Visual Transformer in CV

Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering &
Artificial Intelligence

References

Main

- https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
- https://jalammar.github.io/illustrated-transformer/

blog Sub

- https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632

Newly tutorials

- https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

Main

- https://github.com/lucidrains/vit-pytorch
- https://github.com/FrancescoSaverioZuppichini/ViT
- https://pypi.org/project/vision-transformer-pytorch/

Attention

[출처] http://dmqm.korea.ac.kr/activity/seminar/296

Attention

➢ Key, Query, Value in Attention

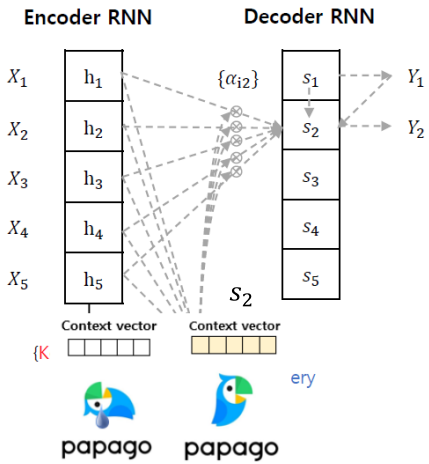
- Attention: Query와 key의 Similarity를 계산한 후 value 의 가중합을 계산
- Attention score: Value 에 곱해지는 가중치
- Considerations
  - ✓ Key, Query, Value = Vectors (Matrix/Tensor)
  - ✓ Similarity function

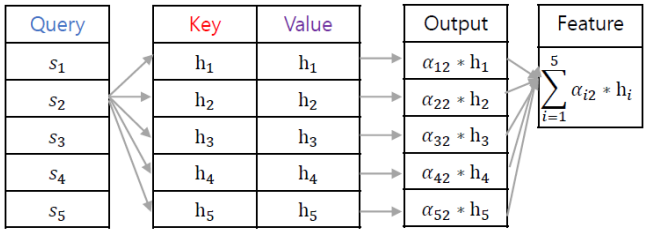
$$A(q,K,V) = \sum_i softmax\big(f(K,q)\big)V$$

Attention in Seq2seq Machine Translation

- Key, Value = Hidden states of encoder, h_i
- Query = Hidden state of decoder, s_i
- Feature = Context vector at time step 2

Query = s_2



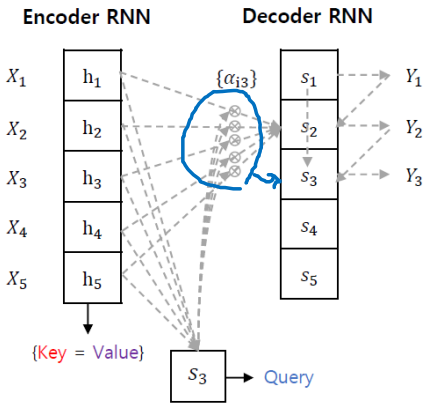


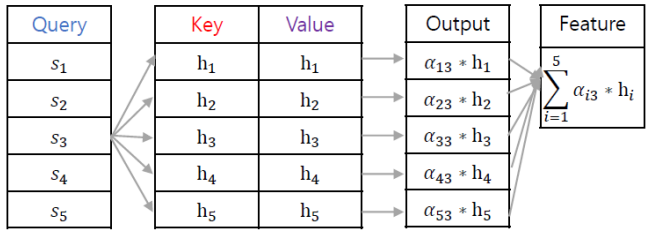
$$\{\alpha_{i2}\} = softmax(f(h_i, s_2))$$

$$feature = \sum_{i=1}^{5} \alpha_{i2} * h_i$$

$$A(q,K,V) = \sum_i softmax(f(K,q))\,V$$

Query = s_3





$$\{\alpha_{i3}\} = softmax(f(h_i, s_3))$$

$$feature = \sum_{i=1}^{5} \alpha_{i3} * h_i$$

$$A(q,K,V) = \sum_i softmax(f(K,q))\,V$$

Attention

[출처] http://dmqm.korea.ac.kr/activity/seminar/296

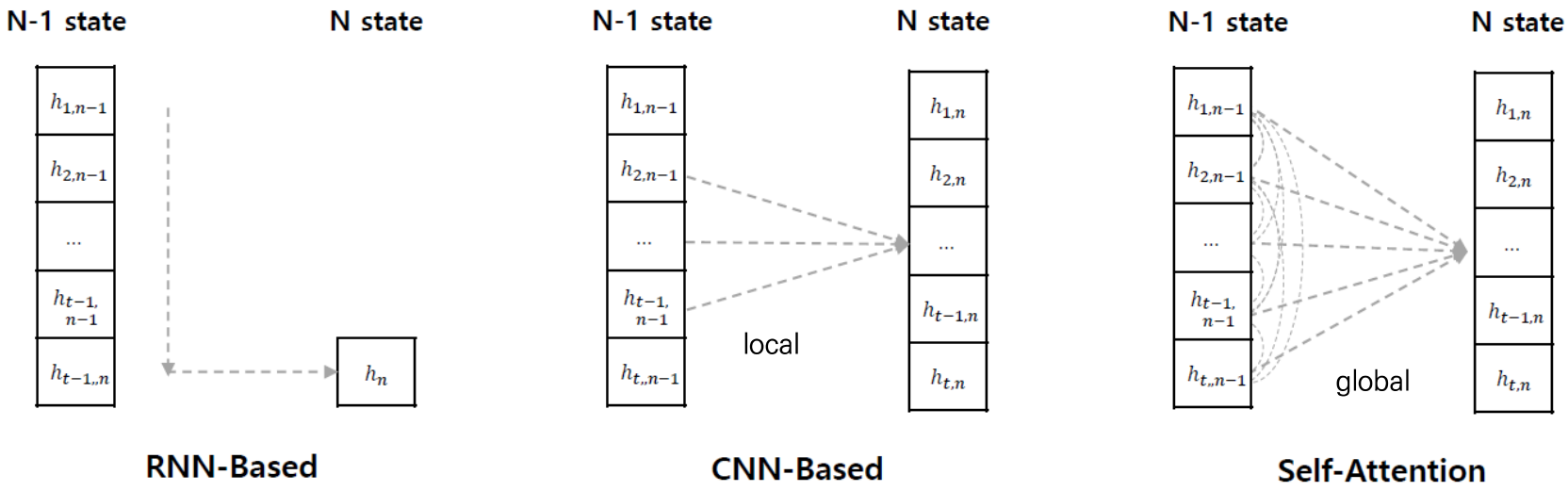
Self-Attention

➢ RNN, CNN 구조를 사용하지 않고, attention 만을 사용하여 feature 표현

- Key = Query = Value = Hidden state of word embedding vector
- Scaled dot-product attention
- Multi-head attention

$$Key = Query = Value$$

$$A(q, K, V) = \sum_i softmax(f(K, q))V$$



RNN-Based

✓ Sequential data → Parallel computing X
✓ Increase Calculation time and complexity
✓ Vanishing gradient / Long term dependency

CNN-Based

✓ Long path length between long-range dependencies

✓ *Inductive bias* : local 영역에 Spatial 정보를 많이 얻을 수 있다는 가정

Self-Attention

✓ 모든 정보를 활용하므로, inductive bias가 부족 – 많은 학습 데이터가 필요

Attention

[출처] http://dmqm.korea.ac.kr/activity/seminar/316
[출처] https://web.eecs.umich.edu/~justincj/slides/eecs498/498_FA2019_lecture13.pdf

Attention vs Self-Attention

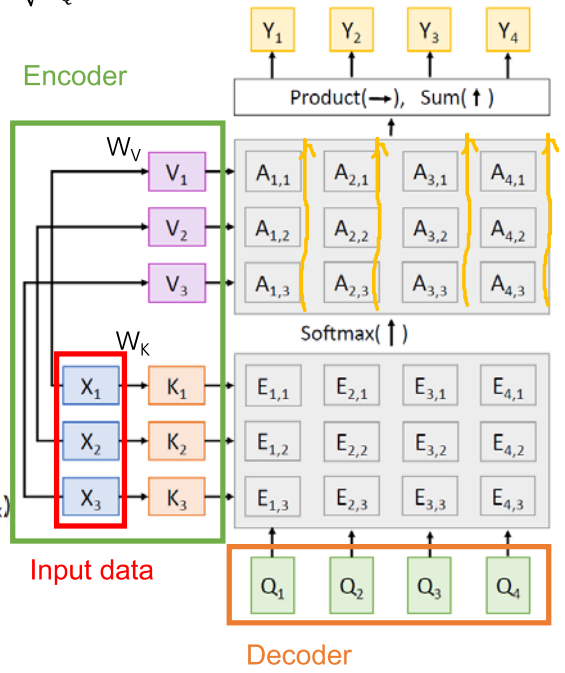
- Attention (Decoder → Query / Encoder → Key, Value) : Encoder, Decoder 사이의 상관관계를 바탕으로 특징 추출
- Self attention (입력 데이터 → Query, Key, Value) : 데이터 내의 상관관계를 바탕으로 특징 추출

$$Y_i = \sum_j softmax(\frac{Q_i (X_j W_K)^T}{\sqrt{D_Q}}) X_j W_V$$

Attention Layer

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_Q x N_X) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = softmax(E, dim=1)$ (Shape: N_Q x N_X)
Output vectors: $Y = AV$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$



N : number, D : Dimension

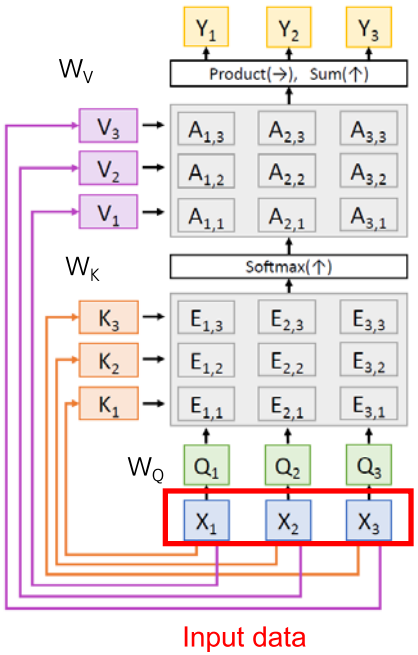
$$Y_i = \sum_j softmax(\frac{X_i W_Q (X_j W_K)^T}{\sqrt{D_Q}}) X_j W_V$$

Self-Attention Layer

One query per input vector

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

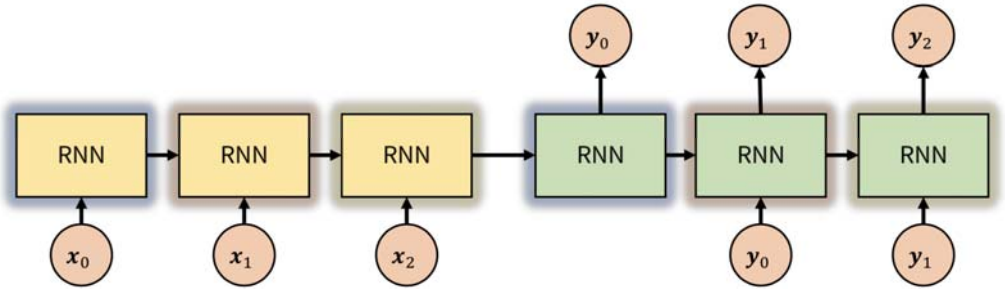
Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = softmax(E, dim=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

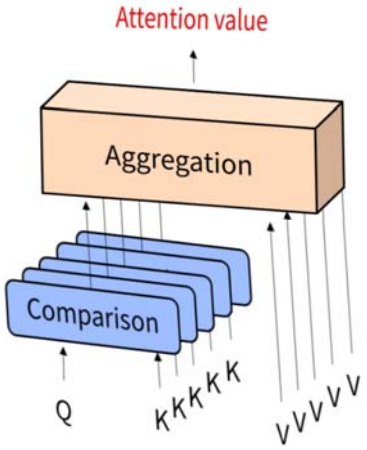
Attention



Decoder 단에서 어떤 Encoder 정보에 '집중'해야 하는지 알 수 있다면, 도움이 될 것이다.

이것이 Attention mechanism의 기본 아이디어!

- Attention mechanism은 key-value 쌍이 있고, query를 보내어 query와 key를 유사 비교를 한 뒤, 유사도를 고려한 Value들을 섞어서 Aggregation한 것이 Attention value이다.



$$q \in \mathbb{R}^n, k_j \in \mathbb{R}^n$$

$$\text{Compare}(q, k_j) = q \cdot k_j = q^T k_j$$

$$\text{Aggregate}(c, V) = \sum_j c_j v_j$$

Compare 함수로는 Dot-Product (Inner Product)가 많이 쓰이며,

Aggregation은 weighted sum을 많이 사용한다.

