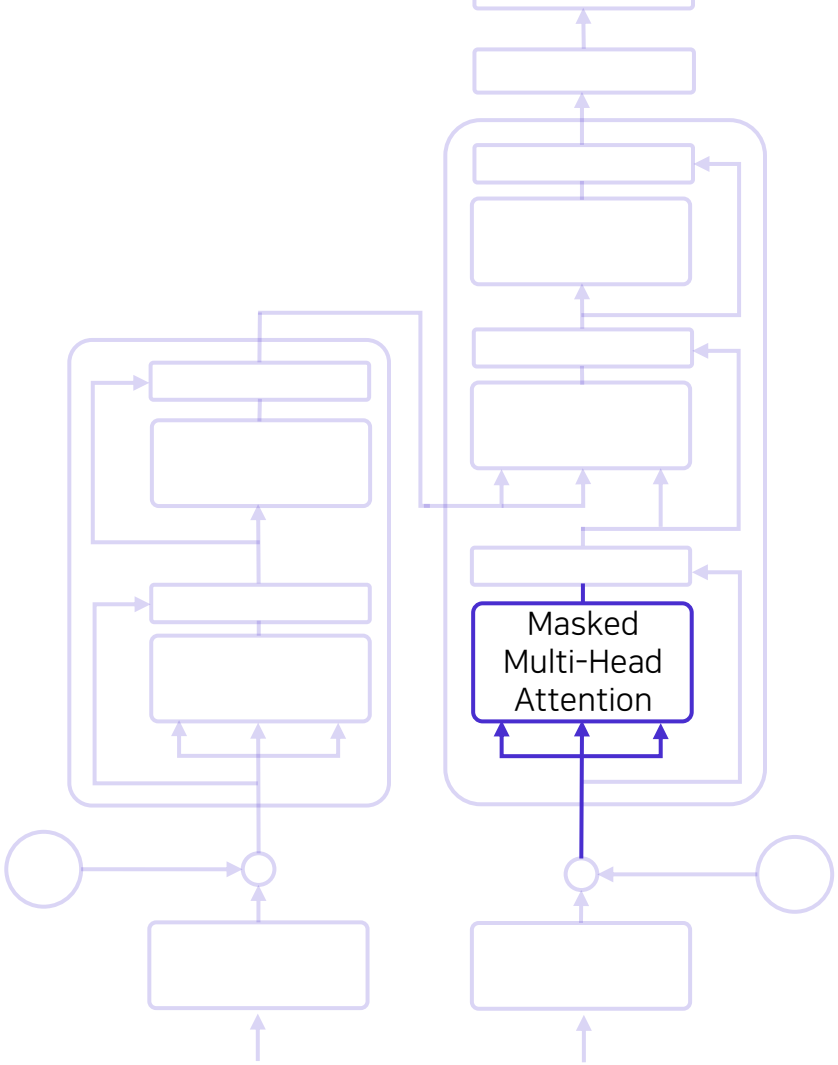


02. Attention is all you need!

Masked Self-Attention (=Causal Attention)

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) = \text{softmax}\left(\frac{QW_i^Q \cdot (KW_i^K)^T}{\sqrt{d_k}}\right)VW_i^V$$



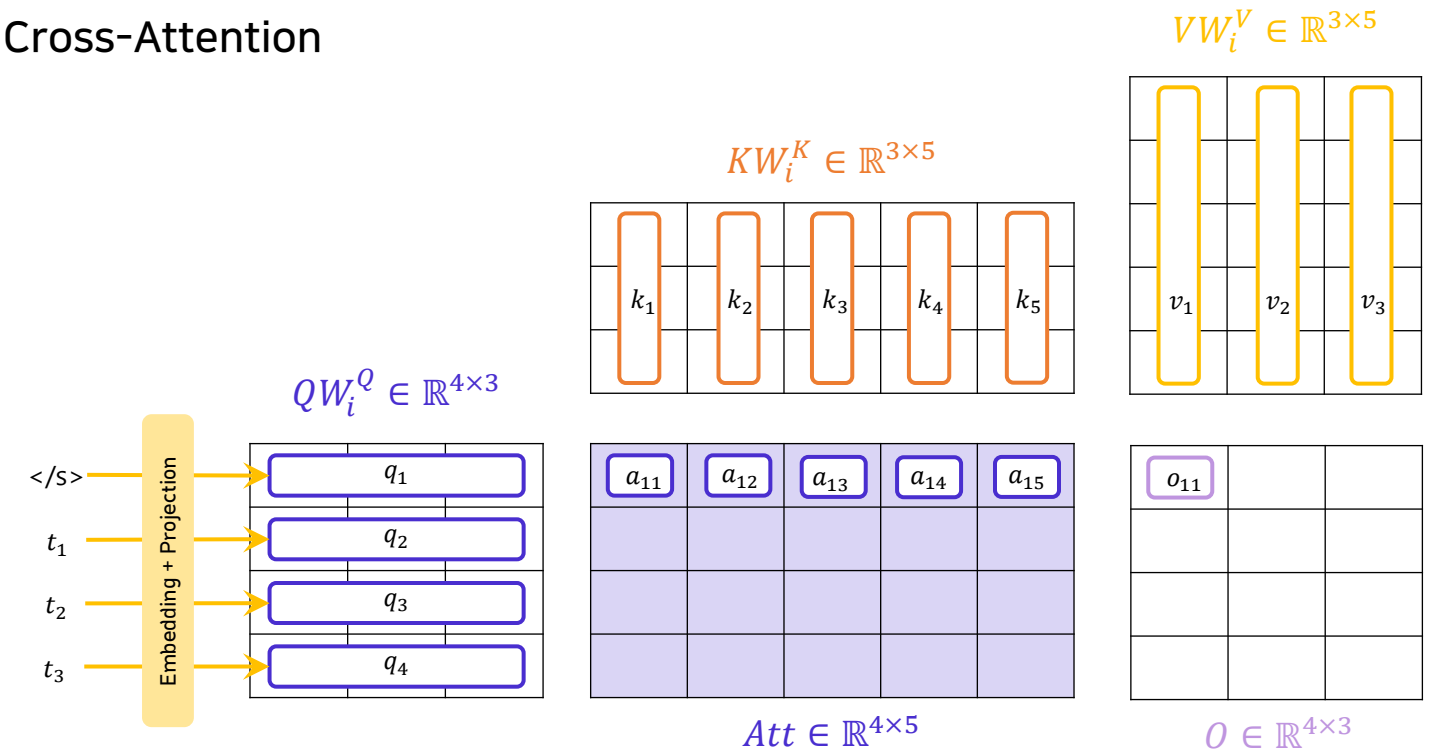


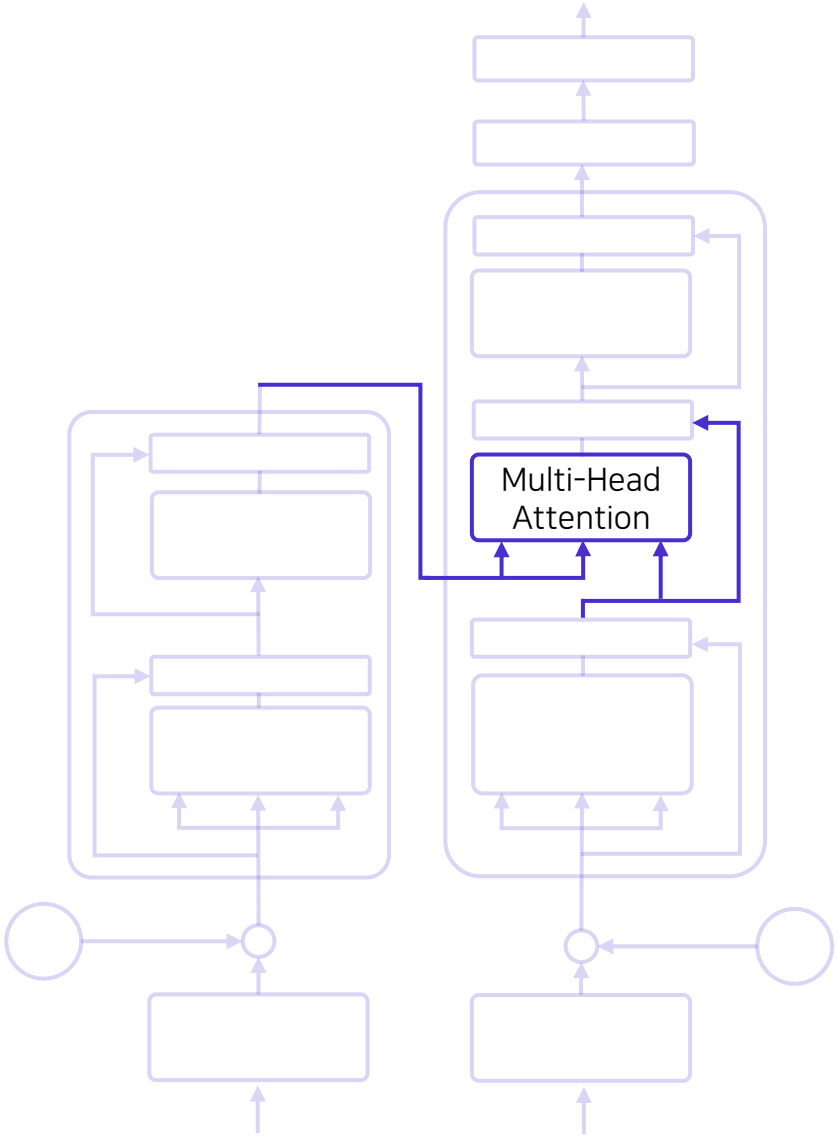
기존의 Self-Attention과 동일합니다.

다른 점이 있다면 Cheating을 방지하기 위해 해당 타임 스텝 이 후에 있는 t는 attention score 를 계산하지 못하도록 masking하는 것!