- 결국 Query와 비슷하면 비슷할 수록 높은 가중치를 주어 출력을 주는 것이다.
- compare 함수로는 Dot-Product가 많이 쓰이며, 여기서 k와 q가 각각 벡터 norm이 1이라면 결국 코사인 유사도를 구하는 것과 동일해 질 것이다. 그러나 길이가 1이 아닐 경우를 생각해서 Dot-product이후에 softmax를 사용하여 전체의 합을 1로 만들어, 각각의 가중치들을 하나의 확률로 사용할 수 있게끔 변환해 주어 사용한다.

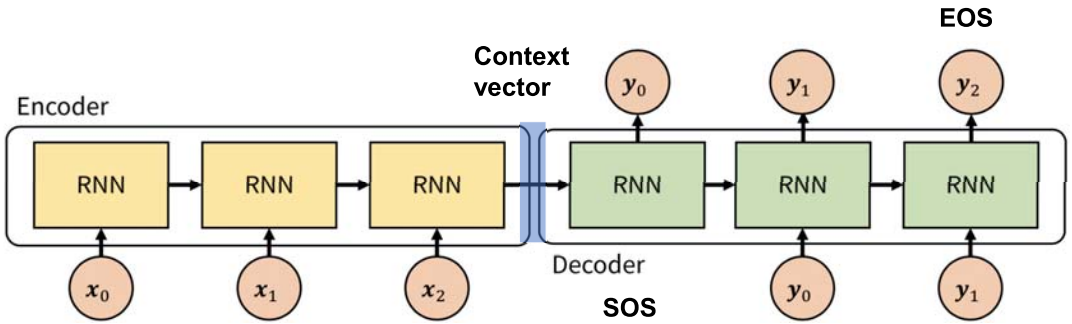
Q에 대해 어떤 K가 유사한지 비교하고, 유사도를 반영하여 V들을 합성한 것이 Attention value이다.

Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

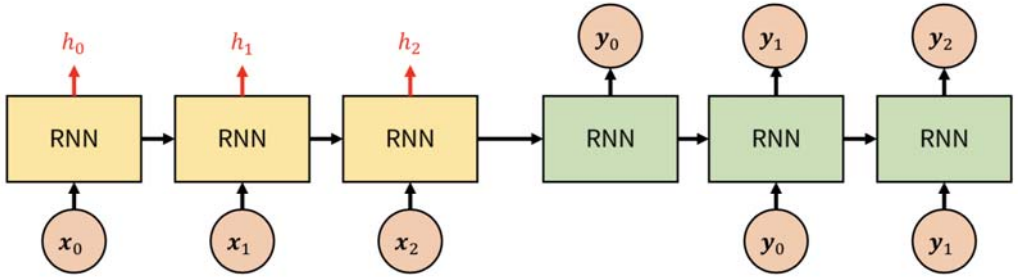
Seq2seq

- Seq2seq 모델은 Encoder구조를 통해 Feature들을 만들게 되고 최종적으로는 출력으로 Context를 생성하여 이 Context 하나에 의지해서 Decoder는 SOS(Start Of Sequence)를 시작으로 출력으로는 단어를 하나씩 내어주는 모델이다.



Seq2seq - Key-Value

- Key-value쌍은 기존의 Context만을 보며 출력을 내주었던 것과 다르게 Decoder부분의 Hidden Layer에 대한 출력을 낼 때, Encoder 부분에 중간중간 부분을 알게 하기 위해서 사용되어진다.
- 직관적으로 생각을 해보면, Decoder에서 어떤 것을 찾고자 한다면, 찾고자 하는 것에 대한 정보는 Encoder에서 찾을 수 있을 것이다. 그렇기에 Key-value가 Encoder의 Hidden State h_i가 되는 것이다.



Seq2seq에서는 Encoder의 hidden layer들을 key와 value로 사용한다.

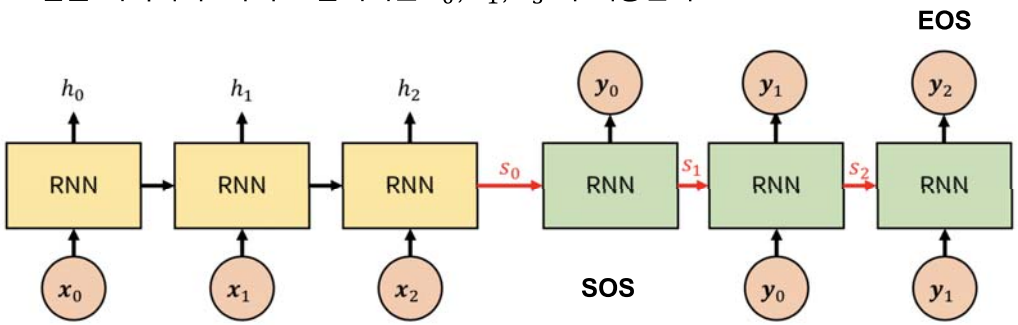
- 현재의 출력을 내기 위해서는 현재의 출력이 사용될 수는 없기 때문에 즉, 미래(예측)를 가지고 현재 출력을 만들어낼 수는 없기 때문에 하나 앞선 Decoder Hidden state s_i를 query로 사용하는 것이다.
- 대부분의 Attention network에서는 key와 value를 같은 값을 사용한다.
  Seq2seq에서는 Encoder의 Hidden Layer들을 key와 value로 사용한다.

Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Seq2seq - Query

- Query는 Decoder의 Hidden Layer들을 사용하는데, 해당 출력을 해야 하는 RNN 구조의 하나 이전의 time-step의 Hidden Layer를 Query로 사용한다는 점을 기억하자! 아래 그림에서는 s_0, s_1, s_3 가 해당한다.



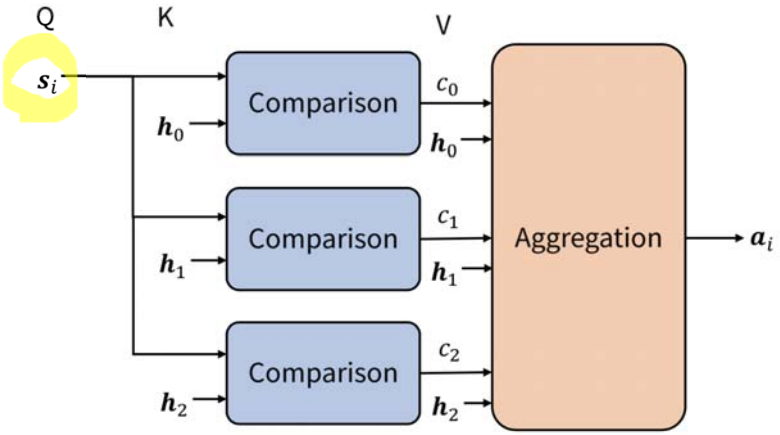
Seq2seq에서는 Decoder의 hidden layer들을 Query로 사용한다.

주의할 점은, Encoder와 달리 하나 앞선 time-step의 hidden layer를 사용한다는 점.

- 현재의 출력을 내기 위해서는 현재의 출력이 사용될 수는 없기 때문에 즉, 미래(예측)를 가지고 현재 출력을 만들어낼 수는 없기 때문에 하나 앞선 Decoder Hidden state s_i를 query로 사용하는 것이다.

Seq2seq – Attention Mechanism

- i-th query가 들어오면 각각 Key와 비교하고, 내적한 뒤 Softmax를 해줘 가중치로 만든 뒤에 각각에 해당하는 Value와 곱해 가중합을 한 것을 Attention value로 산출한다.



i번째 decoder에 대해서 a_i의 attention value를 얻는다.

블록도에 비해 수식이 오히려 간단하다. 비교 함수와 결합 함수의 의미를 잘 이해하자.

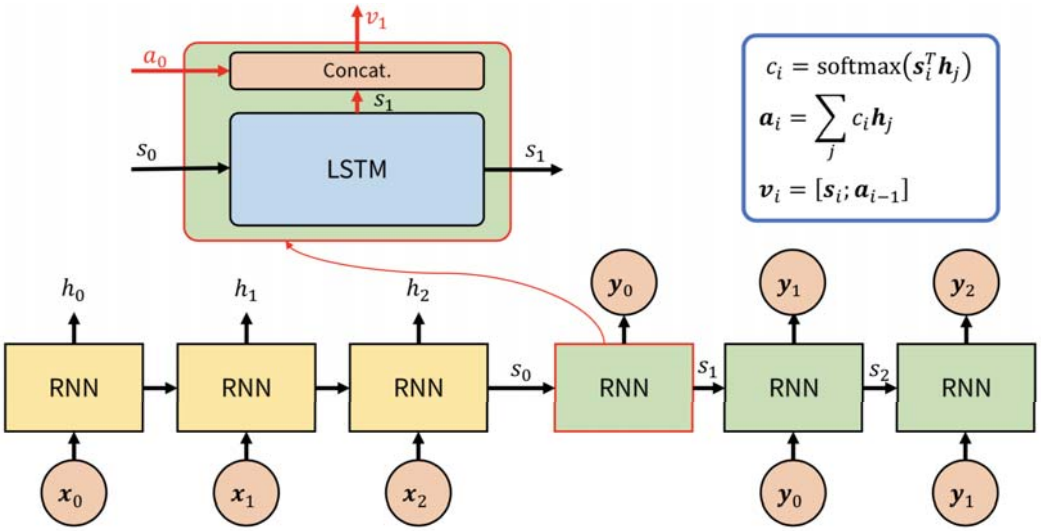
- Key와 Value는 서로 동일한 Encoder의 Hidden State h_i 들이 사용되며, Query는 Decoder에 있는 각각의 Hidden State s_i 들이 될 것이다.
- i-번째 time step에 대한 Query를 보내어 Encoder에 있는 모든 Key와 유사도를 비교해서 최종적으로는 유사도를 고려한 Aggregation한 Attention Value를 출력한다.

Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Seq2seq - Application

- 아래 그림에서 출력과 RNN 구조사이에 실제로는 FC Layer가 하나 존재해서 출력을 One-hot vector로 만들어준다.



RNN으로 Hidden state를 입력하기 전에, attention value를 concatenate하여 입력한다.

Hidden state에 attention value를 concatenate까지 하면 모든 수식적 표현이 끝난다.

- Attention Value(a_0)를 입력 받아 Hidden state s_0 에서 LSTM 구조를 거쳐 Hidden state s_1가 나올 것이다. 이 새롭게 얻어진 Hidden state s_1에 s_0를 통해 얻어진 Attention Value(a_0)와 Concatenate를 하여 v_1를 출력한다.
- 이전에는 Decoder에서 그대로 Hidden state가 나오던 것이 이제는 Encoder의 Hidden state들을 비교해서 만들어낸 Attention Value를 같이 출력함으로써, Encoder 부분의 value들을 잘 가져올 수 있도록 해주었다.

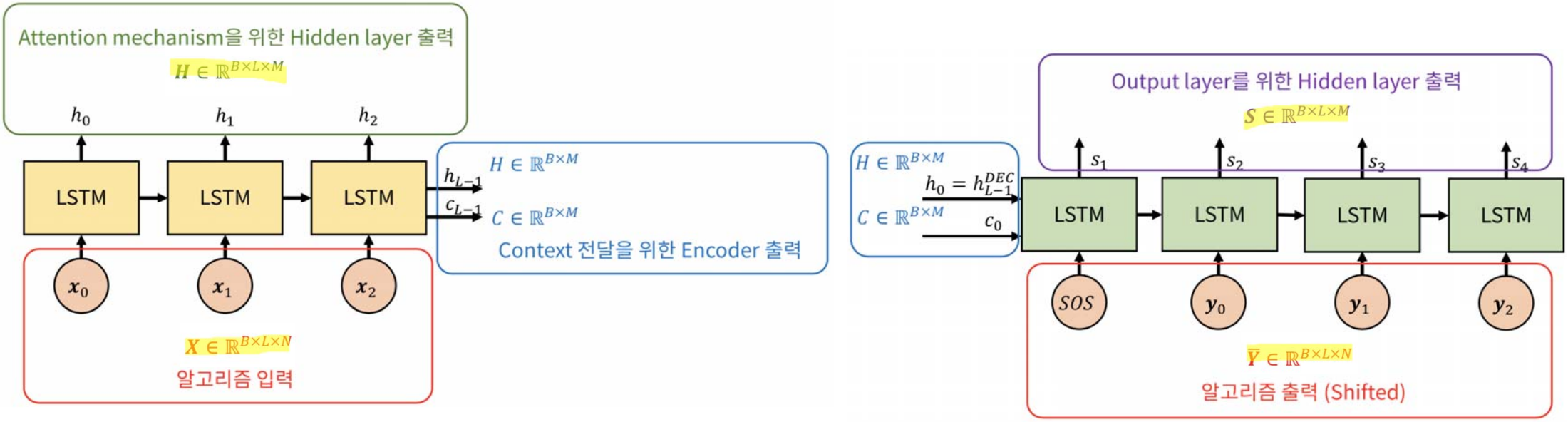
Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Seq2seq – Encoder 입출력

- $X \in R^{BxLxN}$에서 B:Batch size, L:문장의 길이, N:One-hot vector나 embedding Feature의 길이를 의미하며, 여기서 Decode 쪽으로 Context를 넘길 때는 LSTM이라면 Hidden State와 Cell State 둘 다 넘겨주어야 하기에 Batch size (B) X Hidden state의 Feature 갯수 (M) 크기의 tensor를 넘겨줄 것이다.