02. Attention is all you need!

Cross-Attention

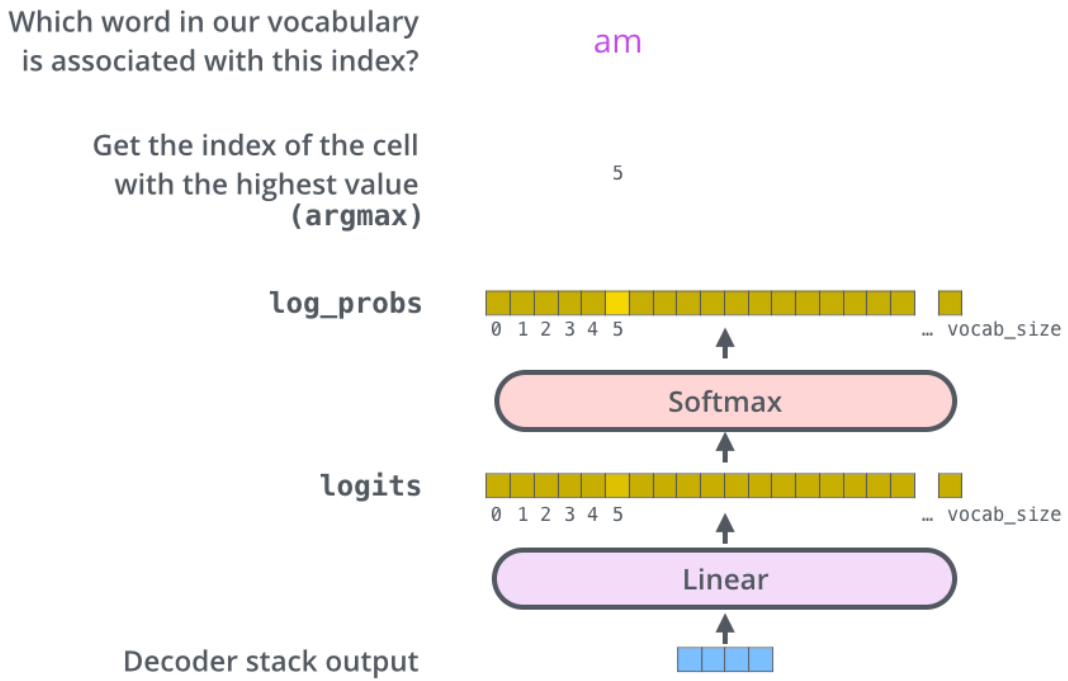


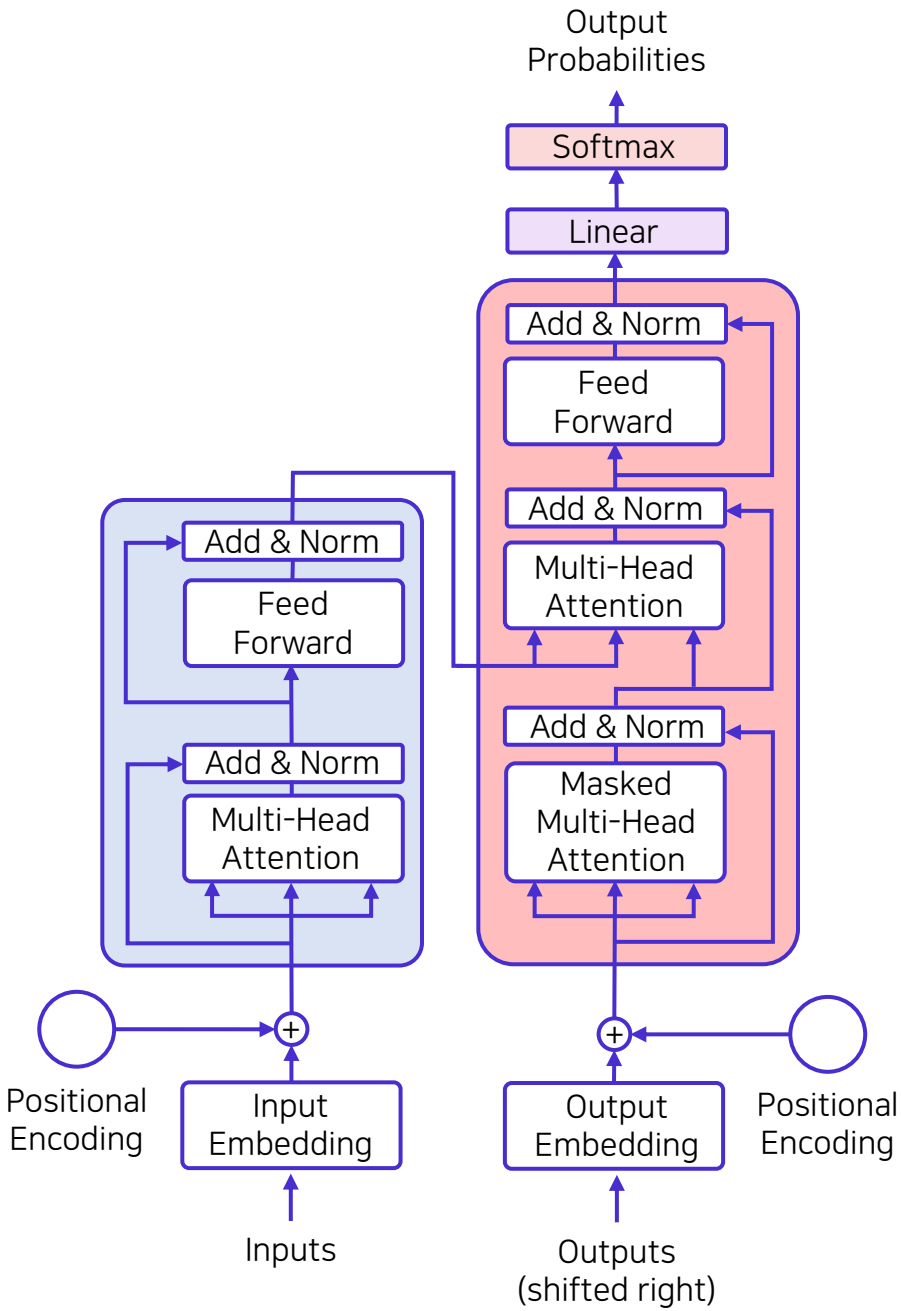


- Bahdanau Attention에서 학습한 Attention Mechanism과 동일
- 번역 등의 task에서 Cross-Attention에 Multi-Head가 중요한 역할을 한다는 사실은 앞에서 다룸

02. Attention is all you need!

최종적인 Architecture!