Seq2seq – Decoder 입출력

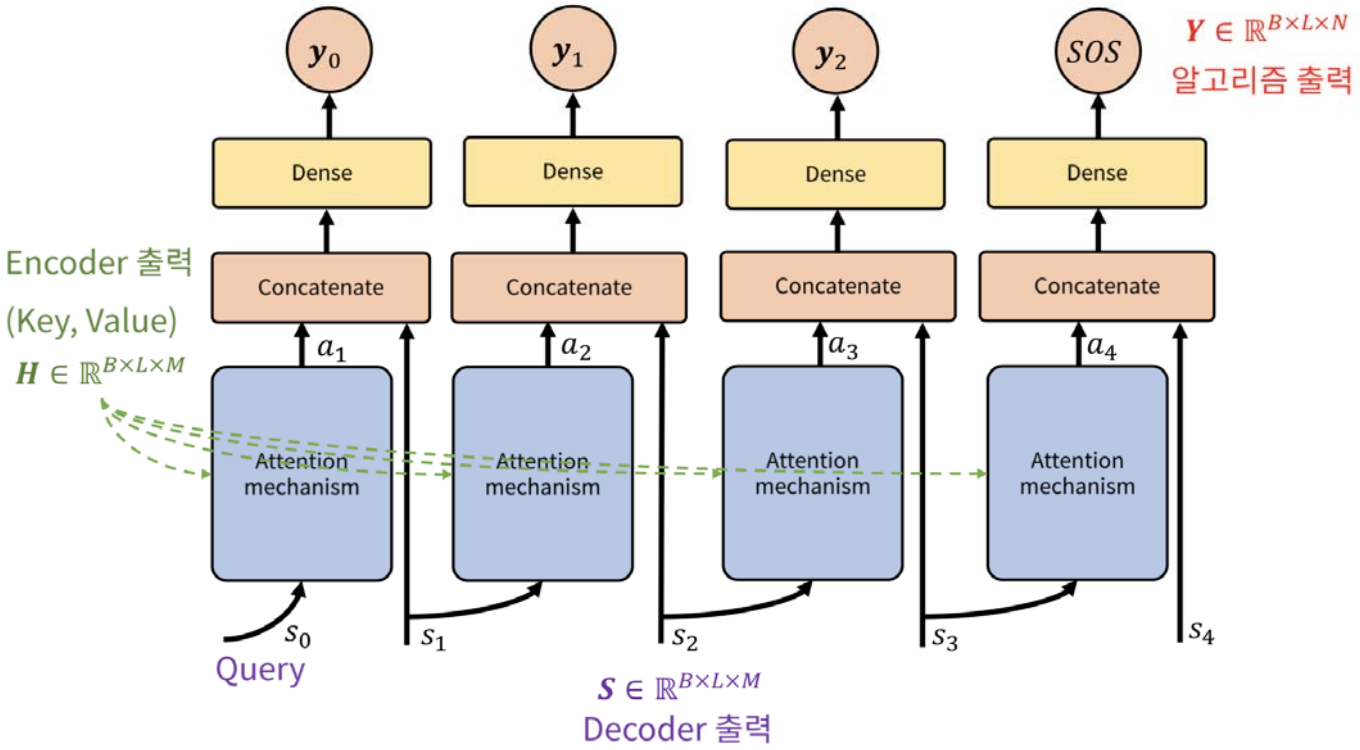


Attention

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Seq2seq – Output w/Attention (학습단계)

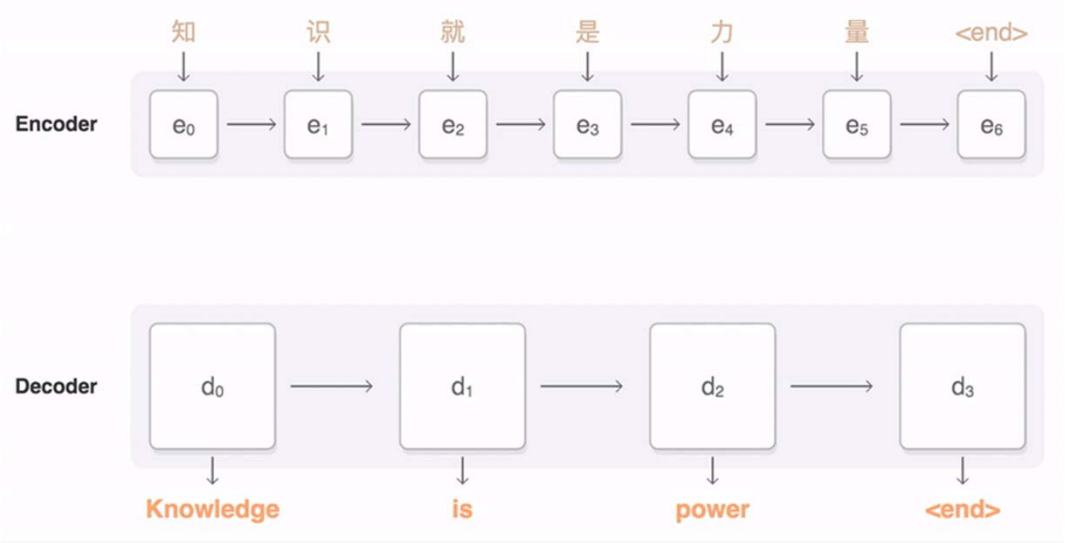
- Encoder의 Hidden State인 H가 Attention mechanism에 Key와 Value로 입력이 되고, Query에는 Decoder의 한 step 앞선 Hidden State를 사용하게 된다.



Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer vs Seq2seq



seq2seq in GNMT, visualization by Google AI Blog

Multi-step attention form ConvS2S via Michal Chromiak's blog

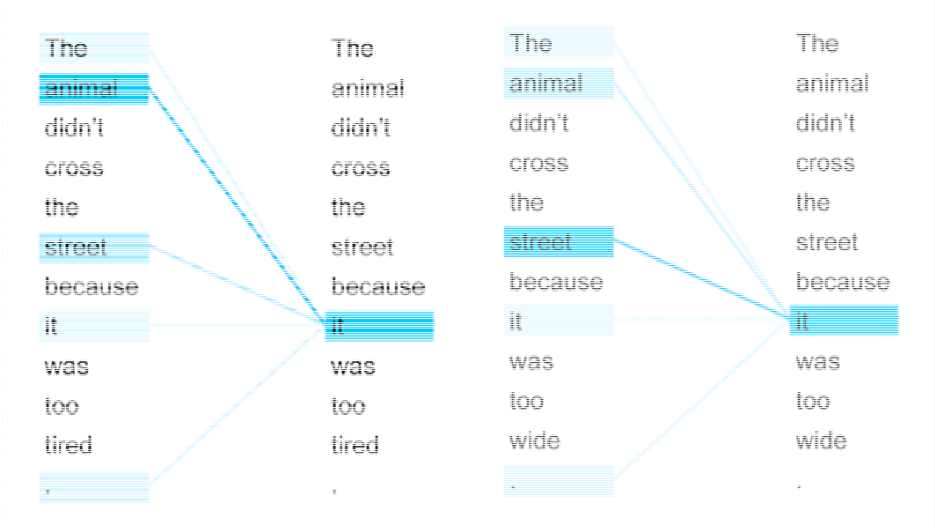
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Comparison of RNN-based, CNN-based and Self-Attention models based on computational efficiency metrics

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer



The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

Transformer step-by-step sequence transduction in form of English-to-French translation. Adopted from Google Blog

Transformer : Attention is all you need

A. Vaswani, et al. (Google Brain), Attention Is All You Need, 2017
https://arxiv.org/abs/1706.03762

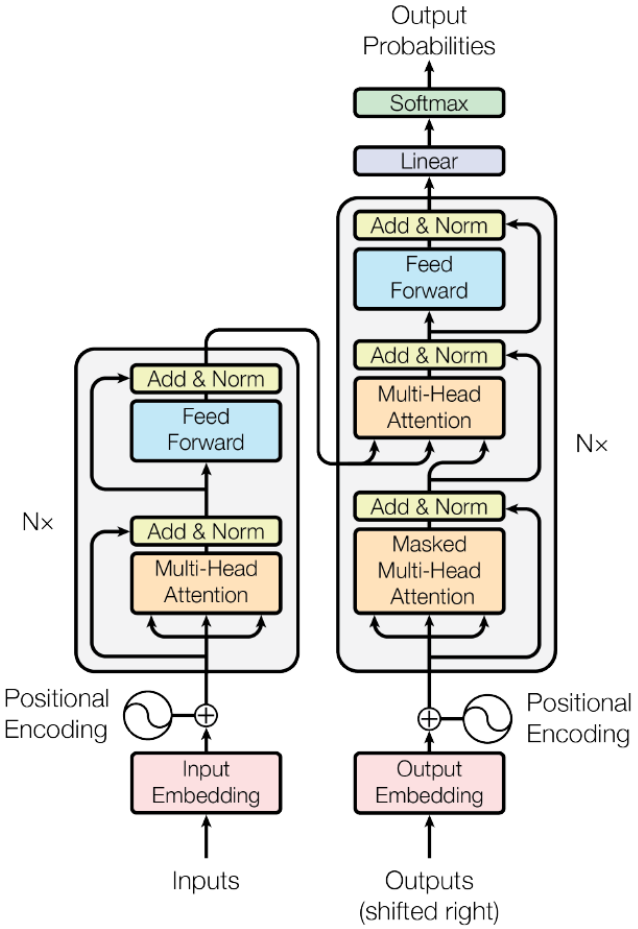


Fig. 1. The Transformer – model architecture

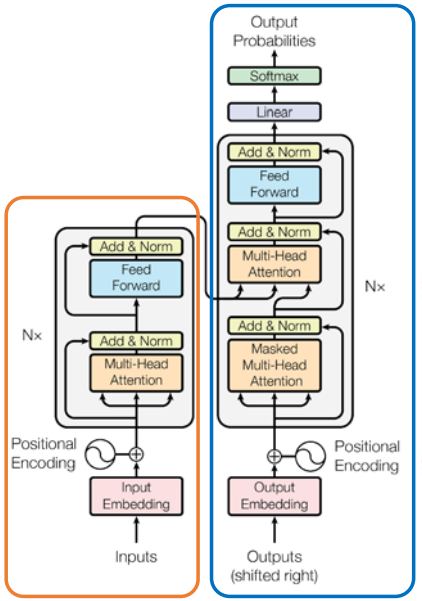
Translation Task에 RNN과 CNN 사용하지 않고, Attention 만을 이용하여 State-of-the-art 성능을 도출한 연구

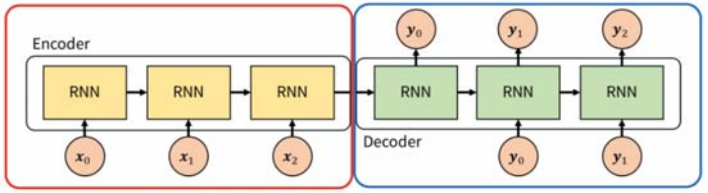
- RNN 같은 경우는 순서대로 입력하기 때문에 입력된 단어의 위치를 따로 표시하지 않아도 되지만, Transformer 구조 같은 경우에는 병렬적으로 계산하므로, 현재 계산하고 있는 단어가 어느 위치에 있는 단어인지를 표현해주어야 해서 *positional encoding*을 사용한다.