02. Attention is all you need!

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

02. Attention is all you need!

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 [11]. 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 [31] and byte-pair [25] 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 [15], 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.

왜 Self-Attention을 사용하는가?

1. Total Complexity per layer
2. Minimum number of sequential operations required
3. Long-range dependencies path length

n < d 일 때 recurrence보다 빠름 (바꿔 말하면, token 길이 길어지면 굉장히 부담)

Convolution의 경우 k < n이면 모든 pair를 보는 것이 불가능 (since local)

그리고 Attention은 해석 가능한 결과를 제공한다고 함

- 이미 학습했듯이 꼭 Interpretable한 것은 아님
- Attention is not explanation
- Attention is not not explanation (어쩌라는거지…?)
- 이 분들 미디움에서 열 띈 토론하심
- 최근 XAI 결과 보면 Attention 기반보단 Attribution 기반으로 XAI하는 듯
- Engineer친구한테 물어본 결과, 굳이 사용? 이런 느낌 (실무에선)

03. Experiment and Results

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 [31]. 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 [17] 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 $warmup\_steps = 4000$.

Transformer 학습에서 Warm-up은 진짜 필수임

왜 그럴까? 이유를 분석해보니 Post-Layer Normalization 구조가 문제였다고 함!

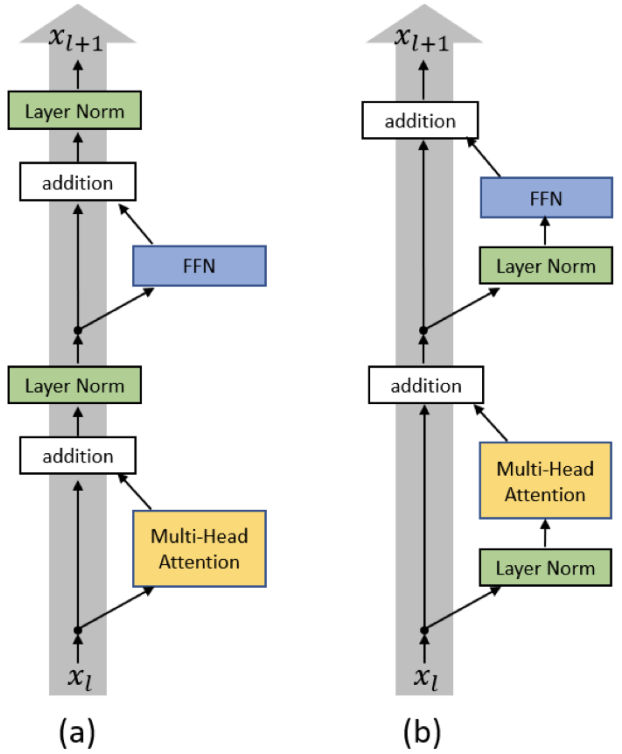


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

5.4 Regularization

We employ three types of regularization during training:

Residual Dropout We apply dropout [27] 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 $\epsilon_{ls} = 0.1$ [30]. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

$$y_k^L = y_k(1 - \alpha) + \alpha/k$$

03. Experiment and Results

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 $\alpha = 0.6$ [31]. 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 [31].

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].

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.

[5] We used values of 2.8, 3.7, 6.0 and 9.5 TFLOPS for K80, K40, M40 and P100, respectively.

WMT14 En->Gr and En-> Fr 풀었음 (오 드디어 독일어!)

세팅에 대해 궁금하면 상세히 읽어보시길 권장!

Beam Search에 대해 실험해본 흔적들을 찾을 수 있음 (사실 GNMT에서 정리해줘서..)