- Seq2seq와 유사한 transformer 사용
- Scaled Dot-Product Attention과 이를 병렬로 나열한 Multi-Head Attention 블록이 핵심

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer vs Seq2seq





- Seq2seq 모델은 Encoder와 Decoder가 있고 그 사이에 Context가 전달
- Transformer 모델은 Input쪽(왼쪽의 빨간색 박스)과 Output쪽(왼쪽의 파란색 박스)로 구성되며, Input쪽에서는 Input embedding이 들어가서 Encoding이 되고 Context가 전달이 되어 Output쪽의 Decoder부분에서 Decoding이 되어 출력이 나옴
- Seq2seq 모델은 RNN로 구성되어 있어서 순차적으로 이루어지고, Transformer 모델은 병렬적으로 계산되므로 Input쪽이 동시에 계산되고 Output쪽이 동시에 계산되는 형태로 학습이 되는 점이 차이점이다.

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

A. Vaswani, et al. (Google Brain), Attention Is All You Need, 2017
https://arxiv.org/abs/1706.03762

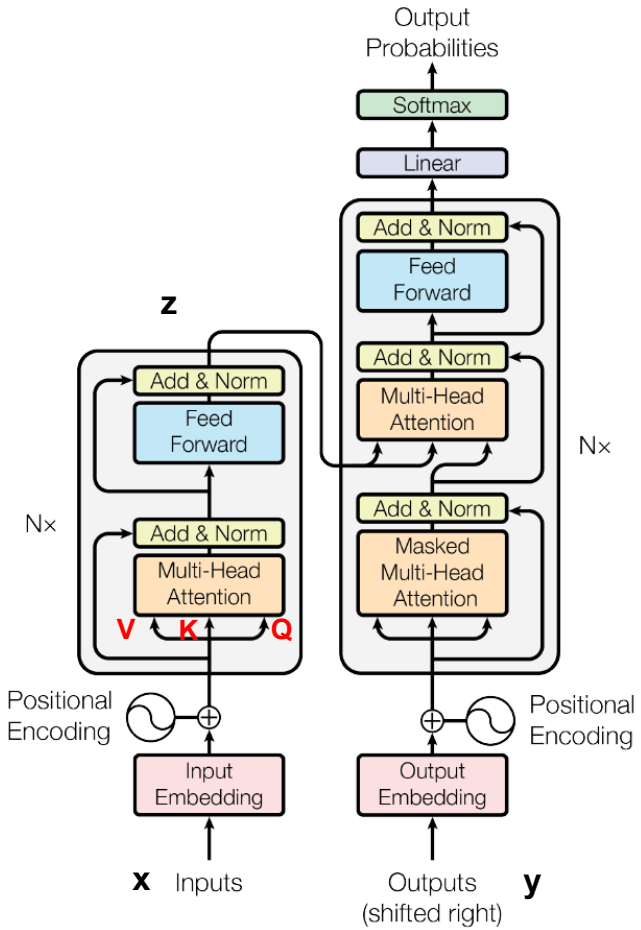


Fig. 1. The Transformer – model architecture

- Seq2seq와 유사한 Transformer 구조 사용
- 제안하는 Scaled Dot-Product Attention과, 이를 병렬로 나열한 Multi-Head Attention 블록이 알고리즘의 핵심
- RNN의 BPTT와 같은 과정이 없으므로 병렬 계산 가능
- 입력된 단어의 위치를 표현하기 위해 Positional Encoding 사용

Encoder

- Maps an input sequence of symbol representations $(x_1, \cdots, x_n)$ to a sequence of continuous representations $(z_1, \cdots, z_n)$
- **A stack of *N*=6 identical layers with 2 sublayers** : **Multi-head self-attention mechanism**, **Position-wise fully connected feed-forward network.**
- **Residual connection** around each of 2 sub-layers, followed by **layer normalization**
- The output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$
- All sub-layers as well as the embedding layer produce outputs of dimension, $d_{\mathrm{model}} = 512$.

Decoder

- Given z, generate a output sequence $(y_1, \cdots, y_n)$ of symbols one element at a time.
- Two sub-layers in each encoder layer, Third sub-layer, which performs multi-head attention over the output of the encoder stack.
- Modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions.

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Input & Output

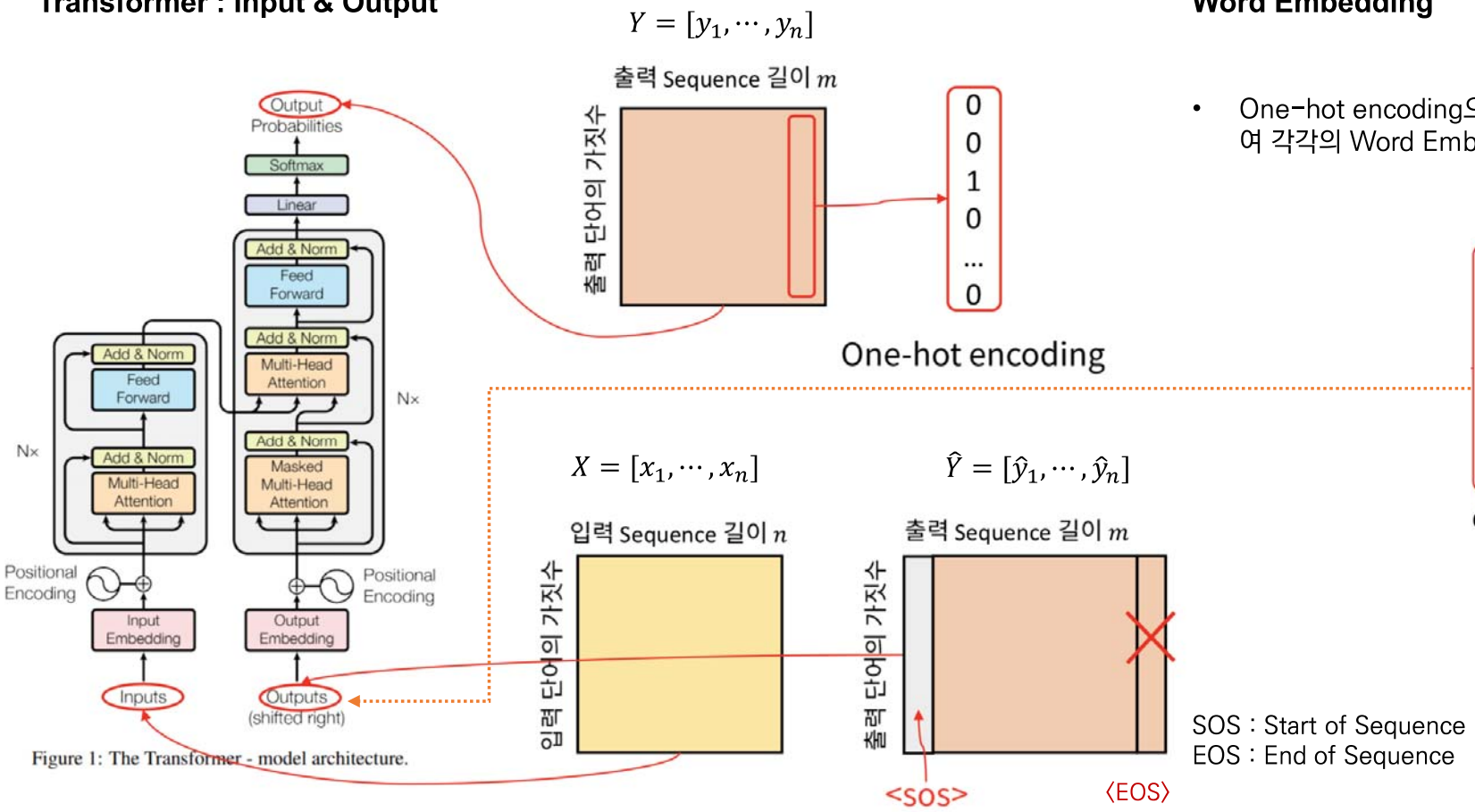


Figure 1: The Transformer - model architecture.

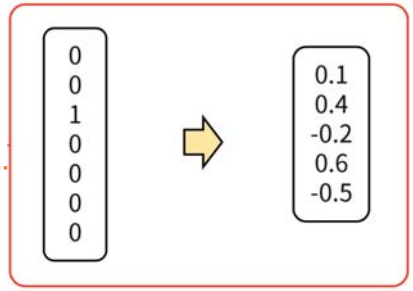
SOS : Start of Sequence
EOS : End of Sequence

Input : 노란색 Matrix 형태로 되어 있으며, 일반적으로는 입력 단어의 가짓수와 출력 단어의 가짓수는 동일할 것이다. 만약 기계번역처럼 2개 언어가 다르면, 다를 것이다!

Output : 원래 Seq2seq 구조에서 보았듯이 shift 시킨 입력을 넣어 주었던 것과 같이 SOS를 넣어주고 EOS를 빼준 형태로 Outputs에 넣어준다.

Word Embedding

- One-hot encoding으로 되어있던 것들을 Embedding하여 각각의 Word Embedding에 넣어준다.

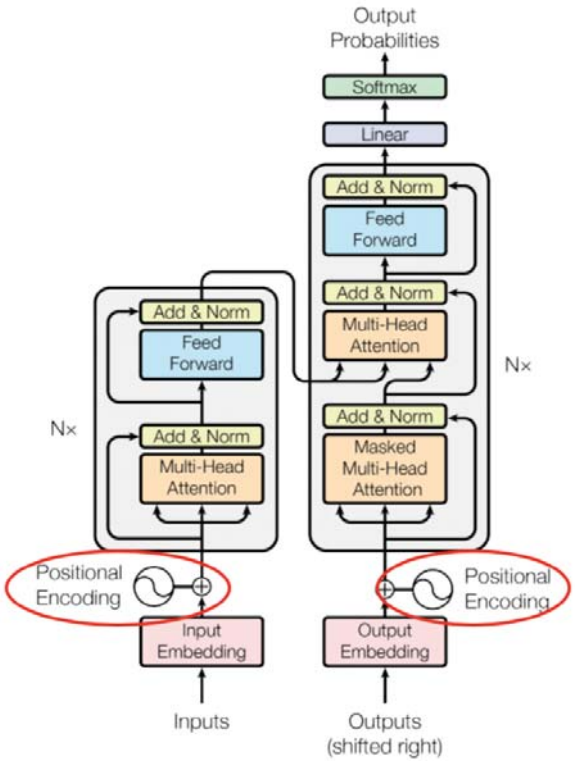


One-Hot Encoding된 단어를 실수 형태로 변경하면서 차원의 수를 줄이는 방법

Transformer : Attention is all you need

Transformer : Positional Encoding

- No Recurrence/Convolution이므로, 시퀀스 상에 토큰의 상대적 또는 절대적 위치에 대한 정보 추가 필요
- Encoder와 Decoder Stacks 아래에 Input Embeddings에 Positional Encodings 추가
- Positional Encoding은 시간적 위치가 다를 때마다 고유 코드를 생성하여 Input Embedding에 더해주는 형태로 구성되어 있다. → 전체 Sequence의 길이 중 상대적 위치에 따라서 고유의 벡터를 생성하여 Embedding된 벡터에 더해준다.
- sin법칙과 cos법칙에 의해 각각 분리해서 쓸수 있는데 결국 덧셈과 뺄셈으로 이 Positional Encoding이 달라지기 때문에 FC Layer에서 학습하는데 용이하게 됨

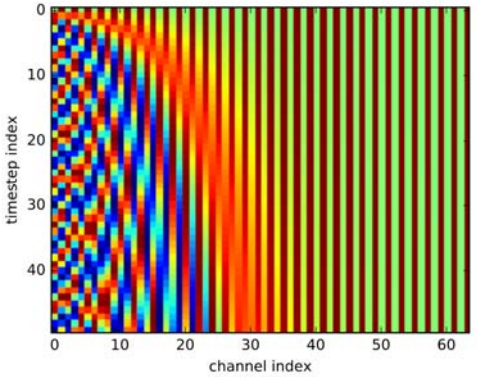


[출처] https://skyjwoo.tistory.com/entry/positional-encoding%EC%9D%B4%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80
https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

- 다양한 주파수의 sin과 cos 함수 사용 : wavelength from 2π to $10000 \cdot 2\pi$

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

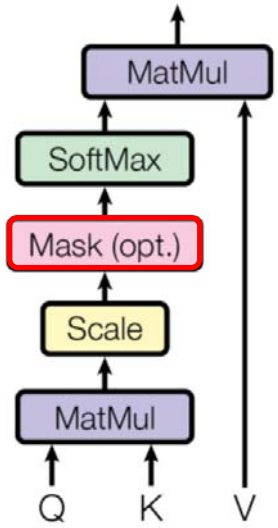
$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$



Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Scaled Dot-Product Attention



- Query, Key-Value의 구조를 띄고 있음
- Q와 K의 비교 함수는 Dot-Product와 Scale로 이루어짐
- Mask를 이용해 Illegal connection의 attention을 금지
- Softmax로 유사도를 0 ~ 1의 값으로 Normalize
- 유사도와 V를 결합해 Attention value 계산

- Input : d_k dim의 queries와 keys, d_v dim의 values

$$Q = [\boldsymbol{q}_0, \boldsymbol{q}_1, \dots, \boldsymbol{q}_n]$$
$$K = [\boldsymbol{k}_0, \boldsymbol{k}_1, \dots, \boldsymbol{k}_n]$$
$$V = [\boldsymbol{v}_0, \boldsymbol{v}_1, \dots, \boldsymbol{v}_n]$$

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

$$\boldsymbol{a} = C^T V = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

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

- Additive attention vs dot-product attention
- dot-product attention : Much faster and more space-efficient

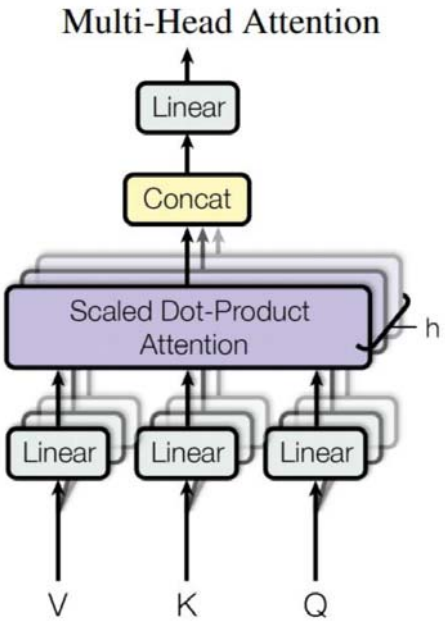
- Small d_k : additive and dot-product attention mechanism perform similarly. Additive attention outperforms dot product attention without scaling for larger d_k.
- Large d_k → dot products grow large in magnitude → pushing the softmax function into regions where it has extremely small gradients.
  → Scale the dot product by $\frac{1}{\sqrt{d_k}}$

- Mask를 이용해서 Illegal connection의 Attention을 금지 : self-attention에 대한 이야기인데, 일반적인 Attention 구조는 Decoder쪽에 Hidden Layer를 통해 Output을 내려면 Encoder 쪽에 Hidden Layer 전체와 비교해서 산출을 해야 하므로 이런 경우는 괜찮지만, Self-attention에서는 Decoder를 똑같은 Decoder 자기 자신과 Attention을 할 수가 있는데 여기서 Decoder 부분의 해당 Hidden Layer를 산출하려면 순차적으로 출력이 나온다고 했을 때 해당 부분의 Decoder보다 이후 시점은 아직 결과가 산출되지 않았기 때문에 그보다 앞선 시점의 Decoder부분에서의 Hidden Layer들만을 사용할 수 있다는 이야기이다. 여기서 비교할 때 사용할 수 없는 Hidden Layer들을 Illegal connection이라고 한다. 이런 Illegal connection은 Mask를 통해 -inf로 보내버리면 Softmax에서 값이 0이 되는 것을 이용하여 attention이 안되도록 구현하고 있다.

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Multi-Head Attention



- Linear 연산 (Matrix Mult)를 이용해 Q, K, V의 차원을 감소
  Q와 K의 차원이 다른 경우 이를 이용해 동일하게 맞춤
- h개의 Attention Layer를 병렬적으로 사용 – 더 넓은 계층
- 출력 직전 Linear 연산을 이용해 Attention Value의 차원을 필요에 따라 변경
- 이 메커니즘을 통해 병렬 계산에 유리한 구조를 가지게 됨

$$\text{Linear}_i(V) = VW_{V,i} \quad W_{V,i} \in \mathbb{R}^{d_v \times d_{model}}$$
$$\text{Linear}_i(K) = KW_{K,i} \quad W_{K,i} \in \mathbb{R}^{d_k \times d_{model}}$$
$$\text{Linear}_i(Q) = QW_{Q,i} \quad W_{Q,i} \in \mathbb{R}^{d_q \times d_{model}}$$

- d_{model} 차원의 Q,K,V를 가지는 Single attention 함수 보다, Q, K, V를 h 배로 d_k , d_k, d_v 차원으로 선형 Project함.
  → Attention 함수를 병렬 처리하고, d_v 차원의 output values 결과

$$\text{MultiHead}(\text{Q}, \text{K}, \text{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})W^O$$
$$\text{where head}_1 = \text{Attention}(Q\text{W}_i^Q K\text{W}_i^K, V\text{W}_i^V)$$

- where projections are parameter matrices $\text{W}_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $\text{W}_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $\text{W}_i^V \in \mathbb{R}^{d_{model} \times d_v}$, $\text{W}_i^O \in \mathbb{R}^{hd_v \times d_k}$
- h=8 parallel attention layers (또는 heads)
- $d_k = d_v = d_{model}$/8=512/8=64

제일 아래 단계의 Linear 연산을 통해 Q,K,V의 차원을 감소(h개로 나눠짐)시키는 것이 중요하다. 또한 가중치 $W_{V,i}$, $W_{K,i}$, $W_{Q,i}$ 의 각각의 Dimension 보다 더 작은 값으로 모델의 Dimension(d_{model})을 해준다. 이는 value, key, query의 차원을 모델에 사용하는 차원으로 차원을 변환시켜주는 의미이기도 하다.

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Masked Multi-Head Attention

- Mask는 RNN의 Decoder단을 생각해 보았을 때, context가 앞에서 뒤로 넘어가면서 이미 구한 것들만 참조를 할 수 있는데, **Transformer구조에서는 병렬적으로 계산을 하기 때문에 self-attention을 할 경우에는 시간적으로 앞에서 일어난 것들에 대해서만 영향을 받게 해주어야 RNN과 동일한 구조가 되기 때문에 Mask를 이용해서 예측하고자 하는 시점을 포함한 미래값들을 가려준다**.

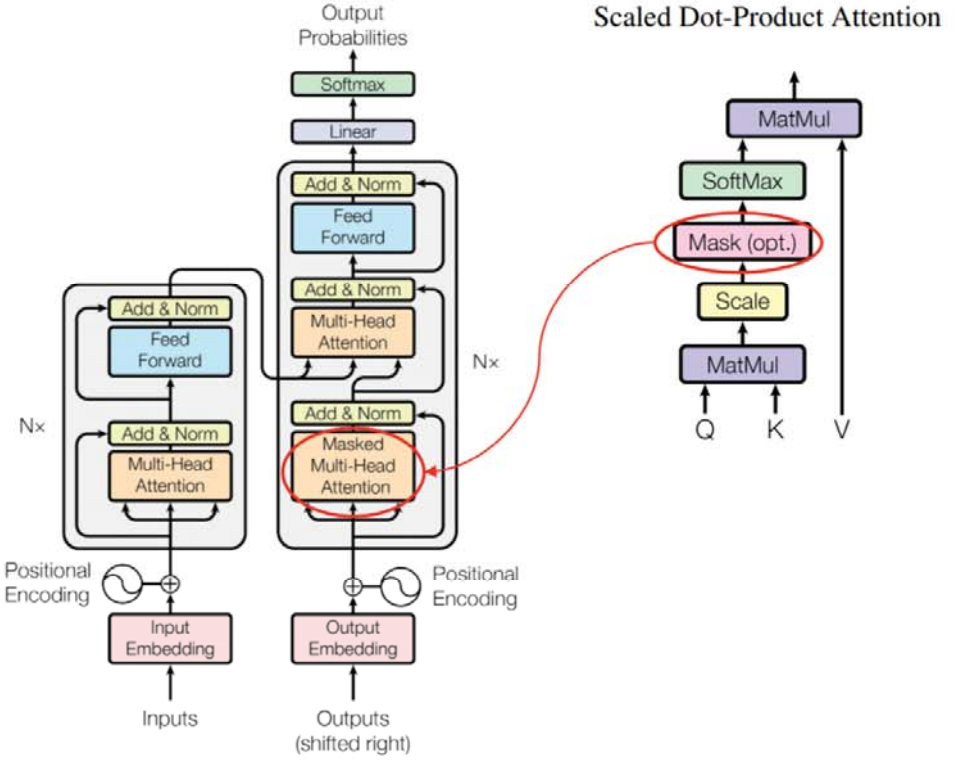
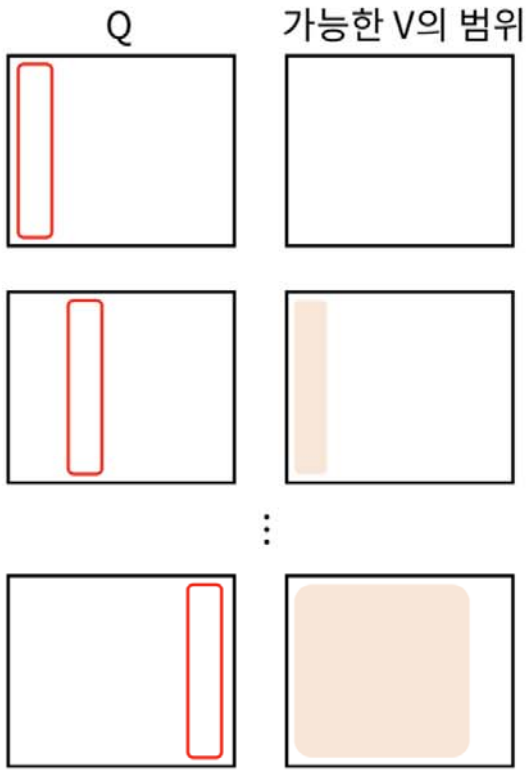


Figure 1: The Transformer - model architecture.



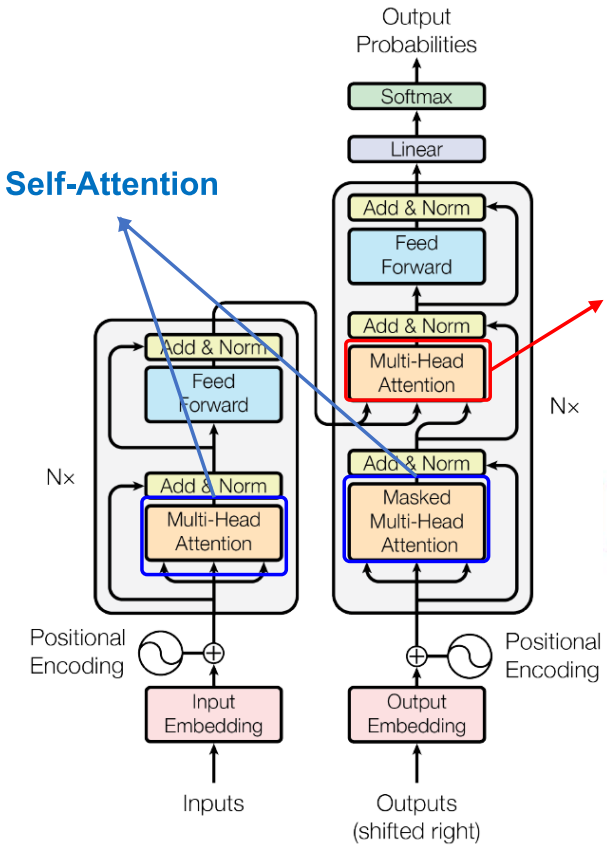
Self-Attention에서 자기 자신을 포함한 미래의 값과는 Attention을 구하지 않기 때문에, Masking을 사용한다.

Transformer : Attention is all you need

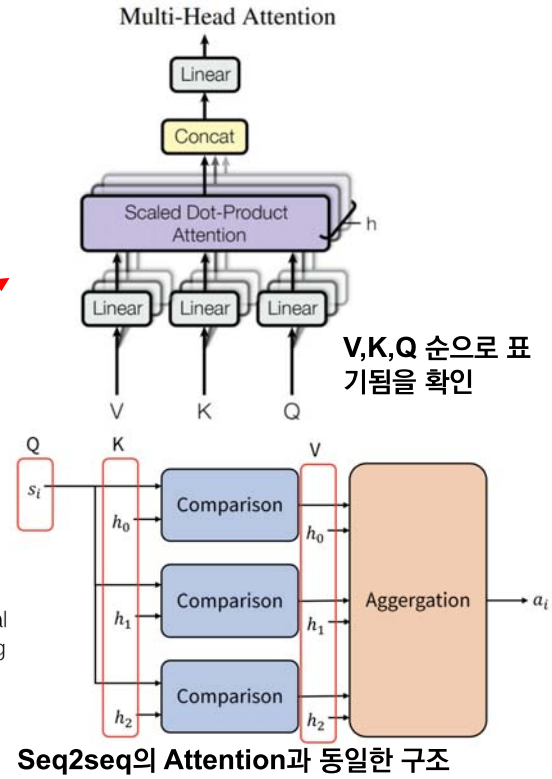
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Multi-Head Attention in Action

- Multi-Head Attention이 Transformer에 어떻게 적용되어 있는지 살펴보자.