- Beam size (k) = 4
- Length penalty (alpha) = 0.1
- Maximum output length = input_length + 50

모델에 대한 실험은 Base vs Large, 과연 Self-Attention이 중요한지, Checkpoint Averaging (Ensemble) 등등을 실험

03. Experiment and Results

Model	All	No UNK°
RNNencdec-30	13.93	24.19
RNNsearch-30	21.50	31.44
RNNencdec-50	17.82	26.71
RNNsearch-50	26.75	34.16
RNNsearch-50⋆	28.45	36.15
Moses	33.30	35.63

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 [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	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**	**41.0**	$2.3 \cdot 10^{19}$	

03. Experiment and Results

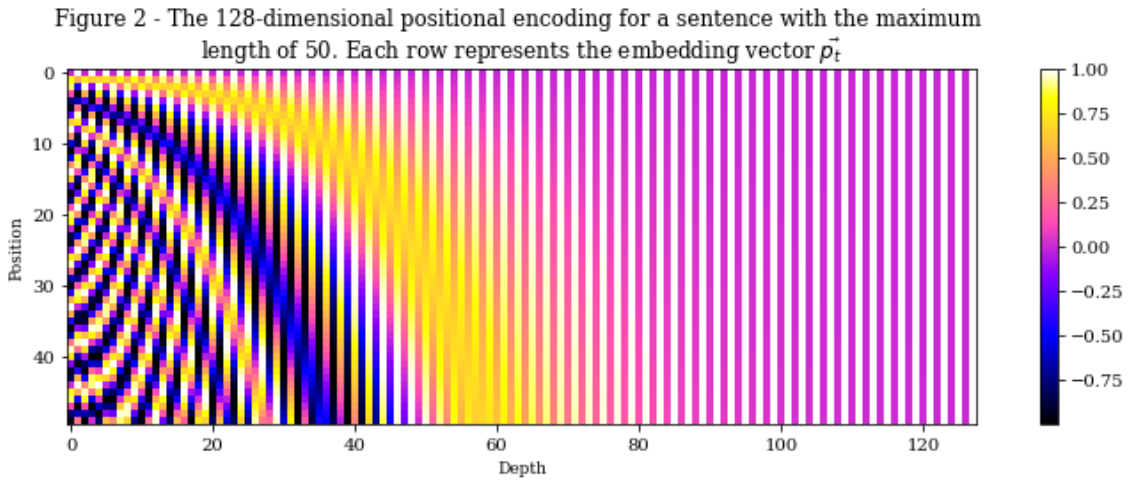
Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to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

	N	d_{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

04. Code Review

```python
import torch
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
%matplotlib inline

plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['Times New Roman'] + plt.rcParams['font.serif']

d = 128 # d_model
sl = 50 # seq_length

def positional_encoding(sl: int, d: int) -> torch.Tensor:
    p = torch.empty(sl, d) # generate positional embedding tensor
    ts = torch.arange(sl).float().view(-1, 1) # positional encoding index
    # weight per each dimensions
    w_k = 1/(10000**(2*torch.arange(d//2).float()/d)).view(1, -1) # Note that d is divisible by 2
    even_sin = torch.sin(ts @ w_k) # sin part (even number)
    odd_cos = torch.cos(ts @ w_k) # cos part (odd number)
    p[:, 0::2] = even_sin
    p[:, 1::2] = odd_cos
    return p

pe = positional_encoding(sl, d)

plt.figure(figsize=(10, 8), facecolor='w')
ax = plt.gca()
im = ax.imshow(pe.numpy(), cmap=plt.get_cmap('gnuplot2'))
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="2%", pad=0.5)
ax.set_xlabel('Depth', fontsize=9)
ax.set_ylabel('Position', fontsize=9)
ax.set_title("Figure 2 - The 128-dimensional positional encoding for a sentence with the maximum \n"
             r"length of 50. Each row represents the embedding vector $\vec{p_t}$",)
plt.colorbar(im, cax=cax)
plt.show()
```

https://github.com/huggingface/transformers/blob/master/src/transformers/models/bart/modeling_bart.py



부스트캠프 AI Tech 2기

boostcamp ai tech

Discussion

Email : jinmang2@gmail.com

GitHub : github.com/jinmang2

Huggingface Hub: huggingface.co/jinmang2

진명훈_T2216