Self-Attention



V,K,Q 순으로 표기됨을 확인

Seq2seq의 Attention과 동일한 구조

- Self-Attention은 Decoder와 동일한 Decoder를 참조하므로 Key와 Query와 Value는 모두 같은 것이다. Encoding 경우 Causual system일 필요가 없으므로 Mask 없이 Key, Value, Query가 그대로 사용될 수 있지만, Decoder 경우 현재 Query하려고 하는 것이 Key와 value가 Query보다 더 앞서서 나올 수 없기 때문에 Mask를 활용한 Masked Multi-Head Attention을 사용한다
- Encoder단의 Self-Attention을 통해서 Attention이 강조되어 있는 Feature들을 추출하고,
- Decoder단에서는 Output Embedding(or 이전의 출력값)이 들어왔을 때 이것을 Masked Multi-Head Attention을 통해 Feature 추출하고, 붉은 색 박스 부분에 이 Decoder를 통해 추출된 Feature가 Query로 들어가고, 나머지 Key, Value는 Encoder를 통해 만들어진 출력을 가지고 입력을 받게 된다. 결국에는 이런 구조는 Seq2seq 모델의 Attention과 동일한 구조가 되게 될 것이다.

Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways: 1) 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 (cite).

2) 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.

3) 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.

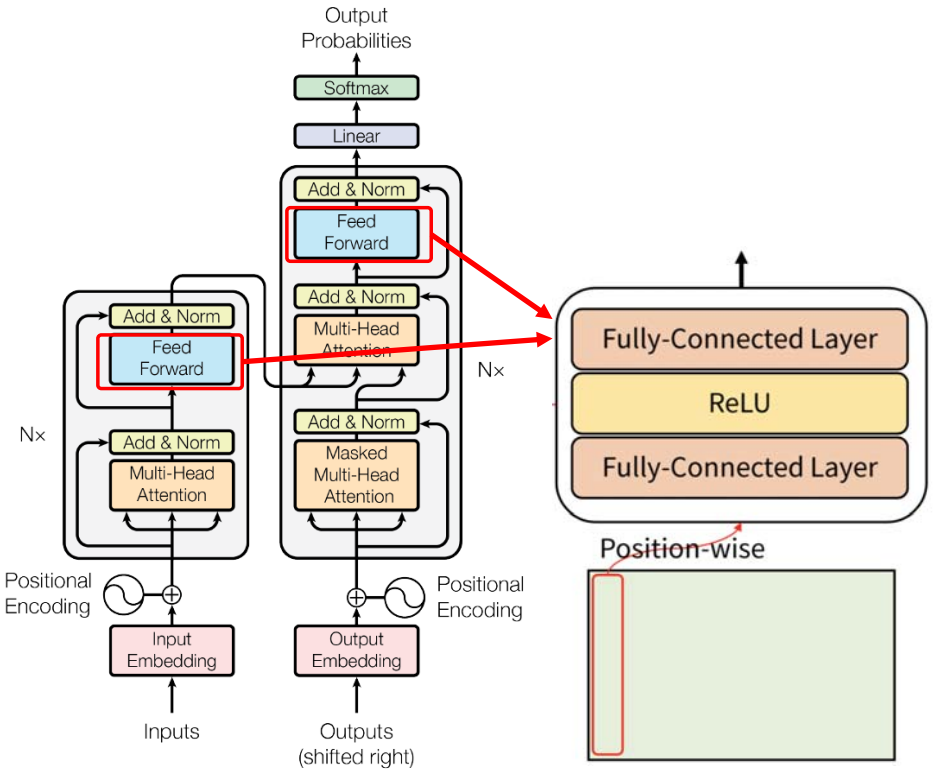
Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Position-Wise Feed-Forward

- Each layer in encoder and decoder contains a fully connected feed-forward network; applied to each position separately and identically.
- Consists of two linear transformation with a ReLU activation

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



- While linear transformations are the same across different positions, they use different parameters from layer to layer.
- Dimensionality of input and output $d_{\text{model}} = 512$
- Inner-layer has dimensionality $d_{\text{ff}} = 2048$

- Position-wise Feed-Forward는 아래 초록색 박스 처럼 가로가 문장의 길이, 세로가 One-hot vector를 크기로 갖는 행렬인데 병렬 처리되는 input 단어 하나마다 동일한 구조의 ReLU activation의 FC Layer층을 공유해서 사용하여 출력한다. 이를 통해 병렬적으로 계산하지만 기존의 FeedForward propagation을 구현할 수 있다.

Transformer : Embeddings and Softmax

- **Learned embeddings** : Used to convert the input/output tokens to vectors of dimension d_{model}.
- **Learned linear transformation and softmax function** : Used to convert the decoder output to predicted next-token probabilities.
- Share weight matrix between two embedding layers and pre-softmax linear transformation.
- In embedding layers, multiply those weights by $\sqrt{d_{\text{model}}}$

Transformer : Attention is all you need

[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

Transformer : Add & Norm

- Feed-Forward가 일어난 다음이나 Self-Attention이 일어난 다음에는 이전의 것(Skip connection)을 가져와서 더 해준 뒤 Layer Normalization을 수행해서 사용하고 있다.
- Layer Normalization은 Batch의 영향을 받지 않는 Normalization이라고 생각하면 됨.

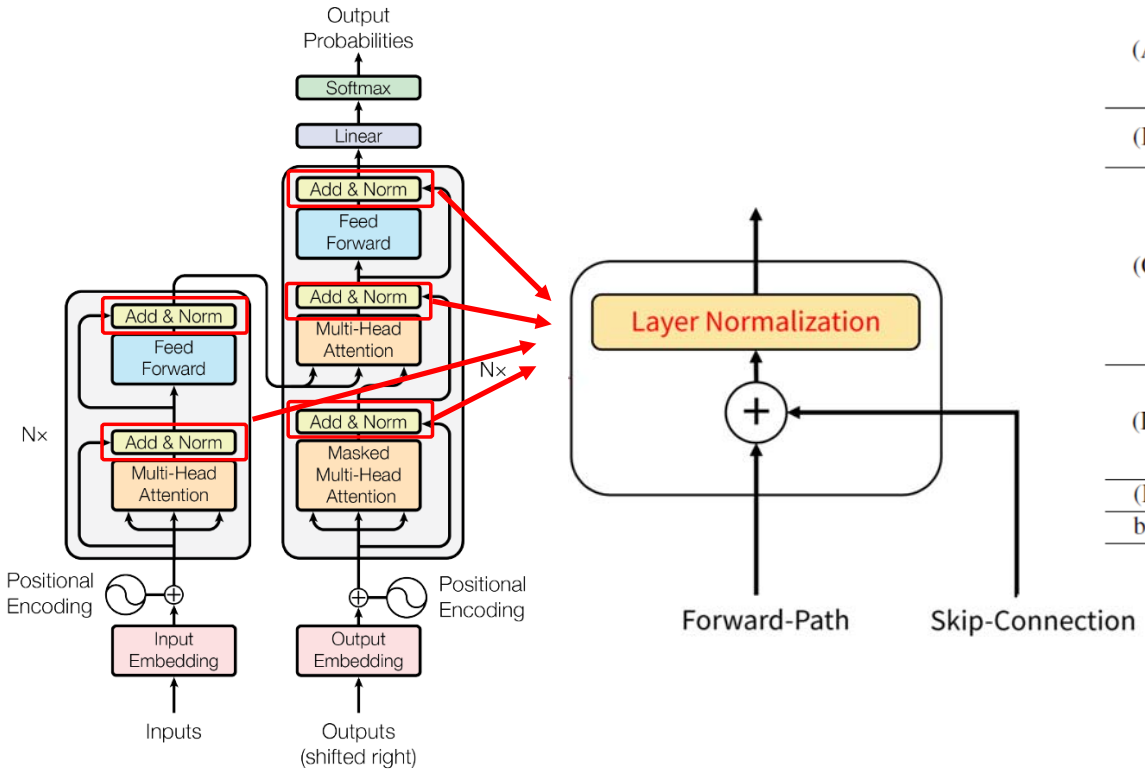


Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to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

출처

Transformer Deep Dive, Diving into the breakthroughs, scientific basis, formulas and code for the transformer architecture.

Carlo Lepelaars

https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

https://wandb.ai/authors/One-Shot-3D-Photography/reports/-Transformer---Vmlldzo0MDIyNDc

코드 예시는 Harvard NLP Group의 The Annotated Transformer 및 트랜스포머에 대한 PyTorch 문서에서 refactor 되었음.

Annotated Transformer : Training 코드 추천

https://nlp.seas.harvard.edu/2018/04/03/attention.html#training

https://github.com/huggingface/transformers/blob/master/notebooks/02-transformers.ipynb

Suggested Papers

Attention Is All You Need (2017)

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018)

Reformer: The Efficient Transformer (2020)

Linformer: Self-Attention with Linear Complexity (2020)

Longformer: The Long-Document Transformer (2020)

Language Models are Few-Shot Learners (GPT-3 paper)(2020)

Online Resources

The Annotated transformer with PyTorch Code

The Illustrated Transformer

The Narrated Transformer Language Model

Attentional Neural Network Models | Łukasz Kaiser | Masterclass

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Tokenization

- 우선, 연산을 수행하기 위해 텍스트를 숫자로 나타낼 방법이 필요하다.
- **Tokenization**는 텍스트 문자열을 압축된 기호의 시퀀스로 구문 분석(parse)하는 프로세스입니다. 이 프로세스에서 각 정수가 텍스트의 일부를 나타내는 정수의 벡터가 생성된다.

✓ Transformer 논문에서는 tokenization 방법으로 **Byte-Pair Encoding (BPE)**를 사용한다. BPE is a form of compression where the most common consecutive bytes (i.e. characters) are compressed into a single byte (i.e. integer).

- 최근 연구에 따르면 **BPE는 Suboptimal**이며 BERT와 같은 최근의 언어 모델은 **WordPiece tokenizer**를 대신 사용한다. **WordPiece는 흔히 전체 단어를 하나의 토큰으로 토큰화**하므로 디코딩이 더 쉽고 직관적으로 보인다. 대조적으로 BPE는 종종 단어의 조각을 토큰화한다. 이는 잘못된 토큰화 기호 및 개별 글자 (즉, 알파벳 문자)가 알 수 없는 토큰으로 토큰화되는 상황으로 이어질 수 있다. 그러므로, 바로 예시에서 사용하는 tokenizer는 WordPiece이다.
- Tokenizer를 텍스트의 Corpus에서 훈련시킬 수 있으나, 현실에서 실무자들은 거의 대부분 **pre-trained tokenizer**를 사용한다. HuggingFace transformers library를 사용하면 사전 훈련된 토크나이저로 쉽게 작업할 수 있다.

```
>>> from transformers import BertTokenizer
>>> tok = BertTokenizer.from_pretrained("bert-base-uncased")
Downloading: 100%|████████████████████████████| 1.04M/1.04M [00:00<00:00, 1.55MB/s]
Downloading: 100%|████████████████████████████| 456k/456k [00:00<00:00, 848kB/s]
>>> tok("Hello, how are you doing?")['inputs_ids']
{'input_ids': [101, 7592, 2129, 2024, 2017, 2725, 1029, 102]}
>>> tok("The Frenchman spoke in the [MASK] language and ate 📎")['input_ids']
{'input_ids': [101, 1996, 26529, 3764, 1999, 1996, 103, 2653, 1998, 8823, 100, 102]}
>>> tok("[CLS] [SEP] [MASK] [UNK]")['input_ids']
{'input_ids': [101, 101, 102, 103, 100, 102]}
```

- Note that the tokenizer automatically includes **a token for the start of an encoding ([CLS] == 101)** and **the end of an encoding (i.e. SEParation) ([SEP] == 102)**. Other special tokens include **masking ([MASK] == 103)** and **an unknown symbol ([UNK] = 100**, e.g. for the 📎 emoji).

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Embeddings

- 텍스트의 적절한 표현을 학습하기 위해, **시퀀스의 각 개별 token은 Embedding을 통해 벡터로 변환**되며, 이는 neural network layer의 한 종류로 볼 수 있다. **Embedding에 대한 Weight들은 Transformer 모델의 나머지 부분과 함께 학습**되기 때문이다. 이는 어휘(vocabulary)의 각 단어 대한 벡터를 포함하고 있으며, 이러한 Weight는 정규 분포 N(0,1)에서 초기화된다.
- 모델 ($E \in \mathbb{R}^{|\text{vocab}| \times d_{\text{model}}}$)을 초기화할 때 어휘(vocab) 의 크기 (|vocab|) 및 모델 (d_{model} =512)의 차원(dimension)을 지정해야 한다.
- 마지막으로 정규화(normalization) 단계로 Weight들은 $\sqrt{d_{\text{model}}}$로 곱해진다.

```
import torch
from torch import nn
class Embed(nn.Module):
    def __init__(self, vocab: int, d_model: int = 512):
        super(Embed, self).__init__()
        self.d_model = d_model
        self.vocab = vocab
        self.emb = nn.Embedding(self.vocab, self.d_model)
        self.scaling = math.sqrt(self.d_model)

    def forward(self, x):
        return self.emb(x) * self.scaling
```

CLASS `torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None, device=None, dtype=None)` [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Variables

~**Embedding.weight** (*Tensor*) – the learnable weights of the module of shape (num_embeddings, embedding_dim) initialized from $\mathcal{N}(0, 1)$

Shape:

- Input: $(*)$, IntTensor or LongTensor of arbitrary shape containing the indices to extract
- Output: $(*, H)$, where $*$ is the input shape and $H =$ embedding_dim

nn.Embedding

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

When `max_norm` is not `None`, `Embedding`'s forward method will modify the `weight` tensor in-place. Since tensors needed for gradient computations cannot be modified in-place, performing a differentiable operation on `Embedding.weight` before calling `Embedding`'s forward method requires cloning `Embedding.weight` when `max_norm` is not `None`. For example:

```
n, d, m = 3, 5, 7
embedding = nn.Embedding(n, d, max_norm=True)
W = torch.randn((m, d), requires_grad=True)
idx = torch.tensor([1, 2])
a = embedding.weight.clone() @ W.t()  # weight must be cloned for this to be differentiable
b = embedding(idx) @ W.t()  # modifies weight in-place
out = (a.unsqueeze(0) + b.unsqueeze(1))
loss = out.sigmoid().prod()
loss.backward()
```

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

```
>>> # example of changing `pad` vector
>>> padding_idx = 0
>>> embedding = nn.Embedding(3, 3, padding_idx=padding_idx)
>>> embedding.weight
Parameter containing:
tensor([[ 0.0000,  0.0000,  0.0000],
        [-0.7895, -0.7089, -0.0364],
        [ 0.6778,  0.5803,  0.2678]], requires_grad=True)
>>> with torch.no_grad():
...     embedding.weight[padding_idx] = torch.ones(3)
>>> embedding.weight
Parameter containing:
tensor([[ 1.0000,  1.0000,  1.0000],
        [-0.7895, -0.7089, -0.0364],
        [ 0.6778,  0.5803,  0.2678]], requires_grad=True)
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Positional Encoding

- Recurrent 및 Convolutional networks과는 대조적으로, 모델 자체는 시퀀스에 embedded tokens의 상대 위치(relative position)에 대한 정보를 가지고 있지 않다. 따라서 Encoder와 Decoder에 대한 Input Embeddings에 인코딩을 추가함으로써 이 위치 정보를 입력해야 한다. 이 정보는 다양한 방법으로 추가할 수 있으며 정적이거나 학습될 수 있다.
- Transformer는 각 위치 (pos)에 대한 sin 및 cos 변환을 사용한다. sin은 짝수 차원 $(2i)$ 에서 사용되며, cos은 홀수 차원 $(2i+1)$에 사용된다.

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

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

- Positional encodings are computed in *log space* to avoid *numerical overflow*.

`torch.arange(start=0, end, step=1, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) → Tensor`

Returns a 1-D tensor of size $\left\lceil \frac{\text{end}-\text{start}}{\text{step}} \right\rceil$ with values from the interval `[start, end)` taken with common difference `step` beginning from *start*.

Note that non-integer `step` is subject to floating point rounding errors when comparing against `end`; to avoid inconsistency, we advise adding a small epsilon to `end` in such cases.

$$\text{out}_{i+1} = \text{out}_i + \text{step}$$

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

```
import torch
from torch import nn
from torch.autograd import Variable

class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int = 512, dropout: float = .1, max_len: int = 5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)

        # Compute the positional encodings in log space
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * -
(torch.log(torch.Tensor([10000.0])) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)], requires_grad=False)
        return self.dropout(x)
```

Transformer 심층분석 PyTorch

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

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 (cite).

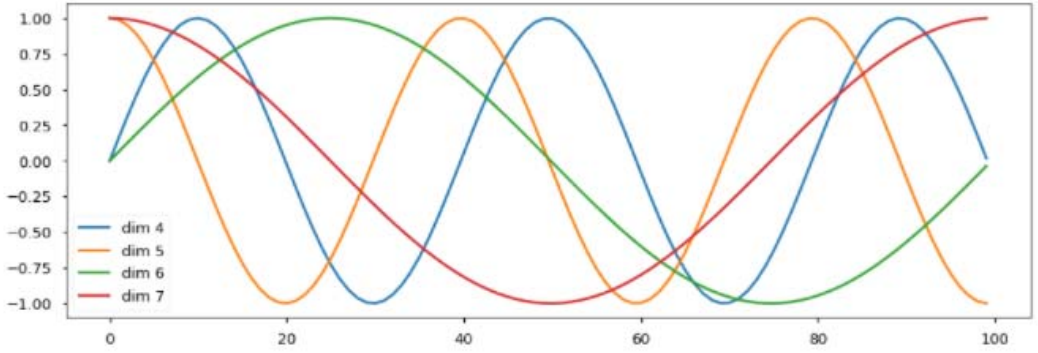
In this work, we use sine and cosine functions of different frequencies:
$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$

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

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 $P_{drop} = 0.1$.

> *Below the positional encoding will add in a sine wave based on position. The frequency and offset of the wave is different for each dimension.*

```
plt.figure(figsize=(15, 5))
pe = PositionalEncoding(20, 0)
y = pe.forward(Variable(torch.zeros(1, 100, 20)))
plt.plot(np.arange(100), y[0, :, 4:8].data.numpy())
plt.legend(["dim %d"%p for p in [4,5,6,7]])
None
```



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

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Multi-Head Attention

- **Attention layer는 Query(Q) 와 Key(K), Value 값 (V)쌍 간의 맵핑을 학습**할 수 있습니다. 이러한 이름의 의미는 특정 NLP 응용 프로그램에 따라 달라지므로 헷갈릴 수 있다.
- 텍스트 생성의 맥락에서, query는 입력의 Embedding이며, value와 key는 Targets로 볼 수 있다. 일반적으로 값과 키는 동일하다.
- "**Scaled dot-product attention**"라 부르는 것은 NLP에서 Attention의 Performance를 높인 하나의 획기적인 것다. 이는 *multiplicative attention*과 동일하나, 추가적으로 Q 와 K 맵핑은 키 차원 d_k 에 의해 크기가 조정(scale)된다. 이를 통해서 Multiplicative attention은 더 큰 차원에서 더 나은 performance를 보여준다. 결과는 softmax activation $softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$ 에 V를 곱한다.

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

```
import torch
from torch import nn
class Attention:
    def __init__(self, dropout: float = 0.):
        super(Attention, self).__init__()
        self.dropout = nn.Dropout(dropout)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, query, key, value, mask=None):
        d_k = query.size(-1)
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        p_attn = self.dropout(self.softmax(scores))
        return torch.matmul(p_attn, value)

    def __call__(self, query, key, value, mask=None):
        return self.forward(query, key, value, mask)
```

`Tensor.masked_fill_(mask, value)`

Fills elements of `self` tensor with `value` where `mask` is True. The shape of `mask` must be broadcastable with the shape of the underlying tensor.

Transformer 심층분석 PyTorch

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Multi-Head Attention

- Decoder에서 attention sub-layer는 특정 위치를 매우 큰 음수 (−1e9 or sometimes even −inf)로 채움으로써 masking 된다. 이는 후속 위치를 처리함으로써 모델이 cheating 하는 것을 예방하기 위한 것다. 이를 통해 모델은 다음 토큰을 예측하려 할 때 이전 위치의 단어에만 주의를 기울일 수 있다.
- attention mechanism 그 자체는 이미 아주 효과적이며 matrix multiplication에 최적화된 GPU 및 TPU과 같은 최신 하드웨어에서 효율적으로 연산될 수 있다. 하지만 a single attention layer는 하나의 표현만을 허용한다. 따라서 Transformer에서 **multiple attention heads**가 사용된다. 이를 통해 모델은 **multiple patterns 및 representations**을 학습할 수 있다.
- The paper uses h = 8 attention layers which are concatenated. The final formula becomes:

$$\text{MultiHead}(\text{Q}, \text{K}, \text{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})W^O$$
$$\text{where } \text{head}_1 = \text{Attention}(Q\text{W}_i^Q K\text{W}_i^K, V\text{W}_i^V)$$

The projection weights ($W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}, W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$) are outputs of a fully-connected (Linear) layer. The authors of the transformer paper use $d_k = d_v = \frac{d_{\text{model}}}{h} = 64$.

```
from torch import nn
from copy import deepcopy
class MultiHeadAttention(nn.Module):
    def __init__(self, h: int = 8, d_model: int = 512, dropout: float = 0.1):
        super(MultiHeadAttention, self).__init__()
        self.d_k = d_model // h
        self.h = h
        self.attn = Attention(dropout)
        self.lindim = (d_model, d_model)
        self.linears = nn.ModuleList([deepcopy(nn.Linear(*self.lindim)) for _ in range(4)])
        self.final_linear = nn.Linear(*self.lindim, bias=False)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)

        nbatches = query.size(0)
        # Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2) \
                             for l, x in zip(self.linears, (query, key, value))]
        x = self.attn(query, key, value, mask=mask)

        # Concatenate and multiply by W^O
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
        return self.final_linear(x)
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Multi-Head Attention

- Technical note : **.transpose**는 기본 메모리 저장소를 원래의 tensor와 공유하기 때문에 **.contiguous** 방법은 **.transpose 다음에 추가된**다. 이후 .view를 호출하려면 인접한(contiguous) tensor가 필요하다 (문서). .view 방법은 효율적인 변환(reshaping), 슬라이싱(slicing) 및 요소별 작업을 수행할 수 있다. (문서)
- 각 head의 차원을 h,로 나누기 때문에, 총 연산은 full dimensionality를 가진 하나의 attention head를 사용하는 것과 유사하다. 그러나, 이 접근 방식을 통한 연산은 헤드를 따라 병렬화(parallelize) 될 수 있으며 이를 통해 최신 하드웨어에서 속도가 크게 향상된다. 이는 합성곱 또는 순환(recurrence) 없이 효과적인 언어 모델을 훈련할 수 있게끔 하는 혁신적인 부분 중 하나이다

```
query, key, value = [l(x).view(query.size(0), -1, self.h, self.d_k).transpose(1, 2) \
                     for l, x in zip(self.linears, (query, key, value))]
nbatches = query.size(0)
x = self.attn(query, key, value, mask=mask)

# Concatenate and multiply by W^O
x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
return self.final_linear(x)
```

Tensor view

```
>>> t = torch.rand(4, 4)
>>> b = t.view(2, 8)
>>> t.storage().data_ptr() == b.storage().data_ptr()  # `t` and `b`
share the same underlying data.
True
# Modifying view tensor changes base tensor as well.
>>> b[0][0] = 3.14
>>> t[0][0]
tensor(3.14)
```

```
>>> base = torch.tensor([[0, 1],[2, 3]])
>>> base.is_contiguous()
True
>>> t = base.transpose(0, 1)  # `t` is a view of `base`. No data
movement happened here.
# View tensors might be non-contiguous.
>>> t.is_contiguous()
False
# To get a contiguous tensor, call `.contiguous()` to enforce
# copying data when `t` is not contiguous.
>>> c = t.contiguous()
```

contiguous

view, transpose, permute **등과 같은 원본** Tensor**의 메타데이터가 변경하는 함수들의 결과값은** non contiguous Tensor**임**

non contiguous Tensor**는 주소값 재배열 연산이 필요할 때 사용할 수 없음**

contiguous **함수로 새로운 메모리에 할당하여** contiguous Tensor**로 변경하면 주소값 재배열이 가능**

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Residual & Layer Normalization

- AI 연구 커뮤니티에서는 residual connections 및 (Batch) normalization와 같은 개념이 performance를 향상시키고, 훈련 시간을 단축하며, 보다 심층적인 네트워크의 훈련을 가능케 한다는 사실을 발견함. 따라서, 모든 attention layer 및 모든 feed forward layer 다음에 Transformer는 residual connection 및 normalization를 갖추고 있다. 추가적으로, 더 나은 일반화(generalization)를 위해 각 레이어에 dropout이 추가된다.

Layer Normalization

- 현대 딥러닝 기반 컴퓨터 비전 모델은 보통 배치 정규화를 포함하고 있다. 그러나 이러한 정규화 유형은 큰 배치 사이즈에 의해 좌우되며, 당연하게도 recurrence에 적합하지 않다. 기존 트랜스포머 아키텍처는 레이어 정규화를 대신 갖추고 있다. 레이어 정규화는 배치 크기가 작더라도 (batchsize<8) 안정적임.

In order to calculate layer normalization, we first calculate the mean μ_i and standard deviation σ_i separately for each sample in the minibatch.

$$\mu_i = \frac{1}{K}\sum_{k=1}^{k} x_{i,k}$$

$$\sigma_i = \text{sqrt}\left(\frac{1}{K}\sum_{k=1}^{k}(x_{i,k} - \mu_i)^2\right)$$

Then, the normalization step is defined as:

$$LN_{\gamma,\beta}(x_i) \equiv \gamma\frac{x-\mu_i}{\sigma_i+\epsilon} + \beta$$

where γ and β are learnable parameters. A small number ϵ is added for numerical stability in case the standard deviation σ_i is 0.

```
from torch import nn
class LayerNorm(nn.Module):
    def __init__(self, features: int, eps: float = 1e-6):
        super(LayerNorm, self).__init__()
        self.gamma = nn.Parameter(torch.ones(features))
        self.beta = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Residual & Layer Normalization

Residual

- A residual connection means that you add the output of a previous layer in the network (i.e sublayer) to the output of the current layer. This allows for very deep networks because the network can essentially 'skip' certain layers.
- The final output of each layer will then be

$$\text{ResidualConnection}(x)=x+\text{Dropout}(\text{SubLayer}(\text{LayerNorm}(x)))$$

```
from torch import nn
class ResidualConnection(nn.Module):
    def __init__(self, size: int = 512, dropout: float = .1):
        super(ResidualConnection,  self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

Position-Wise Feed Forward

On top of every attention layer a feed forward network is added. This consists of two fully-connected layers with a ReLU activation ($\text{ReLU}(x) = \max(0, x)$) and dropout for the inner layer. The standard dimensions used in the transformer paper are $d_{\text{model}} = 512$ for the input layer and $d_{ff} = 2048$ for the inner layer.

The full calculation becomes $\text{FeedForward}(x) = W_2 max(0, xW_1 + B_1) + B_2$.

Note that PyTorch Linear already includes the biases (B_1 and B_2) by default.

```
from torch import nn
class FeedForward(nn.Module):
    def __init__(self, d_model: int = 512, d_ff: int = 2048, dropout: float = .1):
        super(FeedForward, self).__init__()
        self.l1 = nn.Linear(d_model, d_ff)
        self.l2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.l2(self.dropout(self.relu(self.l1(x))))
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Encoder - Decoder

Encoder

- Now we have all the components to build the model encoder and decoder. A **single encoder layer** consists of **a multi-head attention layer** followed by **a feed-forward network**. As mentioned earlier, we also include *residual connections* and *layer normalization*.

$$\text{Encoding}(x,\text{mask}) = \text{FeedForward}(\text{MultiHeadAttention}(x))$$

```
from torch import nn
from copy import deepcopy
class EncoderLayer(nn.Module):
    def __init__(self, size: int, self_attn: MultiHeadAttention, feed_forward: FeedForward, dropout: float = .1):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sub1 = ResidualConnection(size, dropout)
        self.sub2 = ResidualConnection(size, dropout)
        self.size = size

    def forward(self, x, mask):
        x = self.sub1(x, lambda x: self.self_attn(x, x, x, mask))
        return self.sub2(x, self.feed_forward)
```

- The final transformer encoder from the paper consists of 6 identical encoder layers followed by layer normalization.

```
class Encoder(nn.Module):
    def __init__(self, layer, n: int = 6):
        super(Encoder, self).__init__()
        self.layers = nn.ModuleList([deepcopy(layer) for _ in range(n)])
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Encoder - Decoder

Decoder

- The decoding layer is **a masked multi-head attention layer** followed by **multi-head attention layer that includes memory**. **Memory is an output from the encoder**. Lastly, it goes through **a feed-forward network**. Again, all these components include *residual connections* and *layer normalization*.

Decoding(x,memory,mask1, mask2) =

FeedForward(MultiHeadAttention(MultiHeadAttention(x,mask1), memory,mask2))

- As with the final encoder, the decoder in the paper also has 6 identical layers followed by layer normalization.

```
class Decoder(nn.Module):
    def __init__(self, layer: DecoderLayer, n: int = 6):
        super(Decoder, self).__init__()
        self.layers = nn.ModuleList([deepcopy(layer) for _ in range(n)])
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

```
from torch import nn
from copy import deepcopy
class DecoderLayer(nn.Module):
    def __init__(self, size: int, self_attn: MultiHeadAttention, src_attn:
MultiHeadAttention, feed_forward: FeedForward, dropout: float = .1):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sub1 = ResidualConnection(size, dropout)
        self.sub2 = ResidualConnection(size, dropout)
        self.sub3 = ResidualConnection(size, dropout)

    def forward(self, x, memory, src_mask, tgt_mask):
        x = self.sub1(x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sub2(x, lambda x: self.src_attn(x, memory, memory, src_mask))
        return self.sub3(x, self.feed_forward)
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Encoder - Decoder

Encoder-Decoder

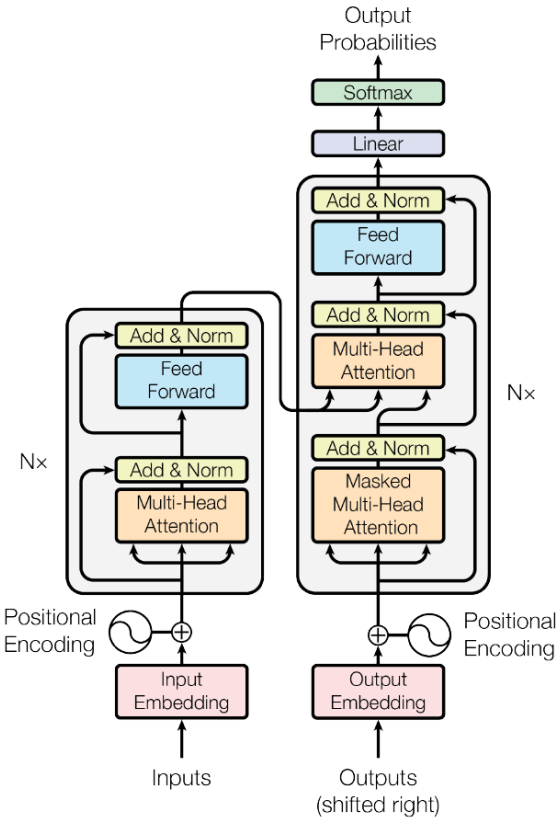
- With this higher level representation of the encoder and decoder we can easily formulate the final encoder-decoder block.

```
from torch import nn
class EncoderDecoder(nn.Module):
    def __init__(self, encoder: Encoder, decoder: Decoder,
                 src_embed: Embed, tgt_embed: Embed, final_layer: Output):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.final_layer = final_layer

    def forward(self, src, tgt, src_mask, tgt_mask):
        return self.final_layer(self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask))

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```



[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Final Output

- 마지막으로, Decoder의 벡터 출력은 최종 출력으로 변환되어야 합니다. 언어 번역과 같은 sequence-to-sequence 문제의 경우 이는 각 position에 대한 총 어휘에 관한 확률 분포이다. Only a fully-connected layer는 decoder outpup을 logits의 행렬로 변환되며, 이는 타깃 어휘의 차원을 갖고 있습니다. 이러한 숫자는 softmax 활성화 함수를 통해 어휘에 대한 확률 분포로 변환된다.

$$\text{LogSoftmax}(x_i) = \log(\frac{\exp(x_i)}{\sum_j \exp(x_j)})$$

- 예를 들어, 번역된 문장이 20개의 토큰을 갖고 있고, 총 어휘는 30000개의 토큰이라고 가정해보자. 그러면 결과 출력은 행렬 $\text{matrix M} \in \mathbb{R}^{20 \times 30000}$이 됨.
- 그런 다음 마지막 차원에 대한 argmax를 취하여 tokenize를 통해 텍스트 열로 디코딩 할 수 있는 출력 토큰 $\text{T} \in \mathbb{R}^{20}$ 의 벡터를 얻을 수 있다.

$$\text{Output(x)} = \text{LogSoftmax}(\max(0, \text{xW}_1 + \text{B}_1))$$

```
from torch import nn
class Output(nn.Module):
    def __init__(self, input_dim: int, output_dim: int):
        super(Output, self).__init__()
        self.l1 = nn.Linear(input_dim, output_dim)
        self.log_softmax = nn.LogSoftmax(dim=-1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        logits = self.l1(x)
        return self.log_softmax(logits)
```

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Model initialization

- 논문에서와 같이 동일한 차원으로 transformer model을 구축한다. Initialization strategy는 **Xavier/Glorot 초기화**이며 이는 $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ 범위의 균일 분포에서 선택하도록 구성된다. 모든 bias 들은 0로 초기화된다.

$$\mathrm{Xavier(W)} \sim \mathrm{U}[-\tfrac{1}{\mathrm{n}}, \tfrac{1}{\mathrm{n}}], \mathrm{B} = 0$$

- This function return a PyTorch model that can be trained for sequence to sequence problems.

```
from torch import nn
def make_model(input_vocab: int, output_vocab: int, d_model: int = 512):
    encoder = Encoder(EncoderLayer(d_model, MultiHeadAttention(), FeedForward()))
    decoder = Decoder(DecoderLayer(d_model, MultiHeadAttention(), MultiHeadAttention(), FeedForward()))
    input_embed= nn.Sequential(Embed(vocab=input_vocab), PositionalEncoding())
    output_embed = nn.Sequential(Embed(vocab=output_vocab), PositionalEncoding())
    output = Output(input_dim=d_model, output_dim=output_vocab)
    model = EncoderDecoder(encoder, decoder, input_embed, output_embed, output)

    # Initialize parameters with Xavier uniform
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model
```

LeCun Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{1}{n_{in}}}$$

LeCun Uniform Initialization

$$W \sim U(-\sqrt{\frac{1}{n_{in}}}, +\sqrt{\frac{1}{n_{in}}})$$

Xaiver Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

Xaiver함수는 비선형함수(ex. sigmoid, tanh)에서 효과적인 결과를 보여준다. 하지만 ReLU 함수에서 사용 시 출력 값이 0으로 수렴할 수 있으므로, 다른 초기화 방법을 사용해야 한다

Xaiver Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

(n_{in} : 이전 layer(input)의 노드 수, n_{out} : 다음 layer의 노드 수)

[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

Model initialization

- 아래에서 토큰화된 입력 및 출력과 함께 이를 사용하는 방법에 대한 더미 예시가 설명 되어 있다. 입력과 출력에 대한 어휘가 101010개만 있다고 가정해보자.

```
# Tokenized symbols for source and target.
>>> src = torch.tensor([[1, 2, 3, 4, 5]])
>>> src_mask = torch.tensor([[1, 1, 1, 1, 1]])
>>> tgt = torch.tensor([[6, 7, 8, 0, 0]])
>>> tgt_mask = torch.tensor([[1, 1, 1, 0, 0]])

# Create PyTorch model
>>> model = make_model(input_vocab=10, output_vocab=10)
# Do inference and take tokens with highest probability through argmax along the vocabulary axis (-1)
>>> result = model(src, tgt, src_mask, tgt_mask)
>>> result.argmax(dim=-1)
tensor([[6, 6, 4, 3, 6]])
```

- 모델은 균일하게 초기화된 가중치를 갖고 있으므로, 출력은 target과 거리가 상당히 크다. 이러한 transformer model을 처음부터 훈련하는 것은 상당한 계산을 필요로 한다. 기본 모델을 훈련하기 위해, 저자는 논문에서 **12시간 동안 8개의 NVIDIA P100 GPU를 훈련**시켰다. 더 큰 모델은 **8개의 GPU를 훈련하는 데 3.5일**을 소요된다! 사전 훈련된 transformer model 사용 및 응용 프로그램에 적합하도록 미세 조정하시기 바란다. HuggingFace Transformers library는 이미 미세 조정을 위한 사전 훈련된 모델을 많이 가지고 있다.
- 훈련 절차를 처음부터 코딩하는 방법에 대해서 더 자세히 알고 싶으시다면 The Annotated Transformer 훈련 섹션 확인